

# Biology 304: Biological Data Analysis

*Paul M. Magwene*

*2018-12-06*



# Contents

<b>1</b>	<b>Introduction</b>	<b>9</b>
1.1	How to use these lecture notes . . . . .	9
<b>2</b>	<b>Data story: Women and children first on the Titanic?</b>	<b>11</b>
2.1	Background . . . . .	11
2.2	Dataset . . . . .	11
2.3	Libraries . . . . .	11
2.4	Read data . . . . .	13
2.5	What's in the data? . . . . .	13
2.6	Categorizing passengers . . . . .	14
2.7	Passenger ages . . . . .	19
2.8	How does age relate to survival? . . . . .	20
2.9	How does class affect survival? . . . . .	25
2.10	Conclusion . . . . .	28
<b>3</b>	<b>Getting Started with R</b>	<b>29</b>
3.1	What is R? . . . . .	29
3.2	What is RStudio? . . . . .	29
3.3	Entering commands in the console . . . . .	29
3.4	Comments . . . . .	30
3.5	Using R as a Calculator . . . . .	30
3.6	The R Help System . . . . .	31
3.7	Variable assignment in R . . . . .	32
3.8	Data types . . . . .	33
3.9	Logical values . . . . .	33
3.10	Character strings . . . . .	34
3.11	Packages . . . . .	35
3.12	Reading data from a file . . . . .	37
3.13	Exploring the “possums” data set . . . . .	37
3.14	Simple tables . . . . .	37
3.15	Simple figures . . . . .	38
3.16	Bar plots . . . . .	38
3.17	Histograms . . . . .	40
3.18	Scatter plots . . . . .	41
<b>4</b>	<b>R Markdown and R Notebooks</b>	<b>45</b>
4.1	R Notebooks . . . . .	45
4.2	Creating an R Notebook . . . . .	45
4.3	The default R Notebook template . . . . .	46
4.4	Code and Non-code blocks . . . . .	46
4.5	Running a code chunk . . . . .	47
4.6	Running all code chunks above . . . . .	48

4.7 “Knitting an” R Markdown to HTML . . . . .	48
4.8 Sharing your reproducible R Notebook . . . . .	49
<b>5 More R Basics: Data structures</b>	<b>51</b>
5.1 Vectors . . . . .	51
5.2 Lists . . . . .	56
5.3 Data frames . . . . .	61
<b>6 Introduction to ggplot2</b>	<b>67</b>
6.1 Loading ggplot2 . . . . .	67
6.2 Example data set: Anderson’s Iris Data . . . . .	68
6.3 Template for single layer plots in ggplot2 . . . . .	68
6.4 An aside about function arguments . . . . .	69
6.5 Strip plots . . . . .	70
6.6 Histograms . . . . .	75
6.7 Faceting to depict categorical information . . . . .	79
6.8 Density plots . . . . .	80
6.9 Violin or Beanplot . . . . .	81
6.10 Boxplots . . . . .	82
6.11 Building complex visualizations with layers . . . . .	83
6.12 Useful combination plots . . . . .	83
6.13 ggplot layers can be assigned to variables . . . . .	85
6.14 Adding titles and tweaking axis labels . . . . .	88
6.15 ggplot2 themes . . . . .	89
6.16 Other aspects of ggplots can be assigned to variables . . . . .	92
6.17 Bivariate plots . . . . .	93
6.18 Bivariate density plots . . . . .	97
6.19 Combining Scatter Plots and Density Plots with Categorical Information . . . . .	99
6.20 Density plots with fill . . . . .	102
6.21 2D bin and hex plots . . . . .	105
6.22 The cowplot package . . . . .	107
<b>7 Introduction to the dplyr package</b>	<b>111</b>
7.1 Libraries . . . . .	111
7.2 Reading data with the readr package . . . . .	111
7.3 A note on “tibbles” . . . . .	112
7.4 Data filtering and transformation with dplyr . . . . .	113
7.5 dplyr’s “verbs” . . . . .	113
7.6 Pipes . . . . .	121
<b>8 Data wrangling, Part I</b>	<b>127</b>
8.1 Libraries . . . . .	127
8.2 Data . . . . .	127
8.3 Renaming data frame columns . . . . .	128
8.4 Dropping unneeded columns . . . . .	129
8.5 Merging data frames . . . . .	131
8.6 Reshaping data with tidyr . . . . .	132
8.7 Using your tidy data . . . . .	136
<b>9 Data wrangling, Part II</b>	<b>143</b>
9.1 Libraries . . . . .	143
9.2 Data . . . . .	143
9.3 Heat maps . . . . .	144
9.4 Long-to-wide conversion using tidyr::spread . . . . .	150
9.5 Exploring bivariate relationships using “wide” data . . . . .	152

<b>10 Functions and control flow statements</b>	<b>161</b>
10.1 Writing your own functions . . . . .	161
10.2 Control flow statements . . . . .	164
10.3 <code>map</code> and related tools . . . . .	169
<b>11 Frequency Distributions and Descriptive Statistics</b>	<b>173</b>
<b>12 Joint Frequency Distributions and Measures of Association</b>	<b>175</b>
<b>13 Introduction to Probability</b>	<b>177</b>
13.1 Terms . . . . .	177
13.2 Frequentist definition of probability . . . . .	178
13.3 Probability distribution . . . . .	179
13.4 Mutually exclusive events . . . . .	180
13.5 Independence . . . . .	181
13.6 General addition rule . . . . .	181
13.7 Conditional probability . . . . .	181
13.8 General multiplication rule . . . . .	182
13.9 Probability trees . . . . .	182
13.10 Law of total probability . . . . .	182
<b>14 Introduction to Sampling Distributions, Part I</b>	<b>185</b>
14.1 Libraries . . . . .	185
14.2 In class experiments . . . . .	185
14.3 Simulating sampling in R . . . . .	186
14.4 Distribution of estimates of the proportion . . . . .	189
<b>15 Introduction to Sampling Distributions, Part II</b>	<b>191</b>
15.1 Libraries . . . . .	191
15.2 Data set: Simulated male heights . . . . .	192
15.3 Seeding the pseudo-random number generator . . . . .	195
15.4 Random sampling from the simulated population . . . . .	195
15.5 Simulated sampling distribution of the mean . . . . .	199
15.6 Standard Error of the Mean . . . . .	203
15.7 Sampling Distribution of the Standard Deviation . . . . .	204
15.8 What happens to the sampling distribution of the mean and standard deviation when our sample size is small? . . . . .	206
<b>16 Introduction to hypothesis testing</b>	<b>211</b>
16.1 Libraries . . . . .	211
16.2 Null and Alternative Hypotheses . . . . .	211
16.3 Rejecting / failing to reject null hypotheses . . . . .	212
16.4 Outcomes of hypothesis tests . . . . .	212
16.5 Using p-values to assess the strength of evidence against the null hypothesis . . . . .	213
16.6 Example: Comparing a sample mean to an hypothesized normal distribution . . . . .	213
16.7 Example: Handedness of toads . . . . .	214
16.8 Example: Measuring the association between maternal smoking and premature births . . . . .	216
<b>17 Introduction to confidence intervals</b>	<b>221</b>
17.1 Confidence Intervals . . . . .	221
17.2 Generic formulation for confidence intervals . . . . .	223
17.3 Example: Confidence intervals for the mean . . . . .	223
17.4 A problem arises! . . . . .	226
<b>18 Normal distributions</b>	<b>227</b>

18.1 Basics about normal distributions . . . . .	227
18.2 Normal distribution, probability density function . . . . .	227
18.3 Approximately normal distributions are very common . . . . .	229
18.4 Central limit theorem . . . . .	229
18.5 Visualizing normal distributions . . . . .	230
18.6 Comparing values from different normal distributions . . . . .	231
18.7 Standard normal distribution . . . . .	232
18.8 88-95-99.7 Rule . . . . .	232
18.9 Percentiles . . . . .	233
18.10 Cutoff points . . . . .	234
18.11 Assessing normality . . . . .	234
<b>19 Comparing sample means</b>	<b>239</b>
19.1 Hypothesis test for the mean using the t-distribution . . . . .	239
19.2 One sample t-test . . . . .	239
19.3 The <code>t.test</code> function in R . . . . .	242
19.4 Two sample t-test . . . . .	243
19.5 Paired t-test . . . . .	246
19.6 The fallacy of indirect comparison . . . . .	248
19.7 Summary table for different t-tests . . . . .	248
<b>20 Analysis of Variance</b>	<b>251</b>
20.1 Hypotheses for ANOVA . . . . .	251
20.2 ANOVA, assumptions . . . . .	251
20.3 ANOVA, key idea . . . . .	251
20.4 Partitioning of sum of squares . . . . .	251
20.5 Mathematical partitioning of sums of squares . . . . .	252
20.6 ANOVA test statistic and sampling distribution . . . . .	253
20.7 ANOVA tables . . . . .	253
20.8 The <code>aov()</code> function . . . . .	254
20.9 Example, circadian rhythm data . . . . .	254
20.10 ANOVA calculations: Step-by-step . . . . .	257
20.11 Visualizing the partitioning of sum-of-squares . . . . .	259
20.12 Which pairs of group means are different? . . . . .	262
20.13 Repeatability . . . . .	263
<b>21 Violations of Assumptions</b>	<b>267</b>
21.1 Graphical methods for detecting deviations from normality . . . . .	267
21.2 Formal test of normality . . . . .	269
21.3 When to ignore violations of assumptions . . . . .	270
21.4 Data transformations . . . . .	270
21.5 Nonparametric tests . . . . .	272
Acknowledgements . . . . .	275
<b>22 Bivariate Linear Regression</b>	<b>277</b>
22.1 Regression terminology . . . . .	277
22.2 The optimality criterion for least-squares regression . . . . .	277
22.3 Solution for the least-squares criterion . . . . .	278
22.4 Libraries . . . . .	278
22.5 Illustrating linear regression with simulated data . . . . .	278
22.6 Specifying Regression Models in R . . . . .	279
22.7 Residuals . . . . .	283
22.8 Regression as sum-of-squares decomposition . . . . .	284
22.9 Variance “explained” by a regression model . . . . .	284
22.10 Broom: a library for converting model results into data frames . . . . .	285

22.11 Example: Predicting lion age based on nose color . . . . .	287
<b>23 Multiple regression</b>	<b>291</b>
23.1 Libraries to install . . . . .	291
23.2 Review of bivariate regression . . . . .	291
23.3 Multiple regression . . . . .	291
23.4 Interpreting Multiple Regression . . . . .	293
23.5 Libraries . . . . .	293
23.6 Example data set: <code>mtcars</code> . . . . .	293
23.7 Visualizing and summarizing the variables of interest . . . . .	293
23.8 3D plots . . . . .	295
23.9 Fitting a multiple regression model in R . . . . .	298
23.10 Visualizing the regression plane . . . . .	299
23.11 Interactive 3D Visualizations Using OpenGL . . . . .	300
23.12 Examining the residuals . . . . .	301
<b>24 Logistic regression</b>	<b>303</b>
24.1 A web app to explore the logistic regression equation . . . . .	303
24.2 Titanic data set . . . . .	304
24.3 Subsetting the data . . . . .	304
24.4 Visualizing survival as a function of age . . . . .	304
24.5 Fitting the logistic regression model . . . . .	305
24.6 Visualizing the logistic regression . . . . .	305
24.7 Impact of sex and passenger class on the models . . . . .	307
24.8 Fitting multiple models based on groupings use <code>dplyr::do</code> . . . . .	308



# Chapter 1

## Introduction

### 1.1 How to use these lecture notes

In this and future materials to be posted on the course website you'll encounter blocks of R code. *Your natural intuition will be to cut and paste commands or code blocks into the R interpreter to save yourself the typing. DO NOT DO THIS!!*

In each of the examples below, I provide example input, but I don't show you the output. It's your job to type in these examples at the R console, evaluate what you typed, and to look at and think critically about the output. **You will make mistakes and generate errors!** Part of learning any new skill is making mistakes, figuring out where you went wrong, and correcting those mistakes. In the process of fixing those errors, you'll learn more about how R works, and how to avoid such errors, or correct bugs in your own code in the future. If you cut and paste the examples into the R interpreter the code will run, but you will learn less than if you input the code yourself and you'll be less capable of applying the concepts in new situations.

The R interpreter, like all programming languages, is very exacting. A misspelled variable or function name, a stray period, or an unbalanced parenthesis will raise an error and finding the sources of such errors can sometimes be tedious and frustrating. Persist! If you read your code critically, think about what you're doing, and seek help when needed (teaching team, R help, Google, etc) you'll eventually get better at debugging your code. But keep in mind that like most new skills, learning to write and debug your code efficiently takes time and experience.



# Chapter 2

## Data story: Women and children first on the Titanic?

This introductory chapter illustrates some of the tools and concepts you'll learn in this class, such as visualization, data restructuring, and model building. By the end of this course, you should be able to carry out similar analyses and make well reasoned interpretation of those analyses for a variety of complex biological data.

### 2.1 Background

On April 10, 1912 the RMS Titanic left Southhampton, England headed for New York. Aboard were 2,435 passengers and 892 crew members. Five days later, about 20 minutes before midnight, the Titanic hit an iceberg in the frigid waters about 375 miles south of New Foundland. Within approximately two and a half hours the ship had split apart and sunk, leaving just over 700 survivors.

### 2.2 Dataset

The `titanic_data.csv` file (available on the course git repository) containers information on 1309 passengers from aboard the Titanic (CSV stands for Comma-Separated-Values, a simple plain text format for storing spreadsheet data). Variables in this data set include gender, age, ticketed class, the passenger's destination, whether they survived, etc. We'll use this data set to explore some of the demographics of the passengers who were aboard the ship, and how their relationship to whether a passenger survived or not. For a detailed description of this data set, see this link.

We'll use this data to explore whether the saying "Women and children first!" applied on the Titanic.

### 2.3 Libraries

First we'll load some R libraries (packages) that contain useful functions that will make our analyses quicker and more efficient. We'll discuss the functions that these libraries provide, and how to use libraries in general, in greater detail in a future lecture.

```
library(ggplot2)
library(readr)
```

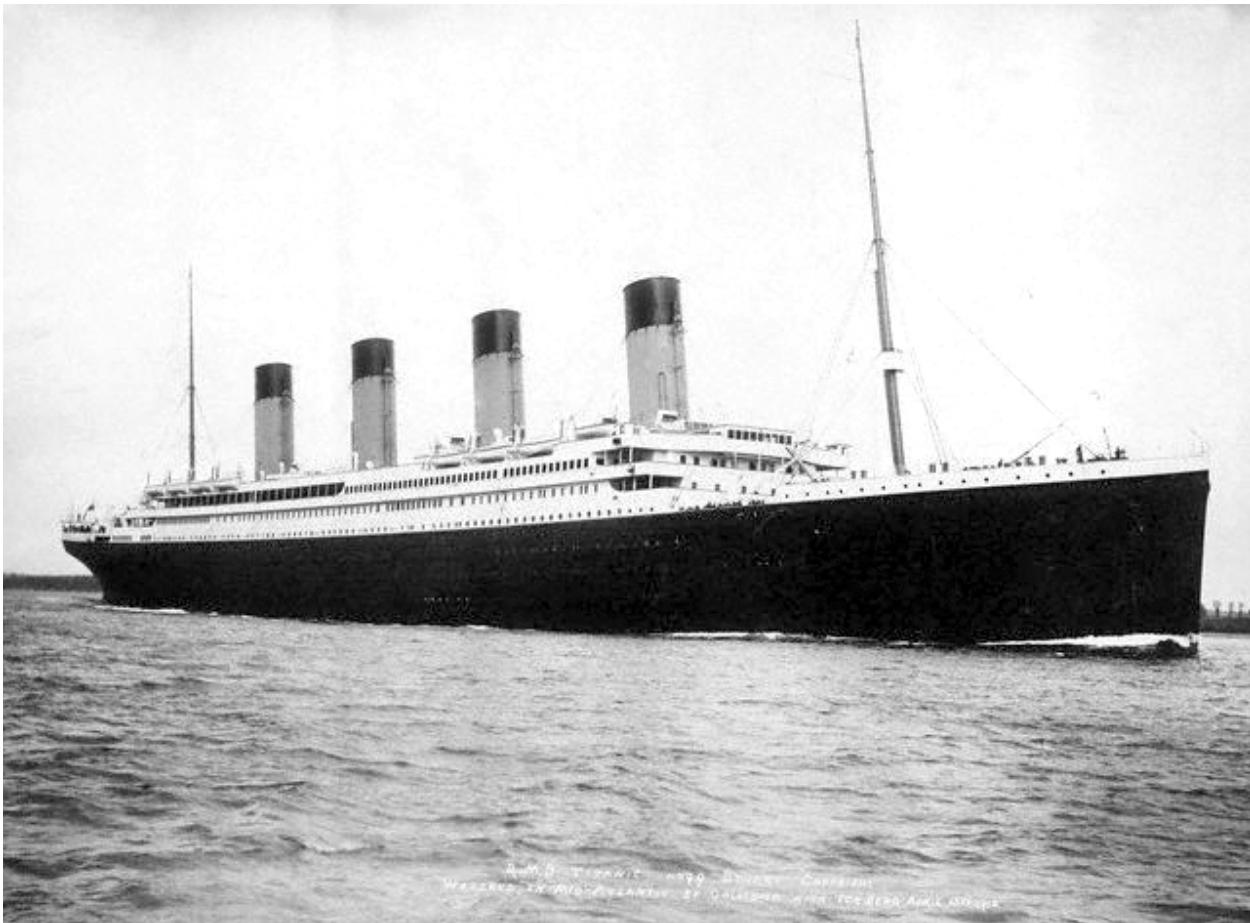


Figure 2.1: The Titanic

```
library(dplyr)
library(tidyr)
library(forcats)
```

## 2.4 Read data

We start by reading in the data from the CSV file.

```
titanic <- read_csv("~/Downloads/titanic_data.csv")
#> Parsed with column specification:
#> cols(
#>   pclass = col_integer(),
#>   survived = col_integer(),
#>   name = col_character(),
#>   sex = col_character(),
#>   age = col_double(),
#>   sibsp = col_integer(),
#>   parch = col_integer(),
#>   ticket = col_character(),
#>   fare = col_double(),
#>   cabin = col_character(),
#>   embarked = col_character(),
#>   boat = col_character(),
#>   body = col_integer(),
#>   home.dest = col_character()
#> )
```

The function `read_csv` does exactly what it advertises – reads a data set from a CSV file and returns it as an object we can compute on. In this case we assigned the variable name `titanic` to our data set. Simple enough!

## 2.5 What's in the data?

The function `read_csv` returns a table, where the columns represent the variables of interest (e.g. sex, age, etc) and the rows represent individuals or observations. Let's take a look at the first few rows of the data set using the `head()` function (can you guess the name of the corresponding function for looking at the last few rows?):

```
head(titanic)
#> # A tibble: 6 x 14
#>   pclass survived name      sex      age    sibsp  parch ticket     fare cabin
#>   <int>     <int> <chr> <chr>   <dbl> <int> <int> <chr>   <dbl> <chr>
#> 1     1       1 Alle~ fema~ 29        0     0 24160 211.3  B5
#> 2     1       1 Alli~ male  0.9167    1     2 113781 151.6  C22 ~
#> 3     1       0 Alli~ fema~ 2          1     2 113781 151.6  C22 ~
#> 4     1       0 Alli~ male  30         1     2 113781 151.6  C22 ~
#> 5     1       0 Alli~ fema~ 25         1     2 113781 151.6  C22 ~
#> 6     1       1 Ande~ male  48        0     0 19952  26.55 E12
#> # ... with 4 more variables: embarked <chr>, boat <chr>, body <int>,
#> #   home.dest <chr>
```

If we simply wanted the dimensions of the data we could do:

```
dim(titanic)
#> [1] 1309 14
```

whereas, if we wanted to get a list of the column names in the data we could do:

```
names(titanic)
#> [1] "pclass"      "survived"    "name"       "sex"        "age"
#> [6] "sibsp"       "parch"       "ticket"     "fare"       "cabin"
#> [11] "embarked"   "boat"        "body"      "home.dest"
```

### 2.5.1 Simple data wrangling

Two variables of interest to us are `pclass` (“passenger class”) and `survived`. These are categorical variables encoded as numbers. Before exploring the data we’re going to create derived “factor” variables from these, which will make our analyses more convenient. I’m also going to recode the “survived” information as the classes “died” and “lived”.

```
titanic <- mutate(titanic,
                    passenger.class = fct_recode(as.factor(pclass),
                                                 "1st" = "1", "2nd" = "2", "3rd" = "3"),
                    survival = fct_recode(as.factor(survived),
                                         "died" = "0", "lived" = "1"))
```

Having added to new variables to our data set, the dimensions and column names have changed:

```
dim(titanic)
#> [1] 1309 16
```

and

```
names(titanic)
#> [1] "pclass"      "survived"    "name"       "sex"        "age"
#> [4] "sibsp"       "parch"       "ticket"     "fare"       "cabin"
#> [7] "embarked"   "boat"        "body"      "home.dest"  "passenger.class"
#> [13] "survival"
```

Note that there are now 16 columns in our data, the original 14 plus our two new derived variables `passenger.class` and `survival`.

## 2.6 Categorizing passengers

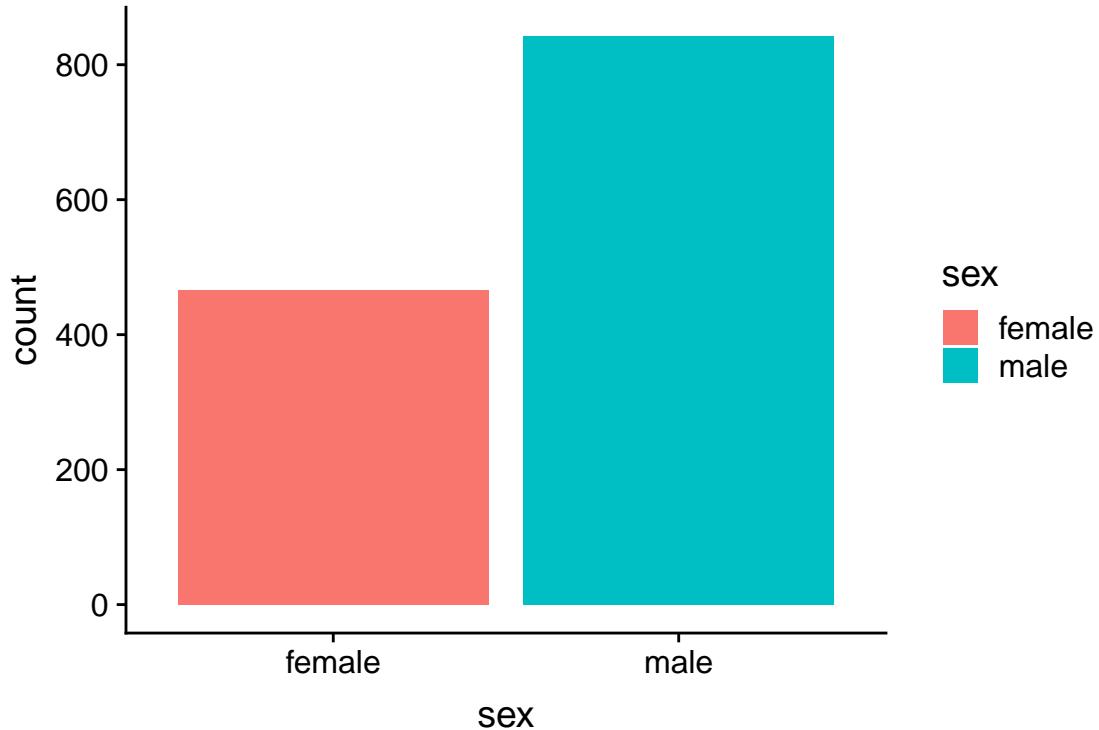
Let’s start by exploring various aspects of the 1309 passengers in our data set.

First, let’s look at the gender breakdown:

```
count(titanic, sex)
#> # A tibble: 2 x 2
#>   sex      n
#>   <chr> <int>
#> 1 female  466
#> 2 male   843
```

We could also represent this data as a bar graph (though a simple table is more efficient in this case):

```
ggplot(titanic) +
  geom_bar(aes(x = sex, fill = sex))
```



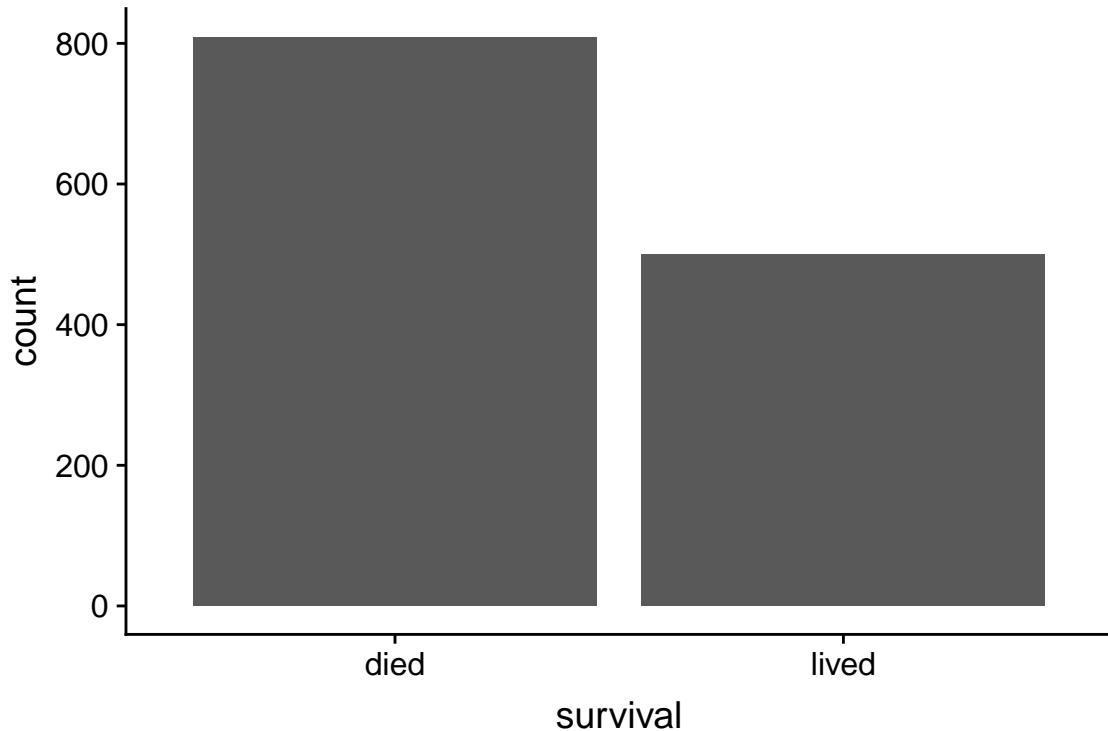
Both our table and bar graph tell us that there are almost twice as many men in our data set as women.

### 2.6.1 How many people survived?

Now let's look at survival information:

```
count(titanic, survival)
#> # A tibble: 2 x 2
#>   survival     n
#>   <fct>     <int>
#> 1 died       809
#> 2 lived      500

ggplot(titanic) +
  geom_bar(aes(x = survival))
```



We see that more than in the data we have at hand, roughly 60% (809 of 1309) of the passengers died.

## 2.6.2 Women first?

We can take our simple explorations a step further, by considering the counts of passengers with respect to multiple variables. Let's look at the relationship between gender and survival:

```
count(titanic, sex, survival)
#> # A tibble: 4 x 3
#>   sex     survival     n
#>   <chr>   <fct>    <int>
#> 1 female  died      127
#> 2 female  lived     339
#> 3 male    died      682
#> 4 male    lived     161
```

### 2.6.2.1 Contingency tables

When looking at counts of multiple variables simultaneously, a more traditional representation than the one above is a “contingency table”. The cells in a contingency table give the counts of individuals with respect to combinations of variables (e.g. # of women who survived, # of women who died, etc). Here's the same data on sex and survival represented as a contingency table:

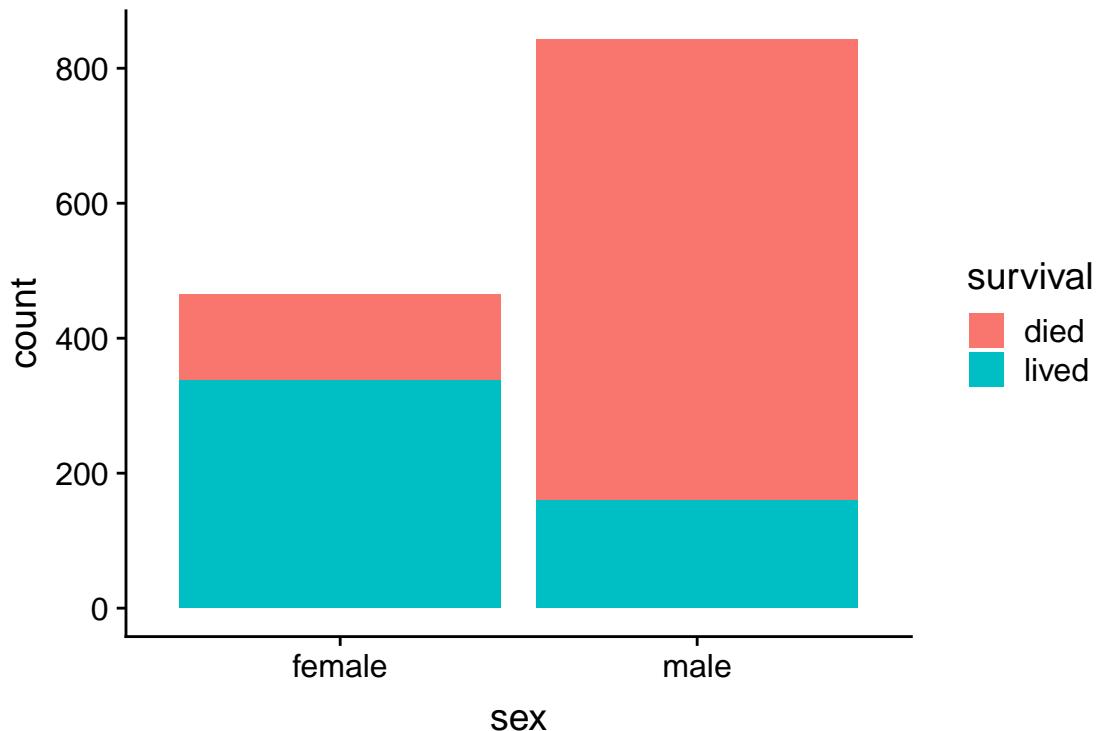
```
count(titanic, sex, survival) %>%
  spread(survival, n)
#> # A tibble: 2 x 3
#>   sex     died  lived
#>   <chr> <int> <int>
```

```
#> 1 female    127    339
#> 2 male      682    161
```

In the code above the symbol `%>%` can be read as “pipe” or “send”. The pipe operator inserts the object before the pipe as the first argument to the function after the pipe. Here we’re piping the output of the `count` function as the input into the `spread` function. We’ll see in later lectures that piping objects makes for very powerful workflows when we do more sophisticated analyses.

We can also create a bar plot to represent the contingency table:

```
ggplot(titanic) +
  geom_bar(aes(sex, fill = survival))
```

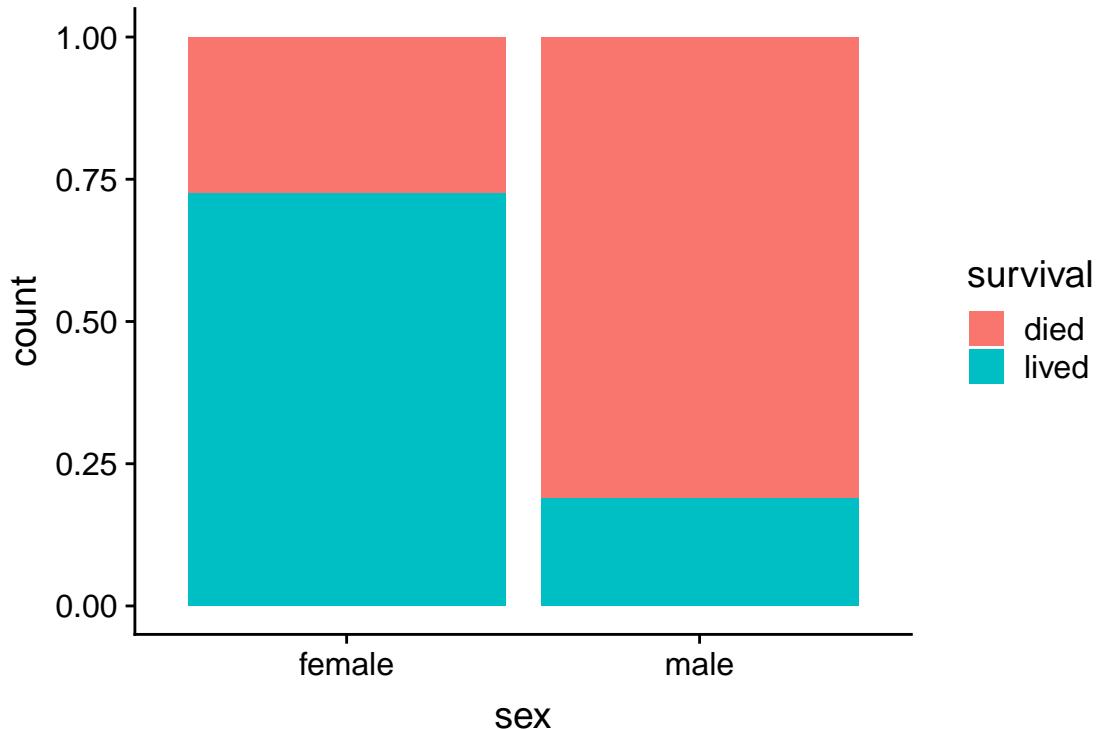


Here we’re already starting to see an interesting pattern – there were nearly twice as many men on the Titanic as women, but proportionally and in absolute numbers more women survived.

### 2.6.3 A bar plot using proportions rather than counts

Sometimes it’s useful to look at proportions rather than absolute numbers. Here’s a figure that allows us to visually assess the different proportions of men and women passengers who survived:

```
ggplot(titanic) +
  geom_bar(aes(sex, fill = survival), position = "fill")
```

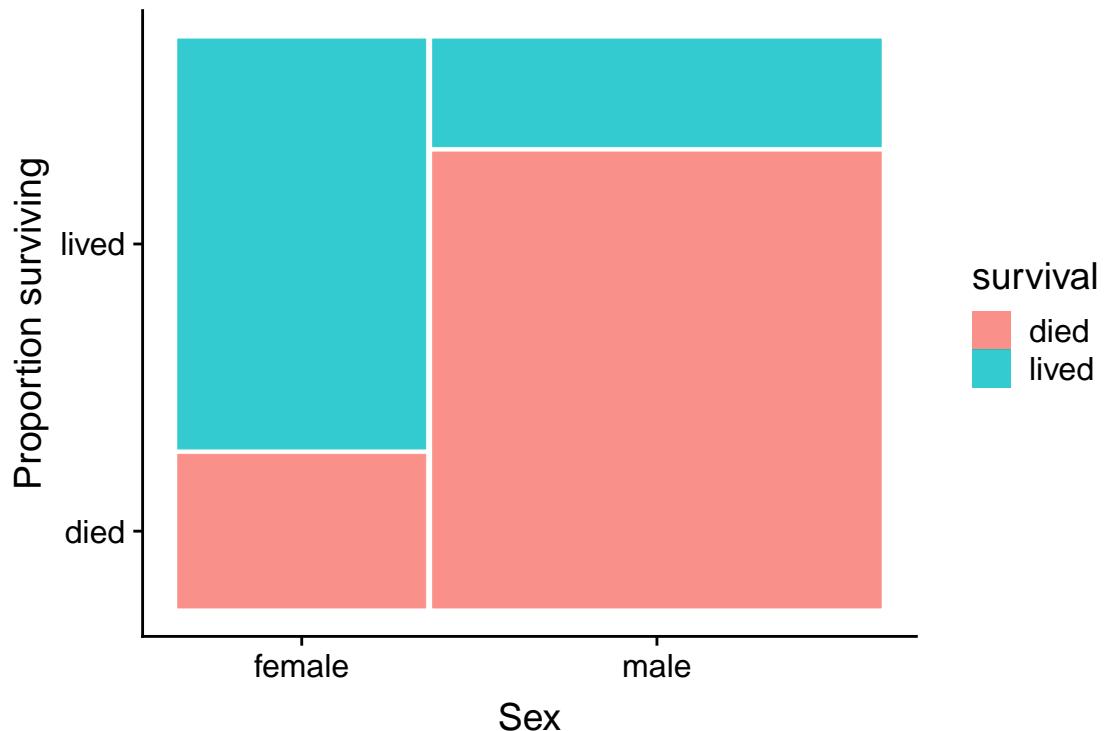


#### 2.6.4 Mosaic plots

A slightly more sophisticated version of a bar plot is called a “mosaic plot”. A mosaic plot is similar to a proportional bar plot but the width of the bars also varies, indicating the relative numbers of observations in each class. To create a mosaic plot we need to import the `geom_mosaic` function from a library called `ggridges`.

```
library(ggridges)

ggplot(titanic) +
  geom_mosaic(aes(x = product(sex), fill = survival)) +
  labs(x = "Sex", y = "Proportion surviving")
```

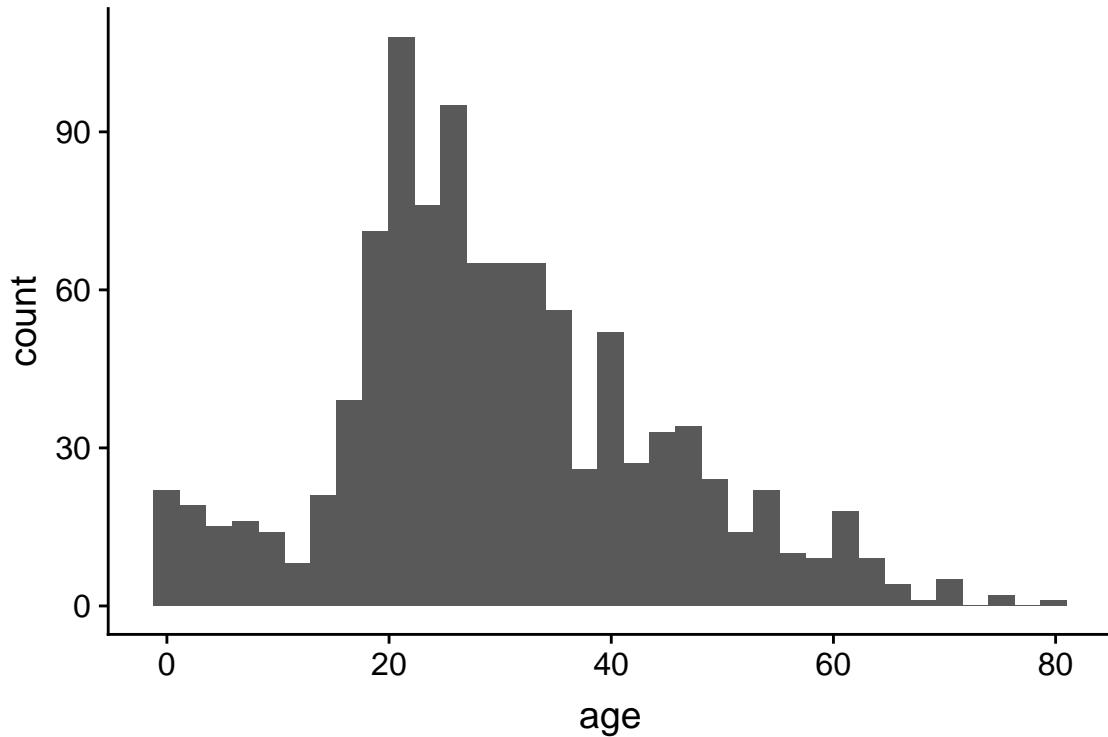


As you can see, the mosaic plot emphasizes both that there were more men than women on the Titanic, as well as the fact that a greater fraction of women survived. This seems like strong evidence for the first part of the phrase “Women and children first”.

## 2.7 Passenger ages

Now let's create a visualization to get a handle on the ages of passengers on the Titanic. A histogram is a common way to visualize the distribution of a continuous variable. Creating a histogram is a simple modification of our earlier examples where we created bar plots.

```
ggplot(titanic) +
  geom_histogram(aes(x = age), bins = 35)
```



The histogram provides a quick visual representation of the frequency of different age groups. It appears that the most common (modal) value of age is a little over 20 years old. We can explicitly calculate the mean and median age as follows:

```
mean(titanic$age, na.rm = TRUE)
#> [1] 29.88113
median(titanic$age, na.rm = TRUE)
#> [1] 28
```

Note that we have to explicitly tell the mean and median functions to drop any missing (NA) values. Alternately, we could have used pipes to calculate the mean and median as so:

```
titanic %>%
  filter(!is.na(age)) %>%
  summarize(mean(age), median(age))
#> # A tibble: 1 x 2
#>   `mean(age)` `median(age)`
#>        <dbl>      <dbl>
#> 1       29.88      28
```

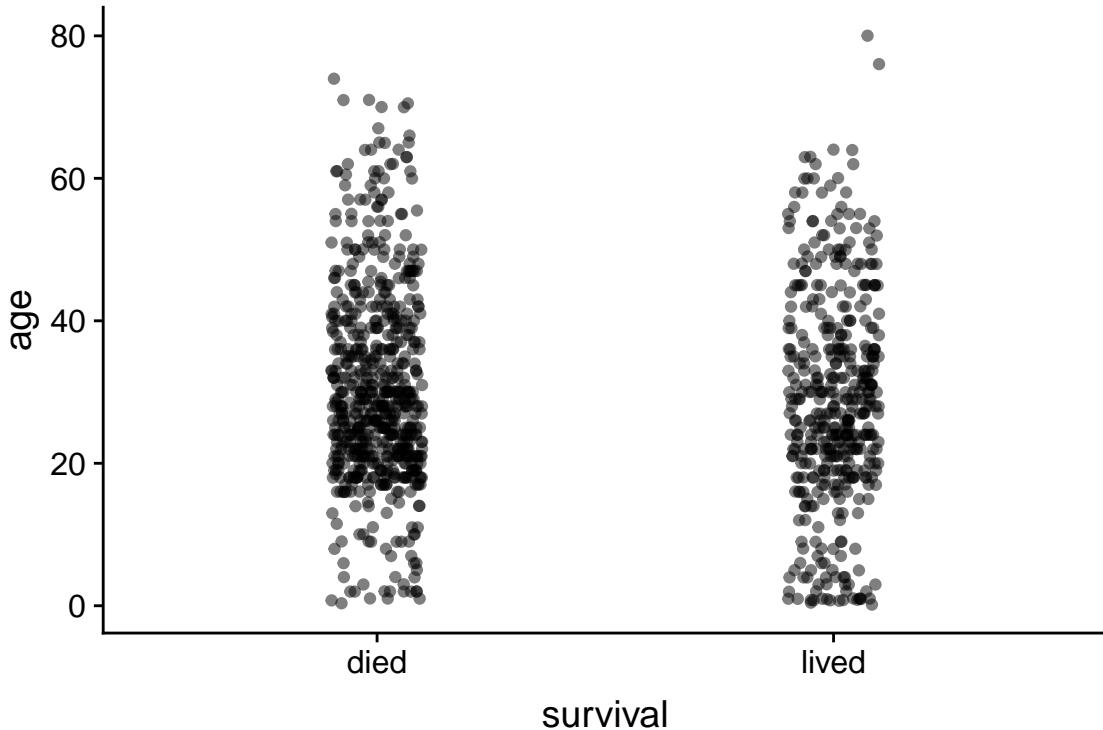
## 2.8 How does age relate to survival?

Now we turn to the question of how age relates to the probability of survival. Age is a continuous variable, while survival is a binary variable.

### 2.8.1 Strip charts

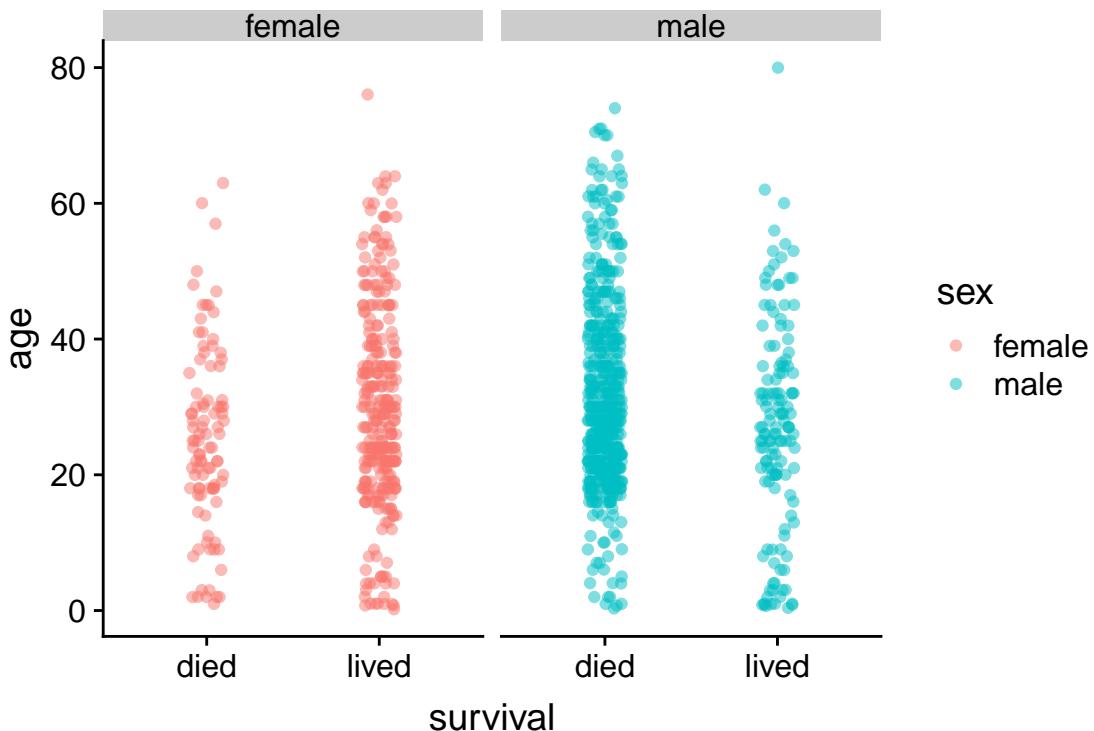
We'll start by creating "strip charts" that plot age on the y-axis, and survival on the x-axis. First we filter out individuals for which we have no age data, then we use a pipe to send the filtered data to the ggplot function:

```
titanic %>%
  filter(!is.na(age)) %>%
  ggplot() +
  geom_jitter(aes(survival, age), width = 0.1, alpha = 0.5)
```



Recall that sex was an important variable in our earlier assessment of the data. Let's create a second strip plot that takes into account both age and sex.

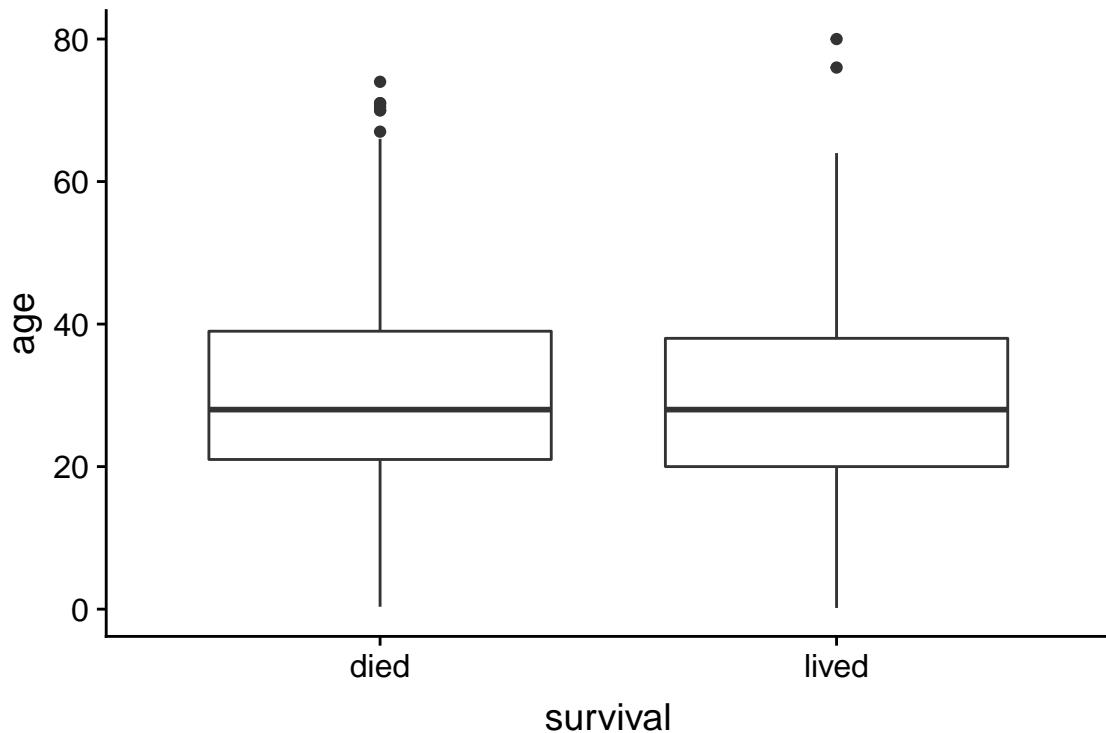
```
titanic %>%
  filter(!is.na(age)) %>%
  ggplot() +
  geom_jitter(aes(survival, age, color = sex), width = 0.1, alpha = 0.5) +
  facet_wrap(~sex)
```



### 2.8.2 Box plots

Another way to look at the data is to use a summary figure called a “box plot”. First the simple boxplot, relating age and survival:

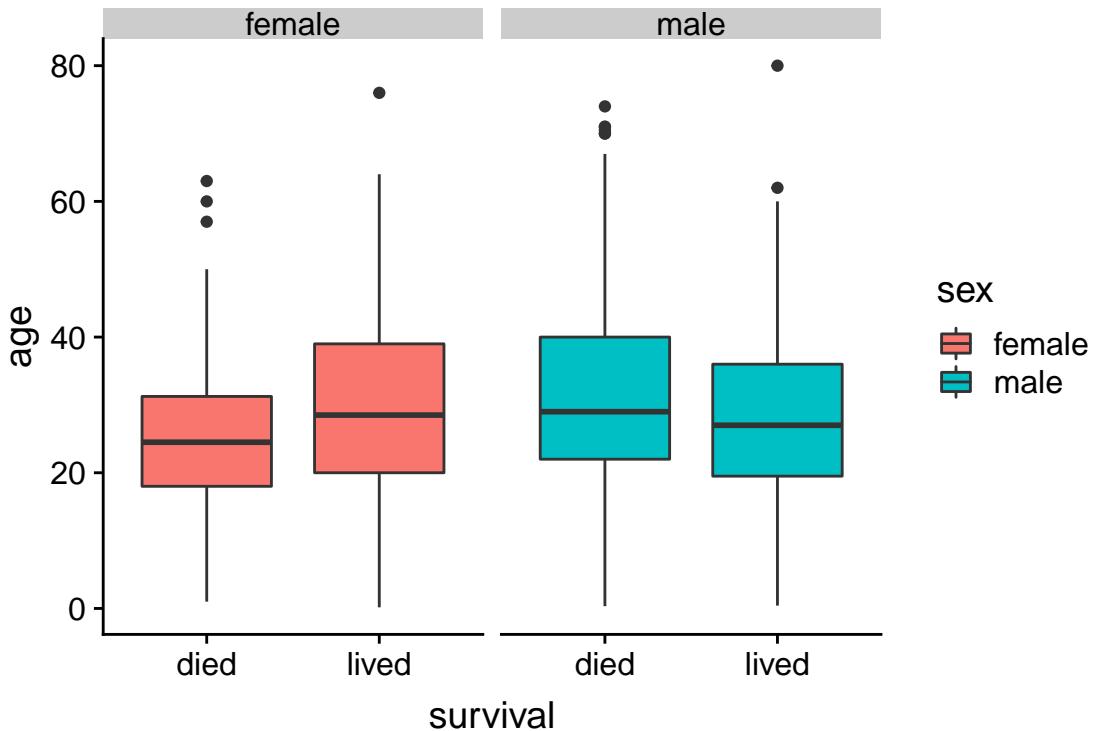
```
titanic %>%
  filter(!is.na(age)) %>%
  ggplot() +
  geom_boxplot(aes(survival, age))
```



A box plot depicts information about the median value (thick central line), the first and third quartiles (lower 25% value, upper 75% value) and outliers. From this simple boxplot, it doesn't look like there is a great difference in the age distribution of passengers who lived vs died.

Now we look at box plot for age-by-survival, conditioned on sex.

```
titanic %>%
  filter(!is.na(age)) %>%
  ggplot() +
  geom_boxplot(aes(survival, age, fill = sex)) +
  facet_wrap(~sex)
```

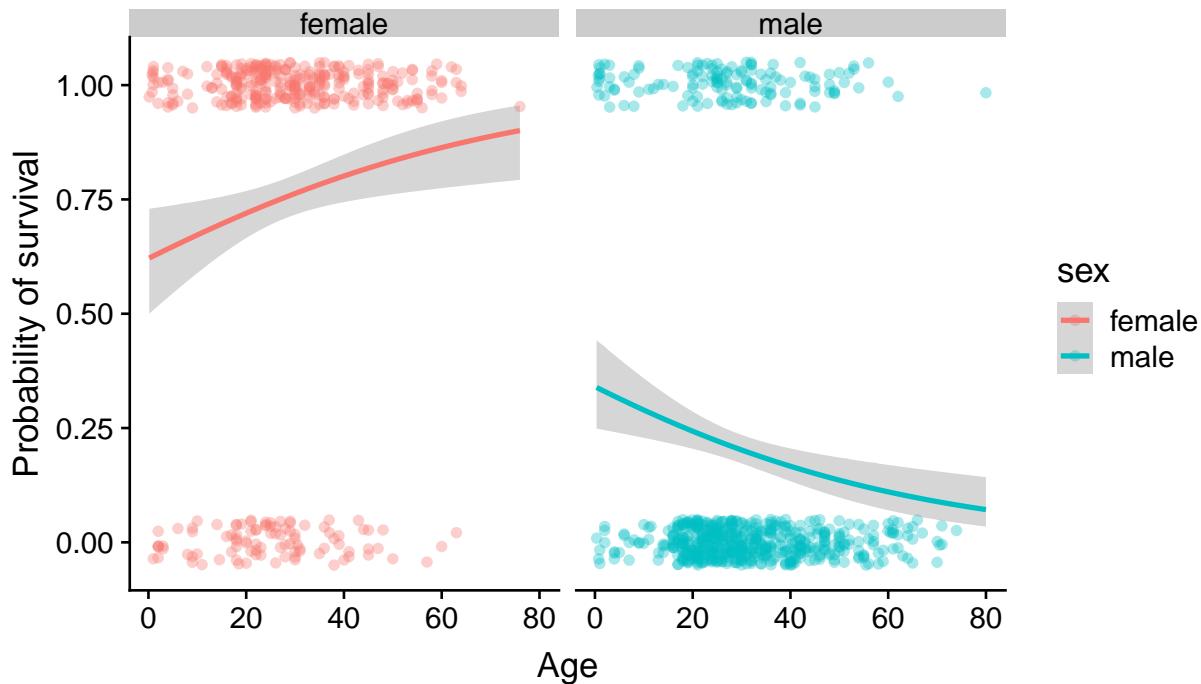


Here we start to see a more interesting pattern. When comparing male passengers, the median age of survivors appears to be a little younger than those who died. However, when comparing female passengers the opposite pattern seems to be at play.

### 2.8.3 Fitting a model relating survival to age and sex

We can get more sophisticated by fitting a formal model to our data. Here we use a technique called logistic regression to model the probability of survival as a function of age, broken down for female and male passengers separately.

```
titanic %>%
  filter(!is.na(age)) %>%
  ggplot(aes(x = age, y = survived, color = sex)) +
  geom_jitter(height = 0.05, alpha = 0.35) +
  geom_smooth(method="glm", method.args = list(family="binomial")) +
  facet_wrap(~sex) +
  labs(x = "Age", y = "Probability of survival")
```



The logistic regression model fit here, seems to support the trend we saw in the box plots. When breaking the data down by sex we find that there is a decreasing probability of survival as age increases for male passengers, but the opposite trend for female passengers. This is pretty interesting – “children first” seemed to hold for men but not for women.

## 2.9 How does class affect survival?

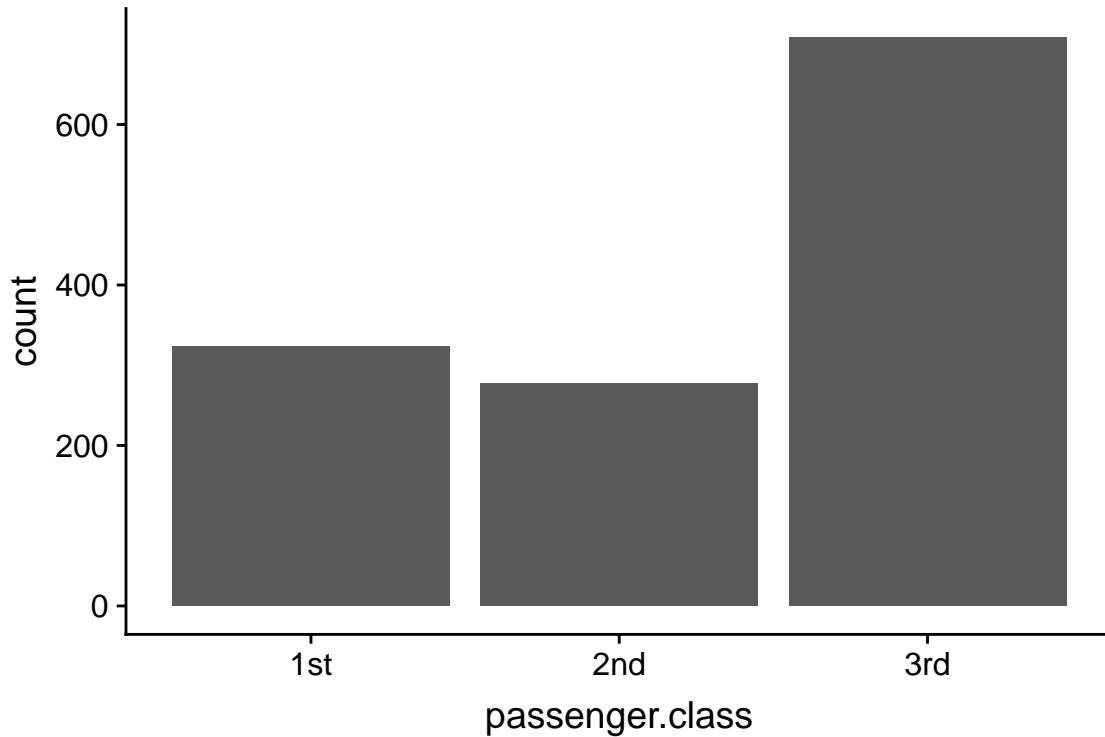
A passenger’s ticket class (a proxy for socioeconomic status) is another interesting variable to consider in conjunction with sex and survival.

First, let’s consider class on its own.

```
count(titanic, passenger.class)
#> # A tibble: 3 x 2
#>   passenger.class     n
#>   <fct>        <int>
#> 1 1st            323
#> 2 2nd            277
#> 3 3rd            709
```

And as a bar graph:

```
ggplot(titanic) +
  geom_bar(aes(x = passenger.class))
```

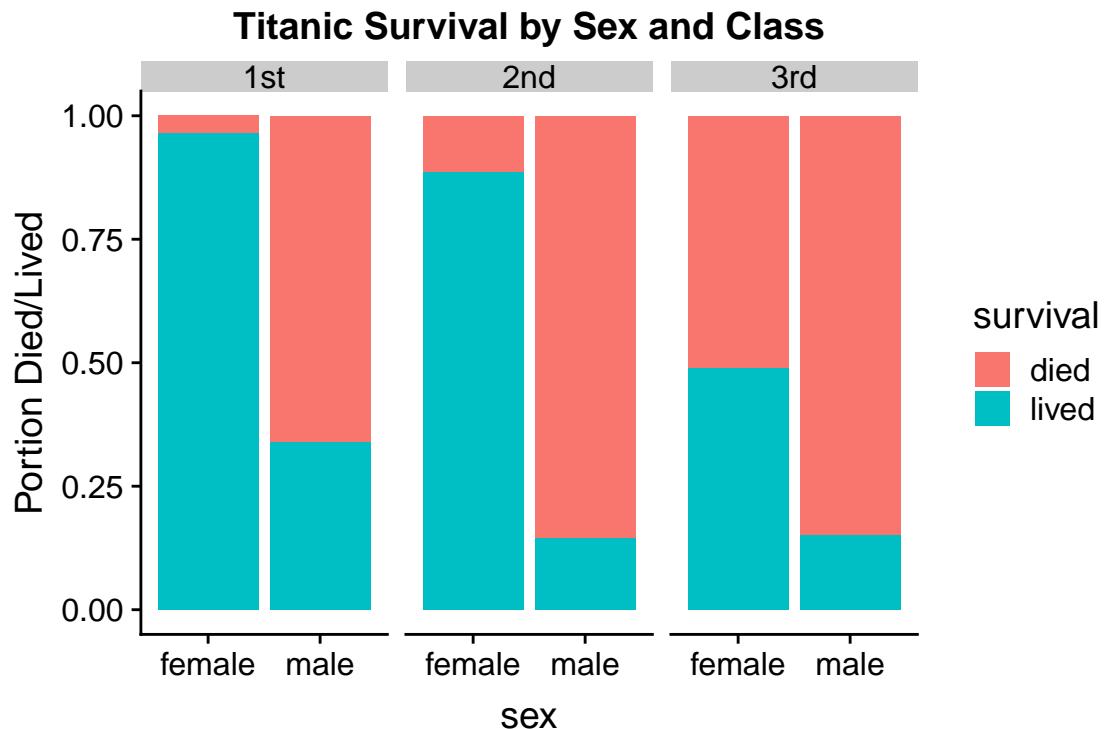


Both our table and bar graph tell us that the largest number of passengers (in the available data) were travelling on third-class tickets. The numbers of first- and second-class ticketed passengers are fairly similar.

### 2.9.1 Combining Sex and Class

Here is a bar plot that takes into account sex, class, and survival by representing passenger class as a “facet” variable – we create different subplots for each class grouping.

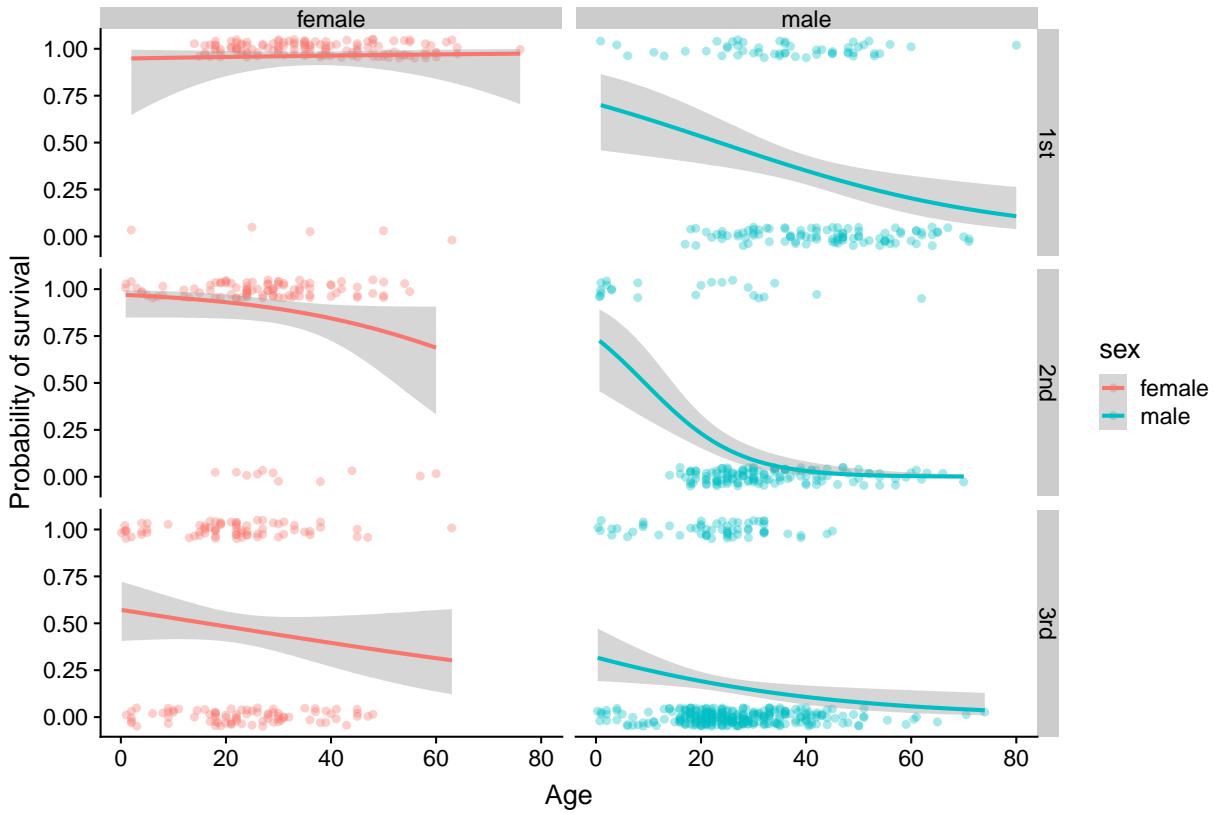
```
ggplot(titanic) +
  geom_bar(aes(sex, fill = survival), position = "fill") +
  facet_wrap(~ passenger.class) +
  labs(y = "Portion Died/Lived", title = "Titanic Survival by Sex and Class")
```



### 2.9.2 Extending the logistic regression model to consider class, age, sex, and survival

As a final example, let generate a visualization and model to jointly consider how sex, age, and class affected survival on the Titanic. Here we treat both sex and passenger class as faceting variables, fitting six different logistic regression models (all combinations of the three classes and two sexes).

```
titanic %>%
  filter(!is.na(age)) %>%
  ggplot(aes(x = age, y = survived, color = sex)) +
  geom_jitter(height = 0.05, alpha = 0.35) +
  geom_smooth(method="glm", method.args = list(family="binomial")) +
  facet_grid(passenger.class~sex) +
  labs(x = "Age", y = "Probability of survival")
```



By conditioning on both sex and ticketing class, we gain even more insights into the data and our assessments of relationships between variables can change.

For female passengers, ticketed class appears to have a strong influence on the relationship between age and survival. We see that almost all of the female first-class passengers survived, and the relationship between age and survival is thus flat. The second class female passengers are fairly similar, though with a slight decrease in the probability of survival among the oldest passengers. It's not until we get to the third class passengers that we see a strong indication of the "children first" relationship playing out in terms of survival.

For the male passengers, the "children first" model seems to fit across classes, but note the generally lower probability of survival across ages when comparing first and third class passengers.

## 2.10 Conclusion

We've only scratched the surface of the possible explorations we could undertake of this interesting data set. However, this introductory "data story" illustrates many of the tools and ideas we'll encounter in this class. You'll be undertaking even more complex data explorations on your own by the end of the semester!

# Chapter 3

## Getting Started with R

### 3.1 What is R?

R is a statistical computing environment and programming language. It is free, open source, and has a large and active community of developers and users. There are many different R packages (libraries) available for conducting a wide variety of different analyses, for everything from genome sequence data to geospatial information.

### 3.2 What is RStudio?

RStudio (<http://www.rstudio.com/>) is an open source integrated development environment (IDE) that provides a nicer graphical interface to R than does the default R GUI.

The figure below illustrates the RStudio interface, in its default configuration. For the exercises in this chapter you'll be primarily entering commands in the "console" window. We'll review key parts of the RStudio interface in greater detail in class.

### 3.3 Entering commands in the console

You can type commands directly in the console. When you hit Return (Enter) on your keyboard the text you typed is evaluated by the R interpreter. This means that the R program reads your commands, makes sure there are no syntax errors, and then carries out any commands that were specified.

Try evaluating the following arithmetic commands in the console:

```
10 + 5  
10 - 5  
10 / 5  
10 * 5
```

If you type an incomplete command and then hit Return on your keyboard, the console will show a continuation line marked by a + symbol. For example enter the incomplete statement (10 + 5 and then hit Enter. You should see something like this.

```
> (10 + 5  
+
```

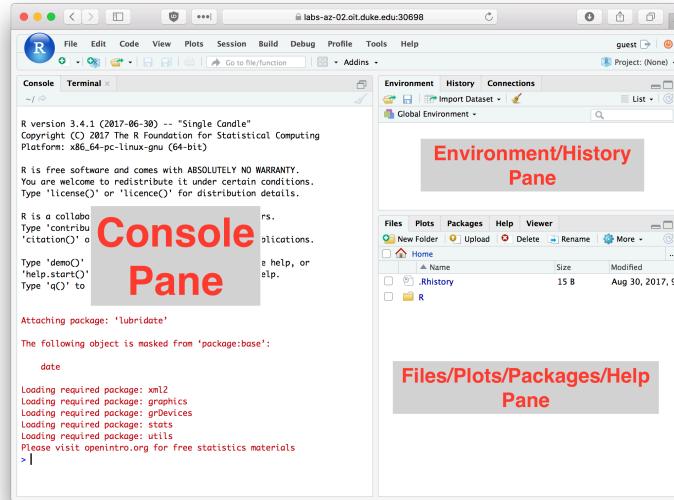


Figure 3.1: RStudio window with the panes labeled

The continuation line tells you that R is waiting for additional input before it evaluates what you typed. Either complete your command (e.g. type the closing parenthesis) and hit Return, or hit the “Esc” key to exit the continuation line without evaluating what you typed.

## 3.4 Comments

When working in the R console, or writing R code, the pound symbol (#) indicates the start of a comment. Anything after the #, up to the end of the current line, is ignored by the R interpreter.

```
# This line will be ignored
5 + 4 # the first part of this line, up to the #, will be evaluated
2 # * 3 + 4 * 5
```

Throughout this course I will often include short explanatory comments in the code examples.

## 3.5 Using R as a Calculator

The simplest way to use R is as a fancy calculator.

```
10 + 2 # addition
10 - 2 # subtraction
10 * 2 # multiplication
10 / 2 # division
10 ^ 2 # exponentiation
10 ** 2 # alternate exponentiation
pi * 2.5^2 # R knows about some constants such as Pi
10 %% 3 # modulus operator -- gives remainder after division
10 %/% 3 # integer division
```

Be aware that certain operators have precedence over others. For example multiplication and division have higher precedence than addition and subtraction. Use parentheses to disambiguate potentially confusing

statements.

```
(10 + 2)/4-5 # was the output what you expected?
(10 + 2)/(4-5) # compare the answer to the above
```

Division by zero produces an object that represents infinite numbers. Infinite values can be either positive or negative

```
1/0 # Inf
-1/0 # -Inf
```

Invalid calculations produce a object called `NaN` which is short for “Not a Number”:

```
0/0 # invalid calculation
```

### 3.5.1 Common mathematical functions

Many commonly used mathematical functions are built into R. Here are some examples:

```
abs(-3) # absolute value
abs(3)
cos(pi/3) # cosine
sin(pi/3) # sine
log(10) # natural logarithm
log10(10) # log base 10
log2(10) # log base 2
exp(1) # exponential function
sqrt(10) # square root
10 ^ 0.5 # same as square root
```

## 3.6 The R Help System

R comes with fairly extensive documentation and a simple help system. You can access HTML versions of the R documentation under the Help tab in Rstudio. The HTML documentation also includes information on any packages you’ve installed. Take a few minutes to browse through the R HTML documentation. In addition to the HTML documentation there is also a search box where you can enter a term to search on (see red arrow in figure below).

### 3.6.1 Getting help from the console

In addition to getting help from the RStudio help tab, you can directly search for help from the console. The help system can be invoked using the `help` function or the `?` operator.

```
help("log")
?log
```

If you are using RStudio, the help results will appear in the “Help” tab of the Files/Plots/Packages/Help/Viewer (lower right window by default).

What if you don’t know the name of the function you want? You can use the `help.search()` function.

```
help.search("log")
```

In this case `help.search("log")` returns all the functions with the string `log` in them. For more on `help.search` type `?help.search`.

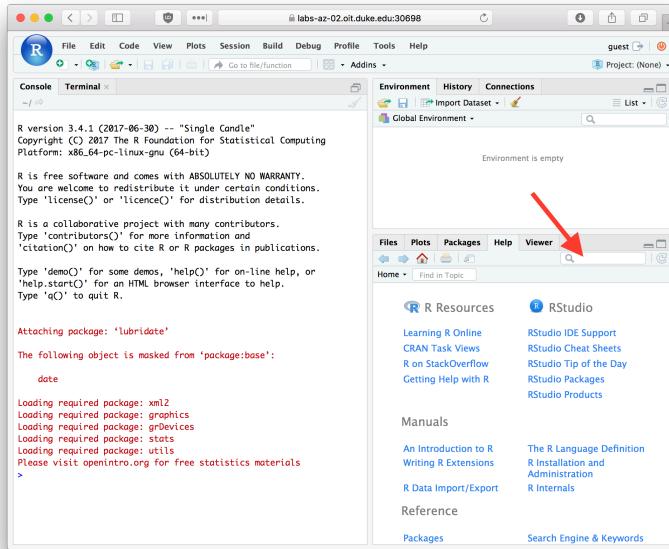


Figure 3.2: The RStudio Help tab

Other useful help related functions include `apropos()` and `example()`. `apropos` returns a list of all objects (including function names) in the current session that match the input string.

```
apropos("log")
```

`example()` provides examples of how a function is used.

```
example(log)
```

## 3.7 Variable assignment in R

An important concept in all programming languages is that of “variable assignment”. Variable assignment is the act of creating labels that point to particular data values in a computer’s memory, which allows us to apply operations to the labels rather than directly to specific values. Variable assignment is an important mechanism of abstracting and generalizing computational operations.

Variable assignment in R is accomplished with the assignment operator, which is designated as `<-` (left arrow, constructed from a left angular bracket and the minus sign). This is illustrated below:

```
x <- 10 # assign the variable name 'x' the value 10
sin(x) # apply the sin function to the value x points to
x <- pi # x now points to a different value
sin(x) # the same function call now produces a different result
```

### 3.7.1 Valid variable names

As described in the R documentation, “A syntactically valid name consists of letters, numbers and the dot or underline characters and starts with a letter or the dot not followed by a number. Names such as ‘`2way`’ are not valid, and neither are the reserved words.”

Here are some examples of valid and invalid variable names. Mentally evaluate these based on the definition above, and then evaluate these in the R interpetter to confirm your understanding :

```
x <- 10
x.prime <- 10
x_prime <- 10
my.long.variable.name <- 10
another_long_variable_name <- 10
_x <- 10
.x <- 10
2.x <- 2 * x
```

## 3.8 Data types

All of our arithmetic examples above produced numerical values. The default numerical data type in R is called a “double”. “double” is short for “double precision floating point value” which refers to the numerical precision that R uses when carrying out calculations.

R has a function called `typeof()` that we can use to get information about an object’s type. Let’s illustrate the use of `typeof()` and confirm that our numerical calculations return objects of the “double” type:

```
typeof(10)
x <- 10.5
typeof(x)
```

## 3.9 Logical values

When we compare values to each other, our calculations no longer return “doubles” but rather `TRUE` and `FALSE` values. This is illustrated below:

```
10 < 9 # is 10 less than 9?
10 > 9 # is 10 greater than 9?
10 <= (5 * 2) # less than or equal to?
10 >= pi # greater than or equal to?
10 == 10 # equals?
10 != 10 # does not equal?
```

`TRUE` and `FALSE` objects are of “logical” data type (known as “Booleans” in many other languages, after the mathematician George Boole).

```
typeof(TRUE)
typeof(FALSE)
x <- FALSE
typeof(x)
```

When working with numerical data, tests of equality can be tricky. For example, consider the following two comparisons:

```
10 == (sqrt(10)^2) # Surprised by the result? See below.
4 == (sqrt(4)^2) # Even more confused?
```

Mathematically we know that both  $(\sqrt{10})^2 = 10$  and  $(\sqrt{4})^2 = 4$  are true statements. Why does R tell us the first statement is false? What we’re running into here are the limits of computer precision. A computer can’t represent  $\sqrt{10}$  exactly, whereas  $\sqrt{4}$  can be exactly represented. Precision in numerical computing is a

complex subject and a detailed discussion is beyond the scope of this course. However, it's important to be aware of this limitation (this limitation is true of any programming language, not just R).

To test “near equality” R provides a function called `all.equal()`. This function takes two inputs – the numerical values to be compared – and returns True if their values are equal up to a certain level of tolerance (defined by the built-in numerical precision of your computer).

```
all.equal(10, sqrt(10)^2)
```

### 3.9.1 Logical operators

Logical values support Boolean operations, like logical negation (“not”), “and”, “or”, “xor”, etc. This is illustrated below:

```
x <- TRUE
y <- FALSE
!x # logical negation -- reads as "not x"
x & y # AND: are x and y both TRUE?
x | y # OR: are either x or y TRUE?
xor(x,y) # XOR: is either x or y TRUE, but not both?
```

The function `isTRUE` is sometimes useful:

```
x <- 5
y <- 10
z <- x > y
isTRUE(z)
```

## 3.10 Character strings

Character strings (“character”) represent single textual characters or a longer sequence of characters. They are created by enclosing the characters in text either single our double quotes.

```
typeof("abc") # double quotes
#> [1] "character"
typeof('abc') # single quotes
#> [1] "character"
```

Character strings have a length, which can be found using the `nchar` function:

```
first.name <- "jasmine"
nchar(first.name)
#> [1] 7

last.name <- 'smith'
nchar(last.name)
#> [1] 5
```

There are a number of built-in functions for manipulating character strings. Here are some of the most common ones.

### 3.10.1 Joining strings

The `paste()` function joins two characters strings together:

```
paste(first.name, last.name)  # join two strings
#> [1] "jasmine smith"
paste("abc", "def")
#> [1] "abc def"
```

Notice that `paste()` adds a space between the strings? If we didn't want the space we can call the `paste()` function with an optional argument called `sep` (short for separator) which specifies the character(s) that are inserted between the joined strings.

```
paste("abc", "def", sep = "")  # join with no space; "" is an empty string
#> [1] "abcdef"
paste("abc", "def", sep = "|") # join with a vertical bar
#> [1] "abc|def"
```

### 3.10.2 Splitting strings

The `strsplit()` function allows us to split a character string into substrings according to matches to a substring. For example, we could break a sentence into its constituent words as follows:

```
sentence <- "Call me Ishmael."
words <- strsplit(sentence, " ")  # split on space
words
#> [[1]]
#> [1] "Call"      "me"        "Ishmael."
```

Notice that `strsplit()` is the reverse of `paste()`.

### 3.10.3 Substrings

The `substr()` function allows us to extract a substring from a character object by specifying the first and last positions (indices) to use in the extraction:

```
substr("abcdef", 2, 5)  # get substring from characters 2 to 5
#> [1] "bcde"
substr(first.name, 1, 3) # get substring from characters 1 to
#> [1] "jas"
```

## 3.11 Packages

Packages are libraries of R functions and data that provide additional capabilities and tools beyond the standard library of functions included with R. Hundreds of people around the world have developed packages for R that provide functions and related data structures for conducting many different types of analyses.

Throughout this course you'll need to install a variety of packages. Here I show the basic procedure for installing new packages from the console as well as from the R Studio interface.

### 3.11.1 Installing packages from the console

The built-in function `install.packages` provides a quick and convenient way to install packages from the R console.

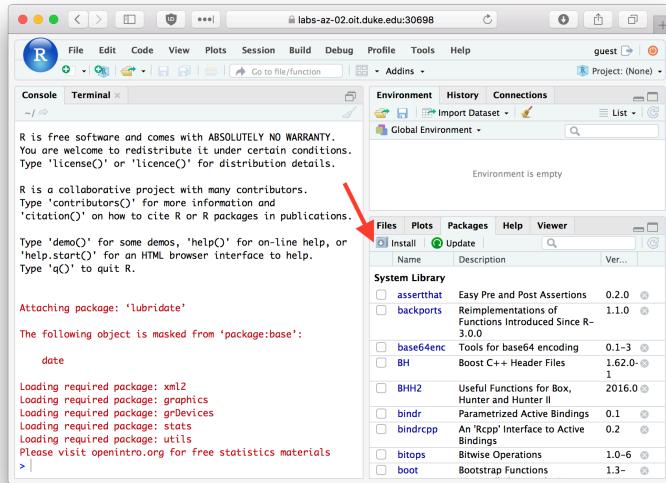


Figure 3.3: The Packages tab in RStudio

### 3.11.2 Install the tidyverse package

To illustrate the use of `install.package`, we'll install a collection of packages (a “meta-package”) called the tidyverse. Here's how to install the tidyverse meta-package from the R console:

```
install.packages("tidyverse", dependencies = TRUE)
```

The first argument to `install.packages` gives the names of the package we want to install. The second argument, `dependencies = TRUE`, tells R to install any additional packages that tidyverse depends on.

### 3.11.3 Installing packages from the RStudio dialog

You can also install packages using a graphical dialog provided by RStudio. To do so pick the `Packages` tab in RStudio, and then click the `Install` button.

In the packages entry box you can type the name of the package you wish to install.

### 3.11.4 Loading packages with the `library()` function

Once a package is installed on your computer, the package can be loaded into your R session using the `library` function. To insure our previous install commands worked correctly, let's load the packages we just installed.

```
library(tidyverse)
```

Since the tidyverse pacakge is a “meta-package” it provides some additional info about the sub-packages that got loaded.

When you load tidyverse, you will also see a message about “Conflicts” as several of the functions provided in the dplyr package (a sub-package in tidyverse) conflict with names of functions provided by the “stats” package which usually gets automatically loaded when you start R. The conflicting funcdtions are `filter` and `lag`. The conflicting functions in the stats package are `lag` and `filter` which are used in time series analysis.

The `dplyr` functions are more generally useful. Furthermore, if you need these masked functions you can still access them by prefacing the function name with the name of the package (e.g. `stats::filter`).

## 3.12 Reading data from a file

We'll use the `read_csv` function defined in the `readr` package (loaded via `tidyverse`) to read a CSV formatted data file. We'll load a data file called `possums.csv`.

```
possums <- read_csv("https://tinyurl.com/bio304-possums-csv")
```

The possums data set includes information from a study of mountain brushtail possums (*Trichosurus caninus*; Lindenmayer DB et al. 1995, Australian Journal of Zoology 43, 449-458.) The investigators recorded variables about individual possum's sex, age, where they were collected, and a range of morphological measurements. For example the variable `skullw` is the width of the skull, `taill` is tail length, etc.

Notice that we read the CSV file directly from a remote file via a URL. If instead, you wanted to load a local file on your computer you would specify the “path” – i.e. the location on your hard drive where you stored the file. For example, here is how I would load the same file if it was stored in the standard downloads directory on my Mac laptop:

```
# this assumes you're running a local version of R (i.e. on your laptop)
possums <- read_csv("/Users/pmagwene/Downloads/possums.csv")
```

## 3.13 Exploring the “possums” data set

Let's take a moment to explore the possums data set. First, how big is it? The `dim` function will tell us the dimensions (# of rows and columns) of the data table we loaded:

```
dim(possums)
```

What are the names of the columns of the table?

```
names(possums)
```

## 3.14 Simple tables

Let's use the `count` function to count the number of male and female possums and the number collected from each of the populations of interest in this study (Victoria and “other”):

```
count(possums, sex)
count(possums, Pop)
```

We can break down the counts by population and sex combined by specifying both variables:

```
count(possums, Pop, sex)
```

How do the results differ if instead you write `count(possums, sex, Pop)`?

Finally, let's get counts for the different age groups in the study:

```
count(possums, age)
```

Notice that besides the nine age groups, there is a grouping for “NA”. “NA” means “not available” and is the standard designation for missing values. This table tells us there are two possums for which age estimates were not available.

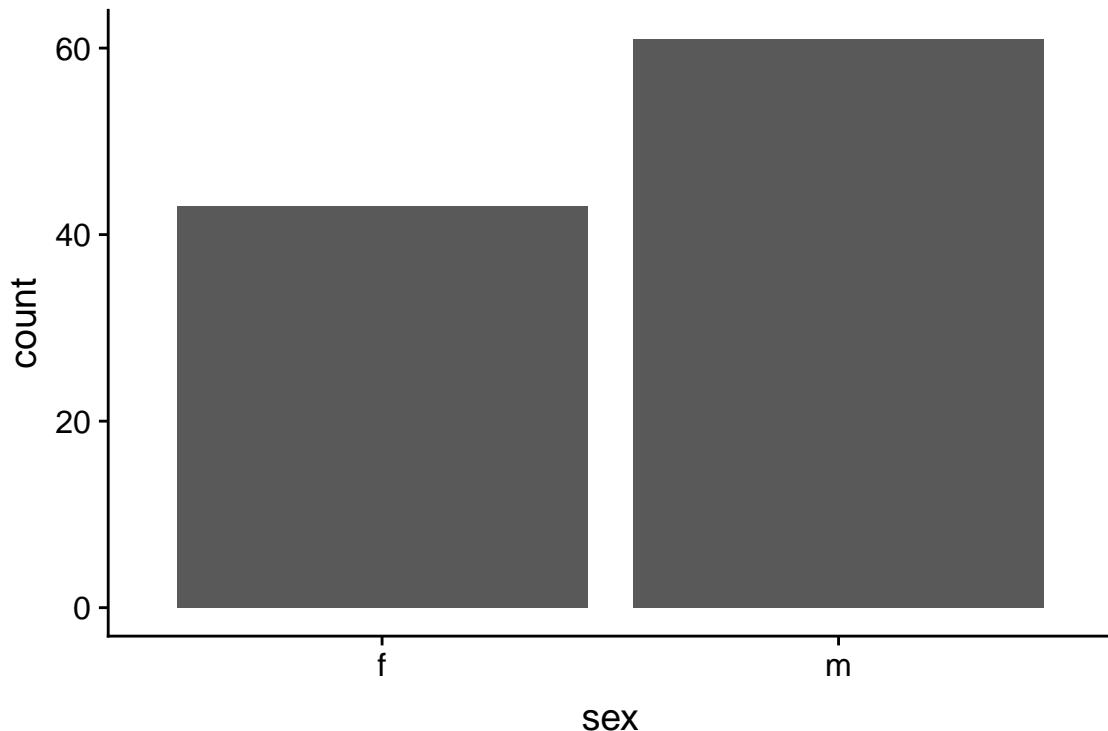
## 3.15 Simple figures

Throughout this course we'll use a package called “ggplot2” to generate figures. The ggplot2 package is part of the tidyverse and is automatically loaded when we load the tidyverse library. In the code below, we'll demonstrate how to use ggplot by example. In a later lecture we'll go into greater detail about how ggplot is structured and the broader conceptual framework that underpins its design.

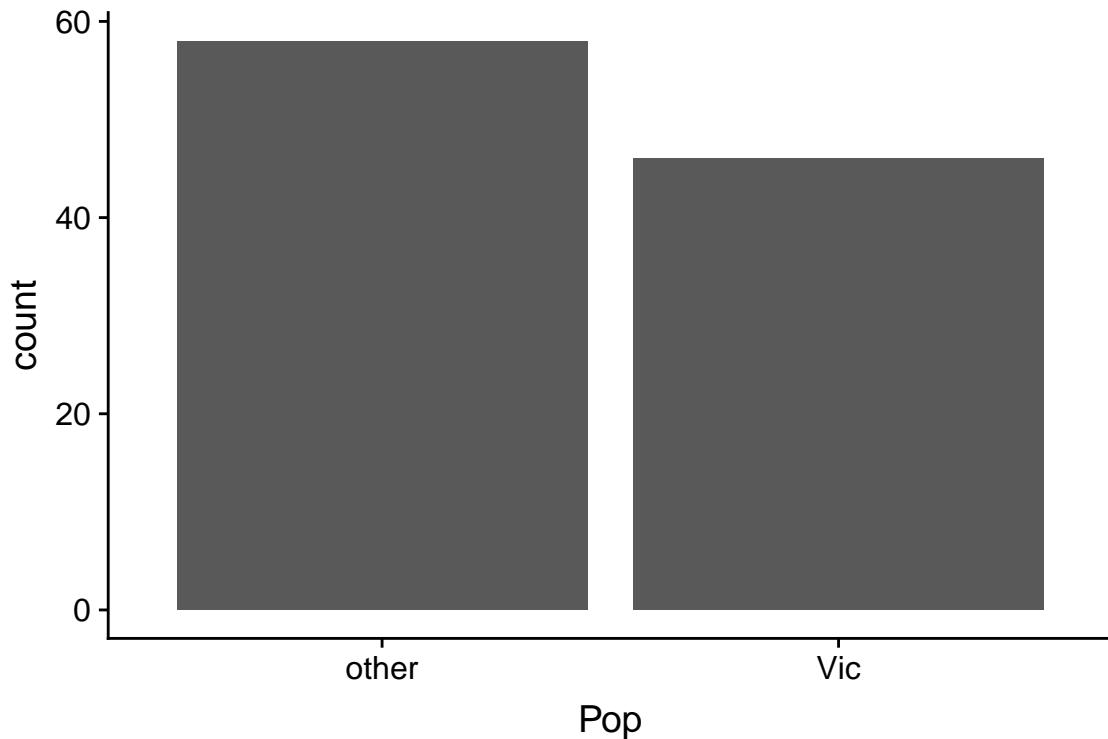
## 3.16 Bar plots

First let's create some simple bar plots as alternate representations of the counts:

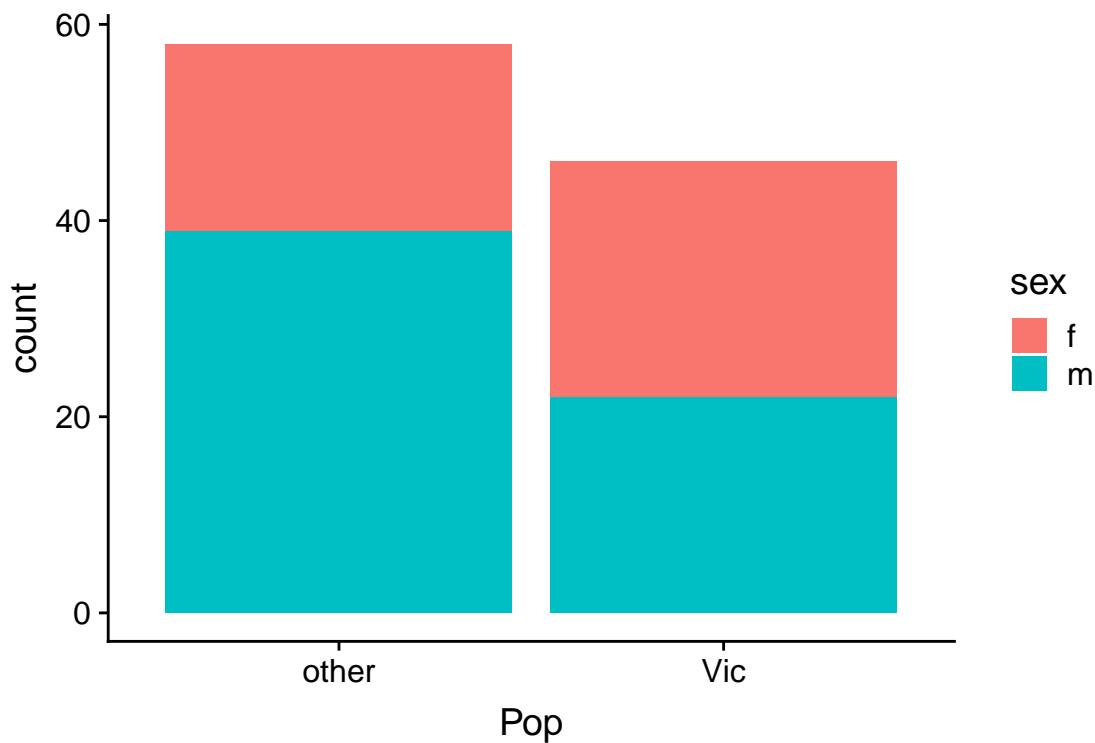
```
ggplot(possums) + geom_bar(aes(x = sex))
```



```
ggplot(possums) + geom_bar(aes(x = Pop))
```



```
ggplot(possums) + geom_bar(aes(x = Pop, fill = sex))
```

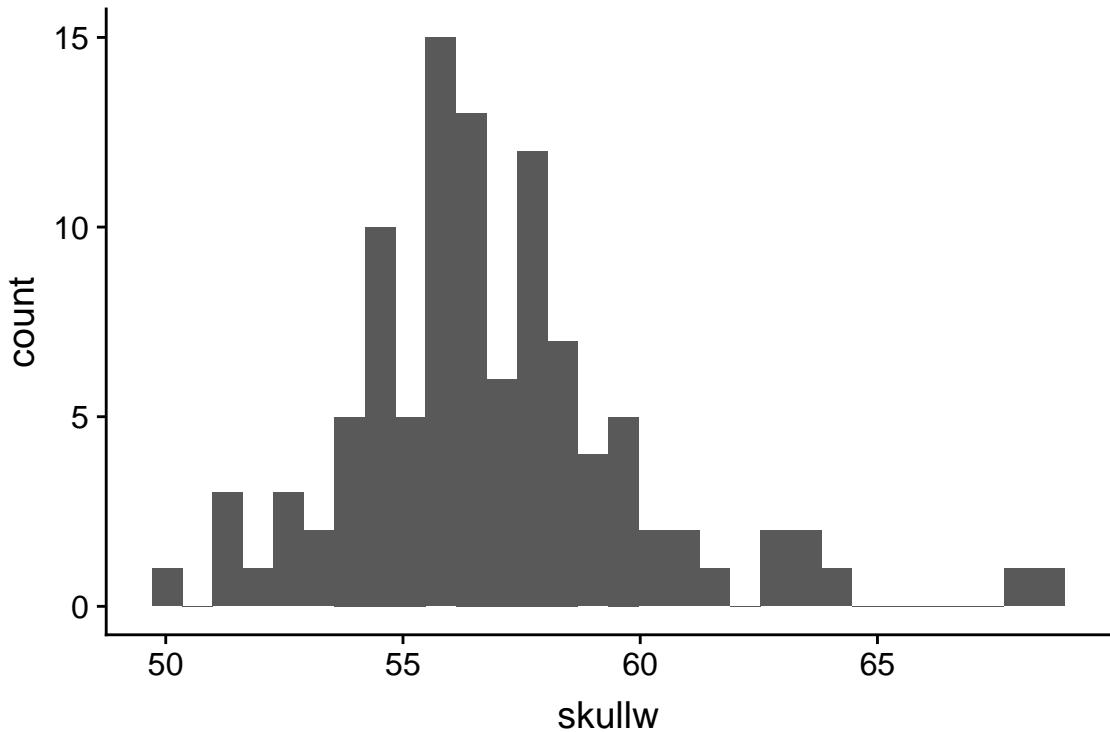


### 3.17 Histograms

A histogram is a special type of bar plot, typically used with continuous data. In a histogram, we divide the range of the data into bins of a given size, and use vertical bars to depict the frequency (count) of observations that fall into each bin. This gives a good sense of the intervals in which most of the observations are found.

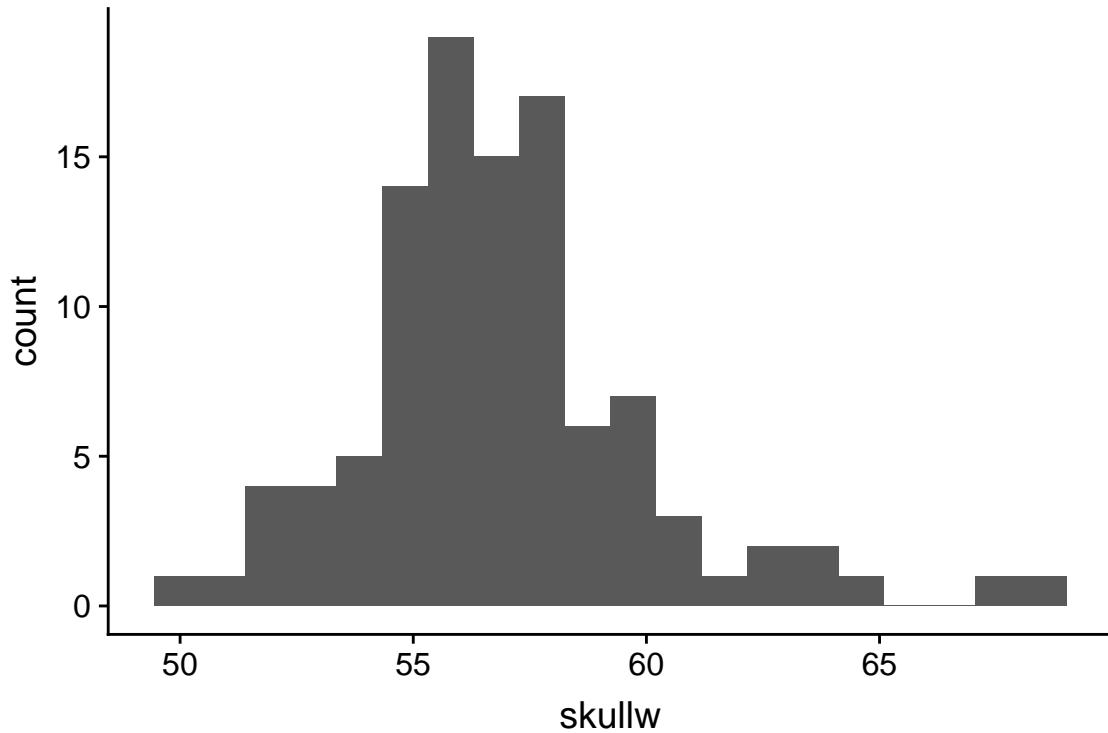
Here is a histogram for the skull width data:

```
ggplot(possums) + geom_histogram(aes(x = skullw))
#> `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Notice the warning message that was generated, about the default number of bins that were used. ggplot is alerting us that that we may want to consider regenerating the plot with a different number of bins. Let's try a smaller number of bins:

```
ggplot(possums) + geom_histogram(aes(x = skullw), bins = 20)
```

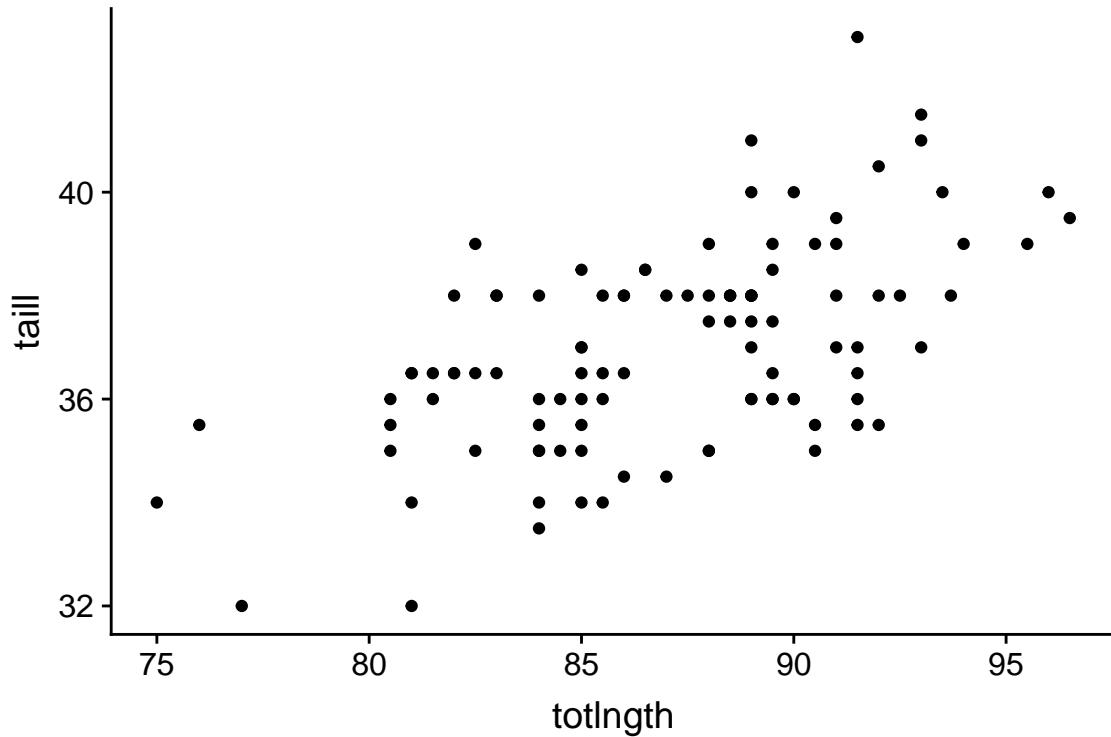


Try creating histograms for `taill` and `totlngth`.

### 3.18 Scatter plots

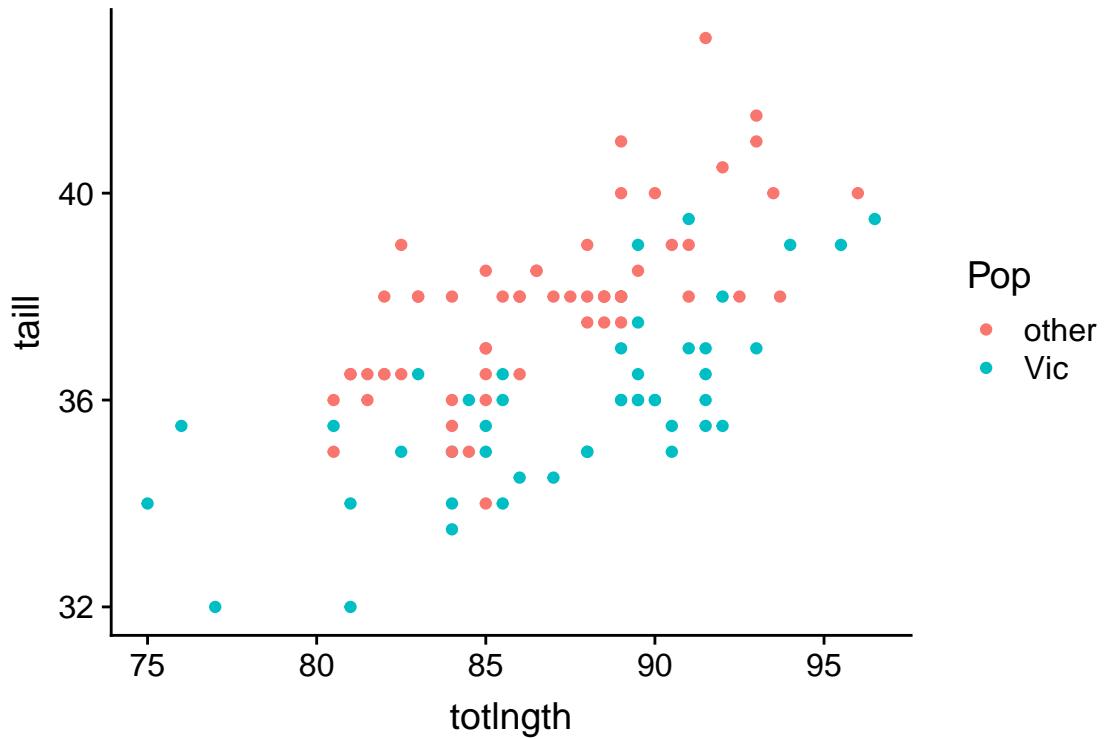
A scatter plot is typically used to represent two numerical variables simultaneously. Each point in a scatter plot is an individual in the data set, and the location of the points represent the measured values of the variables of interest on that individual.

```
ggplot(possums) + geom_point(aes(x = totlngth, y = taill))
```



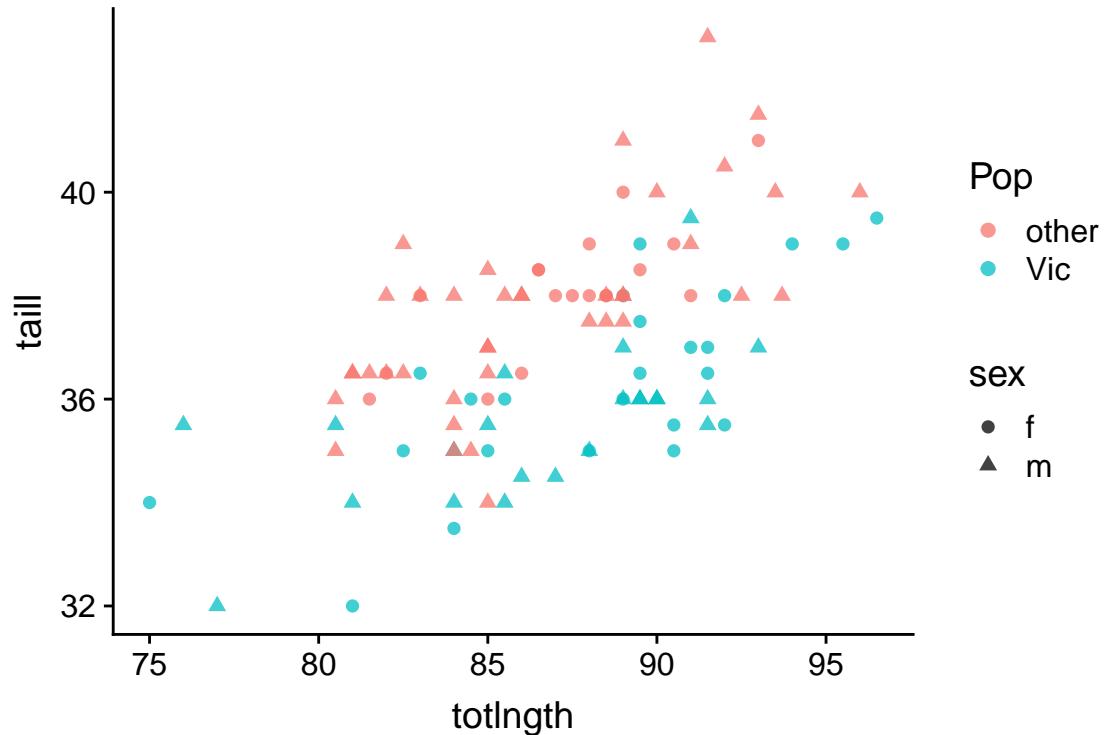
We can add information about categorical variables to our scatter plot by using color or shape to depict different classes

```
ggplot(possums) +
  geom_point(aes(x = totLngth, y = taill, color = Pop))
```



We can represent more than two categorical variables in our scatter plot by using both shape and color. We'll also change the size and transparency of the plotted points (via the `alpha` argument).

```
ggplot(possums) +
  geom_point(aes(x = totlngth, y = taill, color = Pop, shape = sex),
             size = 2, alpha = 0.75)
```



Explore some of the other bivariate relationships in the possums data by creating additional scatter plots. Given 9 numerical variables, how many distinct pairwise scatter plots could you create?



# Chapter 4

## R Markdown and R Notebooks

RStudio comes with a useful set of tools, collectively called R Markdown, for generating “literate” statistical analyses. The idea behind literate statistical computing is that we should try to carry out our analyses in a manner that is transparent, self-explanatory, and reproducible. Literate statistical computing helps to ensure your research is reproducible because:

1. The steps of your analyses are explicitly described, both as written text and the code and function calls used.
2. Analyses can be more easily checked for correctness and reproduced from your literate code.
3. Your literate code can serve as a template for future analyses, saving you time and the trouble of remembering all the gory details.

As we’ll see, R Markdown will allow us to produce statistical documents that integrate prose, code, figures, and nicely formatted mathematics so that we can share and explain our analyses to others. Sometimes those “others” are advisors, supervisors, or collaborators; sometimes the “other” is you six months from now. For the purposes of this class, you will be asked to complete problem sets in the form of R Markdown documents.

R Markdown documents are written in a light-weight markup language called Markdown. Markdown provides simple plain text “formatting” commands for specifying the structured elements of a document. Markdown was invented as a lightweight markup language for creating web pages and blogs, and has been adopted to a variety of different purposes. This chapter provides a brief introduction to the capabilities of R Markdown. For more complete details, including lots of examples, see the [R Markdown Website](#).

### 4.1 R Notebooks

We’re going to create a type of R Markdown document called an “R Notebook”. The R Notebook Documentation describes R Notebooks as so: “An R Notebook is an R Markdown document with code chunks that can be executed independently and interactively, with output visible immediately beneath the input.”

### 4.2 Creating an R Notebook

To create an R Notebook select `File > New File > R Notebook` from the files menu in RStudio.

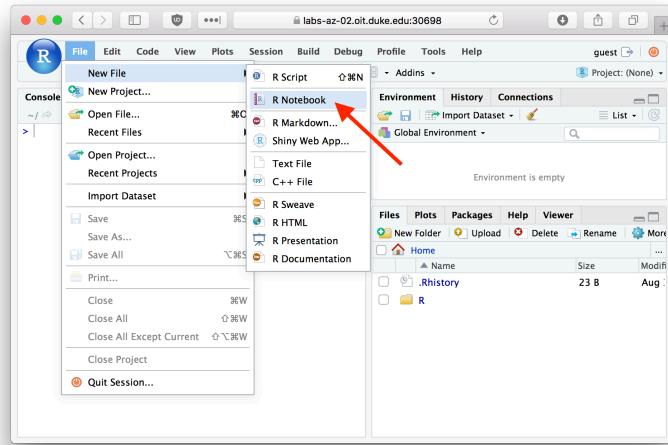


Figure 4.1: Using the File menu to create a new R Notebook.

### 4.3 The default R Notebook template

The standard template that RStudio creates for you includes a header section like the following where you can specify document properties such as the title, author, and change the look and feel of the generated HTML document.

```
---
title: "R Notebook"
output: html_notebook
---
```

The header is followed by several example sections that illustrate a few of the capabilities of R Markdown. Delete these and replace them with your own code as necessary.

### 4.4 Code and Non-code blocks

R Markdown documents are divided into code blocks (also called “chunks”) and non-code blocks. Code blocks are sets of R commands that will be evaluated when the R Markdown document is run or “knitted” (see below). Non-code blocks include explanatory text, embedded images, etc. The default notebook template includes both code and non-code blocks.

#### 4.4.1 Non-code blocks

The first bit of text in the default notebook template is a non-code block that tells you how to use the notebook:

```
This is an [R Markdown] (http://rmarkdown.rstudio.com) Notebook.  
When you execute code within the notebook, the results appear  
beneath the code.
```

Try executing this chunk by clicking the \*Run\* button within the chunk or by placing your cursor inside it and pressing \*Cmd+Shift+Enter\*.

The text of non-code blocks can include lightweight markup information that can be used to format HTML or PDF output generated from the R Markdown document. Here are some examples:

```
# Simple textual formatting
```

This is a paragraph with plain text. Nothing fancy will happen here.

This is a second paragraph with *\*italic\**, **\*\*bold\*\***, and `verbatim` text.

```
# Lists
```

```
## Bullet points lists
```

This is a list with bullet points:

- \* Item a
- \* Item b
- \* Item c

```
## Numbered lists
```

This is a numbered list:

1. Item 1
- #. Item 2
- #. Item 3

```
## Mathematics
```

R Markdown supports mathematical equations, formatted according to LaTeX conventions. Dollar signs (\$) are used to offset mathematics like so:  $x^2 + y^2 = z^2$ .

Notice from the example above that R Markdown supports LaTeX style formatting of mathematical equations. For example,  $x^2 + y^2 = z^2$  appears as  $x^2 + y^2 = z^2$ .

#### 4.4.2 Code blocks

Code blocks are delimited by matching sets of three backward ticks (``). Everything within a code block is interpreted as an R command and is evaluated by the R interpreter. Here's the first code block in the default notebook template:

```
```{r}
plot(cars)
```
```

## 4.5 Running a code chunk

You can run a single code block by clicking the small green “Run” button in the upper right hand corner of the code block as shown in the image below.

If you click this button the commands within this code block are executed, and any generated output is shown below the code block.

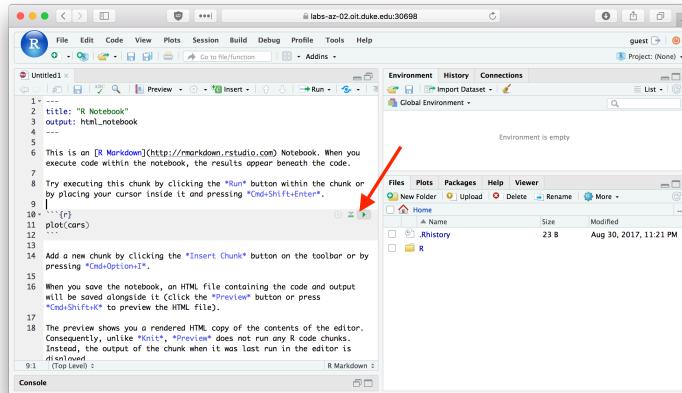


Figure 4.2: Click the Run button to execute a code chunk.

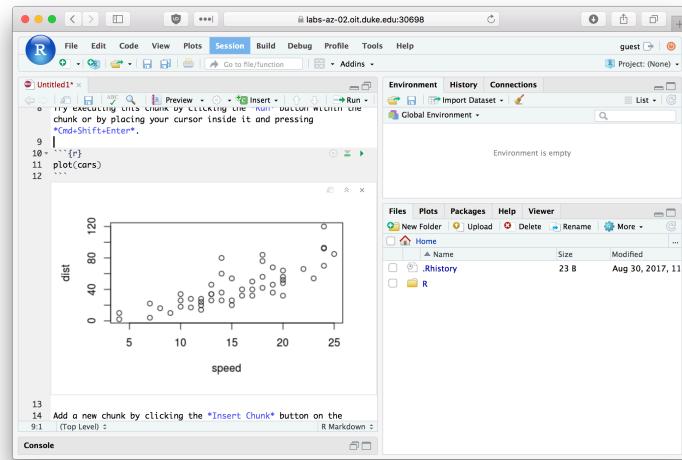


Figure 4.3: An R Notebook showing an embedded plot after executing a code chunk.

Try running the first code block in the default template now. After the code chunk is executed you should see a plot embedded in your R Notebook as shown below:

## 4.6 Running all code chunks above

Next to the “Run” button in each code chunk is a button for “Run all chunks above” (see figure below). This is useful when the code chunk you’re working on depends on calculations in earlier code chunks, and you want to evaluate those earlier code chunks prior to running the focal code chunk.

## 4.7 “Knitting an” R Markdown to HTML

Save your R Notebook as `first_rnotebook.Rmd` (RStudio will automatically add the `.Rmd` extension so you don’t need to type it). You can generate an HTML version of your notebook by clicking the “Preview” menu

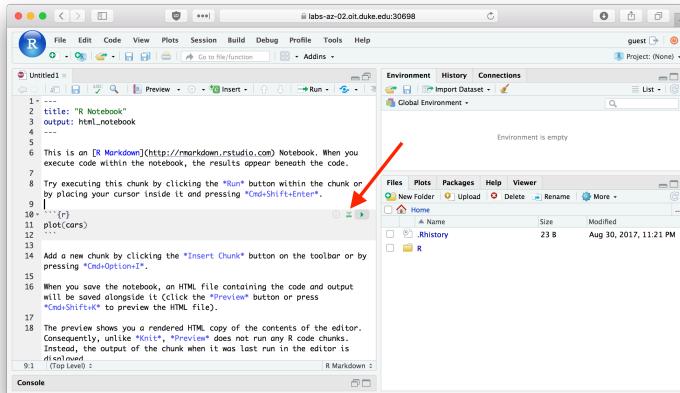


Figure 4.4: Use the 'Run all chunks above' button to evaluate all previous code chunks.

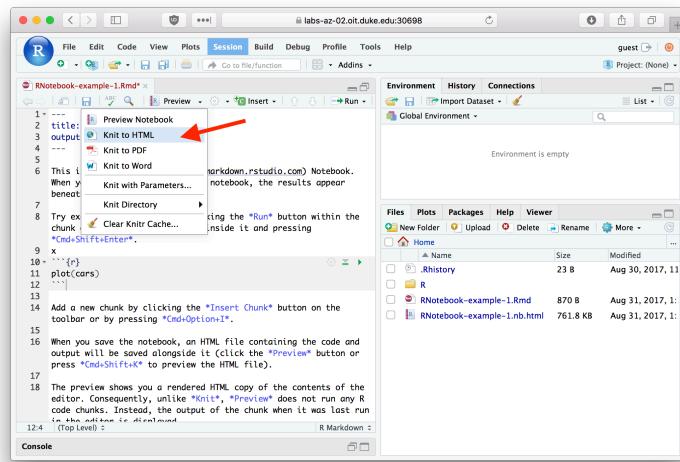


Figure 4.5: Use the 'Knit to HTML' menu to generate HTML output from your R Notebook

on the Notebook taskbar and then choosing “Knit to HTML” (see image below).

When an RMarkdown document is “knit”, all of the code and non-code blocks are executed in a “clean” environment, in order from top to bottom. An output file is generated (HTML or one of the other available output types) that shows the results of executing the notebook. By default RStudio will pop-up a window showing you the HTML output you generated.

Knitting a document is a good way to make sure your analysis is reproducible. If your code compiles correctly when the document is knit, and produces the expected output, there’s a good chance that someone else will be able to reproduce your analyses independently starting with your R Notebook document (after accounting for differences in file locations).

## 4.8 Sharing your reproducible R Notebook

To share your R Notebook with someone else you just need to send them the source R Markdown file (i.e. the file with the .Rmd extension). Assuming they have access to the same source data, another user should be

able to open the notebook file in RStudio and regenerate your analyses by evaluating the individual code chunks or knitting the document.

In this course you will be submitting homework assignments in the form of R Notebook markdown files.

# Chapter 5

## More R Basics: Data structures

In computer science, the term “data structure” refers to the ways that data are stored, retrieved, and organized in a computer’s memory. Common examples include lists, hash tables (also called dictionaries), sets, queues, and trees. Different types of data structures are used to support different types of operations on data.

In R, the three basic data structures are vectors, lists, and data frames.

### 5.1 Vectors

Vectors are the core data structure in R. Vectors store an ordered lists of items, *all of the same type* (i.e. the data in a vector are “homogenous” with respect to their type).

The simplest way to create a vector at the interactive prompt is to use the `c()` function, which is short hand for “combine” or “concatenate”.

```
x <- c(2,4,6,8) # create a vector, assignn it the variable name `x`  
x  
#> [1] 2 4 6 8
```

Vectors in R always have a type (accessed with the `typeof()` function) and a length (accessed with the `length()` function).

```
length(x)  
#> [1] 4  
typeof(x)  
#> [1] "double"
```

Vectors don’t have to be numerical; logical and character vectors work just as well.

```
y <- c(TRUE, TRUE, FALSE, TRUE, FALSE, FALSE)  
y  
#> [1] TRUE TRUE FALSE TRUE FALSE FALSE  
typeof(y)  
#> [1] "logical"  
length(y)  
#> [1] 6  
  
z <- c("How", "now", "brown", "cow")  
z
```

```
#> [1] "How"     "now"      "brown"   "cow"
typeof(z)
#> [1] "character"
length(z)
#> [1] 4
```

You can also use `c()` to concatenate two or more vectors together.

```
x <- c(2, 4, 6, 8)
y <- c(1, 3, 5, 7, 9) # create another vector, labeled y
xy <- c(x,y) # combine two vectors
xy
#> [1] 2 4 6 8 1 3 5 7 9

z <- c(pi/4, pi/2, pi, 2*pi)
xyz <- c(x, y, z) # combine three vectors
xyz
#> [1] 2.0000000 4.0000000 6.0000000 8.0000000 1.0000000 3.0000000 5.0000000
#> [8] 7.0000000 9.0000000 0.7853982 1.5707963 3.1415927 6.2831853
```

### 5.1.1 Vector Arithmetic

The basic R arithmetic operations work on numeric vectors as well as on single numbers (in fact, behind the scenes in R single numbers *are* vectors!).

```
x <- c(2, 4, 6, 8, 10)
x * 2 # multiply each element of x by 2
#> [1] 4 8 12 16 20
x - pi # subtract pi from each element of x
#> [1] -1.1415927 0.8584073 2.8584073 4.8584073 6.8584073

y <- c(0, 1, 3, 5, 9)
x + y # add together each matching element of x and y
#> [1] 2 5 9 13 19
x * y # multiply each matching element of x and y
#> [1] 0 4 18 40 90
x/y # divide each matching element of x and y
#> [1] Inf 4.000000 2.000000 1.600000 1.111111
```

Basic numerical functions operate element-wise on numerical vectors:

```
sin(x)
#> [1] 0.9092974 -0.7568025 -0.2794155 0.9893582 -0.5440211
cos(x * pi)
#> [1] 1 1 1 1 1
log(x)
#> [1] 0.6931472 1.3862944 1.7917595 2.0794415 2.3025851
```

### 5.1.2 Vector recycling

When vectors are not of the same length R ‘recycles’ the elements of the shorter vector to make the lengths conform.

```
x <- c(2, 4, 6, 8, 10)
length(x)
#> [1] 5
z <- c(1, 4, 7, 11)
length(z)
#> [1] 4
x + z
#> [1]  3  8 13 19 11
```

In the example above `z` was treated as if it was the vector `(1, 4, 7, 11, 1)`.

### 5.1.3 Simple statistical functions for numeric vectors

Now that we've introduced vectors as the simplest data structure for holding collections of numerical values, we can introduce a few of the most common statistical functions that operate on such vectors.

First let's create a vector to hold our sample data of interest. Here I've taken a random sample of the lengths of the last names of students enrolled in Bio 723 during Spring 2018.

```
len.name <- c(7, 7, 6, 2, 9, 9, 7, 4, 10, 5)
```

Some common statistics of interest include minimum, maximum, mean, median, variance, and standard deviation:

```
sum(len.name)
#> [1] 66
min(len.name)
#> [1] 2
max(len.name)
#> [1] 10
mean(len.name)
#> [1] 6.6
median(len.name)
#> [1] 7
var(len.name) # variance
#> [1] 6.044444
sd(len.name) # standard deviation
#> [1] 2.458545
```

The `summary()` function applied to a vector of doubles produce a useful table of some of these key statistics:

```
summary(len.name)
#>   Min. 1st Qu. Median   Mean 3rd Qu.   Max.
#> 2.00  5.25  7.00  6.60  8.50 10.00
```

### 5.1.4 Indexing Vectors

Accessing the element of a vector is called “indexing”. Indexing is the process of specifying the numerical positions (indices) that you want to take access from the vector.

For a vector of length  $n$ , we can access the elements by the indices  $1 \dots n$ . We say that R vectors (and other data structures like lists) are ‘one-indexed’. Many other programming languages, such as Python, C, and Java, use zero-indexing where the elements of a data structure are accessed by the indices  $0 \dots n - 1$ . Indexing errors are a common source of bugs.

Indexing a vector is done by specifying the index in square brackets as shown below:

```
x <- c(2, 4, 6, 8, 10)
length(x)
#> [1] 5

x[1] # return the 1st element of x
#> [1] 2

x[4] # return the 4th element of x
#> [1] 8
```

Negative indices are used to exclude particular elements. `x[-1]` returns all elements of `x` except the first.

```
x[-1]
#> [1] 4 6 8 10
```

You can get multiple elements of a vector by indexing by another vector. In the example below, `x[c(3,5)]` returns the third and fifth element of `x`:

```
x[c(3,5)]
#> [1] 6 10
```

### 5.1.5 Comparison operators applied to vectors

When the comparison operators, such as “greater than” (`>`), “less than or equal to” (`<=`), equality (`==`), etc. are applied to numeric vectors, they return logical vectors:

```
x <- c(2, 4, 6, 8, 10, 12)
x < 8 # returns TRUE for all elements less than 8
#> [1] TRUE TRUE TRUE FALSE FALSE FALSE
```

Here’s a fancier example:

```
x > 4 & x < 10 # greater than 4 AND less than 10
#> [1] FALSE FALSE TRUE TRUE FALSE FALSE
```

### 5.1.6 Combining Indexing and Comparison of Vectors

A very powerful feature of R is the ability to combine the comparison operators (which return TRUE or FALSE values) with indexing. This facilitates data filtering and subsetting.

Here’s an example:

```
x <- c(2, 4, 6, 8, 10)
x[x > 5]
#> [1] 6 8 10
```

In the first example we retrieved all the elements of `x` that are larger than 5 (read as “`x` where `x` is greater than 5”). Notice how we got back all the elements where the statement in the brackets was TRUE.

You can string together comparisons for more complex filtering.

```
x[x < 4 | x > 8] # less than four OR greater than 8
#> [1] 2 10
```

In the second example we retrieved those elements of `x` that were smaller than four *or* greater than six. Combining indexing and comparison is a powerful concept which we’ll use repeatedly in this course.

### 5.1.7 Vector manipulation

You can combine indexing with assignment to change the elements of a vectors:

```
x <- c(2, 4, 6, 8, 10)
x[2] <- -4
x
#> [1] 2 -4 6 8 10
```

You can also use indexing vectors to change multiple values at once:

```
x <- c(2, 4, 6, 8, 10)
x[c(1, 3, 5)] <- 6
x
#> [1] 6 4 6 8 6
```

Using logical vectors to manipulate the elements of a vector also works:

```
x <- c(2, 4, 6, 8, 10)
x[x > 5] = 5 # truncate all values to have max value 5
x
#> [1] 2 4 5 5 5
```

### 5.1.8 Vectors from regular sequences

There are a variety of functions for creating regular sequences in the form of vectors.

```
1:10 # create a vector with the integer values from 1 to 10
#> [1] 1 2 3 4 5 6 7 8 9 10
20:11 # a vector with the integer values from 20 to 11
#> [1] 20 19 18 17 16 15 14 13 12 11

seq(1, 10) # like 1:10
#> [1] 1 2 3 4 5 6 7 8 9 10
seq(1, 10, by = 2) # 1:10, in steps of 2
#> [1] 1 3 5 7 9
seq(2, 4, by = 0.25) # 2 to 4, in steps of 0.25
#> [1] 2.00 2.25 2.50 2.75 3.00 3.25 3.50 3.75 4.00
```

### 5.1.9 Additional functions for working with vectors

The function `unique()` returns the unique items in a vector:

```
x <- c(5, 2, 1, 4, 6, 9, 8, 5, 7, 9)
unique(x)
#> [1] 5 2 1 4 6 9 8 7
```

`rev()` returns the items in reverse order (without changing the input vector):

```
y <- rev(x)
y
#> [1] 9 7 5 8 9 6 4 1 2 5
x # x is still in original order
#> [1] 5 2 1 4 6 9 8 5 7 9
```

There are a number of useful functions related to sorting. Plain `sort()` returns a new vector with the items in sorted order:

```
sorted.x <- sort(x) # returns items of x sorted
sorted.x
#> [1] 1 2 4 5 5 6 7 8 9 9

x      # but x remains in its unsorted state
#> [1] 5 2 1 4 6 9 8 5 7 9
```

The related function `order()` gives the indices which would rearrange the items into sorted order:

```
order(x)
#> [1] 3 2 4 1 8 5 9 7 6 10
```

`order()` can be useful when you want to sort one list by the values of another:

```
students <- c("fred", "tabitha", "beatriz", "jose")
class.rank <- c(4, 2, 1, 3)

students[order(class.rank)] # get the students sorted by their class.rank
#> [1] "beatriz" "tabitha" "jose"     "fred"
```

`any()` and `all()`, return single boolean values based on a specified comparison provided as an argument:

```
y <- c(2, 4, 5, 6, 8)

any(y > 5) # returns TRUE if any of the elements are TRUE
#> [1] TRUE

all(y > 5) # returns TRUE if all of the elements are TRUE
#> [1] FALSE
```

`which()` returns the *indices* of the vector for which the input is true:

```
which(y > 5)
#> [1] 4 5
```

## 5.2 Lists

R lists are like vectors, but unlike a vector where all the elements are of the same type, the elements of a list can have arbitrary types (even other lists). Lists are a powerful data structure for organizing information, because there are few constraints on the shape or types of the data included in a list.

Lists are easy to create:

```
l <- list('Bob', pi, 10)
```

Note that lists can contain arbitrary data. Lists can even contain other lists:

```
l <- list('Bob', pi, 10, list("foo", "bar", "baz", "qux"))
```

Lists are displayed with a particular format, distinct from vectors:

```
l
#> [[1]]
#> [1] "Bob"
#>
```

```
#> [[2]]
#> [1] 3.141593
#>
#> [[3]]
#> [1] 10
#>
#> [[4]]
#> [[4]][[1]]
#> [1] "foo"
#>
#> [[4]][[2]]
#> [1] "bar"
#>
#> [[4]][[3]]
#> [1] "baz"
#>
#> [[4]][[4]]
#> [1] "qux"
```

In the example above, the correspondence between the list and its display is obvious for the first three items. The fourth element may be a little confusing at first. Remember that the fourth item of `1` was another list. So what's being shown in the output for the fourth item is the nested list.

An alternative way to display a list is using the `str()` function (short for “structure”). `str()` provides a more compact representation that also tells us what type of data each element is:

```
str(1)
#> List of 4
#> $ : chr "Bob"
#> $ : num 3.14
#> $ : num 10
#> $ :List of 4
#> ...$ : chr "foo"
#> ...$ : chr "bar"
#> ...$ : chr "baz"
#> ...$ : chr "qux"
```

### 5.2.1 Length and type of lists

Like vectors, lists have length:

```
length(1)
#> [1] 4
```

But the type of a list is simply “list”, not the type of the items within the list. This makes sense because lists are allowed to be heterogeneous (i.e. hold data of different types).

```
typeof(1)
#> [1] "list"
```

### 5.2.2 Indexing lists

Lists have two indexing operators. Indexing a list with single brackets, like we did with vectors, returns a new list containing the element at index  $i$ . Lists also support double bracket indexing (`x[[i]]`) which returns

the *bare* element at index  $i$  (i.e. the element without the enclosing list). This is a subtle but important point so make sure you understand the difference between these two forms of indexing.

### 5.2.2.1 Single bracket list indexing

First, let's demonstrate single bracket indexing of the lists `l` we created above.

```
l[1]           # single brackets, returns list('Bob')
#> [[1]]
#> [1] "Bob"
typeof(l[1])  # notice the list type
#> [1] "list"
```

When using single brackets, lists support indexing with ranges and numeric vectors:

```
l[3:4]
#> [[1]]
#> [1] 10
#>
#> [[2]]
#> [[2]][[1]]
#> [1] "foo"
#>
#> [[2]][[2]]
#> [1] "bar"
#>
#> [[2]][[3]]
#> [1] "baz"
#>
#> [[2]][[4]]
#> [1] "qux"
l[c(1, 3, 5)]
#> [[1]]
#> [1] "Bob"
#>
#> [[2]]
#> [1] 10
#>
#> [[3]]
#> NULL
```

### 5.2.2.2 Double bracket list indexing

If double bracket indexing is used, the object at the given index in a list is returned:

```
l[[1]]        # double brackets, return plain 'Bob'
#> [1] "Bob"
typeof(l[[1]]) # notice the 'character' type
#> [1] "character"
```

Double bracket indexing does not support multiple indices, but you can chain together double bracket operators to pull out the items of sublists. For example:

```
# second item of the fourth item of the list
l[[4]][[2]]
#> [1] "bar"
```

### 5.2.3 Naming list elements

The elements of a list can be given names when the list is created:

```
p <- list(first.name='Alice', last.name="Qux", age=27, years.in.school=10)
```

You can retrieve the names associated with a list using the `names` function:

```
names(p)
#> [1] "first.name"           "last.name"          "age"                "years.in.school"
```

If a list has named elements, you can retrieve the corresponding elements by indexing with the quoted name in either single or double brackets. Consistent with previous usage, single brackets return a list with the corresponding named element, whereas double brackets return the bare element.

For example, make sure you understand the difference in the output generated by these two indexing calls:

```
p["first.name"]
#> $first.name
#> [1] "Alice"

p[["first.name"]]
#> [1] "Alice"
```

### 5.2.4 The \$ operator

Retrieving named elements of lists (and data frames as we'll see), turns out to be a pretty common task (especially when doing interactive data analysis) so R has a special operator to make this more convenient. This is the `$` operator, which is used as illustrated below:

```
p$first.name # equivalent to p[["first.name"]]
#> [1] "Alice"
p$age         # equivalent to p[["age"]]
#> [1] 27
```

### 5.2.5 Changing and adding lists items

Combining indexing and assignment allows you to change items in a list:

```
suspect <- list(first.name = "unknown",
                  last.name = "unknown",
                  aka = "little")

suspect$first.name <- "Bo"
suspect$last.name <- "Peep"
suspect[[3]] <- "LITTLE"

str(suspect)
#> List of 3
#> $ first.name: chr "Bo"
```

```
#> $ last.name : chr "Peep"
#> $ aka       : chr "LITTLE"
```

By combining assignment with a new name or an index past the end of the list you can add items to a list:

```
suspect$age <- 17 # add a new item named age
suspect[[5]] <- "shepardess" # create an unnamed item at position 5
```

Be careful when adding an item using indexing, because if you skip an index an intervening NULL value is created:

```
# there are only five items in the list, what happens if we
# add a new item at position seven?
suspect[[7]] <- "wanted for sheep stealing"

str(suspect)
#> List of 7
#> $ first.name: chr "Bo"
#> $ last.name : chr "Peep"
#> $ aka       : chr "LITTLE"
#> $ age       : num 17
#> $           : chr "shepardess"
#> $           : NULL
#> $           : chr "wanted for sheep stealing"
```

### 5.2.6 Combining lists

The `c` (combine) function we introduced to create vectors can also be used to combine lists:

```
list.a <- list("little", "bo", "peep")
list.b <- list("has lost", "her", "sheep")
list.c <- c(list.a, list.b)
list.c
#> [[1]]
#> [1] "little"
#>
#> [[2]]
#> [1] "bo"
#>
#> [[3]]
#> [1] "peep"
#>
#> [[4]]
#> [1] "has lost"
#>
#> [[5]]
#> [1] "her"
#>
#> [[6]]
#> [1] "sheep"
```

### 5.2.7 Converting lists to vectors

Sometimes it's useful to convert a list to a vector. The `unlist()` function takes care of this for us.

```
# a homogeneous list
ex1 <- list(2, 4, 6, 8)
unlist(ex1)
#> [1] 2 4 6 8
```

When you convert a list to a vector make sure you remember that vectors are homogeneous, so items within the new vector will be “coerced” to have the same type.

```
# a heterogeneous list
ex2 <- list(2, 4, 6, c("bob", "fred"), list(1 + 0i, 'foo'))
unlist(ex2)
#> [1] "2"     "4"     "6"     "bob"   "fred"  "1+0i" "foo"
```

Note that `unlist()` also unpacks nested vectors and lists as shown in the second example above.

## 5.3 Data frames

Along with vectors and lists, data frames are one of the core data structures when working in R. A data frame is essentially a list which represents a data table, where each column in the table has the same number of rows and every item in the a column has to be of the same type. Unlike standard lists, the objects (columns) in a data frame must have names. We've seen data frames previously, for example when we loaded data sets using the `read_csv` function.

### 5.3.1 Creating a data frame

While data frames will often be created by reading in a data set from a file, they can also be created directly in the console as illustrated below:

```
age <- c(30, 26, 21, 29, 25, 22, 28, 24, 23, 20)
sex <- rep(c("M", "F"), 5)
wt.in.kg <- c(88, 76, 67, 66, 56, 74, 71, 60, 52, 72)

df <- data.frame(age = age, sex = sex, wt = wt.in.kg)
```

Here we created a data frame with three columns, each of length 10.

### 5.3.2 Type and class for data frames

Data frames can be thought of as specialized lists, and in fact the type of a data frame is “list” as illustrated below:

```
typeof(df)
#> [1] "list"
```

To distinguish a data frame from a generic list, we have to ask about its “class”.

```
class(df) # the class of our data frame
#> [1] "data.frame"
class(1) # compare to the class of our generic list
#> [1] "list"
```

The term “class” comes from a style/approach to programming called “object oriented programming”. We won’t go into explicit detail about how object oriented programming works in this class, though we will exploit many of the features of objects that have a particular class.

### 5.3.3 Length and dimension for data frames

Applying the `length()` function to a data frame returns the number of columns. This is consistent with the fact that data frames are specialized lists:

```
length(df)
#> [1] 3
```

To get the dimensions (number of rows and columns) of a data frame, we use the `dim()` function. `dim()` returns a vector, whose first value is the number of rows and whose second value is the number of columns:

```
dim(df)
#> [1] 10 3
```

We can get the number of rows and columns individually using the `nrow()` and `ncol()` functions:

```
nrow(df) # number of rows
#> [1] 10
ncol(df) # number of columns
#> [1] 3
```

### 5.3.4 Indexing and accessing data frames

Data frames can be indexed by either column index, column name, row number, or a combination of row and column numbers.

#### 5.3.4.1 Single bracket indexing of the columns of a data frame

The *single bracket operator with a single numeric index* returns a data frame with the corresponding column.

```
df[1] # get the first column (=age) of the data frame
#> age
#> 1 30
#> 2 26
#> 3 21
#> 4 29
#> 5 25
#> 6 22
#> 7 28
#> 8 24
#> 9 23
#> 10 20
```

The *single bracket operator with multiple numeric indices* returns a data frame with the corresponding columns.

```
df[1:2] # first two columns
#> age sex
#> 1 30 M
#> 2 26 F
#> 3 21 M
```

```
#> 4 29 F
#> 5 25 M
#> 6 22 F
#> 7 28 M
#> 8 24 F
#> 9 23 M
#> 10 20 F
df [c(1, 3)] # columns 1 (=age) and 3 (=wt)
#> age wt
#> 1 30 88
#> 2 26 76
#> 3 21 67
#> 4 29 66
#> 5 25 56
#> 6 22 74
#> 7 28 71
#> 8 24 60
#> 9 23 52
#> 10 20 72
```

Column names can be substituted for indices when using the single bracket operator:

```
df ["age"]
#> age
#> 1 30
#> 2 26
#> 3 21
#> 4 29
#> 5 25
#> 6 22
#> 7 28
#> 8 24
#> 9 23
#> 10 20

df [c("age", "wt")]
#> age wt
#> 1 30 88
#> 2 26 76
#> 3 21 67
#> 4 29 66
#> 5 25 56
#> 6 22 74
#> 7 28 71
#> 8 24 60
#> 9 23 52
#> 10 20 72
```

#### 5.3.4.2 Single bracket indexing of the rows of a data frame

To get specific rows of a data frame, we use single bracket indexing with an additional comma following the index. For example to get the first row a data frame we would do:

```
df[1,]    # first row
#>   age sex wt
#> 1 30 M 88
```

This syntax extends to multiple rows:

```
df[1:2,]  # first two rows
#>   age sex wt
#> 1 30 M 88
#> 2 26 F 76

df[c(1, 3, 5),] # rows 1, 3 and 5
#>   age sex wt
#> 1 30 M 88
#> 3 21 M 67
#> 5 25 M 56
```

### 5.3.4.3 Single bracket indexing of both the rows and columns of a data frame

Single bracket indexing of data frames extends naturally to retrieve both rows and columns simultaneously:

```
df[1, 2]  # first row, second column
#> [1] M
#> Levels: F M
df[1:3, 2:3] # first three rows, columns 2 and 3
#>   sex wt
#> 1  M 88
#> 2  F 76
#> 3  M 67

# you can even mix numerical indexing (rows) with named indexing of columns
df[5:10, c("age", "wt")]
#>   age wt
#> 5 25 56
#> 6 22 74
#> 7 28 71
#> 8 24 60
#> 9 23 52
#> 10 20 72
```

### 5.3.4.4 Double bracket and \$ indexing of data frames

Whereas single bracket indexing of a data frame always returns a new data frame, double bracket indexing and indexing using the \$ operator, returns vectors.

```
df[["age"]]
#> [1] 30 26 21 29 25 22 28 24 23 20
typeof(df[["age"]])
#> [1] "double"

df$wt
#> [1] 88 76 67 66 56 74 71 60 52 72
typeof(df$wt)
#> [1] "double"
```

### 5.3.5 Logical indexing of data frames

Logical indexing using boolean values works on data frames in much the same way it works on vectors. Typically, logical indexing of a data frame is used to filter the rows of a data frame.

For example, to get all the subject in our example data frame who are older than 25 we could do:

```
# NOTE: the comma after 25 is important to insure we're indexing rows!
df[df$age > 25, ]
#>   age sex wt
#> 1  30  M 88
#> 2  26  F 76
#> 4  29  F 66
#> 7  28  M 71
```

Similarly, to get all the individuals whose weight is between 60 and 70 kgs we could do:

```
df[(df$wt >= 60 & df$wt <= 70),]
#>   age sex wt
#> 3  21  M 67
#> 4  29  F 66
#> 8  24  F 60
```

### 5.3.6 Adding columns to a data frame

Adding columns to a data frame is similar to adding items to a list. The easiest way to do so is using named indexing. For example, to add a new column to our data frame that gives the individuals ages in number of days, we could do:

```
df[["age.in.days"]] <- df$age * 365
dim(df)
#> [1] 10  4
```



# Chapter 6

## Introduction to ggplot2

Pretty much any statistical plot can be thought of as a **mapping** between data and one or more visual representations. For example, in a scatter plot we map two ordered sets of numbers (the variables of interest) to points in the Cartesian plane (x,y-coordinates). The representation of data as points in a plane can be thought of as a type of **geometric mapping**. In a histogram, we divide the range of a variable of interest into bins, count the number of observations in each bin, and represent those counts as bars. The process of counting the data in bins is a type of **statistical transformation** (summing in this case), while the representation of the counts as bars is another example of a geometric mapping. Both types of plots can be further embellished with additional information, such as coloring the points or bars based on a categorical variable of interest, changing the shape of points, etc. These are examples of **aesthetic mappings**. An additional operation that is frequently useful is **faceting** (also called conditioning), in which a series of subplots are created to show particular subsets of the data.

The package `ggplot2` is based on a formalized approach for building statistical graphics as a combination of geometric mappings, aesthetic mappings, statistical transformations, and faceting (conditioning). In `ggplot2`, complex figures are built up by combining layers – where each layer includes a geometric mapping, an aesthetic mapping, and a statistical transformation – along with any desired faceting information.

Many of the key ideas behind `ggplot2` (and its predecessor, “`ggplot`”) are based on a book called “The Grammar of Graphics” (Leland Wilkinson, 1985). The “grammar of graphics” is the “gg” in the `ggplot2` name.

### 6.1 Loading ggplot2

`ggplot2` is one of the packages included in the `tidyverse` meta-package we installed during the previous class session (see the previous lecture notes for instruction if you have not installed `tidyverse`). If we load the `tidyverse` package, `ggplot2` is automatically loaded as well.

```
library(tidyverse)
```

However if we wanted to we could load only `ggplot2` as follows:

```
library(ggplot2) # not necessary if we already loaded tidyverse
```

## 6.2 Example data set: Anderson's Iris Data

To illustrate ggplot2 we'll use a dataset called `iris`. This data set was made famous by the statistician and geneticist R. A. Fisher who used it to illustrate many of the fundamental statistical methods he developed (Recall that Fisher was one of the key contributors to the modern synthesis in biology, reconciling evolution and genetics in the early 20th century). The data set consists of four morphometric measurements for specimens from three different iris species (*Iris setosa*, *I. versicolor*, and *I. virginica*). Use the R help to read about the iris data set (`?iris`). We'll be using this data set repeatedly in future weeks so familiarize yourself with it.

The iris data is included in a standard R package (`datasets`) that is made available automatically when you start up R. As a consequence we don't need to explicitly load the iris data from a file. Let's take a few minutes to explore this iris data set before we start generating plots:

```
names(iris) # get the variable names in the dataset
#> [1] "Sepal.Length" "Sepal.Width" "Petal.Length" "Petal.Width"
#> [5] "Species"
dim(iris) # dimensions given as rows, columns
#> [1] 150   5
head(iris) # can you figure out what the head function does?
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#> 1      5.1       3.5      1.4       0.2    setosa
#> 2      4.9       3.0      1.4       0.2    setosa
#> 3      4.7       3.2      1.3       0.2    setosa
#> 4      4.6       3.1      1.5       0.2    setosa
#> 5      5.0       3.6      1.4       0.2    setosa
#> 6      5.4       3.9      1.7       0.4    setosa
tail(iris) # what about the tail function?
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#> 145     6.7       3.3      5.7       2.5  virginica
#> 146     6.7       3.0      5.2       2.3  virginica
#> 147     6.3       2.5      5.0       1.9  virginica
#> 148     6.5       3.0      5.2       2.0  virginica
#> 149     6.2       3.4      5.4       2.3  virginica
#> 150     5.9       3.0      5.1       1.8  virginica
```

## 6.3 Template for single layer plots in ggplot2

A basic template for building a single layer plot using ggplot2 is shown below. When creating a plot, you need to replace the text in brackets (e.g. `<DATA>`) with appropriate objects, functions, or arguments:

```
# NOTE: this is pseudo-code. It will not run!
```

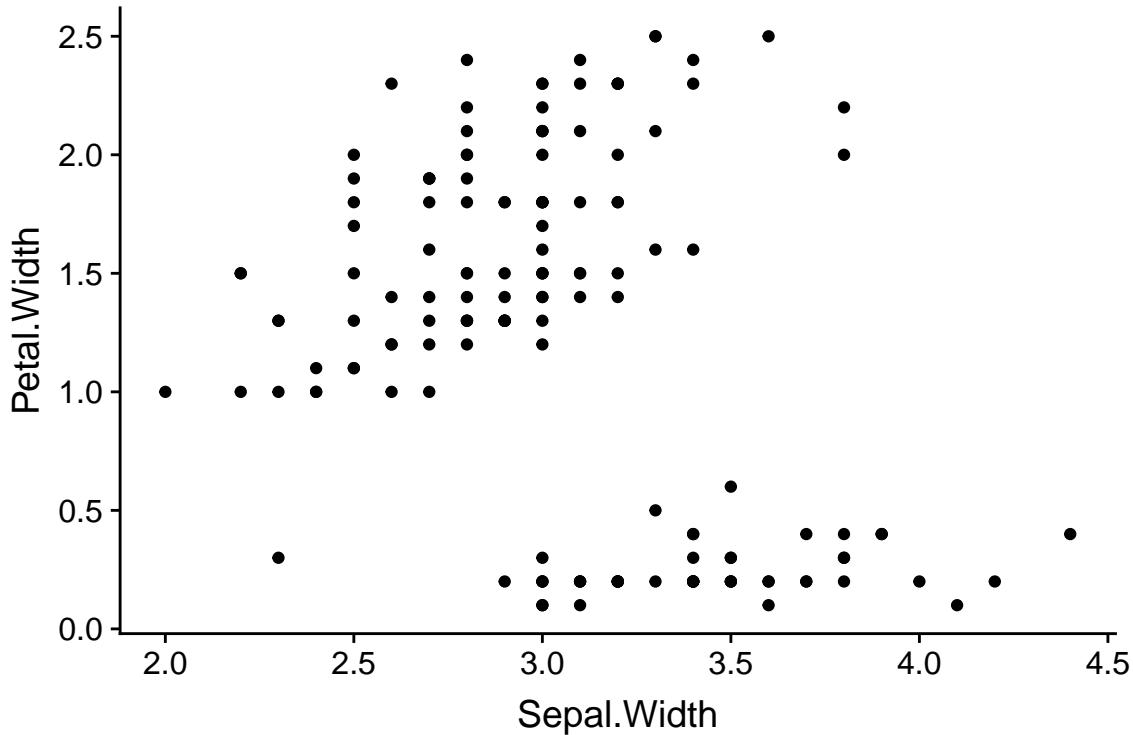
```
ggplot(data = <DATA>) +
  <GEOM_FUNCTION>(mapping = aes(<MAPPINGS>))
```

The base function `ggplot()` is responsible for creating the coordinate system in which the plot will be displayed. To this coordinate system we add a geometric mapping (called a “geom” for short) that specifies how data gets mapped into the coordinate system (e.g. points, bars, etc). Included as an input to the geom function is the aesthetic mapping function that specifies which variables to use in the geometric mapping (e.g. which variables to treat as the x- and y-coordinates), colors, etc.

For example, using this template we can create a scatter plot that shows the relationship between the variables `Sepal.Width` and `Petal.Width`. To do so we substitute `iris` for `<DATA>`, `geom_point` for `<GEOM_FUNCTION>`,

and `x = Sepal.Width` and `y = Petal.Width` for <MAPPINGS>.

```
ggplot(data = iris) +
  geom_point(mapping = aes(x = Sepal.Width, y = Petal.Width))
```



If we were to translate this code block to English, we might write it as “Using the iris data frame as the source of data, create a point plot using each observation’s Sepal.Width variable for the x-coordinate and the Petal.Width variable for the y-coordinate.”

## 6.4 An aside about function arguments

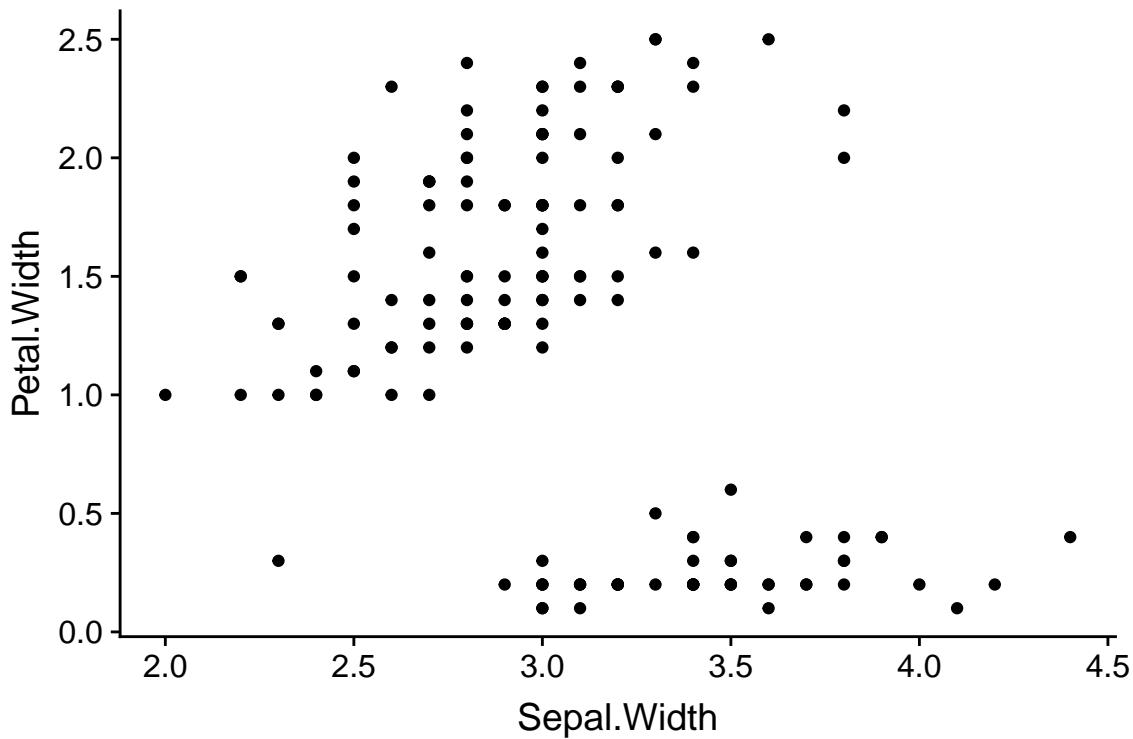
The inputs to a function are also known as “arguments”. In R, when you call a function you can specify the arguments by keyword (i.e. using names specified in the function definition) or by position (i.e. the order of the inputs).

In our bar plot above, we’re using keyword arguments. For example, in the line `ggplot(data = iris)`, `iris` is treated as the “data” argument. Similarly, in the second line, `aes(x = Sepal.Width, y = Petal.Width)` is the “mapping” argument to `geom_bar`. Note that `aes` is itself a function (see `?aes`) that takes arguments that can be specified positionally or with keywords.

If we wanted to, we could instead use position arguments when calling a function, by passing inputs to the function corresponding to the order they are specified in the function definition. For example, take a minute to read the documentation for the `ggplot` function (`?ggplot`). Near the top of the help page you’ll see a description of how the function is called under “Usage”. Reading the Usage section you’ll see that the “data” argument is the first positional argument to `ggplot`. Similarly, if you read the docs for the `geom_point` function you’ll see that `mapping` is the first positional argument for that function.

The equivalent of our previous example, but now using positional arguments is:

```
ggplot(iris) + # note we dropped the "data = " part
# note we dropped the "mapping = " part from the geom_point call
geom_point(aes(x = Sepal.Width, y = Petal.Width))
```



The upside of using positional arguments is that it means less typing, which is useful when working interactively at the console (or in an R Notebook). The downside to using positional arguments is you need to remember or lookup the order of the arguments. Using positional arguments can also make your code less “self documenting” in the sense that it is less explicit about how the inputs are being treated. While the argument “x” is the first argument to the `aes` function, I chose to explicitly include the argument name to make it clear what variable I’m plotting on the x-axis.

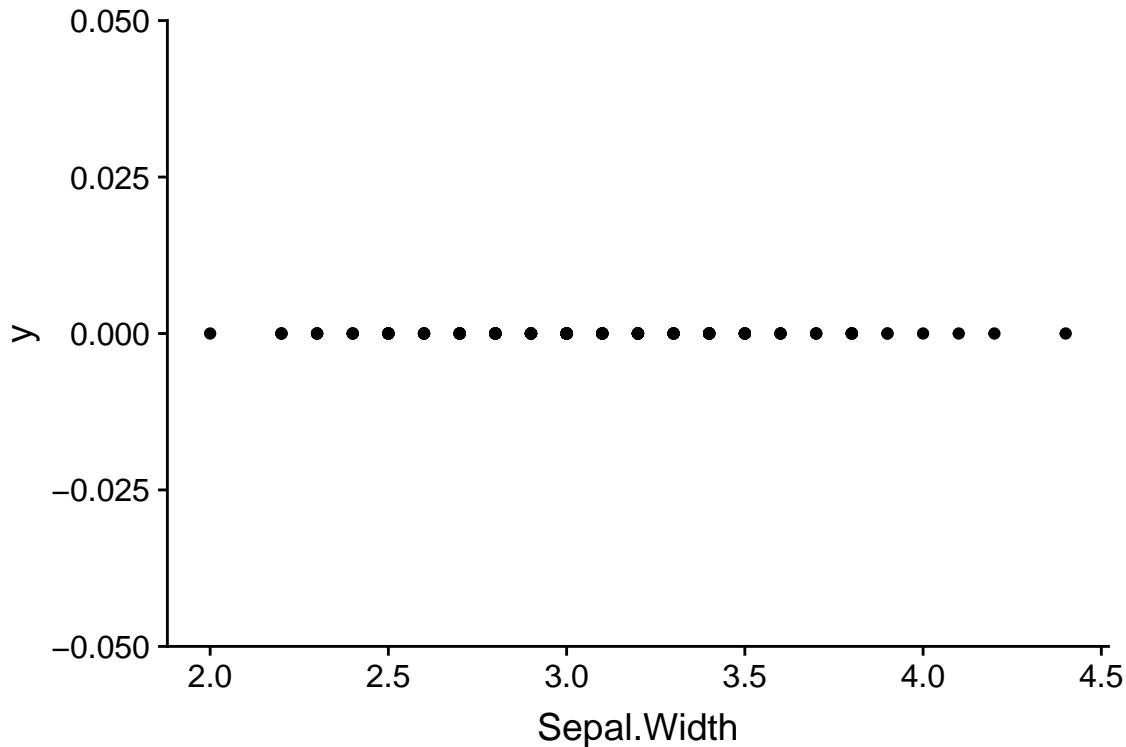
We will cover function arguments in greater detail a class session or two from now, when we learn how to write our own functions.

## 6.5 Strip plots

One of the simplest visualizations of a continuous variable is to draw points along a number line, where each point represent the value of one of the observations. This is sometimes called a “strip plot”.

First, we’ll use the `geom_point` function as shown below to generate a strip plot for the `Sepal.Width` variable in the iris data set.

```
ggplot(data = iris) +
  geom_point(aes(x = Sepal.Width, y = 0))
```



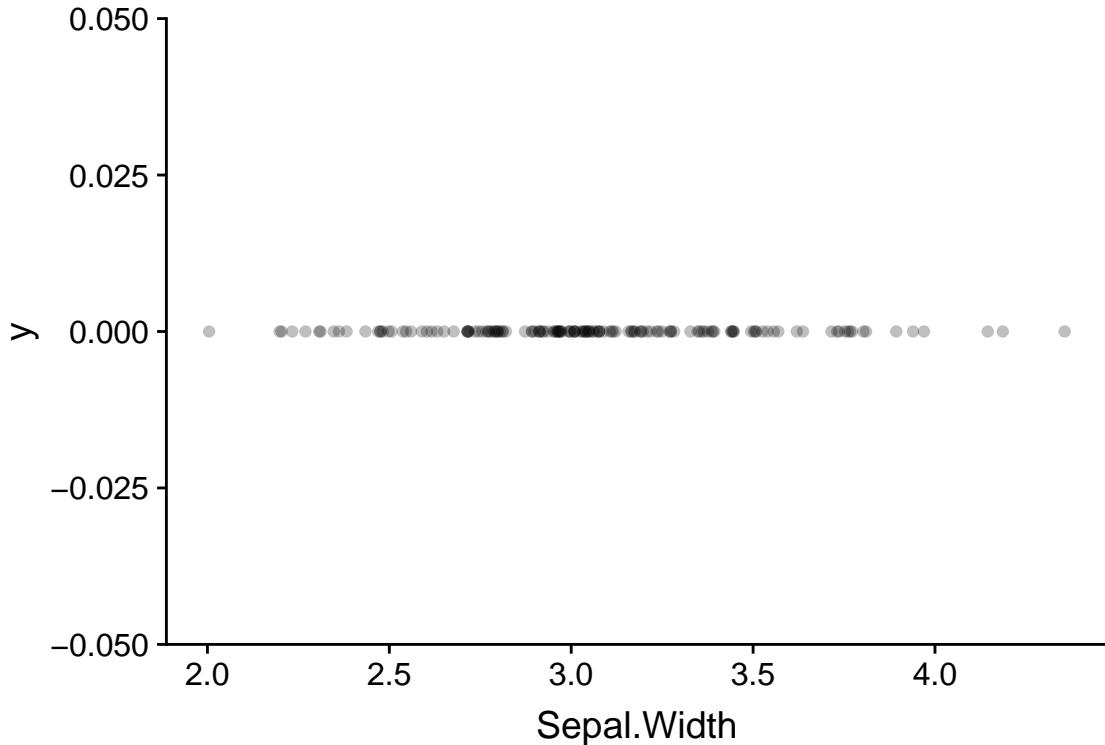
### 6.5.1 Jittering data

There should have been 150 points plotted in the figure above (one for each of the iris plants in the data set), but visually it looks like only about 25 or 30 points are shown. What's going on? If you examine the iris data, you'll see that all the measures are rounded to the nearest tenth of a centimeter, so that there are a large number of observations with identical values of Sepal.Width. This is a limitation of the precision of measurements that was used when generating the data set.

To provide a visual clue that there are multiple observations that share the same value, we can slightly "jitter" the values (randomly move points a small amount in either in the vertical or horizontal direction). Jittering is used solely to enhance visualization, and any statistical analyses you carry out would be based on the original data. When presenting your data to someone else, should note when you've used jittering so as not to misconvey the actual data.

Jittering can be accomplished using `geom_jitter`, which is derived from `geom_point`:

```
ggplot(data = iris) +
  geom_jitter(aes(x = Sepal.Width, y = 0),
              width = 0.05, height = 0, alpha = 0.25)
```



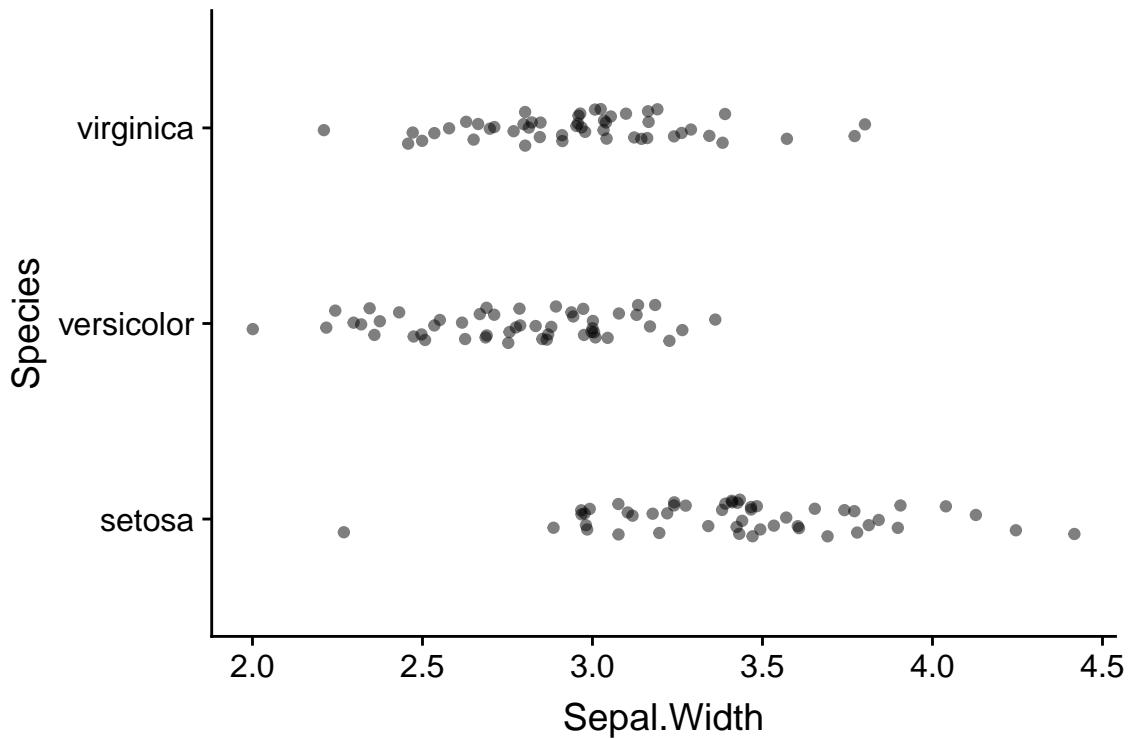
The `width` and `height` arguments specify the maximum amount (as fractions of the data) to jitter the observed data points in the horizontal (width) and vertical (height) directions. Here we only jitter the data in the horizontal direction. The `alpha` argument controls the transparency of the points – the valid range of alpha values is 0 to 1, where 0 means completely transparent and 1 is completely opaque.

Within a geom, arguments outside of the `aes` mapping apply uniformly across the visualization (i.e. they are fixed values). For example, setting ‘`alpha = 0.25`’ made all the points transparent.

### 6.5.2 Adding categorical information

Recall that there are three different species represented in the data: Iris setosa, I. versicolor, and I. virginica. Let’s see how to generate a strip plot that also includes a breakdown by species.

```
ggplot(data = iris) +
  geom_jitter(aes(x = Sepal.Width, y = Species),
              width=0.05, height=0.1, alpha=0.5)
```

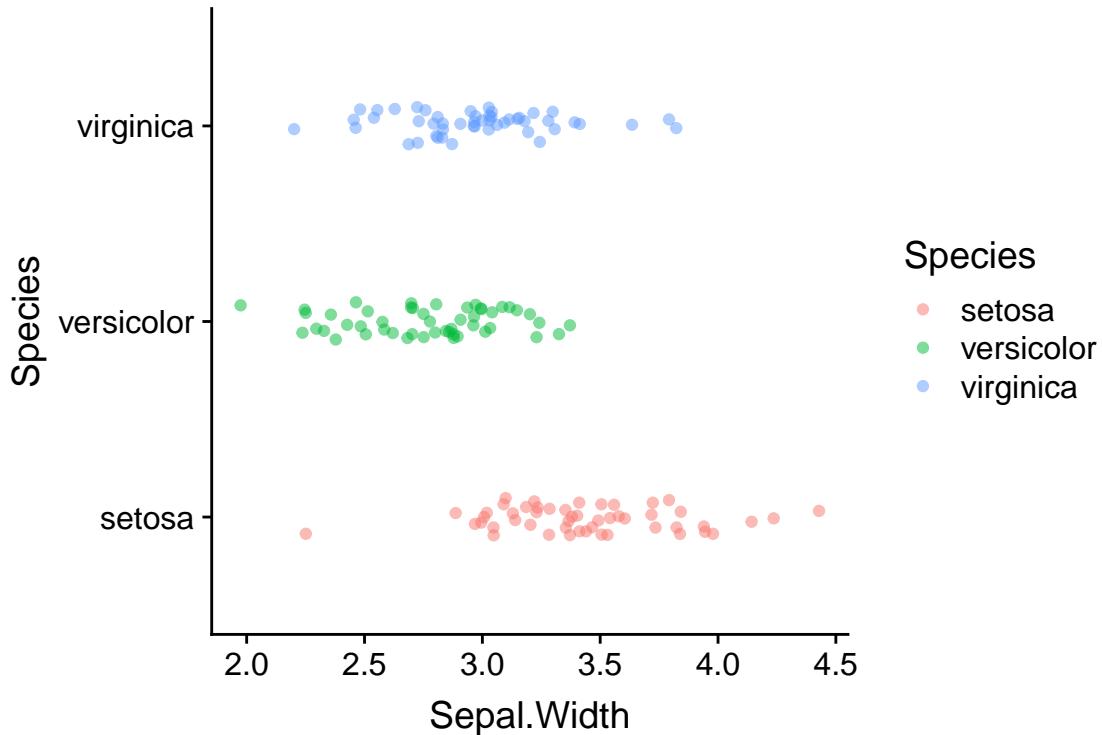


That was easy! All we had to do was change the aesthetic mapping in `geom_jitter`, specifying “Species” as the y variable. I also added a little vertical jitter as well to better separate the points.

Now we have a much better sense of the data. In particular it’s clear that the *I. setosa* specimens generally have wider sepals than samples from the other two species.

Let’s tweak this a little by also adding color information, to further emphasize the distinct groupings. We can do this by adding another argument to the aesthetic mapping in `geom_jitter`.

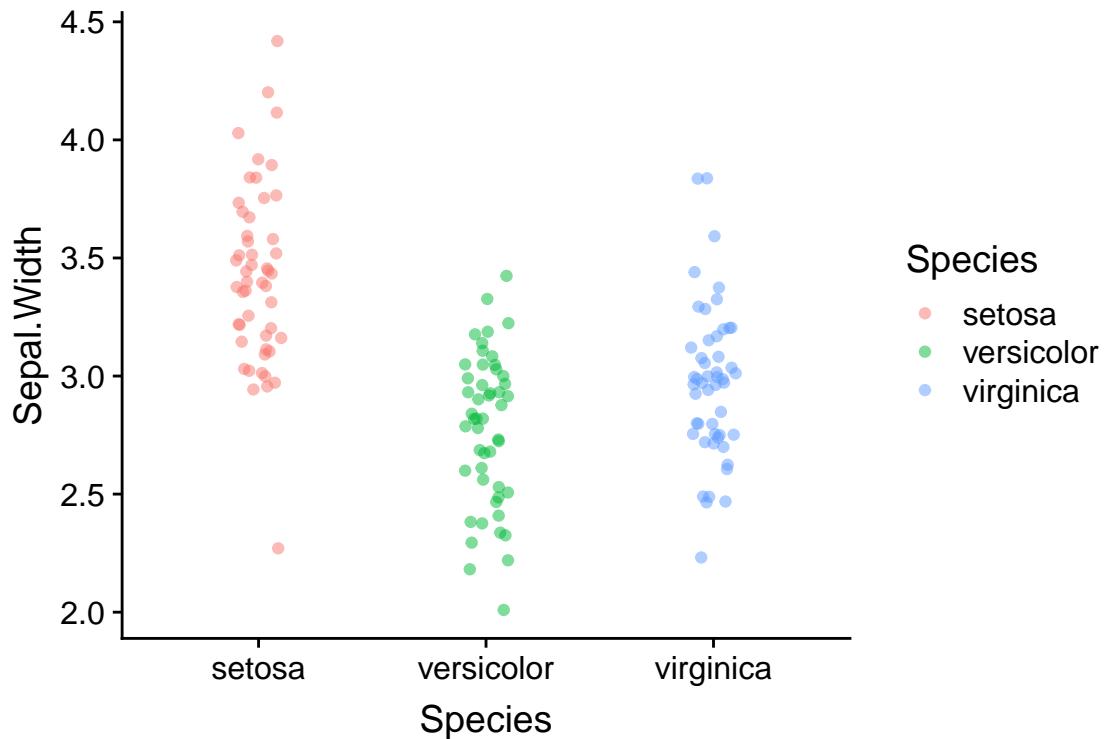
```
ggplot(data = iris) +
  geom_jitter(aes(x = Sepal.Width, y = Species, color=Species),
              width=0.05, height=0.1, alpha=0.5)
```



### 6.5.3 Rotating plot coordinates

What if we wanted to rotate this plot 90 degrees, depicting species on the x-axis and sepal width on the y-axis. For this example, it would be easy to do this by simply swapping the variables in the `aes` mapping argument. However an alternate way to do this is with a coordinate transformation function. Here we use `coord_flip` to flip the x- and y-axes:

```
ggplot(data = iris) +
  geom_jitter(aes(x = Sepal.Width, y = Species, color=Species),
              width=0.05, height=0.1, alpha=0.5) +
  coord_flip()
```



We'll see other uses of coordinate transformations in later lectures.

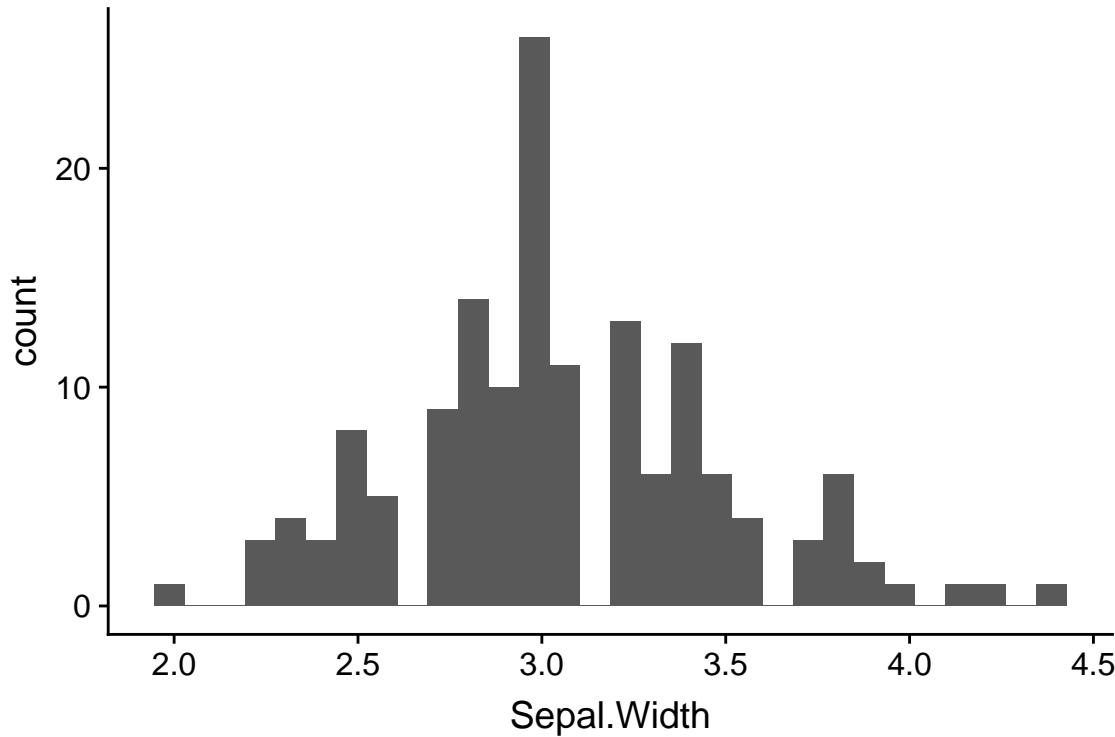
## 6.6 Histograms

Histograms are probably the most common way to depict univariate data. In a histogram rather than showing individual observations, we divide the range of the data into a set of bins, and use vertical bars to depict the number (frequency) of observations that fall into each bin. This gives a good sense of the intervals in which most of the observations are found.

The geom, `geom_histogram`, takes care of both the geometric representation and the statistical transformations of the data necessary to calculate the counts in each bin.

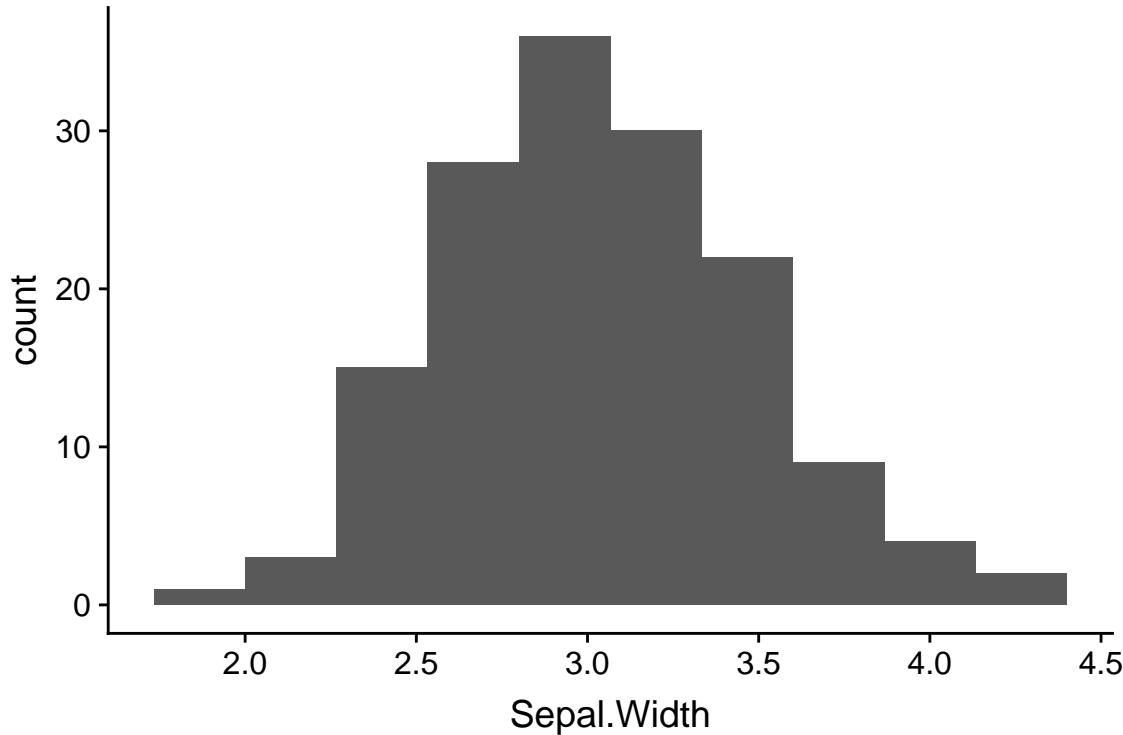
Here's the simplest way to use `geom_histogram`:

```
ggplot(iris) +
  geom_histogram(aes(x = Sepal.Width))
#> `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

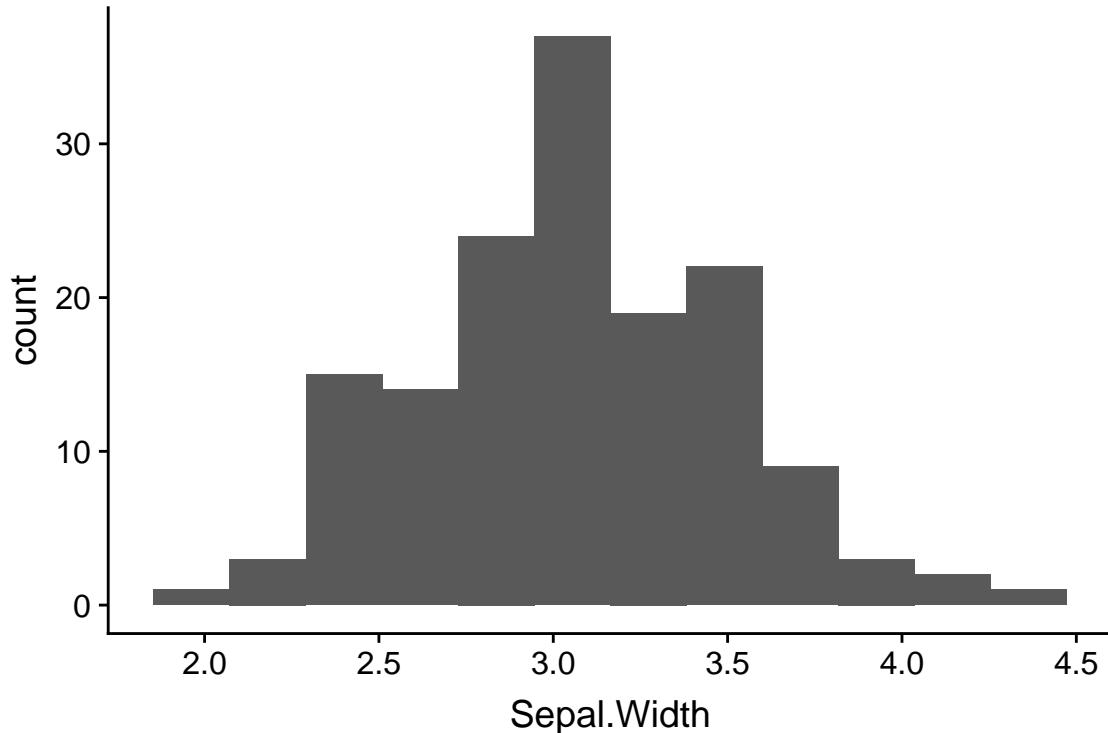


The default number of bins that `geom_histogram` uses is 30. For modest size data sets this is often too many bins, so it's worth exploring how the histogram changes with different bin numbers:

```
ggplot(iris) +  
  geom_histogram(aes(x = Sepal.Width), bins = 10)
```



```
ggplot(iris) +
  geom_histogram(aes(x = Sepal.Width), bins = 12)
```



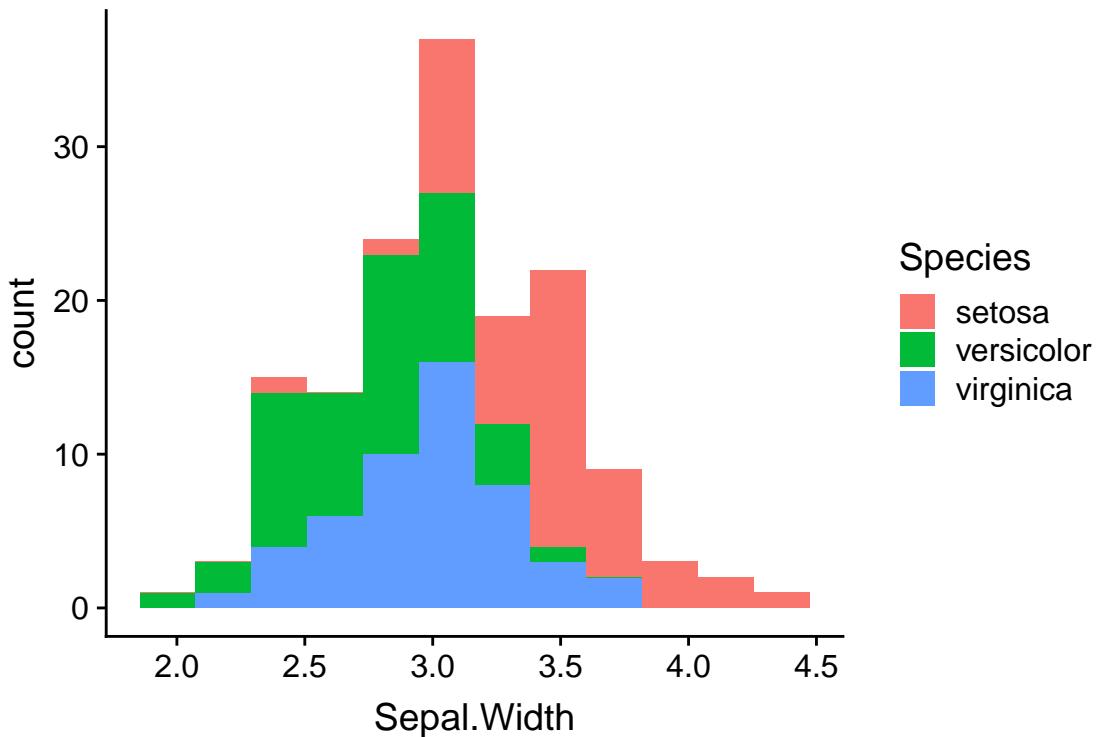
One important thing to note when looking at these histograms with different numbers of bins is that the number of bins used can change your perception of the data. For example, the number of peaks (modes) in the data can be very sensitive to the bin number as can the perception of gaps.

### 6.6.1 Variations on histograms when considering categorical data

As before, we probably want to break the data down by species. Here we're faced with some choices about how we depict that data. Do we generate a “stacked histogram” to where the colors indicate the number of observations in each bin that belong to each species? Do we generate side-by-side bars for each species? Or Do we generate separate histograms for each species, and show them overlapping?

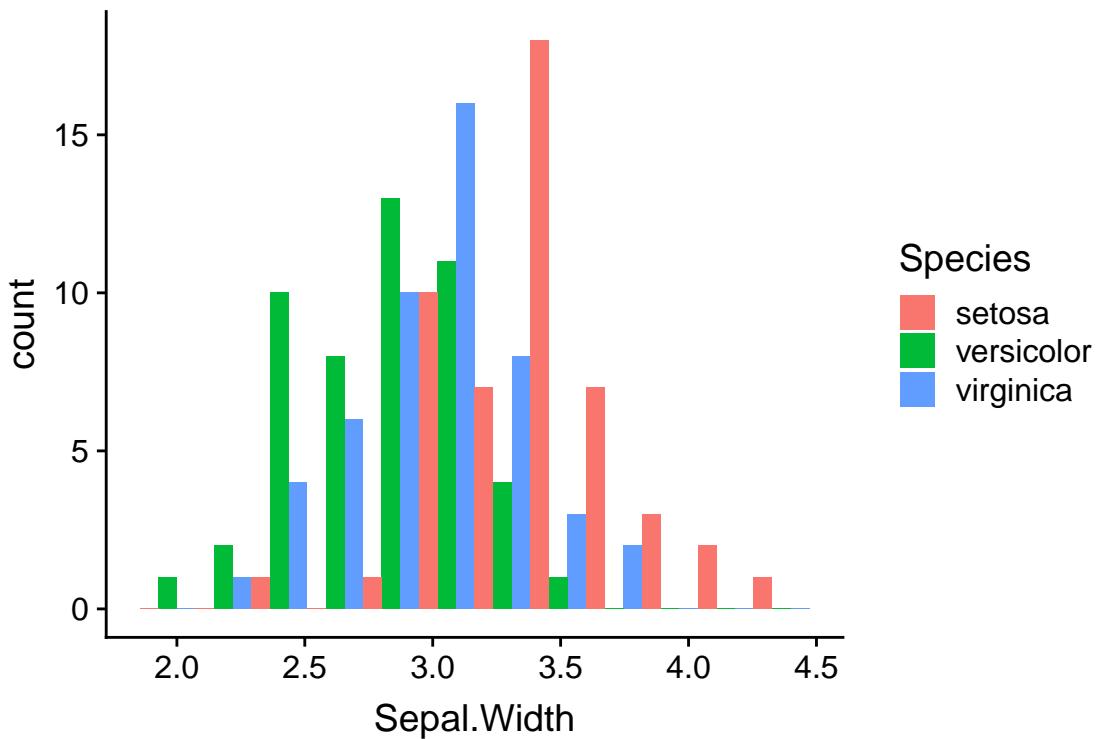
Stacked histograms are the default if we associate a categorical variable with the bar fill color:

```
ggplot(iris) +
  geom_histogram(aes(x = Sepal.Width, fill = Species),
                bins = 12)
```



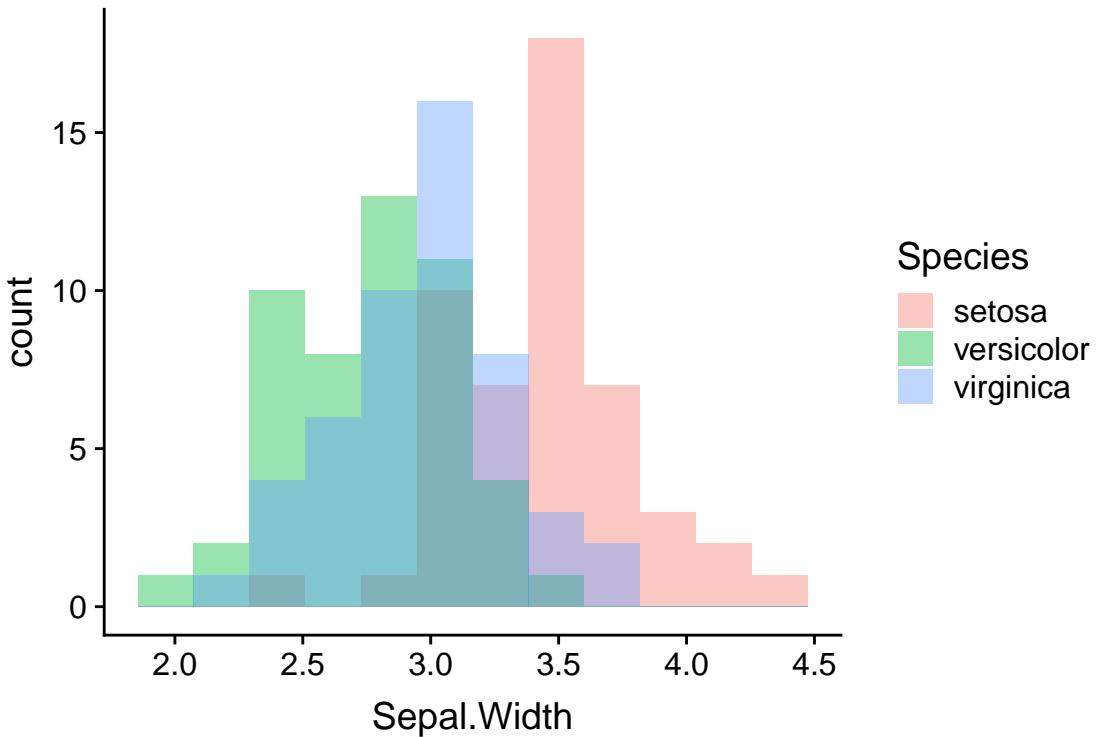
To get side-by-side bars, specify “dodge” as the position argument to `geom_histogram`.

```
ggplot(iris) +
  geom_histogram(aes(x = Sepal.Width, fill = Species),
                 bins = 12, position = "dodge")
```



If you want overlapping histograms, use `position = "identity"` instead. When generating overlapping histograms like this, you probably want to make the bars semi-transparent so you can distinguish the overlapping data.

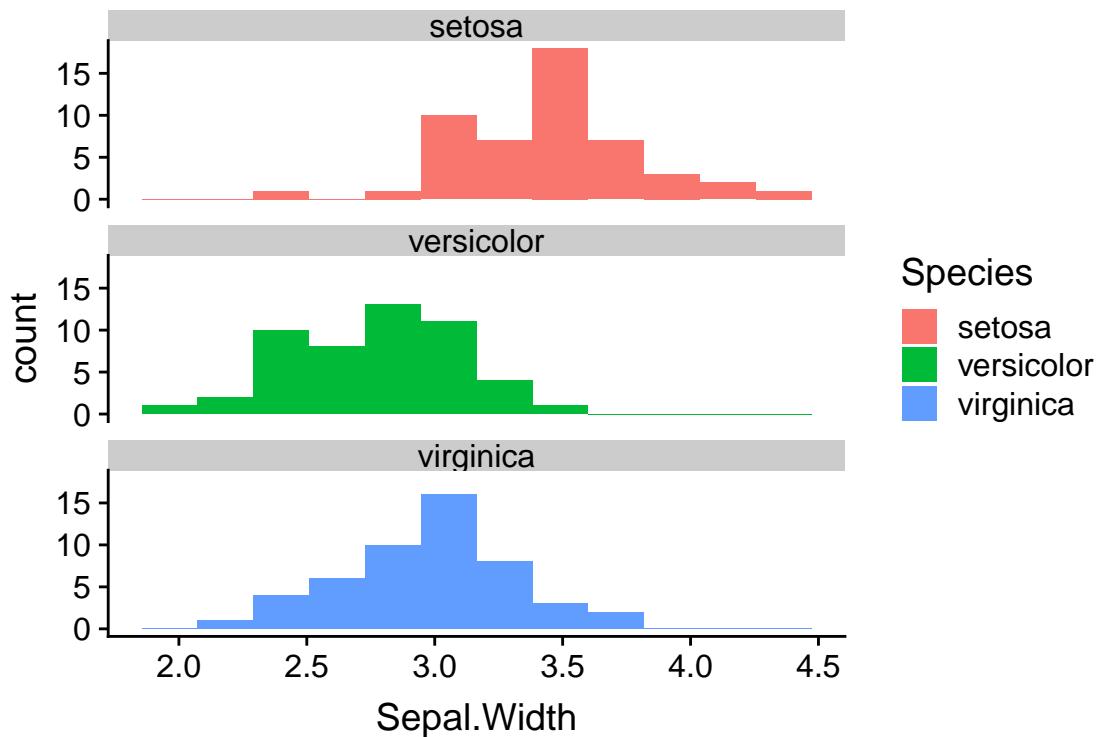
```
ggplot(iris) +
  geom_histogram(aes(x = Sepal.Width, fill = Species),
                 bins = 12, position = "identity", alpha = 0.4)
```



## 6.7 Faceting to depict categorical information

Yet another way to represent the histograms for the three species is to use faceting, which creates subplots for each species. Faceting is the operation of subsetting the data with respect to a discrete or categorical variable of interest, and generating the same plot type for each subset. Here we use the “`ncol`” argument to the `facet_wrap` function to specify that the subplots should be drawn in a single vertical column to facilitate comparison of the distributions.

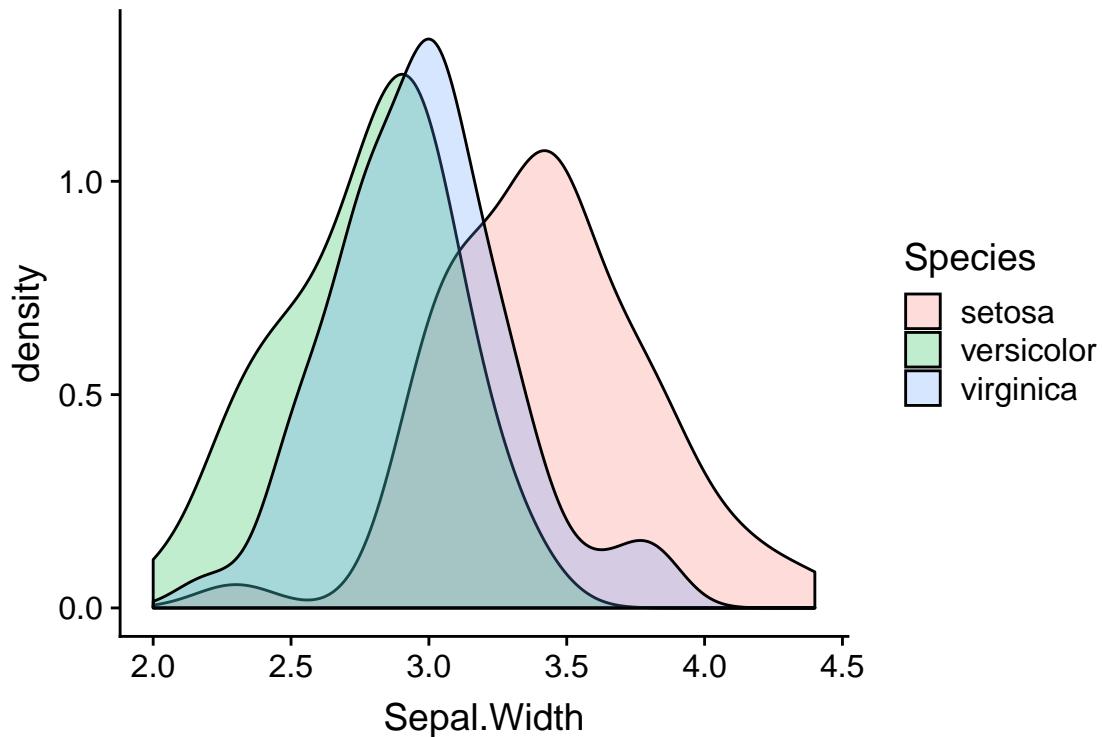
```
ggplot(iris) +
  geom_histogram(aes(x = Sepal.Width, fill = Species), bins = 12) +
  facet_wrap(~Species, ncol = 1)
```



## 6.8 Density plots

One shortcoming of histograms is that they are sensitive to the choice of bin margins and the number of bins. An alternative is a “density plot”, which you can think of as a smoothed version of a histogram.

```
ggplot(iris) +  
  geom_density(aes(x = Sepal.Width, fill = Species), alpha=0.25)
```



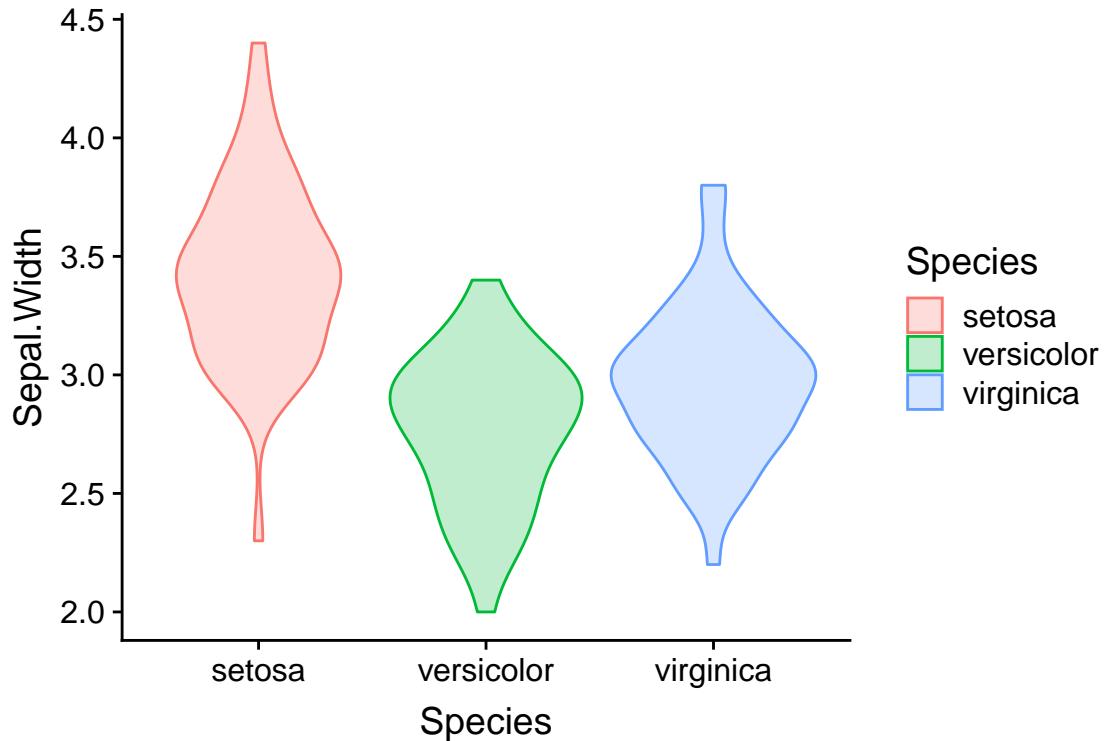
Density plots still make some assumptions that affect the visualization, in particular a “smoothing bandwidth” (specified by the argument `bw`) which determines how coarse or granular the density estimation is.

Note that the vertical scale on a density plot is no longer counts (frequency) but probability density. In a density plot, the total area under the plot adds up to one. Intervals in a density plot therefore have a probabilistic interpretation.

## 6.9 Violin or Beanplot

A violin plot (sometimes called a bean plot) is closely related to a density plot. In fact you can think of a violin plot as a density plot rotated 90 degrees and mirrored left/right.

```
ggplot(iris) +
  geom_violin(aes(x = Species, y = Sepal.Width, color = Species, fill=Species),
              alpha = 0.25)
```

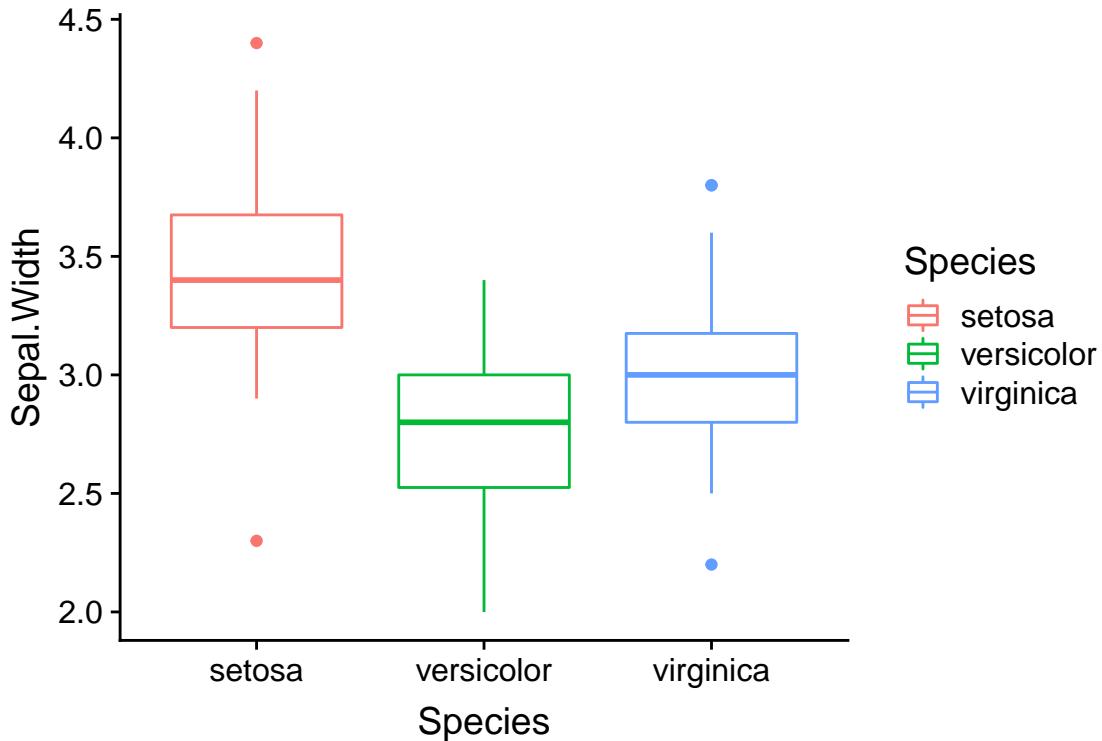


## 6.10 Boxplots

Boxplots are another frequently used univariate visualization. Boxplots provide a compact summary of single variables, and are most often used for comparing distributions between groups.

A standard box plot depicts five useful features of a set of observations: 1) the median (center most line); 2 and 3) the first and third quartiles (top and bottom of the box); 4) the whiskers of a boxplot extend from the first/third quartile to the highest value that is within  $1.5 * \text{IQR}$ , where IQR is the inter-quartile range (distance between the first and third quartiles); 5) points outside of the whiskers are usually considered extremal points or outliers. There are many variants on box plots, particularly with respect to the “whiskers”. It’s always a good idea to be explicit about what a box plot you’ve created shows.

```
ggplot(iris) +
  geom_boxplot(aes(x = Species, y = Sepal.Width, color = Species))
```



Boxplots are most commonly drawn with the categorical variable on the x-axis.

## 6.11 Building complex visualizations with layers

All of our ggplot2 examples up to now have involved a single geom. We can think of geoms as “layers” of information in a plot. One of the powerful features of plotting using ggplot2 is that it is trivial to combine layers to make more complex plots.

The template for multi-layered plots is a simple extension of the single layer:

```
ggplot(data = <DATA>) +
  <GEOM_FUNCTION1>(mapping = aes(<MAPPINGS>)) +
  <GEOM_FUNCTION2>(mapping = aes(<MAPPINGS>))
```

## 6.12 Useful combination plots

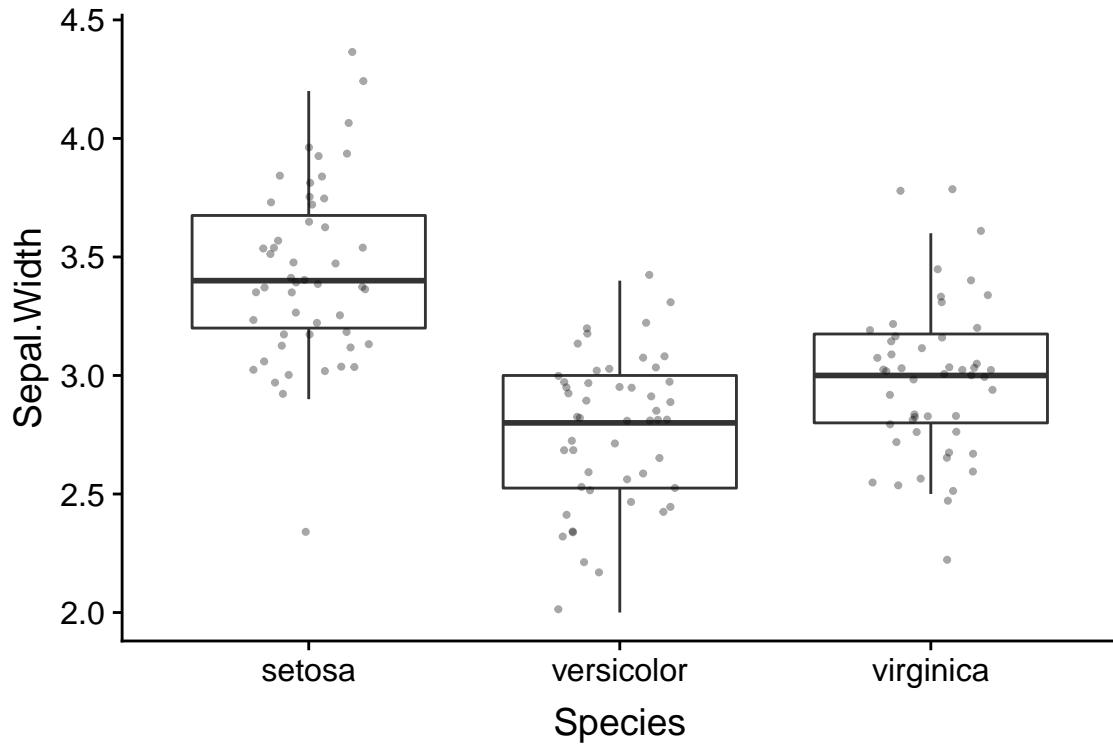
Boxplot or violin plots represent visual summaries/simplifications of the underlying data. This is useful but sometimes key information is lost in the process of summarizing. Combining these plots with a strip plot give you both the “birds eye view” as well as granular information.

### 6.12.1 Boxplot plus strip plot

Here’s an example of combining box plots and strip plots:

```
ggplot(iris) +
  # outlier.shape = NA suppresses the depiction of outlier points in the boxplot
  geom_boxplot(aes(x = Species, y = Sepal.Width), outlier.shape = NA) +
```

```
# size sets the point size for the jitter plot
geom_jitter(aes(x = Species, y = Sepal.Width), width=0.2, height=0.05, alpha=0.35, size=0.75)
```

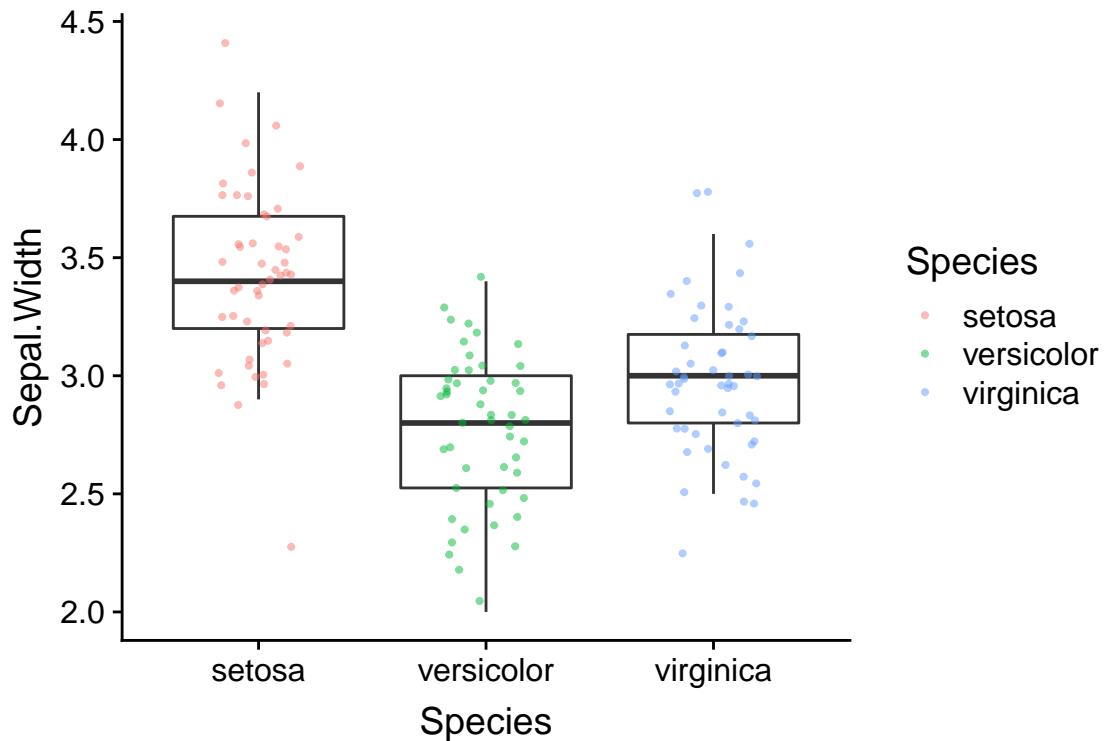


Note that I suppressed the plotting of outliers in `geom_boxplot` so as not to draw the same points twice (the individual data are drawn by `geom_jitter`).

### 6.12.2 Setting shared aesthetics

The example above works well, but you might have noticed that there's some repetition of code. In particular, we set the same aesthetic mapping in both `geom_boxplot` and `geom_jitter`. It turns out that creating layers that share some of the same aesthetic values is a common case. To deal with such cases, you can specify *shared aesthetic mappings* as an argument to the `ggplot` function and then set additional aesthetics specific to each layer in the individual geoms. Using this approach, our previous example can be written more compactly as follow.

```
ggplot(iris, mapping = aes(x = Species, y = Sepal.Width)) +
  geom_boxplot(outlier.shape = NA) +
  # note how we specify a layer specific aesthetic in geom_jitter
  geom_jitter(aes(color = Species), width=0.2, height=0.05, alpha=0.5, size=0.75)
```



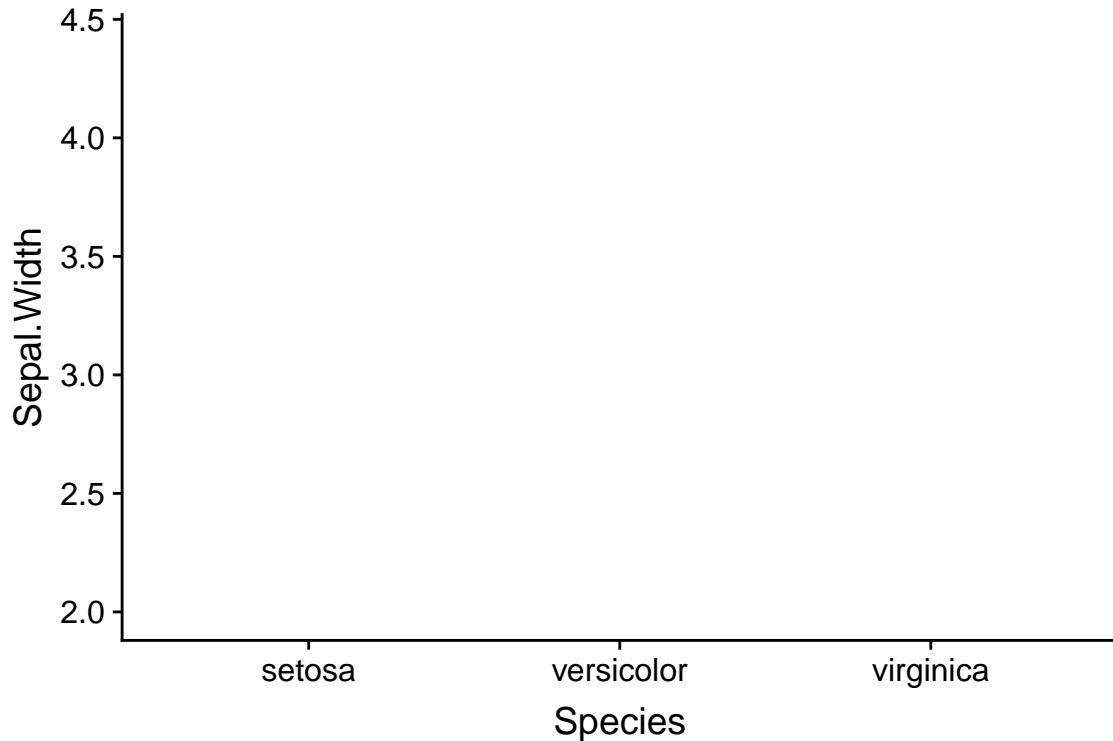
### 6.13 ggplot layers can be assigned to variables

The function `ggplot()` returns a “plot object” that we can assign to a variable. The following example illustrates this:

```
# create base plot object and assign to variable p
# this does NOT draw the plot
p <- ggplot(iris, mapping = aes(x = Species, y = Sepal.Width))
```

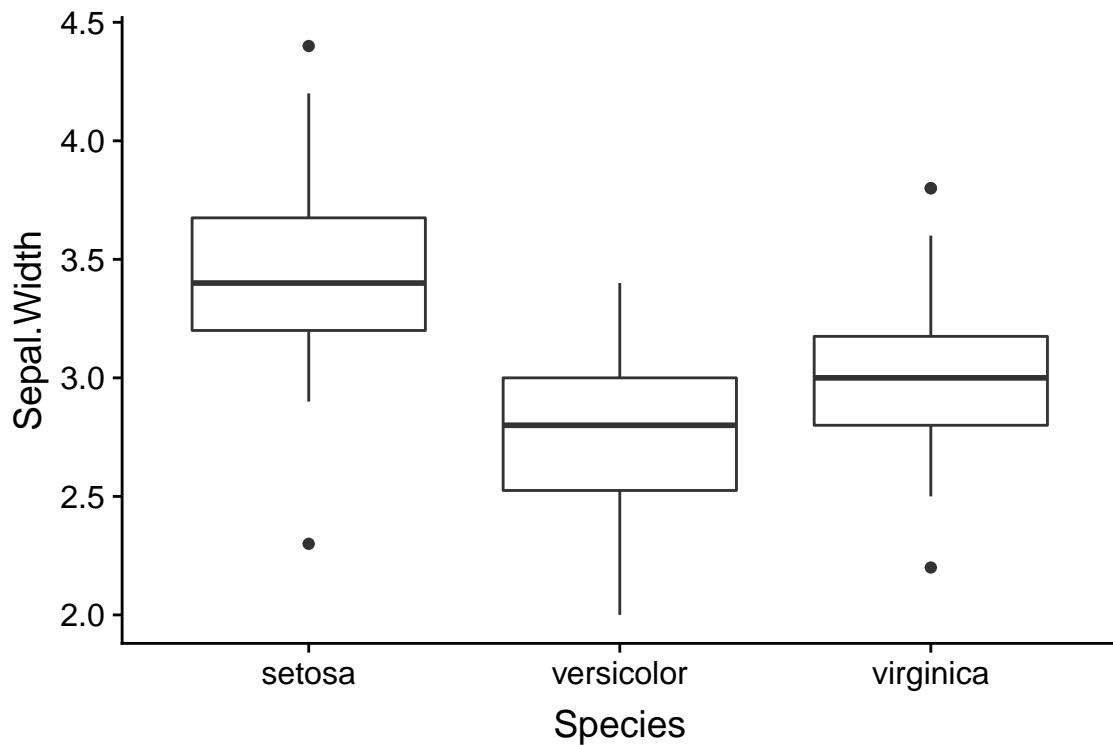
In the code above we created a plot object and assigned it to the variable `p`. However, the plot wasn’t drawn. To draw the plot object we evaluate it as so:

```
p # try to draw the plot object
```



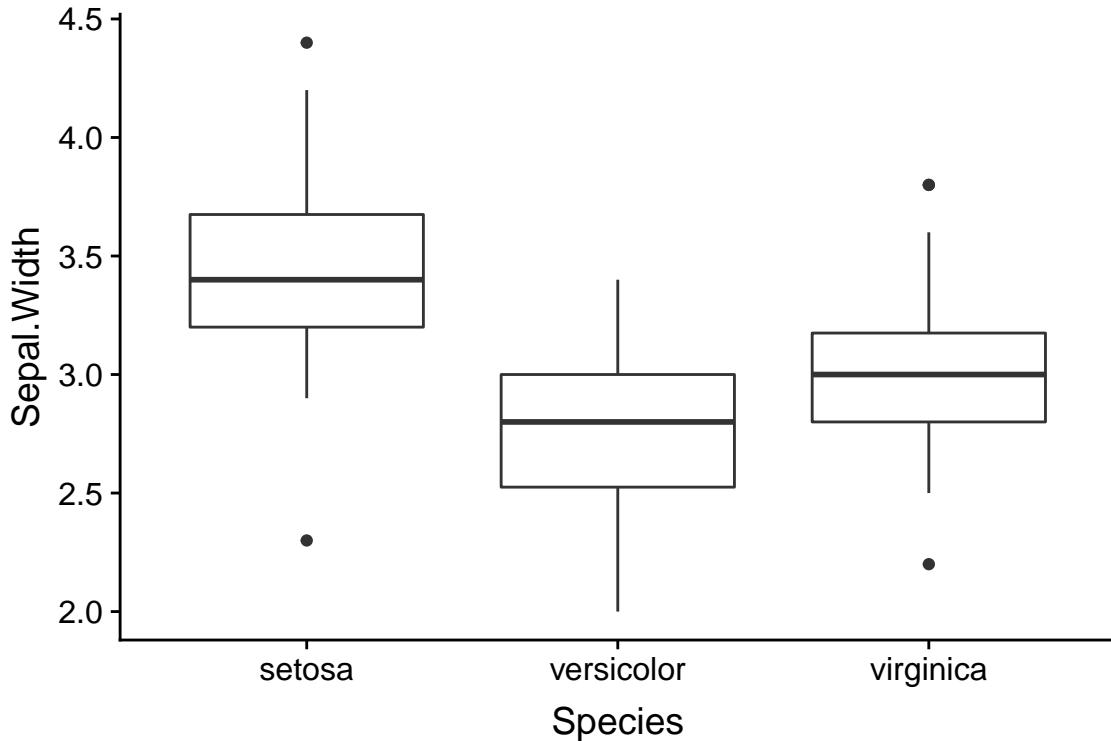
The code block above didn't generate an image, because we haven't added a geom to the plot to determine how our data should be drawn. We can add a geom to our pre-created plot object as so:

```
# add a point geom to our base layer and draw the plot  
p + geom_boxplot()
```



If we wanted to we could have assigned the geom to a variable as well:

```
box.layer <- geom_boxplot()
p + box.layer
```



In this case we don't really gain anything by creating an intermediate variable, but for more complex plots or when considering different versions of a plot this can be very useful.

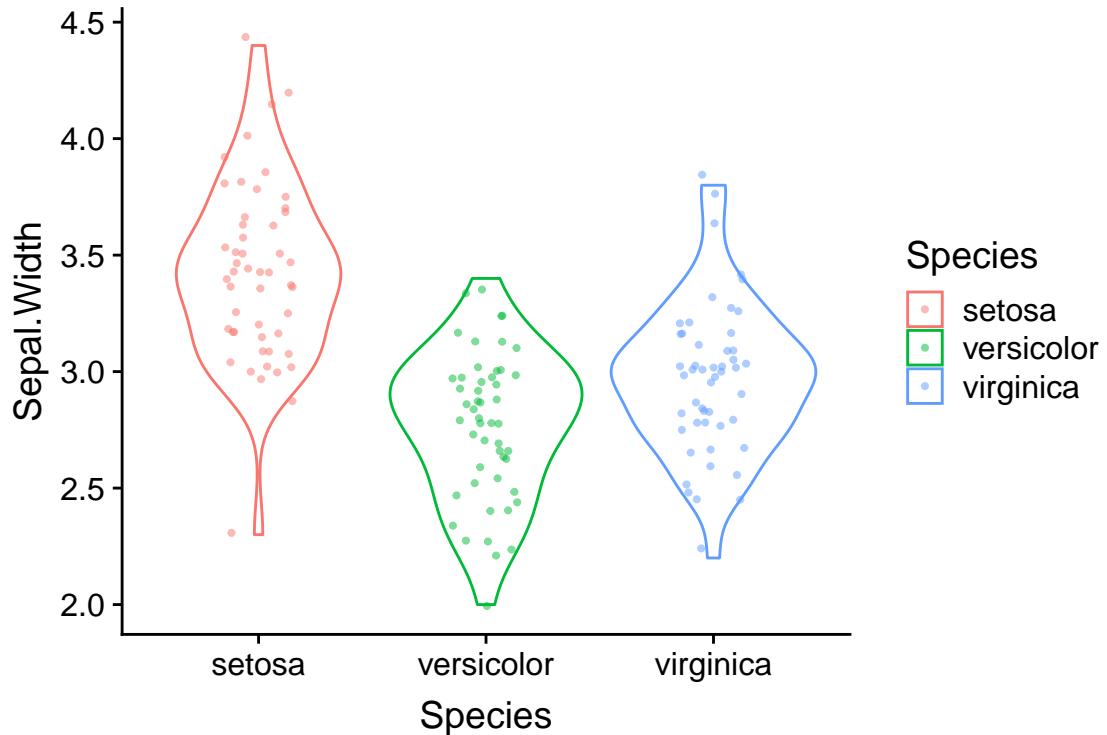
### 6.13.1 Violin plot plus strip plot

Here is the principle of combining layers, applied to a combined violin plot + strip plot. Again, we set shared aesthetic mappings in ggplot function call and this time we assign individual layers of the plot to variables.

```
p <- ggplot(iris, mapping = aes(x = Species, y = Sepal.Width, color = Species))

violin.layer <- geom_violin()
jitter.layer <- geom_jitter(width=0.15, height=0.05, alpha=0.5, size=0.75)

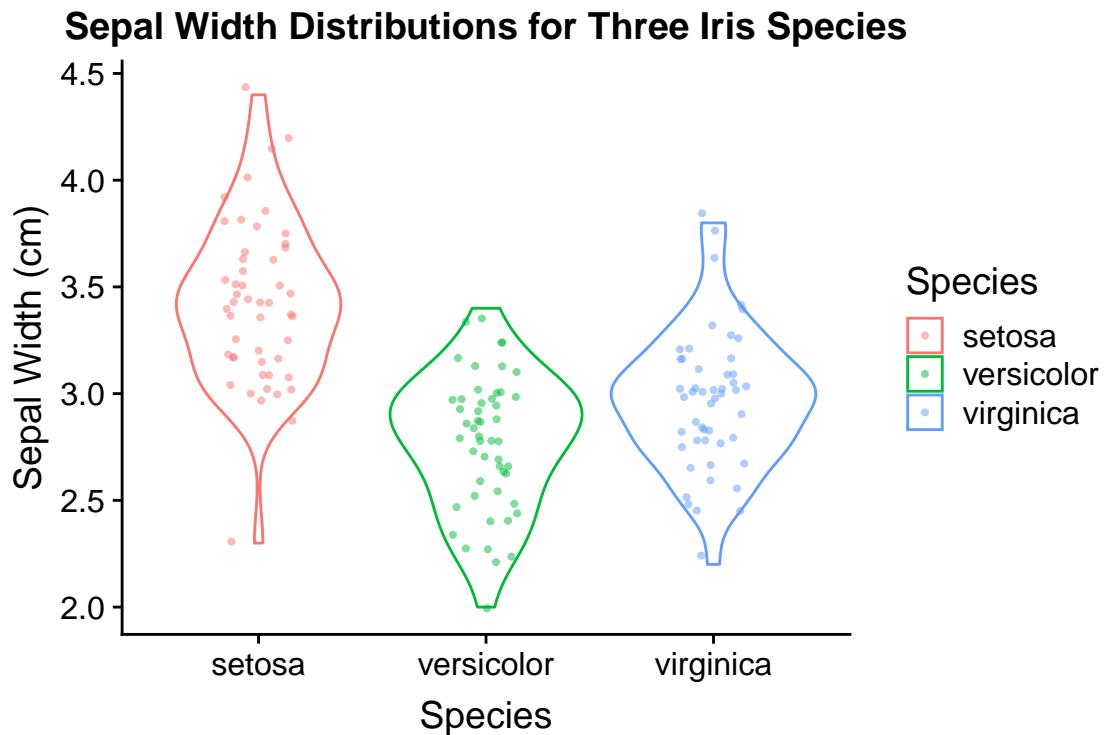
p + violin.layer + jitter.layer # combined layers of plot and draw
```



## 6.14 Adding titles and tweaking axis labels

ggplot2 automatically adds axis labels based on the variable names in the data frame passed to `ggplot`. Sometimes these are appropriate, but more presentable figures you'll usually want to tweak the axis labs (e.g. adding units). The `labs` (short for labels) function allows you to do so, and also let's you set a title for your plot. We'll illustrate this by modifying our previous figure. Note that we save considerable amounts of re-typing since we had already assigned three of the plot layers to variables in the previous code block:

```
p + violin.layer + jitter.layer +
  labs(x = "Species", y = "Sepal Width (cm)",
       title = "Sepal Width Distributions for Three Iris Species")
```



## 6.15 ggplot2 themes

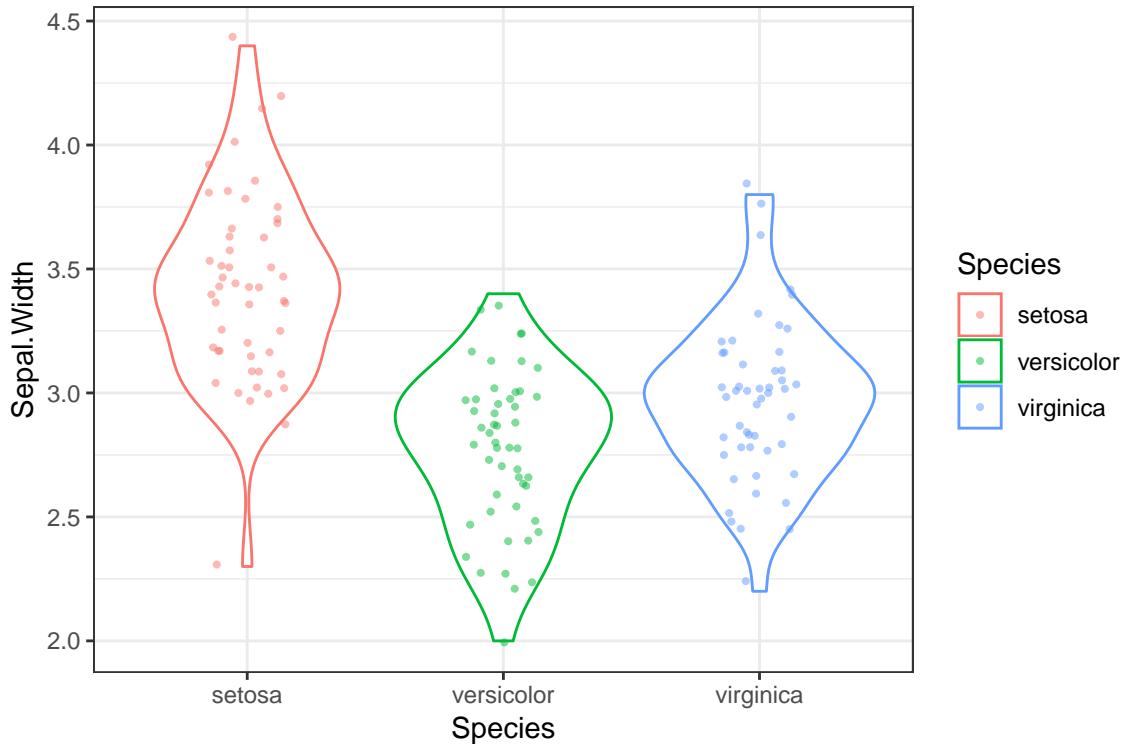
By now you're probably familiar with the default "look" of plots generated by ggplot2, in particular the ubiquitous gray background with a white grid. This default works fairly well in the context of RStudio notebooks and HTML output, but might not work as well for a published figure or a slide presentation. Almost every individual aspect of a plot can be tweaked, but ggplot2 provides an easier way to make consistent changes to a plot using "themes". You can think of a theme as adding another layer to your plot. Themes should generally be applied after all the other graphical layers are created (geoms, facets, labels) so the changes they create affect all the prior layers.

There are eight default themes included with ggplot2, which can be invoked by calling the corresponding theme functions: `theme_gray`, `theme_bw`, `theme_linedraw`, `theme_light`, `theme_dark`, `theme_minimal`, `theme_classic`, and `theme_void` (See <http://ggplot2.tidyverse.org/reference/ggtheme.html> for a visual tour of all the default themes)

For example, let's generate a boxplot using `theme_bw` which gets rid of the gray background:

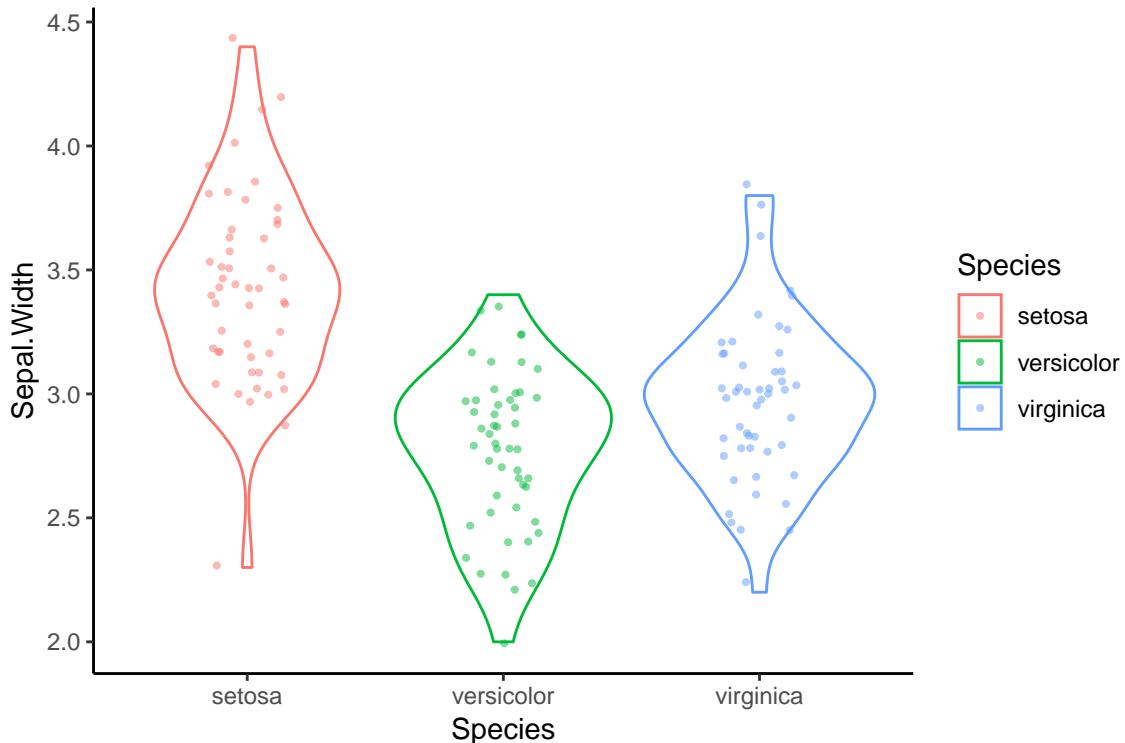
```
# create another variable to hold combination of three previous
# ggplot layers. I'm doing this because I'm going to keep re-using
# the same plot in the following code blocks
violin.plus.jitter <- p + violin.layer + jitter.layer

violin.plus.jitter + theme_bw()
```



Another theme, `theme_classic`, removes the grid lines completely, and also gets rid of the top-most and right-most axis lines.

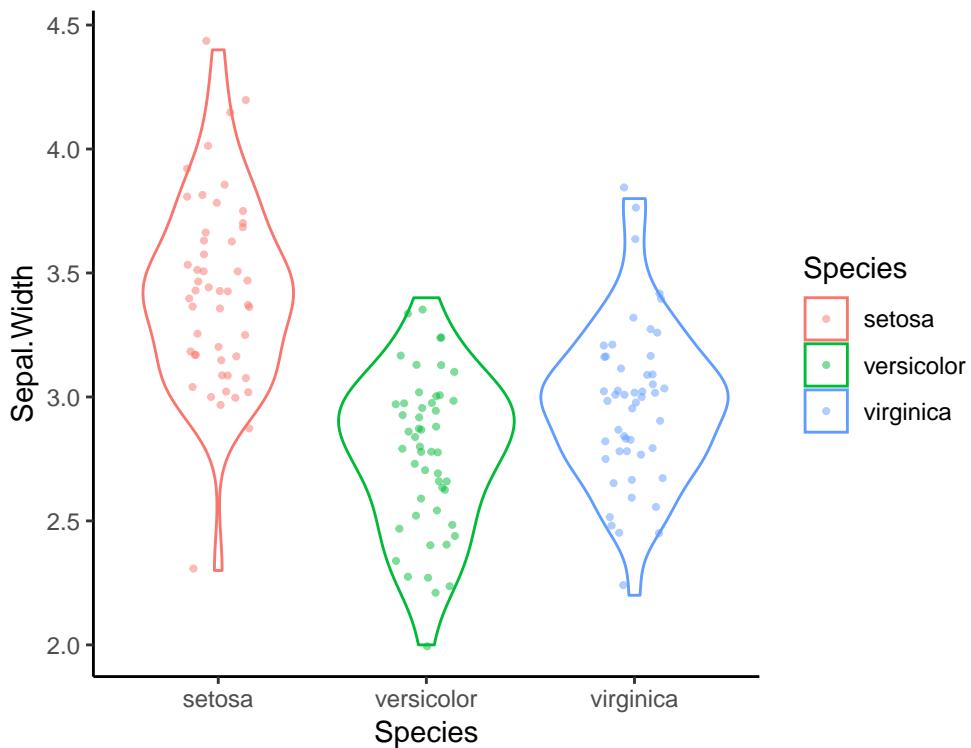
```
violin_plus_jitter + theme_classic()
```



### 6.15.1 Further customization with `ggplot2::theme`

In addition to the eight complete themes, there is a `theme` function in `ggplot2` that allows you to tweak particular elements of a theme (see `?theme` for all the possible options). For example, to tweak just the aspect ratio of a plot (the ratio of width to height), you can set the `aspect.ratio` argument in `theme`:

```
violin.plus.jitter + theme_classic() + theme(aspect.ratio = 1)
```

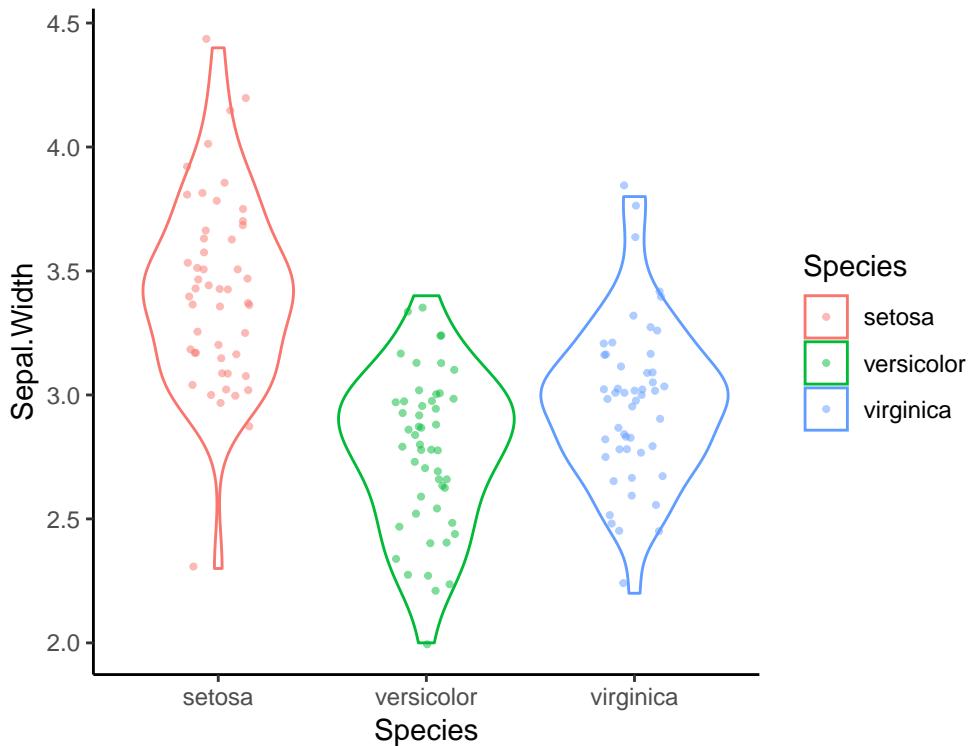


Theme related function calls can be combined to generate new themes. For example, let's create a theme called `my.theme` by combining `theme_classic` with a call to `theme`:

```
my.theme <- theme_classic() + theme(aspect.ratio = 1)
```

We can then apply this theme as so:

```
violin.plus.jitter + my.theme
```



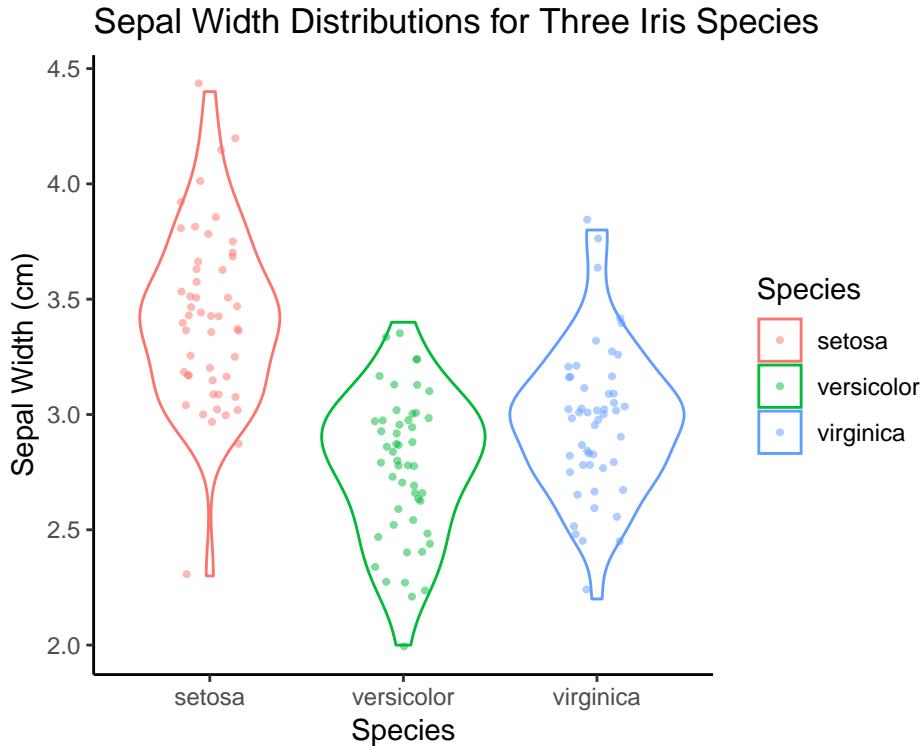
## 6.16 Other aspects of ggplots can be assigned to variables

Plot objects, geoms and themes are not the only aspects of a figure that can be assigned to variables for later use. For example, we can create a label object:

```
my.labels <- labs(x = "Species", y = "Sepal Width (cm)",  
                  title = "Sepal Width Distributions for Three Iris Species")
```

Combining all of our variables as so, we generate our new plot:

```
violin.plus.jitter + my.labels + my.theme
```



## 6.17 Bivariate plots

Now we turn our attention to some useful representations of bivariate distributions.

For the purposes of these illustrations I'm initially going to restrict my attention to just one of the three species represented in the iris data set – the *I. setosa* specimens. This allows us to introduce a very useful base function called `subset()`. `subset()` will return subsets of a vector or data frames that meets the specified conditions. This can also be accomplished with conditional indexing but `subset()` is usually less verbose.

```
# create a new data frame composed only of the I. setosa samples
setosa.only <- subset(iris, Species == "setosa")
```

In the examples that follow, I'm going to illustrate different ways of representing the same bivariate distribution – the joint distribution of Sepal Length and Sepal Width – over and over again. To avoid repetition, let's assign the base ggplot layer to a variable as we did in our previous examples. We'll also pre-create a label layer.

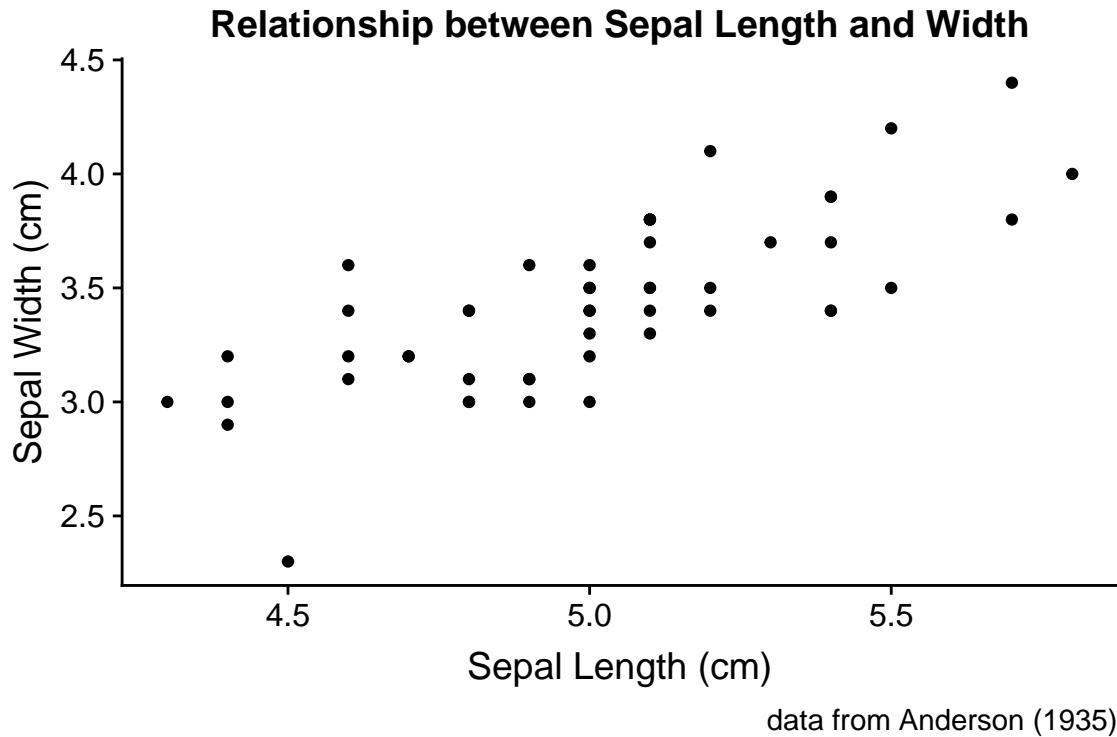
```
setosa.sepals <- ggplot(setosa.only,
                         mapping = aes(x = Sepal.Length, y = Sepal.Width))

sepal.labels <- labs(x = "Sepal Length (cm)", y = "Sepal Width (cm)",
                      title = "Relationship between Sepal Length and Width",
                      caption = "data from Anderson (1935)")
```

### 6.17.1 Scatter plots

A scatter plot is one of the simplest representations of a bivariate distribution. Scatter plots are simple to create in ggplot2 by specifying the appropriate X and Y variables in the aesthetic mapping and using `geom_point` for the geometric mapping.

```
setosa.sepals + geom_point() + sepal.labels
```

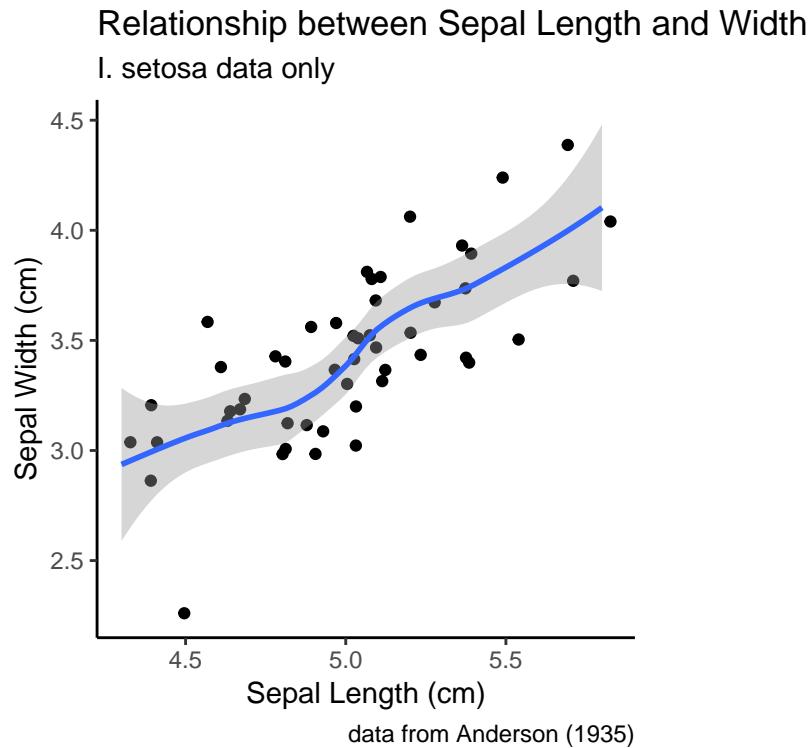


### 6.17.2 Adding a trend line to a scatter plot

ggplot2 makes it easy to add trend lines to plots. I use “trend lines” here to refer to representations like regression lines, smoothing splines, or other representations mean to help visualize the relationship between pairs of variables. We’ll spend a fair amount of time exploring the mathematics and interpretation of regression lines and related techniques in later lectures, but for now just think about trends lines as summary representations for bivariate relationships.

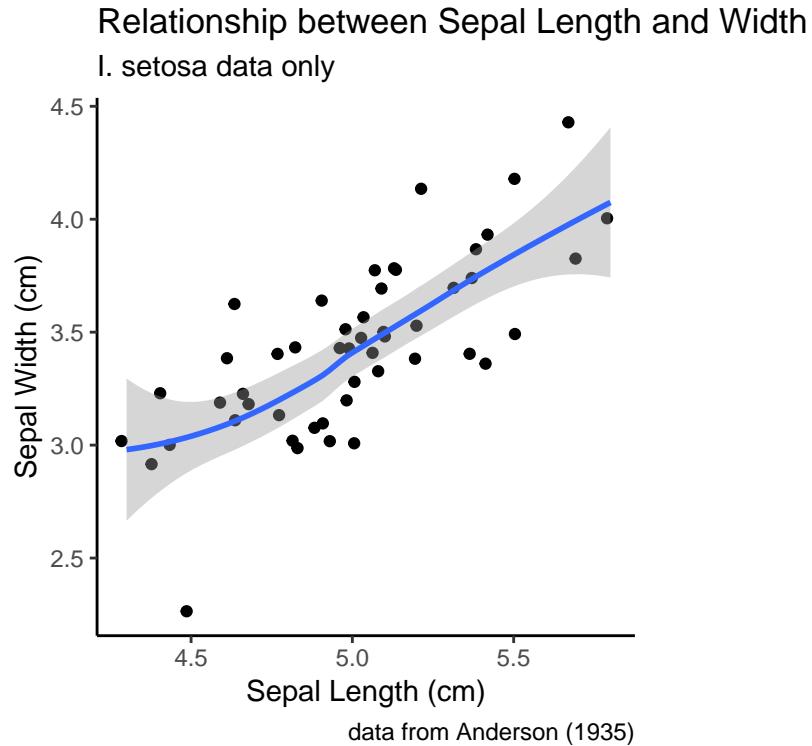
Trend lines can be created using `geom_smooth`. Let’s add a default trend line to our *I. setosa* scatter plot of the Sepal Width vs Sepal Length:

```
setosa.sepals +
  geom_jitter() + # using geom_jitter to avoid overplotting of points
  geom_smooth() +
  sepal.labels + labs(subtitle = "I. setosa data only") +
  my.theme
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



The default trend line that `geom_smooth` fits is generated by a technique called “LOESS regression”. LOESS regression is a non-linear curve fitting method, hence the squiggly trend line we see above. The smoothness of the LOESS regression is controlled by a parameter called `span` which is related to the proportion of points used. We’ll discuss LOESS in detail in a later lecture, but here’s an illustration how changing the `span` affects the smoothness of the fit curve:

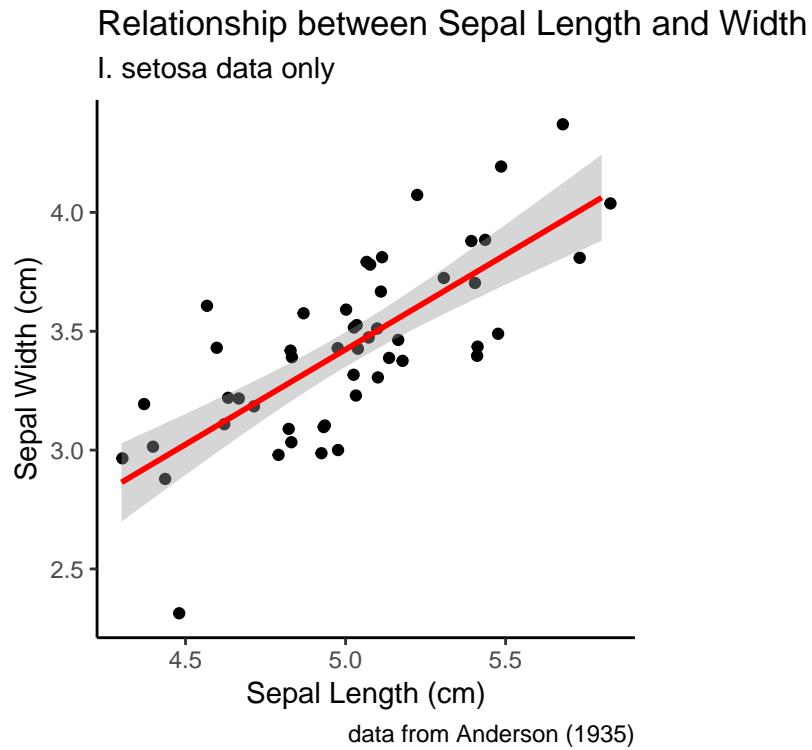
```
setosa.sepals +
  geom_jitter() + # using geom_jitter to avoid overplotting of points
  geom_smooth(span = 0.95) +
  sepal.labels + labs(subtitle = "I. setosa data only") +
  my.theme
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



#### 6.17.2.1 Linear trend lines

If instead we want a straight trend line, as would typically be depicted for a linear regression model we can specify a different statistical method:

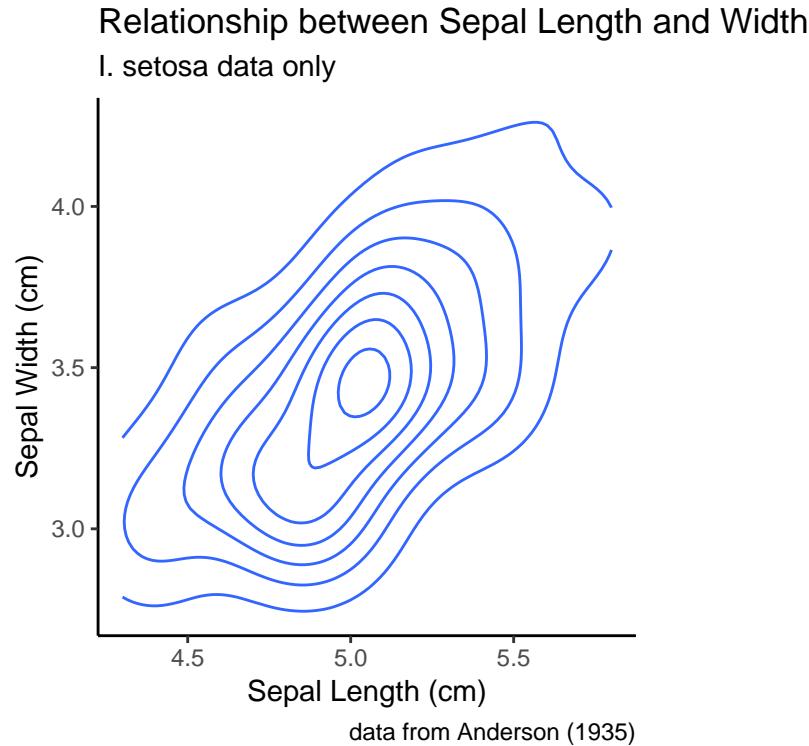
```
setosa.sepals +
  geom_jitter() + # using geom_jitter to avoid overplotting of points
  geom_smooth(method = "lm", color = "red") + # using linear model ("lm")
  sepal.labels + labs(subtitle = "I. setosa data only") +
  my.theme
```



## 6.18 Bivariate density plots

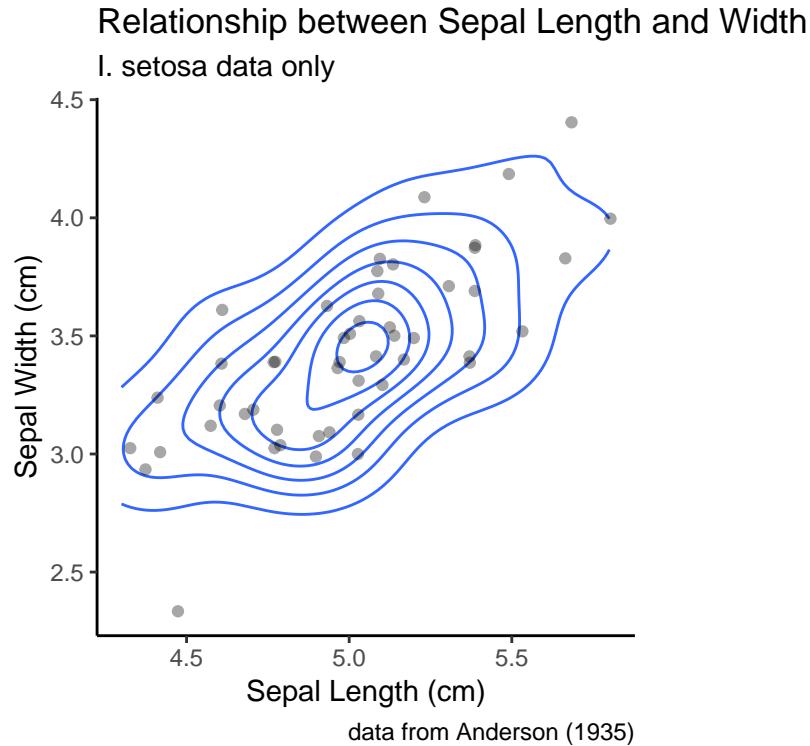
The density plot, which we introduced as a visualization for univariate data, can be extended to two-dimensional data. In a one dimensional density plot, the height of the curve was related to the relatively density of points in the surrounding region. In a 2D density plot, nested contours (or contours plus colors) indicate regions of higher local density. Let's illustrate this with an example:

```
setosa.sepals +
  geom_density2d() +
  sepal.labels + labs(subtitle = "I. setosa data only") +
  my.theme
```



The relationship between the 2D density plot and a scatter plot can be made clearer if we combine the two:

```
setosa.sepals +
  geom_density_2d() +
  geom_jitter(alpha=0.35) +
  sepal.labels +
  labs(subtitle = "I. setosa data only") +
  my.theme
```

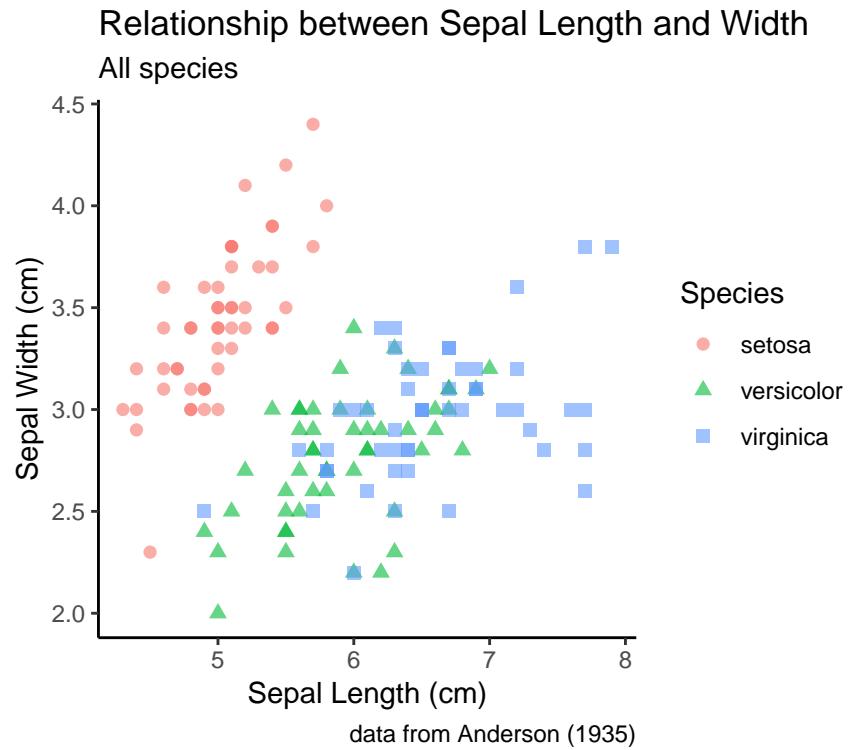


## 6.19 Combining Scatter Plots and Density Plots with Categorical Information

As with many of the univariate visualizations we explored, it is often useful to depict bivariate relationships as we change a categorical variable. To illustrate this, we'll go back to using the full iris data set.

```
all.sepals <- ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width))

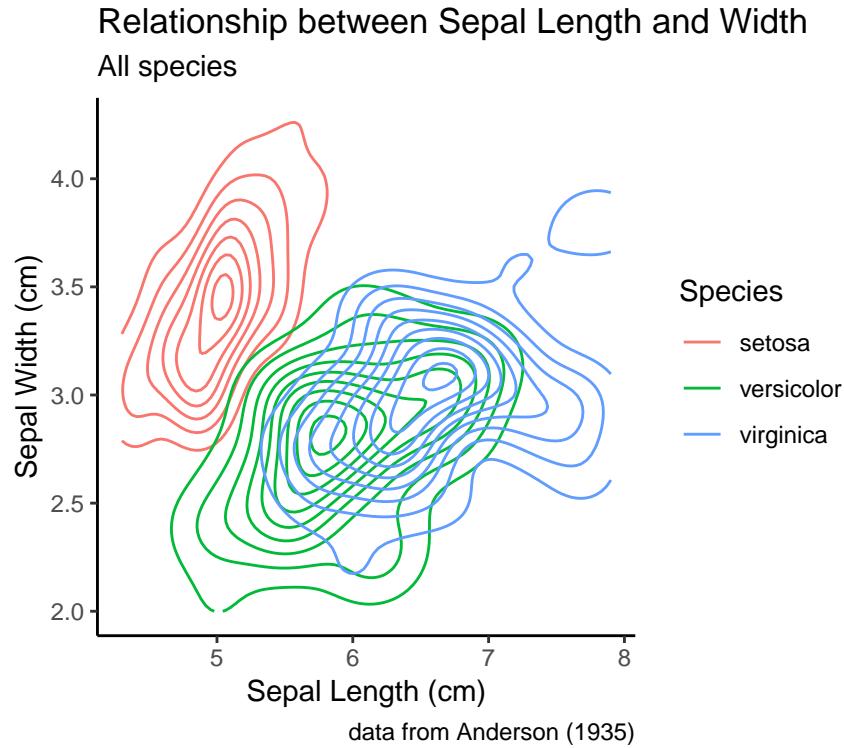
all.sepals +
  geom_point(aes(color = Species, shape = Species), size = 2, alpha = 0.6) +
  sepal.labels +
  labs(subtitle = "All species") +
  my.theme
```



Notice how in our aesthetic mapping we specified that both color and shape should be used to represent the species categories.

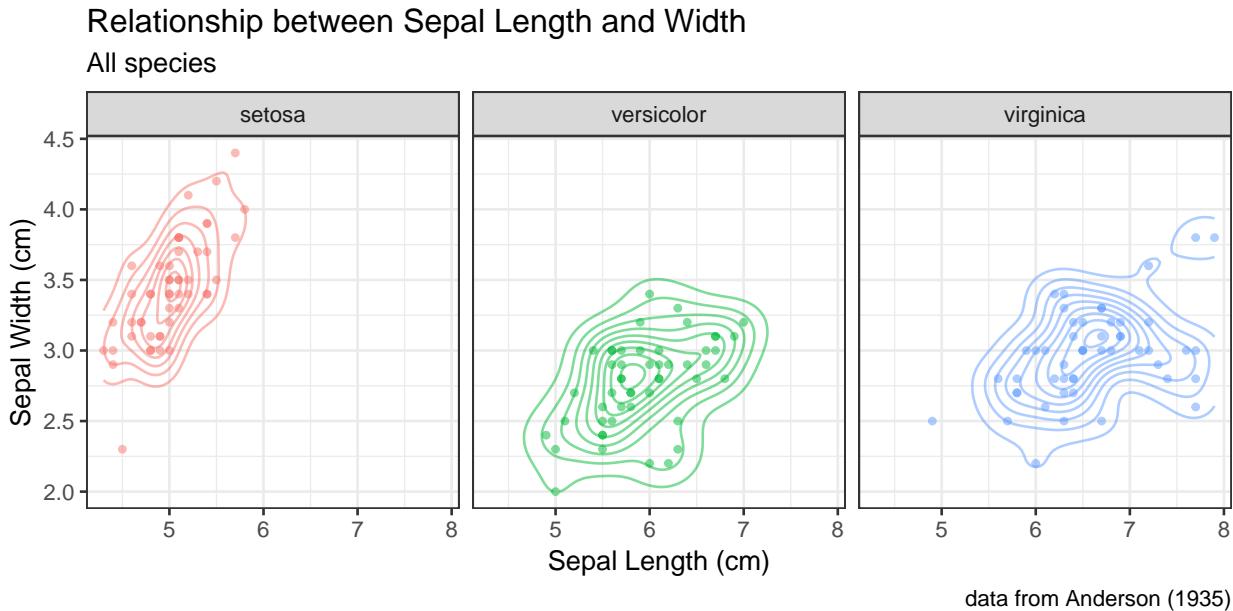
The same thing can be accomplished with a 2D density plot.

```
all.sepals +
  geom_density_2d(aes(color = Species)) +
  sepal.labels + labs(subtitle = "All species") +
  my.theme
```



As you can see, in the density plots above, when you have multiple categorical variables and there is significant overlap in the range of each sub-distribution, figures can become quite busy. As we've seen previously, faceting (conditioning) can be a good way to deal with this. Below a combination of scatter plots and 2D density plots, combined with facetting on the species variable.

```
all.sepals +
  geom_density_2d(aes(color = Species), alpha = 0.5) +
  geom_point(aes(color = Species), alpha=0.5, size=1) +
  facet_wrap(~ Species) +
  sepal.labels + labs(subtitle = "All species") +
  theme_bw() +
  theme(aspect.ratio = 1, legend.position = "none") # get rid of legend
```

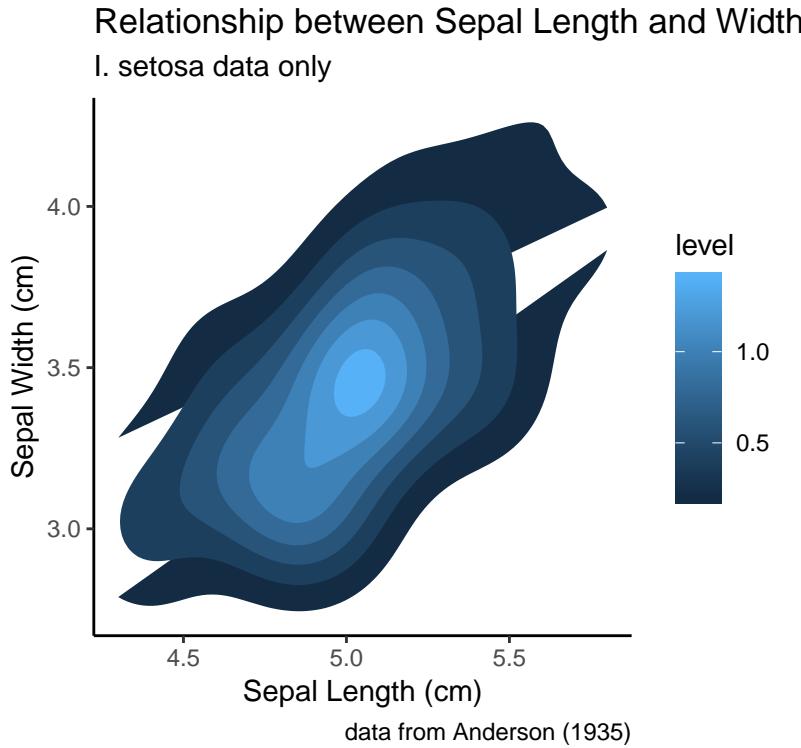


In this example I went back to using a theme that includes grid lines to facilitate more accurate comparisons of the distributions across the facets. I also got rid of the legend, because the information there was redundant.

## 6.20 Density plots with fill

Let's revisit our earlier single species 2D density plot. Instead of simply drawing contour lines, let's use color information to help guide the eye to areas of higher density. To draw filled contours, we use a sister function to `geom_density_2d` called `stat_density_2d`:

```
setosa.sepals +
  stat_density_2d(aes(fill = ..level..), geom = "polygon") +
  sepal.labels + labs(subtitle = "I. setosa data only") +
  my.theme
```



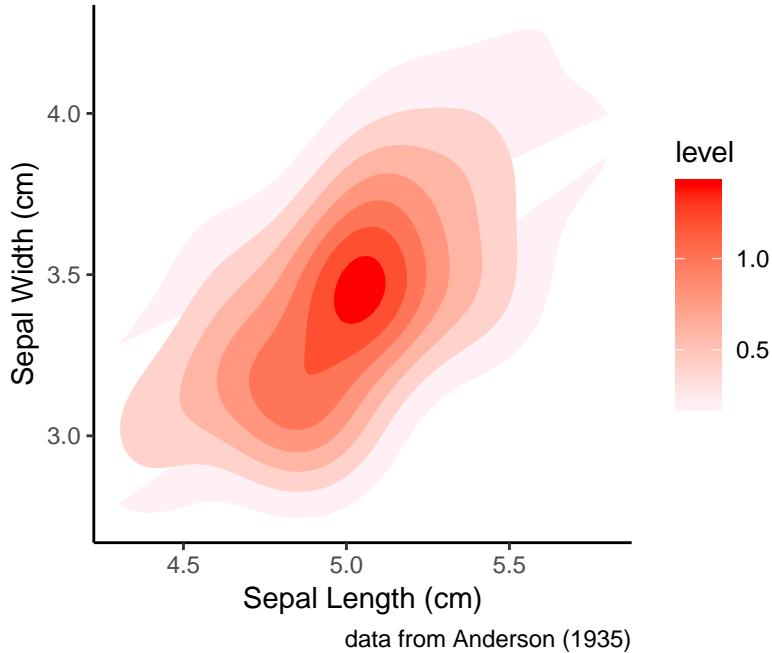
Using the default color scale, areas of low density are drawn in dark blue, whereas areas of high density are drawn in light blue. I personally find this dark -to-light color scale non-intuitive for density data, and would prefer that darker regions indicate area of higher density. If we want to change the color scale, we can use the `a scale function` (in this case `scale_fill_continuous`) to set the color values used for the low and high values (this function we'll interpolate the intervening values for us).

NOTE: when specifying color names, R accepts standard HTML color names (see the Wikipedia page on web colors for a list). We'll also see other ways to set color values in a later class session.

```
setosa.sepals +
  stat_density_2d(aes(fill = ..level..), geom = "polygon") +
  # lavenderblush is the HTML standard name for a light purplish-pink color
  scale_fill_continuous(low="lavenderblush", high="red") +
  sepal.labels + labs(subtitle = "I. setosa data only") +
  my.theme
```

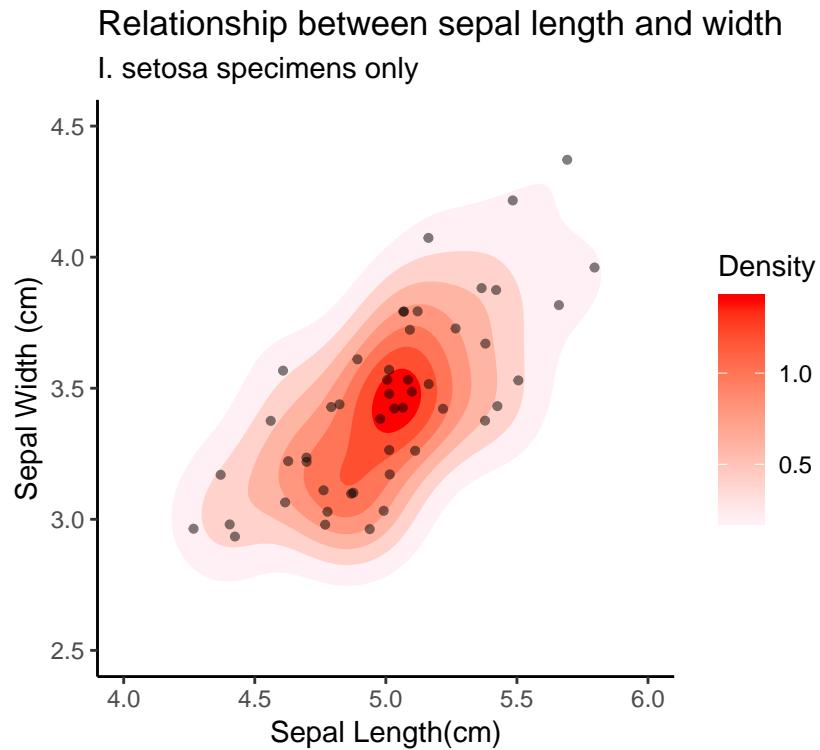
## Relationship between Sepal Length and Width

I. setosa data only



The two contour plots we generated looked a little funny because the contours are cutoff due to the contour regions being outside the limits of the plot. To fix this, we can change the plot limits using the `lims` function as shown in the following code block. We'll also add the scatter (jittered) to the emphasize the relationship between the levels, and we'll change the title for the color legend on the right by specifying a text label associated with the fill arguments in the `labs` function.

```
setosa.sepals +
  stat_density_2d(aes(fill = ..level..), geom = "polygon") +
  scale_fill_continuous(low="lavenderblush", high="red") +
  geom_jitter(alpha=0.5, size = 1.1) +
  
  # customize labels, including legend label for fill
  labs(x = "Sepal Length(cm)", y = "Sepal Width (cm)",
       title = "Relationship between sepal length and width",
       subtitle = "I. setosa specimens only",
       fill = "Density") +
  
  # Set plot limits
  lims(x = c(4,6), y = c(2.5, 4.5)) +
  my.theme
```

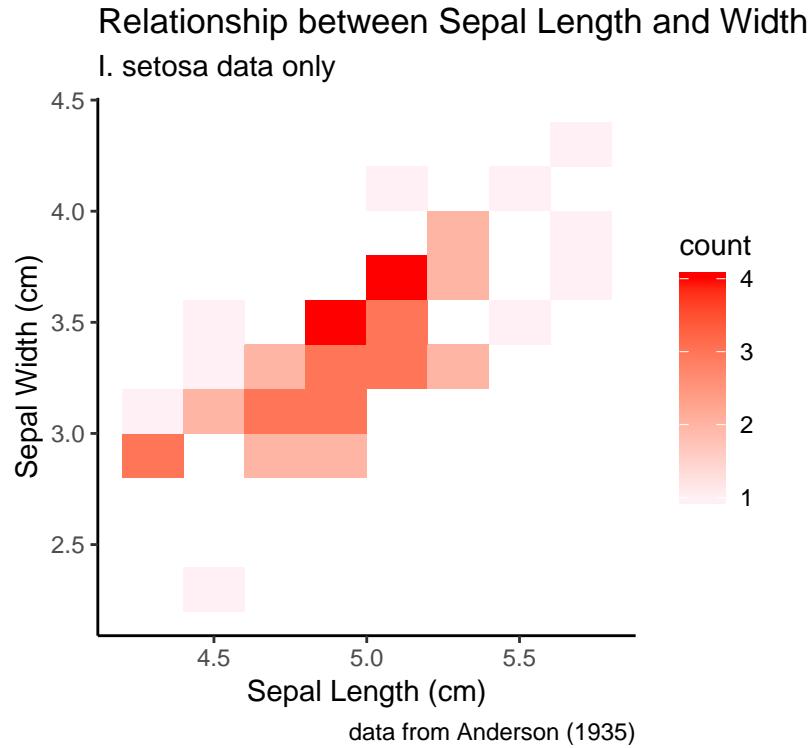


## 6.21 2D bin and hex plots

Two dimensional bin and hex plots are alterative ways to represent the joint density of points in the Cartesian plane. Here are examples of to generate these plot types. Compare them to our previous examples.

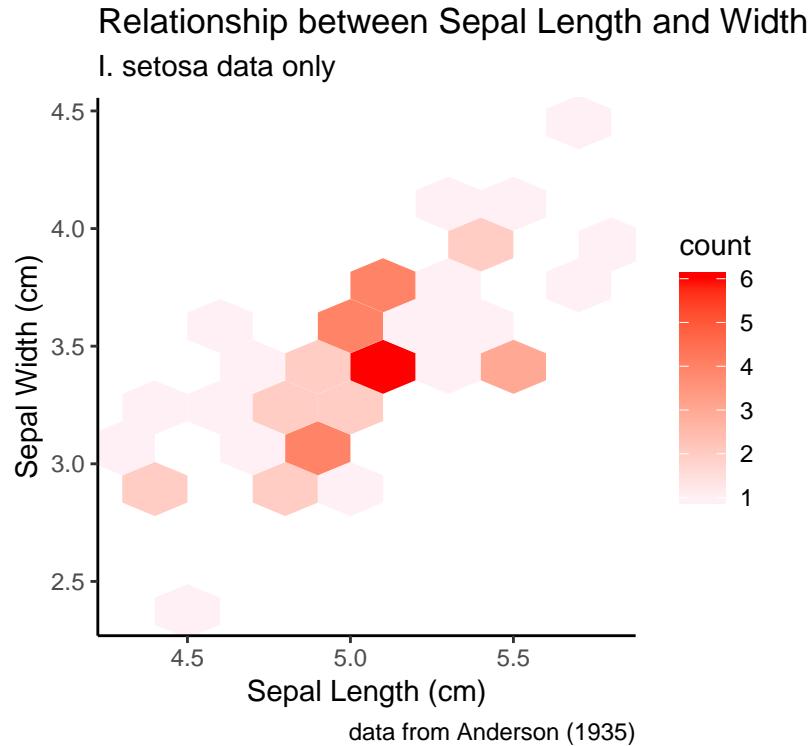
A 2D bin plot can be thought of as a 2D histogram:

```
setosa.sepals +
  geom_bin2d(binwidth = 0.2) +
  scale_fill_continuous(low="lavenderblush", high="red") +
  sepal.labels +
  labs(subtitle = "I. setosa data only") +
  my.theme
```



A hex plot is similar to a 2D bin plot but uses hexagonal regions instead of squares. Hexagonal bins are useful because they can sometimes avoid visual artefacts sometimes apparent with square bins:

```
setosa.sepals +
  geom_hex(binwidth = 0.2) +
  scale_fill_continuous(low="lavenderblush", high="red") +
  sepal.labels + labs(subtitle = "I. setosa data only") +
  my.theme
```



## 6.22 The cowplot package

A common task when preparing visualizations for scientific presentations and manuscripts is combining different plots as subfigures of a larger figure. To accomplish this we'll use a package called `cowplot` that complements the power of `ggplot2`. Install `cowplot` either via the command line or the R Studio GUI (see Section 3.11).

```
library(cowplot) # assumes package has been installed
```

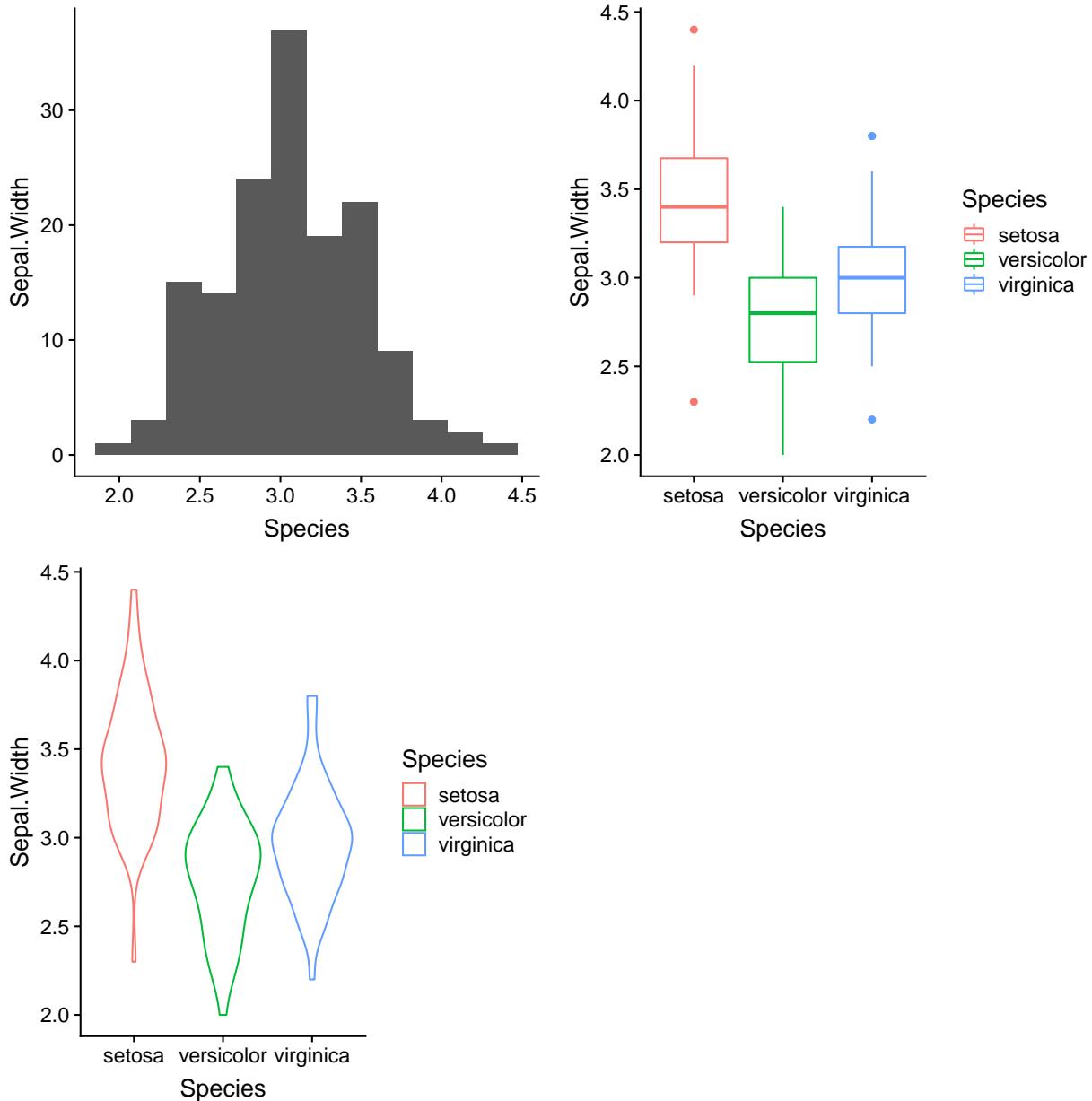
`cowplot` allows us to create individual plots using `ggplot`, and then arrange them in a grid-like fashion with labels for each plot of interest, as you would typically see in publications. The core function of `cowplot` is `plot_grid()`, which allows the user to layout the sub-plots in an organized fashion and add labels as necessary.

To illustrate `plot_grid()` let's create three different representations of the distribution of sepal width in the `iris` data set, and combine them into a single figure:

```
p <- ggplot(iris,
             mapping = aes(x = Species, y = Sepal.Width, color = Species))

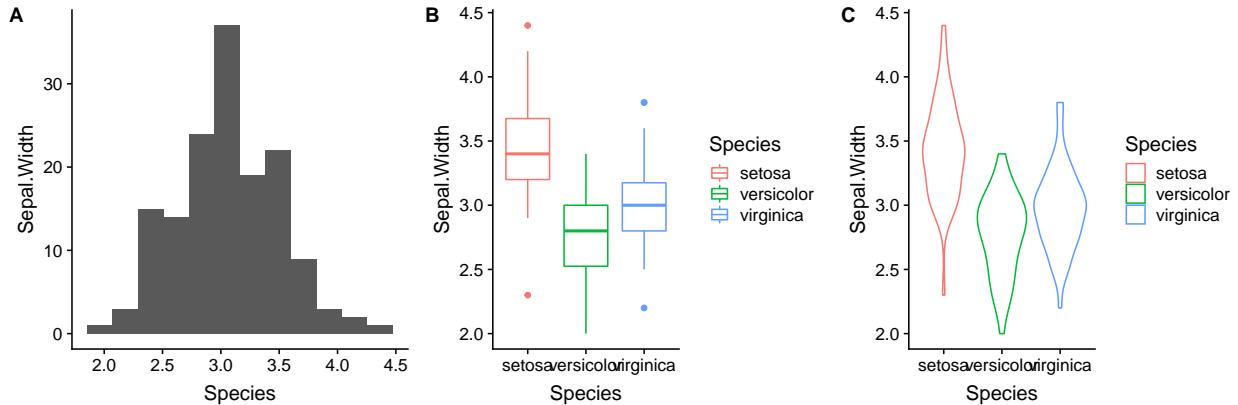
# for the histogram we're going to override the mapping because
# geom_histogram only takes an x argument
plot.1 <- p +
  geom_histogram(bins=12,
                 mapping = aes(x = Sepal.Width), inherit.aes = FALSE)
plot.2 <- p + geom_boxplot()
plot.3 <- p + geom_violin()

plot_grid(plot.1, plot.2, plot.3)
```



If instead, we wanted to layout the plots in a single row we could change the call to `plot_grid` as so:

```
plot_grid(plot.1, plot.2, plot.3,
          nrow = 1, labels = c("A", "B", "C"))
```



Notice we also added labels to our sub-plots.



# Chapter 7

## Introduction to the `dplyr` package

In today's class we introduce a new package, `dplyr`, which, along with `ggplot2` will be used in almost every class session. We will also explore in a little more depth the `readr` package, for reading tabular data.

### 7.1 Libraries

Both `readr` and `dplyr` are members of the tidyverse, so a single invocation of `library()` makes the functions defined in these two packages available for our use:

```
library(tidyverse)
```

### 7.2 Reading data with the `readr` package

The `readr` package defines a number of functions for reading data tables from common file formats like Comma-Separated-Value (CSV) and Tab-Separated-Value (TSV) files.

The two most frequently used `readr` functions we'll use in this class are `read_csv()` and `read_tsv()` for reading CSV and TSV files respectively. There are some variants of these basic function, which you can read about by invoking the help system (`?read_csv`).

#### 7.2.1 Example data: NC Births

For today's hands on session we'll use a data set that contains information on 150 cases of mothers and their newborns in North Carolina in 2004. This data set is available at the following URL:

- <https://github.com/bio304-class/bio304-course-notes/raw/master/datasets/nc-births.txt>

The births data is a TSV file, so we'll use the `read_tsv()` function to read it:

```
births <- read_tsv("https://github.com/bio304-class/bio304-course-notes/raw/master/datasets/nc-births.txt")
#> Parsed with column specification:
#> cols(
#>   fAge = col_integer(),
#>   mAge = col_integer(),
#>   weeks = col_integer(),
#>   premature = col_character(),
#>   visits = col_integer(),
```

```
#>   gained = col_integer(),
#>   weight = col_double(),
#>   sexBaby = col_character(),
#>   smoke = col_character()
#> )
```

Notice that when you used `read_tsv()` the function printed information about how it “parsed” the data (i.e. the types it assigned to each of the columns).

The variables in the data set are:

- father’s age (`fAge`),
- mother’s age (`mAge`),
- weeks of gestation (`weeks`)
- whether the birth was premature or full term (`premature`)
- number of OB/GYN visits (`visits`)
- mother’s weight gained in pounds (`gained`)
- babies birth weight (`weight`)
- sex of the baby (`sexBaby`)
- whether the mother was a smoker (`smoke`).

Notice too that we read the TSV file directly from a remote location via a URL. If instead, you wanted to load a local file on your computer you would specify the “path” – i.e. the location on your hard drive where you stored the file. For example, here is how I would load the same file if it was stored in the Downloads directory on my Mac laptop:

```
# load the data from a local file
births <- read_tsv("/Users/pmagwene/Downloads/nc-births.txt")
```

### 7.2.2 Reading Excel files

The tidyverse also includes a package called `readxl` which can be used to read Excel spreadsheets (recent versions with `.xls` and `.xlsx` extensions). Excel files are somewhat more complicated to deal with because they can include separate “sheets”. We won’t use `readxl` in this class, but documentation and examples of how `readxl` is used can be found at the page linked above.

## 7.3 A note on “tibbles”

You may have noticed that most of the functions defined in tidyverse related packages return not data frames, but rather something called a “tibble”. You can think about tibbles as light-weight data frames. In fact if you ask about the “class” of a tibble you’ll see that it includes `data.frame` as one of its classes as well as `tbl` and `tbl_df`.

```
class(births)
#> [1] "tbl_df"     "tbl"        "data.frame"
```

There are some minor differences between data frame and tibbles. For example, tibbles print differently in the console and don’t automatically change variable names and types in the same way that standard data frames do. Usually tibbles can be used wherever a standard data frame is expected, but you may occasionally find a function that only works with a standard data frame. It’s easy to convert a tibble to a standard data frame using the `as.data.frame` function:

```
births.std.df <- as.data.frame(births)
```

For more details about tibbles, see the Tibbles chapter in R for Data Analysis.

## 7.4 Data filtering and transformation with dplyr

`dplyr` is powerful tool for data filter and transformation. In the same way that `ggplot2` attempts to provide a “grammar of graphics”, `dplyr` aims to provide a “grammar of data manipulation”. In today’s material we will see how `dplyr` complements and simplifies standard data frame indexing and subsetting operations. However, `dplyr` is focused only on data frames and doesn’t completely replace the basic subsetting operations, and so being adept with both `dplyr` and the indexing approaches we’ve seen previously is important. If you’re curious about the name “`dplyr`”, the package’s originator Hadley Wickham says it’s supposed to invoke the idea of pliers for data frames (Github: Meaning of dplyrs name)

## 7.5 dplyr’s “verbs”

The primary functions in the `dplyr` package can be thought of as a set of “verbs”, each verb corresponding to a common data manipulation task. Some of the most frequently used verbs/functions in `dplyr` include:

- `select` – select columns
- `filter` – filter rows
- `mutate` – create new columns
- `arrange` – reorder rows
- `summarize` – summarize values
- `group_by` – split data frame on some grouping variable. Can be powerfully combined with `summarize`

All of these functions return new data frames rather than modifying the existing data frame (though some of the functions support in place modification of data frames via optional arguments). We illustrate these below by example using the NC births data.

### 7.5.1 select

The `select` function subsets the columns (variables) of a data frame. For example, to select just the weeks and weight columns from the `births` data set we could do:

```
wks.weight <- select(births, weeks, weight)
dim(wks.weight) # dim should be 50 x 2
#> [1] 150   2
head(wks.weight)
#> # A tibble: 6 x 2
#>   weeks weight
#>   <int>  <dbl>
#> 1    39   6.88
#> 2    39   7.69
#> 3    40   8.88
#> 4    40    9
#> 5    40   7.94
#> 6    40   8.25
```

The equivalent using standard indexing would be:

```
wks.wt.alt <- births[c("weeks", "weight")]
dim(wks.wt.alt)
#> [1] 150  2
head(wks.wt.alt)
#> # A tibble: 6 x 2
#>   weeks weight
#>   <dbl>   <dbl>
#> 1     39    6.88
#> 2     39    7.69
#> 3     40    8.88
#> 4     40     9
#> 5     40    7.94
#> 6     40    8.25
```

#### Notes:

\* The first argument to all of the `dplyr` functions is the data frame you're operating on

- When using functions defined in `dplyr` and `ggplot2` variable names are (usually) not quoted or used with the `$` operator. This is a design feature of these libraries and makes it easier to carry out interactive analyses because it saves a fair amount of typing.

### 7.5.2 filter

The `filter` function returns those rows of the data set that meet the given logical criterion.

For example, to get all the premature babies in the data set we could use `filter` as so:

```
premies <- filter(births, premature == "premie")
dim(premies)
#> [1] 21  9
```

The equivalent using standard indexing would be:

```
premies.alt <- births[births$premature == "premie",]
```

The `filter` function will work with more than one logical argument, and these are joined together using Boolean AND logic (i.e. intersection). For example, to find those babies that were premature *and* whose mothers were smokers we could do:

```
smoking.premies <- filter(births, premature == "premie", smoke == "smoker")
```

The equivalent call using standard indexing is:

```
# don't forget the trailing comma to indicate rows!
smoking.premies.alt <- births[(births$premature == "premie") & (births$smoke == "smoker"),]
```

`filter` also accepts logical statements chained together using the standard Boolean operators. For example, to find babies who were premature *or* whose moms were older than 35 you could use the OR operator `|`:

```
premies.or.oldmom <- filter(births, premature == "premie" | fAge > 35)
```

### 7.5.3 mutate

The `mutate` function creates a new data frame that is the same as input data frame but with additional variables (columns) as specified by the function arguments. In the example below, I create two new variables, `weight.in.kg` and a `mom.smoked`:

```
# to make code more readable it's sometime useful to spread out
# function arguments over multiple lines like I've done here
births.plus <- mutate(births,
                      weight.in.kg = weight / 2.2,
                      mom.smoked = (smoke == "smoker"))

head(births.plus)
#> # A tibble: 6 x 11
#>   fAge  mAge weeks premature visits gained weight sexBaby smoke
#>   <int> <int> <int> <chr>     <int> <int> <dbl> <chr>    <chr>
#> 1   31    30    39 full term     13     1  6.88 male    smok-
#> 2   34    36    39 full term      5    35  7.69 male    nons-
#> 3   36    35    40 full term     12    29  8.88 male    nons-
#> 4   41    40    40 full term     13    30    9 female  nons-
#> 5   42    37    40 full term    NA    10  7.94 male    nons-
#> 6   37    28    40 full term     12    35  8.25 male    smok-
#> # ... with 2 more variables: weight.in.kg <dbl>, mom.smoked <lgl>
```

The equivalent using standard indexing would be to create a new data frame from `births`, appending the new variables to the end as so:

```
births.plus.alt <- data.frame(births,
                               weight.in.kg = births$weight / 2.2,
                               mom.smoked = (births$smoke == "smoker"))
```

#### 7.5.4 arrange

`arrange` creates a new data frame where the rows are sorted according to their values for one or more variables. For example, to sort by mothers age we could do:

```
young.moms.first <- arrange(births, mAge)
head(young.moms.first)
#> # A tibble: 6 x 9
#>   fAge  mAge weeks premature visits gained weight sexBaby smoke
#>   <int> <int> <int> <chr>     <int> <int> <dbl> <chr>    <chr>
#> 1   18    15    37 full term     12    76  8.44 male    nonsmoker
#> 2   NA    16    40 full term      4    12    6 female  nonsmoker
#> 3   21    16    38 full term     15    75  7.56 female smoker
#> 4   26    17    38 full term     11    30  9.5  female nonsmoker
#> 5   17    17    29 premie       4    10  2.63 female nonsmoker
#> 6   20    17    40 full term     17    38  7.19 male   nonsmoker
```

The equivalent to `arrange` using standard indexing would be to use the information returned by the `order` function:

```
young.moms.first.alt <- births[order(births$mAge),]
head(young.moms.first.alt)
#> # A tibble: 6 x 9
#>   fAge  mAge weeks premature visits gained weight sexBaby smoke
#>   <int> <int> <int> <chr>     <int> <int> <dbl> <chr>    <chr>
#> 1   18    15    37 full term     12    76  8.44 male    nonsmoker
#> 2   NA    16    40 full term      4    12    6 female  nonsmoker
#> 3   21    16    38 full term     15    75  7.56 female smoker
#> 4   26    17    38 full term     11    30  9.5  female nonsmoker
```

```
#> 5   17   17   29 premie    4   10  2.63 female nonsmoker
#> 6   20   17   40 full term  17   38  7.19 male  nonsmoker
```

When using `arrange`, multiple sorting variables can be specified:

```
sorted.by.moms.and.dads <- arrange(births, mAge, fAge)
head(sorted.by.moms.and.dads)
#> # A tibble: 6 x 9
#>   fAge  mAge weeks premature visits gained weight sexBaby smoke
#>   <int> <int> <int> <chr>     <int> <int> <dbl> <chr>   <chr>
#> 1   18    15    37 full term    12    76  8.44 male  nonsmoker
#> 2   21    16    38 full term    15    75  7.56 female smoker
#> 3   NA    16    40 full term    4     12   6   female nonsmoker
#> 4   17    17    29 premie      4     10  2.63 female nonsmoker
#> 5   20    17    40 full term    17    38  7.19 male  nonsmoker
#> 6   26    17    38 full term    11    30  9.5   female nonsmoker
```

If you want to sort in descending order, you can combine `arrange` with the `desc` (=descend) function, also defined in `dplyr`:

```
old.moms.first <- arrange(births, desc(mAge))
head(old.moms.first)
#> # A tibble: 6 x 9
#>   fAge  mAge weeks premature visits gained weight sexBaby smoke
#>   <int> <int> <int> <chr>     <int> <int> <dbl> <chr>   <chr>
#> 1   NA    41    33 premie      13     0  5.69 female nonsmoker
#> 2   41    40    40 full term    13    30   9   female nonsmoker
#> 3   33    40    36 premie      13    23  7.81 female nonsmoker
#> 4   40    40    38 full term    13    38  7.31 male  nonsmoker
#> 5   46    39    38 full term    10    35  6.75 male  smoker
#> 6   NA    38    32 premie      10    16  2.19 female smoker
```

### 7.5.5 summarize

`summarize` applies a function of interest to one or more variables in a data frame, reducing a vector of values to a single value and returning the results in a data frame. This is most often used to calculate statistics like means, medians, count, etc. As we'll see below, this is powerful when combined with the `group_by` function.

```
summarize(births,
          mean.wt = mean(weight),
          median.wks = median(weeks))
#> # A tibble: 1 x 2
#>   mean.wt median.wks
#>   <dbl>      <dbl>
#> 1  7.046       39
```

You'll need to be diligent if your data has missing values (NAs). For example, by default the `mean` function returns NA if any of the input values are NA:

```
summarize(births,
          mean.gained = mean(gained))
#> # A tibble: 1 x 1
#>   mean.gained
#>   <dbl>
#> 1        NA
```

However, if you read the `mean` docs (`?mean`) you'll see that there is an `na.rm` argument that indicates whether NA values should be removed before computing the mean. This is what we want so we instead call summarize as follows:

```
summarize(births,
          mean.gained = mean(gained, na.rm = TRUE))
#> # A tibble: 1 x 1
#>   mean.gained
#>   <dbl>
#> 1     32.45
```

### 7.5.6 group\_by

The `group_by` function implicitly adds grouping information to a data frame.

```
# group the births by whether mom smoked or not
by_smoking <- group_by(births, smoke)
```

The object returned by `group_by` is a “grouped data frame”:

```
class(by_smoking)
#> [1] "grouped_df" "tbl_df"      "tbl"        "data.frame"
```

Some functions, like `count()` and `summarize()` (see below) know how to use the grouping information. For example, to count the number of births conditional on mother smoking status we could do:

```
count(by_smoking)
#> # A tibble: 2 x 2
#> # Groups:   smoke [2]
#>   smoke      n
#>   <chr>    <int>
#> 1 nonsmoker  100
#> 2 smoker      50
```

`group_by` also works with multiple grouping variables, with each added grouping variable specified as an additional argument:

```
by_smoking.and.mAge <- group_by(births, smoke, mAge > 35)
```

### 7.5.7 Combining grouping and summarizing

Grouped data frames can be combined with the `summarize` function we saw above. For example, if we wanted to calculate mean birth weight, broken down by whether the baby's mother smoked or not we could call `summarize` with our `by_smoking` grouped data frame:

```
summarize(by_smoking, mean.wt = mean(weight))
#> # A tibble: 2 x 2
#>   smoke      mean.wt
#>   <chr>    <dbl>
#> 1 nonsmoker  7.180
#> 2 smoker      6.779
```

Similarly to get the mean birth weight of children conditioned on mothers smoking status and age:

```
summarize(by_smoking.and.mAge, mean(weight))
#> # A tibble: 4 x 3
```

```
#> # Groups:   smoke [?]
#>   smoke      `mAge > 35` `mean(weight)`
#>   <chr>      <lgl>           <dbl>
#> 1 nonsmoker FALSE            7.171
#> 2 nonsmoker TRUE             7.258
#> 3 smoker     FALSE            6.832
#> 4 smoker     TRUE             6.302
```

### 7.5.8 Scoped variants of `mutate` and `summarize`

Both the `mutate()` and `summarize()` functions provide “scoped” alternatives, that allow us to apply the operation on a selection of variables. These variants are often used in combination with grouping. We’ll look at the `summarize` versions – `summarize_all()`, `summarize_at()`, and `summarize_if()`. See the documentation (`?summarize_all`) for descriptions of the `mutate` versions.

#### 7.5.8.1 `summarize_all()`

`summarize_all()` applies a one or more functions to all columns in a data frame. Here we illustrate a simple version of this with the `iris` data:

```
# group by species
by_species <- group_by(iris, Species)
# calculate the mean of every variable, grouped by species
summarize_all(by_species, mean)
#> # A tibble: 3 x 5
#>   Species   Sepal.Length Sepal.Width Petal.Length Petal.Width
#>   <fct>       <dbl>        <dbl>        <dbl>        <dbl>
#> 1 setosa      5.006       3.428       1.462       0.246
#> 2 versicolor  5.936       2.77        4.26        1.326
#> 3 virginica   6.588       2.974       5.552       2.026
```

Note that if we try and apply `summarize_all()` in the same way to the grouped data frame `by_smoking` we’ll get a bunch of warning messages:

```
summarize_all(by_smoking, mean)
#> # A tibble: 2 x 9
#>   smoke      fAge  mAge weeks premature visits gained weight sexBaby
#>   <chr>      <dbl> <dbl> <dbl>       <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 nonsmoker    NA  26.9  38.55       NA    NA    NA  7.180    NA
#> 2 smoker       NA   26    38.54       NA   10.8   NA  6.779    NA
```

Here’s an example of one of these warnings:

```
Warning messages:
1: In mean.default(premature) :
  argument is not numeric or logical: returning NA
```

This message is telling us that we can’t apply the `mean()` function to the data frame column `premature` because this is not a numerical or logical vector. Despite this and the other similar warnings, `summarize_all()` does return a result, but the means for any non-numeric values are replaced with NAs, as shown below:

```
# A tibble: 2 x 9
  smoke      fAge  mAge weeks premature visits gained weight sexBaby
  <chr>      <dbl> <dbl> <dbl>       <dbl> <dbl> <dbl> <dbl>
1 nonsmoker    NA  26.9  38.6       NA    NA    NA  7.18      NA
```

```
2 smoker      NA  26.0  38.5      NA   10.8      NA   6.78      NA
```

If you examine the output above, you'll see that there are several variables that are numeric, however we still got NAs when we calculated the grouped means. This is because those variables contain NA values. The `mean` function has an optional argument, `na.rm`, which tells the function to remove any missing data before calculating the mean. Thus we can modify our call to `summarize_all` as follows:

```
# calculate mean of all variables, grouped by smoking status
summarize_all(by_smoking, mean, na.rm = TRUE)
#> # A tibble: 2 x 9
#>   smoke      fAge  mAge weeks premature visits gained weight sexBaby
#>   <chr>     <dbl> <dbl> <dbl>    <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 nonsmoker 29.81  26.9  38.55     NA   11.86  32.55  7.180   NA
#> 2 smoker     29.71   26    38.54     NA   10.8   32.27  6.779   NA
```

Note that the non-numeric data columns still lead to NA values.

### 7.5.8.2 summarize\_if()

`summarize_if()` is similar to `summarize_all()`, except it only applies the function of interest to those variables that match a particular predicate (i.e. are TRUE for a particular TRUE/FALSE test).

Here we use `summarize_if()` to apply the `mean()` function to only those variables (columns) that are numeric.

```
# calculate mean of all numeric variables, grouped by smoking status
summarize_if(by_smoking, is.numeric, mean, na.rm = TRUE)
#> # A tibble: 2 x 7
#>   smoke      fAge  mAge weeks visits gained weight
#>   <chr>     <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 nonsmoker 29.81  26.9  38.55 11.86  32.55  7.180
#> 2 smoker     29.71   26    38.54 10.8   32.27  6.779
```

### 7.5.8.3 summarize\_at()

`summarize_at()` allows us to apply functions of interest only to specific variables.

```
# calculate mean of gained and weight variables, grouped by smoking status
summarize_at(by_smoking, c("gained", "weight"), mean, na.rm = TRUE)
#> # A tibble: 2 x 3
#>   smoke      gained weight
#>   <chr>     <dbl> <dbl>
#> 1 nonsmoker 32.55  7.180
#> 2 smoker     32.27  6.779
```

All three of the scoped summarize functions can also be used to apply multiple functions, by wrapping the function names in a call to `dplyr::funs()`:

```
# calculate mean and std deviation of
# gained and weight variables, grouped by smoking status
summarize_at(by_smoking, c("gained", "weight"), funs(mean, sd), na.rm = TRUE)
#> # A tibble: 2 x 5
#>   smoke      gained_mean weight_mean gained_sd weight_sd
#>   <chr>     <dbl>       <dbl>      <dbl>      <dbl>
#> 1 nonsmoker 32.55       7.180     15.23     1.434
#> 2 smoker     32.27       6.779     16.65     1.597
```

`summarize_at()` accepts as the argument for variables a character vector of column names, a numeric vector of column positions, or a list of columns generated by the `dplyr::vars()` function, which can be used as so:

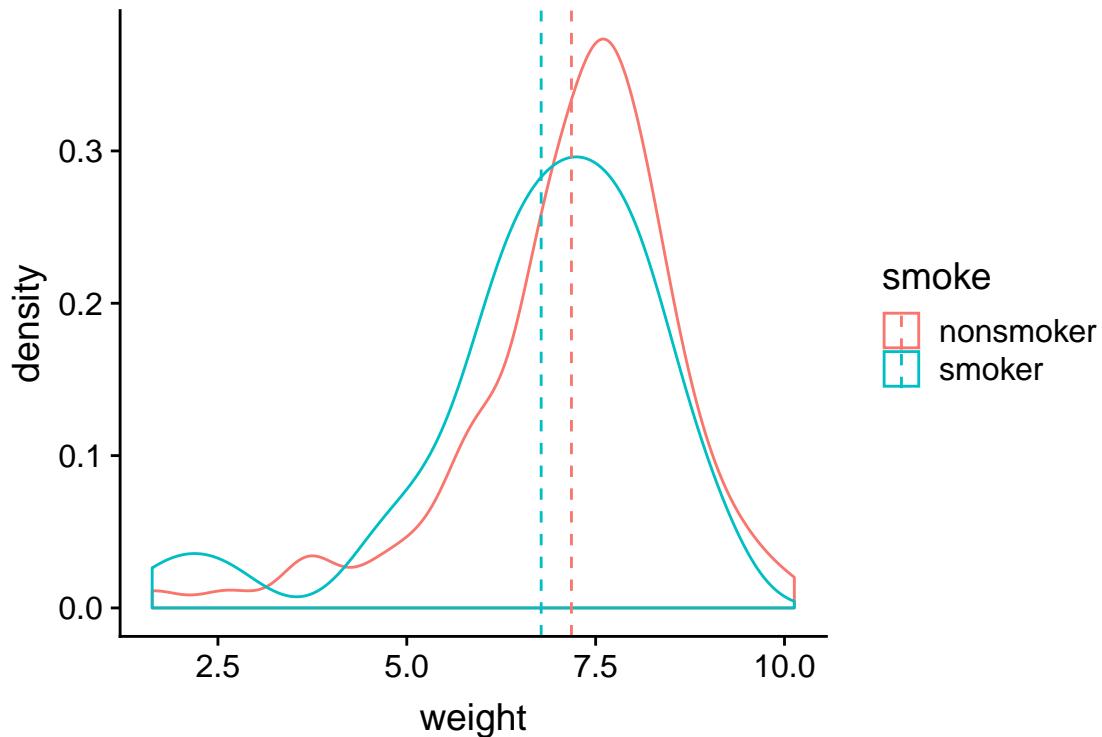
```
# reformatting to promote readability of arguments
summarize_at(by_smoking,
             vars(gained, weight),
             funs(mean, sd),
             na.rm = TRUE)
#> # A tibble: 2 x 5
#>   smoke    gained_mean weight_mean gained_sd weight_sd
#>   <chr>      <dbl>       <dbl>      <dbl>      <dbl>
#> 1 nonsmoker  32.55      7.180     15.23     1.434
#> 2 smoker     32.27      6.779     16.65     1.597
```

### 7.5.9 Combining summarize with grouping aesthetics in ggplot2

We've already seen an instance of grouping (conditioning) when we used aesthetics like color or fill to distinguish subgroups in different types of statistical graphics. Below is an example where we integrate information from a `group_by/summarize` operation into a plot:

```
# calculate mean weights, conditioned on smoking status
wt.by.smoking <- 
  summarize(by_smoking, mean_weight = mean(weight, na.rm = TRUE))

# create density plot for all the data
# and then use geom_vline to draw vertical lines at the means for
# each group
ggplot(births) +
  geom_density(aes(x = weight, color = smoke)) + # data drawn from births
  geom_vline(data = wt.by.smoking, # note use of different data frame!
             mapping = aes(xintercept = mean_weight, color = smoke),
             linetype = 'dashed')
```



## 7.6 Pipes

`dplyr` includes a very useful operator available called a pipe available to us. Pipes are powerful because they allow us to chain together sets of operations in a very intuitive fashion while minimizing nested function calls. We can think of pipes as taking the output of one function and feeding it as the *first argument* to another function call, where we've already specified the subsequent arguments.

Pipes are actually defined in another package called `magrittr`. We'll look at the basic pipe operator and then look at a few additional “special” pipes that `magrittr` provides.

### 7.6.1 Install and load `magrittr`

In `magrittr` is not already installed, install it via the command line or the RStudio GUI. Having done so, you will need to load `magrittr` via the `library()` function:

```
library(magrittr)
```

### 7.6.2 The basic pipe operator

The pipe operator is designated by `%>%`. Using pipes, the expression `x %>% f()` is equivalent to `f(x)` and the expression `x %>% f(y)` is equivalent to `f(x,y)`. The documentation on pipes (see `?magrittr`) uses the notation `lhs %>% rhs` where `lhs` and `rhs` are short for “left-hand side” and “right-hand side” respectively. I'll use this same notation in some of the explanations that follow.

```
births %>% head()    # same as head(births)
#> # A tibble: 6 x 9
#>   fAge  mAge weeks premature visits gained weight sexBaby smoke
```

```
#>   <int> <int> <int> <chr>      <int> <int> <dbl> <chr> <chr>
#> 1   31    30    39 full term     13     1  6.88 male   smoker
#> 2   34    36    39 full term     5     35  7.69 male   nonsmoker
#> 3   36    35    40 full term     12     29  8.88 male   nonsmoker
#> 4   41    40    40 full term     13     30  9   female nonsmoker
#> 5   42    37    40 full term     NA     10  7.94 male   nonsmoker
#> 6   37    28    40 full term     12     35  8.25 male   smoker
births %>% head # you can even leave the parentheses out
#> # A tibble: 6 x 9
#>   fAge mAge weeks premature visits gained weight sexBaby smoke
#>   <int> <int> <int> <chr>      <int> <int> <dbl> <chr> <chr>
#> 1   31    30    39 full term     13     1  6.88 male   smoker
#> 2   34    36    39 full term     5     35  7.69 male   nonsmoker
#> 3   36    35    40 full term     12     29  8.88 male   nonsmoker
#> 4   41    40    40 full term     13     30  9   female nonsmoker
#> 5   42    37    40 full term     NA     10  7.94 male   nonsmoker
#> 6   37    28    40 full term     12     35  8.25 male   smoker
births %>% head(10) # same as head(births, 10)
#> # A tibble: 10 x 9
#>   fAge mAge weeks premature visits gained weight sexBaby smoke
#>   <int> <int> <int> <chr>      <int> <int> <dbl> <chr> <chr>
#> 1   31    30    39 full term     13     1  6.88 male   smoker
#> 2   34    36    39 full term     5     35  7.69 male   nonsmoker
#> 3   36    35    40 full term     12     29  8.88 male   nonsmoker
#> 4   41    40    40 full term     13     30  9   female nonsmoker
#> 5   42    37    40 full term     NA     10  7.94 male   nonsmoker
#> 6   37    28    40 full term     12     35  8.25 male   smoker
#> 7   35    35    28 premie      6     29  1.63 female nonsmoker
#> 8   28    21    35 premie      9     15  5.5   female smoker
#> 9   22    20    32 premie      5     40  2.69 male   smoker
#> 10  36    25    40 full term     13     34  8.75 female nonsmoker
```

Multiple pipes can be chained together, such that `x %>% f() %>% g() %>% h()` is equivalent to `h(g(f(x)))`.

```
# equivalent to: head(arrange(births, weight), 10)
births %>% arrange(weight) %>% head(10)
#> # A tibble: 10 x 9
#>   fAge mAge weeks premature visits gained weight sexBaby smoke
#>   <int> <int> <int> <chr>      <int> <int> <dbl> <chr> <chr>
#> 1   35    35    28 premie      6     29  1.63 female nonsmoker
#> 2   NA    18    33 premie      7     40  1.69 male   smoker
#> 3   NA    38    32 premie      10    16  2.19 female smoker
#> 4   17    17    29 premie      4     10  2.63 female nonsmoker
#> 5   22    20    32 premie      5     40  2.69 male   smoker
#> 6   38    37    26 premie      5     25  3.63 male   nonsmoker
#> 7   25    22    34 premie      10    20  3.75 male   nonsmoker
#> 8   NA    24    38 full term   16    50  3.75 female nonsmoker
#> 9   30    25    35 premie      15    40  4.5   male   smoker
#> 10  19    20    34 premie      13     6  4.5   male   nonsmoker
```

When there are multiple piping operations, I like to arrange the statements vertically to help emphasize the flow of processing and to facilitate debugging and/or modification. I would usually rearrange the above code block as follows:

```

births %>%
  arrange(weight) %>%
  head(10)
#> # A tibble: 10 x 9
#>   fAge  mAge weeks premature visits gained weight sexBaby smoke
#>   <int> <int> <int> <chr>     <int> <int> <dbl> <chr>   <chr>
#> 1   35    35    28 premie      6    29  1.63 female nonsmoker
#> 2   NA    18    33 premie      7    40  1.69 male   smoker
#> 3   NA    38    32 premie     10    16  2.19 female smoker
#> 4   17    17    29 premie      4    10  2.63 female nonsmoker
#> 5   22    20    32 premie      5    40  2.69 male   smoker
#> 6   38    37    26 premie      5    25  3.63 male   nonsmoker
#> 7   25    22    34 premie     10    20  3.75 male   nonsmoker
#> 8   NA    24    38 full term  16    50  3.75 female nonsmoker
#> 9   30    25    35 premie     15    40  4.5   male   smoker
#> 10  19    20    34 premie     13     6  4.5   male   nonsmoker

```

### 7.6.3 An example without pipes

To illustrate how pipes help us, first let's look at an example set of analysis steps without using pipes. Let's say we wanted to explore the relationship between father's age and baby's birth weight. We'll start this process of exploration by generating a bivariate scatter plot. Being good scientists we want to express our data in SI units, so we'll need to converts pounds to kilograms. You'll also recall that a number of the cases have missing data on father's age, so we'll want to remove those before we plot them. Here's how we might accomplish these steps:

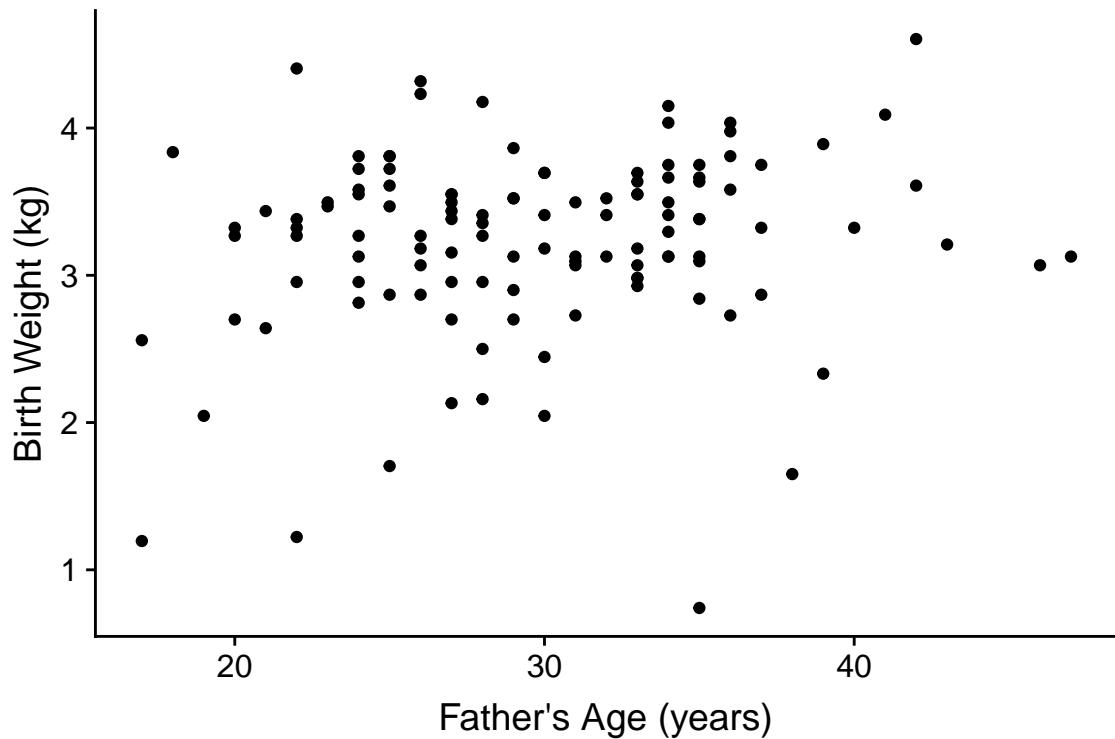
```

# add a new column for weight in kg
births.kg <- mutate(births, weight.kg = weight / 2.2)

# filter out the NA fathers
filtered.births <- filter(births.kg, !is.na(fAge))

# create our plot
ggplot(filtered.births, aes(x = fAge, y = weight.kg)) +
  geom_point() +
  labs(x = "Father's Age (years)", y = "Birth Weight (kg)")

```



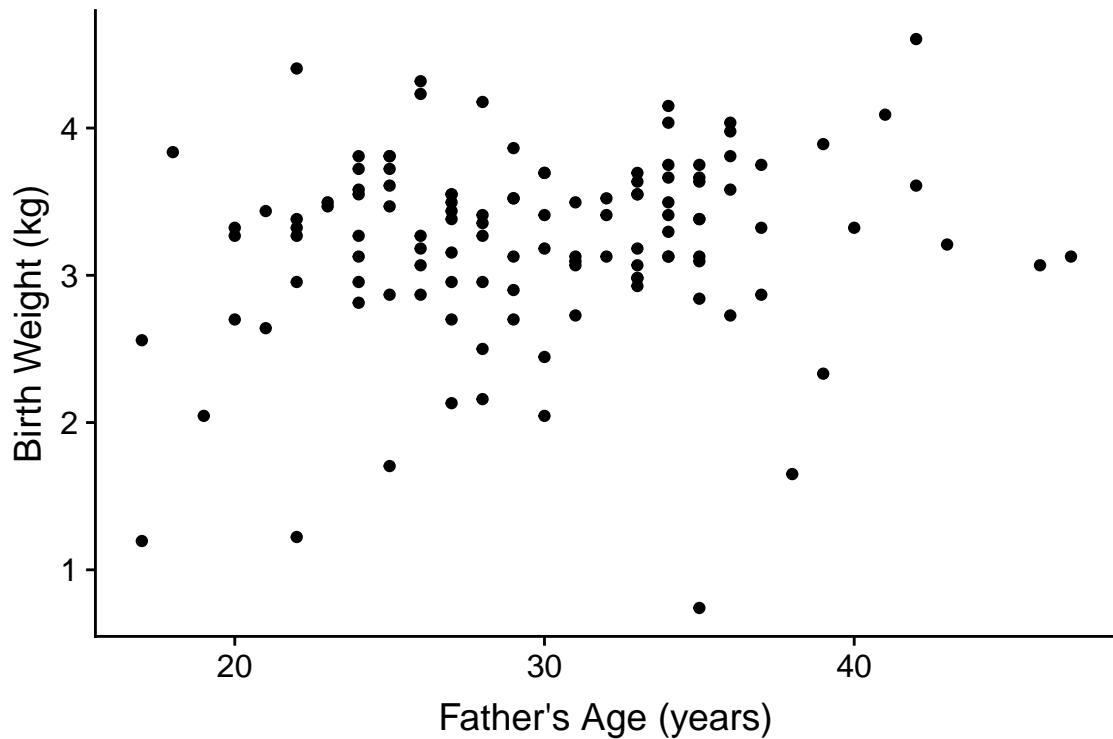
Notice that we created two “temporary” data frames along the way – `births.kg` and `filtered.births`. These probably aren’t of particular interest to us, but we needed to generate them to build the plot we wanted. If you were particularly masochistic you could avoid these temporary data frames by using nested functions call like this:

```
# You SHOULD NOT write nested code like this.
# Code like this is hard to debug and understand!
ggplot(filter(mutate(births, weight.kg = weight / 2.2), !is.na(fAge)),
       aes(x = fAge, y = weight.kg)) +
  geom_point() +
  labs(x = "Father's Age (years)", y = "Birth Weight (kg)")
```

#### 7.6.4 The same example using pipes

The pipe operator makes the output of one statement (`lhs`) as the first input of a following function (`rhs`). This simplifies the above example to:

```
births %>%
  mutate(weight.kg = weight / 2.2) %>%
  filter(!is.na(fAge)) %>%
  ggplot(aes(x = fAge, y = weight.kg)) +
  geom_point() +
  labs(x = "Father's Age (years)", y = "Birth Weight (kg)")
```



In the example above, we feed the data frame into the `mutate` function. `mutate` expects a data frame as a first argument, and subsequent arguments specify the new variables to be created. `births %>% mutate(weight.kg = weight / 2.2)` is thus equivalent to `mutate(births, weight.kg = weight / 2.2)`. We then pipe the output to `filter`, removing NA fathers, and then pipe that output as the input to `ggplot`.

As mentioned previously, it's good coding style to write each discrete step as its own line when using piping. This make it easier to understand what the steps of the analysis are as well as facilitating changes to the code (commenting out lines, adding lines, etc)

### 7.6.5 Assigning the output of a statement involving pipes to a variable

It's important to recognize that pipes are simply a convenient way to chain together a series of expression. Just like any other compound expression, the output of a series of pipe statements can be assigned to a variable, like so:

```
stats.old.moms <-
  births %>%
  filter(mAge > 35) %>%
  summarize(median.gestation = median(weeks),
            mean.weight = mean(weight))

stats.old.moms
#> # A tibble: 1 x 2
#>   median.gestation mean.weight
#>       <int>          <dbl>
#> 1           38         6.939
```

Note that our summary table, `stats.old.moms`, is itself a data frame.

### 7.6.6 Compound assignment pipe operator

A fairly common operation when working interactively in R is to update an existing data frame. `magrittr` defines another pipe operator – `%<>%` – called the “compound assignment” pipe operator, to facilitate this. The compound assignment pipe operator has the basic usage `lhs %<>% rhs`. This operator evaluates the function on the `rhs` using the `lhs` as the first argument, and *then* updates the `lhs` with the resulting value. This is simply shorthand for writing `lhs <- lhs %>% rhs`.

```
stats.old.moms %<>% # note compound pipe operator!
  mutate(mean.weight.kg = mean.weight / 2.2)
```

### 7.6.7 The dot operator with pipes

When working with pipes, sometimes you’ll want to use the `lhs` in multiple places on the `rhs`, or as something other than the first argument to the `rhs`. `magrittr` provides for this situation by using the dot (`.`) operator as a placeholder. Using the dot operator, the expression `y %>% f(x, .)` is equivalent to `f(x,y)`.

```
c("dog", "cakes", "sauce", "house") %>% # create a vector
  sample(1) %>% # pick a random single element of that vector
  str_c("hot", .) # string concatenate the pick with the word "hot"
#> [1] "hotcakes"
```

### 7.6.8 The exposition pipe operator

`magrittr` defines another operator called the “exposition pipe operator”, designed `%$%`. This operator exposes the names in the `lhs` to the expression on the `rhs`.

Here is an example of using the exposition pipe operator to simply return the vector of weights:

```
births %>%
  filter(premature == "premie") %$% # note the different pipe operator!
  weight
#> [1] 1.63 5.50 2.69 6.50 7.81 4.75 3.75 2.19 6.81 4.69 6.75 4.50 5.94 4.50
#> [15] 5.06 5.69 1.69 6.31 2.63 5.88 3.63
```

If we wanted to calculate the minimum and maximum weight of premature babies in the data set we could do the following (though I’d usually prefer `summarize()` unless I needed the results in the form of a vector):

```
births %>%
  filter(mAge > 35) %$% # note the different pipe operator!
  c(min(weight), max(weight))
#> [1] 2.19 10.13
```

# Chapter 8

## Data wrangling, Part I

In the real world you'll often create a data set (or be given one) in a format that is less than ideal for analysis. This can happen for a number of reasons. For example, the data may have been recorded in a manner convenient for collection and visual inspection, but which does not work well for analysis and plotting. Or the data may be an amalgamation of multiple experiments, in which each of the experimenters used slightly different naming conventions. Or the data may have been produced by an instrument that produces output with a fixed format. Sometimes important experimental information is included in the column headers of a spreadsheet.

Whatever the case, we often find ourselves in the situation where we need to “wrangle” our data into a “tidy” format before we can proceed with visualization and analysis. The “R for Data Science” text discusses some desirable rules for “tidy” data in order to facilitate downstream analyses. These are:

1. Each variable must have its own column.
2. Each observation must have its own row.
3. Each value must have its own cell.

In this lecture we're going to walk through an extended example of wrangling some data into a “tidy” format.

### 8.1 Libraries

```
library(magrittr)
library(stringr)
library(tidyverse)
library(cowplot)
```

### 8.2 Data

To illustrate a data wrangling pipeline, we're going to use a gene expression microarray data set, based on the following paper:

- Spellman PT, et al. 1998. Comprehensive identification of cell cycle-regulated genes of the yeast *Saccharomyces cerevisiae* by microarray hybridization. Mol Biol Cell 9(12): 3273-97.

In this paper, Spellman and colleagues tried to identify all the genes in the yeast genome (>6000 genes) that exhibited oscillatory behaviors suggestive of cell cycle regulation. To do so, they combined gene expression measurements from six different types of cell cycle synchronization experiments.

Download the Spellman data to your filesystem from this link (right-click the “Download” button and save to your Downloads folder or similar).

I suggest that once you download the data, you open it in a spreadsheet program (e.g. Excel) or use the RStudio Data Viewer to get a sense of what the data looks like.

Let’s load it into R, using the `read_tsv()` function, using the appropriate file path.

```
# the filepath may differ on your computer
spellman <- read_tsv("~/Downloads/spellman-combined.txt")
#> Parsed with column specification:
#> cols(
#>   .default = col_double(),
#>   X1 = col_character(),
#>   clb = col_character(),
#>   alpha = col_character(),
#>   cdc15 = col_character(),
#>   cdc28 = col_character(),
#>   elu = col_character()
#> )
#> See spec(...) for full column specifications.
```

The initial dimensions of the data frame are:

```
dim(spellman)
#> [1] 6178 83
```

The six types of cell cycle synchronization experiments included in this data set are:

1. synchronization by alpha-factor = “alpha”
2. synchronization by cdc15 temperature sensitive mutants = “cdc15”
3. synchronization by cdc28 temperature sensitive mutants = “cdc28”
4. synchronization by elutration = “elu”
5. synchronization by cln3 mutant strains = “cln3”
6. synchronization by clb2 mutant strains = “clb2”

## 8.3 Renaming data frame columns

Notice that when we imported the data we got a warning message: `Missing column names filled in: 'X1' [1]`. In a data frame, every column must have a name. The first column of our data set did not have a name in the header, so `read_tsv` automatically gave it the name `X1`.

Our first task is to give the first column a more meaningful name. This column gives “systematic gene names” – a standardized naming scheme for genes in the yeast genome. We’ll use `dplyr::rename` to rename `X1` to `gene`. Note that `rename` can take multiple arguments if you need to rename multiple columns simultaneously.

```
spellman.clean <-
  spellman %>%
  rename(gene = X1)
```

Note the use of the compound assignment operator – `%>%` – from the `magrittr` package, which we introduced in our last class session.

## 8.4 Dropping unneeded columns

Take a look at the Spellman data again in your spreadsheet program (or the RStudio data viewer). You'll notice there are some blank columns. For example there is a column with the header "alpha" that has no entries. These are simply visual organizing elements that the creator of the spreadsheet added to separate the different experiments that are included in the data set.

We can use `dplyr::select()` to drop columns by prepending column names with the negative sign:

```
# drop the alpha column keeping all others
spellman.clean %<%
  select(-alpha)
```

Note that usually `select()` keeps only the variables you specify. However if the first expression is negative, `select` will instead automatically keep all variables, dropping only those you specify.

### 8.4.1 Finding all empty columns

In the example above, we looked at the data and saw that the "alpha" column was empty, and thus dropped it. This worked because there are only a modest number of columns in the data frame in it's initial form. However, if our data frame contained thousands of columns, this "look and see" procedure would not be efficient. Can we come up with a general solution for removing empty columns from a data frame?

When you load a data frame from a spreadsheet, empty cells are given the value `NA`. In previous class sessions we were introduced to the function `is.na()` which tests each value in a vector or data frame for whether it's `NA` or not. We can count `NA` values in a vector by summing the output of `is.na()`. Conversely we can count the number of "not `NA`" items by using the negation operator (`!`):

```
# count number of NA values in the alpha0 column
sum(is.na(spellman$alpha0))
#> [1] 165

# count number of values that are NOT NA in alpha0
sum(!is.na(spellman$alpha0))
#> [1] 6013
```

This seems like it should get us close to a solution but `sum(is.na(...))` when applied to a data frame counts `NAs` across the entire data frame, not column-by-column.

```
# doesn't do what we hoped!
sum(is.na(spellman))
#> [1] 59017
```

If we want sums of `NAs` by column, we instead use the `colSums()` function:

```
# get number of NAs by column
colSums(is.na(spellman))
#>      X1    cln3-1    cln3-2      clb    clb2-2    clb2-1      alpha
#>      0      193      365     6178      454      142     6178
#> alpha0  alpha7  alpha14  alpha21  alpha28  alpha35  alpha42
#>     165      525      191     312      267      207      123
#> alpha49  alpha56  alpha63  alpha70  alpha77  alpha84  alpha91
#>     257      147      186     185      178      155      329
#> alpha98  alpha105 alpha112 alpha119    cdc15   cdc15_10  cdc15_30
#>     209      174      222     251     6178      677      477
#> cdc15_50 cdc15_70 cdc15_80 cdc15_90 cdc15_100 cdc15_110 cdc15_120
#>     501      608      573      562      606      570      611
```

```
#> cdc15_130 cdc15_140 cdc15_150 cdc15_160 cdc15_170 cdc15_180 cdc15_190
#>     495      574      811      583      571      803      613
#> cdc15_200 cdc15_210 cdc15_220 cdc15_230 cdc15_240 cdc15_250 cdc15_270
#>    1014      573      741      596      847      379      537
#> cdc15_290    cdc28    cdc28_0   cdc28_10   cdc28_20   cdc28_30   cdc28_40
#>     426     6178      122       72       67       55       66
#> cdc28_50  cdc28_60  cdc28_70  cdc28_80  cdc28_90  cdc28_100 cdc28_110
#>     56       82       84       75      237      165      319
#> cdc28_120  cdc28_130  cdc28_140  cdc28_150  cdc28_160        elu      elu0
#>     312     1439     2159      521      543     6178      122
#> elu30    elu60    elu90    elu120    elu150    elu180    elu210
#>    153     175     132      103      119      111      118
#> elu240    elu270    elu300    elu330    elu360    elu390
#>    131     110     112      112      156      114
```

Columns with *all* missing values can be more conveniently found by asking for those columns where the number of “not missing” values is zero:

```
# get names of all columns for which all rows are NA
# using standard indexing
names(spellman)[colSums(!is.na(spellman)) == 0]
#> [1] "clb"    "alpha"  "cdc15"  "cdc28"  "elu"
```

We can combine the `colSums(!is.na())` idiom with the `dplyr::select_if` function to quickly remove all empty columns as so:

```
spellman.clean %<>%
  # keep ONLY the non-empty columns
  select_if(colSums(!is.na(.)) > 0)
```

## 8.4.2 Dropping columns by matching names

Only two time points from the cln3 and clb2 experiments were reported in the original publication. Since complete time series are unavailable for these two experimental conditions we will drop them from further consideration.

`select()` can be called be called with a number of “helper function” (`?select_helpers`). Here we’ll illustrate the `matches()` helper function which matches column names to a “regular expression”. Regular expressions (also referred to as “regex” or “regexp”) are a way of specifying patterns in strings. For the purposes of this document we’ll illustrate regexs by example; for a more detailed explanation of regular expressions see the the regex help(`?regex`) and the Chapter on Strings in “R for Data Analysis”:

Let’s see how to drop all the “cln3” and “clb2” columns from the data frame using `matches()`:

```
spellman.clean %<>%
  select(-matches("cln3"))
  select(-matches("clb2"))
```

If we wanted we could have collapsed our two match statements into one as follows:

```
spellman.clean %<>%
  select(-matches("cln3|clb2"))
```

In this second example, the character “|” is specifying an OR match within the regular expression, so this regular expression matches column names that contain “cln3” OR “clb2”.

## 8.5 Merging data frames

Often you'll find yourself in the situation where you want to combine information from multiple data sources. The usual requirement is that the data sources have one or more shared columns, that allow you to relate the entities or observations (rows) between the data sets. `dplyr` provides a variety of `join` functions to handle different data merging operators.

To illustrating merging or joining data sources, we'll add information about each genes "common name" and a description of the gene functions to our Spellman data set. I've prepared a file with this info based on info I downloaded from the Saccharomyces Genome Database.

```
gene.info <- read_csv("https://github.com/bio304-class/bio304-course-notes/raw/master/datasets/yeast-ORFinfo.csv")
#> Parsed with column specification:
#> cols(
#>   ftr.name = col_character(),
#>   std.name = col_character(),
#>   description = col_character()
#> )
```

Having loaded the data, let's get a quick overview of it's structure:

```
names(gene.info)
#> [1] "ftr.name"      "std.name"      "description"
dim(gene.info)
#> [1] 6610      3
head(gene.info)
#> # A tibble: 6 x 3
#>   ftr.name std.name description
#>   <chr>     <chr>     <chr>
#> 1 YAL069W  <NA>      Dubious open reading frame; unlikely to encode a fun-
#> 2 YAL068W-A <NA>      Dubious open reading frame; unlikely to encode a fun-
#> 3 YAL068C   PAU8       Protein of unknown function; member of the seripaupe-
#> 4 YAL067W-A <NA>      Putative protein of unknown function; identified by ~
#> 5 YAL067C   SEO1       Putative permease; member of the allantoate transpor-
#> 6 YAL066W  <NA>      Dubious open reading frame; unlikely to encode a fun-
```

In `gene.info`, the `ftr.name` column corresponds to the `gene` column in our Spellman data set. The `std.name` column gives the "common" gene name (not every gene has a common name so there are lots of NAs). The `description` column gives a brief textual description of what the gene product does.

To combine `spellmean.clean` with `gene.info` we use the `left_join` function defined in `dplyr`. As noted in the description of the function, `left_join(x, y)` returns "all rows from x, and all columns from x and y. Rows in x with no match in y will have NA values in the new columns." In addition, we have to specify the column to join by using the `by` argument to `left_join`.

```
spellman.merged <-
  left_join(spellman.clean, gene.info, by = c("gene" = "ftr.name"))
```

By default, the joined columns are merged at the end of the data frame, so we'll reorder variables to bring the `std.name` and `description` to the second and thirds columns, preserving the order of all the other columns.

```
spellman.merged %<>%
  select(gene, std.name, description, everything())

spellman.merged
#> # A tibble: 6,178 x 76
#>   gene  std.name description alpha0 alpha7 alpha14 alpha21 alpha28 alpha35
```

```
#>   <chr> <chr>   <chr>      <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
#> 1 YALO~ TFC3 Subunit of~ -0.15 -0.15 -0.21  0.17 -0.42 -0.44
#> 2 YALO~ VPS8 Membrane-b~ -0.11  0.1   0.01  0.06  0.04 -0.26
#> 3 YALO~ EFB1 Translatio~ -0.14 -0.71  0.1   -0.32 -0.4   -0.580
#> 4 YALO~ <NA> Dubious op~ -0.02 -0.48 -0.11  0.12 -0.03  0.19
#> 5 YALO~ SSA1 ATPase inv~ -0.05 -0.53 -0.47 -0.06  0.11 -0.07
#> 6 YALO~ ERP2 Member of ~ -0.6   -0.45 -0.13  0.35 -0.01  0.49
#> 7 YALO~ FUN14 Integral m~ -0.28 -0.22 -0.06  0.22  0.25  0.13
#> 8 YALO~ SPO7 Putative r~ -0.03 -0.27  0.17 -0.12 -0.27  0.06
#> 9 YALO~ MDM10 Subunit of~ -0.05  0.13  0.13 -0.21 -0.45 -0.21
#> 10 YALO~ SWC3 Protein of~ -0.31 -0.43 -0.3   -0.23 -0.13 -0.07
#> # ... with 6,168 more rows, and 67 more variables: alpha42 <dbl>,
#> #   alpha49 <dbl>, alpha56 <dbl>, alpha63 <dbl>, alpha70 <dbl>,
#> #   alpha77 <dbl>, alpha84 <dbl>, alpha91 <dbl>, alpha98 <dbl>,
#> #   alpha105 <dbl>, alpha112 <dbl>, alpha119 <dbl>, cdc15_10 <dbl>,
#> #   cdc15_30 <dbl>, cdc15_50 <dbl>, cdc15_70 <dbl>, cdc15_80 <dbl>,
#> #   cdc15_90 <dbl>, cdc15_100 <dbl>, cdc15_110 <dbl>, cdc15_120 <dbl>,
#> #   cdc15_130 <dbl>, cdc15_140 <dbl>, cdc15_150 <dbl>, cdc15_160 <dbl>,
#> #   cdc15_170 <dbl>, cdc15_180 <dbl>, cdc15_190 <dbl>, cdc15_200 <dbl>,
#> #   cdc15_210 <dbl>, cdc15_220 <dbl>, cdc15_230 <dbl>, cdc15_240 <dbl>,
#> #   cdc15_250 <dbl>, cdc15_270 <dbl>, cdc15_290 <dbl>, cdc28_0 <dbl>,
#> #   cdc28_10 <dbl>, cdc28_20 <dbl>, cdc28_30 <dbl>, cdc28_40 <dbl>,
#> #   cdc28_50 <dbl>, cdc28_60 <dbl>, cdc28_70 <dbl>, cdc28_80 <dbl>,
#> #   cdc28_90 <dbl>, cdc28_100 <dbl>, cdc28_110 <dbl>, cdc28_120 <dbl>,
#> #   cdc28_130 <dbl>, cdc28_140 <dbl>, cdc28_150 <dbl>, cdc28_160 <dbl>,
#> #   elu0 <dbl>, elu30 <dbl>, elu60 <dbl>, elu90 <dbl>, elu120 <dbl>,
#> #   elu150 <dbl>, elu180 <dbl>, elu210 <dbl>, elu240 <dbl>, elu270 <dbl>,
#> #   elu300 <dbl>, elu330 <dbl>, elu360 <dbl>, elu390 <dbl>
```

## 8.6 Reshaping data with `tidyverse`

The `tidyverse` package provides functions for reshaping or tidying data frames. `tidyverse` is yet another component of the `tidyverse`, and thus was loaded by the `library(tidyverse)`.

We're going to look at two functions `tidyverse::gather()` and `tidyverse::extract()`, and how they can be combined with now familiar `dplyr` functions we've seen previously. The reading assignment for today's class session covers a variety of other functions defined in `tidyverse`.

The Spellman data, as I provided it to you, is in what we would call "wide" format. Each column (besides the `gene` column) corresponds to an experimental condition *and* time point. For example, "alpha0" is the alpha-factor experiment at time point 0 mins; "alpha7" is the alpha-factor experiment at time point 7 mins, etc. The cells within each column correspond to the expression of a corresponding gene (given by the first column which we renamed `gene`) in that particular experiment at that particular time point.

In every column (except "gene"), the cells represents the same abstract property of interest – the expression of a gene of interest in a particular experiment/time point. Our first task will be to rearrange our "wide" data frame that consists of many different columns representing gene expression into a "long" data frame with just a single column representing expression. We'll also create a new column to keep track of which experiment and time point the measurement came from.

### 8.6.1 Wide to long conversions using `tidyr::gather`

`tidyr::gather()` takes multiple columns, and collapses them together into a smaller number of new columns. When using `gather()` you give the names of the *new* columns to create, as well as the names of any existing columns `gather()` should *not* collect together.

Here we want to collapse all 73 or the expression columns – “alpha0” to “elu390” – into two columns: 1) a column to represent the expt/time point of the measurement, and 2) a column to represent the corresponding expression value. The column we don’t want to touch are the `gene`, `std.name`, and `description`.

```
# convert "wide" data to "long"
spellman.long <-
  spellman.merged %>%
  gather(expt.and.time, expression, -gene, -std.name, -description)
```

Take a moment to look at the data in the “long format”:

| gene   | std.name | description                            | expt.and.time | expression |
|--------|----------|--|---------------|------------|
| YAL00~ | TFC3     | Subunit of RNA polymerase III ~ alpha0 | <chr>         | <dbl>      |
| YAL00~ | VPS8     | Membrane-binding component of ~ alpha0 |               | -0.15      |
| YAL00~ | EFB1     | Translation elongation factor ~ alpha0 |               | -0.11      |
| YAL00~ | <NA>     | Dubious open reading frame; un~ alpha0 |               | -0.14      |
| YAL00~ | SSA1     | ATPase involved in protein fol~ alpha0 |               | -0.02      |
| YAL00~ | ERP2     | Member of the p24 family invol~ alpha0 |               | -0.05      |
|        |          |  |               | -0.6       |

And compare the dimensions of the wide data to the new data:

```
dim(spellman.merged) # for comparison
#> [1] 6178 76
dim(spellman.long)
#> [1] 450994      5
```

As you see, we’ve gone from a data frame with 6178 rows and 76 columns (wide format), to a new data frame with 450994 rows and 5 columns (long format).

### 8.6.2 Extracting information from combined variables using `tidyr::extract`

The column `expt.and.time` violates one of our principles of tidy data: “Each variable must have its own column.” This column conflates two different types of information – the experiment type and the time point of the measurement. Our next task is to split this information up into two new variables, which will help to facilitate downstream plotting and analysis.

One complicating factor is that the different experiments/time combinations have different naming conventions:

- The “alpha” and “elu” experiments are of the form “alpha0”, “alpha7”, “elu0”, “elu30”, etc. In this case, the first part of the string gives the experiment type (either alpha or elu) and the following digits give the time point.
- In the “cdc15” and “cdc28” experiments the convention is slightly different; they are of the form “cdc15\_0”, “cdc15\_10”, “cdc28\_0”, “cdc28\_10”, etc. Here the part of the string before the underscore gives the experiment type, and the digits after the underscore give the time point.

Because of the differences in naming conventions, we will find it easiest to break up `spellman.long` into a series of sub-data sets corresponding to each experiment type in order to extract out the experiment and

time information. After processing each data subset separately, we will join the modified sub-data frames back together.

### 8.6.3 Subsetting rows

Let's start by getting just the rows corresponding to the “alpha” experiment/times. Here we use `dplyr::filter` in combination with `stringr::str_detect` to get all those rows in which the `expt.and.time` variable contains the string “alpha”.

```
alpha.long <-  
  spellman.long %>%  
  filter(str_detect(expt.and.time, "alpha"))

# look at the new data frame  
dim(alpha.long)
#> [1] 111204      5
head(alpha.long, n = 10)
#> # A tibble: 10 x 5
#>   gene std.name description          expt.and.time expression
#>   <chr> <chr>    <chr>           <chr>            <dbl>
#> 1 YAL00~ TFC3   Subunit of RNA polymerase III~ alpha0     -0.15
#> 2 YAL00~ VPS8   Membrane-binding component of~ alpha0     -0.11
#> 3 YAL00~ EFB1   Translation elongation factor~ alpha0     -0.14
#> 4 YAL00~ <NA>   Dubious open reading frame; u~ alpha0     -0.02
#> 5 YAL00~ SSA1   ATPase involved in protein fo~ alpha0     -0.05
#> 6 YAL00~ ERP2   Member of the p24 family invo~ alpha0     -0.6
#> 7 YAL00~ FUN14  Integral mitochondrial outer ~ alpha0     -0.28
#> 8 YAL00~ SPO7   Putative regulatory subunit o~ alpha0     -0.03
#> 9 YAL01~ MDM10  Subunit of both the ERMES and~ alpha0     -0.05
#> 10 YAL01~ SWC3  Protein of unknown function; ~ alpha0    -0.31
```

### 8.6.4 Splitting columns

Having subsetted the data, we can now split `expt.and.time` into two new variables – `expt` and `time`. To do this we use `tidyr::extract`.

```
alpha.long %>%
  tidyr::extract(expt.and.time,      # column we're extracting from
                 c("expt", "time"), # new columns we're creating
                 regex="(alpha)([[digit]])+", # regexp (see below)
                 convert=TRUE)       # automatically convert column types

# NOTE: I'm being explicit about saying tidyr::extract because the
# magrittr package defines a different extract function
```

Let's take a moment to look at the `regex` argument to `extract` – `regex="(alpha)([[digit]])+"`. The regex is specified as a character string. Each part we want to match and extract is surround by parentheses. In this case we have two sets of parentheses corresponding to the two matches we want to make. The first part of the regex is `(alpha)`; here we're looking to make an exact match to the string “alpha”. The second part of the regex reads `([[digit]])+`. `[[digit]]` indicates we're looking for a numeric digit. The `+` after `[[digit]]` indicates that we want to match *one or more* digits (i.e. to get a match we need to find at least one digit, but more than one digit should also be a match).

Let's take a look at the new version of `alpha.long` following application of `extract`:

```
head(alpha.long, n = 10)
#> # A tibble: 10 x 6
#>   gene std.name description      expt    time expression
#>   <chr> <chr>   <chr>          <chr> <int>     <dbl>
#> 1 YAL00~ TFC3   Subunit of RNA polymerase III t~ alpha    0   -0.15
#> 2 YAL00~ VPS8   Membrane-binding component of t~ alpha    0   -0.11
#> 3 YAL00~ EFB1   Translation elongation factor 1~ alpha    0   -0.14
#> 4 YAL00~ <NA>  Dubious open reading frame; unl~ alpha    0   -0.02
#> 5 YAL00~ SSA1   ATPase involved in protein fold~ alpha    0   -0.05
#> 6 YAL00~ ERP2   Member of the p24 family involv~ alpha    0   -0.6
#> 7 YAL00~ FUN14  Integral mitochondrial outer me~ alpha    0   -0.28
#> 8 YAL00~ SPO7   Putative regulatory subunit of ~ alpha    0   -0.03
#> 9 YAL01~ MDM10  Subunit of both the ERMES and t~ alpha    0   -0.05
#> 10 YAL01~ SWC3  Protein of unknown function; co~ alpha   0   -0.31
```

Notice our two new variables, both of which have appropriate types!

A data frame for the elutriation data can be created similarly:

```
elu.long <- 
  spellman.long %>%
  filter(str_detect(expt.and.time, "elu")) %>%
  tidyr::extract(expt.and.time,      # column we're extracting from
                 c("expt", "time"), # new columns we're creating
                 regex="(elu)([[[:digit:]])+", # regexp (see below)
                 convert=TRUE)       # automatically convert column types
```

#### 8.6.4.1 A fancier regex for the cdc experiments

Now let's process the cdc experiments (cdc15 and cdc28). As before we extract the corresponding rows of the data frame using `filter` and `str_detect`. We then split `expt.and.time` using `tidyr::extract`. In this case we carry out the two steps in a single code block using pipes:

```
cdc.long <-
  spellman.long %>%
  # both cdc15 and cdc28 contain "cdc" as a sub-string
  filter(str_detect(expt.and.time, "cdc")) %>%
  tidyr::extract(expt.and.time,
                 c("expt", "time"),
                 regex="(cdc15|cdc28)_([[[:digit:]])+", # note the fancier regex
                 convert=TRUE)
```

The regex – `"(cdc15|cdc28)_([[[:digit:]])+"` – is slightly fancier in this example. As before there are two parts we're extracting: `(cdc15|cdc28)` and `([[[:digit:]])+`. The first parenthesized regexp is an “OR” – i.e. match “cdc15” or “cdc28”. The second parenthesized regexp is the same as we saw previously. Separating the two parenthesized regexps is an underscore (`_`). The underscore isn't parenthesized because we only want to use it to make a match *not* to extract the corresponding match.

#### 8.6.5 Combining data frames

If you have two or more data frames with identical columns, the rows of the data frames can be combined into a single data frame using `rbind` (defined in the `base` package). For example, to reassemble the `alpha.long`, `elu.long`, and `cdc.long` data frames into a single data frame we do:

```
spellman.final <- rbind(alpha.long, elu.long, cdc.long)

# check the dimensions of the new data frame
dim(spellman.final)
#> [1] 450994      6
```

### 8.6.6 Sorting data frame rows

Currently the `spellman.final` data frame is sorted by time point and experiment.

```
head(spellman.final, n = 10)
#> # A tibble: 10 x 6
#>   gene std.name description          expt  time expression
#>   <chr> <chr>   <chr>           <chr> <int>    <dbl>
#> 1 YAL00~ TFC3  Subunit of RNA polymerase III t~ alpha     0   -0.15
#> 2 YAL00~ VPS8  Membrane-binding component of t~ alpha     0   -0.11
#> 3 YAL00~ EFB1  Translation elongation factor 1~ alpha     0   -0.14
#> 4 YAL00~ <NA> Dubious open reading frame; unl~ alpha     0   -0.02
#> 5 YAL00~ SSA1  ATPase involved in protein fold~ alpha     0   -0.05
#> 6 YAL00~ ERP2  Member of the p24 family involv~ alpha     0   -0.6
#> 7 YAL00~ FUN14 Integral mitochondrial outer me~ alpha     0   -0.28
#> 8 YAL00~ SPO7  Putative regulatory subunit of ~ alpha     0   -0.03
#> 9 YAL01~ MDM10 Subunit of both the ERMES and t~ alpha     0   -0.05
#> 10 YAL01~ SWC3 Protein of unknown function; co~ alpha    0   -0.31
```

It might be useful instead to sort by gene and experiment. To do this we can use `dplyr::arrange`:

```
spellman.final %<>%
  arrange(gene, expt)

# look again at the rearranged data
head(spellman.final, n = 10)
#> # A tibble: 10 x 6
#>   gene std.name description          expt  time expression
#>   <chr> <chr>   <chr>           <chr> <int>    <dbl>
#> 1 YAL00~ TFC3  Subunit of RNA polymerase III t~ alpha     0   -0.15
#> 2 YAL00~ TFC3  Subunit of RNA polymerase III t~ alpha     7   -0.15
#> 3 YAL00~ TFC3  Subunit of RNA polymerase III t~ alpha    14   -0.21
#> 4 YAL00~ TFC3  Subunit of RNA polymerase III t~ alpha    21   0.17
#> 5 YAL00~ TFC3  Subunit of RNA polymerase III t~ alpha    28   -0.42
#> 6 YAL00~ TFC3  Subunit of RNA polymerase III t~ alpha    35   -0.44
#> 7 YAL00~ TFC3  Subunit of RNA polymerase III t~ alpha    42   -0.15
#> 8 YAL00~ TFC3  Subunit of RNA polymerase III t~ alpha    49   0.24
#> 9 YAL00~ TFC3  Subunit of RNA polymerase III t~ alpha    56   -0.1
#> 10 YAL00~ TFC3 Subunit of RNA polymerase III t~ alpha    63   NA
```

## 8.7 Using your tidy data

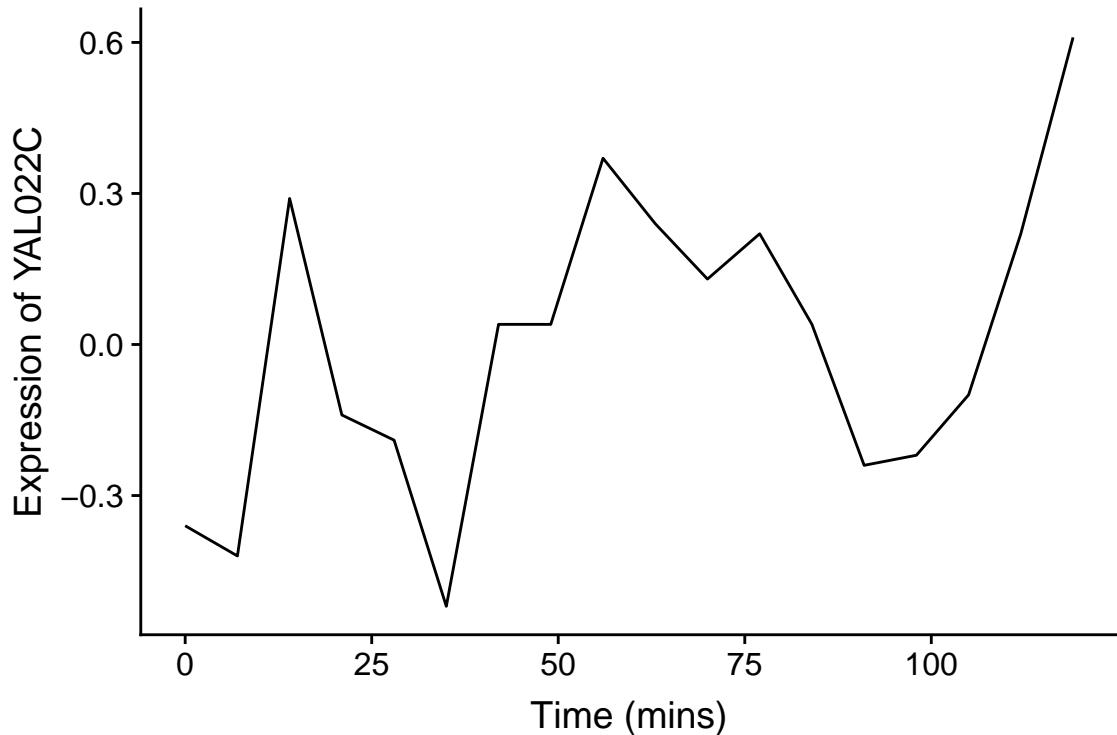
Whew – that was a fair amount of work to tidy our data! But having done so we can now carry out a wide variety of very powerful analyses.

### 8.7.1 Visualizing gene expression time series

Let's start by walking through a series of visualizations of gene expression time series. Each plot will show the expression of one or more genes, at different time points, in one or more experimental conditions. Our initial visualizations exploit the “long” versions of the tidy data.

First a single gene in a single experimental condition:

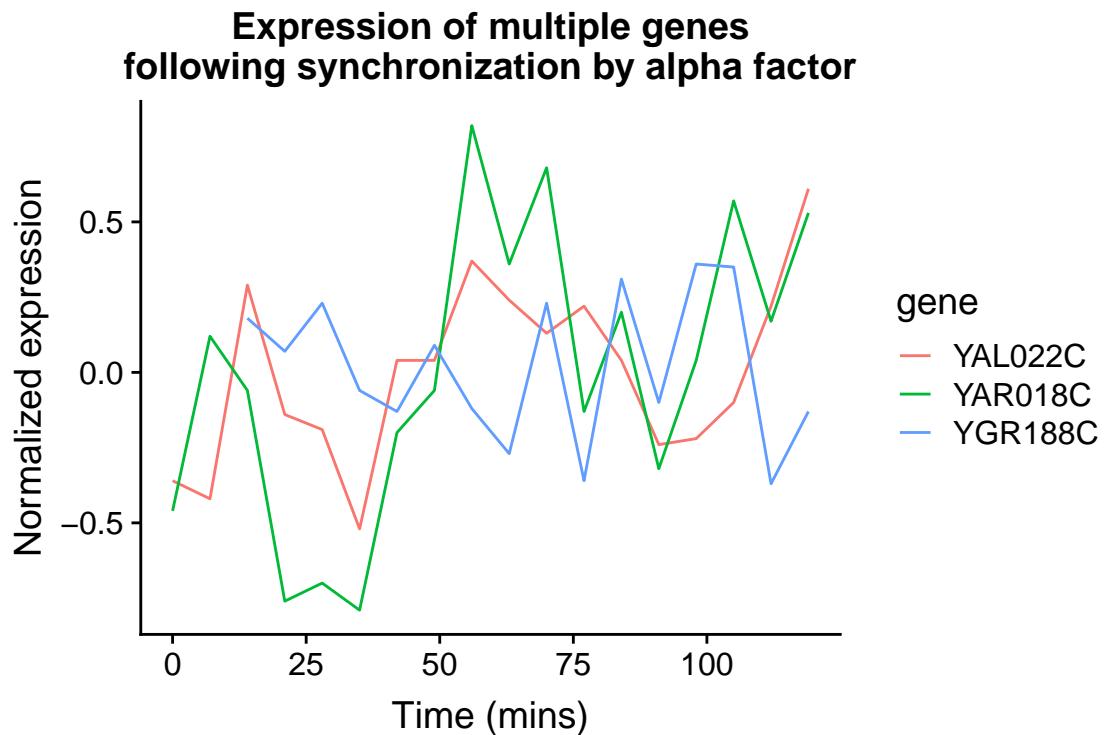
```
spellman.final %>%
  filter(expt == "alpha", gene == "YAL022C") %>%
  ggplot(aes(x = time, y = expression)) +
  geom_line() +
  labs(x = "Time (mins)", y = "Expression of YAL022C")
```



We can easily modify the above code block to visualize the expression of multiple genes of interest:

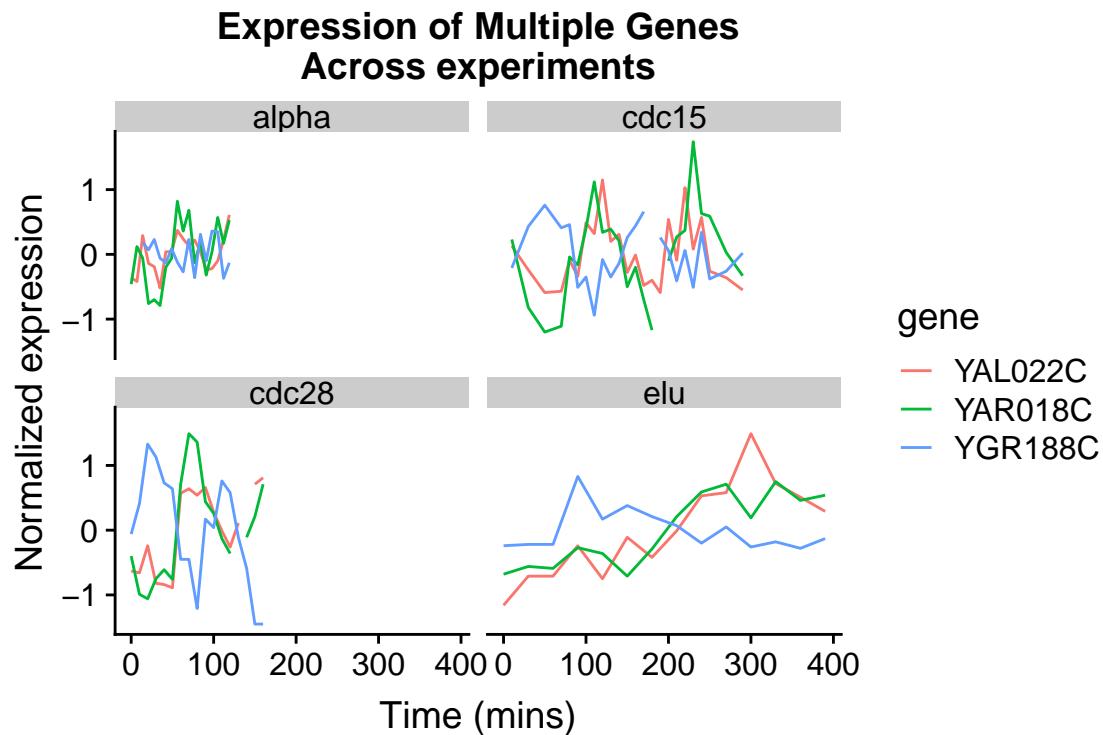
```
genes.of.interest <- c("YAL022C", "YAR018C", "YGR188C")

spellman.final %>%
  filter(gene %in% genes.of.interest, expt == "alpha") %>%
  ggplot(aes(x = time, y = expression, color = gene)) +
  geom_line() +
  labs(x = "Time (mins)", y = "Normalized expression",
       title = "Expression of multiple genes\\nfollowing synchronization by alpha factor")
```



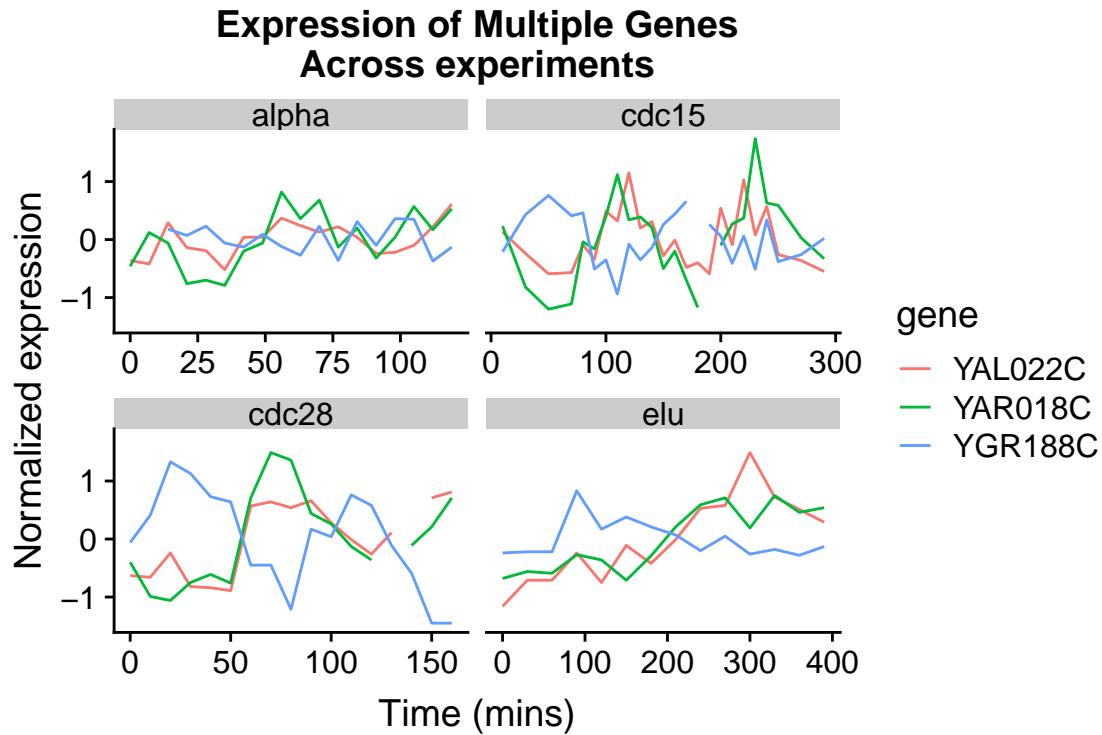
By employing `facet_wrap()` we can visualize the relationship between this set of genes in each of the experiment types:

```
spellman.final %>%
  filter(gene %in% genes.of.interest) %>%
  ggplot(aes(x = time, y = expression, color = gene)) +
  geom_line() +
  facet_wrap(~ expt) +
  labs(x = "Time (mins)", y = "Normalized expression",
       title = "Expression of Multiple Genes\nAcross experiments")
```



The different experimental treatments were carried out for varying lengths of time due to the differences in their physiological effects. Plotting them all on the same time scale can obscure that patterns of oscillation we might be interested in, so let's modify our code block so that plots that share the same y-axis, but have differently scaled x-axes.

```
spellman.final %>%
  filter(gene %in% genes.of.interest) %>%
  ggplot(aes(x = time, y = expression, color = gene)) +
  geom_line() +
  facet_wrap(~ expt, scales = "free_x") +
  labs(x = "Time (mins)", y = "Normalized expression",
       title = "Expression of Multiple Genes\nAcross experiments")
```



### 8.7.2 Finding the most variable genes

When dealing with very large data sets, one ad hoc filtering criteria that is often employed is to focus on those variables that exhibit the greatest variation (variation is measure of the spread of data; we will give a precise definition in a later lecture). To do this, we first need to order our variables (genes) by their variation. Let's see how we can accomplish this using our long data frame:

```
by.variance <-
  spellman.final %>%
  group_by(gene) %>%
  summarize(expression.var = var(expression, na.rm = TRUE)) %>%
  arrange(desc(expression.var))

head(by.variance)
#> # A tibble: 6 x 2
#>   gene    expression.var
#>   <chr>        <dbl>
#> 1 YLR286C     2.157
#> 2 YNR067C     1.733
#> 3 YNL327W     1.653
#> 4 YGL028C     1.571
#> 5 YHL028W     1.521
#> 6 YKL164C     1.515
```

The code above calculates the variance of each gene but ignores the fact that we have different experimental conditions. To take into account the experimental design of the data at hand, let's calculate the average variance across the experimental conditions:

```
by.avg.variance <-
  spellman.final %>%
  group_by(gene, expt) %>%
  summarize(expression.var = var(expression, na.rm = TRUE)) %>%
  group_by(gene) %>%
  summarize(avg.expression.var = mean(expression.var)) %>%
  arrange(desc(avg.expression.var))

head(by.avg.variance)
#> # A tibble: 6 x 2
#>   gene      avg.expression.var
#>   <chr>            <dbl>
#> 1 YFR014C         3.579
#> 2 YFR053C         2.377
#> 3 YBL032W         2.299
#> 4 YDR274C         2.173
#> 5 YLR286C         2.128
#> 6 YMR206W         1.937
```

Based on the average expression variance across experimental conditions, let's get the names of the 1000 most variable genes:

```
top.genes.1k <- by.avg.variance[1:1000,]$gene

head(top.genes.1k)
#> [1] "YFR014C" "YFR053C" "YBL032W" "YDR274C" "YLR286C" "YMR206W"
```



# Chapter 9

## Data wrangling, Part II

In our last class session we walked through a complex, data wrangling pipeline involving a genome-wide gene expression data set. Starting with the raw data, we demonstrated several cleaning, transformation, and reshaping steps that resulted in a data set that allowed us to examine how gene expression changed over time across multiple experimental conditions. The final form of our data was what we referred to as a “long” format. The key variables of the final data frame were `gene`, `expt`, `time`, and `expression`. The structure of this long data frame facilitated the creation of time series plots, filtering by gene and/or condition, grouping operations by gene, etc.

In today’s session we’re going to demonstrate a new type of visualization called a “heat map” that is useful for high dimensional data. Then we’ll show how to go convert our “long” data frame to a “wide” data frame, and show how this wide data frame facilitates analyses focused on how gene expression changes in concert (covary). We’ll also show how we can combine our long and wide views of the data to create new insights into interesting patterns in the data.

### 9.1 Libraries

```
library(magrittr)
library(tidyverse)
```

### 9.2 Data

So we don’t have to re-create our previous analysis, I’ve posted a CSV file with the “long” version of the cleaned Spellman data set at the link given below:

```
spellman.long <- read_csv("https://github.com/bio304-class/bio304-course-notes/raw/master/datasets/spellman.csv")
```

Let’s remind ourselves of the basic structure of this data frame:

```
head(spellman.long)
#> # A tibble: 6 x 4
#>   gene    expt    time  expression
#>   <chr>   <chr>   <int>      <dbl>
#> 1 YAL001C alpha      0     -0.15
#> 2 YAL002W alpha      0     -0.11
#> 3 YAL003W alpha      0     -0.14
```

```
#> 4 YAL004W alpha      0     -0.02
#> 5 YAL005C alpha      0     -0.05
#> 6 YAL007C alpha      0     -0.6
dim(spellman.long)
#> [1] 450994      4
```

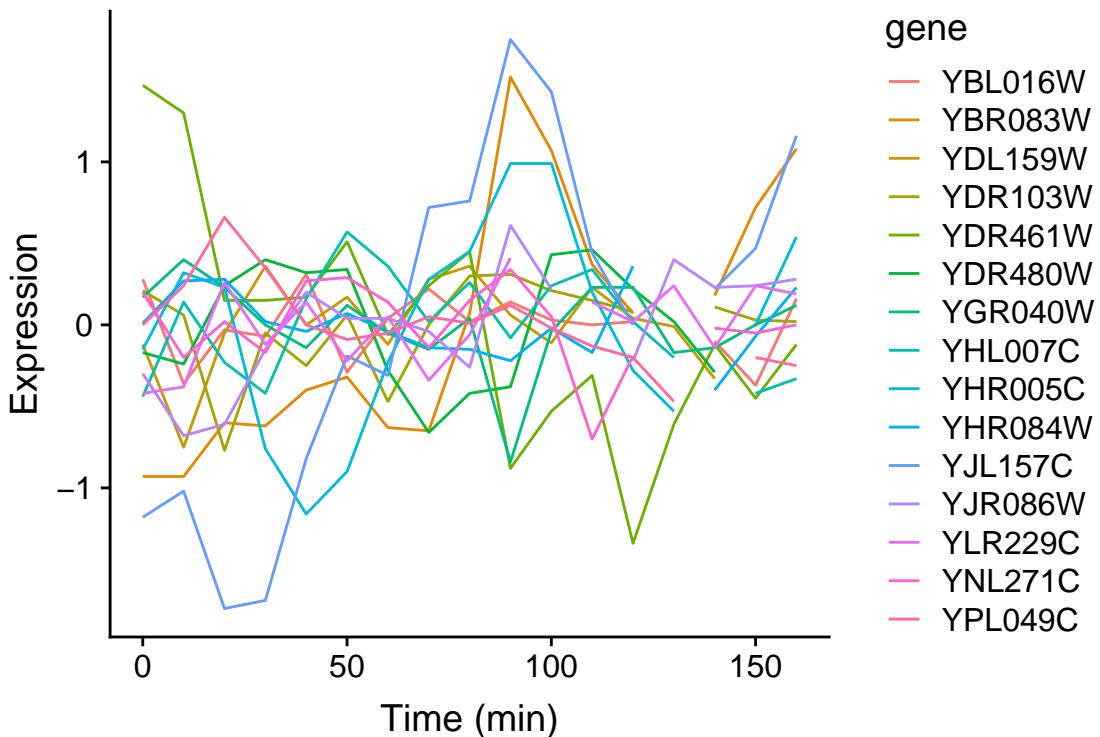
There are just four columns (variables) in this data set, but more than 450,000 rows, representing all the various combinations of genes (>6000), experimental conditions (4), and time points (variable across experiments).

### 9.3 Heat maps

In our prior visualizations we've used line plots to depict how gene expression changes over time. For example here are line plots for 15 genes in the data set, in the cdc28 experimental conditions:

```
genes.of.interest <- c("YHR084W", "YBR083W", "YPL049C", "YDR480W",
                      "YGR040W", "YLR229C", "YDL159W", "YBL016W",
                      "YDR103W", "YJL157C", "YNL271C", "YDR461W",
                      "YHL007C", "YHR005C", "YJR086W")
```

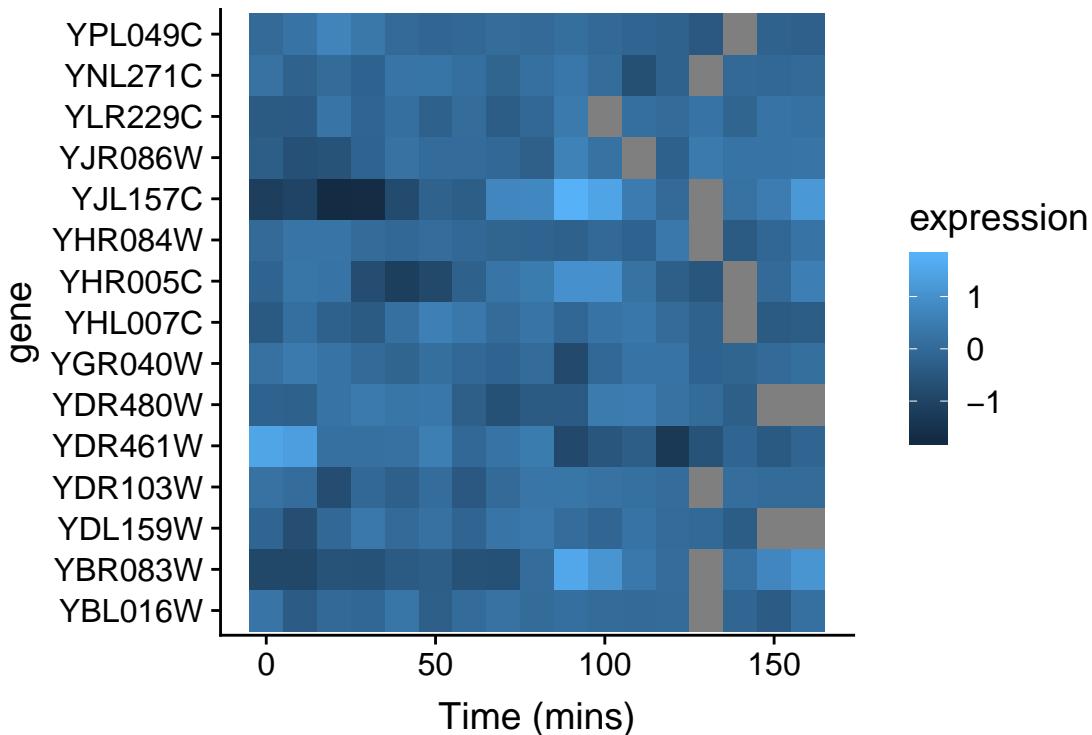
```
spellman.long %>%
  filter(expt == "cdc28", gene %in% genes.of.interest) %>%
  ggplot(aes(x = time, y = expression, color= gene)) +
  geom_line() +
  labs(x = "Time (min)", y = "Expression")
```



Even with just 10 overlapping line plots, this figure is quite busy and it's hard to make out the individual behavior of each gene.

An alternative approach to depicting such data is a “heat map” which depicts the same information in a grid like form, with the expression values indicated by color. Heat maps are good for depicting large amounts of data and providing a coarse “10,000 foot view”. We can create a heat map using `geom_tile` as follows:

```
spellman.long %>%
  filter(expt == "cdc28", gene %in% genes.of.interest) %>%
  ggplot(aes(x = time, y = gene)) +
  geom_tile(aes(fill = expression)) +
  xlab("Time (mins)")
```



This figure represents the same information as our line plot, but now there is a row for each gene, and the expression of that gene at a given time point is represented by color (scale given on the right). Missing data is shown as gray boxes. Unfortunately, the default color scale used by ggplot is a very subtle gradient from light to dark blue. This makes it hard to distinguish patterns of change. Let’s now see how we can improve that.

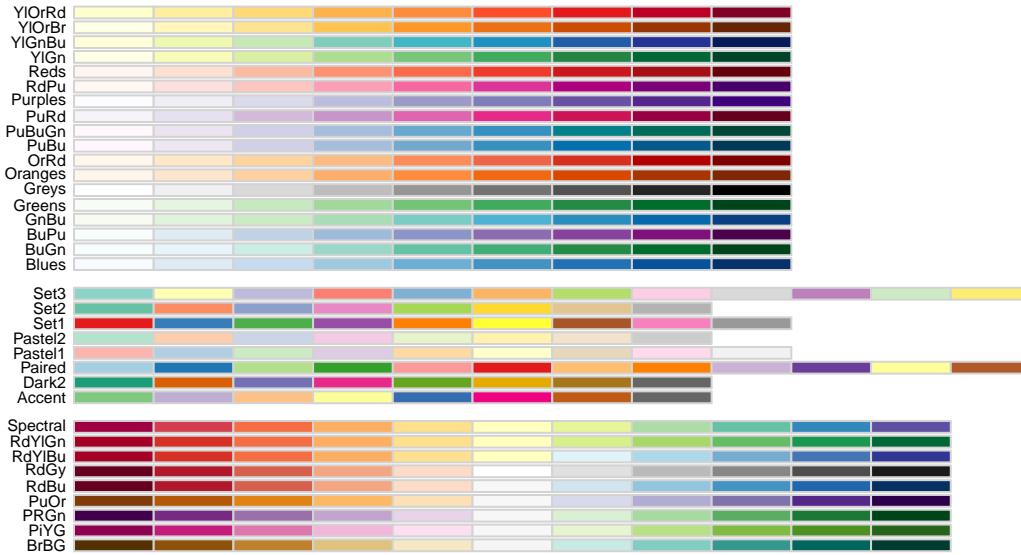
### 9.3.1 Better color schemes with RColorBrewer

The `RColorBrewer` package provides nice color schemes that are useful for creating heat maps. `RColorBrewer` defines a set of color palettes that have been optimized for color discrimination, many of which are color blind friendly, etc. Install the `RColorBrewer` package using the command line or the RStudio GUI.

Once you’ve installed the `RColorBrewer` package you can see the available color palettes as so:

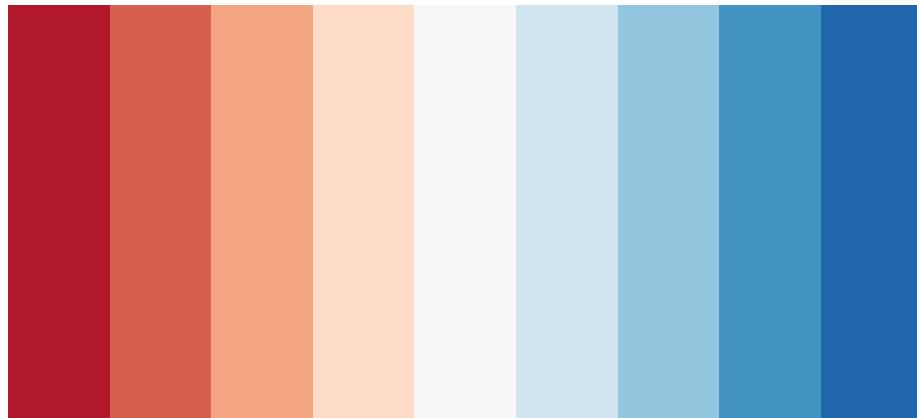
```
library(RColorBrewer)

# show representations of the palettes
par(cex = 0.5) # reduce size of text in the following plot
display.brewer.all()
```



We'll use the Red-to-Blue ("RdBu") color scheme defined in RColorBrewer, however we'll reverse the scheme so blues represent low expression and reds represent high expression. We'll divide the range of color values into 9 discrete bins.

```
# displays the RdBu color scheme divided into a palette of 9 colors
display.brewer.pal(9, "RdBu")
```

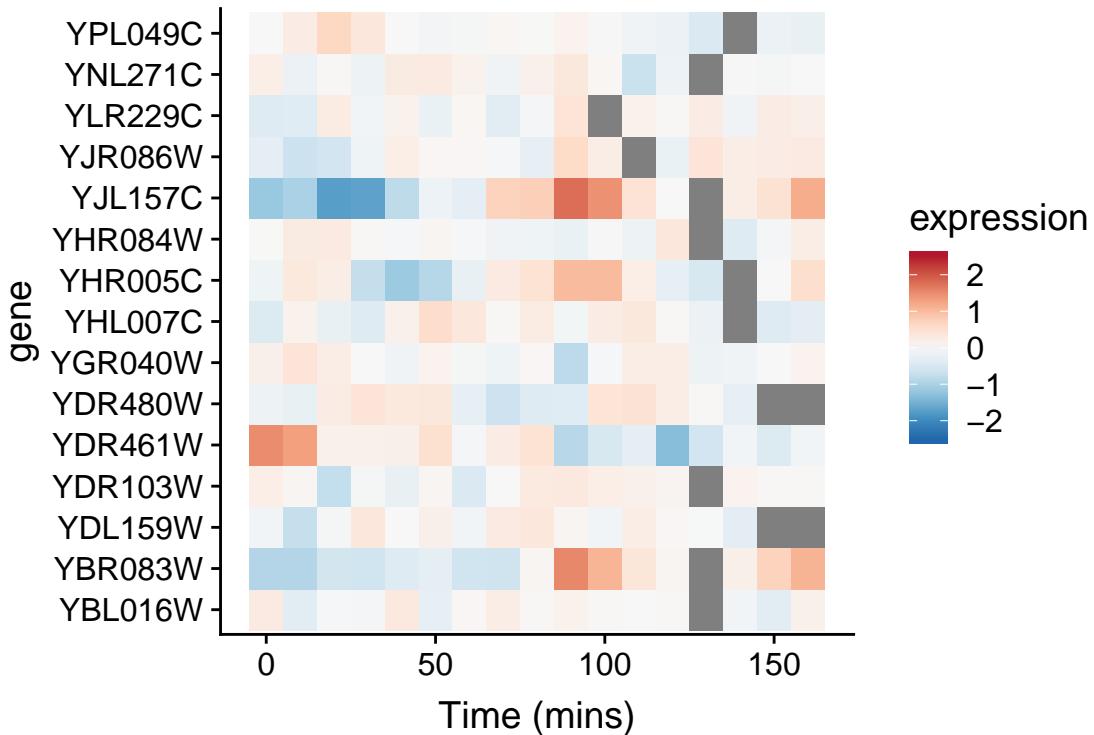


RdBu (divergent)

```
# assign the reversed (blue to red) RdBu palette
color.scheme <- rev(brewer.pal(9,"RdBu"))
```

Now let's regenerate the heat map we created previously with this new color scheme. To do this we specify a gradient color scale using the `scale_fill_gradientn()` function from ggplot. In addition to specifying the color scale, we also constrain the limits of the scale to insure it's symmetric about zero.

```
spellman.long %>%
  filter(expt == "cdc28", gene %in% genes.of.interest) %>%
  ggplot(aes(x = time, y = gene)) +
  geom_tile(aes(fill = expression)) +
  scale_fill_gradientn(colors=color.scheme,
                      limits = c(-2.5, 2.5)) +
  xlab("Time (mins)")
```



### 9.3.2 Looking for patterns using sorted data and heat maps

The real power of heat maps becomes apparent when you rearrange the rows of the heat map to emphasize patterns of interest.

For example, let's create a heat map in which we sort genes by the time of their maximal expression. This is one way to identify genes that reach their peak expression at similar times, which is one criteria one might use to identify genes acting in concert.

For simplicities sake we will restrict our attention to the cdc28 experiment, and only consider the 1000 most variables genes with no more than one missing observation in this experimental condition.

```
cdc28 <-
  spellman.long %>%
  filter(expt == "cdc28") %>%
  group_by(gene) %>%
  filter(sum(is.na(expression)) <= 1) %>%
  ungroup    # removes grouping information from data frame

top1k.genes <-
  cdc28 %>%
  group_by(gene) %>%
  summarize(expression.var = var(expression, na.rm = TRUE)) %>%
  arrange(desc(expression.var)) %$%
  gene[1:1000]

top1k.cdc28 <-
  cdc28 %>%
  filter(gene %in% top1k.genes)
```

To find the time of maximum expression we'll employ the function `which.max` (`which.min`), which finds the index of the maximum (minimum) element of a vector. For example to find the index of the maximum expression measurement for YAR018C we could do:

```
top1k.cdc28 %>%
  filter(gene == "YAR018C") %$% # note the exposition pipe operator!
  which.max(expression)
#> [1] 8
```

From the code above we find that the index of the observation at which YAR018C is maximal at 8. To get the corresponding time point we can do something like this:

```
top1k.cdc28 %>%
  filter(gene == "YAR018C") %$% # again note the exposition pipe operator!
  time[which.max(expression)]
#> [1] 70
```

Thus YAR018C expression peaks at 70 minutes in the cdc28 experiment.

To find the index of maximal expression of all genes we can apply the `dplyr::group_by()` and `dplyr::summarize()` functions

```
peak.expression.cdc28 <-  

  top1k.cdc28 %>%
  group_by(gene) %>%
  summarise(peak = which.max(expression))  
  

head(peak.expression.cdc28)
#> # A tibble: 6 x 2
#>   gene      peak
#>   <chr>     <int>
#> 1 YAL003W     10
#> 2 YAL005C      2
#> 3 YAL022C     17
#> 4 YAL028W      5
#> 5 YAL035C-A    12
#> 6 YAL038W     15
```

Let's sort the order of genes by their peak expression:

```
peak.expression.cdc28 %<% arrange(peak)
```

We can then generate a heatmap where we sort the rows (genes) of the heatmap by their time of peak expression. We introduce a new geom – `geom_raster` – which is like `geom_tile` but better suited for large data (hundreds to thousands of rows)

The explicit sorting of the data by peak expression is carried out in the call to `scale_y_discrete()` where the limits (and order) of this axis are set with the `limits` argument (see `scale_y_discrete` and `discrete_scale` in the ggplot2 docs).

```
# we reverse the ordering because geom_raster (and geom_tile)
# draw from the bottom row up, whereas we want to depict the
# earliest peaking genes at the top of the figure
gene.ordering <- rev(peak.expression.cdc28$gene)  
  

top1k.cdc28 %>%
  ggplot(aes(x = time, y = gene)) +
  geom_raster(aes(fill = expression)) + #
```

```
scale_fill_gradientn(limits=c(-2.5, 2.5),
                     colors=color.scheme) +
  scale_y_discrete(limits=gene.ordering) +
  labs(x = "Time (mins)", y = "Genes",
       title = "1000 most variable genes",
       subtitle = "Sorted by time of peak expression") +
  # the following line suppresses tick and labels on y-axis
  theme(axis.text.y = element_blank(), axis.ticks.y = element_blank())
```

The brightest red regions in each row of the heat map correspond to the times of peak expression, and the sorting of the rows helps to highlight those gene whose peak expression times are similar.

## 9.4 Long-to-wide conversion using `tidyverse::spread`

Our long data frame consists of four variables – `gene`, `expt`, `time`, and `expression`. This made it easy to create visualizations and summaries where time and expression were the primaries variables of interest, and gene and experiment type were categories we could condition on.

To facilitate analyses that emphasize comparison between genes, we want to create a new data frame in which each gene is itself treated as a variable of interest along with time, and experiment type remains a categorical variable. In this new data frame rather than just four columns in our data frame, we'll have several thousand columns – one for each gene.

To accomplish this reshaping of data, we'll use the function `tidyverse::spread()`. `tidyverse::spread()` is the inverse of `tidyverse::gather()`. `gather()` took multiple columns and collapsed them together into a smaller number of new columns. The `tidyverse` documentation calls this “collapsing into key-value pairs”. By contrast, `spread()` creates new columns by spreading “key-value pairs” (a column representing the “keys” and a column representing the “values”) into multiple columns.

Here let's use `spread()` to use the gene names (the “key”) and expression measures (the “values”) to create a new data frame where the genes are the primary variables (columns) of the data.

```
spellman.wide <-
  spellman.long %>%
  spread(gene, expression)
```

Now let's examine the dimensions of this wide version of the data:

```
dim(spellman.wide)
#> [1] 73 6180
```

And here's a visual view of the first few rows and columns of the wide data:

```
spellman.wide[1:5, 1:8]
#> # A tibble: 5 x 8
#>   expt    time YAL001C YAL002W YAL003W YAL004W YAL005C YAL007C
#>   <chr> <int>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
#> 1 alpha     0   -0.15   -0.11   -0.14   -0.02   -0.05   -0.6
#> 2 alpha     7   -0.15     0.1   -0.71   -0.48   -0.53   -0.45
#> 3 alpha    14   -0.21     0.01     0.1   -0.11   -0.47   -0.13
#> 4 alpha    21    0.17     0.06   -0.32    0.12   -0.06    0.35
#> 5 alpha    28   -0.42     0.04    -0.4   -0.03    0.11   -0.01
```

From this view we infer that the rows of the data set represent the various combination of experimental condition and time points, and the columns represents the 6178 genes in the data set plus the two columns for `expt` and `time`.

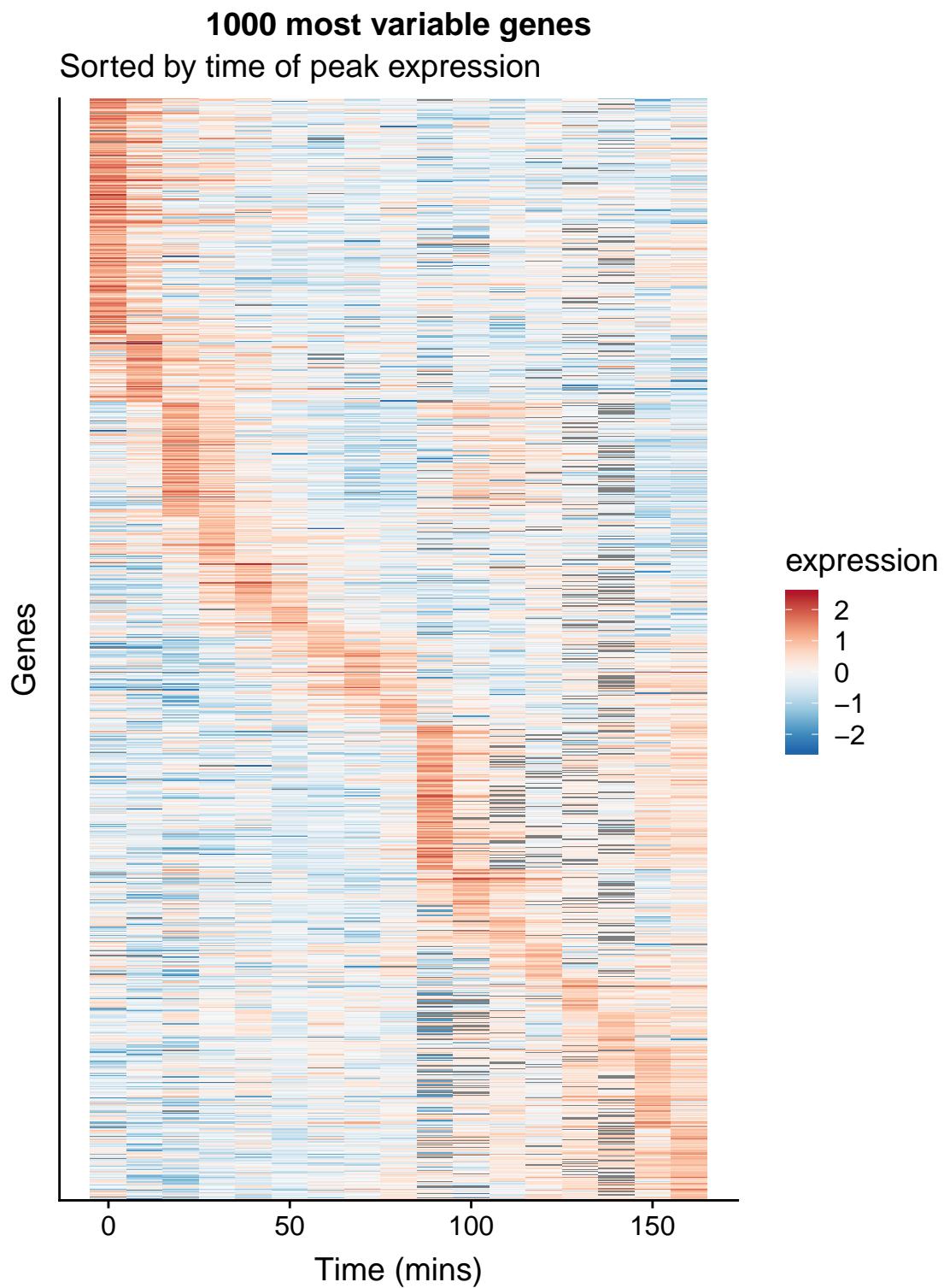


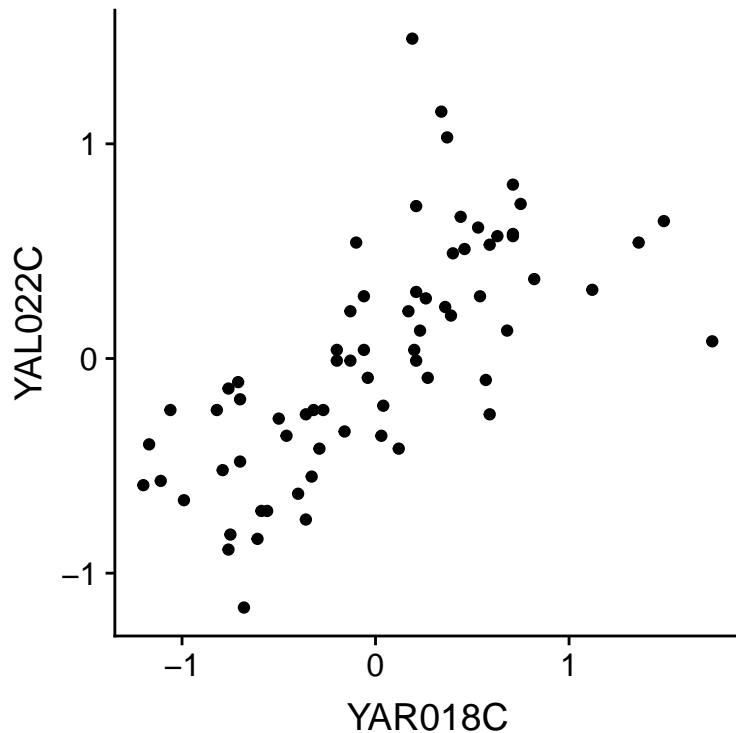
Figure 9.1: A Heatmap showing genes in the `cdc28` experiment, sorted by peak expression

## 9.5 Exploring bivariate relationships using “wide” data

The “long” version of our data frame proved useful for exploring how gene expression changed over time. By contrast, our “wide” data frame is more convenient for exploring how pairs of genes covary together. For example, we can generate bivariate scatter plots depicting the relationship between two genes of interest:

```
two.gene.plot <-
  spellman.wide %>%
  filter(!is.na(YAR018C) & !is.na(YAL022C)) %>% # remove NAs
  ggplot(aes(x = YAR018C, y = YAL022C)) +
  geom_point() +
  theme(aspect.ratio = 1)

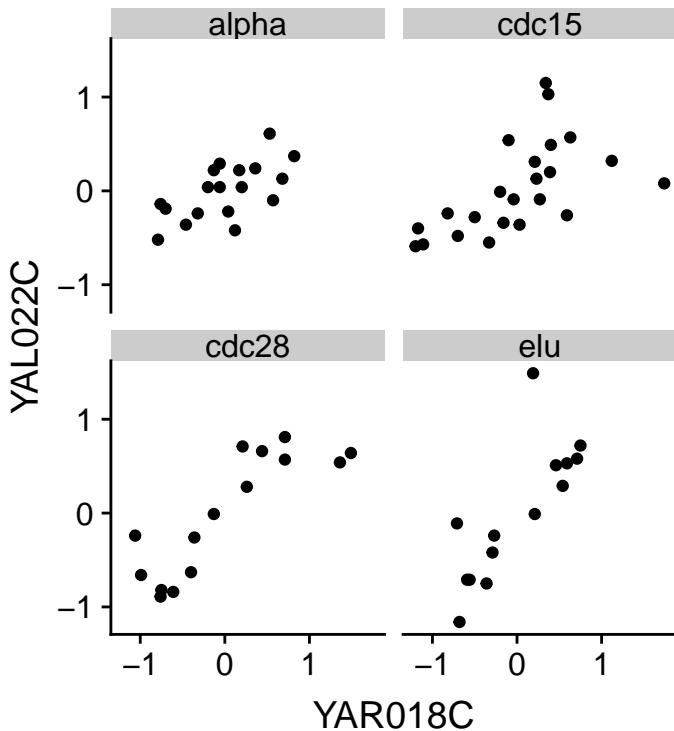
two.gene.plot
```



From the scatter plot we infer that the two genes are “positively correlated” with each other, meaning that high values of one tend to be associated with high values of the other (and the same for low values).

We can easily extend this visualization to facet the plot based on the experimental conditions:

```
two.gene.plot + facet_wrap(~expt, nrow = 2, ncol = 2)
```



A statistic we use to measure the degree of association between pairs of continuous variables is called the “correlation coefficient”. Briefly, correlation is a measure of linear association between a pair of variables, and ranges from -1 to 1. A value near zero indicates the variables are uncorrelated (no linear association), while values approaching +1 indicate a strong positive association (the variables tend to get bigger or smaller together) while values near -1 indicate strong negative association (when one variable is larger, the other tends to be small).

Let's calculate the correlation between YAR018C and YAL022C:

```
spellman.wide %>%
  filter(!is.na(YAR018C) & !is.na(YAL022C)) %>%
  summarize(cor = cor(YAR018C, YAL022C))
#> # A tibble: 1 x 1
#>   cor
#>   <dbl>
#> 1 0.6915
```

The value of the correlation coefficient for YAR018C and YAL022C, ~0.69, indicates a fairly strong association between the two genes.

As we did for our visualization, we can also calculate the correlation coefficients for the two genes under each experimental condition:

```
spellman.wide %>%
  filter(!is.na(YAR018C) & !is.na(YAL022C)) %>%
  group_by(expt) %>%
  summarize(cor = cor(YAR018C, YAL022C))
#> # A tibble: 4 x 2
#>   expt     cor
#>   <chr>  <dbl>
#> 1 alpha  0.6341
```

```
#> 2 cdc15 0.5751
#> 3 cdc28 0.8474
#> 4 elu 0.7867
```

This table suggests that the strength of correlation between YAR018C and YAL022C may depend on the experimental conditions, with the highest correlations evident in the cdc28 and elu experiments.

### 9.5.1 Large scale patterns of correlations

Now we'll move from considering the correlation between two specific genes to looking at the correlation between many pairs of genes. As we did in the previous section, we'll focus specifically on the 1000 most variable genes in the cdc28 experiment.

First we filter our wide data set to only consider the cdc28 experiment and those genes in the top 1000 most variable genes in cdc28:

```
top1k.cdc28.wide <-
  top1k.cdc28 %>%
  spread(gene, expression)
```

With this restricted data set, we can then calculate the correlations between every pair of genes as follows:

```
cdc28.correlations <-
  top1k.cdc28.wide %>%
  select(-expt, -time) %>% # drop expt and time
  cor(use = "pairwise.complete.obs")
```

The argument `use = "pairwise.complete.obs"` tells the correlation function that for each pair of genes to use only the pairwise where there is a value for both genes (i.e. neither one can be NA).

Given  $n$  genes, there are  $n \times n$  pairs of correlations, as seen by the dimensions of the correlation matrix.

```
dim(cdc28.correlations)
#> [1] 1000 1000
```

To get the correlations with a gene of interest, we can index with the gene name on the rows of the correlation matrix. For example, to get the correlations between the gene YAR018C and the first 10 genes in the top 1000 set:

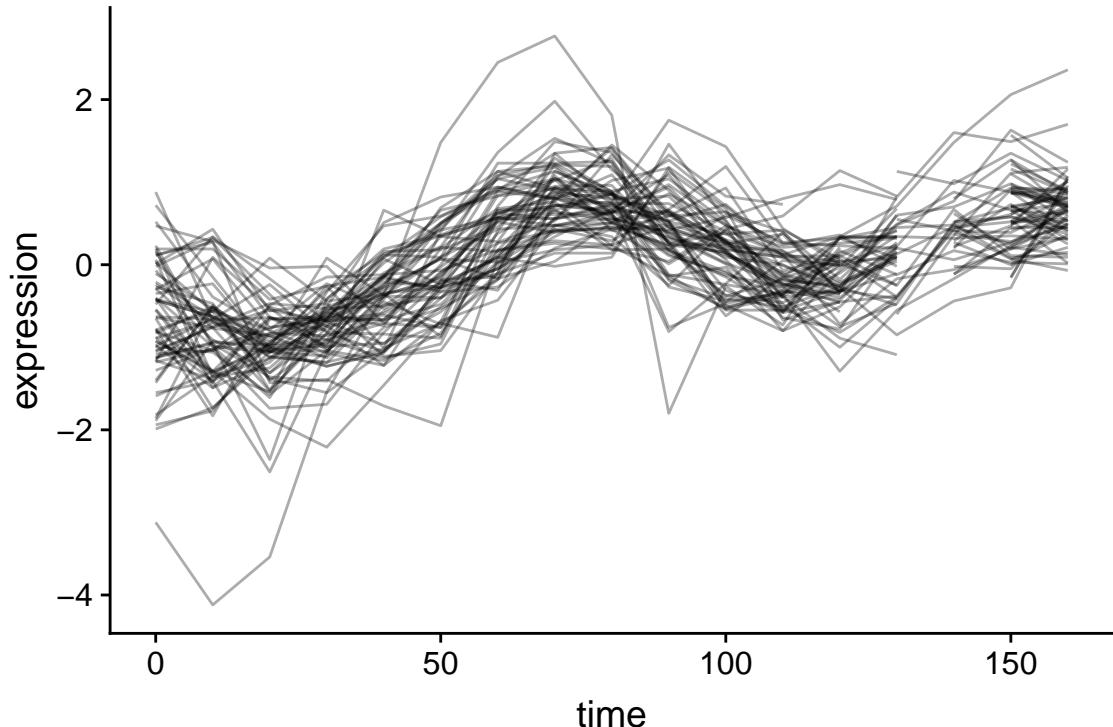
```
cdc28.correlations["YAR018C", 1:10]
#>      YAL003W      YAL005C      YAL022C      YAL028W      YAL035C-A      YAL038W
#>  0.07100626 -0.53315493  0.84741624  0.33379901 -0.22316755 -0.03984599
#>      YAL044C      YAL048C      YAL060W      YAL062W
#>  0.32253692  0.12220221  0.49445700 -0.60972118
```

In the next statement we extract the names of the genes that have correlations with YAR018C greater than 0.6. First we test genes to see if they have a correlation with YAR018C greater than 0.6, which returns a vector of TRUE or FALSE values. This vector of Boolean values is then used to index into the rownames of the correlation matrix, pulling out the gene names where the statement was true.

```
pos.corr.YAR018C <- rownames(cdc28.correlations)[cdc28.correlations["YAR018C", ] > 0.6]
length(pos.corr.YAR018C)
#> [1] 65
```

We then return to our long data to show this set of genes that are strongly positively correlated with YAR018C.

```
top1k.cdc28 %>%
  filter(gene %in% pos.corr.YAR018C) %>%
  ggplot(aes(x = time, y = expression, group = gene)) +
  geom_line(alpha = 0.33) +
  theme(legend.position = "none")
```



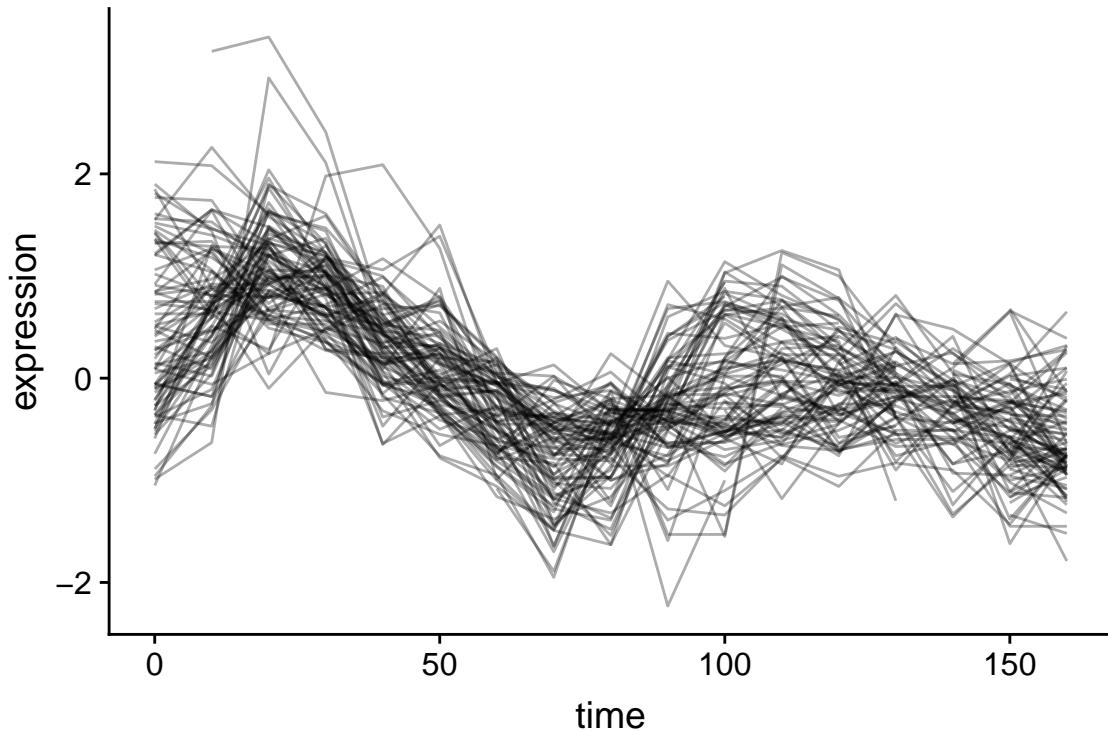
As is expected, genes with strong positive correlations with YAR018C. show similar temporal patterns with YAR018C.

We can similarly filter for genes that have negative correlations with YAR018C.

```
neg.corr.YAR018C <- colnames(cdc28.correlations)[cdc28.correlations["YAR018C",] <= -0.6]
```

As before we generate a line plot showing these genes:

```
top1k.cdc28 %>%
  filter(gene %in% neg.corr.YAR018C) %>%
  ggplot(aes(x = time, y = expression, group = gene)) +
  geom_line(alpha = 0.33) +
  theme(legend.position = "none")
```



### 9.5.2 Adding new columns and combining filtered data frames

Now let's create a new data frame by: 1) filtering on our list of genes that have strong positive and negative correlations with YAR018C; and 2) creating a new variable, "corr.with.YAR018C", which indicates the sign of the correlation. We'll use this new variable to group genes when we create the plot.

```
pos.corr.df <-  
  top1k.cdc28 %>%  
    filter(gene %in% pos.corr.YAR018C) %>%  
    mutate(corr.with.YAR018C = "positive")  
  
neg.corr.df <-  
  top1k.cdc28 %>%  
    filter(gene %in% neg.corr.YAR018C) %>%  
    mutate(corr.with.YAR018C = "negative")  
  
combined.pos.neg <- rbind(pos.corr.df, neg.corr.df)
```

Finally, we plot the data, colored according to the correlation with YAR018C:

```
ggplot(combined.pos.neg,  
       aes(x = time, y = expression, group = gene,  
            color = corr.with.YAR018C)) +  
  geom_line(alpha=0.25) +  
  geom_line(aes(x = time, y = expression),  
            data = filter(top1k.cdc28, gene == "YAR018C"),  
            color = "DarkRed", size = 2, alpha=0.5) +  
  # changes legend title and values for color scale  
  scale_color_manual(name = "Correlation with YAR018C",
```

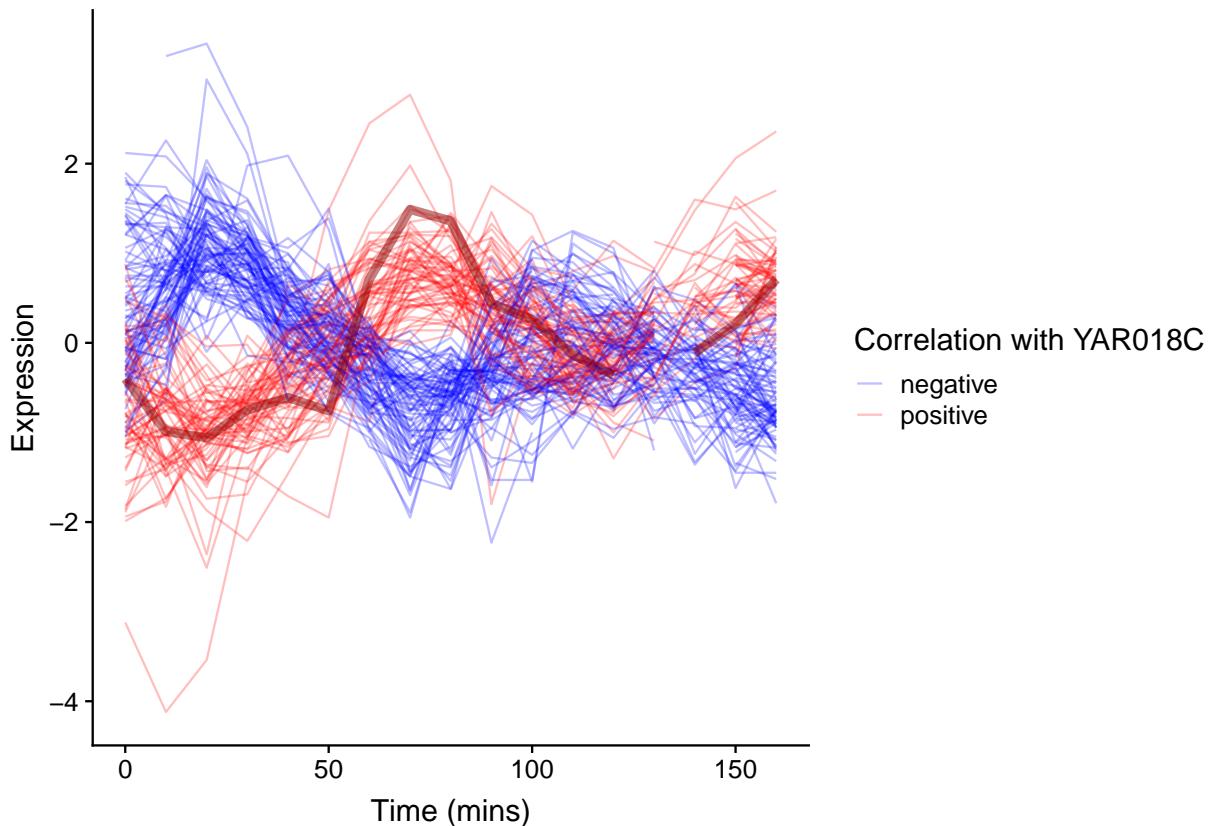
```

values = c("blue", "red")) +
labs(title = "Genes strongly positively and negatively correlated with YAR018C",
subtitle = "YAR018C shown in dark red",
x = "Time (mins)", y = "Expression")

```

### Genes strongly positively and negatively correlated with YAR018C

YAR018C shown in dark red



#### 9.5.3 A heat mapped sorted by correlations

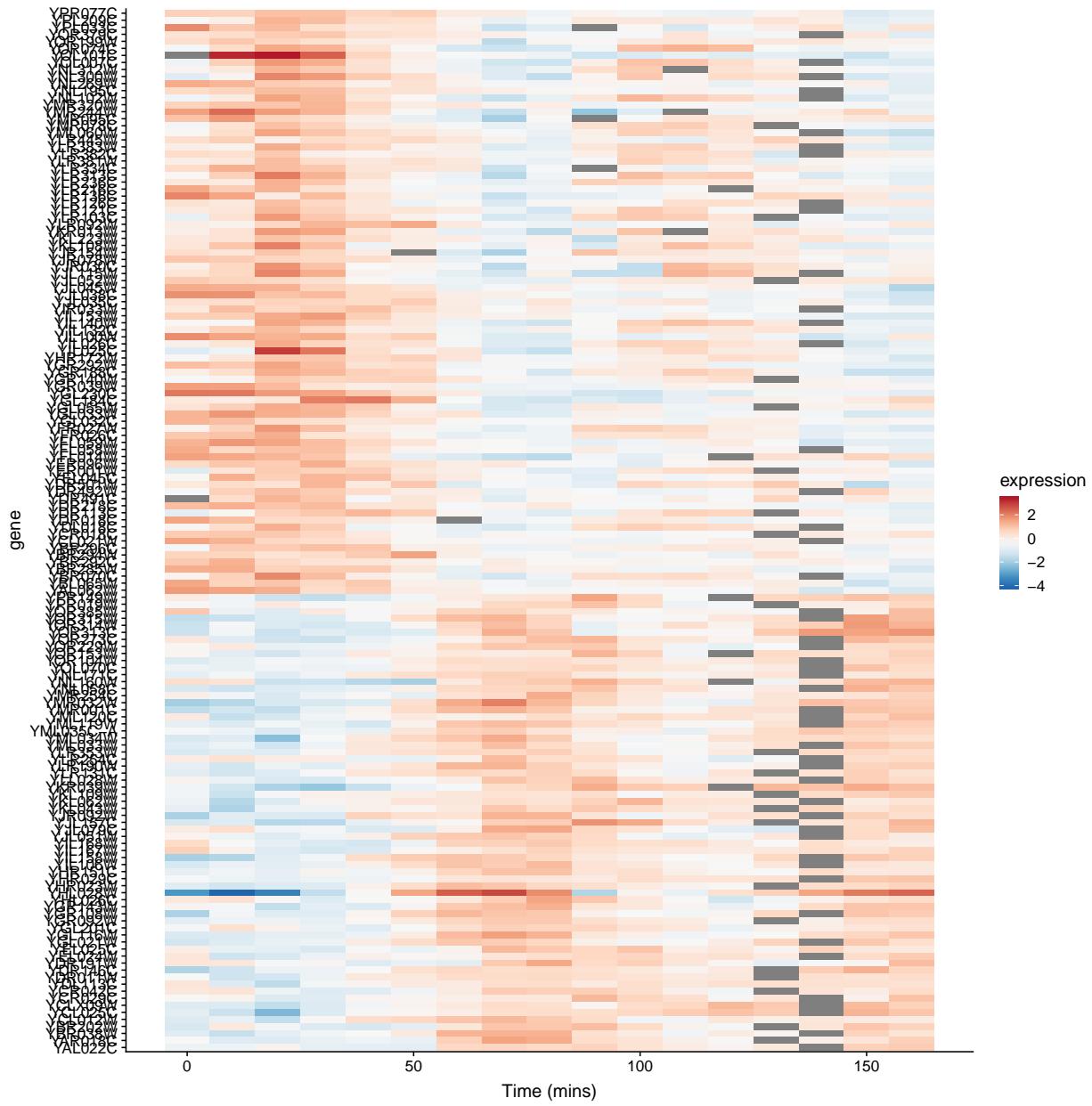
In our previous heat map example figure, we sorted genes according to peak expression. Now let's generate a heat map for the genes that are strongly correlated (both positive and negative) with YAR018C. We will sort the genes according to the sign of their correlation.

```

# re-factor gene names so positive and negative genes are spatially distinct in plot
combined.pos.neg$gene <-
  factor(combined.pos.neg$gene,
  levels = c(pos.corr.YAR018C, neg.corr.YAR018C))

combined.pos.neg %>%
  ggplot(aes(x = time, y = gene)) +
  geom_tile(aes(fill = expression)) +
  scale_fill_gradientn(colors=color.scheme) +
  xlab("Time (mins)")

```



The breakpoint between the positively and negatively correlated sets of genes is quite obvious in this figure.

#### 9.5.4 A “fancy” figure

Recall that we introduced the `cowplot` library in Chapter 6, as a way to combine different `ggplot` outputs into subfigures such as you might find in a published paper. Here we’ll make further use `cowplot` to combine our heatmap and line plot visualizations of genes that covary with `YAR018C`.

```
library(cowplot)
```

`cowplot`’s `draw_plot()` function allows us to place plots at arbitrary locations and with arbitrary sizes onto the canvas. The coordinates of the canvas run from 0 to 1, and the point  $(0, 0)$  is in the lower left corner of the canvas. We’ll use `draw_plot` to draw a complex figure with a heatmap on the left, and two smaller line plots on the right.

```

pos.corr.lineplot <-
  combined.pos.neg %>%
  filter(gene %in% pos.corr.YAR018C) %>%
  ggplot(aes(x = time, y = expression, group = gene)) +
  geom_line(alpha = 0.33, color = 'red') +
  labs(x = "Time (mins)", y = "Expression",
       title = "Genes Positively correlated\nwith YAR018C")

neg.corr.lineplot <-
  combined.pos.neg %>%
  filter(gene %in% neg.corr.YAR018C) %>%
  ggplot(aes(x = time, y = expression, group = gene)) +
  geom_line(alpha = 0.33, color = 'blue') +
  labs(x = "Time (mins)", y = "Expression",
       title = "Genes negatively correlated\nwith YAR018C")

heat.map <-
  ggplot(combined.pos.neg, aes(x = time, y = gene)) +
  geom_tile(aes(fill = expression)) +
  scale_fill_gradientn(colors=color.scheme) +
  labs(x = "Time (mins)", y = "Gene") +
  theme(legend.position = "none")

```

The coordinates of the canvas run from 0 to 1, and the point (0, 0) is in the lower left corner of the canvas. We’ll use `draw_plot` to draw a complex figure with a heatmap on the left, and two smaller line plots on the right.

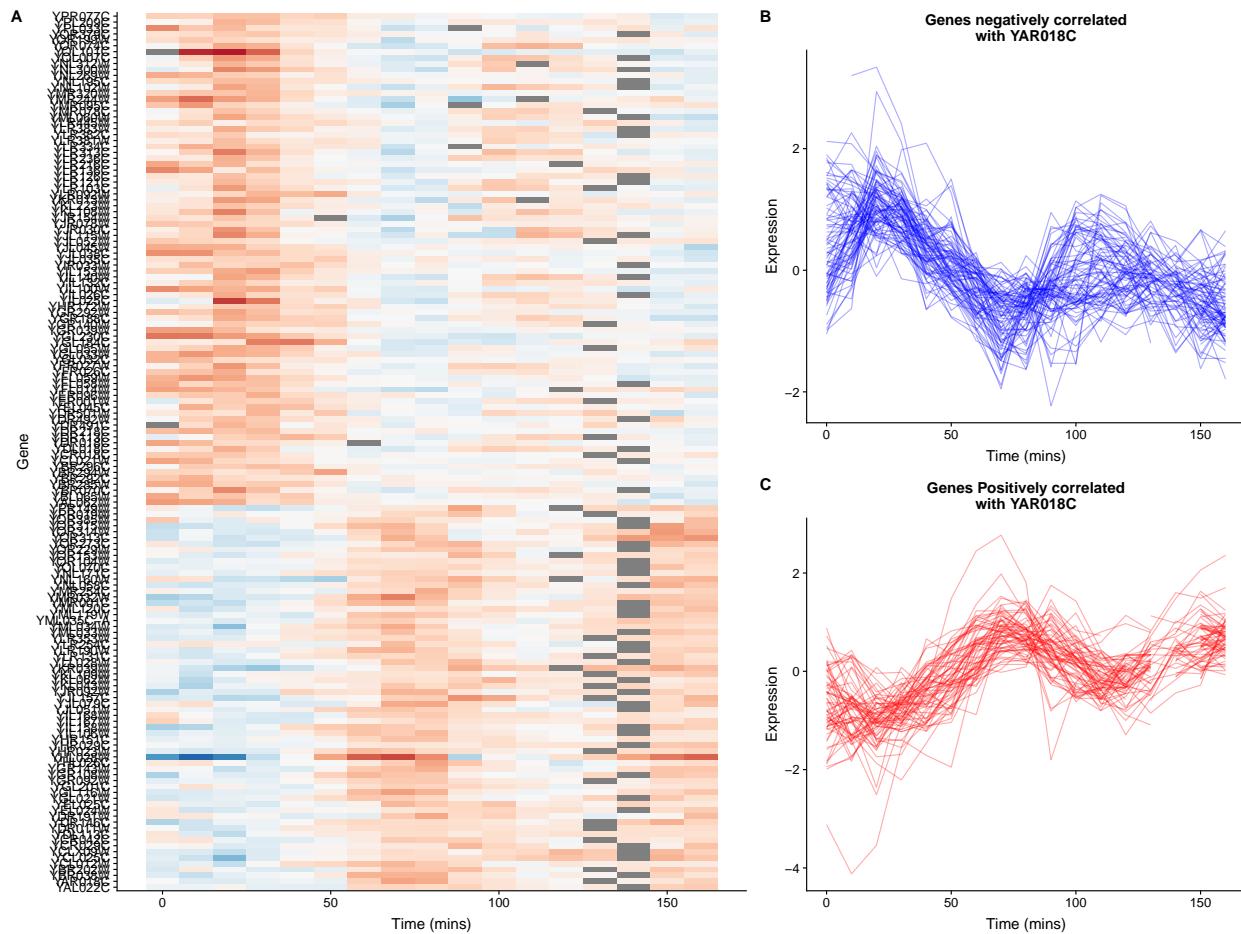
I determined the coordinates below by experimentation to create a visually pleasing layout.

```

fancy.plot <- ggdraw() +
  draw_plot(heat.map, 0, 0, width = 0.6) +
  draw_plot(neg.corr.lineplot, 0.6, 0.5, width = 0.4, height = 0.5) +
  draw_plot(pos.corr.lineplot, 0.6, 0, width = 0.4, height = 0.5) +
  draw_plot_label(c("A", "B", "C"), c(0, 0.6, 0.6), c(1, 1, 0.5), size = 15)

fancy.plot

```



# Chapter 10

## Functions and control flow statements

### 10.1 Writing your own functions

So far we've been using a variety of built in functions in R. However the real power of a programming language is the ability to write your own functions. Functions are a mechanism for organizing and abstracting a set of related computations. We usually write functions to represent sets of computations that we apply frequently, or to represent some conceptually coherent set of manipulations to data.

The general form of an R function is as follows:

```
funcname <- function(arg1, arg2) {  
  # one or more expressions that operate on the fxn arguments  
  # last expression is the object returned  
  # or you can explicitly return an object  
}
```

To make this concrete, here's an example where we define a function to calculate the area of a circle:

```
area.of.circle <- function(r){  
  return(pi * r^2)  
}
```

Since R returns the value of the last expression in the function, the `return` call is optional and we could have simply written:

```
area.of.circle <- function(r){  
  pi * r^2  
}
```

Very short and concise functions are often written as a single line. In practice I'd probably write the above function as:

```
area.of.circle <- function(r) {pi * r^2}
```

The `area.of.circle` function takes one argument, `r`, and calculates the area of a circle with radius `r`. Having defined the function we can immediately put it to use:

```
area.of.circle(3)  
#> [1] 28.27433  
  
radius <- 4
```

```
area.of.circle(radius)
#> [1] 50.26548
```

If you type a function name without parentheses R shows you the function's definition. This works for built-in functions as well (though sometimes these functions are defined in C code in which case R will tell you that the function is a `.Primitive`).

### 10.1.1 Function arguments

Function arguments can specify the data that a function operates on or parameters that the function uses. Function arguments can be either required or optional. In the case of optional arguments, a default value is assigned if the argument is not given.

Take for example the `log` function. If you examine the help file for the `log` function (type `?log` now) you'll see that it takes two arguments, referred to as `x` and `base`. The argument `x` represents the numeric vector you pass to the function and is a required argument (see what happens when you type `log()` without giving an argument). The argument `base` is optional. By default the value of `base` is  $e = 2.71828\dots$ . Therefore by default the `log` function returns natural logarithms. If you want logarithms to a different base you can change the `base` argument as in the following examples:

```
log(2) # log of 2, base e
#> [1] 0.6931472
log(2,2) # log of 2, base 2
#> [1] 1
log(2, 4) # log of 2, base 4
#> [1] 0.5
```

Because base 2 and base 10 logarithms are fairly commonly used, there are convenient aliases for calling `log` with these bases.

```
log2(8)
#> [1] 3
log10(100)
#> [1] 2
```

### 10.1.2 Writing functions with optional arguments

To write a function that has an optional argument, you can simply specify the optional argument and its default value in the function definition as so:

```
# a function to substitute missing values in a vector
sub.missing <- function(x, sub.value = -99){
  x[is.na(x)] <- sub.value
  return(x)
}
```

You can then use this function as so:

```
m <- c(1, 2, NA, 4)
sub.missing(m, -999) # explicitly define sub.value
#> [1] 1 2 -999 4
sub.missing(m, sub.value = -333) # more explicit syntax
#> [1] 1 2 -333 4
sub.missing(m) # use default sub.value
#> [1] 1 2 -99 4
```

```
m # notice that m wasn't modified within the function
#> [1] 1 2 NA 4
```

Notice that when we called `sub.missing` with our vector `m`, the vector did *not* get modified in the function body. Rather a new vector, `x` was created within the function and returned. However, if you did the missing value substitute outside of a function call, then the vector would be modified:

```
n <- c(1, 2, NA, 4)
n[is.na(n)] <- -99
n
#> [1] 1 2 -99 4
```

### 10.1.3 Putting R functions in Scripts

When you define a function at the interactive prompt and then close the interpreter your function definition will be lost. The simple way around this is to define your R functions in a script that you can than access at any time.

In RStudio choose **File > New File > R Script**. This will bring up a blank editor window. Type your function(s) into the editor. Everything in this file will be interpreted as R code, so you should not use the code block notation that is used in Markdown notebooks. Save the source file in your R working directory with a name like `myfxns.R`.

```
# functions defined in myfxns.R

area.of.circle <- function(r) {pi * r^2}

area.of.rectangle <- function(l, w) {l * w}

area.of.triangle <- function(b, h) {0.5 * b * h }
```

Once your functions are in a script file you can make them accessible by using the `source` function, which reads the named file as input and evaluates any definitions or statements in the input file (See also the **Source** button in the R Studio GUI):

```
source("myfxns.R")
```

Having sourced the file you can now use your functions like so:

```
radius <- 3
len <- 4
width <- 5
base <- 6
height <- 7

area.of.circle(radius)
#> [1] 28.27433
area.of.rectangle(len, width)
#> [1] 20
area.of.triangle(base, height)
#> [1] 21
```

Note that if you change the source file, such as correcting a mistake or adding a new function, you need to call the `source` function again to make those changes available.

## 10.2 Control flow statements

Control flow statements control the order of execution of different pieces of code. They can be used to do things like make sure code is only run when certain conditions are met, to iterate through data structures, to repeat something until a specified event happens, etc. Control flow statements are frequently used when writing functions or carrying out complex data transformation.

### 10.2.1 if and if-else statements

`if` and `if-else` blocks allow you to structure the flow of execution so that certain expressions are executed only if particular conditions are met.

The general form of an `if` expression is:

```
if (Boolean expression) {
  Code to execute if
  Boolean expression is true
}
```

Here's a simple `if` expression in which we check whether a number is less than 0.5, and if so assign a value to a variable.

```
x <- runif(1) # runif generates a random number between 0 and 1
face <- NULL # set face to a NULL value

if (x < 0.5) {
  face <- "heads"
}
face
#> NULL
```

The `else` clause specifies what to do in the event that the `if` statement is *not* true. The combined general form of an `if-else` expression is:

```
if (Boolean expression) {
  Code to execute if
  Boolean expression is true
} else {
  Code to execute if
  Boolean expression is false
}
```

Our previous example makes more sense if we include an `else` clause.

```
x <- runif(1)

if (x < 0.5) {
  face <- "heads"
} else {
  face <- "tails"
}

face
#> [1] "heads"
```

With the addition of the `else` statement, this simple code block can be thought of as simulating the toss of a coin.

### 10.2.1.1 if-else in a function

Let's take our “if-else” example above and turn it into a function we'll call `coin.flip`. A literal re-interpretation of our previous code in the context of a function is something like this:

```
# coin.flip.literal takes no arguments
coin.flip.literal <- function() {
  x <- runif(1)
  if (x < 0.5) {
    face <- "heads"
  } else {
    face <- "tails"
  }
  face
}
```

`coin.flip.literal` is pretty long for what it does — we created a temporary variable `x` that is only used once, and we created the variable `face` to hold the results of our `if-else` statement, but then immediately returned the result. This is inefficient and decreases readability of our function. A much more compact implementation of this function is as follows:

```
coin.flip <- function() {
  if (runif(1) < 0.5) {
    return("heads")
  } else {
    return("tails")
  }
}
```

Note that in our new version of `coin.flip` we don't bother to create temporary the variables `x` and `face` and we immediately return the results within the `if-else` statement.

### 10.2.1.2 Multiple if-else statements

When there are more than two possible outcomes of interest, multiple `if-else` statements can be chained together. Here is an example with three outcomes:

```
x <- sample(-5:5, 1) # sample a random integer between -5 and 5

if (x < 0) {
  sign.x <- "Negative"
} else if (x > 0) {
  sign.x <- "Positive"
} else {
  sign.x <- "Zero"
}

sign.x
#> [1] "Negative"
```

### 10.2.2 for loops

A `for` statement iterates over the elements of a sequence (such as vectors or lists). A common use of `for` statements is to carry out a calculation on each element of a sequence (but see the discussion of `map` below)

or to make a calculation that involves all the elements of a sequence.

The general form of a for loop is:

```
for (elem in sequence) {
  Do some calculations or
  Evaluate one or more expressions
}
```

As an example, say we wanted to call our `coin.flip` function multiple times. We could use a for loop to do so as follows:

```
flips <- c() # empty vector to hold outcomes of coin flips
for (i in 1:20) {
  flips <- c(flips, coin.flip()) # flip coin and add to our vector
}
flips
#> [1] "tails" "tails" "heads" "tails" "tails" "tails" "tails" "heads"
#> [9] "heads" "heads" "tails" "tails" "heads" "tails" "tails" "heads"
#> [17] "heads" "heads" "heads" "heads"
```

Let's use a for loop to create a `multi.coin.flip` function that accepts an optional argument `n` that specifies the number of coin flips to carry out:

```
multi.coin.flip <- function(n = 1) {
  # create an empty character vector of length n
  flips <- vector(mode="character", length=n)
  for (i in 1:n) {
    flips[i] <- coin.flip()
  }
  flips
}
```

With this new definition, a single call of `coin.flip` returns a single outcome:

```
multi.coin.flip()
#> [1] "tails"
```

And calling `multi.coin.flip` with a numeric argument returns multiple coin flips:

```
multi.coin.flip(n=10)
#> [1] "heads" "tails" "heads" "heads" "tails" "tails" "tails" "tails"
#> [9] "tails" "tails"
```

### 10.2.2.1 Efficiency tip

An alternate way to write the `multi.coin.flip` function above would be:

```
## This is inefficient, see description below
multi.coin.flip.alt <- function(n = 1) {
  flips <- c()
  for (i in 1:n) {
    flips <- c(flips, coin.flip())
  }
  flips
}
```

If you know the final length of your vector, it is much faster to create an empty vector of the needed length:

e.g., `vector(mode="character", length=n) # runs fast`

than it is to create an empty vector of zero length, and then extend it sequentially:

e.g., `flips <- c(flips, coin.flip()) # runs slow`

### 10.2.3 break statement

A `break` statement allows you to exit a loop even if it hasn't completed. This is useful for ending a control statement when some criteria has been satisfied. `break` statements are usually nested in `if` statements.

In the following example we use a `break` statement inside a `for` loop. In this example, we pick random real numbers between 0 and 1, accumulating them in a vector (`random.numbers`). The `for` loop insures that we never pick more than 20 random numbers before the loop ends. However, the `break` statement allows the loop to end prematurely if the number picked is greater than 0.95.

```
random.numbers <- c()

for (i in 1:20) {
  x <- runif(1)
  random.numbers <- c(random.numbers, x)
  if (x > 0.95) {
    break
  }
}

random.numbers
#> [1] 0.49017818 0.16224965 0.40936666 0.79296667 0.05298049 0.70821049
#> [7] 0.93746360 0.12307803 0.64256986 0.19413841 0.01367272 0.82446620
#> [13] 0.60485234 0.39682428 0.28413612 0.48721146 0.54119176 0.59512634
#> [19] 0.57009194 0.81468425
```

### 10.2.4 repeat loops

A `repeat` loop will loop indefinitely until we explicitly break out of the loop with a `break` statement. For example, here's an example of how we can use `repeat` and `break` to simulate flipping coins until we get a head:

```
ct <- 0
repeat {
  flip <- coin.flip()
  ct <- ct + 1
  if (flip == "heads"){
    break
  }
}

ct
#> [1] 2
```

### 10.2.5 next statement

A `next` statement allows you to halt the processing of the current iteration of a loop and immediately move to the next item of the loop. This is useful when you want to skip calculations for certain elements of a

sequence:

```
sum.not.div3 <- 0

for (i in 1:20) {
  if (i %% 3 == 0) { # skip summing values that are evenly divisible by three
    next
  }
  sum.not.div3 <- sum.not.div3 + i
}
sum.not.div3
#> [1] 147
```

### 10.2.6 while statements

A `while` statement iterates as long as the condition statement it contains is true. In the following example, the `while` loop calls `coin.flip` until “heads” is the result, and keeps track of the number of flips. Note that this represents the same logic as the `repeat-break` example we saw earlier, but in a more compact form.

```
first.head <- 1

while(coin.flip() == "tails"){
  first.head <- first.head + 1
}

first.head
#> [1] 1
```

### 10.2.7 ifelse

The `ifelse` function is equivalent to a `for`-loop with a nested `if-else` statement. `ifelse` applies the specified test to each element of a vector, and returns different values depending on if the test is true or false.

Here's an example of using `ifelse` to replace `NA` elements in a vector with zeros.

```
x <- c(3, 1, 4, 5, 9, NA, 2, 6, 5, 4)
newx <- ifelse(is.na(x), 0, x)
newx
#> [1] 3 1 4 5 9 0 2 6 5 4
```

The equivalent for-loop could be written as:

```
x <- c(3, 1, 4, 5, 9, NA, 2, 6, 5, 4)
newx <- c() # create an empty vector
for (elem in x) {
  if (is.na(elem)) {
    newx <- c(newx, 0) # append zero to newx
  } else {
    newx <- c(newx, elem) # append elem to newx
  }
}
newx
#> [1] 3 1 4 5 9 0 2 6 5 4
```

The `ifelse` function is clearly a more compact and readable way to accomplish this.

## 10.3 map and related tools

Another common situation is applying a function to every element of a list or vector. Again, we could use a `for` loop, but the `map` functions often are better alternatives.

NOTE: `map` is a relative newcomer to R and must be loaded with the `purrr` package (`purrr` is loaded when we load `tidyverse`). Although base R has a complicated series of “apply” functions (`apply`, `lapply`, `sapply`, `vapply`, `mapply`), `map` provides similar functionality with a more consistent interface. We won’t use the `apply` functions in this class, but you may see them in older code.

```
library(tidyverse)
```

---

### 10.3.1 basic `map`

Typically, `map` takes two arguments – a sequence (a vector, list, or data frame) and a function. It then applies the function to each element of the sequence, returning the results as a list.

To illustrate `map`, let’s consider an example with a list of 2-vectors, where each vector gives the min and max values of some variable of interest for individuals in a sample (e.g. resting heart rate and maximum heart rate during exercise). We can use the `map` function to quickly generate the difference between the resting and maximum heart rates:

```
heart.rates <- list(bob = c(60, 120), fred = c(79, 150), jim = c(66, 110))
diff.fxn <- function(x) {x[2] - x[1]}

map(heart.rates, diff.fxn)
#> $bob
#> [1] 60
#>
#> $fred
#> [1] 71
#>
#> $jim
#> [1] 44
```

As a second example, here’s how we could use `map` to get the class of each object in a list:

```
x <- list(c(1,2,3), "a", "b", list(lead = "Michael", keyboard = "Jermaine"))
map(x, class)
#> [[1]]
#> [1] "numeric"
#>
#> [[2]]
#> [1] "character"
#>
#> [[3]]
#> [1] "character"
#>
#> [[4]]
#> [1] "list"
```

---

### 10.3.2 `map_if` and `map_at`

`map_if` is a variant of `map` that takes a predicate function (a function that evaluates to TRUE or FALSE) to determine which elements of the input sequence are transformed by the map function. All elements of the sequence that do not meet the predicate are left un-transformed. Like `map`, `map_if` always returns a list.

Here's an example where we use `map_if` to apply the `stringr::str_to_upper` function to those columns of a data frame that are character vectors, and apply `abs` to obtain the absolute value of a numeric column:

```
a <- rnorm(6)
b <- c("a", "b", "c", "d", "e", "f")
c <- c("u", "v", "w", "x", "y", "z")
df <- data_frame(a, b, c)
head(df)
#> # A tibble: 6 x 3
#>       a     b     c
#>   <dbl> <chr> <chr>
#> 1  0.3854 a      u
#> 2 -1.468  b      v
#> 3  0.6795 c      w
#> 4  0.4681 d      x
#> 5  1.645   e      y
#> 6  1.289   f      z

df2 <- map_if(df, is.character, str_to_upper)
df2 <- map_if(df2, is.numeric, abs)
head(df2)
#> $a
#> [1] 0.3854417 1.4683696 0.6794608 0.4681072 1.6448192 1.2885668
#>
#> $b
#> [1] "A" "B" "C" "D" "E" "F"
#>
#> $c
#> [1] "U" "V" "W" "X" "Y" "Z"
```

Note that `df2` is a list, not a data frame. We can convert `df2` to a data frame `df3`:

```
# Next, create data frame df3
df3 <- data_frame(df2$a, df2$b, df2$c)
head(df3)
#> # A tibble: 6 x 3
#>   `df2$a` `df2$b` `df2$c`
#>   <dbl> <chr>   <chr>
#> 1  0.3854 A       U
#> 2  1.468  B       V
#> 3  0.6795 C       W
#> 4  0.4681 D       X
#> 5  1.645   E       Y
#> 6  1.289   F       Z
```

But it requires an extra step to rename the columns:

```
# Check column names, then rename
names(df3)
#> [1] "df2$a" "df2$b" "df2$c"
names(df3) <- c("a", "b", "c")
```

```
head(df3)
#> # A tibble: 6 x 3
#>   a     b     c
#>   <dbl> <chr> <chr>
#> 1 0.3854 A     U
#> 2 1.468   B     V
#> 3 0.6795 C     W
#> 4 0.4681 D     X
#> 5 1.645   E     Y
#> 6 1.289   F     Z
```

If our goal is to apply functions to the columns of a data frame, it may be easier with `mutate`:

```
df4 <- df %>% as_tibble() %>%
  mutate(a = abs(a),
        b = str_to_upper(b),
        c = str_to_upper(c))

head(df4)
#> # A tibble: 6 x 3
#>   a     b     c
#>   <dbl> <chr> <chr>
#> 1 0.3854 A     U
#> 2 1.468   B     V
#> 3 0.6795 C     W
#> 4 0.4681 D     X
#> 5 1.645   E     Y
#> 6 1.289   F     Z
```

### 10.3.3 mapping in parallel using `map2`

The `map2` function applies a transformation function to two sequences in parallel. The following example illustrates this:

```
first.names <- c("John", "Mary", "Fred")
last.names <- c("Smith", "Hernandez", "Kidogo")

map2(first.names, last.names, str_c, sep = " ")
#> [[1]]
#> [1] "John Smith"
#>
#> [[2]]
#> [1] "Mary Hernandez"
#>
#> [[3]]
#> [1] "Fred Kidogo"
```

Note how we can specify arguments to the transformation function as additional arguments to `map2` (i.e., the `sep` argument gets passed to `str_c`)

### 10.3.4 `map` variants that return vectors

`map`, `map_if`, and `map_at` always return lists. The `purrr` library also has a series of `map` variants that return vectors:

- `map_lgl` (for logical vectors)
- `map_chr` (for character vectors)
- `map_int` (integer vectors)
- `map_dbl` (double vectors)

```
# compare the outputs of map and map_chr
a <- map(letters[1:6], str_to_upper)
str(a)
#> List of 6
#> $ : chr "A"
#> $ : chr "B"
#> $ : chr "C"
#> $ : chr "D"
#> $ : chr "E"
#> $ : chr "F"

b <- map_chr(letters[1:6], str_to_upper)
str(b) # a vector
#> chr [1:6] "A" "B" "C" "D" "E" "F"
```

Here's an example using `map_dbl`, where we create a data frame with three columns, and compute the median of each column:

```
# Make data frame for analysis
df <- tibble(a = rnorm(100), b = rnorm(100), c = rnorm(100))

map_dbl(df, median) # median of each column of df
#>           a           b           c
#>  0.1430839 -0.0463604  0.1669213
```

## **Chapter 11**

# **Frequency Distributions and Descriptive Statistics**

See the lecture slides.



## Chapter 12

# Joint Frequency Distributions and Measures of Association

See the lecture slides.



# Chapter 13

## Introduction to Probability

The exposition here largely follows that in Chapter 5 of Whitlock & Schluter.

### 13.1 Terms

- *Outcome* – the result of a process or experiment
- *Random Trial* – a process or experiment that has two or more possible outcomes whose occurrence can not be predicted with certainty
- *Event* – a subset of the possible outcomes of a random trial

#### 13.1.1 Examples of random trials, outcomes, and events

##### 13.1.1.1 Random trials

Classic examples of random trials

- flipping a coin
- rolling a die
- choosing a shuffled deck
- picking three balls, without replacement, from an urn filled with 3 white and 2 black balls

Biological examples of random trials

- Determining the sex of offspring in a genetic cross
- Weight loss/gain following treatment with a drug
- Count the plant species in one acre of forest

##### 13.1.1.2 Outcomes

Classic examples:

- Coins: heads or tails
- Dice: The numbers 1 to  $n$ , where  $n$  is the number of sides on the die that was thrown
- Cards: Any of the numbered or face cards and their suits
- Balls and urns: the number of black and white balls drawn

Biological examples:

- Sex of offspring: male and female
- Weight loss/gain: positive and negative real numbers in an interval
- Species: integers values  $\geq 0$

### 13.1.1.3 Events

Classic examples:

- Coins: heads or tails (the events are the outcomes when the experiment is a single flip)
- Dice: rolled a specific number, rolled an even number, rolled a number greater than three, etc
- Cards: drew a face card, drew a heart, drew an ace of hearts, etc
- Balls and urns: all white balls, only one white ball, etc

Biological examples:

- Sex of offspring: male or female
- Weight loss/gain: lost weight, lost more than 5 lbs, gained between 10 and 20 lbs, etc
- Species: counted 10 species, counted more than 25 species, etc

## 13.2 Frequentist definition of probability

*Probability* of an event – the proportion of times the event would occur if we repeated a random trial an infinite (or very large) number of times under the same conditions.

- To indicate the probability of an event  $A$ , we write  $P(A)$
- The probability of all possible outcomes of a random trial must sum to one.
- The *complement* of an event is all the possible outcomes of a random trial that are *not* the event. Let  $A^c$  indicate the complement of the event  $A$ . Then  $P(A^c) = 1 - P(A)$

### 13.2.1 Examples: Probability

#### 13.2.1.1 Classic examples

In classic probabilistic examples, where we understand (approximately) the physical constraints and symmetries of a random trial, we can often assign theoretical probabilities:

- Coins: With a fair coin, the probability of each face is 0.5
- Dice: Given a fair 20-side die, the probability of rolling a 15 or better is  $6/20 = 0.3$
- Cards: In randomly shuffled standard (French) 52-card deck, the probability of drawing a heart is  $13/54 = 0.25$ ; the probability of getting an ace is  $4/52 = \sim 0.077$ ; the probability of drawing the ace of hearts is  $1/52 = \sim 0.0192$  The probability of not drawing an ace is  $1 - 1/52 = \sim 0.9808$
- Balls and urns: If you make three draws (without replacement) from an urn filled with three white and two black balls, the probability of drawing three white balls is 0.1. We'll illustrate how to calculate this below.

#### 13.2.1.2 Biological examples

For real-world examples, we can not usually invoke physical symmetries to assign theoretical probabilities *a priori* to an event (though sometimes we'll use such symmetries when stating “null hypotheses”; more on this in a later lecture),

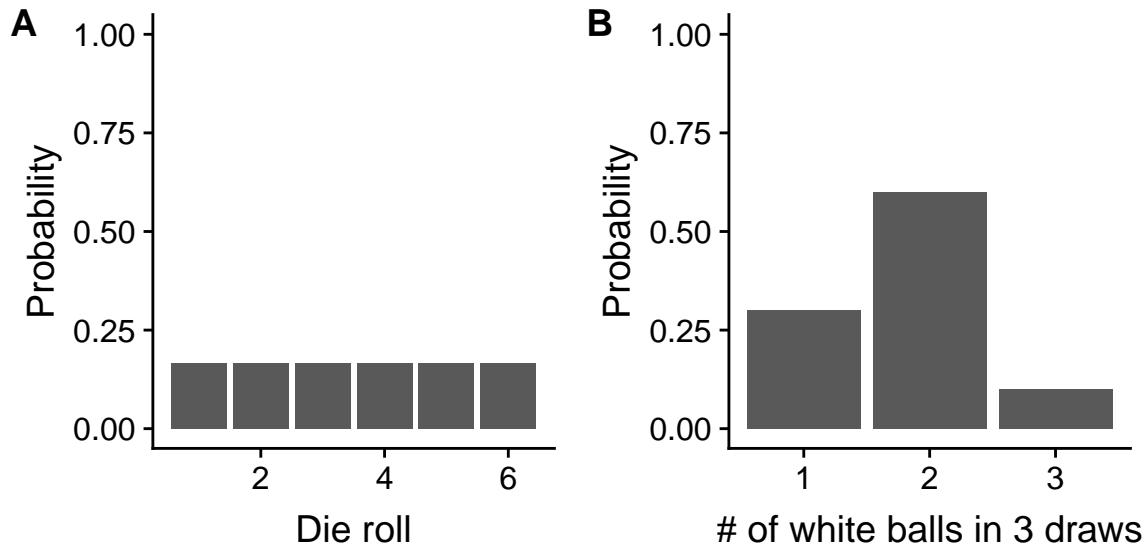


Figure 13.1: Discrete probability distributions. A) Probability distribution for a single roll of a fair 6-sided die; B) Probability distribution for the number of white balls observed in three draws, without replacement, from an urn filled with 3 white balls and 2 black balls.

- Sex of offspring in humans: In human populations, the *sex ratio at birth* is *not* 1:1. The probability of a child being male is ~0.512, and the probability of having a female child is ~0.488. This surprising deviation from the 1:1 ratio is well documented. Additional factors contributes to further deviations in actual human populations. See for example Hesketh and Xing (2006), Abnormal sex ratios in human populations: causes and consequences. PNAS 103(36):13271-5.

### 13.3 Probability distribution

*Probability distribution* – A list, or equivalent representation, of the probabilities of all mutually exclusive outcomes of a random trial. Two events are mutually exclusive if than cannot both occur at the same time. The total probabilities in a probability distribution sums to 1.

For most cases of biological interest, probability distributions are unknowable and thus we use relative frequency distributions to estimate the underlying probability distributions of interest (relative frequencies are sometimes referred to as empirical probabilities).

Referring back to our earlier “frequentist definition”, another way of thinking about a probability distribution is as relative frequency distribution as the number of observations approaches the size (in some cases infinite) of the population under study (using the broad definition of “population” discussed several lectures ago)

#### 13.3.1 Discrete probability distribution

*Discrete probability distribution* – the probability of each possible value of a discrete variable. Discrete probability distributions apply to categorical variables, ordinal variables, and discrete numerical variables. The total probabilities must sum to one.

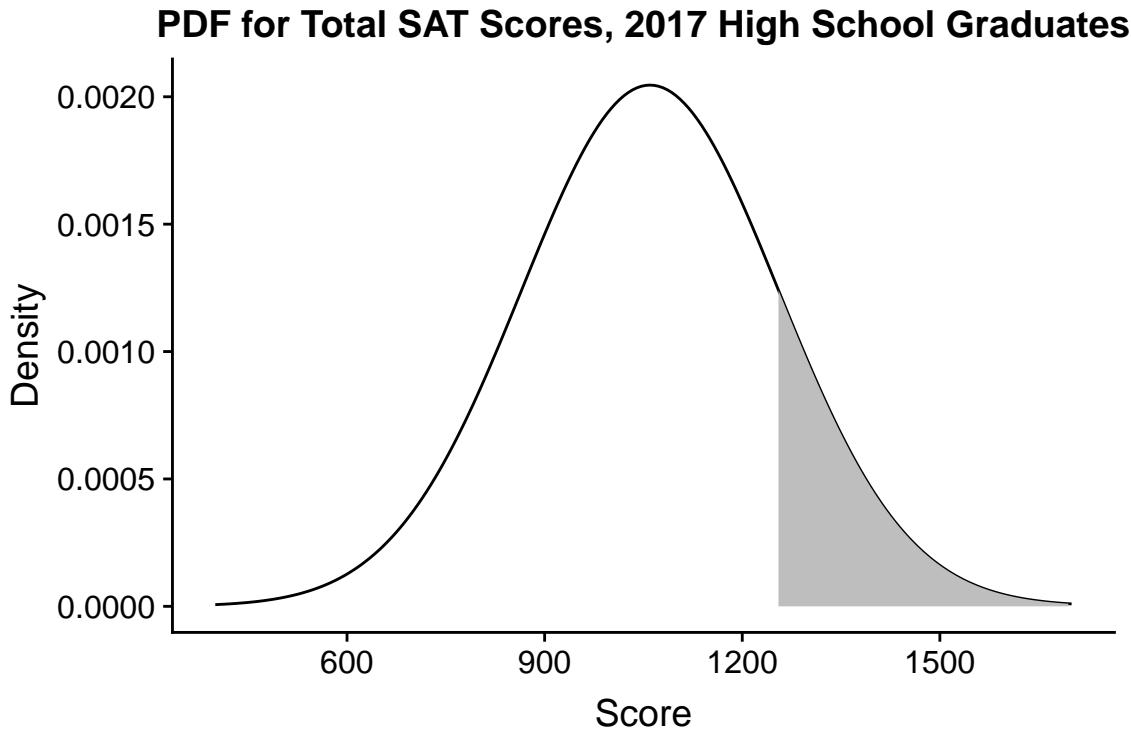


Figure 13.2: Figure 2. Distribution of total SAT scores for 2017 high school graduates. Assuming a normal distribution with mean = 1060, standard deviation = 195, based on data reported in the [2017 SAT annual report](<https://reports.collegeboard.org/pdf/2017-total-group-sat-suite-assessments-annual-report.pdf>). The probability that a randomly chosen student got a score better than 1255 is represented by the shaded area;  $P(\text{Score} > 1255) = 0.1587$ .

### 13.3.2 Continuous probability distribution

*Continuous probability distribution* – for continuous numerical variables we do not assign probability to specific numerical values, but rather to numerical intervals. We represent a continuous probability distribution using a “Probability Density Function” (PDF). The integral of a PDF over an interval gives the probability that the variable represented by that PDF lies within the specified interval.

## 13.4 Mutually exclusive events

*Mutually exclusive events* are events that can *not* both occur *simultaneously* in the same random trial.

### 13.4.1 Addition rule, mutually exclusive events

If A and B are mutually exclusive, then the probability of either event occurring is the sum of their individual probabilities:

$$P(A \text{ or } B) = P(A) + P(B)$$

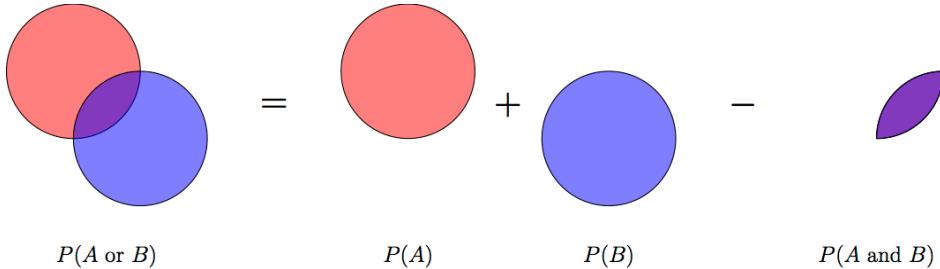


Figure 13.3: Graphical illustration of the general addition rule for probabilities.

## 13.5 Independence

- *Independence* – two events are independent if the occurrence of one does not inform us about the probability that the second.
- *Dependence* – any events that are not independent are considered to be *dependent*.

### 13.5.1 Multiplication rule, independent events

The simple version of the multiplication rules states that if events A and B are independent than:

$$P(A \text{ and } B) = P(A)P(B)$$

## 13.6 General addition rule

The general form of the addition rule states:

$$P(A \text{ or } B) = P(A) + P(B) - P(A \text{ and } B)$$

Graphically, this can be represented as:

## 13.7 Conditional probability

*Conditional probability* – is the probability that an event occurs given that a condition is met.

- Denoted:  $P(A|B)$ . Read this as “the probability of A given B” or “the probability of A conditioned on B”.

### 13.7.1 Example: Conditional probability

Consider our urns and balls example, in which we make three draws (without replacement) from an urn filled with three white balls and two black balls.

- The initial probability of drawing a black ball,  $P(B) = 2/5 = 0.4$
- If the first draw was a white ball, the probability of drawing a black ball is now,  $P(B|\text{1st ball was white}) = 2/4 = 0.5$

## 13.8 General multiplication rule

The general form of the multiplication rule is:

$$P(A \text{ and } B) = P(A)P(B|A)$$

### 13.8.1 Example: General multiplication rule

Consider our urns and balls example again. What is the probability that you draw, without replacement, three balls and they're all white?

- The initial probability of drawing a white ball in the first draw is  $P(W) = 3/5$
- The probability of drawing a white ball in the second draw, conditional on the first ball being white is  $P(W|\text{1st White}) = 1/2$
- Therefore the  $P(\text{1st White and 2nd White}) = P(W)P(W|\text{first White}) = 3/10$
- The probability of drawing a white ball in the third draw, conditional on the first two balls being white is  $P(W|\text{1st White and 2nd White}) = 1/3$
- Therefore the  $P(\text{1st White and 2nd White and 3rd White}) = P(\text{1st White and 2nd White})P(W|\text{1st White and 2nd White}) = (3/10)(1/3) = 1/10$

## 13.9 Probability trees

Probability trees are diagrams that help calculate the probabilities of combinations of events across multiple random trials. A probability tree for the urns and balls example follows below.

The nodes in a probability tree represent the possible outcomes of each random trial. A path from the root node to one of the tips of the probability tree represents a sequence of outcomes resulting from the successive random trials. Along the edges of the probability tree we write the probability of each outcome for each trial. The probability of a specific sequence of outcomes is calculated by multiplying the probabilities along the path that represents that sequence.

To solve the problem we looked at previously – the probability of getting all white balls in three draws from the urn without replacement – we find all the sequences that yield this outcome. In this case there is only one sequence that results in three white balls. The product of the probabilities along the path representing this sequence is  $3/5 \times 2/4 \times 1/3 = 1/10$ .

Another way to think about a probability tree is that values along the branches at a given point in the tree represent the conditional probabilities of the next possible outcomes, given all the previous outcomes.

## 13.10 Law of total probability

The law of total probability states:

$$P(A) = \sum_{\text{all values of } B} P(B)P(A|B)$$

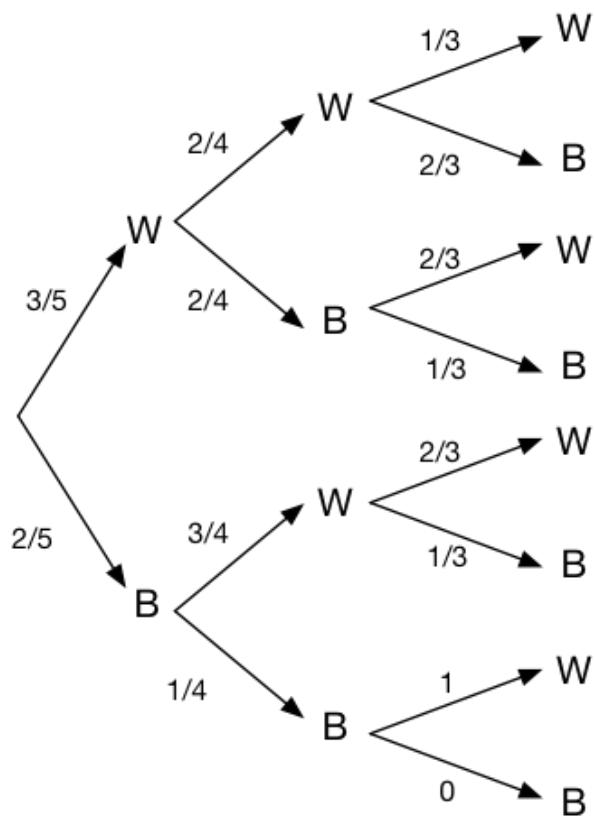


Figure 13.4: Probability tree for the urns and balls example



# Chapter 14

## Introduction to Sampling Distributions, Part I

### 14.1 Libraries

```
library(tidyverse)
set.seed(20181019) # initializes RNG
```

### 14.2 In class experiments

#### 14.2.1 Experiment 1

1. Sample 10 grains of rice from the large population provided by the instructor
2. Count the number of brown grains in your sample
3. Enter your counts in this Google Sheets spreadsheet
4. Once all the entire class has entered their data, download the spreadsheet as a CSV file, load it into R as a data frame (`rice`)
5. Add a new column to the data frame that gives the proportion (relative frequency) of brown rice in each student's sample
6. Generate a frequency distribution plot for the proportion of brown rice
7. Discuss as a class

Key points:

- We all did the same experiment
- We all sampled from the same population
- We all came up with slightly different estimates of the proportion of brown rice grains
- If we take all of our individual estimates and combine, we have a distribution of estimated proportions

#### 14.2.2 Experiment 2

1. Sample 30 grains of rice from the large population provided by the instructor
2. Count the number of brown grains in your sample
3. Add your new counts as additional rows to the Google Sheets spreadsheet

4. Re-download and re-load the data, again estimating the proportion of brown rice for each student's sample
5. Generate a faceted plot giving the frequency distribution for the proportion of brown rice in samples of size 10 and samples of size 30.
6. Discuss as a class

Key points: - With larger sample sizes our estimates still very - But the spread of our estimates has decreased

## 14.3 Simulating sampling in R

Before we can simulate a sampling experiment in a computer we have to make some assumptions about the probability distribution of the variables we're simulating.

To illustrate this let's make explicit what's going on in our rice grain experiment:

- We have a large population of rice grains
- A fraction,  $p$ , of grains in that population are brown
- We draw  $n$  grains of rice from the population; all of the draws are independent
- We count the number of brown rice grains, recording the value as  $k$

### 14.3.1 Binomial distribution

Under this scenario, what is the probability the we drew  $k$  brown grains from a sample of size  $n$ , if the true proportion of brown grains is  $p$ ?

It turns out we can solve this problem for a the following formula:

$$P(k; n, p) = \binom{n}{k} p^k (1-p)^{n-k}$$

Where:

- $\binom{n}{k}$  is called the “binomial coefficient” and it gives the different combinations that would result in  $k$  successes (brown grains) in  $n$  draws. For example, in our rice experiment one combination for  $k = 2$  would be: (brown, brown, white, white...); a second combination could be (brown, white, brown, white, white, ...)
- $p^k (1-p)^{n-k}$  is the probability of  $k$  successes (brown grains) and  $(n - k)$  failures (white grains) for each of the combinations

The above formula is the “Probability Mass Function” (pmf) for the **Binomial Distribution** – it gives the probability of a particular outcome. The complete binomial distribution for  $n$  trials given a probability of success  $p$  is written as  $B(n, p)$  and is calculated by evaluating the probability mass function for all values of  $k$  from  $0, \dots, n$ .

#### 14.3.1.1 Binomial distribution in R

R has a built-in function, `dbinom()` that calculates the probability mass function of the binomial distribution at given values of  $k$ . Take a moment to read the help on the `dbinom()` function in R.

To calculate the Binomial pmf for  $k = 5, n = 10, p = 0.3$  we call `dbinom()` as so:

```
dbinom(5, 10, 0.3)
#> [1] 0.1029193
```

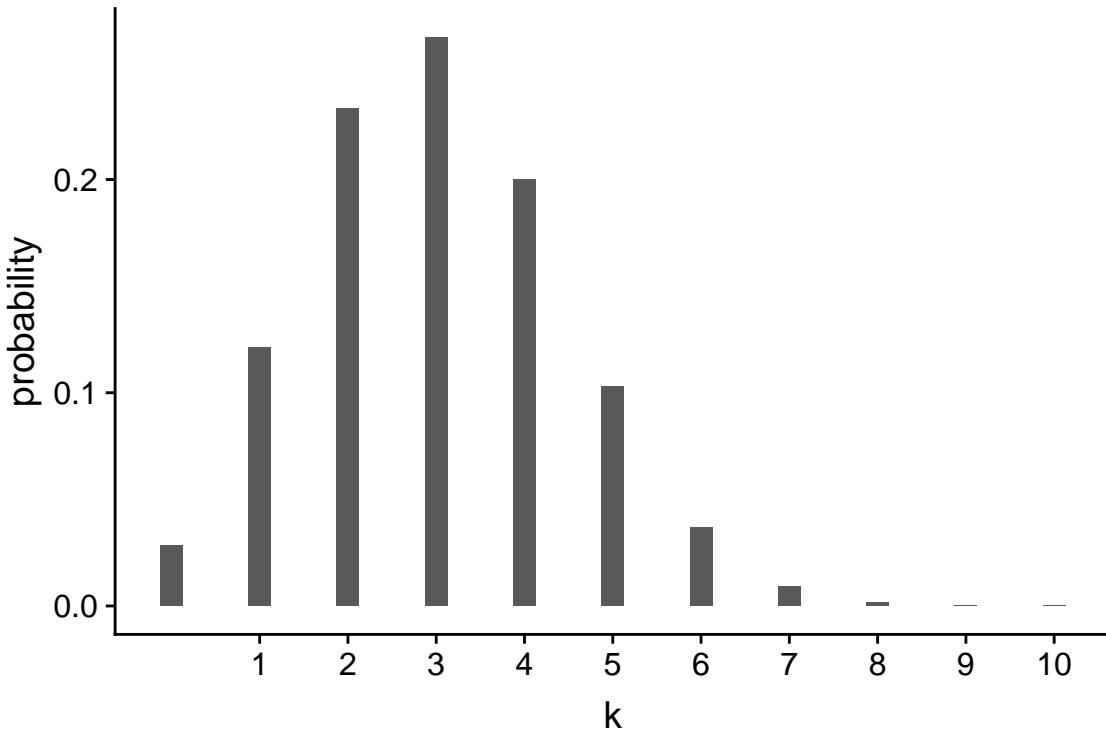
This tells us that if the true proportion of brown rice in the population is 0.3, about 10% of samples of size 10 will have 5 brown grains in them.

The `dbinom()` function can also take a vector of values of  $k$  as its first argument. Here we evaluate all values of  $k$  between 0 and 10, for the same scenario ( $n = 10$  draws; probability of success  $p = 0.3$ ), storing the results in a data frame.

```
k <- 0:10
binomial.ex1 <- data_frame(k = k,
                             probability = dbinom(k, 10, 0.3))
binomial.ex1
#> # A tibble: 11 x 2
#>       k   probability
#>   <int>     <dbl>
#> 1     0 0.02825
#> 2     1 0.1211
#> 3     2 0.2335
#> 4     3 0.2668
#> 5     4 0.2001
#> 6     5 0.1029
#> 7     6 0.03676
#> 8     7 0.009002
#> 9     8 0.001447
#> 10    9 0.0001378
#> 11   10 0.000005905
```

And now plotting the distribution, using `geom_col()` which is like `geom_bar()` but that derives the height of the bars to plot directly from the specified `y` aesthetic.

```
binomial.ex1 %>%
  ggplot(aes(x = k, y = probability)) +
  geom_col(width=0.25) +
  scale_x_continuous(breaks=1:10)
```



### 14.3.2 Sampling from the binomial distribution in R

To simulate sampling from the binomial distribution in R we can use the `rbinom()` function.

The arguments to `rbinom()` the number of samples, the number of draws in each sample, and the probability of success. For example, to simulate the single sample of size 10 that you generated at the beginning of class (making an assumption about the actual probability in the population) you would call `rbinom()` like so:

```
# draw 1 sample of size 10, where P(success) = 0.3
rbinom(1, 10, 0.3)
#> [1] 2
```

The value returned by `rbinom()` is the number of successes.

You can also specify more than one sample to be generated. For example, there are roughly 20 students in today's class session. To simulate all of our 20 samples we can call `rbinom()` as so:

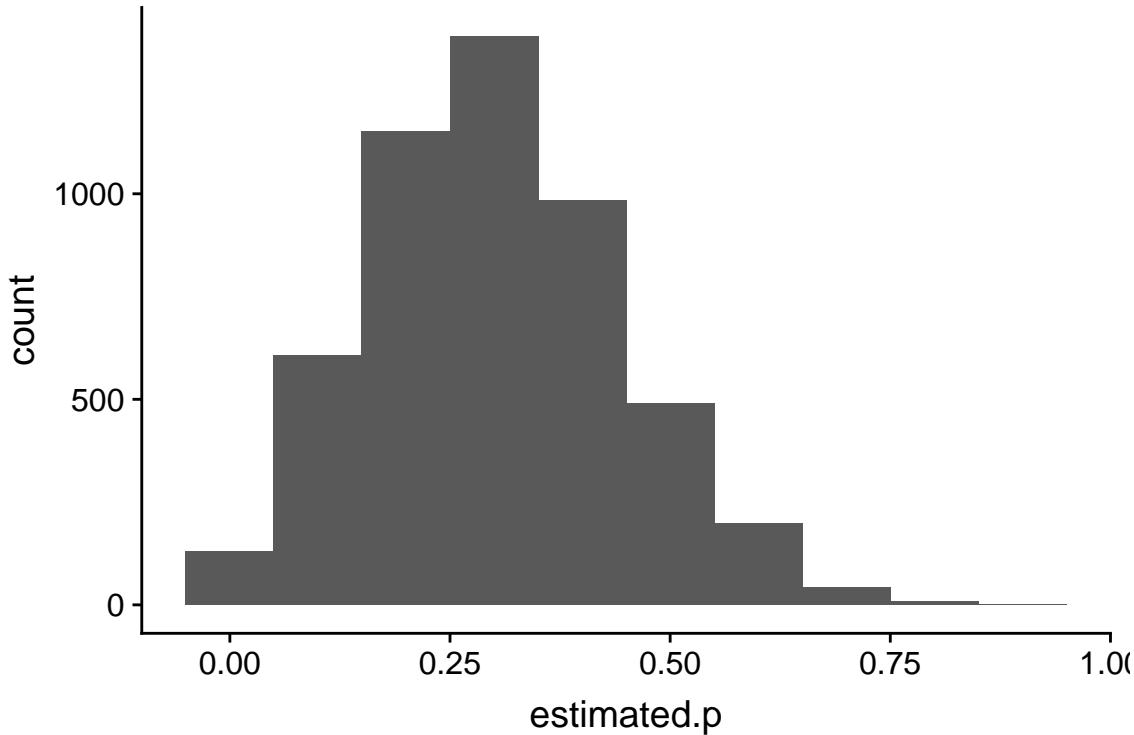
```
# 20 samples of size 10, where P(success) = 0.3
rbinom(20, 10, 0.3)
#> [1] 1 3 3 2 5 6 2 1 3 2 3 3 3 3 5 3 6 2 5 2
```

Let's repeat that, but for several thousand samples of size 10, and then plot the results in terms of the proportion of successes:

```
nsamples = 5000
ssize = 10
p = 0.3

df10 <- data_frame(ssize = rep(ssize, nsamples),
                     k = rbinom(nsamples, ssize, p),
                     estimated.p = k/ssize)
```

```
df10 %>%
  ggplot(aes(x = estimated.p)) +
  geom_histogram(bins=10)
```



Here we're plotting a distribution of the estimates of the proportion of successes (e.g. relative frequency of brown rice grains) in many samples of size 10.

## 14.4 Distribution of estimates of the proportion

Repeat the simulation, but for samples of size 30:

```
nsamples = 5000
ssize = 30
p = 0.3

df30 <- data_frame(ssize = rep(ssize, nsamples),
                     k = rbinom(nsamples, ssize, p),
                     estimated.p = k/ssize)
```

Repeat the simulation, but for samples of size 100:

```
nsamples = 5000
ssize = 100
p = 0.3

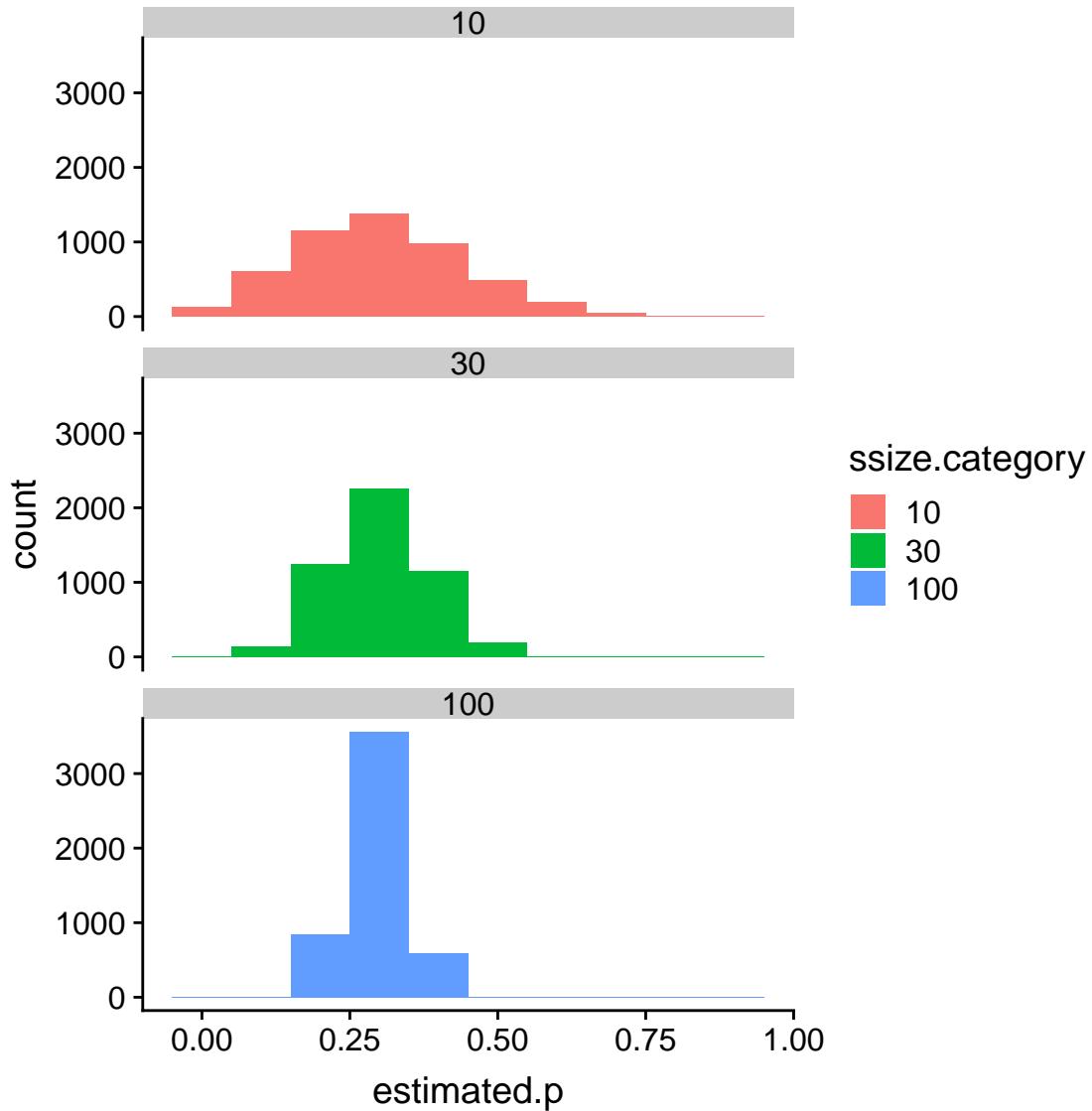
df100 <- data_frame(ssize = rep(ssize, nsamples),
                      k = rbinom(nsamples, ssize, p),
                      estimated.p = k/ssize)
```

Combine the three data frames:

```
df.combined <- bind_rows(df10, df30, df100)
```

Plot distributions of estimates of  $p$  for different sample sizes:

```
df.combined %>%
  mutate(ssize.category = as.factor(ssize)) %>%
  ggplot(aes(x = estimated.p, fill = ssize.category)) +
  geom_histogram(bins=10) +
  facet_wrap(~ssize.category, ncol=1)
```



# Chapter 15

## Introduction to Sampling Distributions, Part II

Usually when we collect biological data, it's because we're trying to learn about some underlying "population" of interest. Population here could refer to an actual population (e.g. all males over 20 in the United States; brushtail possums in the state of Victoria, Australia), an abstract population (e.g. corn plants grown from Monsanto "round up ready" seed; yeast cells with genotypes identical to the reference strain S288c), outcomes of a stochastic process we can observe and measure (e.g. meiotic recombination in flies; hadrons detected at the LHC during a particle collision experiment), etc.

It is often impractical or impossible to measure all objects/individuals in a population of interest, so we take a **sample** from the population and make measurements on the variables of interest in that sample. We do so with the hope that the various statistics we calculate on the variables of interest in that sample will be useful estimates of those same statistics in the underlying population.

However, we must always keep in mind that the statistics we calculate from our sample will almost never exactly match those of the underlying population. That is when we collect a sample, and measure a statistic (e.g. mean) on variable X in the sample, there is a degree of *uncertainty* about how well our estimate matches the true value of that statistic for X in the underlying population.

*Statistical inference* is about quantifying the uncertainty associated with statistics and using that information to test hypotheses and evaluate models.

Today we're going to review a fundamental concept in statistical inference, the notion of a *sampling distribution* for a statistic of interest. A sampling distribution is the probability distribution of a given statistic for samples of a given size. Traditionally sampling distributions were derived analytically. In this class session we'll see how to approximate sampling distributions for any a statistic using computer simulation.

### 15.1 Libraries

```
library(tidyverse)
library(magrittr)
library(stringr)
```

## 15.2 Data set: Simulated male heights

To illustrate the concept of sampling distributions, we'll use a simulated data set to represent the underlying population we're trying to estimate statistics for. This will allow us to compare the various statistics we calculate and their sampling distributions to their “true” values.

Let's simulate a population consisting of 25,000 individuals with a single trait of interest – height (measured in centimeters). We will simulate this data set based on information about the distribution of the heights of adult males in the US from a study carried out from 2011-2014 by the US Department of Health and Human Services<sup>1</sup>.

```
# male mean height and sd in centimeters from USDHHS report
true.mean <- 175.7
true.sd <- 15.19
```

### 15.2.1 Properties of the underlying population

Heights in human populations are approximately normally distributed, so we'll assume that the distribution of our simulated variable is also normally distributed. Let's take a moment to visualize the probability distribution of a normal distribution with a mean and standard deviation as given above. Here we use the `dnorm()` function to generate the probability density for different heights.

```
pop.distn <-
  data_frame(height = seq(100, 250, 0.5),
             density = dnorm(height, mean = true.mean, sd = true.sd))

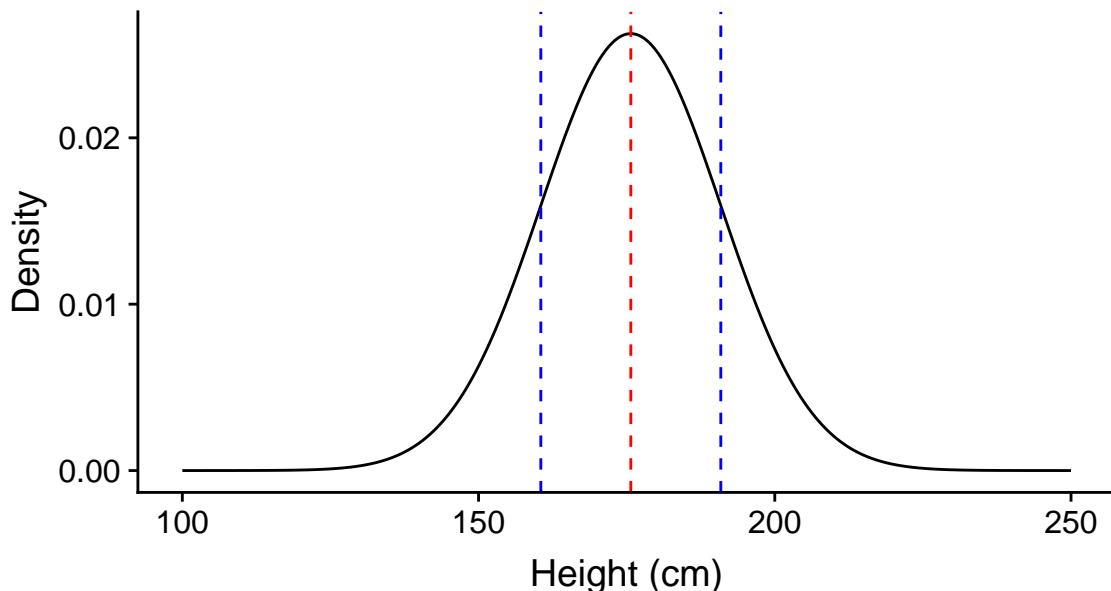
ggplot(pop.distn) +
  geom_line(aes(height, density)) +
  # vertical line at mean
  geom_vline(xintercept = true.mean, color="red", linetype="dashed") +
  # vertical line at mean + 1SD
  geom_vline(xintercept = true.mean + true.sd,
             color = "blue", linetype="dashed") +
  # vertical line at mean - 1SD
  geom_vline(xintercept = true.mean - true.sd,
             color = "blue", linetype="dashed") +
  labs(x = "Height (cm)", y = "Density",
       title = "Distribution of Heights in the Population of Interest",
       subtitle = "Red and blue lines indicate the mean \nand ±1 standard deviation respectively.")
```

---

<sup>1</sup>US Dept. of Health and Human Services; et al. (August 2016). “Anthropometric Reference Data for Children and Adults: United States, 2011–2014” (PDF). National Health Statistics Reports. 11. [https://www.cdc.gov/nchs/data/series/sr\\_03/sr03\\_039.pdf](https://www.cdc.gov/nchs/data/series/sr_03/sr03_039.pdf)

## Distribution of Heights in the Population of Interest

Red and blue lines indicate the mean and  $\pm 1$  standard deviation respectively.



### 15.2.2 Other R functions related to the normal distribution

As shown above `dnorm()` function calculates the probability density at given values of a variable `x`, given the specified mean and standard deviation.

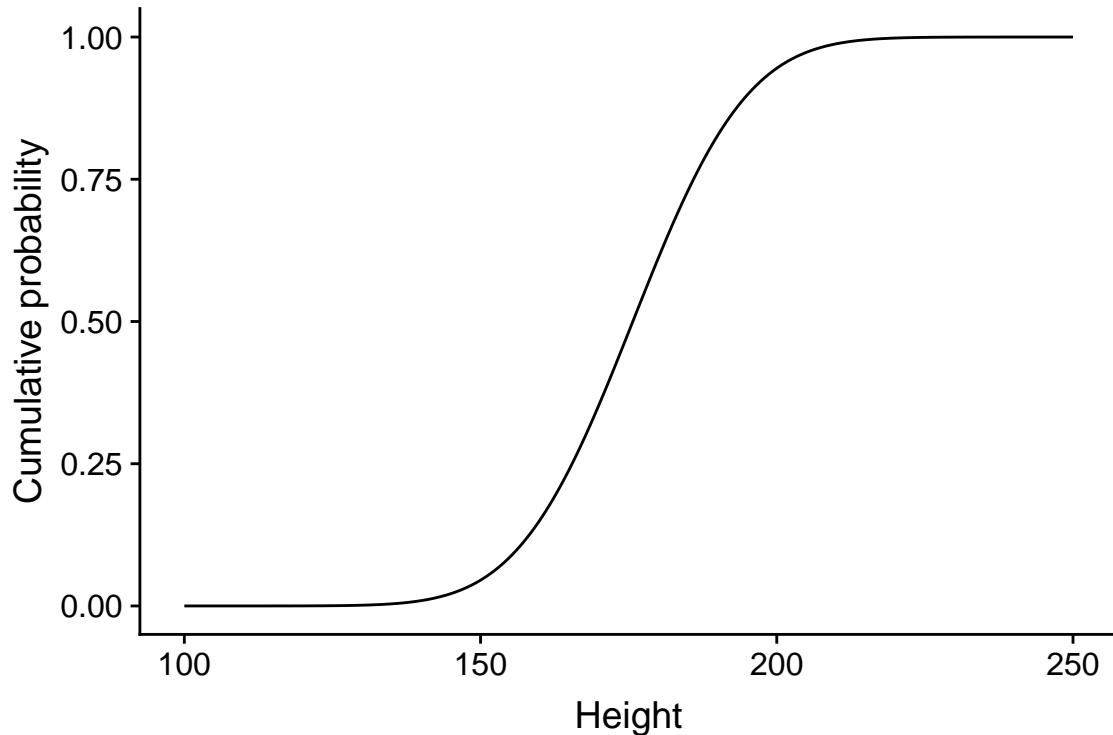
`pnorm()` gives the cumulative density function (also known as the distribution function) for the normal distribution, as shown below:

```

cdf <-
  data_frame(height = seq(100, 250, 0.5),
             cum.prob = pnorm(height, true.mean, true.sd))

ggplot(cdf) +
  geom_line(aes(height, cum.prob)) +
  labs(x = "Height", y = "Cumulative probability")

```



`qnorm()` is the quantile function for the normal distribution. The input is the probabilities of interest (single value or vector), and the mean and standard deviation of the distribution. The output is the corresponding value of the variable corresponding to the given percentiles.

For example, to estimate the lower 30th percentile of heights in adult males in the US we can use `qnorm()` as follows:

```
qnorm(0.3, true.mean, true.sd)
#> [1] 167.7344

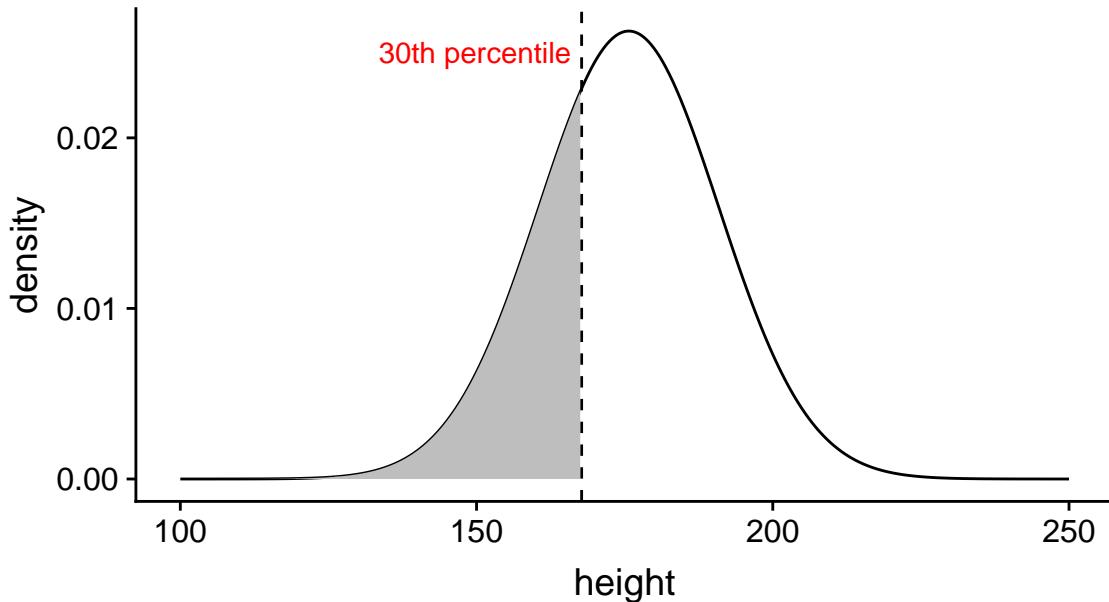
perc.30 <- qnorm(0.3, true.mean, true.sd)

label.offset <- 18 # determined by trial and error to make a
# nice looking figure

heights.less.perc.30 <- seq(100, perc.30, by=0.5)
density.less.perc.30 <- dnorm(heights.less.perc.30, true.mean, true.sd)

ggplot(pop.distn) +
  geom_line(aes(x = height, y = density)) +
  geom_vline(xintercept = perc.30, linetype='dashed') +
  geom_area(aes(x = heights.less.perc.30, y = density.less.perc.30),
            fill = "gray", data = data_frame(x = heights.less.perc.30)) +
  annotate("text", x = perc.30 - label.offset, y = 0.025,
          label = "30th percentile", color = 'red') +
  labs(title = "Probability distribution as calculated by dnorm()\nand the 30th percentile as calculated by qnorm()")
```

**Probability distribution as calculated by dnorm()  
and the 30th percentile as calculated by qnorm()  
for a normal distribution with mean and sd as given in the te**



### 15.3 Seeding the pseudo-random number generator

When carrying out simulations, we employ random number generators (e.g. to choose random samples). Most computers can not generate true random numbers – instead they use algorithms that approximate the generation of random numbers (pseudo-random number generators). One important difference between a true random number generator and a pseudo-random number generator is that we can regenerate a series of pseudo-random numbers if we know the “seed” value that initialized the algorithm. We can specifically set this seed value, so that we can guarantee that two different people evaluating this notebook get the same results, even though we’re using (pseudo)random numbers in our simulation.

```
# make our simulation repeatable by seeding RNG
set.seed(20180321)
```

### 15.4 Random sampling from the simulated population

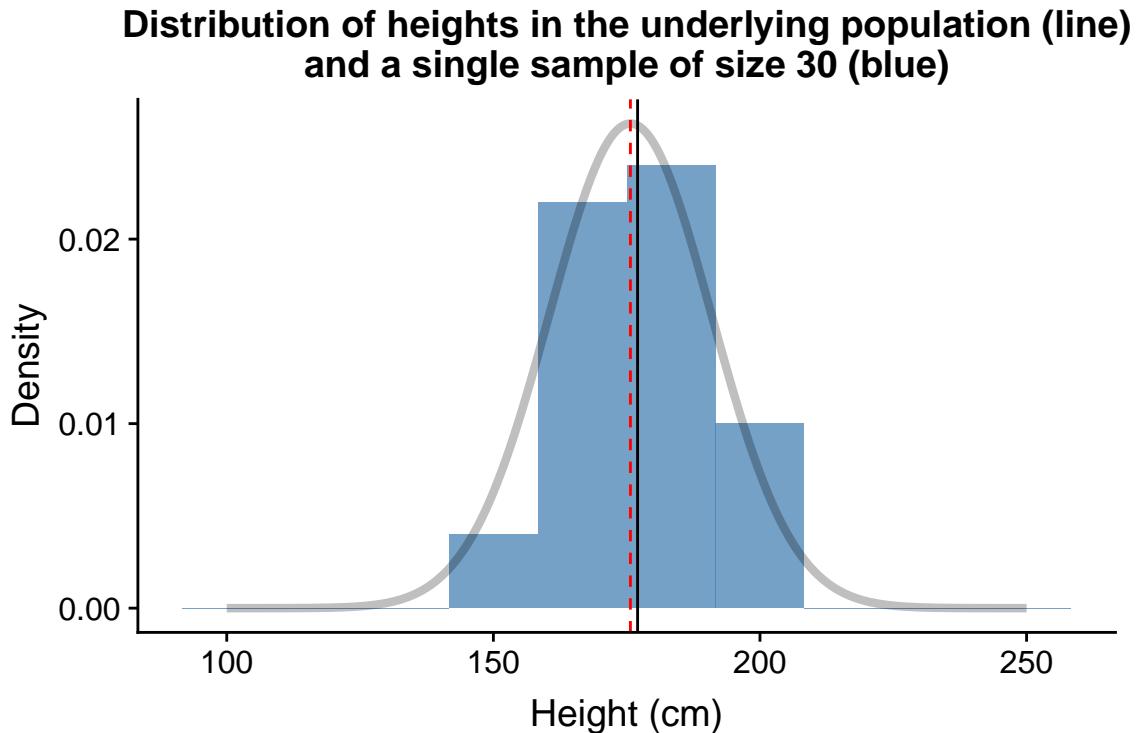
Let’s simulate the process of taking a single sample of 30 individuals from our population, using the `rnorm()` function which takes samples of size `n` from a normal distribution with the given mean and standard deviation:

```
sample.a <-
  data_frame(height = rnorm(n = 30, mean = true.mean, sd = true.sd))
```

Now we’ll create a histogram of the height variable in our sample. For reference we’ll also plot the probability for the true population (but remember, in the typical case you don’t know what the true population looks like)

```
sample.a %>%
  ggplot(aes(x = height)) +
```

```
geom_histogram(aes(y = ..density..),
               fill = 'steelblue', alpha=0.75, bins=10) +
  geom_line(data=pop.distn, aes(x = height, y = density),
            alpha=0.25, size=1.5) +
  geom_vline(xintercept = true.mean, linetype = "dashed", color="red") +
  geom_vline(xintercept = mean(sample.a$height), linetype = "solid") +
  labs(x = "Height (cm)", y = "Density",
       title = "Distribution of heights in the underlying population (line)\nand a single sample of size 30 (blue)")
```



The dashed vertical line represent the true mean of the population, the solid line represents the sample mean. Comparing the two distributions we see that while our sample of 30 observations is relatively small, its location (center) and spread that are roughly similar to those of the underlying population.

Let's create a table giving the estimates of the mean and standard deviation in our sample:

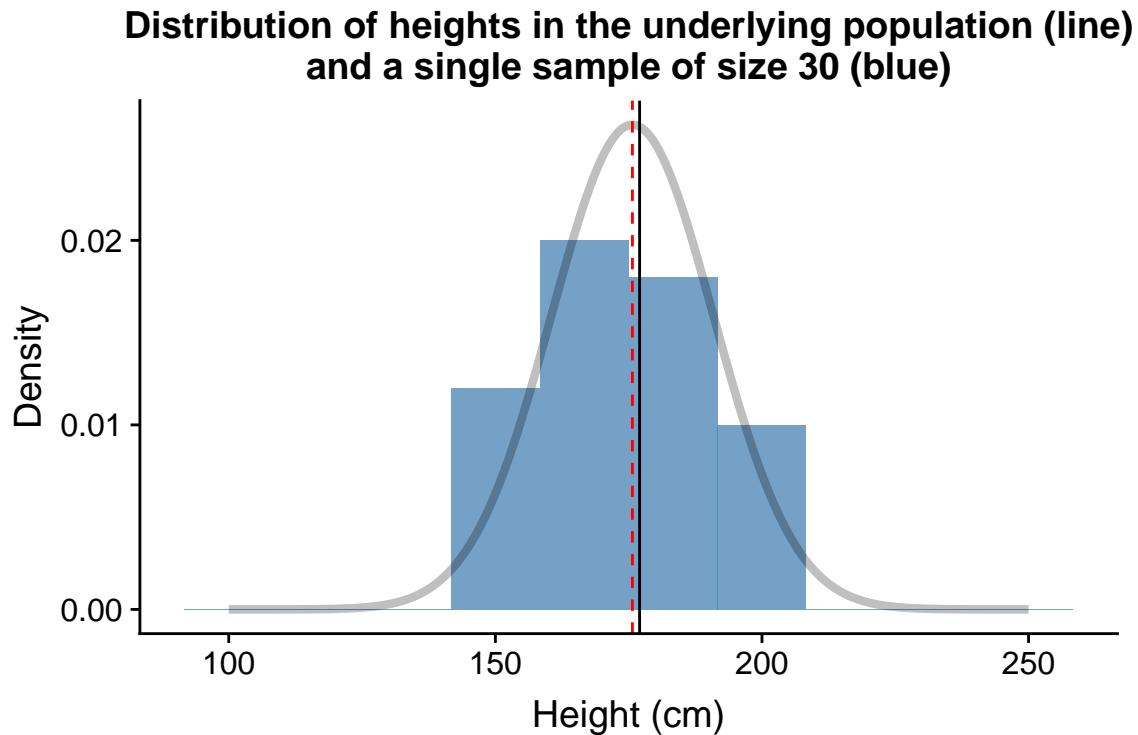
```
sample.a %>%
  summarize(sample.mean = mean(height),
           sample.sd = sd(height))
#> # A tibble: 1 x 2
#>   sample.mean sample.sd
#>     <dbl>      <dbl>
#> 1      177.1     14.08
```

Based on our sample, we estimate that the mean height of males in our population of interest is 177.0519552cm with a standard deviation of 14.0770991cm.

### 15.4.1 Another random sample

Let's step back and think about our experiment. We took a random sample of 30 individuals from the population. The very nature of a "random sample" means we could just as well have gotten a different collection of individuals in our sample. Let's take a second random sample of 25 individuals and see what the data looks like this time:

```
sample.b <-  
  data_frame(height = rnorm(30, mean = true.mean, sd = true.sd))  
  
sample.b %>%  
  ggplot(aes(x = height)) +  
  geom_histogram(aes(y = ..density..),  
                 fill = 'steelblue', alpha=0.75, bins=10) +  
  geom_line(data=pop.distn, aes(x = height, y = density),  
            alpha=0.25, size=1.5) +  
  geom_vline(xintercept = true.mean, linetype = "dashed", color="red") +  
  geom_vline(xintercept = mean(sample.b$height), linetype = "solid") +  
  labs(x = "Height (cm)", y = "Density",  
       title = "Distribution of heights in the underlying population (line)\nand a single sample of size 30 (blue)")
```



```
sample.b %>%  
  summarize(sample.mean = mean(height),  
           sample.sd = sd(height))  
#> # A tibble: 1 x 2  
#>   sample.mean sample.sd  
#>     <dbl>      <dbl>  
#> 1     174.4     16.88
```

This time we estimated the mean height to be 174.3783114 cm and the standard deviation to be 16.883491

cm.

### 15.4.2 Simulating the generation of many random samples

When we estimate population parameters, like the mean and standard deviation, based on a *sample*, our estimates will differ from the true population values by some amount. Any given random sample might provide better or worse estimates than another sample.

We can't know how good our estimates of statistics like the mean and standard deviation are from any specific sample, but we can study the behavior of such estimates *across many simulated samples* and learn something about how well our estimates do on average, as well the spread of these estimates.

### 15.4.3 A function to estimate statistics of interest in a random sample

First we're going to write a function called `rnorm.stats` that carries out the following steps:

- Take a random sample of size `n` from a distribution with a given mean (`mu`) and standard deviation (`sigma`)
- Calculate the mean and standard deviation of the random sample
- Return a table giving the sample size, sample mean, and sample standard deviation, represented as a data frame

```
rnorm.stats <- function(n, mu, sigma) {
  the.sample <- rnorm(n, mu, sigma)
  data_frame(sample.size = n,
             sample.mean = mean(the.sample),
             sample.sd = sd(the.sample))
}
```

Let's test `rsample.stats()` for a sample of size 30, drawn from a poplution with a mean and standard deviation corresponding to our height exmaple:

```
rnorm.stats(30, true.mean, true.sd)
#> # A tibble: 1 x 3
#>   sample.size sample.mean sample.sd
#>       <dbl>      <dbl>     <dbl>
#> 1        30      176.4     15.08
```

### 15.4.4 Generating statistics for many random samples

Now we'll see how to combine `rnorm.stats` with two additional functions to repeatedly run the `rsample.stats` function:

```
df.samples.of.30 <-
  rerun(2500, rnorm.stats(30, true.mean, true.sd)) %>%
  bind_rows()
```

The function `rerun` is defined in the `purrr` library (automatically loaded with tidyverse). `purrr:rerun()` re-runs an expression(s) multiple times. The first argument to `rerun()` is the number of times you want to re-run, and the following arguments are the expressions to be re-run. Thus the second line of the code block above re-runs the `rnorm.stats` function 2500 times, generating sample statistics for samples of size 30 each time it's run. `rerun` returns a list whose length is the specified number of runs.

The third line includes a call the `dplyr::bind_rows()`. This simply takes the list that `rerun` returns and collapses the list into a single data frame. `df.samples.of.30` is thus a data frame in which each row gives

the sample size, sample mean, and sample standard deviation for a random sample of 30 individuals drawn from our underlying population with a normally distributed variable.

```
head(df.samples.of.30)
#> # A tibble: 6 x 3
#>   sample.size sample.mean sample.sd
#>       <dbl>      <dbl>     <dbl>
#> 1       30      170.8    15.36
#> 2       30      174.6    14.93
#> 3       30      175.7    16.79
#> 4       30      176.5    13.94
#> 5       30      175.3    13.10
#> 6       30      176.6    15.49
```

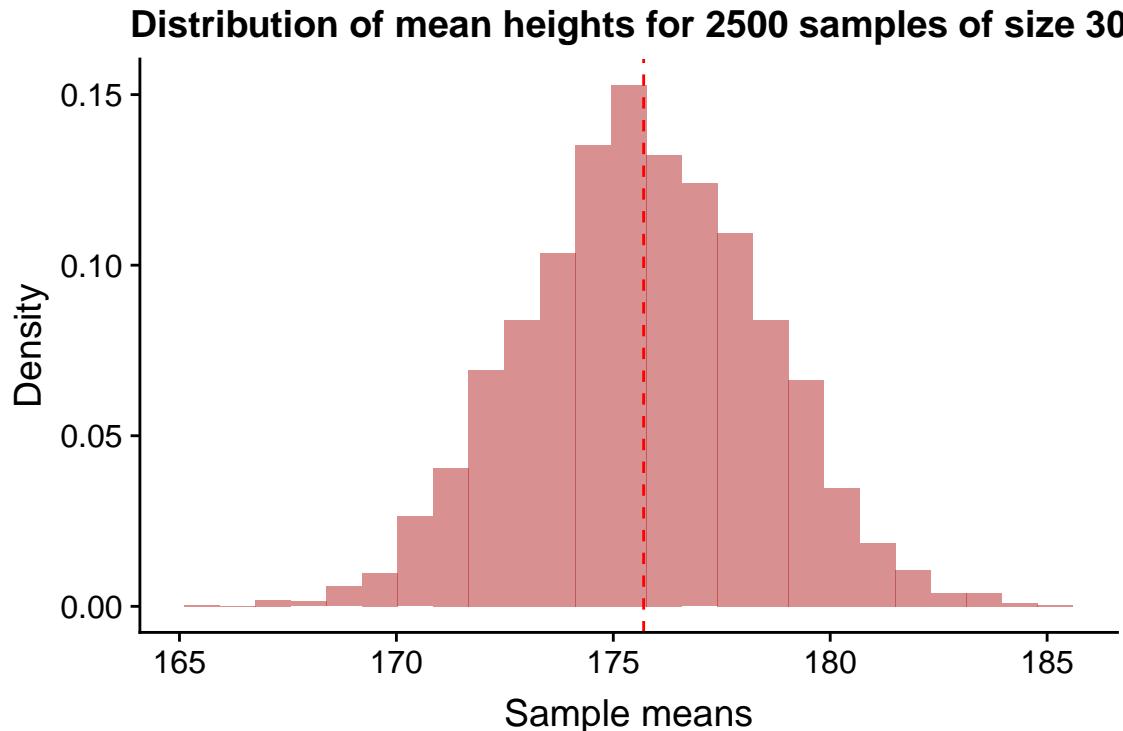
## 15.5 Simulated sampling distribution of the mean

Let's review what we just did:

- We generated 2500 samples of size 30, sampling a variable with an underlying normal distribution
- For each of the samples we calculated the mean and standard deviation *in that sample*
- We combined each of those estimates of the mean and standard deviation into a data frame

The 2500 estimates of the mean we generated represents a new distribution – what we will call a **sampling distribution of the mean for samples of size 30**. Let's plot this sampling distribution:

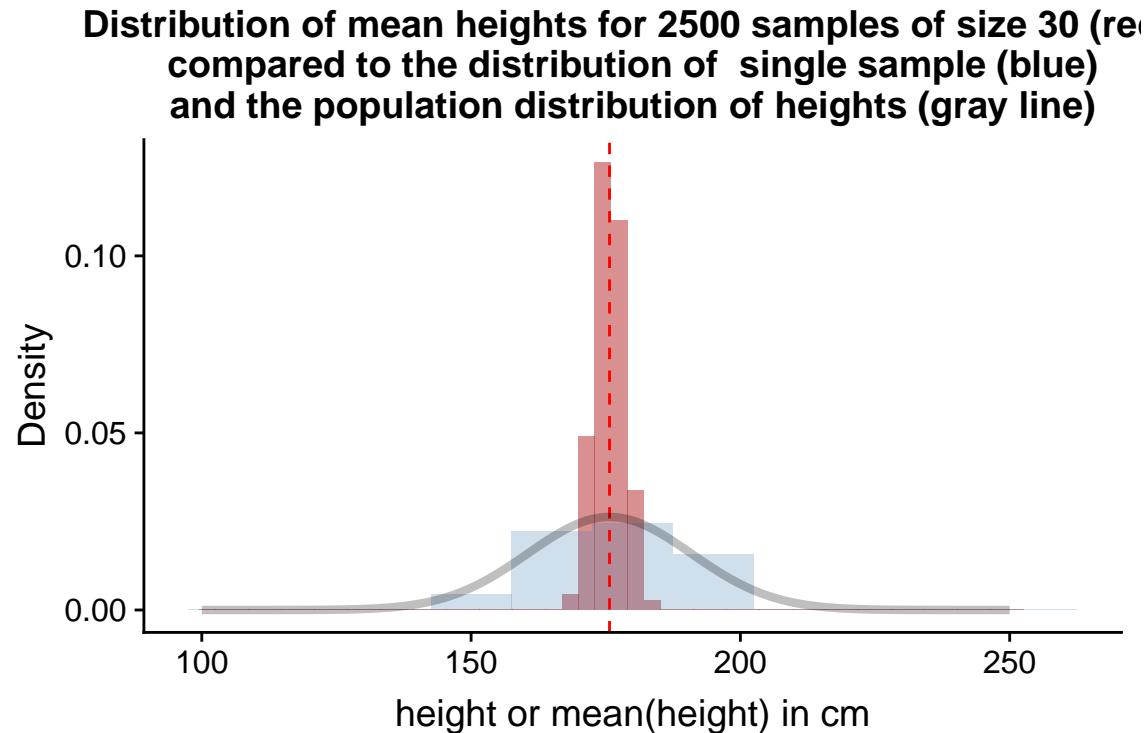
```
ggplot(df.samples.of.30, aes(x = sample.mean, y = ..density..)) +
  geom_histogram(bins=25, fill = 'firebrick', alpha=0.5) +
  geom_vline(xintercept = true.mean, linetype = "dashed", color="red") +
  labs(x = "Sample means", y = "Density",
       title = "Distribution of mean heights for 2500 samples of size 30")
```



### 15.5.1 Differences between sampling distribution and sample/population distributions

Note that this is *not* a sample distribution of the variable of interest (“heights”), but rather the distribution of *means of the variable of interest* (“mean heights”) you would get if you took many random samples (in one sample you’d estimate the mean height as 180cm, in another you’d estimate it as 172 cm, etc). To emphasize this point, let’s compare the simulated sampling distribution to the population distribution of the the variable:

```
ggplot(df.samples.of.30, aes(x = sample.mean, y = ..density..)) +
  geom_histogram(bins=50, fill = 'firebrick', alpha=0.5) +
  geom_histogram(data=sample.a,
                 aes(x = height, y = ..density..),
                 bins=11, fill='steelblue', alpha=0.25) +
  geom_vline(xintercept = true.mean, linetype = "dashed", color='red') +
  geom_line(data=pop.distn, aes(x = height, y = density), alpha=0.25, size=1.5) +
  labs(x = "height or mean(height) in cm", y = "Density",
       title = "Distribution of mean heights for 2500 samples of size 30 (red )\ncompared to the distri")
```



### 15.5.2 Use sampling distributions to understand the behavior of statistics of interest

The particular sampling distribution of the mean, as simulated above, is a probability distribution that we can use to estimate the probability that a sample mean falls within a given interval, assuming our sample is a random sample of size 30 drawn from our underlying population.

From our visualization, we see that the distribution of sample mean heights is approximately centered around the true mean height. Most of the sample estimates of the mean height are within 5 cm of the true population mean height, but a small number of estimates of the sample mean are off by nearly 10cm.

Let's make this more precise by calculating the mean and standard deviation of the sampling distribution of means:

```
df.samples.of.30 %>%
  summarize(mean.of.means = mean(sample.mean),
           sd.of.means = sd(sample.mean))
#> # A tibble: 1 x 2
#>   mean.of.means    sd.of.means
#>       <dbl>        <dbl>
#> 1      175.7       2.740
```

### 15.5.3 Sampling distributions for different sample sizes

In the example above we simulated the sampling distribution of the mean for samples of size 30. How would the sampling distribution change if we increased the sample size? In the next code block we generate sampling distributions of the mean (and standard deviation) for samples of size 50, 100, 250, and 500.

```
df.samples.of.50 <-
  rerun(2500, rnorm.stats(50, true.mean, true.sd)) %>%
  bind_rows()

df.samples.of.100 <-
  rerun(2500, rnorm.stats(100, true.mean, true.sd)) %>%
  bind_rows()

df.samples.of.250 <-
  rerun(2500, rnorm.stats(250, true.mean, true.sd)) %>%
  bind_rows()

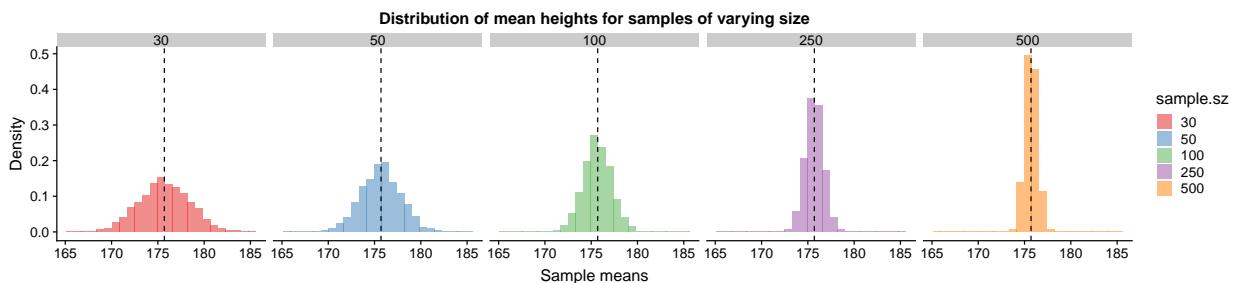
df.samples.of.500 <-
  rerun(2500, rnorm.stats(500, true.mean, true.sd)) %>%
  bind_rows()
```

To make plotting and comparison easier we will combine each of the individual data frames, representing the different sampling distributions for samples of a given size, into a single data frame.

```
df.combined <-
  bind_rows(df.samples.of.30,
            df.samples.of.50,
            df.samples.of.100,
            df.samples.of.250,
            df.samples.of.500) %>%
  # create a factor version of sample size to facilitate plotting
  mutate(sample.sz = as.factor(sample.size))
```

We then plot each of the individual sampling distributions, faceting on sample size.

```
ggplot(df.combined, aes(x = sample.mean, y = ..density.., fill = sample.sz)) +
  geom_histogram(bins=25, alpha=0.5) +
  geom_vline(xintercept = true.mean, linetype = "dashed") +
  facet_wrap(~ sample.sz, nrow = 1) +
  scale_fill_brewer(palette="Set1") # change color palette
  labs(x = "Sample means", y = "Density",
       title = "Distribution of mean heights for samples of varying size")
```



#### 15.5.4 Discussion of trends for sampling distributions of different sample sizes

The key trend we see when comparing the sampling distributions of the mean for samples of different size is that as the sample size gets larger, the spread of the sampling distribution of the mean becomes narrower around the true mean. This means that *as sample size increases, the uncertainty associated with our estimates of the mean decreases*.

Let's create a table, grouped by sample size, to help quantify this pattern:

```
sampling.distn.mean.table <-
  df.combined %>%
  group_by(sample.size) %>%
  summarize(mean.of.means = mean(sample.mean),
            sd.of.means = sd(sample.mean))

sampling.distn.mean.table
#> # A tibble: 5 x 3
#>   sample.size  mean.of.means  sd.of.means
#>       <dbl>        <dbl>        <dbl>
#> 1      30     175.7     2.740
#> 2      50     175.7     2.173
#> 3     100     175.7     1.509
#> 4     250     175.7     0.9756
#> 5     500     175.7     0.6656
```

## 15.6 Standard Error of the Mean

We see from the graph and table above that our estimates of the mean cluster more tightly about the true mean as our sample size increases. This is obvious when we compare the standard deviation of our mean estimates as a function of sample size.

The standard deviation of the sampling distribution of a statistic of interest is called the **Standard Error** of that statistic. Here, through simulation, we are approximating the **Standard Error of the Mean**.

One can show mathematically that the expected Standard Error of the Mean as a function of sample size and the standard deviation of the underlying population distribution is:

$$\text{Standard Error of Mean} = \frac{\sigma}{\sqrt{n}}$$

where  $\sigma$  is the population standard deviation (i.e. the “true” standard deviation), and  $n$  is the sample size. This result for the standard error of the mean is true regardless of the form of the underlying population distribution.

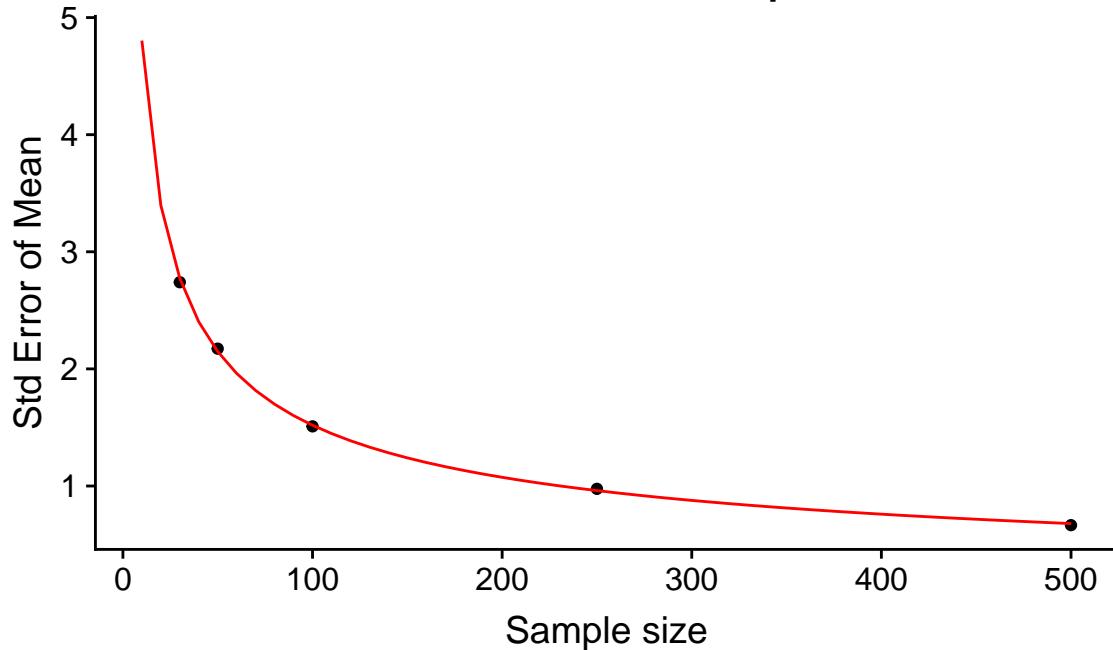
Let's compare that theoretical expectation to our simulated results:

```
se.mean.theory <- sapply(seq(10,500,10),
                         function(n){ true.sd/sqrt(n) })

df.se.mean.theory <- data_frame(sample.size = seq(10,500,10),
                                 std.error = se.mean.theory)

ggplot(sampling.distn.mean.table, aes(x = sample.size, y = sd.of.means)) +
  # plot standard errors of mean based on our simulations
  geom_point() +
  # plot standard errors of the mean based on theory
  geom_line(aes(x = sample.size, y = std.error), data = df.se.mean.theory, color="red") +
  labs(x = "Sample size", y = "Std Error of Mean",
       title = "A comparison of theoretical (red line) and simulated (points) estimates of\nthe standard error of the mean")
```

**comparison of theoretical (red line) and simulated (points) estim  
the standard error of the mean for samples of different size**



We see that as sample sizes increase, the standard error of the mean decreases. This means that as our samples get larger, our uncertainty in our sample estimate of the mean (our best guess for the population mean) gets smaller.

## 15.7 Sampling Distribution of the Standard Deviation

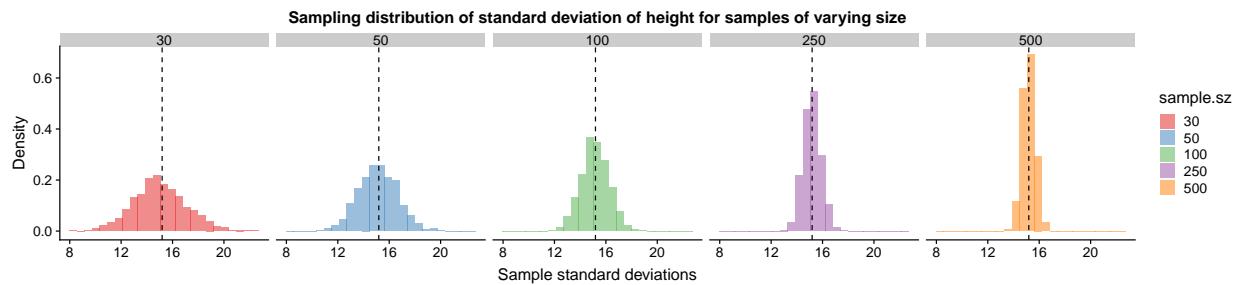
Above we explored how the sampling distribution of the mean changes with sample size. We can similarly explore the sampling distribution of any other statistic, such as the standard deviation, or the median, or the range, etc.

Recall that when we drew random samples we calculated the standard deviation of each of those samples in addition to the mean. We can look at the location and spread of the estimates of the standard deviation:

```
sampling.distn.sd.table <-  
  df.combined %>%  
  group_by(sample.size) %>%  
  summarize(mean.of.sds = mean(sample.sd),  
           sd.of.sds = sd(sample.sd))  
  
sampling.distn.sd.table  
#> # A tibble: 5 x 3  
#>   sample.size  mean.of.sds  sd.of.sds  
#>       <dbl>      <dbl>     <dbl>  
#> 1        30     15.02     2.046  
#> 2        50     15.12     1.500  
#> 3       100     15.16     1.089  
#> 4       250     15.17     0.6873  
#> 5       500     15.19     0.4957
```

As we did for the sampling distribution of the mean, we can visualize the sampling distribution of the standard deviation as shown below:

```
ggplot(df.combined, aes(x = sample.sd, y = ..density.., fill = sample.sz)) +
  geom_histogram(bins=25, alpha=0.5) +
  geom_vline(xintercept = true.sd, linetype = "dashed") +
  facet_wrap(~ sample.sz, nrow = 1) +
  scale_fill_brewer(palette="Set1") +
  labs(x = "Sample standard deviations", y = "Density",
       title = "Sampling distribution of standard deviation of height for samples of varying size")
```



The key trend we saw when examining the sampling distribution of the mean is also apparent for standard deviation – bigger samples lead to tighter sampling distributions and hence less uncertainty in the sample estimates of the standard deviation.

### 15.7.1 Standard error of standard deviations

For normally distributed data the expected Standard Error of the Standard Deviation (i.e. the standard deviation of standard deviations!) is approximately:

$$\text{Standard Error of Standard Deviation} \approx \frac{\sigma}{\sqrt{2(n - 1)}}$$

where  $\sigma$  is the population standard deviation, and  $n$  is the sample size.

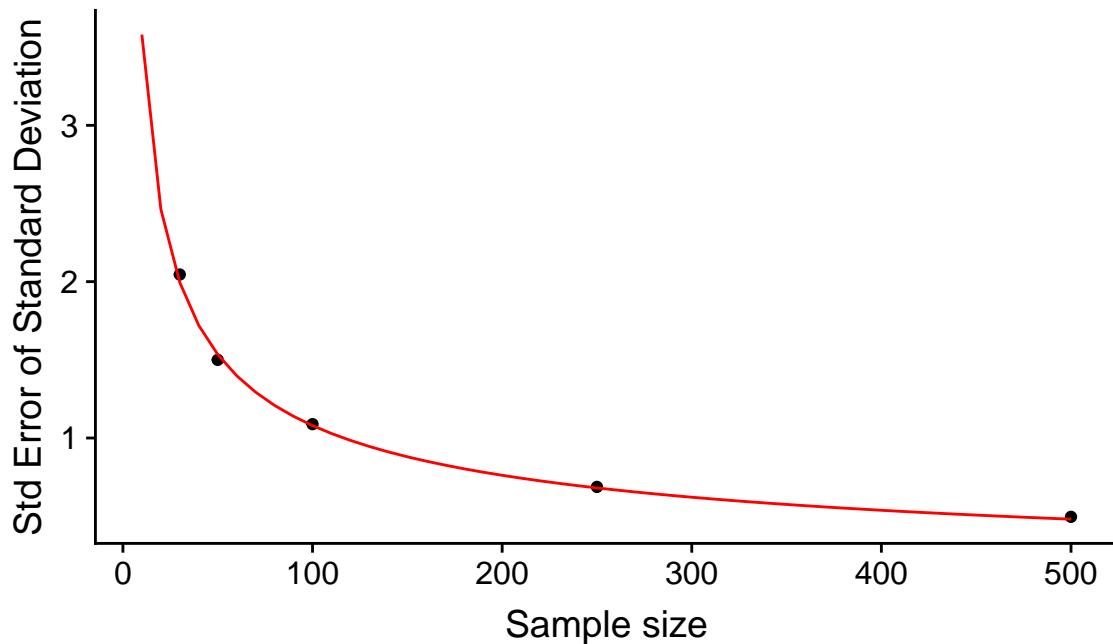
As before, let's visually compare the theoretical expectation to our simulated estimates.

```
se.sd.theory <- sapply(seq(10, 500, 10),
                      function(n){ true.sd/sqrt(2*(n-1))})

df.se.sd.theory <- data_frame(sample.size = seq(10,500,10),
                               std.error = se.sd.theory)

ggplot(sampling.distn.sd.table, aes(x = sample.size, y = sd.of.sds)) +
  # plot standard errors of mean based on our simulations
  geom_point() +
  # plot standard errors of the mean based on theory
  geom_line(aes(x = sample.size, y = std.error), data = df.se.sd.theory, color="red") +
  labs(x = "Sample size", y = "Std Error of Standard Deviation",
       title = "A comparison of theoretical (red line) and simulated (points) estimates of\nthe standard error of standard deviations")
```

**comparison of theoretical (red line) and simulated (points) estimates of the standard error of the standard deviation for samples of different sizes**



## 15.8 What happens to the sampling distribution of the mean and standard deviation when our sample size is small?

We would hope that, regardless of sample size, the sampling distributions of both the mean and standard deviation should be centered around the true population value,  $\mu$  and  $\sigma$  respectively. That seemed to be the case for the modest to large sample sizes we've looked at so far (30 to 500 observations). Does this also hold for small samples? Let's use simulation to explore how well this expectation is met for small samples.

As we've done before, we simulate the sampling distribution of the mean and standard deviation for samples of varying size.

```
# sample sizes we'll consider
ssizes <- c(2, 3, 4, 5, 7, 10, 20, 30, 50)

# number of samples to draw *for each sample size*
nsamples <- 2500

# create a data frame with empty columns
df.combined.small <- data.frame(sample.size = double(),
                                   sample.mean = double(),
                                   sample.sd = double(),
                                   estimated.SE = double(),
                                   sample.zscore = double())

for (i in ssizes) {
  df.samples.of.size.i <-
    rerun(nsamples, rnorm.stats(i, true.mean, true.sd)) %>%
    bind_rows()
```

```

df.combined.small <-
  bind_rows(df.combined.small, df.samples.of.size.i)

}

df.combined.small %<>%
  mutate(sample.sz = as.factor(sample.size))

```

### 15.8.1 For small samples, sample standard deviations systematically underestimate the population standard deviation

Let's examine how the well centered the sampling distributions of the mean and standard deviation are around their true values, as a function of sample size.

First a table summarizing this information:

```

by.sample.size <-
  df.combined.small %>%
  group_by(sample.size) %>%
  summarize(mean.of.means = mean(sample.mean),
            mean.of.sds = mean(sample.sd))

by.sample.size
#> # A tibble: 9 x 3
#>   sample.size  mean.of.means  mean.of.sds
#>       <dbl>        <dbl>        <dbl>
#> 1         2     175.2      11.93
#> 2         3     175.8      13.51
#> 3         4     175.7      13.99
#> 4         5     175.7      14.15
#> 5         7     175.8      14.52
#> 6        10     175.7      14.78
#> 7        20     175.7      14.98
#> 8        30     175.6      15.07
#> 9        50     175.7      15.12

```

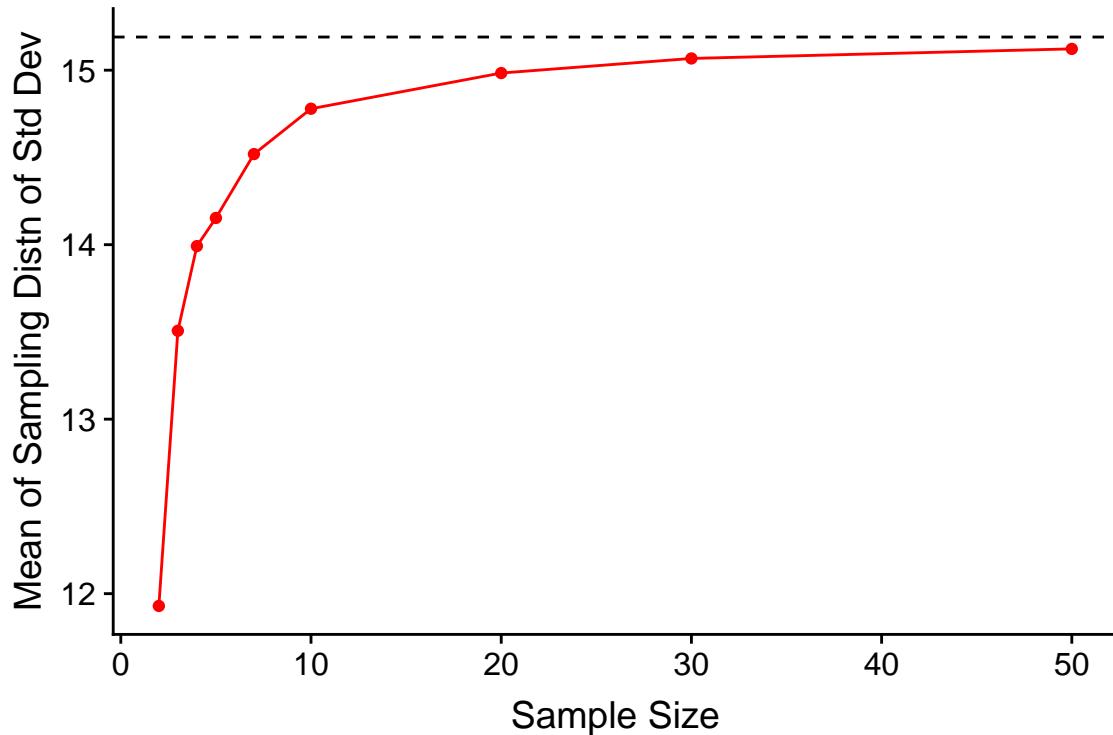
We see that the sampling distributions of means are well centered around the true mean for both small and medium, and there is no systematic bias one way or the other. By contrast the sampling distribution of standard deviations tends to underestimate the true standard deviation when the samples are small (less than 30 observations).

We can visualize this bias as shown here:

```

ggplot(by.sample.size, aes(x = sample.size, y = mean.of.sds)) +
  geom_point(color = 'red') +
  geom_line(color = 'red') +
  geom_hline(yintercept = true.sd, color = 'black', linetype='dashed') +
  labs(x = "Sample Size", y = "Mean of Sampling Distrn of Std Dev")

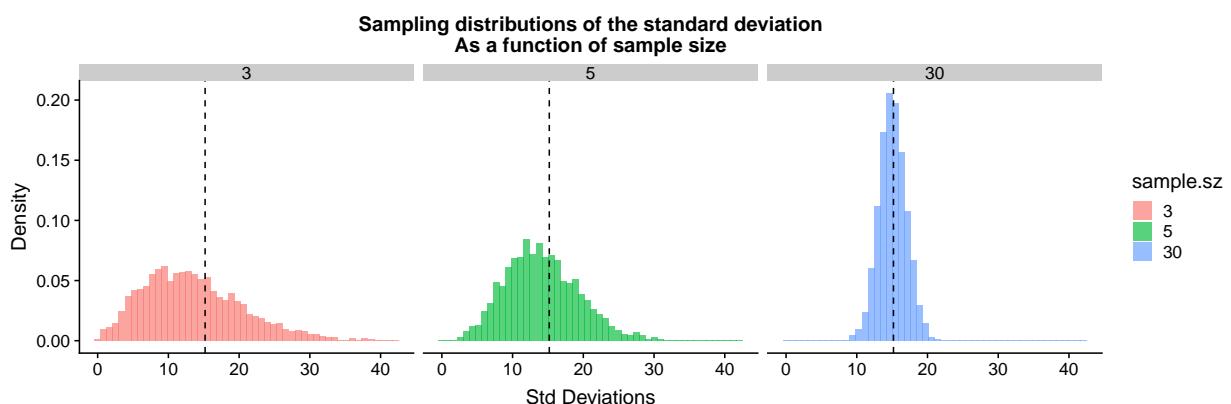
```



The source of this bias is clear if we look at the sampling distribution of the standard deviation for samples of size 3, 5, and 30.

```
filtered.df <-
  df.combined.small %>%
  filter(sample.size %in% c(3, 5, 30))

ggplot(filtered.df, aes(x = sample.sd, y = ..density.., fill = sample.sz)) +
  geom_histogram(bins=50, alpha=0.65) +
  facet_wrap(~sample.size, nrow = 1) +
  geom_vline(xintercept = true.sd, linetype = 'dashed') +
  labs(x = "Std Deviations", y = "Density",
       title = "Sampling distributions of the standard deviation\nAs a function of sample size")
```



There's very clear indication that the the sampling distribution of standard deviations is not centered around the true value for  $n = 3$  and for  $n = 5$ , however with samples of size 30 the sampling distribution of the

## 15.8. WHAT HAPPENS TO THE SAMPLING DISTRIBUTION OF THE MEAN AND STANDARD DEVIATION WHEN

standard deviation appears fairly well centered around the true value of the underlying population.

### 15.8.2 Underestimates of the standard deviation given small $n$ lead to underestimates of the SE of the mean

When sample sizes are small, sample estimates of the standard deviation,  $s_x$ , tend to underestimate the true standard deviation,  $\sigma$ , then it follows that sample estimates of the standard error of the mean,  $SE_{\bar{x}} = \frac{s_x}{\sqrt{n}}$ , must tend to underestimate the true standard error of the mean,  $SE_{\mu} = \frac{\sigma}{\sqrt{n}}$ .



# Chapter 16

## Introduction to hypothesis testing

In hypothesis testing we compare statistical properties of our observed data to the same properties we would expect to see under a *null hypothesis*.

More specifically we compare our point estimate of a statistic of interest, based on our data, to the *sampling distribution of that statistic* under a given null hypothesis.

### 16.1 Libraries

```
library(magrittr)
library(tidyverse)
library(broom)
```

### 16.2 Null and Alternative Hypotheses

When carrying out statistical hypothesis testing we must formulate a “null hypothesis” and an “alternative hypothesis” (these always come as a pair) that jointly describe the set of possible values for a statistic of interest. We must also make some assumptions, either based on theory or inferred from the data, about the distributional properties of the sampling distribution of the statistic of interest.

NOTE: statistical hypotheses are not scientific hypotheses, but rather statistical statements about a population. Statistical hypotheses can help us to determine which predictions stemming from scientific hypotheses are consistent with the data, but statistical hypotheses are not “statements about the existence and possible causes of natural phenomena” (Whitlock & Schluter 2014).

#### 16.2.1 Null hypotheses

Whitlock & Schuluter: “A **null hypothesis** is a specific statement about a population parameter made for the purpose of argument. A good null hypothesis is a statement that would be interesting to reject.”

- Null hypotheses typically correspond to outcomes that would suggest “no difference” or “no effect” of the treatment, grouping, or other types of comparisons one makes with data
- Sometimes a null expectation is based on prior observation or from theoretical considerations
- A null hypothesis is always *specific* – specifies on particular value of the parameter being studied (though sometimes this is implicit when written in words)

- The standard mathematical notation to indicate a null hypothesis is to write  $H_0$  (“H-zero” or “H-naught”)

Examples of null hypotheses:

- $H_0$ : The density of dolphins is the same in areas with and without drift-net fishing
- $H_0$  :The effect of ACE inhibitors on blood pressure does not differ from administering a placebo
- $H_0$ : There is no correlation between maternal smoking and the probability of premature births

### 16.2.2 Alternative hypotheses

Whitlock & Schluter: “The **alternative hypothesis** includes all other feasible values for the population parameter besides the value stated in the null hypothesis”

- Alternative hypotheses usually include parameter values that are predicted by a scientific hypothesis, but often include other feasible values as well
- The standard mathematical notation to indicate a null hypothesis is to write  $H_A$

Examples of alternative hypotheses:

- $H_A$ : The density of dolphins differs in areas with and without drift-net fishing
- $H_A$  :The effect of ACE inhibitors on blood pressure differs from administration of a placebo
- $H_A$ : There is a non-zero correlation between maternal smoking and the probability of premature births

## 16.3 Rejecting / failing to reject null hypotheses

When carrying out statistical hypothesis testing **the null hypothesis is the only statement being tested with the data.**

- If the data are consistent with the null hypothesis, we have “failed to reject the null hypothesis”. This is *not* the same as accepting the null hypothesis.
- If the data are inconsistent with the null hypothesis, we “reject the null hypothesis” and say the data support the alternative hypothesis
- Note that because the alternative hypothesis is usually formulated in terms of all other possible values of a parameter of interest, rejecting the null hypothesis does not allow us to make a probabilistic statement about the value of that parameter.

## 16.4 Outcomes of hypothesis tests

In reality, a null hypothesis is either true or false. When you carry out a hypothesis test, there are two possible test outcomes – you reject the null hypothesis or you fail to reject the null hypothesis. It is typical to represent the different combinations of the reality / statistical tests in a table like the following:

|            |  | do not reject $H_0$ | reject $H_0$                            |
|------------|--|---------------------|---|
| $H_0$ true | okay                                   |                     | Type 1 error (false positive), $\alpha$ |
| $H_A$ true | Type 2 error (false negative), $\beta$ |                     | okay                                    |

When we specify a significance threshold,  $\alpha$ , for hypothesis testing, this controls the false positive rate (also called Type I error) of our test. The false negative rate (also called Type II error) is often referred to as  $\beta$ . In general, there is a tradeoff between the false positive and false negative rate – the lower the false positive rate the higher the false negative rate, and vice versa.

## 16.5 Using p-values to assess the strength of evidence against the null hypothesis

To assess the strength of the evidence against the null hypothesis, we can ask “what is the probability of observing a statistic of interest that is **at least as favorable** to the alternative hypothesis, **if the null hypothesis were true?**”

Mathematically, we pose this question with respect to the expected *sampling distribution of the statistic of interest* under the null hypothesis (this is called the *null distribution*). For example, when testing a hypothesis involving means, we would ask “What is the probability of observing my sample mean, with respect to the expected distribution of sample means, if the null hypothesis was true.”

**P-value** The p-value is the probability of observing data at least as favorable to the alternative hypothesis as our current data set, if the null hypothesis is true.

**Small p-values** give us evidence to support the *rejection* of the null hypothesis. Conventionally p-values of less than 0.05 or 0.01 are used in many scientific fields, though recently there have been calls in some fields to redefine this convention (see for example Benjamin et al. 2017, <https://www.nature.com/articles/s41562-017-0189-z.pdf>).

## 16.6 Example: Comparing a sample mean to an hypothesized normal distribution

Occasionally one wishes to ask whether a sample mean, from a sample of size  $n$ , is consistent with having been drawn from a normal distribution with a specified mean and standard deviation,  $N(\mu, \sigma)$ .

### 16.6.1 Null and alternative hypotheses

- $H_0$ : the sample mean,  $\bar{x}$  is from a normal distribution with a mean  $\mu$  and standard deviation  $\sigma$
- $H_A$ : the sample mean is not from such a distribution

### 16.6.2 Sampling distribution of the mean

For a normal distribution with parameters,  $N(\mu, \sigma)$ , the sampling distribution of the mean is itself normally distributed, with parameters  $N(\mu, \sigma/\sqrt{n})$ .

A reminder that the standard deviation of the sampling distribution of the mean,  $\sigma/\sqrt{n}$ , is usually referred to as the “standard error of the mean”

### 16.6.3 Calculating a p-value

Since the sampling distribution of the mean is normally distributed, as described above we can use the `pnorm()` function to

Let’s apply this to the comparison we made in last class, involving a comparison of the sample mean of tail length in Victoria possums to a hypothesized normal distribution based on Queensland possum. In that example, our hypothesized normal distribution had parameters  $N(37.9 \text{ cm}, 1.71 \text{ cm})$ . Our Victoria possum sample size was  $n = 5$ , with a mean  $x = 35.9 \text{ cm}$ .

Our null hypothesis is framed in a manner that requires a two-tailed test, as under the alternative hypothesis the sample could be drawn from a distribution with either a higher or lower mean.

```

# parameters of null
null.mu <- 37.9
null.sigma <- 1.71

# sample statistics
observed.mean <- 35.9
n <- 5

diff.mean <- abs(null.mu - observed.mean)

# standard deviation of sampling distribution under null hypotheses
null.SE.mean <- null.sigma/sqrt(n)

# calculate p-value
p.value <-
  pnorm(null.mu - diff.mean, null.mu, null.SE.mean, lower.tail = TRUE) +
  pnorm(null.mu + diff.mean, null.mu, null.SE.mean, lower.tail = FALSE)

p.value
#> [1] 0.008915324

```

The p-value given above should be very close to the estimate you generated via simulation in the previous assignment.

## 16.7 Example: Handedness of toads

From Whitlock & Schluter:

Most humans are right handed. Do animals besides human exhibit biased handedness? Bisazza et al. (1996) tested the possibility of biased handedness in European toads, *Bufo bufo*. They sampled 18 toads and used a behavioral assay to determine the preferred hand each individual frog used to perform a task.

### 16.7.1 Null and alternative hypotheses

- $H_0$ : toads do not exhibit any bias towards right or left handedness, i.e.  $p(\text{right handed}) = 0.5$
- $H_A$ : right and left handedness are not equally frequent in the population, i.e.  $p(\text{right haned}) \neq 0.5$

### 16.7.2 Data

Of the 18 toads tested, 14 were right-handed and 4 were left handed. Therefore, the observed proportion of right-handed toads was 0.778

### 16.7.3 Sampling distribution for proportions: Binomial distribution

To assess how likely it would be to observe the proportion 0.778 of right-handed toads, *if the null hypothesis of equal probability of right- and left-handed toads was true*, we need to know the appropriate sampling distribution.

Here we're dealing with binary outcomes for each outcome – each frog is either right or left handed. For data with binary outcomes we can arbitrary call one outcome a “success”. If we assume that there is a

fixed probability of getting a success, and each observation is independent, than the **binomial distribution** provides the probability distribution for the number of “successes” in a given number of trials (observations).

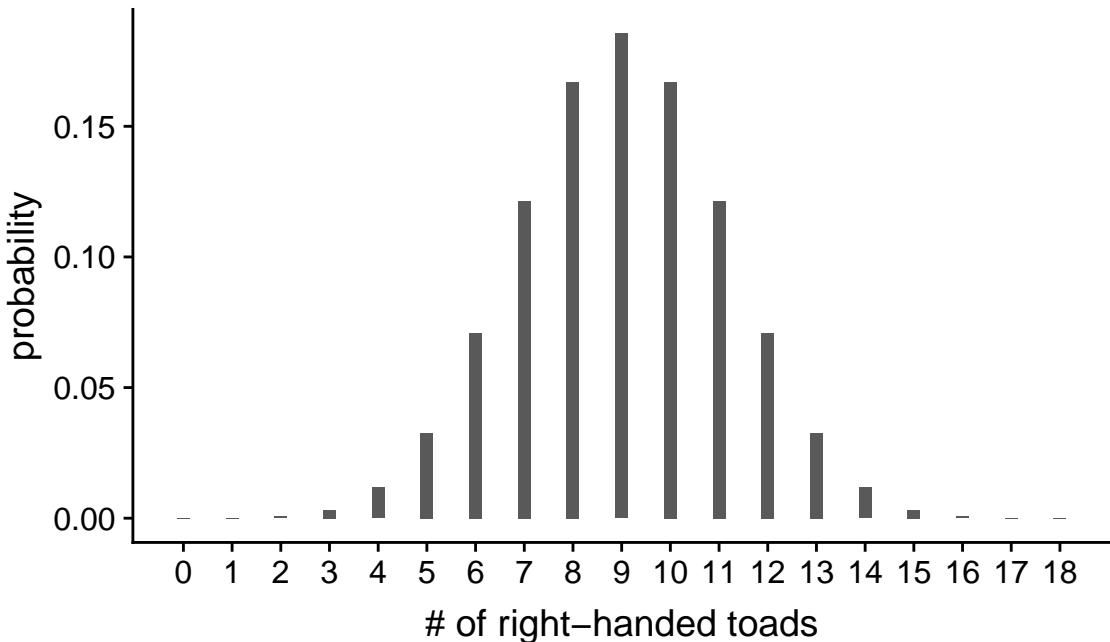
Let’s call the outcome “right-handed” the successful outcome and let’s generate the expected counts of successes under the null hypothesis (i.e.  $p(\text{right handed}) = 0.5$ ). As we saw in a previous lecture, we can use `dbinom()` function, which is the probability mass function for the binomial distribution. Given a vector of outcome of interest, the number of trials, and the probability of a success, `dbinom` calculates the probability of each outcome.

```
possible.outcomes <- seq(0, 18)
H0.prob = 0.5
prob.distn <- dbinom(x = possible.outcomes, size = 18, prob = H0.prob)

null.df <- data.frame(outcomes = possible.outcomes, probs = prob.distn)

ggplot(null.df) +
  geom_col(aes(x = outcomes, y = probs), width=0.25) +
  scale_x_continuous(breaks = possible.outcomes) +
  labs(x = "# of right-handed toads", y = "probability",
       title = "The null distribution for the observed number of right handed toads\nBinomial distribution, p = 0.5, n = 18")
```

**The null distribution for the observed number of right handed toads  
Binomial distribution,  $p = 0.5, n = 18$**



#### 16.7.4 Calculating a p-value for the binomial test

Our alternative hypothesis was stated as “right and left handedness are not equally frequent in the population”. This reflects the fact that we did not have a strong a priori prediction about the direction of a potential bias in handedness among toads.

Recall that a p-value represents the probability of observing the parameter of interest (counts in this case) *at least as extreme* as that in our data, under the null distribution. There are two ways to deviate from the null hypothesis – observing more right-handed toads than we expect OR observing fewer right-handed

toads than we expect. To calculate an appropriate p-value in this case we must consider deviations in both directions. This is what we call a **two-tailed test** because we must consider both tails of the sampling distribution.

In the present case, we need to add up the probability of observing 14 or more right-handed toads (right tail of the null distribution) and the probability of observing 4 or fewer right-handed toads (left tail of the null distribution).

```
left.tail <- filter(null.df, outcomes <= 4) %$% probs %>% sum
right.tail <- filter(null.df, outcomes >= 14) %$% probs %>% sum

p.value <- left.tail + right.tail
p.value
#> [1] 0.03088379
```

Our calculated p-value is approximately 0.031. If we the conventional significance threshold of  $\alpha = 0.05$ , our conclusion would be “we reject the null hypothesis of equal probability of right- and left-handed toads”.

### 16.7.5 The `binom.test()` function

The previous calculations can be conveniently carried out using the R function `binom.test()`. The arguments to `binom.test()` are the number of successes (`x`), the number of trials (`n`), and the hypothesized probability of success (`p`).

```
b.test <- binom.test(4, 18, p=0.5, alternative="two.sided")
b.test
#>
#> Exact binomial test
#>
#> data: 4 and 18
#> number of successes = 4, number of trials = 18, p-value = 0.03088
#> alternative hypothesis: true probability of success is not equal to 0.5
#> 95 percent confidence interval:
#> 0.06409205 0.47637277
#> sample estimates:
#> probability of success
#> 0.2222222
```

As shown above, display the results of `binom.test()` provides a simple written summary. You can retrieve specific values using named fields (see the documentation fo the full list):

```
b.test$p.value # gives the p-value under null
#> [1] 0.03088379
b.test$estimate # gives the estimated probability from the data
#> probability of success
#> 0.2222222
```

## 16.8 Example: Measuring the association between maternal smoking and premature births

We've previously explored the NC Births data set which includes information on mothers age, smoking status, weight gained, birth weight, premature status, etc. We'll use this data to explore the relationship between maternal smoking and premature births:

## 16.8. EXAMPLE: MEASURING THE ASSOCIATION BETWEEN MATERNAL SMOKING AND PREMATURE BIRTHS

```

births <- read_tsv("https://raw.githubusercontent.com/Bio204-class/bio204-datasets/master/births.txt")
#> Parsed with column specification:
#> cols(
#>   fAge = col_integer(),
#>   mAge = col_integer(),
#>   weeks = col_integer(),
#>   premature = col_character(),
#>   visits = col_integer(),
#>   gained = col_integer(),
#>   weight = col_double(),
#>   sexBaby = col_character(),
#>   smoke = col_character()
#> )

xtabs(~premature + smoke, births)
#>
#>           smoke
#> premature nonsmoker smoker
#>   full term      87     42
#>   premie        13      8

```

### 16.8.1 Null and alternative hypotheses:

- $H_0$ : there is no association between maternal smoking and premature birth
- $H_A$ : there is an association between maternal smoking and premature birth

### 16.8.2 Contingency table analysis using the $\chi^2$ statistic

In our class session on contingency analysis we introduced the  $\chi^2$  statistic as a measure of association between categorical variables (see previous notes).

Giving a contingency table of observed counts, we can calculate the expected counts under independence of the variables. Based on the observed and expected counts, the  $\chi^2$  (chi-squared) statistic is defined as:

$$\chi^2 = \frac{(O_{11} - E_{11})^2}{E_{11}} + \frac{(O_{12} - E_{12})^2}{E_{12}} + \dots + \frac{(O_{mn} - E_{mn})^2}{E_{mn}} \quad (16.1)$$

(16.2)

$$= \sum_{i=1}^m \sum_{j=1}^n \frac{(O_{ij} - E_{ij})^2}{E_{ij}} \quad (16.3)$$

where  $m$  and  $n$  are the number of categories of the two variables under consideration. The larger the  $\chi^2$ -statistic the stronger the evidence that the categorical variables are *not independent*.

### 16.8.3 $\chi^2$ -distribution

This sampling distribution of the  $\chi^2$ -statistic *when the rows and columns variables of a contingency table are independent* is the  $\chi^2$ -distribution. The shape of the  $\chi^2$ -distribution depends on a parameter called the “degrees of freedom” (abbreviated df). For contingency table analysis, the degrees of freedom is:  $df = (m - 1)(n - 1)$  where  $m$  and  $n$  are the number of rows and columns of the table.

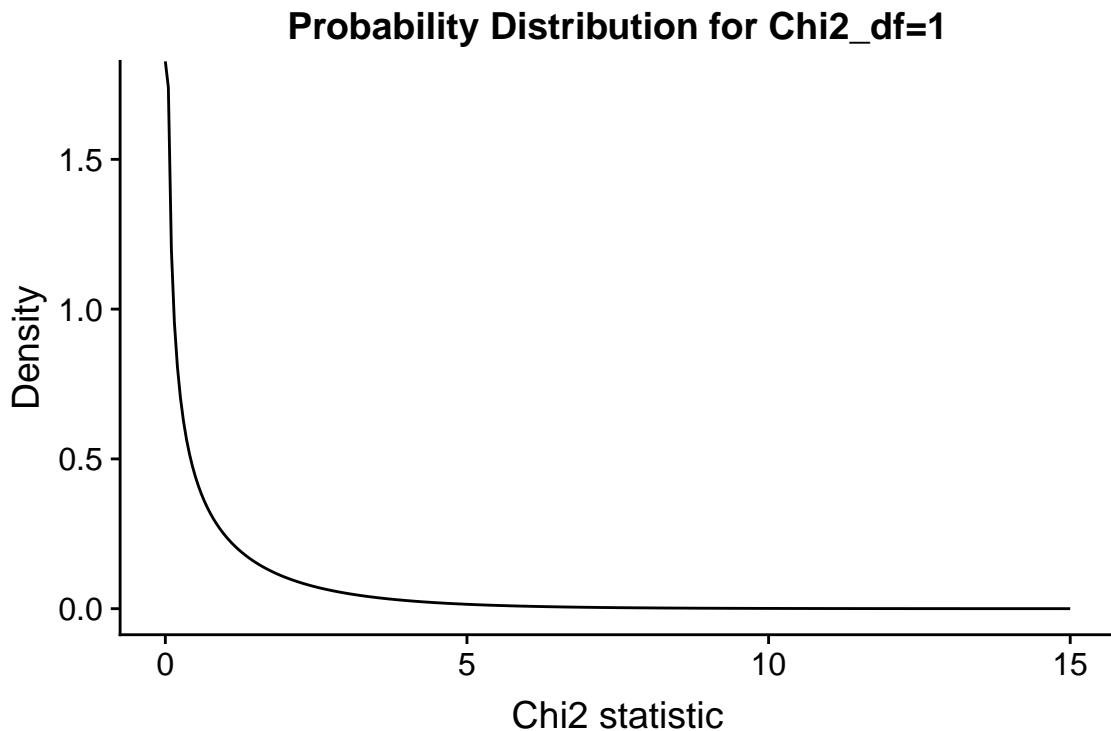
Here is the  $\chi^2$ -distribution with  $df = 1$ :

```

chi2.values <- seq(0, 15, by = 0.05)
chi2.density <- dchisq(chi2.values, df = 1)
chi2.df <- data.frame(chi2 = chi2.values, density = chi2.density)

ggplot(chi2.df, aes(chi2, density)) +
  geom_line() +
  labs(x = "Chi2 statistic", y = "Density",
       title = "Probability Distribution for Chi2_df=1")

```



For  $\chi^2$  analysis of contingency tables, hypothesis tests are always one-tailed. That is, in contingency analysis we are always asking “what is the probability of observing a  $\chi^2$  statistic at least this large under the null hypothesis of no association?”

#### 16.8.4 Carrying out a hypothesis test using the $\chi^2$

Under the null distribution, the sampling distribution of the  $\chi^2$  statistic is given by the  $\chi^2$  distribution with  $df = 1$ . We use the built-in `chisq.test` function to calculate the  $\chi^2$  value for the observed data and to calculate a corresponding p-value.

```

chi2.births <- chisq.test(births$premature, births$smoke, correct = FALSE)
chi2.results <- glance(chi2.births)
chi2.results
#> # A tibble: 1 x 4
#>   statistic p.value parameter method
#>       <dbl>    <dbl>      <int> <chr>
#> 1     0.2492   0.6177          1 Pearson's Chi-squared test

```

## 16.8. EXAMPLE: MEASURING THE ASSOCIATION BETWEEN MATERNAL SMOKING AND PREMATURE BIRTHS

### 16.8.5 Interpretation of the $\chi^2$ -test

Our point estimate of the  $\chi^2$  statistic for the births data is 0.2491694 and the associated p-value is 0.6176605. Given this, we say we “fail to reject the null hypothesis of no difference the rate of premature births between smoking and non-smoking mothers”.

For further insight into why the  $\chi^2$  statistic is small in this case, compare the observed and expected counts. As you’ll see, the values are barely differnt:

```
chi2.births$observed
#>           births$smoke
#>   births$premature nonsmoker smoker
#>       full term      87     42
#>       premie        13      8
chi2.births$expected
#>           births$smoke
#>   births$premature nonsmoker smoker
#>       full term      86     43
#>       premie        14      7
```

As mentioned previously, failing to reject a null hypothesis is not the same as accepting the null hypothesis. When it comes to human health and disease and environmental exposures, the magnitude of effects is often very small and you often need thousands of samples to detect an effect. For example, a relationship between maternal smoking and premature births is well supported by multiple large studies. However, the sample we’re using in this case is rather modest (150 births) and hence not well powered to detect modest effects. We’ll discuss this further when we talk about the topic of statistical power.



# Chapter 17

## Introduction to confidence intervals

Recall the concept of the **sampling distribution of a statistic** – this is simply the probability distribution of the statistic of interest you would observe if you took a large number of random samples of a given size from a population of interest and calculated that statistic for each of the samples.

You learned that the standard deviation of the sampling distribution of a statistic has a special name – the **standard error** of that statistic. The standard error of a statistic provides a way to quantify the uncertainty of a statistic across random samples. Here we show how to use information about the standard error of a statistic to calculate plausible ranges for a statistic of interest that take into account the uncertainty of our estimates. We call such plausible ranges **Confidence Intervals**.

### 17.1 Confidence Intervals

We know that given a random sample from a population of interest, the value of a statistic of interest is unlikely to be exactly equal to the true population value of that statistic. However, our simulations have taught us a number of things:

1. As sample size increases, the *sample estimate* of the given statistic is more likely to be close to the true value of that statistic
2. As sample size increases, the standard error of the statistic decreases

We will define an “X% percent confidence interval for a statistic of interest”, as an interval (upper and lower bound) that when calculated from a random sample, would include the true population value of the statistic of interest, X% of the time.

This quote from the NIST page on confidence intervals, which I’ve adapted to refer to any statistic, helps to make this concrete regarding confidence intervals:

As a technical note, a 95% confidence interval does not mean that there is a 95% probability that the interval contains the true [statistic]. The interval computed from a given sample either contains the true [statistic] or it does not. Instead, **the level of confidence is associated with the method of calculating the interval** ... That is, for a 95% confidence interval, if many samples are collected and the confidence interval computed, in the long run about 95% of these intervals would contain the true [statistic].

The idea behind a 95% confidence interval is illustrated in the following figure:

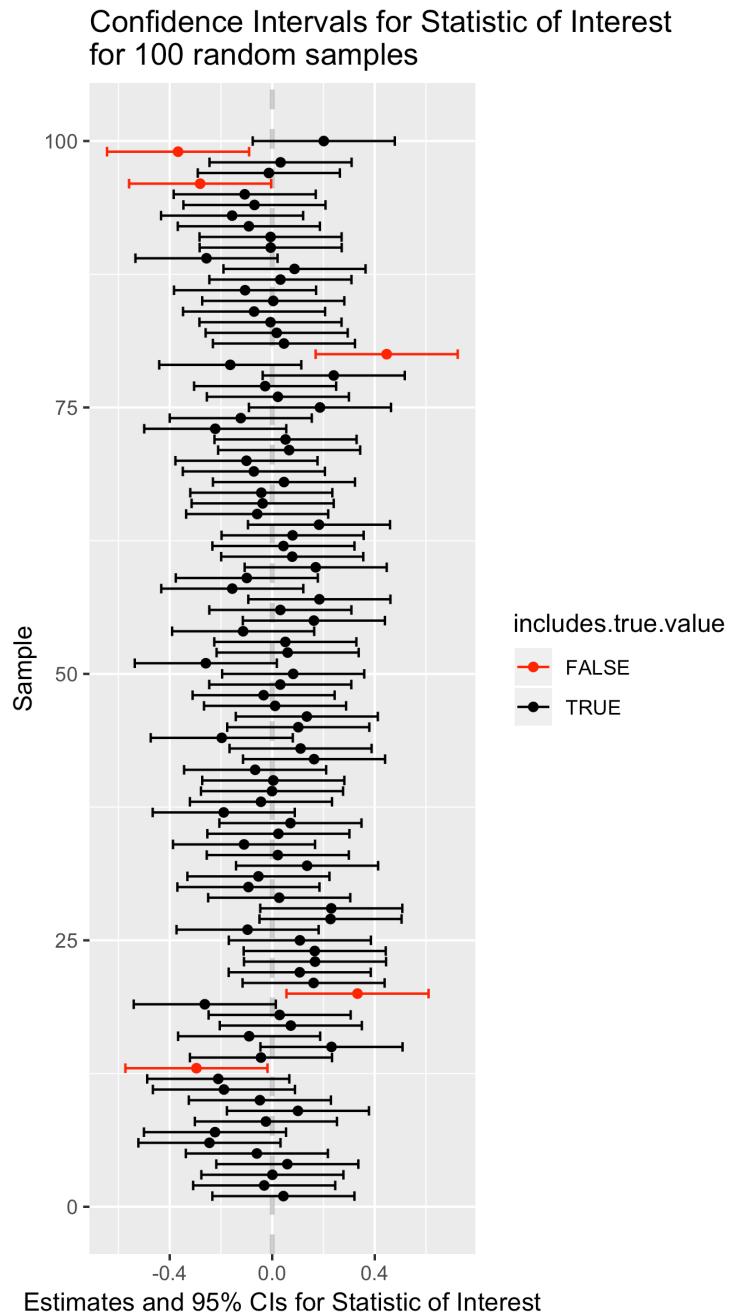


Figure 17.1: Point estimates and confidence intervals for a theoretical statistic of interest.

## 17.2 Generic formulation for confidence intervals

We define the  $(100 \times \beta)\%$  confidence interval for the statistic  $\phi$  as the interval:

$$CI_\beta = \phi_n \pm (z \times SE_{\phi,n})$$

Where:

- $\phi_n$  is the statistic of interest in a random sample of size  $n$
- $SE_{\phi,n}$  is the standard error of the statistic  $\phi$  (via simulation or analytical solution)

And the value of  $z$  is chosen so that:

- across many different random samples of size  $n$ , the true value of the  $\phi$  in the population of interest would fall within the interval approximately  $(100 \times \beta)\%$  of the time

So rather than estimating a single value of  $\phi$  from our data, we will use our observed data plus knowledge about the sampling distribution of  $\phi$  to estimate a range of plausible values for  $\phi$ . The size of this interval will be chosen so that if we considered many possible random samples, the true population value of  $\phi$  would be bracketed by the interval in  $(100 \times \beta)\%$  of the samples.

## 17.3 Example: Confidence intervals for the mean

To make the idea of a confidence interval more concrete, let's consider confidence intervals for the mean of a normally distributed variable.

Recall that if a variable  $X$  is normally distributed in a population of interest,  $X \sim N(\mu, \sigma)$ , then the sampling distribution of the mean of  $X$  is also normally distributed with mean  $\mu$ , and standard error  $SE_{\bar{X}} = \frac{\sigma}{\sqrt{n}}$ :

$$\bar{X} \sim N\left(\mu, \frac{\sigma}{\sqrt{n}}\right)$$

In our simulation we will explore how varying the value of  $z$  changes the percentage of times that the confidence interval brackets the true population mean.

### 17.3.1 Simulation of means

In our simulation we're going to generate a large number of samples, and for each sample we will calculate the sample estimate of the mean, and then quantify how much each sample mean differs from the true mean in terms of units of the population standard error of the mean. We'll then use this information to calibrate the wide of our confidence intervals.

For the sake of simplicity we'll simulate sampling from the "Standard Normal Distribution" – a normal distribution with mean  $\mu = 0$ , and standard deviation  $\sigma = 1$ .

First we load our standard libraries:

```
library(tidyverse)
library(magrittr)
```

Then we write our basic framework for our simulations:

```
rnorm.stats <- function(n, mu, sigma) {
  s <- rnorm(n, mu, sigma)
  df <- data_frame(sample.size = n,
    sample.mean = mean(s),
```

```

        sample.sd = sd(s),
        pop.SE = sigma/sqrt(n))
}

```

And then use this to simulate samples of size 25.

```

set.seed(20180328) # initialize RNG seed

true.mean <- 0
true.sd <- 11
n <- 25

samples.25 <-
  rerun(10000, rnorm.stats(n, true.mean, true.sd)) %>%
  bind_rows()

```

### 17.3.2 Distance between sample means and true means

We append a new column to our `samples.25` data frame, which is the result of calculating the distance of each sample mean from the true mean, expressed in terms of units of the population standard error of the mean:

```

samples.25 <-
  samples.25 %>%
  mutate(z.pop = (sample.mean - true.mean)/pop.SE)

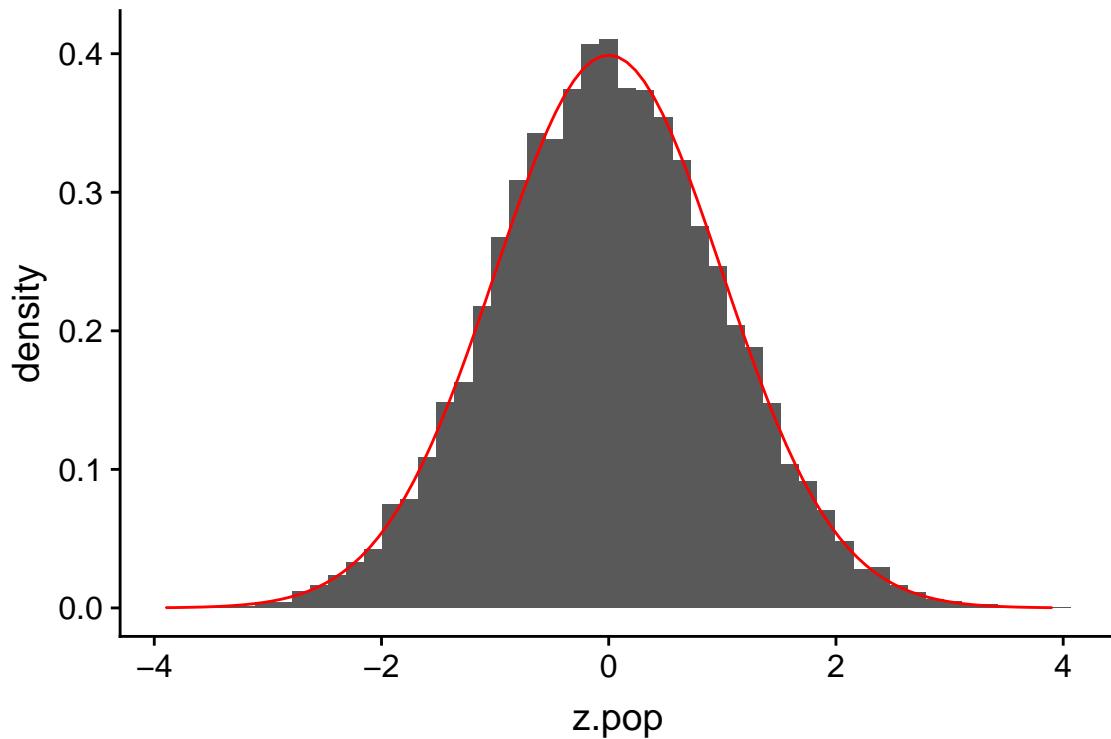
```

Since the sampling distribution of the mean of a normally distributed variable ( $N(\mu, \sigma)$ ) is itself normally distributed ( $N(\mu, SE_{\bar{X}})$ ), then the distribution of  $z = \frac{\bar{X} - \mu}{SE}$  is  $N(0, 1)$ . This is illustrated in the figure below where we compare our simulated z-scores to the theoretical expectation:

```

SE <- 1
ggplot(samples.25) +
  geom_histogram(aes(x = z.pop, y=..density..), bins=50) +
  stat_function(fun = function(x){dnorm(x, mean=0, sd=SE)}, color="red")

```



For a given value of  $z$  we can ask what fraction of our simulated means fall within  $\pm z$  standard errors of the true mean.

```
samples.25 %>%
  summarize(frac.win.1SE = sum(abs(z.pop) <= 1)/n(),
            frac.win.2SE = sum(abs(z.pop) <= 2)/n())
#> # A tibble: 1 x 2
#>   frac.win.1SE  frac.win.2SE
#>   <dbl>        <dbl>
#> 1     0.6826      0.9549
```

We see that roughly 68% of our sample means are within 1 SE of the true mean; ~95% are within 2 SEs.

If we wanted to get exact multiples of the SE corresponding to different percentiles of the distribution of z-scores, based on the theoretical result ( $z$  scores  $\sim N(0, 1)$ ), we can use the `qnorm()` function:

```
frac.of.interest <- c(0.68, 0.90, 0.95, 0.99)

# we use 1 - frac to get left most critical value
# we divide by two here to account for area under left and right tails
left.critical.value <- qnorm((1 - frac.of.interest)/2, mean = 0, sd=1)

data_frame(Percentile = frac.of.interest * 100,
           Critical.value = abs(left.critical.value))
#> # A tibble: 4 x 2
#>   Percentile Critical.value
#>       <dbl>        <dbl>
#> 1       68          0.9945
#> 2       90          1.645
#> 3       95          1.960
#> 4       99          2.576
```

### 17.3.3 Calculating a CI

If we knew the standard error of the mean for variable of interest, in order to a confidence interval we could simply look up the corresponding critical value for our percentile of interest in a table like the one above and calculate our CI as:

$$\bar{X} \pm \text{critical value} \times SE_{\bar{X}}$$

For example, we see that the critical value for 95% CIs is  $\sim 1.96$ .

## 17.4 A problem arises!

If you're a critical reader you should have noticed that calculating confidence intervals using the above formula presumes we know the standard error of the mean for the variable of interest. If we knew the standard deviation,  $\sigma$ , of our variable, we could calculate this as  $SE_{\bar{x}} = \frac{\sigma}{\sqrt{n}}$  but in general we do not know  $\sigma$  either.

Instead we must estimate the standard error of the mean using our sample standard deviation:

$$\widehat{SE}_{\bar{x}} = \frac{s_x}{\sqrt{n}}$$

This introduces another level of uncertainty and also a complication. The complication is due to the fact that for small samples, sample estimates of the standard deviation tend to be biased (smaller) relative to the true population standard deviation (see workbook Chapter 15).

In the assignment for this class session you will explore how we can deal with the fact of biased estimates of standard deviations for small samples, and how it effects our calculation of confidence intervals for the mean.

# Chapter 18

## Normal distributions

This exposition is based on Diez et al. 2015, OpenIntro Statistics (3rd Edition) and Whitlock and Schluter, 2015. The Analysis of Biological Data (2nd Edition).

### 18.1 Basics about normal distributions

Normal distributions are:

- Unimodal
- Symmetric
- Described by two parameters –  $\mu$  (mean) and  $\sigma$  (standard deviation)

#### 18.1.1 Notation

- If a variable  $X$  is approximately normally distributed with mean,  $\mu$ , and standard deviation,  $\sigma$ , we write:  $X \sim N(\mu, \sigma)$

### 18.2 Normal distribution, probability density function

The probability density function for a normal distribution  $N(\mu, \sigma)$  is described by the following equation:

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

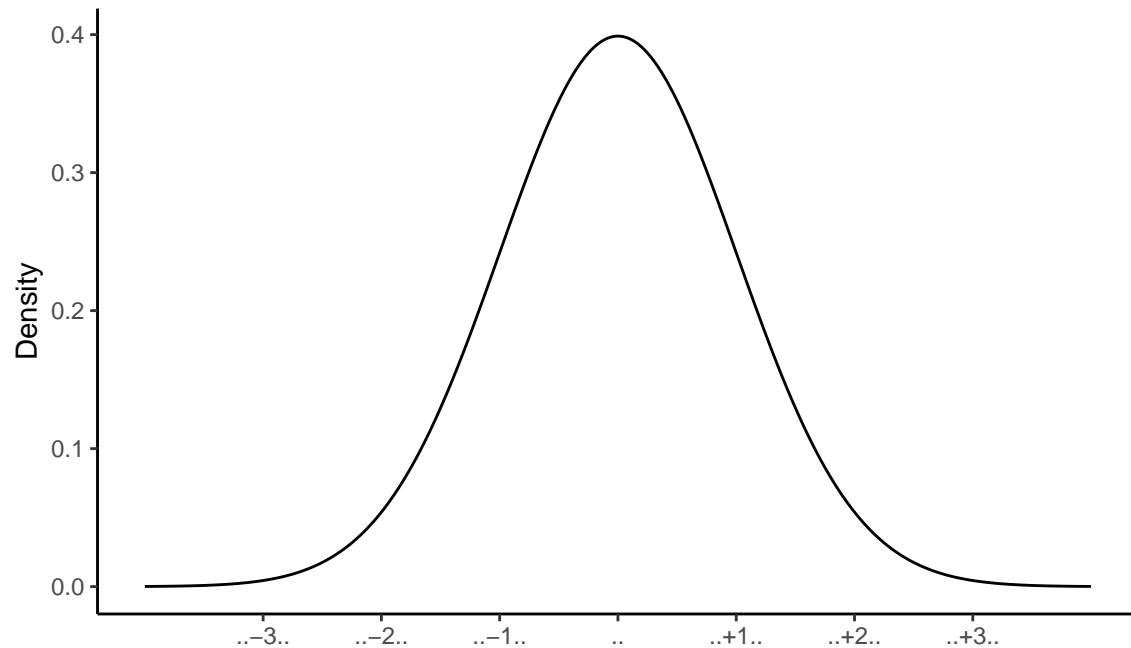
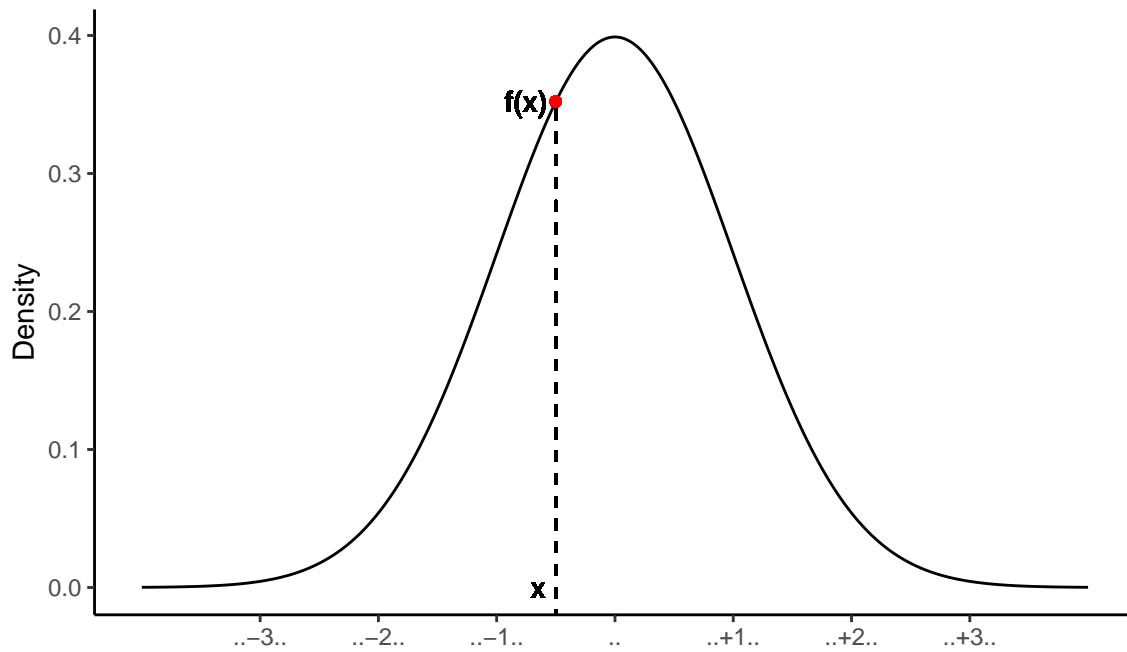


Figure 18.1: A normal distribution with mean  $\mu$  and standard deviation  $\sigma$



### 18.2.1 `dnorm()` calculates the normal pdf

- In R, the function `dnorm(x, mu, sigma)` calculates the probability density of  $N(\mu, \sigma)$  at the point  $x$

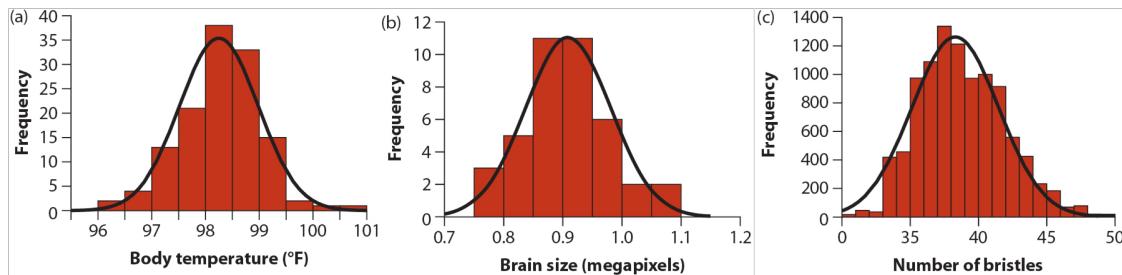


Figure 18.2: Examples of biological variables that are nearly normal. From Whitlock and Schluter, Chap 10

## 18.3 Approximately normal distributions are very common

The normal approximation is a good approximation to patterns of variation seen in biology, economics, and many other fields.

The following figure, from your textbook, shows distributions for (a) human body temperature, (b) university undergraduate brain size, and (c) numbers of abdominal bristles on Drosophila fruit flies:

## 18.4 Central limit theorem

Why is the normal distribution so ubiquitous? A key reason is the “Central Limit Theorem”

The **Central Limit Theorem (CLT)** states the sum or mean of a large number of random measurements sampled from a population is approximately normally distributed, *regardless of the shape of the distribution from which they are drawn*.

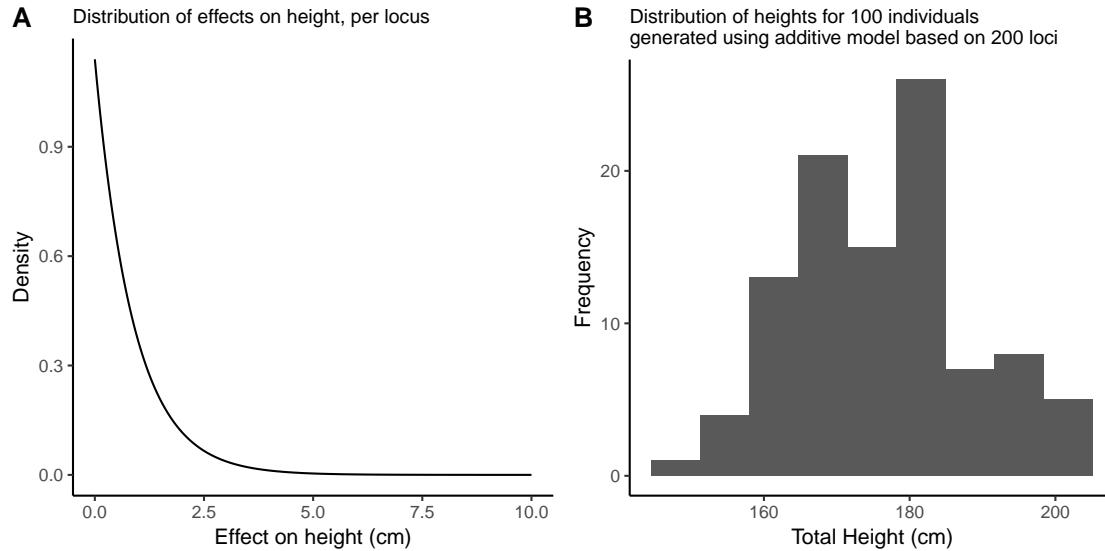
Many biological traits can be thought of as being produced by the summation many small effects. Even if those effect have a non-normal distribution, the sum of their effects is approximately normal.

### 18.4.1 Example: Continuous variation from discrete loci

Studies of the genetic basis of traits like height or weight, indicate that traits like these have a “multigenic” basis. That is, there are many genomic regions (loci) that each contribute a small amount to difference in height among individuals.

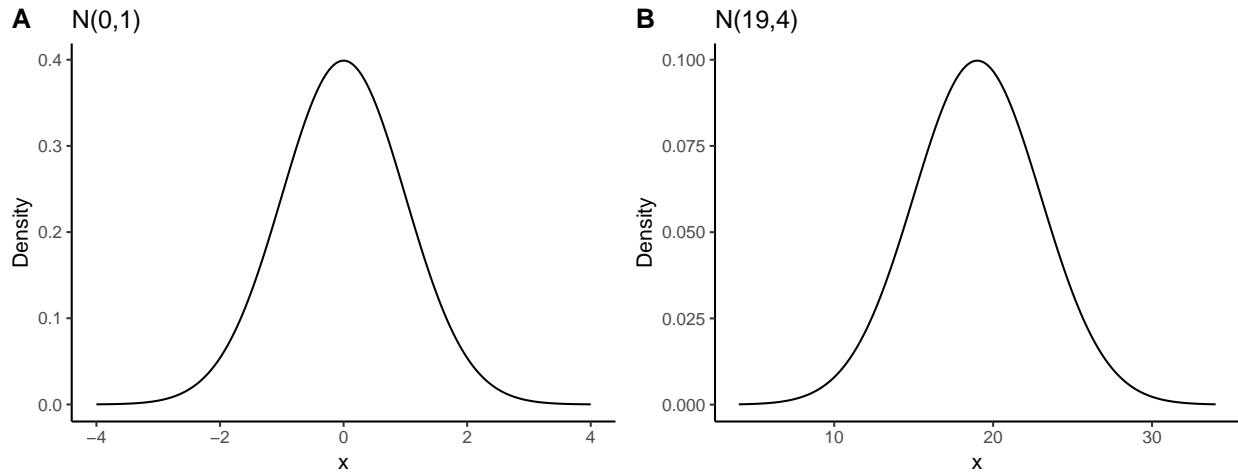
For the sake of illustration let’s assume there are 200 loci scattered across the genome that affect height. And that each locus has an effect on size that is exponentially distributed with a mean of 0.8cm, and that an individual’s total height is the sum of the effects at each of these individual loci.

In the figure below, the first plot show what the distribution of effect sizes looks. This is quite clearly a non-normal distribution. The second plot show what the distribution of heights of 100 individuals generated using the additive model above would look like. This second plot is approximately normal, as predicted by the CLT.

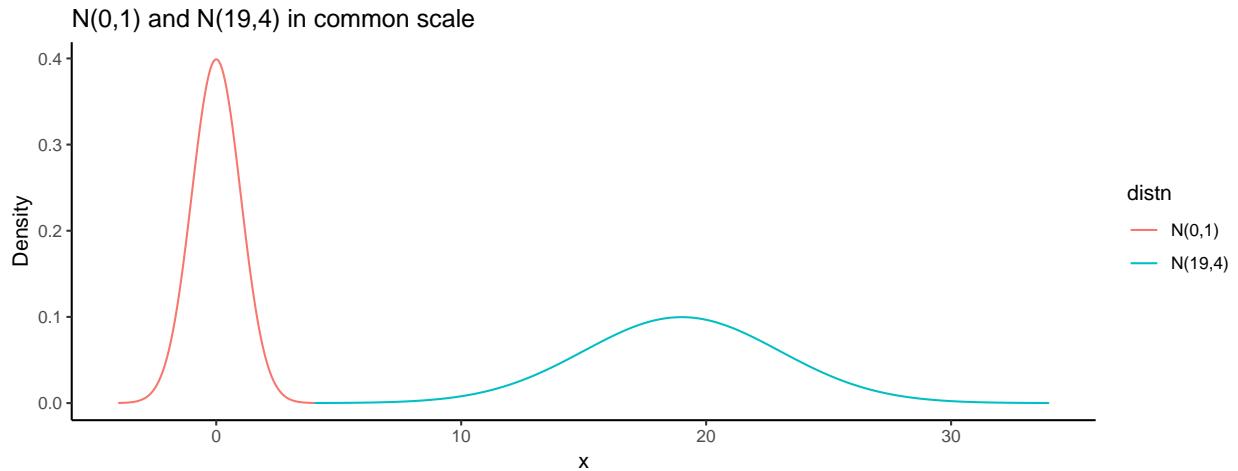


## 18.5 Visualizing normal distributions

- Different normal distributions look alike when plotted on their own scales

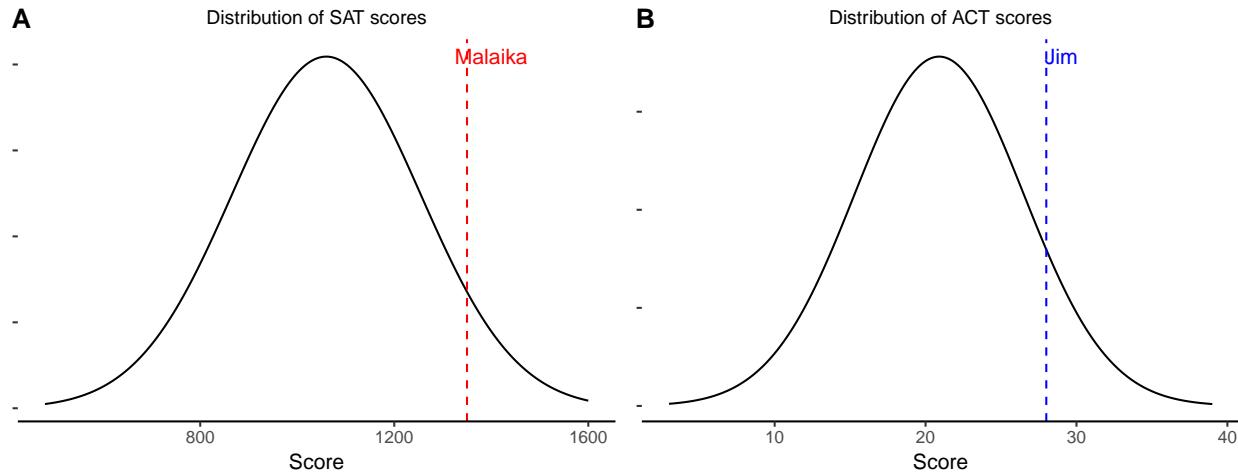


- Must plot normals on a common scale to see the differences



## 18.6 Comparing values from different normal distributions

Q: SAT scores are approximately normally distributed with a mean of 1060 and a standard deviation of 195. ACT scores are approximately normal with a mean of 20.9 and a standard deviation of 5.6. A college admissions officer wants to determine which of the two applicants scored better on their standardized test with respect to the other test takers: Malaika, who earned an 1350 on her SAT, or Jim, who scored a 28 on his ACT?



A: Since the scores are measured on different scales we can not directly compare them, however we can measure the difference of each score in terms of units of standard deviation

### 18.6.1 Standardized or Z-scores

Differences from the mean, measured in units of standard deviation are called “standardized scores” or “Z-scores”

- The Z score of an observation is the number of standard deviations it falls above or below the mean.

$$Z_i = \frac{x_i - \mu}{\sigma}$$

- For our SAT/ACT example above:

$$Z_{\text{Malaika}} = \frac{1350 - 1060}{195} = 1.49 \quad (18.1)$$

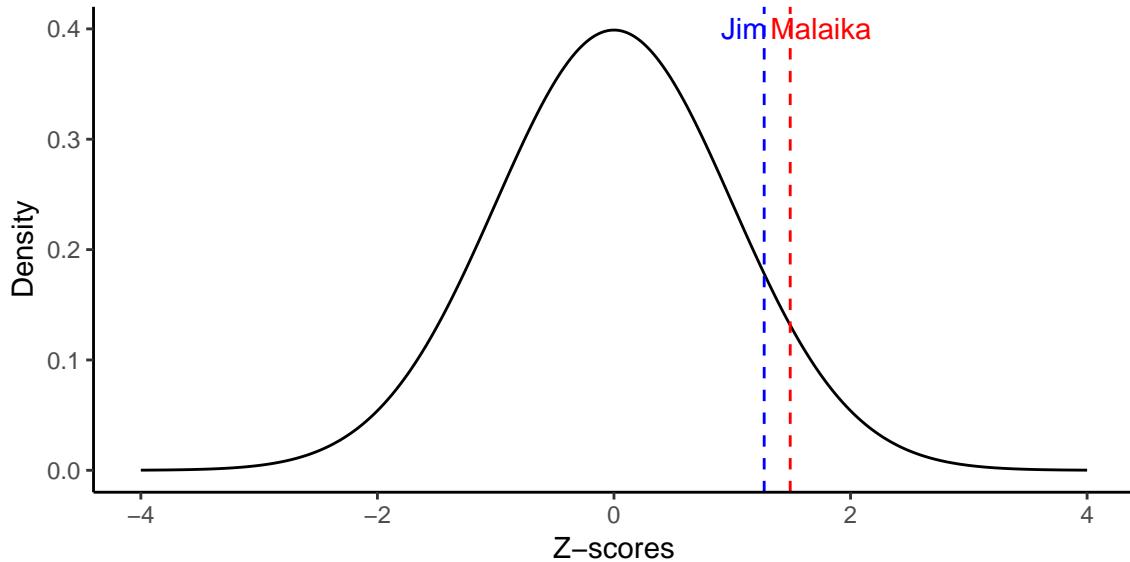
(18.2)

$$Z_{\text{Jim}} = \frac{28 - 20.9}{5.6} = 1.27 \quad (18.3)$$

(18.4)

In this case, Malaika's score is 1.49 standard deviation above the mean, while Jim's is 1.27. Based on this, Malaika scored better than Jim.

### Comparison of Malaika's and Jim's standardized college entrance exam scores



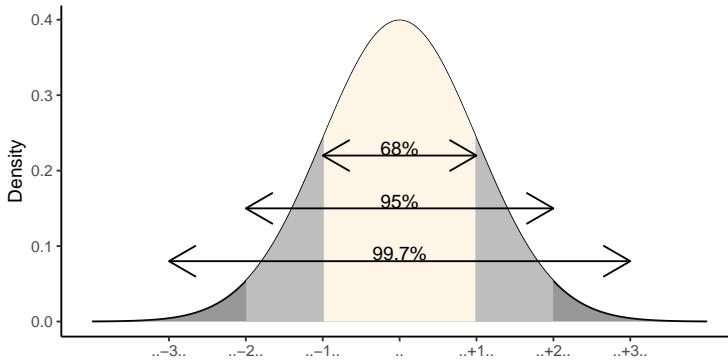
## 18.7 Standard normal distribution

- If  $X \sim N(\mu, \sigma)$  then the standardized distribution,  $Z_X \sim N(0, 1)$ . If  $X$  is normally distributed, then the Z-scores based on  $X$  have a mean of 0 and a standard deviation of 1.
- $N(0, 1)$  is known as the **standard normal distribution**

## 18.8 88-95-99.7 Rule

If data are approximately normally distributed:

- ~68% of observations lie within 1 SD of the mean
- ~95% of observations lie within 2 SD of the mean
- ~99.7% of observations lie within 3 SD of the mean



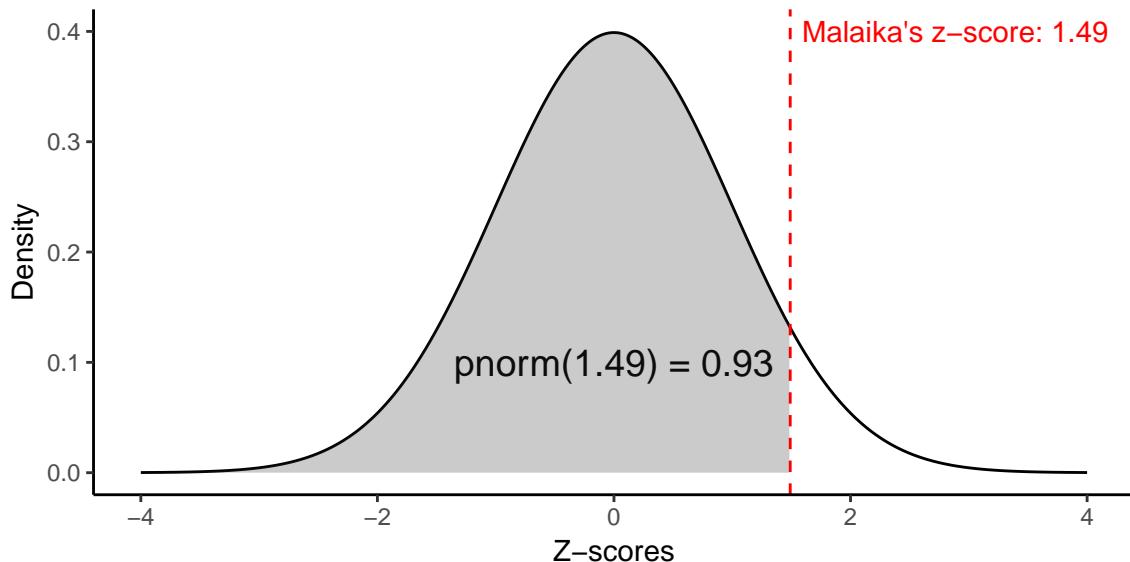
## 18.9 Percentiles

- The *percentile* is the percentage of observations that fall below a given point,  $q$
- In R, for a normal distribution the fraction of observations below a given point (the probability that a random observation drawn from the distribution is less than the given value) can be calculated using the `pnorm(q, mu, sigma)` function:

```
# Malaika's z-score was 1.49. What percentile was she in?
pnorm(1.49)
#> [1] 0.9318879
```

Therefore, Malaika is approximately at the 93-percentile. Note that we didn't have to include the mean and standard deviation in the call to `pnorm` because we're dealing with standardized scores, and the defaults for `pnorm` are `mean = 0` and `sd = 1`. A similar calculation would show that Jim's percentile is 89.7957685.

Malaika scored higher than ~93% of other exam takers on the SAT



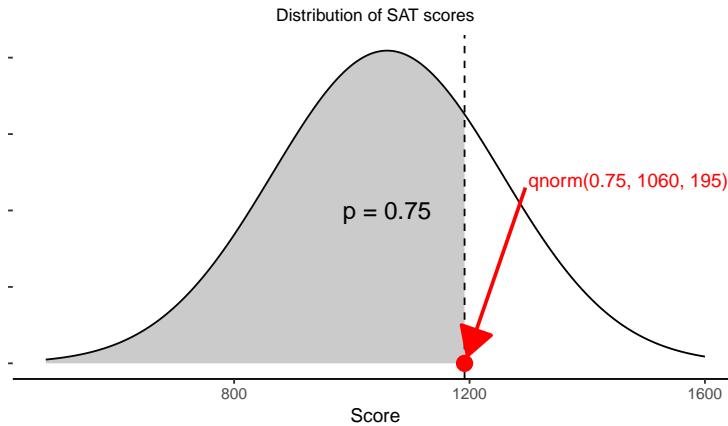
Note that if we want the fraction of the data to the right of a value  $q$ , we can subtract the value from one ( $1 - \text{pnorm}(1.49)$ ) or set the `lower.tail = FALSE` argument in `pnorm`.

```
pnorm(1.49, lower.tail = FALSE) # same as (1 - pnorm(1.49))
#> [1] 0.06811212
```

## 18.10 Cutoff points

- When we use the `pnorm()` function we specify a point,  $q$ , and it gives us the corresponding fraction of values,  $p$ , that fall below that point in a normal distribution
- If instead we want to specify a fraction  $p$ , and get the corresponding point,  $q$ , on the normal distribution, we use the `qnorm(p, mu, sigma)` function.

```
# To get the 75-th percentile (3rd quartile) of SAT scores
# based on the parameters provided previously
qnorm(0.75, 1060, 195)
#> [1] 1191.526
```



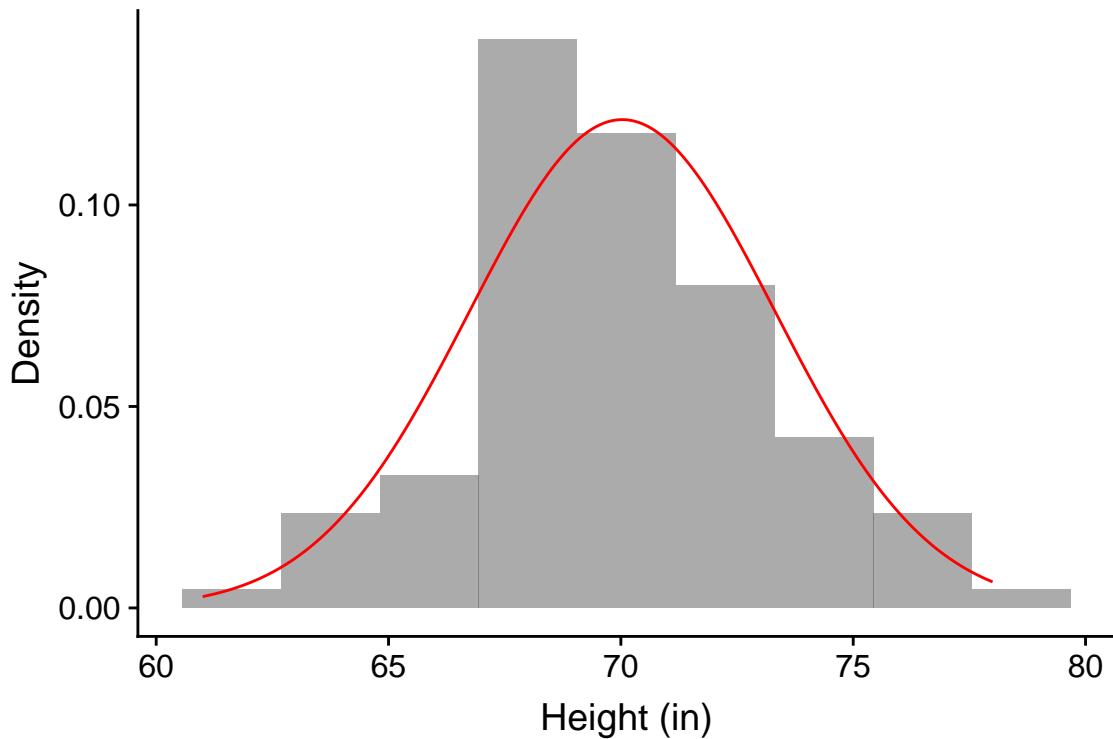
## 18.11 Assessing normality

There are a number of graphical tools we have at our disposal to assess approximate normality based on observations of a variable of interest. There are also some formal tests we can apply. Here we focus on the graphical tools.

### 18.11.1 Comparing histograms to theoretical normals

One of the simplest approaches to assessing approximate normality for a variable of interest is to plot a histogram of the observed variable, and then to overlay on that histogram the probability density function you would expect for a normal distribution with the same mean and standard deviation.

In the example below I show a histogram of heights from a sample of 100 men, overlaid with the PDF of a normal distribution with the mean and standard deviation as estimated from the sample.

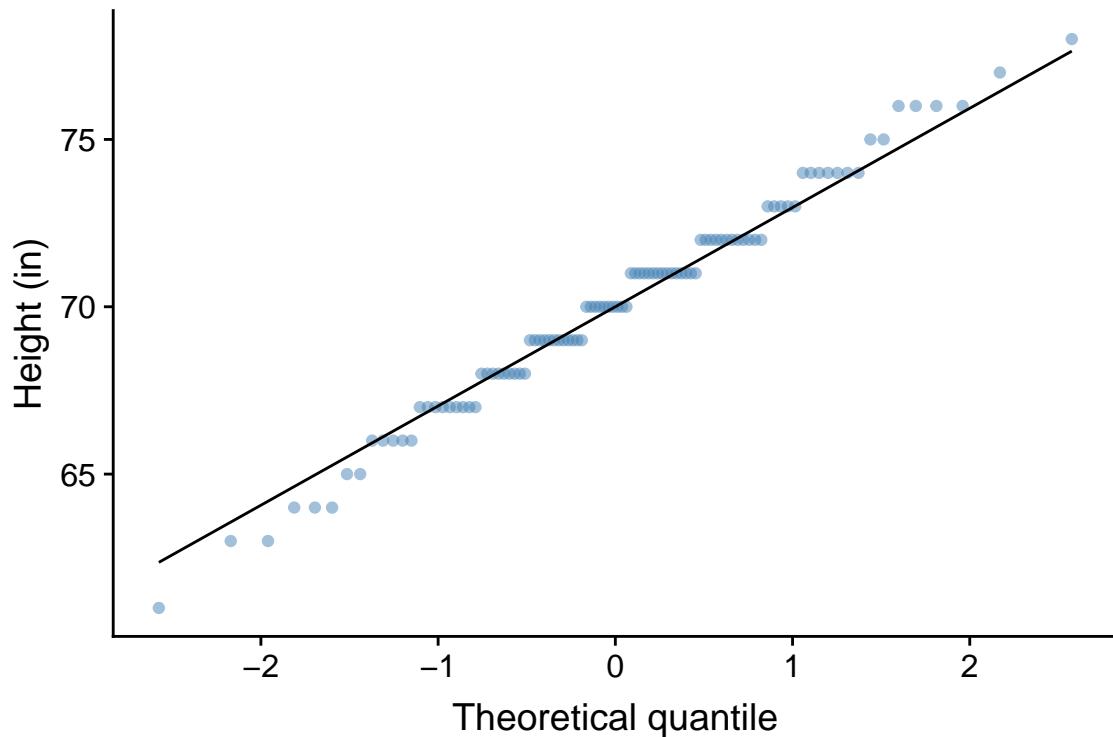


The histogram matches fairly well to the theoretical normal, but histograms are rather coarse visualizations when sample sizes are modest.

### 18.11.2 Normal probability plot

A second graphical tool for assessing normality is a “normal probability plot”. A normal probability plot is a type of scatter plot for which the x-axis represents theoretical quantiles of a normal distribution, and the y-axis represents the observed quantiles of our observed data. If the observed data perfectly matched the normal distribution with the same mean and standard deviation, then all the points should fall on a straight line. Deviations from normality are represented by runs of points off the line.

The `ggplot` functions `geom_qq()` and `geom_qq_line()` take care of the necessary calculations required to generate a normal probability plot. Here is the normal probability plot for the male height data:



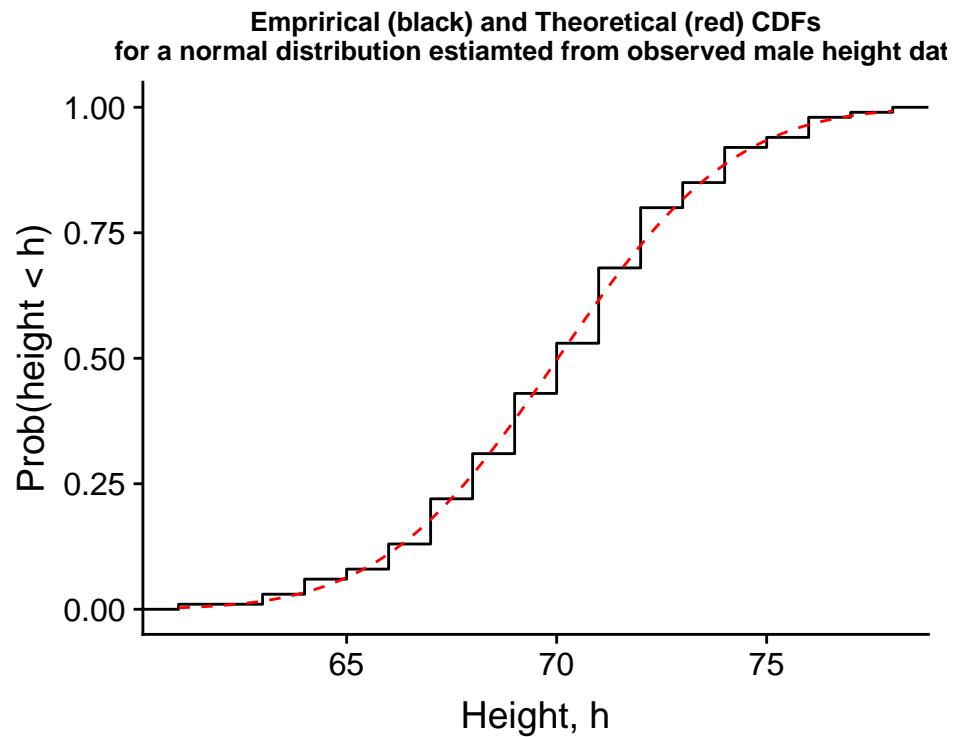
This plot suggests that the male heights are approximately normally distributed, though there are maybe a few more very short men and a few less very tall men in our sample than we would expect under perfect normality.

### 18.11.3 Comparing the empirical CDF to the theoretical CDF

A third visual approach is to estimate a cumulative distribution function (CDF) for the variable of interest from the data and compare this to the theoretical cumulative distribution function you'd expect for a normal distribution (as provided by `pnorm()`). When you estimate a cumulative distribution function from data, this is called an “empirical CDF”.

The function `ggplot::stat_ecdf` estimates the empirical CDF for us and plots it, we can combine this with `stat_function()` to plot the theoretical CDF using `pnorm`, as shown below for the height data.

```
male.heights %>%
  ggplot(aes(x = heights)) +
  stat_ecdf() +
  stat_function(fun=pnorm,
               args=list(mean = mean.height, sd = sd.height),
               color='red', linetype='dashed', n = 200) +
  labs(x = "Height, h", y = "Prob(height < h)",
       title = "Empirical (black) and Theoretical (red) CDFs\nfor a normal distribution estimated from the data") +
  theme(plot.title = element_text(size=10))
```



Here the match between the empirical CDF and the theoretical CDF is pretty good, again suggesting that the data is approximately normal.



# Chapter 19

## Comparing sample means

### 19.1 Hypothesis test for the mean using the t-distribution

We consider three different situations for hypothesis tests regarding means.

### 19.2 One sample t-test

A one sample t-test is appropriate when you want to compare an observed sample mean to some a priori hypothesis about what the mean should be.

- $H_0$ : The mean of variable  $X$  equals  $\mu_0$  (some a priori value for the mean)
- $H_A$ : The mean of variable  $X$  does not equal  $\mu_0$

#### 19.2.1 One sample t-test, test statistic

To carry out a one sample t-test, first calculate the test statistic:

$$t^* = \frac{\bar{x} - \mu_0}{SE_{\bar{x}}}$$

where  $\bar{x}$  is the sample mean and  $SE_{\bar{x}}$  is the sample standard error of the mean ( $SE_{\bar{x}} = s_x/\sqrt{n}$ ). In words,  $t^*$  measures the difference between the observed mean, and the null hypothesis mean, in unit of standard error.

To calculate a P-value, we compare the test statistic to the t-distribution with the appropriate degrees of freedom to calculate the probability that you'd observe a mean value at least as extreme as  $\bar{x}$  if the null hypothesis was true. For a two-tailed test this is:

$$P = P(t < -|t^*|) + P(t > |t^*|)$$

Since the t-distribution is symmetric, this simplifies to:

$$P = 2 \times P(t > |t^*|)$$

### 19.2.2 Assumptions of one sample t-tests

- Data are randomly sampled from the population
- The variable of interest is approximately normally distributed

### 19.2.3 Example: Gene expression in mice

You are an investigator studying the effects of various drugs on the expression of key genes that regulate apoptosis. The gene YFG1 is one such gene of interest. It has been previously established, using very sample sizes, that average expression level of the gene YFG1 in untreated (control) mice is 10 units.

You treat a sample of mice with Drug X, and measure the expression of the gene YFG1 following treatment. For a sample of five mice you observe the following expression values:

- $YFG1 = \{11.25, 10.5, 12, 11.75, 10\}$

You wish to determine whether the average expression of YFG1 in mice treated with the drug differs from control mice. The null and alternative hypotheses for this hypothesis test are:

- $H_0$ : the mean expression of YFG1 is 10
- $H_A$ : the mean expression of YFG1 does not equal 10

It's relatively easy to calculate the various quantities of interest needed to carry out a one sided t-test:

```
library(tidyverse)

mu0 = 10 # mean under H0
mice.1sample <- data_frame(YFG1 = c(11.25, 10.5, 12, 11.75, 10))

YFG1.tstats <-
  mice.1sample %>%
  summarize(sample.mean = mean(YFG1),
           sample.sd = sd(YFG1),
           sample.se = sample.sd/sqrt(n()),
           df = n() - 1,
           mu0 = mu0,
           t.star = (sample.mean - mu0)/sample.se,
           P.value = 2 * pt(abs(t.star), df = df, lower.tail = FALSE))

YFG1.tstats
#> # A tibble: 1 x 7
#>   sample.mean sample.sd sample.se     df    mu0 t.star P.value
#>       <dbl>      <dbl>      <dbl> <dbl> <dbl>  <dbl>
#> 1       11.1      0.8404    0.3758     4    10  2.927  0.04295
```

Under the conventional  $\alpha = 0.05$ , we reject the null hypothesis that the mean expression of YFG1 in mice treated with the drug is the same as the mean expression of YFG1 in control mice.

### 19.2.4 Confidence intervals for the mean

The hypothesis test using a one-side t-test we carried out above is approximately equivalent to asking if the null mean,  $\mu_0$ , falls within the 95% confidence intervals estimated for the sample mean.

Recall that the  $100(1 - \alpha)\%$  confidence interval for the mean can be calculated as

$$CI_{\alpha} = \bar{x} \pm (t_{\alpha/2, df} \times \widehat{SE}_{\bar{x}})$$

Let's modify the table of  $t$ -related stats we created in the previous code block to include the lower and upper limits of the 95% confidence interval for the mean:

```
YFG1.tstats <-
  YFG1.tstats %>%
  mutate(ci95.lower = sample.mean - abs(qt(0.025, df = df)) * sample.se,
        ci95.upper = sample.mean + abs(qt(0.025, df = df)) * sample.se)

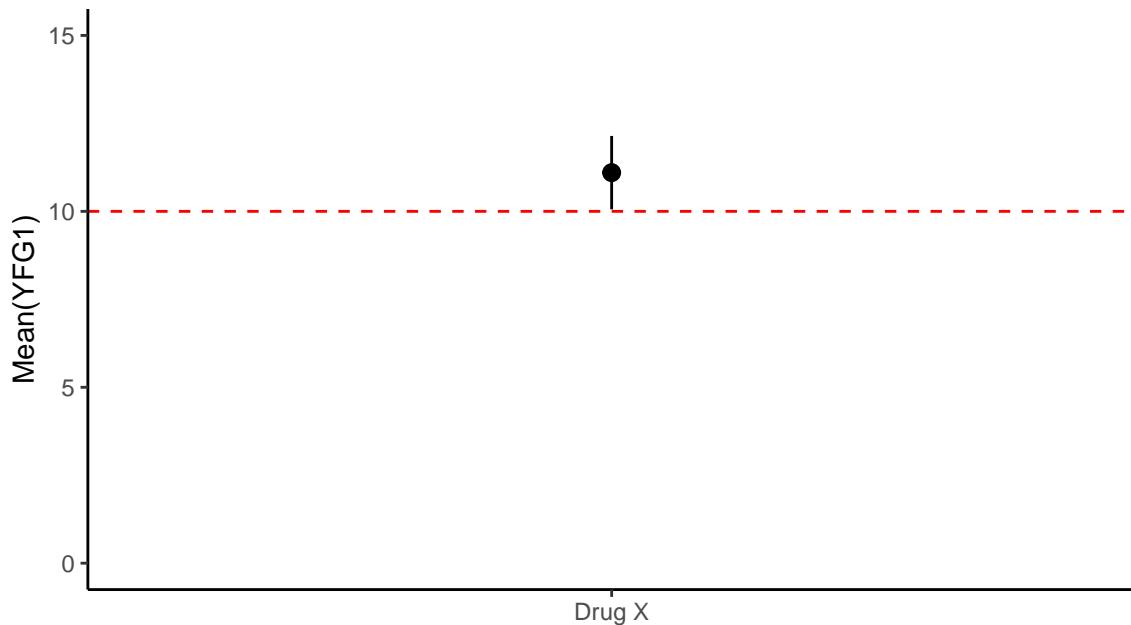
YFG1.tstats
#> # A tibble: 1 x 9
#>   sample.mean sample.sd sample.se     df    mu0 t.star P.value ci95.lower
#>       <dbl>      <dbl>     <dbl>   <dbl> <dbl>   <dbl>    <dbl>      <dbl>
#> 1       11.1     0.8404    0.3758     4     10  2.927  0.04295     10.06
#> # ... with 1 more variable: ci95.upper <dbl>
```

If we wanted to create a figure illustrating our 95% CI for the mean of YFG1 following drug treatment, relative to the mean under the null hypothesis we can use the `geom_pointrange()` function to draw the interval (for more info on `geom_pointrange()` and related functions see the ggplot2 documentation).

```
YFG1.tstats %>%
  ggplot() +
  geom_pointrange(aes(x = 1,
                       y = sample.mean,
                       ymin = ci95.lower, ymax = ci95.upper)) +
  scale_x_continuous(breaks=c(1), labels=c("Drug X")) +
  geom_hline(yintercept = mu0, color='red', linetype = 'dashed') +
  ylim(0,15) +
  labs(x = "", y = "Mean(YFG1)",
       title = "Sample mean and 95% CI for expression of YFG1 in treated mice",
       subtitle = "Red line indicates mean expression in control mice.") +
  theme_classic() +
  theme(plot.title = element_text(size = 12),
        plot.subtitle = element_text(size = 10))
```

Sample mean and 95% CI for expression of YFG1 in treated mice

Red line indicates mean expression in control mice.



### 19.3 The `t.test` function in R

The built-in `t.test()` function will take care of all the calculations we did by hand above.

For a one sample t-test `t.test` we need to pass in the variable of interest, and the null hypothesis mean value, `mu`:

```
YFG1_t.test <- t.test(mice.1sample$YFG1, mu = 10)
YFG1_t.test
#>
#> One Sample t-test
#>
#> data: mice.1sample$YFG1
#> t = 2.9268, df = 4, p-value = 0.04295
#> alternative hypothesis: true mean is not equal to 10
#> 95 percent confidence interval:
#> 10.05652 12.14348
#> sample estimates:
#> mean of x
#> 11.1
```

The `broom::tidy()` function defined in the “broom” package is a convenient way to display the output of many model tests in R. Load the broom (install it first if need be) and use `tidy()` to represent the information returned by `t.test()` as a data frame:

```
library(broom)

tidy(YFG1_t.test)
#> # A tibble: 1 x 8
```

```
#>   estimate statistic p.value parameter conf.low conf.high method
#>   <dbl>      <dbl>      <dbl>      <dbl>      <dbl> <chr>
#> 1    11.1     2.927  0.04295      4     10.06    12.14 One S-
#> # ... with 1 more variable: alternative <chr>
```

## 19.4 Two sample t-test

We use a two sample t-test to analyze the difference between the means of the same variable measured in two different groups or treatments. It is assumed that the two groups are independent samples from two populations.

- $H_0$ : The mean of variable  $X$  in group 1 is the same as the mean of  $X$  in group 2, i.e.,  $\bar{x}_1 = \bar{x}_2$ . This is equivalent to  $\bar{x}_1 - \bar{x}_2 = 0$ .
- $H_A$ : The mean of variable  $X$  in group 1 is not the same as the mean of  $X$  in group 2, i.e.  $\bar{x}_1 \neq \bar{x}_2$ . This is equivalent to  $\bar{x}_1 - \bar{x}_2 \neq 0$ .

### 19.4.1 Standard error for the difference in means

In a two-sample t-test, we have to account for the uncertainty associated with the means of both groups, which we express in terms of the standard error of the difference in the means between the groups:

$$SE_{\bar{x}_1 - \bar{x}_2} = \sqrt{s_p^2 \left( \frac{1}{n_1} + \frac{1}{n_2} \right)}$$

where

$$s_p^2 = \frac{df_1 s_1^2 + df_2 s_2^2}{df_1 + df_2}$$

$s_p^2$  is called the “pooled sample variance” and is a weighted average of the sample variances,  $s_1^2$  and  $s_2^2$ , of the two groups.

### 19.4.2 Two sample t-test, test statistic

Given the standard error for the difference in means between groups as defined above, we define our test statistic for a two sample t-test as:

$$t^* = \frac{(\bar{x}_1 - \bar{x}_2)}{SE_{\bar{x}_1 - \bar{x}_2}}$$

The degrees of freedom for this test statistic are:

$$df = df_1 + df_2 = n_1 + n_2 - 2$$

### 19.4.3 Assumptions of two sample t-test

- Data are randomly sampled from the population
- Paired differences are normally distributed
- Standard deviation is the same in both populations

### 19.4.4 Example: Comparing the effects of two drugs

You treat samples of mice with two drugs, X and Y. We want to know if the two drugs have the same average effect on expression of the gene *YFG1*. The measurements of *YFG1* in samples treated with X and Y are as follows:

- X = {11.25, 10.5, 12, 11.75, 10}
- Y = {8.75, 10, 11, 9.75, 10.5}

For simplicity, we skip the “by-hand” calculations and simply use the built-in `t.test` function.

```
mice.2sample <- data_frame(YFG1_X = c(11.25, 10.5, 12, 11.75, 10),
                             YFG1_Y = c(8.75, 10, 11, 9.75, 10.5))

ttest.2sample <-
  t.test(mice.2sample$YFG1_X, mice.2sample$YFG1_Y)

ttest.2sample
#>
#> Welch Two Sample t-test
#>
#> data: mice.2sample$YFG1_X and mice.2sample$YFG1_Y
#> t = 2.0605, df = 7.9994, p-value = 0.07331
#> alternative hypothesis: true difference in means is not equal to 0
#> 95 percent confidence interval:
#> -0.1310858 2.3310858
#> sample estimates:
#> mean of x mean of y
#> 11.1 10.0
```

This output provides information on:

- the data vectors used in this analysis
- t, df, and p-value
- the alternative hypothesis
- the 95% CI for the difference between the group means
- the group means

Using a type I error cutoff of  $\alpha = 0.05$ , we fail to reject the null hypothesis that the mean expression of *YFG1* is different in mice treated with Drug X versus those treated with Drug Y.

As we saw previously, `broom::tidy()` is a good way to turn the results of the `t.test()` function into a convenient table for further computation or plotting:

```
tidy(ttest.2sample)
#> # A tibble: 1 x 10
#>   estimate estimate1 estimate2 statistic p.value parameter conf.low
#>   <dbl>     <dbl>     <dbl>     <dbl>    <dbl>    <dbl>
#> 1 1.100     11.1      10       2.060  0.07331    7.999  -0.1311
#> # ... with 3 more variables: conf.high <dbl>, method <chr>,
#> #   alternative <chr>
```

### 19.4.5 Specifying `t.test()` in terms of a formula

In the example above, our data frame included two columns for YFG1 expression values – `YFG1_X` and `YFG1_Y` – representing the expression measurements under the two drug treatments. This is not a very “tidy” way to organize our data, and is somewhat limiting when we want to create plots and do other analyses. Let’s use some of the tools we’ve seen earlier for tidying and restructuring data frames to unite these into a single column, and create a new column indicating treatment type:

```
mice.long <-  
  mice.2sample %>%  
  gather(expt, expression) %>%  
  separate(expt, c("gene", "treatment"), sep="_")  
  
head(mice.long)  
#> # A tibble: 6 x 3  
#>   gene   treatment expression  
#>   <chr>  <chr>      <dbl>  
#> 1 YFG1    X            11.25  
#> 2 YFG1    X            10.5  
#> 3 YFG1    X            12  
#> 4 YFG1    X            11.75  
#> 5 YFG1    X            10  
#> 6 YFG1    Y            8.75
```

Using this “long” data frame we can carry out the `t.test` as follows:

```
ttest.2sample <- t.test(expression ~ treatment, data = mice.long)  
ttest.2sample  
#>  
#> Welch Two Sample t-test  
#>  
#> data: expression by treatment  
#> t = 2.0605, df = 7.9994, p-value = 0.07331  
#> alternative hypothesis: true difference in means is not equal to 0  
#> 95 percent confidence interval:  
#> -0.1310858 2.3310858  
#> sample estimates:  
#> mean in group X mean in group Y  
#>           11.1             10.0
```

This long version of the data is also more easily used for calculating confidence intervals of the mean for each treatment, and for plotting as illustrated below

```
ci.by.treatment <-  
  mice.long %>%  
  group_by(treatment) %>%  
  summarize(mean = mean(expression),  
            se = sd(expression)/sqrt(n()),  
            tcrit = abs(pt(0.025, df = n() - 1)),  
            ci.low = mean - tcrit * se,  
            ci.hi = mean + tcrit * se)  
  
ci.by.treatment  
#> # A tibble: 2 x 6  
#>   treatment  mean     se   tcrit ci.low ci.hi  
#>   <chr>      <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
```

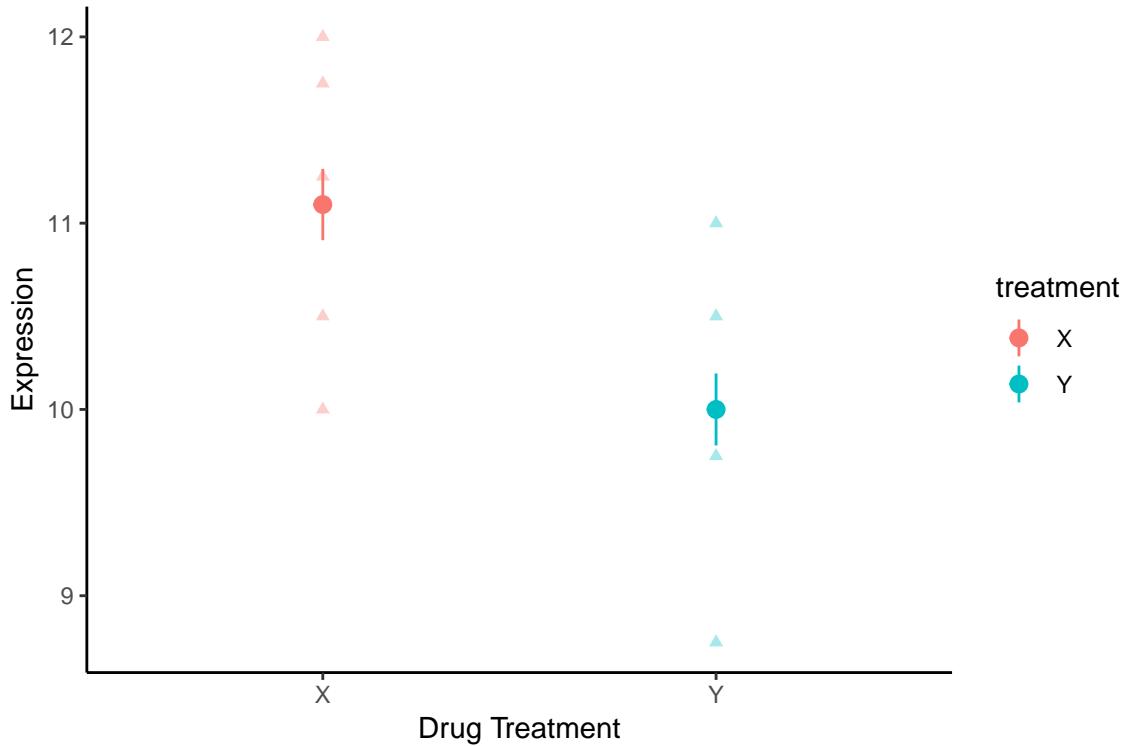


Figure 19.1: Expression of YFG1 for different drug treatments. Triangles are individual measurements. Circles and lines indicate group means and 95% CIs of means.

```
#> 1 X      11.1 0.3758 0.5094 10.91 11.29
#> 2 Y      10   0.3791 0.5094  9.807 10.19
```

Here I combine the raw expression measurements and mean and confidence intervals into a single plot:

```
mice.long %>%
  ggplot(aes(x = treatment, y = expression, color=treatment)) +
  geom_point(alpha = 0.35, shape=17) +
  geom_pointrange(data = ci.by.treatment,
                  aes(x = treatment, y = mean,
                      ymin = ci.low, ymax= ci.hi)) +
  labs(x = "Drug Treatment", y = "Expression") +
  theme_classic() +
  theme(plot.caption = element_text(size=8))
```

## 19.5 Paired t-test

In a paired t-test there are two groups/treatments, but the samples in the two groups are paired or matched. This typically arises in “before-after” studies where the same individual/object is measured at different time points, before and after application of a treatment. The repeated measurement of the same individual/object means that we can’t treat the two sets of observations as independent. Null and alternative hypotheses are thus typically framed in terms of a mean difference between time points/conditions

- $H_0$ : The mean difference of variable  $X$  in the paired measurements is zero, i.e.,  $\bar{D} = 0$  where  $D =$

$$X_{\text{after}} - X_{\text{before}}$$

- $H_A$ : The mean difference of variable  $X$  in the paired measurements is not zero, i.e.,  $\bar{D} \neq 0$

### 19.5.1 Paired t-test, test statistic

- Let the variable of interest for individual  $i$  in the paired conditions be designated  $x_{i,\text{before}}$  and  $x_{i,\text{after}}$
- Let  $D_i = x_{i,\text{after}} - x_{i,\text{before}}$  be the paired difference for individual  $i$
- Let  $\bar{D}$  be the mean difference and  $s_D$  be the standard deviation of the differences
- The standard error of the mean difference is  $SE(\bar{D}) = \frac{s_D}{\sqrt{n}}$

The test statistic is thus:

$$t^* = \frac{\bar{D}}{SE(\bar{D})}$$

Under the null hypothesis, this statistic follows a t-distribution with  $n - 1$  degrees of freedom.

### 19.5.2 Assumptions of paired t-test

- Data are randomly sampled from the population
- Paired differences are normally distributed

### 19.5.3 Paired t-test, example

You measure the expression of gene  $YFG1$  in five mice. You then treat those five mice with drug Z and measure gene expression again.

- $YFG1$  expression before treatment = {12, 11.75, 11.25, 10.5, 10}
- $YFG1$  expression after treatment = {11, 10, 10.5, 8.75, 9.75}

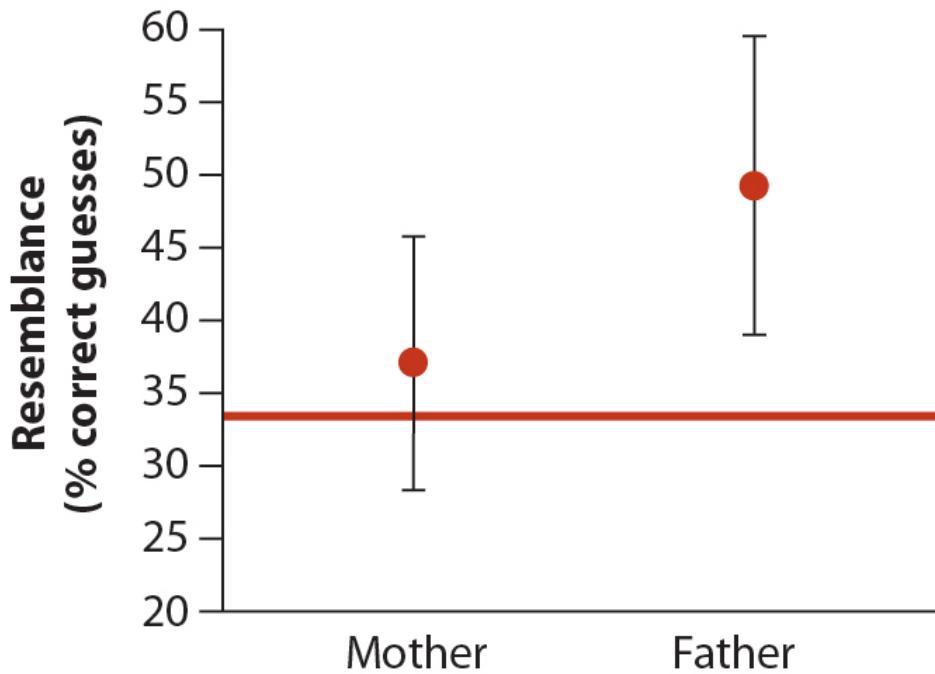
```
mice.paired <- data_frame(YFG1.before = c(12, 11.75, 11.25, 10.5, 10),
                            YFG1.after = c(11, 10, 10.5, 8.75, 9.75))

t.test(mice.paired$YFG1.before, mice.paired$YFG1.after,
       paired = TRUE)
#>
#>   Paired t-test
#>
#> data: mice.paired$YFG1.before and mice.paired$YFG1.after
#> t = 3.773, df = 4, p-value = 0.01955
#> alternative hypothesis: true difference in means is not equal to 0
#> 95 percent confidence interval:
#>  0.2905341 1.9094659
#> sample estimates:
#> mean of the differences
#>                      1.1
```

Using a type I error cutoff of  $\alpha = 0.05$ , we reject the null hypothesis of no difference in the average expression of  $YFG1$  before and after treatment with Drug Z.

## 19.6 The fallacy of indirect comparison

WS 12.5 considers an example where baby photos were compared to photos of their mother, father, and several unrelated pictures. Volunteers were asked to identify the mother and father of each baby, and the accuracy of their choices was compared between mothers and fathers. In Fig. 12.5-1 (below), the horizontal line shows the null hypotheses expected for random guessing, while means and 95% CIs are shown for mothers and fathers. The success rate for picking fathers was significantly better than random expectation, while the CI for mothers overlapped the null expected value. Given these CIs, the authors of the study concluded that babies resemble their fathers more than their mothers.



This is an example of the fallacy of indirect comparison: “Comparisons between two groups should always be made directly, not indirectly by comparing both to the same null hypothesis” WS 12.5.

---

### 19.6.1 Interpreting confidence intervals in light of two sample t-tests

Figure 12.6-1 from WS (below) considers the relationship between overlap of confidence intervals and significant difference between groups ( $H_0$ : group means do not differ). Shown are group means and their 95% confidence intervals in three cases. (a) When 95% CIs do not overlap, then group means will be significantly different. (b) When the confidence interval of one group overlaps the mean value for the other group, then the groups are not significantly different. Finally, (c) when the confidence intervals overlap each other, but they do not overlap the mean of the other group, then the result of the hypothesis test is unclear.

In each case, figures with confidence intervals are helpful, but it is the P-value from our test of  $H_0$  that provides clear indication for our statistical conclusions.

## 19.7 Summary table for different t-tests

Here is a summary table giving the test statistic and degrees of freedom for each of the different types of t-tests described above. Notice that they all boil down to a difference of two means, expressed in units of

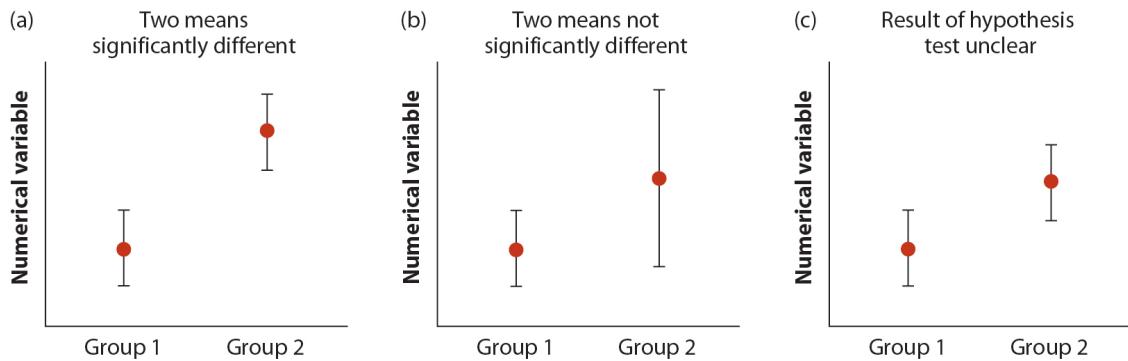


Figure 19.2: Figure from Whitlock and Schluter, Chapter 12.

standard error. The associated P-value associated with each test is thus a measure of how surprising that scaled difference in means is under the null model.

|            | test statistic   | df              |
|------------|--|-----------------|
| one-sample | $\frac{\bar{x} - \mu_0}{SE_{\bar{x}}}$                       | $n - 1$         |
| two-sample | $\frac{\bar{x}_1 - \bar{x}_2}{SE_{(\bar{x}_1 - \bar{x}_2)}}$ | $n_1 + n_2 - 2$ |
| paired     | $\frac{\bar{D} - 0}{SE_{\bar{D}}}$                           | $n - 1$         |



# Chapter 20

## Analysis of Variance

*t*-tests are the standard approach for comparing means between two groups. When you want to compare means between more than two groups a technique called “Analysis of Variance” (ANOVA) is used.

### 20.1 Hypotheses for ANOVA

When using ANOVA to compare means, the null and alternative hypotheses are:

- $H_0$ : The means of all the groups are equal
- $H_A$ : At least one of the means is different from the others

### 20.2 ANOVA, assumptions

ANOVA assumes:

- The measurements in every group represent a random sample from the corresponding population
- The variable of interest is normally distributed
- The variance is approximately the same in all the groups

### 20.3 ANOVA, key idea

The key idea behind ANOVA is that:

- If the observations in each group are drawn from populations with equal means (and variances) then the variation *between* group means should be similar to the inter-individual variation *within groups*.

### 20.4 Partitioning of sum of squares

Another way to think about ANOVA is as a “partitioning of variance”. The total variance among all the individuals across groups can be decomposed into:

- 1) variance of the group means around the “grand mean”;
- 2) variance of individuals around the group means.

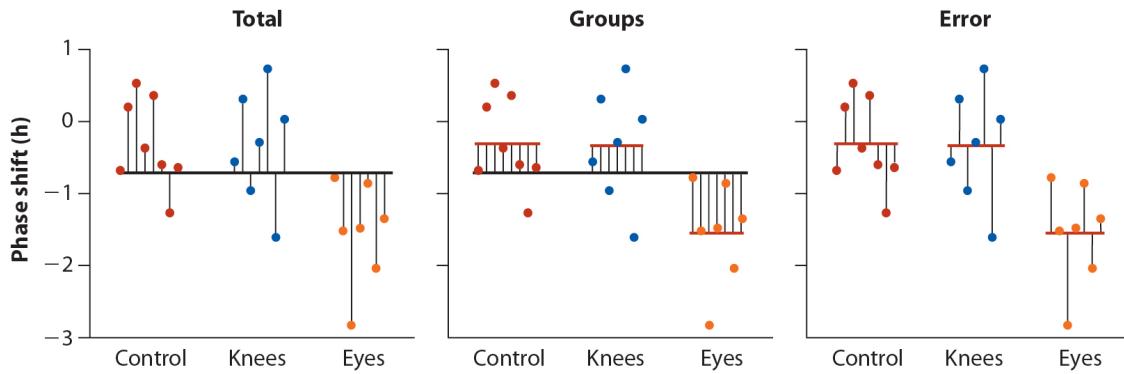


Figure 20.1: Whitlock and Schluter, Fig 15.1.2 – Illustrating the partitioning of sum of squares into  $MS_{group}$  and  $MS_{error}$  components.

However, rather than using variance we use sums of square deviations around the respective means (usually shortened to “sums of squares”).

This decomposition is represented visually in the figure below:

## 20.5 Mathematical partitioning of sums of squares

Variable  $X$  with a total sample of  $N$  observations, partitioned into  $k$  groups. The sample size of the  $g$ -th group is  $n_g$ , and thus  $N = \sum_{g=1}^k n_g$ . Let  $\bar{X}$  indicate the grand mean of  $X$  and  $\bar{X}_g$  indicate the mean of  $X$  in the  $g$ -th group.

### Total sums of squares

We call the sum of the squared deviations around the grand mean the “total sum of squares” ( $SS_{total}$ ).

$$SS_{total} = \sum_{i=1}^N (x_i - \bar{X})^2$$

- The total degrees of freedom is:  $df_{total} = N - 1$

### Group sum of squares and group mean square deviation

The sum of squared deviations of the group means around the grand mean is called the “group sum of squares”:

$$SS_{group} = \sum_{g=1}^k n_g (\bar{X}_g - \bar{X})^2$$

- The degrees of freedom associated with the group sum of squares is:  $df_{group} = k - 1$
- Define the “group mean squared deviation” as:

$$MS_{group} = \frac{SS_{group}}{k - 1}$$

### Error sum of squares and error mean square deviation

The sum of squared deviations of the individual observations about their respective group means is called the “error sum of squares”:

$$SS_{error} = \sum_{g=1}^k \sum_{i=1}^{n_g} (x_{i,g} - \bar{X}_g)^2$$

- The degrees of freedom associated with the error sum of squares is:  $df_{\text{error}} = N - k$
- Define the “error mean squared deviation” as:

$$MS_{\text{error}} = \frac{SS_{\text{error}}}{N - k}$$

**Variation explained:**  $R^2$

We can summarize the contribution of the group differences to the total variation in the data, using the  $R^2$  value.

To do this we note that the total sum of squares is the sum of the group and error sum of squares:

$$SS_{\text{total}} = SS_{\text{group}} + SS_{\text{error}}$$

The  $R^2$ -value is thus defined as:

$$R^2 = \frac{SS_{\text{groups}}}{SS_{\text{total}}}$$

## 20.6 ANOVA test statistic and sampling distribution

### F-statistic

The test statistic used in ANOVA is designated  $F$ , and is defined as follows:

$$F = \frac{\text{group mean square}}{\text{error mean square}} = \frac{MS_{\text{group}}}{MS_{\text{error}}}$$

- Under the null hypothesis, the between group and within group variances are similar and thus the  $F$  statistic should be approximately 1.
- Large values of the  $F$ -statistic means that the between group variance exceeds the within group variance, indicating that at least one of the means is different from the others

### The F-distribution

The sampling distribution of the  $F$ -statistic is called the  $F$ -distribution.

The  $F$ -distribution depends on two parameters:

- 1) the degrees of freedom associated with the group sum of squares,  $df_{\text{group}} = k - 1$ ;
- 2) the degrees of freedom associated with the error sum of squares,  $df_{\text{error}} = N - k$ ;

We designate a particular  $F$ -distribution as  $F_{k-1, N-k}$ . We will illustrate what an F-distribution looks like for particular parameters below.

## 20.7 ANOVA tables

The results of an analysis of variance test are often presented in the form of a table organized as follows:

| Source | SS                  | df      | MS                  | F                                     |
|--------|---------------------|---------|---------------------|---------------------------------------|
| Group  | $SS_{\text{group}}$ | $k - 1$ | $MS_{\text{group}}$ | $MS_{\text{group}}/MS_{\text{error}}$ |
| Error  | $SS_{\text{error}}$ | $N - k$ | $MS_{\text{error}}$ |                                       |
| Total  | $SS_{\text{total}}$ | $N - 1$ |                     |                                       |

## 20.8 The `aov()` function

As you would suspect, there is a built in R function to carry out ANOVA. This function is designated `aov()`. `aov` takes a formula style argument where the variable of interest is on the left, and the grouping variable indicated on the right.

```
aov(variable.of.interest ~ grouping.variable, data = df)
```

## 20.9 Example, circadian rythm data

Your textbook describes an exemplar data set from a study designed to test the effects of light treatment on circadian rhythms (see Whitlock & Schluter, Example 15.1).

- The investigators randomly assigned 22 individuals to one of three treatment groups and measured phase shifts in melatonin production. The treatment groups were:
  - control group (8 individuals)
  - light applied on the back of the knee (7 individuals)
  - light applied to the eyes (7 individuals)

These data are available at: ABD-circadian-rythms.csv

The null and alternative hypotheses associated with the ANOVA of these data are:

- $H_0$ : the means of the treatments groups are the same
- $H_1$ : the mean of at least one of the treatment groups is different from the others

### Libraries

```
library(tidyverse)
library(magrittr)
library(cowplot)
library(broom)
set.seed(20180113)
```

### Load the data

```
circadian <- read_csv("https://raw.githubusercontent.com/bio304-class/bio304-course-notes/master/datasets/circadian.csv")
```

The data is very simple: just two columns indicating treatment group and the variable of interest:

```
head(circadian, n=3)
#> # A tibble: 3 x 2
#>   treatment shift
#>   <chr>     <dbl>
#> 1 control    0.53
#> 2 control    0.36
#> 3 control    0.2
```

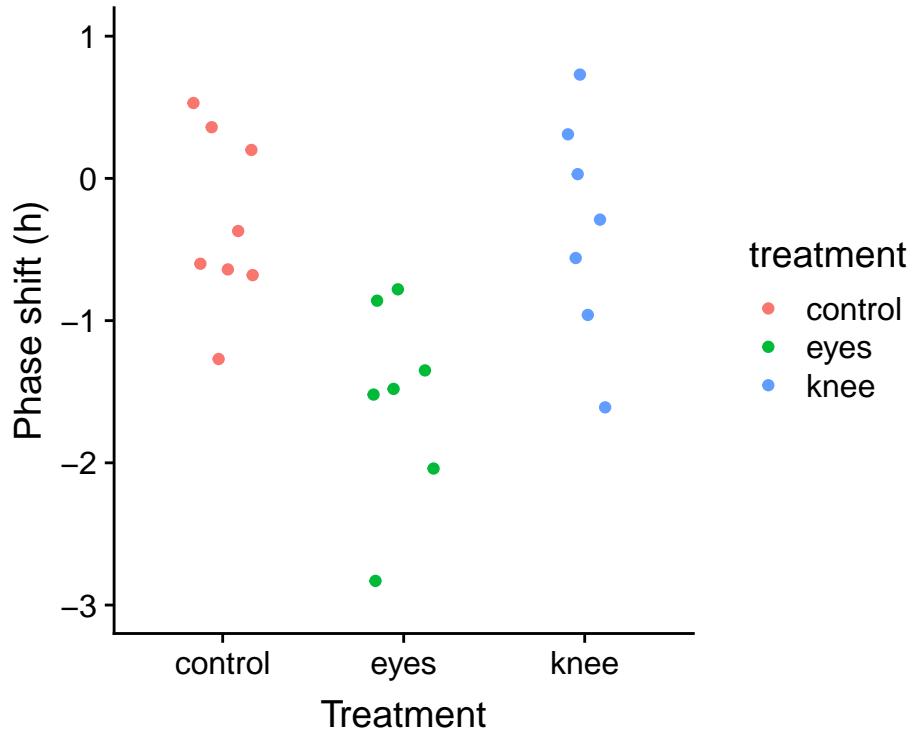
### 20.9.1 Visualizing the data

As is our standard practice, let's start by visualizing the data. We'll create a point plot depicting the observations colored by treatment group.

```
# we're going to re-use our jittering across plots so
# assign it to a variable
pd <- position_jitter(width=0.2, height=0)
```

```
point.plot <-
  circadian %>%
  ggplot(aes(x=treatment, y=shift,
             color=treatment, group=row.names(circadian))) +
  geom_point(position = pd) +
  ylim(-3,1) +
  labs(x = "Treatment", y = "Phase shift (h)")

point.plot
```



## 20.9.2 Carrying out the ANOVA

From our visualization it certainly seems like there may be differences among the group (treatment) means. Let's test this formally using the `aov()` function:

```
circadian.aov <- aov(shift ~ treatment, data = circadian)
```

The `summary` function applied to the `aov` fit will print out a typical ANOVA table and calculate the associated P-value for the  $F$  test statistic:

```
summary(circadian.aov)
#>           Df Sum Sq Mean Sq F value    Pr(>F)
#> treatment     2   7.224   3.612   7.289 0.00447 **
#> Residuals   19   9.415   0.496
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

If you want the ANOVA table in a form you can compute with, the `broom::tidy` function we explored previously comes in handy:

```

circadian.aov.table <- tidy(circadian.aov)
circadian.aov.table
#> # A tibble: 2 x 6
#>   term      df sumsq meansq statistic  p.value
#>   <chr>     <dbl> <dbl>  <dbl>    <dbl>
#> 1 treatment  2  7.224  3.612    7.289  0.004472
#> 2 Residuals 19  9.415  0.4955   NA      NA

```

The table above tells us that the  $F$ -statistic for this ANOVA is  $\sim 7.29$ . The table also tells us that the P-value associated with this  $F$ -statistic, is quite small,  $P\text{-value} < 0.005$ .

The P-value can be calculated explicitly using the `pf()` function (similar to the `pnorm()` and `pt()` functions we've seen previously):

```

circadian.F.stat <- circadian.aov.table$statistic[1]
pf(circadian.F.stat, df1 = 2, df2 = 19, lower.tail = FALSE)
#> [1] 0.004472271

```

Note that we set `lower.tail = FALSE` to calculate the probability of getting an  $F$ -statistic this large or greater.

## Visualizing the F-distribution

Let's draw the corresponding  $F$ -distribution,  $F_{2,19}$ , using the `df()` function (parallel to `dnorm()` and `dt()`), with the region corresponding to an  $F$ -statistic greater than 7.29 shaded red. Because the area in the right tail we're interested in is quite small, in a second plot we've zoomed in on this region.

```

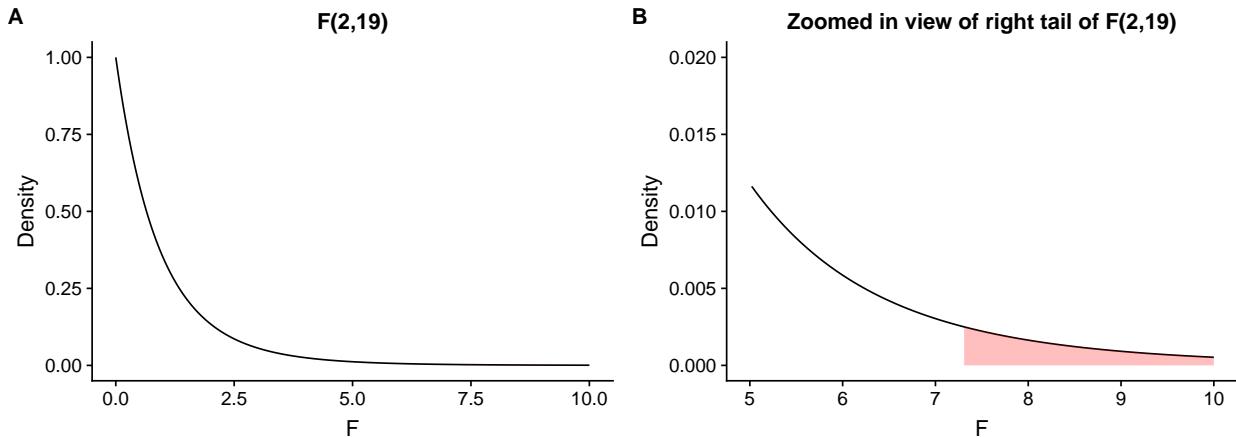
fdist <- data_frame(f = seq(0, 10, length.out = 250),
                      density = df(f, df1=2, df2=19))

plot.a <-
  ggplot(fdist, aes(f, density)) +
  geom_line() +
  geom_area(data = filter(fdist, f >= 7.29), fill='red', alpha=0.25) +
  labs(x = "F", y = "Density", title="F(2,19)")

plot.b <-
  plot.a + xlim(5,10) + ylim(0, 0.02) +
  labs(title = "Zoomed in view of right tail of F(2,19)")

plot_grid(plot.a, plot.b, labels="AUTO")

```



### Critical values of the F-distribution

If we wanted to know what the critical F value is for a corresponding type I error rate we can use the `qf()` function:

```
# the critical value of F for alpha = 0.05
qf(0.05, 2, 19, lower.tail = FALSE)
#> [1] 3.521893
```

## 20.10 ANOVA calculations: Step-by-step

The `aov()` function carries out all the ANOVA calculations behind the scenes. It's useful to pull back the curtain and see how the various quantities are calculated.

### Total SS

```
grand.mean <- mean(circadian$shift)

# total sum of squares
total.table <-
  circadian %>%
  summarize(SS = sum((shift - grand.mean)**2),
            df = n() - 1)

total.table
#> # A tibble: 1 x 2
#>       SS     df
#>   <dbl> <dbl>
#> 1 16.64    21
```

### Group SS and MS

We use `group_by` and `summarize` to calculate group means and the group deviates (the difference between the group means and the grand mean):

```
group.df <-
  circadian %>%
  group_by(treatment) %>%
  summarize(n = n(),
            group.mean = mean(shift),
            grand.mean = grand.mean,
```

```

group.deviates = group.mean - grand.mean)

group.df
#> # A tibble: 3 x 5
#>   treatment     n group.mean grand.mean group.deviates
#>   <chr>      <int>     <dbl>     <dbl>          <dbl>
#> 1 control       8     -0.3088    -0.7127        0.4040
#> 2 eyes          7     -1.551     -0.7127       -0.8387
#> 3 knee          7     -0.3357    -0.7127        0.3770

```

Having calculated the group deviates, we calculate the group sum of squares and related quantities:

```

group.table <-
  group.df %>%
  summarize(SS = sum(n * group.deviates**2),
            k = n(),
            df = k-1,
            MS = SS/df)

group.table
#> # A tibble: 1 x 4
#>   SS     k     df     MS
#>   <dbl> <int> <dbl> <dbl>
#> 1 7.224     3     2 3.612

```

### Error SS and MS

Next we turn to variation of the individual observations around the group means, which is the basis of the error sum of squares and mean square. Again we calculate this in two steps:

```

error.df <-
  circadian %>%
  group_by(treatment) %>%
  mutate(group.mean = mean(shift),
        error.deviates = shift - group.mean) %>%
  summarize(SS = sum(error.deviates**2),
            n = n())
  
error.df
#> # A tibble: 3 x 3
#>   treatment     SS     n
#>   <chr>      <dbl> <int>
#> 1 control     2.670     8
#> 2 eyes        2.993     7
#> 3 knee        3.752     7

```

Now we calculate the error sum of squares and related quantities:

```

error.table <-
  error.df %>%
  summarize(SS = sum(SS),
            k = n(),
            N = sum(n),
            df = N - k,
            MS = SS/df)

error.table

```

```
#> # A tibble: 1 x 5
#>   SS      k     N    df      MS
#>   <dbl> <int> <int> <int>  <dbl>
#> 1 9.415     3     22     19  0.4955
```

### Calculating the F-statistic and $R^2$

Having calculated our estimates of between group variance and within group variance ( $MS_{\text{group}}$  and  $\text{MS}_{\text{error}}$ ) we're now ready to calculate the  $F$  test statistic.

```
F.stat <- group.table$MS/error.table$MS
F.stat
#> [1] 7.289449
```

The variation “explained” by the group is:

```
R2 <- group.table$SS/total.table$SS
R2
#> [1] 0.4341684
```

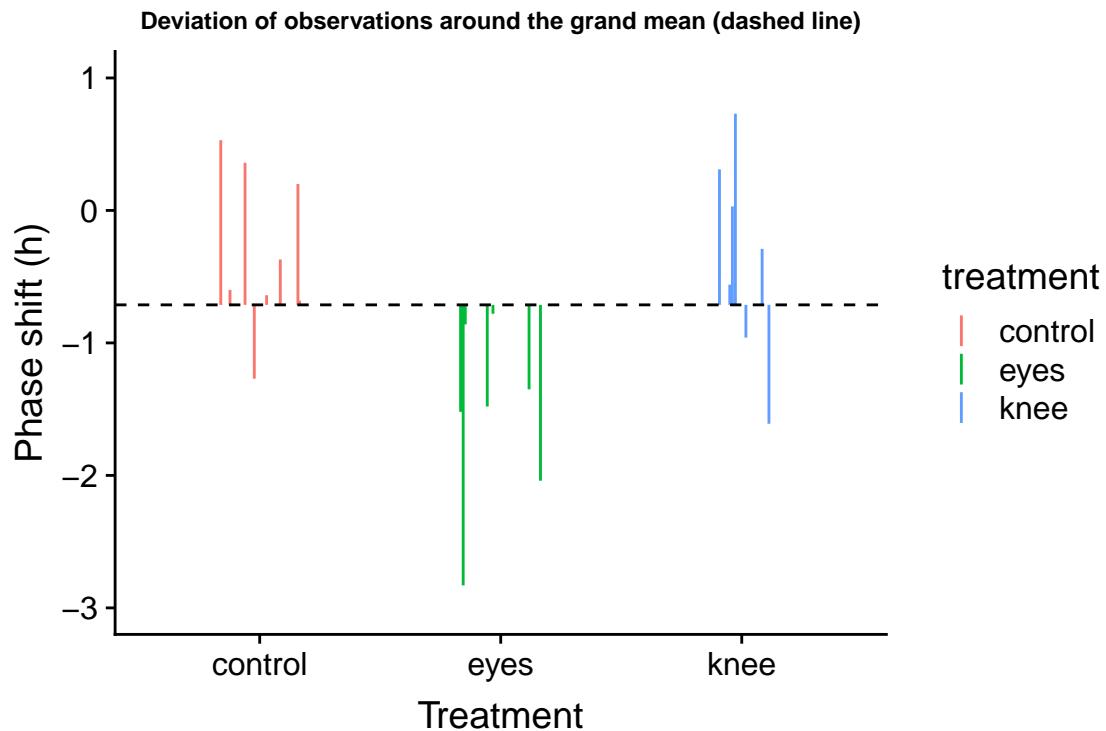
Thus about 43% of the total sum of squared deviation among subjects, with respect to the phase shift variable, is explained by differences in light treatment.

## 20.11 Visualizing the partitioning of sum-of-squares

### Total SS

```
total.plot <-
  circadian %>%
  ggplot(aes(x=treatment, y=shift,
             color=treatment, group=row.names(circadian))) +
  geom_linerange(aes(ymin = grand.mean, ymax = shift), position = pd) +
  geom_hline(yintercept = grand.mean, linetype='dashed') +
  ylim(-3,1) +
  labs(x = "Treatment", y = "Phase shift (h)",
       title = "Deviation of observations around the grand mean (dashed line)") +
  theme(plot.title = element_text(size=9))

total.plot
```

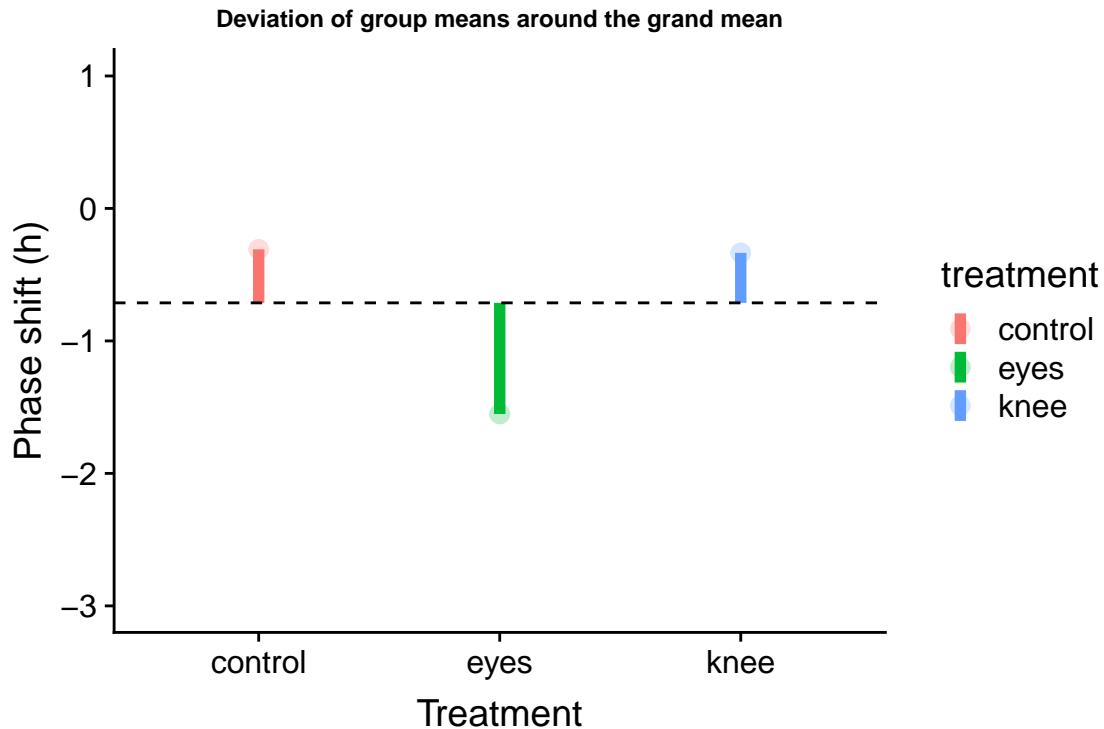


### Group SS

Let's visualize the difference of the group means from the grand mean:

```
group.plot <-
  group.df %>%
    ggplot(aes(x = treatment, y = group.mean, color=treatment)) +
      geom_linerange(aes(ymin = grand.mean, ymax = group.mean), size=2) +
      geom_point(size = 3, alpha = 0.25) +
      geom_hline(yintercept = grand.mean, linetype='dashed') +
      ylim(-3,1) +
      labs(x = "Treatment", y = "Phase shift (h)",
           title = "Deviation of group means around the grand mean") +
      theme(plot.title = element_text(size=9))

group.plot
```

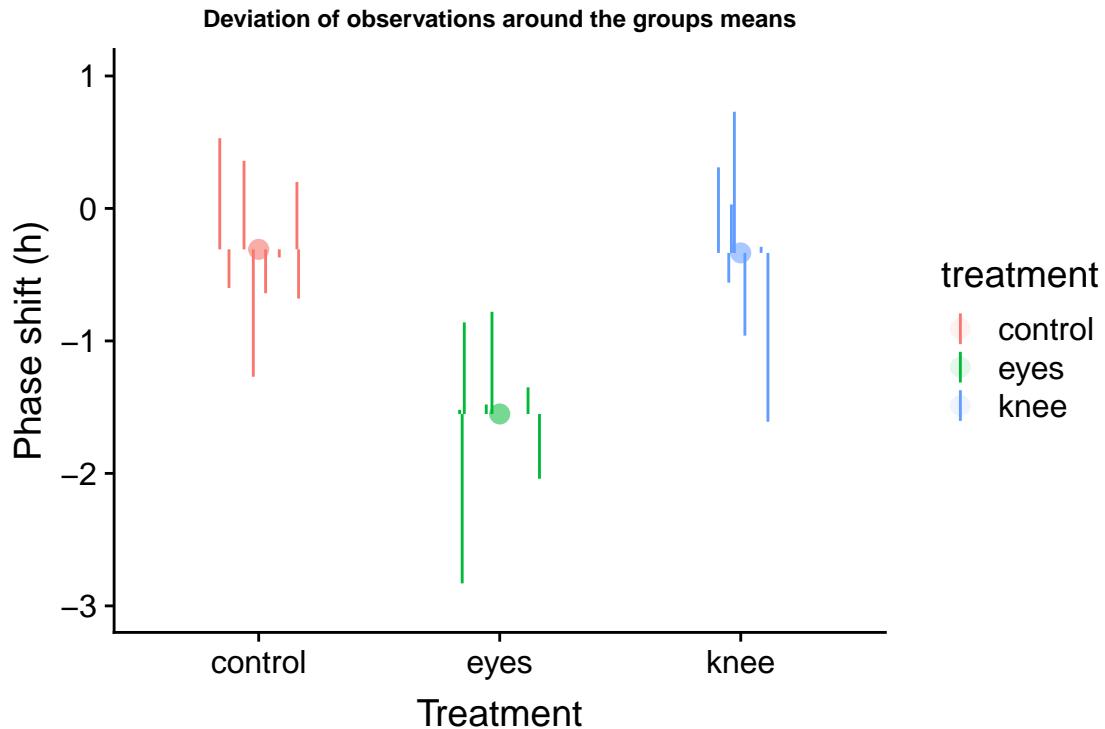


### Error SS

We can visualize the individual deviates around the group means as so:

```
error.plot <-
  circadian %>%
  group_by(treatment) %>%
  mutate(group.mean = mean(shift)) %>%
  ggplot(aes(x = treatment, y = shift, color = treatment)) +
  geom_point(aes(y = group.mean), size=3, alpha=0.1) +
  geom_linerange(aes(ymin = group.mean, ymax = shift), position = pd) +
  ylim(-3,1) +
  labs(x = "Treatment", y = "Phase shift (h)",
       title = "Deviation of observations around the groups means") +
  theme(plot.title = element_text(size=9))

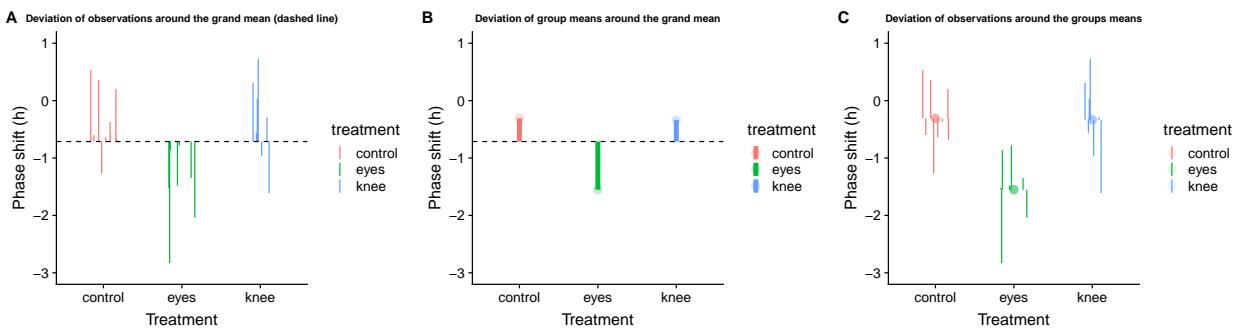
error.plot
```



### Combined visualization

We can combine our three plots created above into a single figure using `cowplot:::plot_grid`:

```
combined.plot <- plot_grid(total.plot, group.plot, error.plot,
                           labels = c("A", "B", "C"), nrow = 1)
combined.plot
```



## 20.12 Which pairs of group means are different?

If an ANOVA indicates that at least one of the group means is different than the others, the next question is usually “which pairs are different?”. There are slightly different tests for what are called “planned” versus “unplanned” comparisons. Your textbook discusses the differences between these two types

Here we focus on a common test for unplanned comparisons, called the Tukey Honest Significant Differences test (referred to as the Tukey-Kramer test in your textbook). The Tukey HSD test controls for the “family-wise error rate”, meaning it tries to keep the overall false positive (Type I error) rate at a specified value.

### 20.12.1 Tukey-Kramer test

The function `TukeyHSD` implements the Tukey-Kramer test. The input to `TukeyHSD` is the fit from `aov`:

```
TukeyHSD(circadian.aov)
#> Tukey multiple comparisons of means
#> 95% family-wise confidence level
#>
#> Fit: aov(formula = shift ~ treatment, data = circadian)
#>
#> $treatment
#>          diff      lwr       upr     p adj
#> eyes-control -1.24267857 -2.1682364 -0.3171207 0.0078656
#> knee-control -0.02696429 -0.9525222  0.8985936 0.9969851
#> knee-eyes     1.21571429  0.2598022  2.1716263 0.0116776
```

Here again, the `broom::tidy` function comes in handy:

```
tidy(TukeyHSD(circadian.aov))
#> # A tibble: 3 x 6
#>   term      comparison estimate conf.low conf.high adj.p.value
#>   <chr>    <chr>        <dbl>    <dbl>    <dbl>    <dbl>
#> 1 treatment eyes-control -1.243    -2.168    -0.3171   0.007866
#> 2 treatment knee-control -0.02696   -0.9525   0.8986   0.9970
#> 3 treatment knee-eyes     1.216    0.2598   2.172    0.01168
```

The Tukey HSD test by default give us 95% confidence intervals for the differences in means between each pair of groups, and an associated P-value for the null hypothesis of equal means between pairs. Interpreting the results above, we see that we fail to reject the null hypothesis of equal means for the knee and control treatment groups (i.e. we have no statistical support to conclude they are different). However, we reject the null hypothesis for equality of means between control and eye treatments and between knee and eye treatments. We have evidence that light treatments applied to the eye cause a mean negative shift in the phase of melatonin production relative to control and knee treatment groups.

## 20.13 Repeatability

- Nearly every measure of a continuous measurement or assay we apply in biology (and other sciences) has associated with it some **measurement error**.
- Measurement error is usually not “biological variation” of interest, but rather “technical variation” associated with our ability to measure quantities of interest precisely
  - **Example:** Have three people independently measure the length of a human femur to the nearest millimeter. Unless the values are aggressively rounded, there is a high likelihood that you’ll get three different values.

### Estimating Repeatability using ANOVA

ANOVA can be used to estimate the **Repeatability** of a measure, which provides a way to quantify how much of the variance we observe between individuals is due to measurement error.

#### Estimating Repeatability: Experimental steps

- Repeatedly, but independently, measure the same variable in the same individual or other unit of observation

- Carry out the same repeated measurements across individuals

### Estimating Repeatability: Statistical steps

- Calculate  $MS_{\text{groups}}$  and  $MS_{\text{error}}$  where individuals are the grouping unit.
- Estimate the variance among groups,  $s_A^2$  as:

$$s_A^2 = \frac{MS_{\text{groups}} - MS_{\text{error}}}{n}$$

where  $n$  is the number of replicate measures per individual.

- Estimate the repeatability as:

$$\text{Repeatability} = \frac{s_A^2}{s_A^2 + MS_{\text{error}}}$$

- $0 \leq \text{Repeatability} \leq 1$ ; values near zero indicate nearly all variance is due to measurement error, values near one indicate small fraction of variance due to measurement error

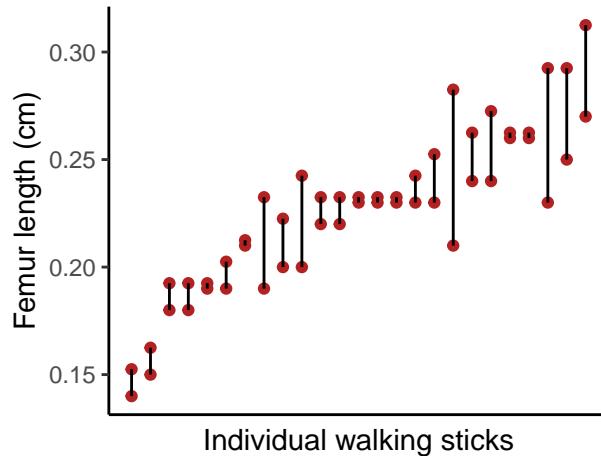
### Repeatability: Walking stick example

Nosil and Crespi measured various morphological features of walking stick insects from digital photographs. For each specimen they took two independent photographs and measured femur length (in cm) on each photograph.

```
walking.sticks <- read_csv("https://github.com/bio304-class/bio304-course-notes/raw/master/datasets/ABD-1000-walking-sticks.csv")

by.specimen <-
  walking.sticks %>%
  group_by(specimen) %>%
  summarize(min.fl = min(femurLength), max.fl = max(femurLength), avg.fl = 0.5*(min.fl + max.fl)) %>%
  arrange(avg.fl) %>%
  ungroup() %>%
  mutate(rank.fl = rank(avg.fl, ties.method = "first"))

ggplot(by.specimen, aes(x = rank.fl)) +
  geom_point(aes(y = min.fl), color="firebrick") +
  geom_point(aes(y = max.fl + 0.0025), color="firebrick") +
  geom_linerange(aes(ymin = min.fl, ymax = max.fl + 0.0025)) +
  labs(x = "Individual walking sticks", y = "Femur length (cm)") +
  theme_classic() +
  theme(axis.text.x=element_blank(),
        axis.ticks.x=element_blank())
```



First we carry out the ANOVA in the standard way, using the specimen variable as the grouping variable:

```
sticks.aov <- aov(femurLength ~ as.factor(specimen),
                     data = walking.sticks)
sticks.table <- tidy(sticks.aov)
sticks.table
#> # A tibble: 2 x 6
#>   term            df    sumsq   meansq statistic   p.value
#>   <chr>          <dbl>   <dbl>     <dbl>      <dbl>     <dbl>
#> 1 as.factor(specimen)  24 0.05913  0.002464     6.921  0.000004077
#> 2 Residuals        25 0.008900 0.0003560     NA       NA
```

From the ANOVA table, use  $MS_{\text{group}}$  and  $MS_{\text{error}}$  to calculate  $s_A$ :

```
repeatability.table <- data_frame(
  MS.groups = sticks.table$meansq[1],
  MS.error = sticks.table$meansq[2],
  sA = (MS.groups - MS.error)/2,
  repeatability = sA/(sA + MS.error))

repeatability.table
#> # A tibble: 1 x 4
#>   MS.groups  MS.error      sA  repeatability
#>   <dbl>      <dbl>    <dbl>        <dbl>
#> 1 0.002464 0.0003560 0.001054        0.7475
```

From the table we calculated:

- $s_A^2 = 0.0010539$
- Repeatability = 0.7475028

A repeatability of 0.75 indicates that we would estimate that  $\sim 75\%$  of the total variance in femur length measurements in our population of walking stick insects is the result of true differences in femur length between individuals, while roughly 25% of the variance is attributable to measurement error.



# Chapter 21

## Violations of Assumptions

Both t-tests and ANOVA make assumptions about the variable of interest. Namely, both assume:

- The variable of interest approximately normally distributed
- The variance of the variable is approximately the same in the groups being compared

When our data violate the assumptions of standard statistical tests, we have several options:

- Ignore violations of the assumptions, especially with large sample sizes and modest violations
- Transform the data (e.g., log transformation)
- Nonparametric methods
- Permutation tests

### Libraries

```
library(tidyverse)
library(magrittr)
library(cowplot)
library(broom)
```

### 21.1 Graphical methods for detecting deviations from normality

As we have seen, histogram and density plot can highlight distributions that deviate from normality.

Look for:  
\* distinctly non-symmetric spread – long tails, data up against a lower or upper limit  
\* multiple modes  
\* Varying bin widths (histograms) and the degree of smoothing (density plots) can both be useful.

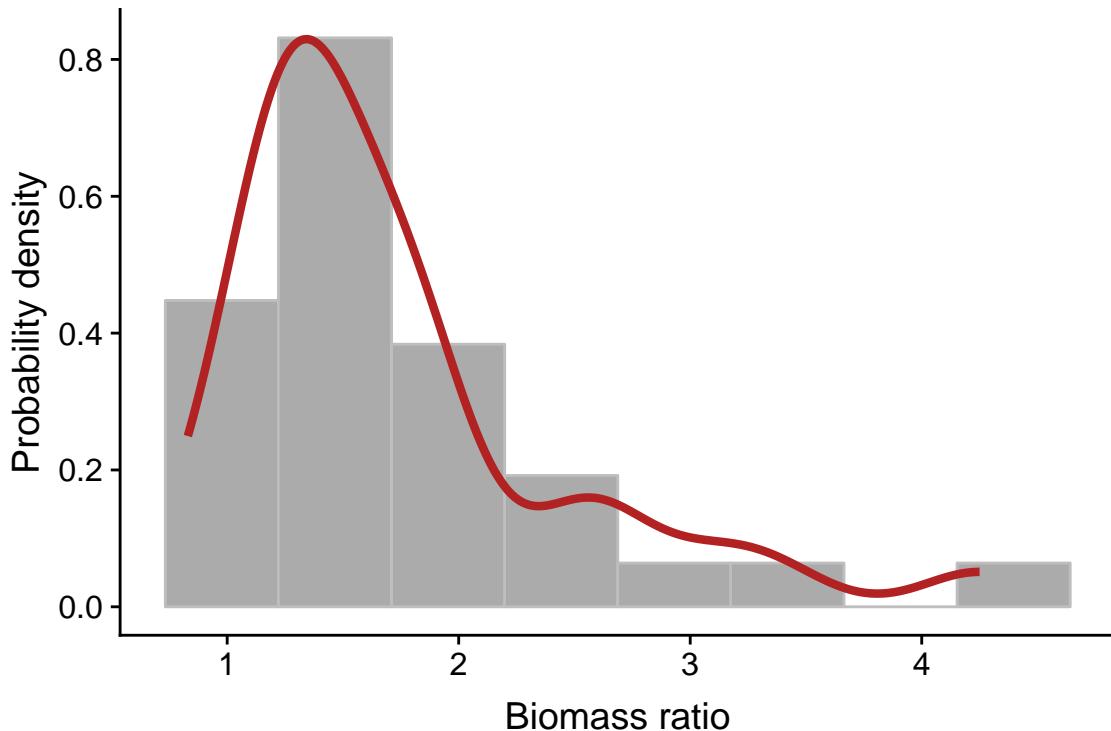
#### Example data: Marine reserves

Example 13.1 in your text book describes a study aimed at understanding whether marine reserves are effective at preserving marine life. The data provided are “biomass ratios” between reserves and similar unprotected sites nearby. A ratio of one means the reserve and the nearby unprotected site had equal biomass; values greater than one indicate greater biomass in the reserve while values less than one indicate greater biomass in the unprotected site.

```
library(tidyverse)
library(magrittr)
library(cowplot)

marine.reserves <- read_csv("https://github.com/bio304-class/bio304-course-notes/raw/master/datasets/ABR.csv")
```

```
ggplot(marine.reserves, aes(x = biomassRatio, y = ..density..)) +
  geom_histogram(bins=8, color='gray', alpha=0.5) +
  stat_density(geom="line",
               adjust = 1.25, # adjust is in multiples of smoothing bandwidth
               color = 'firebrick', size=1.5) +
  labs(x = "Biomass ratio", y = "Probability density")
```

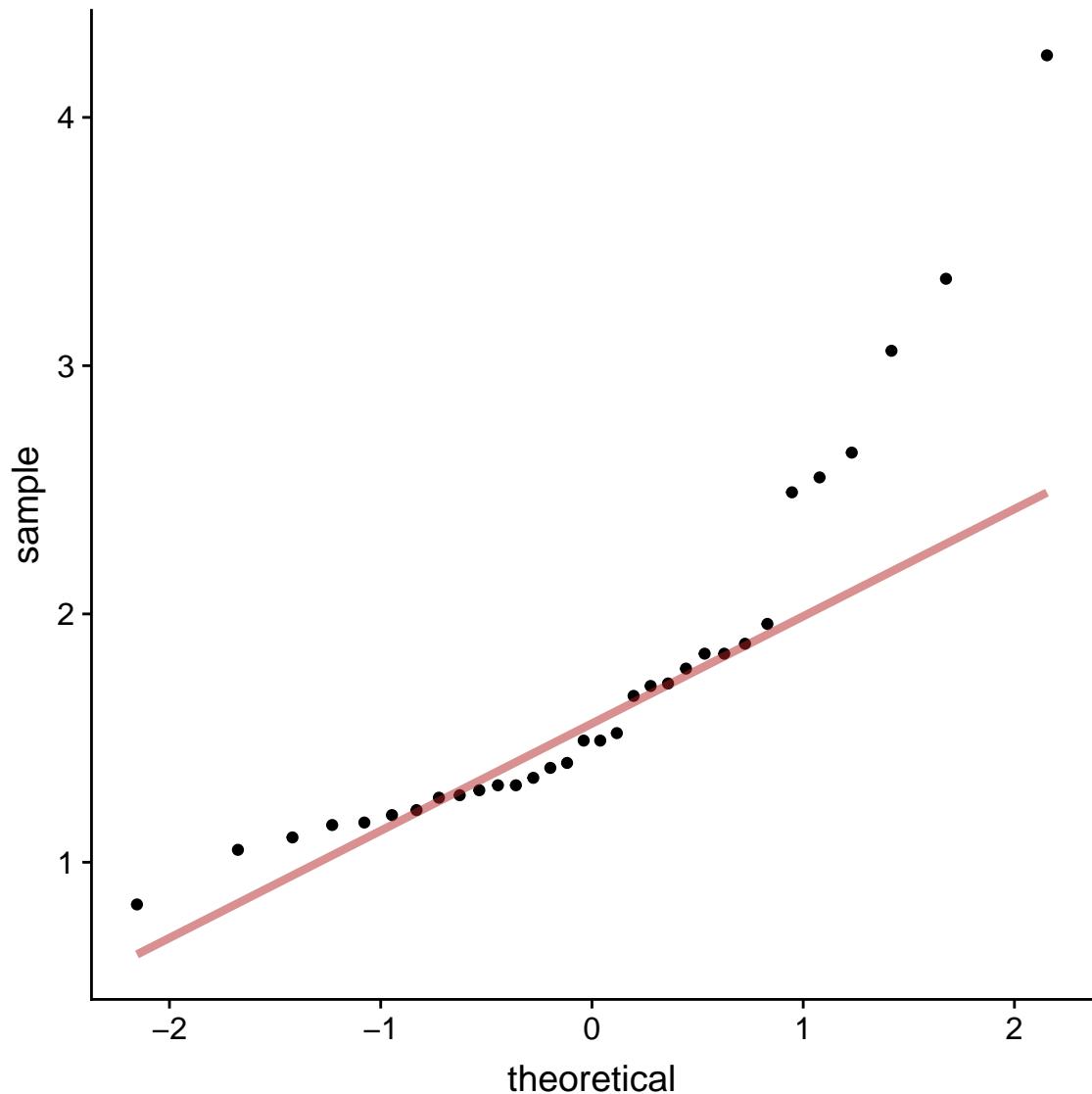


Both the histogram and the density plot suggest the biomass ratio distribution is strongly right skewed.

## Normal probability plots

Normal probability plots are another visualization tools we saw previously for judging deviations from normality. If the observed data are approximately normally distributed, the scatter of points should fall roughly along a straight line with slope equal to the standard deviation of the data and intercept equal to the mean of the data.

```
ggplot(marine.reserves, aes(sample = biomassRatio)) +
  geom_qq() +
  geom_qq_line(color = 'firebrick', size = 1.5, alpha = 0.5)
```



## 21.2 Formal test of normality

There are number of formal tests of normality, with the “Shapiro-Wilk” test being among the most common. The Shapiro-Wilk test is essentially a formalization of the procedure we do by eye when judging a QQ-plot.

The null hypothesis for the Shapiro-Wilk test is that the data are drawn from a normal distribution. The test statistic for the Shapiro-Wilks tests is called  $W$ , and it measures the ratio of sumed deviations from the mean for a theoretical normal relative to the observed data. Small values of  $W$  are evidence of departure from normality. As in all other cases we've examined, the observed  $W$  is compared to the expected sampling distribution of  $W$  under the null hypothesis to estimate a P-value. Small P-values reject the null hypothesis of normality.

```
shapiro.test(marine.reserves$biomassRatio)
#>
#> Shapiro-Wilk normality test
#>
```

```
#> data: marine.reserves$biomassRatio
#> W = 0.81751, p-value = 8.851e-05
```

## 21.3 When to ignore violations of assumptions

Estimation and testing of means tends to be robust to violation of assumptions of normality and homogeneity of variance. This robustness is a consequence of the Central Limit Theorem.

However, if “two groups are being compared and both differ from normality in different ways, then even subtle deviations from normality can cause errors in the analysis (even with fairly large sample sizes)” (W&S Ch 13).

## 21.4 Data transformations

A common approach to dealing with non-normal variables is to apply a mathematical function that transforms the data to more nearly normal distribution. Analyses and inferences are then done on the transformed data.

**Log transformation:** This widely-used transformation can only be applied to positive numbers, since  $\log(x)$  is undefined when  $x$  is negative or zero. However, addition of a small positive constant can be used to ensure that all data are positive. For example, if a vector  $x$  includes negative numbers, then  $x + \text{abs}(\min(x)) + 1$  results in positive numbers that can be log transformed.

**Arcsine square root transformation:**  $p' = \arcsin[\sqrt{p}]$  is often used when the data are proportions. If the original data are percentages, first divide by 100 to convert to proportions.

**Square root transformation:**  $Y' = \sqrt{Y + K}$ , where  $K$  is a constant to ensure that no data points are negative. Results from the square root transformation are often very similar to log transformation.

**Square transformation:**  $Y' = Y^2$ . This may be helpful when the data are skewed left. .

**Antilog transformation:**  $Y' = e^Y$  may be helpful if the square transformation does not resolve the problem. Only usable when all data points have the same sign. If all data points are less than zero, take the absolute value before applying the antilog transformation.

**Reciprocal transformation:**  $Y' = \frac{1}{Y}$  When data are right skewed, this transformation may be helpful. If all data points are less than zero, take the absolute value before applying the reciprocal transformation.

### Example: log transformation to deal with skew

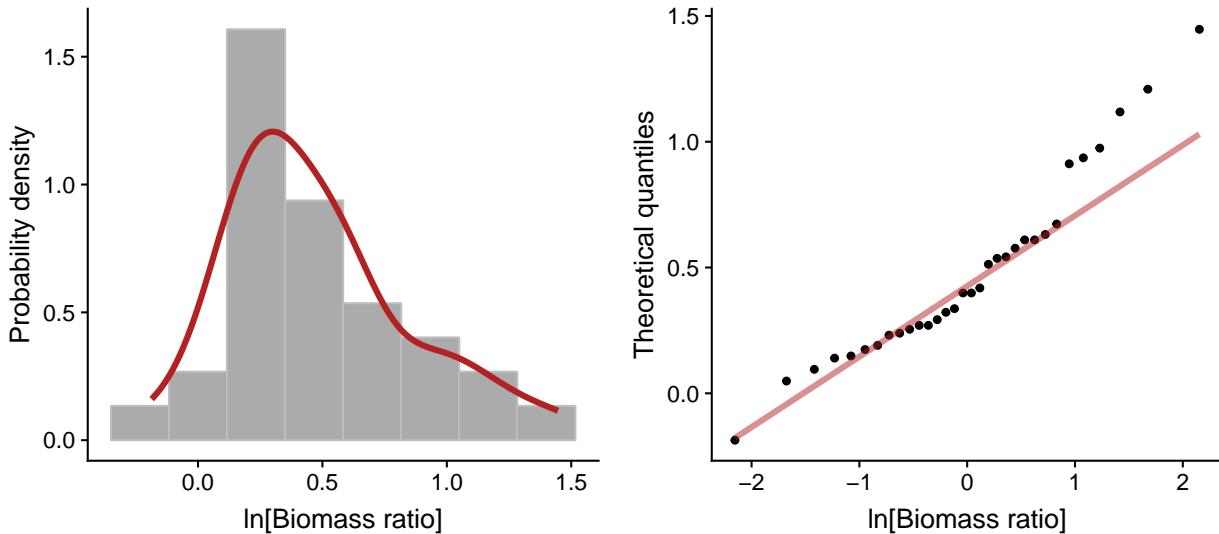
The biomass ratio variable in the marine reserves data is strongly right skewed. Let’s see how well a log transformation does in terms of normalizing this data:

```
ln.biomass.hist <-
  ggplot(marine.reserves, aes(x = log(biomassRatio), y = ..density..)) +
  geom_histogram(bins=8, color='gray', alpha=0.5) +
  stat_density(geom="line",
               adjust = 1.25,    # adjust is in multiples of smoothing bandwidth
               color = 'firebrick', size=1.5) +
  labs(x = "ln[Biomass ratio]", y = "Probability density")

ln.biomass.qq <-
  ggplot(marine.reserves, aes(sample = log(biomassRatio))) +
  geom_qq() +
```

```
geom_qq_line(color = 'firebrick', size = 1.5, alpha = 0.5) +
  labs(x = "ln[Biomass ratio]", y = "Theoretical quantiles")

plot_grid(ln.biomass.hist, ln.biomass.qq)
```



The log-transformed biomass ratio data is still a bit right skewed though not nearly as much as before. Applying the Shapiro-Wilk test to this transformed data, we fail to reject the null hypothesis of normality at  $\alpha = 0.05$ .

```
shapiro.test(log(marine.reserves$biomassRatio))
#>
#> Shapiro-Wilk normality test
#>
#> data: log(marine.reserves$biomassRatio)
#> W = 0.93795, p-value = 0.06551
```

### 21.4.1 Confidence intervals and data transformations

Sometimes we need to transform our data before analysis, and then we want to know the confidence intervals on the original scale. In such cases, we back-transform the upper and lower CIs to the original scale of measurement.

Let's calculate 95% CIs for the log-transformed biomass ratio data:

```
log.biomass.CI <-
  marine.reserves %>%
  mutate(log.biomassRatio = log(biomassRatio)) %>%
  summarize(mean = mean(log.biomassRatio),
            sd = sd(log.biomassRatio),
            n = n(),
            SE = sd/sqrt(n),
            CI.low.log = mean - abs(qt(0.025, df = n-1)) * SE,
            CI.high.log = mean + abs(qt(0.025, df = n-1)) * SE) %>%
  select(CI.low.log, CI.high.log)

log.biomass.CI
```

```
#> # A tibble: 1 x 2
#>   CI.low.log CI.high.log
#>   <dbl>       <dbl>
#> 1     0.3470     0.6112
```

This analysis finds the 95% CI around the mean of the log transformed data ranges is 0.347018, 0.6112365.

To understand the biological implications of this analysis, we back-transform the log data results to the original scale, computing the *antilog* by using function `exp`. After this, the back-transformed CI is now on the scale of  $e^{\log X} = X$ .

Letting  $\mu' = \text{mean}(\log.X)$ , we have  $0.347 < \mu' < 0.611$  on the log scale.

On the original scale:

```
log.biomass.CI %>%
  # back-transform CI to original scale
  mutate(CI.low.original = exp(CI.low.log),
         CI.high.original = exp(CI.high.log))
#> # A tibble: 1 x 4
#>   CI.low.log CI.high.log CI.low.original CI.high.original
#>   <dbl>       <dbl>       <dbl>       <dbl>
#> 1     0.3470     0.6112     1.415      1.843
```

Hence,  $1.41 < \text{geometric mean} < 1.84$ .

The *geometric mean* equals  $\sqrt[n]{x_1 x_2 \cdots x_n}$ , computed by multiplying all these  $n$  numbers together, then taking the  $n^{\text{th}}$  root.

Biologically, “this 95% confidence interval indicates that marine reserves have 1.41 to 1.84 times more biomass on average than the control sites do” (W&S 13.3)

## 21.5 Nonparametric tests

Non-parametric tests are ones that make no (or fewer) assumptions about the particular distributions from which the data of interest are drawn. Nonparametric tests are helpful for non-normal data distributions, especially with outliers. Typically, these tests use the ranks of the data points rather than the actual values of the data. Because they are not using all available data, nonparametric tests have lower statistical power than parametric tests. Nonparametric approaches still assume that the data are a random sample from the population.

We present here two common non-parametric alternatives to t-tests and ANOVA.

### 21.5.1 Mann-Whitney U-test: non-parametric alternative to t-test

When the normality assumption is not met, and none of the standard mathematical transformations are suitable (or desirable), the **Mann-Whitney U-test** can be used in place of a two-sample *t*-test to compare the *distribution* of two groups. The Mann-Whitney U-test compares the data distribution of two groups, without requiring the normality assumptions of the two-sample *t*-test. The null hypothesis of this test is that the within-group data *distributions* are the same. Therefore, it is possible to reject the null hypothesis because the distributions differ, *even if the means are the same*. The Mann-Whitney U-test is sensitive to unequal variances or different patterns of skew. Therefore, this test should be used to compare distributions, not to compare means.

Confusingly, the Mann-Whitney U-test is also called the Mann-Whitney-Wilcoxon test. Do not confuse this with another test called the Wilcoxon signed-rank test.

Your textbook details the calculations involved, here we simply demonstrate the appropriate R function.

### Example data: Sexual cannibalism in crickets

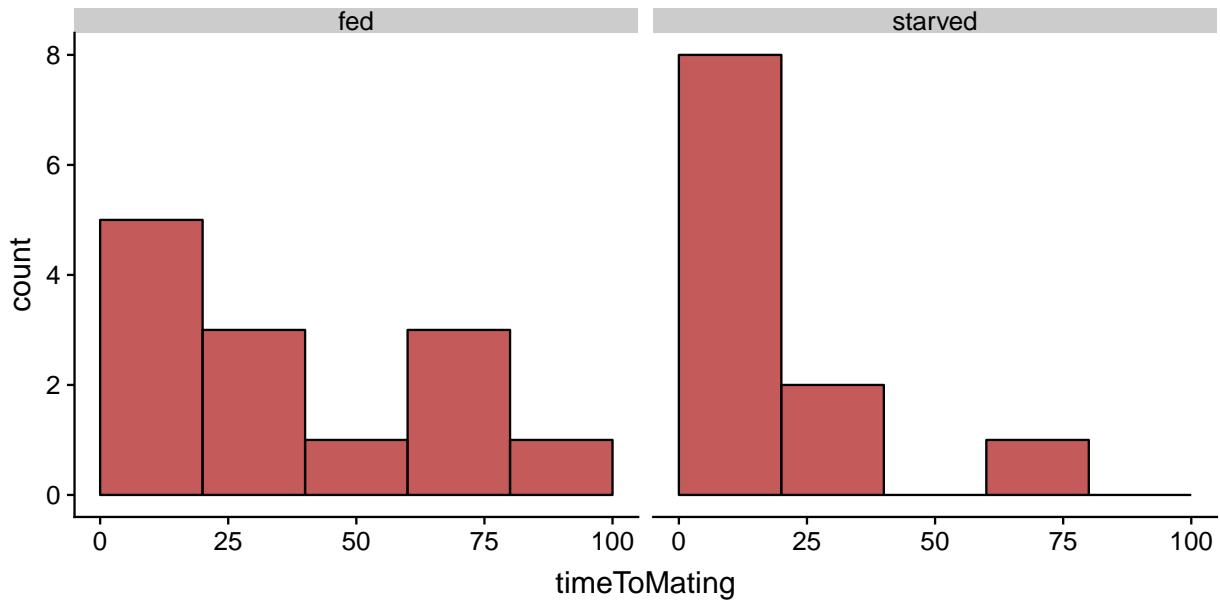
Example 13.5 in your textbook demonstrates the Mann-Whitney U-test with an application to a study of sexual cannibalism in sage crickets. During mating, male sage crickets offer their fleshy hind wings to females to eat. Johnson et al. (1999) studied whether female sage crickets are more likely to mate if they are hungry. They measured the time to mating (in hours) for female crickets that had been starved or fed. The null and alternative hypotheses for this comparison are:

- $H_0$ : The distribution of time to mating is the same for starved and fed female crickets
- $H_A$ : The distribution of time to mating differs between starved and fed female crickets

Here's what the data look like:

```
crickets <- read_csv("https://github.com/bio304-class/bio304-course-notes/raw/master/datasets/ABD-cricket.csv")
#> Parsed with column specification:
#> cols(
#>   feedingStatus = col_character(),
#>   timeToMating = col_double()
#> )
```

```
ggplot(crickets, aes(x = timeToMating)) +
  geom_histogram(binwidth = 20, alpha = 0.75, fill = 'firebrick', color='black', center=10) +
  facet_wrap(~ feedingStatus, nrow=1)
```



As is apparent from the histograms, these data are decidedly non-normal. None of the standard mathematical transforms appear to work well in normalizing these data either.

To perform the Mann-Whitney U test, we use function `wilcox.test` (details here). This can take a data frame as input:

```
wilcox.test(timeToMating ~ feedingStatus,
            alternative = "two.sided", paired = FALSE, data = crickets)
#>
#> Wilcoxon rank sum test
```

```
#>
#> data: timeToMating by feedingStatus
#> W = 88, p-value = 0.3607
#> alternative hypothesis: true location shift is not equal to 0
```

The test statistic  $W$  is the equivalent to the  $U$  test statistic of the Mann Whitney U-test. In this case, our observed value of  $U$  ( $W$ ) is quite probable ( $p = 0.4$ ) under the null hypothesis of no difference in the distributions between the two groups, therefore we fail to reject the null hypothesis.

### 21.5.2 Kruskal-Wallis test: non-parametric alternative to ANOVA

When there are more than two groups to compare, the non-parametric alternative to ANOVA is the Kruskal-Wallis Test. Like the Mann-Whitney U-test, the Kruskal-Wallis test is based on ranks rather than observed values. While the Kruskal-Wallis test does not assume the data are normally distributed, it still assumes the distributions of the variable of interest have similar shapes across the groups. The test statistic for the Kruskal-Wallis test is designated  $H$ .

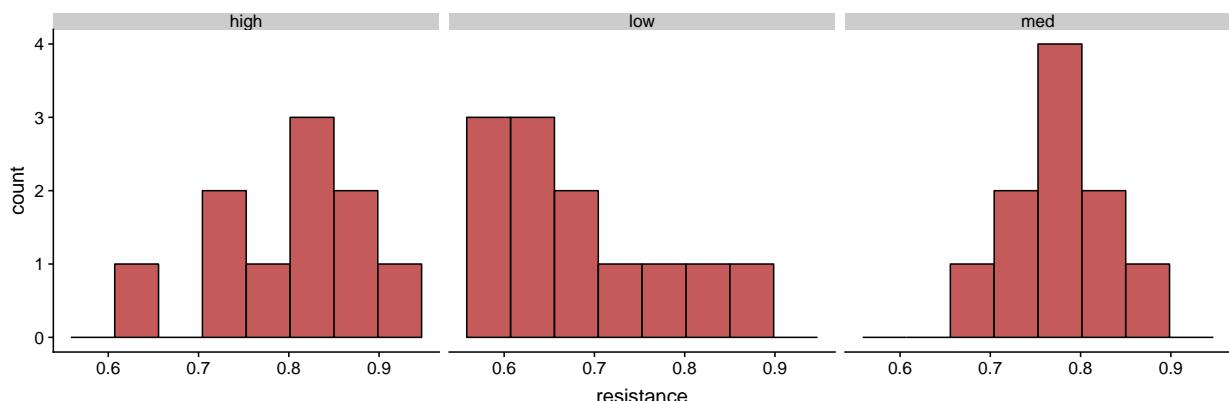
#### Example data: Daphnia resistance to cyanobacteria

This is from Assignment Problem 13.17 in Whitlock & Schlüter.

Daphnia is a freshwater planktonic crustacean. Hairson et al. (1999) were interested in local adaptation of Daphnia populations in Lake Constance (bordering Germany, Austria, and Switzerland). Cyanobacteria, is a toxic food type that has increased in density in Lake Constance since the 1960s in response to increased nutrients. The investigators collected Daphnia eggs from sediments laid down during years of low, medium, and high cyanobacteria density and measured resistance to cyanobacteria in each group. Visual inspection of these data suggest they violate the normality assumptions of standard ANOVA.

```
daphnia <- read_csv("https://github.com/bio304-class/bio304-course-notes/raw/master/datasets/ABD-daphnia.csv")
#> Parsed with column specification:
#> cols(
#>   cyandensity = col_character(),
#>   resistance = col_double()
#> )

ggplot(daphnia, aes(x = resistance)) +
  geom_histogram(bins=8, alpha = 0.75, fill = 'firebrick', color='black') +
  facet_wrap(~ cyandensity)
```



To carry out the Kruskal-Wallis test, and summarize the output in a data frame using `broom::tidy` we do:

```
kw.results <- kruskal.test(resistance ~ as.factor(cyandensity), daphnia)
tidy(kw.results)
#> # A tibble: 1 x 4
#>   statistic p.value parameter method
#>       <dbl>    <dbl>      <int> <chr>
#> 1     8.200  0.01658      2 Kruskal-Wallis rank sum test
```

Based on a the estimated P-value, we have evidence upon which to reject the null hypothesis of equal means at a Type I error rate of  $\alpha = 0.5$ . However some caution is warranted as the shape of the distributions appear to be somewhat different among the groups.

## Acknowledgements

Figures and examples from W&S Chap 13; lecture notes from T. Mitchell-Olds.



## Chapter 22

# Bivariate Linear Regression

Statistical models are quantitative statements about how we think variables are related to each other.

Linear models are among the simplest statistical models. In a linear model relating two variables  $X$  and  $Y$ , the general form of the model can be stated as “I assume that  $Y$  can be expressed as a linear function of  $X$ ”. The process of *model fitting* is then the task of finding the coefficients (parameters) of the linear model which best fit the observed data.

Linear functions are those whose graphs are straight lines. A linear function of a variable  $X$  is usually written as:

$$\hat{Y} = f(X) = a + bX$$

where  $a$  and  $b$  are constants. In geometric terms  $b$  is the *slope of the line* and  $a$  is the value of the function when  $X$  is zero (usually referred to as the “Y-intercept”). The slope tells you how much  $Y$  changes per unit change of  $X$ .

There are infinitely many such linear functions of  $X$  we could define. Which linear function provides the best fit given our observed values of  $X$  and  $Y$ ?

### 22.1 Regression terminology

- **Predictors, explanatory, or independent variable** – the variables from which we want to make our prediction.
- **Outcomes, dependent, or response variable** – the variable we are trying to predict in our regression.

### 22.2 The optimality criterion for least-squares regression

In order to fit a model to data, we have to specify some criterion for judging how well alternate models perform.

In linear regression, the optimality criterion can be expressed as “Find the linear function,  $f(X)$ , that minimizes the following quantity:”

$$\sum(y_i - f(x_i))^2$$

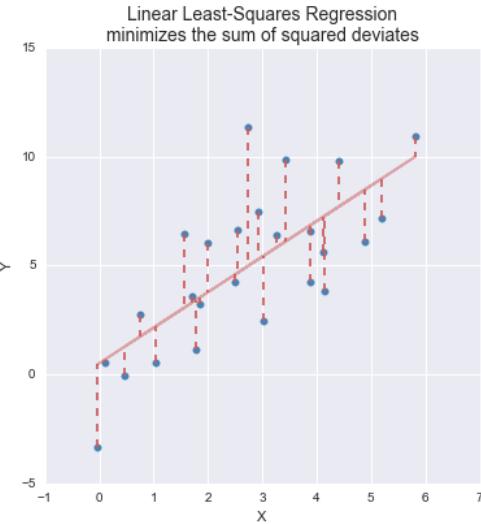


Figure 22.1: A graphical representation of the optimality criterion in bivariate least squares linear regression.

That is, our goal is to find the linear function of  $X$  that minimizes the squared deviations in the  $Y$  direction.

### 22.3 Solution for the least-squares criterion

With a little calculus and linear algebra one can show that the values of  $b$  (slope) and  $a$  (intercept) that minimize the sum of squared deviations described above are:

$$b = \frac{s_{xy}}{s_x^2} = r_{xy} \frac{s_y}{s_x} \quad (22.1)$$

$$(22.2)$$

$$a = \bar{Y} - b\bar{X} \quad (22.3)$$

where  $r_{xy}$  is the correlation coefficient between  $X$  and  $Y$ , and  $s_x$  and  $s_y$  are the standard deviations of  $X$  and  $Y$  respectively.

### 22.4 Libraries

```
library(tidyverse)
library(ggExtra) # a new library, provides ggMarginal plot (see below)
# install if you don't already have it
```

### 22.5 Illustrating linear regression with simulated data

To illustrate how regression works, we'll use a simulated data set where we specify the relationship between two variables,  $X$  and  $Y$ . Using a simulation is desirable because it allows us to know what the “true” underlying model that relates  $X$  and  $Y$  is, so we can evaluate how well we do in terms of recovering the model.

Let's generate two vectors representing the variable,  $X$  and  $Y$ , where  $Y$  is a function of  $X$  plus some independent noise. As specified below, the "true" model is  $Y = 1.5X + 1.0 + \epsilon_y$  where  $\epsilon_y$  is a noise term.

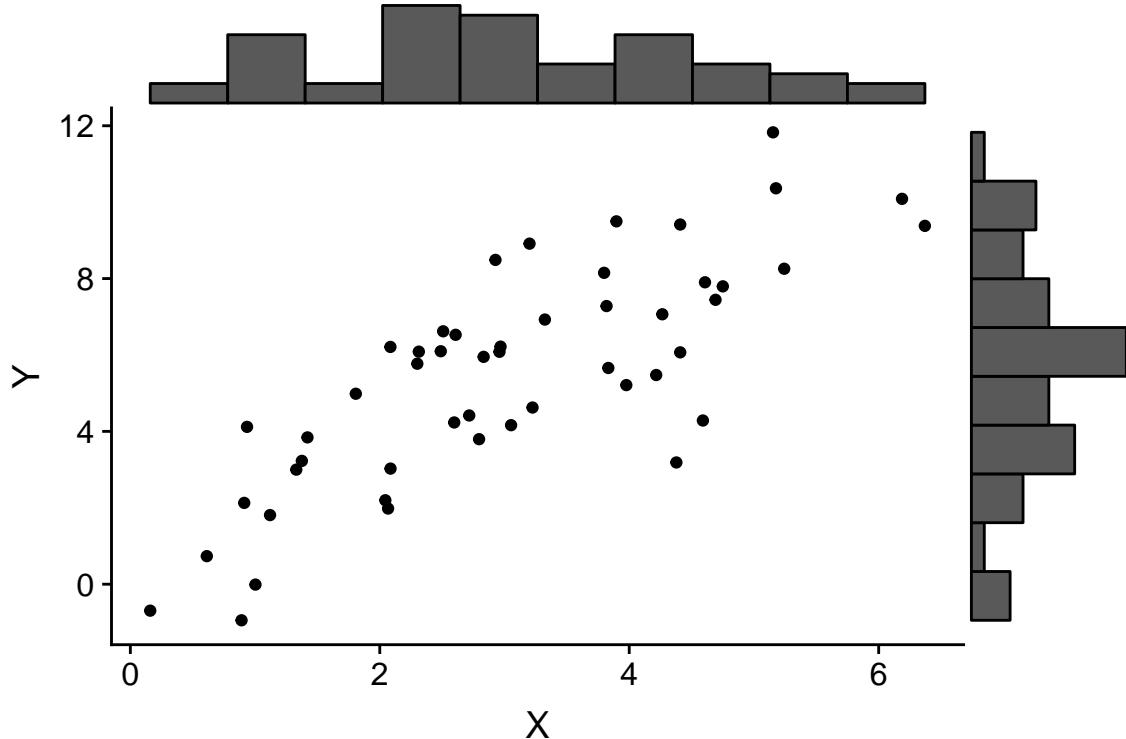
```
# this seeds our random number generator
# by setting a seed, we can make random number generation reproducible
set.seed(20160921)

npts <- 50
X <- seq(1, 5, length.out = npts) + rnorm(npts)
a <- 1.0
b <- 1.5
Y <- b*X + a + rnorm(npts, sd = 2) # Y = 1.5X + 1.0 + noise

df.xy <- data.frame(X = X, Y = Y)
```

Having generated some simulated data, let's visualize it.

```
p <- ggplot(df.xy, aes(x = X, y = Y)) + geom_point()
ggMarginal(p, type = "histogram", bins = 11)
```



## 22.6 Specifying Regression Models in R

As one would expect, R has a built-in function for fitting linear regression models. The function `lm()` can be used not only to carry out bivariate linear regression but a wide range of linear models, including multiple regression, analysis of variance, analysis of covariance, and others.

```
fit.xy <- lm(Y ~ X, df.xy)
```

The first argument to `lm` is an R "formula", the second argument is a data frame.

Recall that formulas are R's way of specifying models, though they find other uses as well (e.g. we saw the formula syntax when we introduced the `facet_wrap` and `facet_grid` functions from `ggplot`, and in the context of ANOVA). The general form of a formula in R is `response variable ~ explanatory variables`. In the code example above, we have only a single explanatory variable, and thus our response variable is Y and our explanatory variable is X.

The `lm` function returns a list with a number of different components. The ones of most interest to us are `fitted.values`, `coefficients`, `residuals`, and (see the `lm` documentation for full details.)

```
fit.xy
#>
#> Call:
#> lm(formula = Y ~ X, data = df.xy)
#>
#> Coefficients:
#> (Intercept)          X
#>     0.7688      1.5511
```

Calling `summary` on a fit model provides more detailed output:

```
summary(fit.xy)
#>
#> Call:
#> lm(formula = Y ~ X, data = df.xy)
#>
#> Residuals:
#>    Min     1Q Median     3Q    Max
#> -4.3723 -1.2391 -0.1677  1.4593  3.1808
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)  0.7688     0.5755   1.336   0.188
#> X            1.5511     0.1699   9.129 4.58e-12 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 1.772 on 48 degrees of freedom
#> Multiple R-squared:  0.6345, Adjusted R-squared:  0.6269
#> F-statistic: 83.34 on 1 and 48 DF,  p-value: 4.581e-12
```

As we saw in previous R functions for implementing statistical test, the model object is actually a list-like object with multiple fields:

```
typeof(fit.xy)
#> [1] "list"

names(fit.xy)
#> [1] "coefficients"   "residuals"       "effects"        "rank"
#> [5] "fitted.values"  "assign"         "qr"             "df.residual"
#> [9] "xlevels"        "call"          "terms"          "model"
```

### 22.6.1 Fitted values

The component `fitted.values` gives the predicted values of  $Y$  ( $\hat{Y}$  in the equations above) for each observed value of  $X$ . We can plot these predicted values of  $Y$ , as shown below. Notice how the predicted values all fall on a line (the regression line itself!).

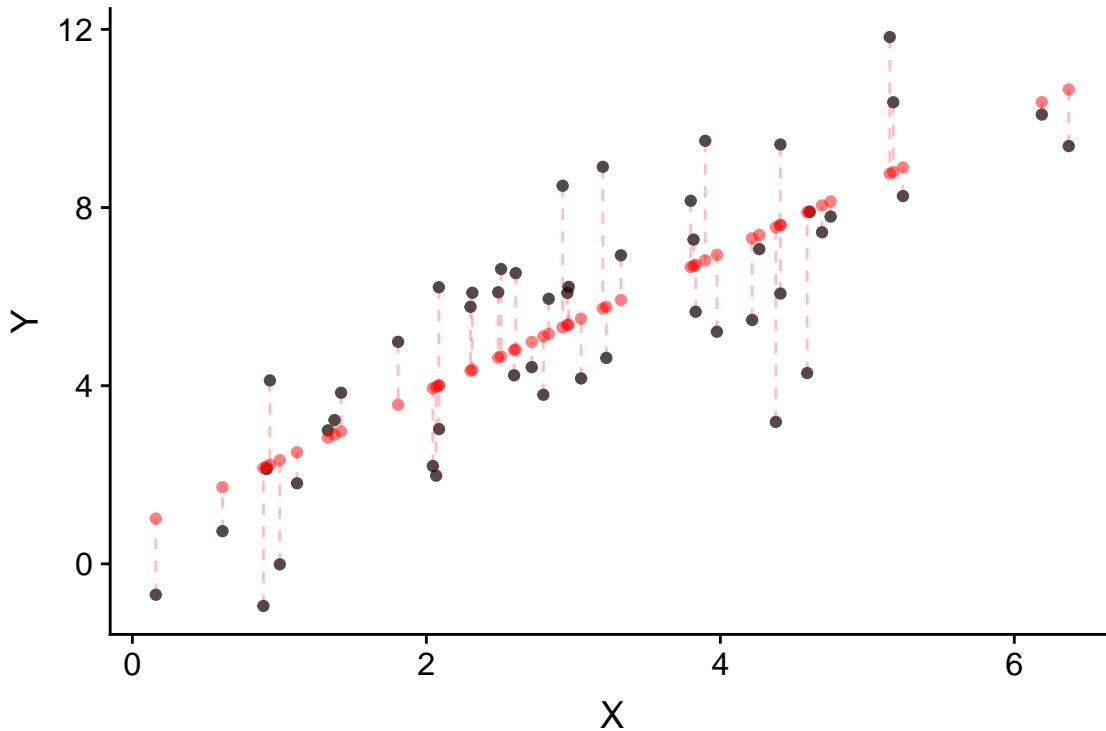


Figure 22.2: Observed (black) and predicted (red) values in a linear regression of  $Y$  on  $X$ . Dashed lines indicate the residuals from the regression.

```
ggplot(df.xy, aes(x = X, y = Y)) +
  geom_point(alpha=0.7) +                               # observed data
  geom_point(aes(x = X, y = fit.xy$fitted.values),   # predicted data
             color='red', alpha=0.5) +
  geom_segment(aes(xend = X, yend = fit.xy$fitted.values),
               color='red', linetype='dashed', alpha=0.25)
```

## 22.6.2 Getting the model coefficients

The `coefficients` components gives the value of the model parameters, namely the intercept and slope.

```
> fit.xy$coefficients
#> (Intercept)          X
#>  0.7687897  1.5510691
```

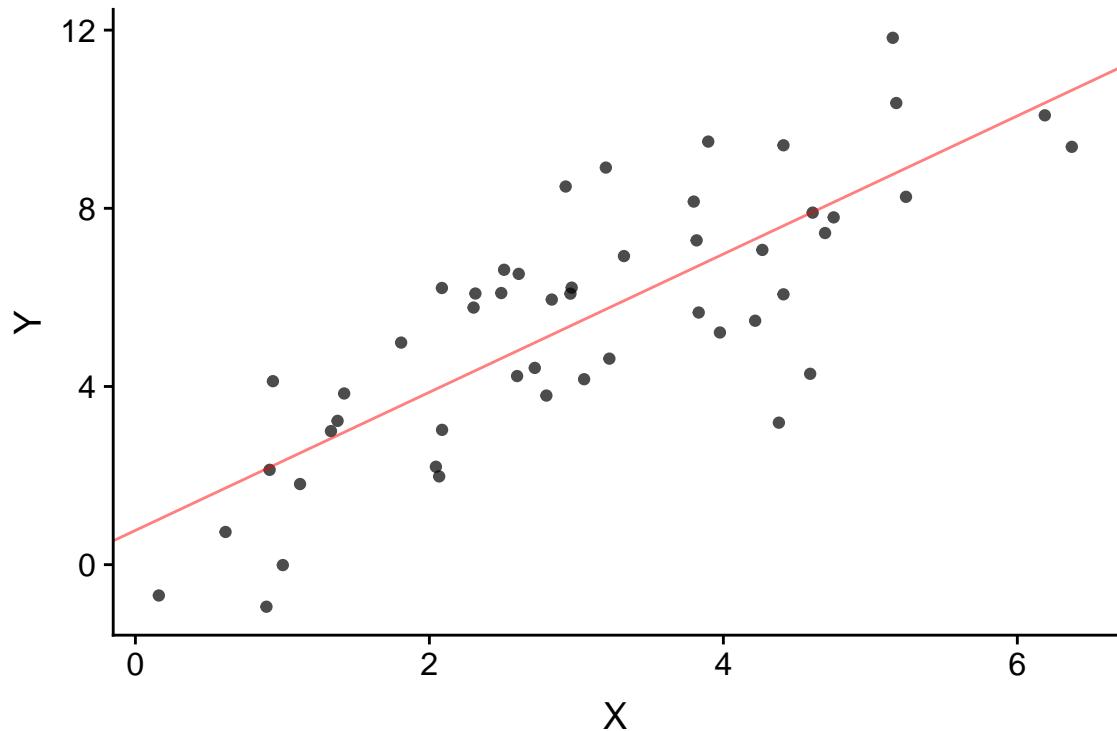
As shown above, the estimated slope is 1.5510691 and the estimated intercept is 0.7687897. The model estimated by our linear regression is thus  $\hat{Y} = 0.769 + 1.55X$ .

Recall that because this is a synthetic example, we know the “true” underlying model, which is  $Y = 1.5X + 1.0 + \epsilon_x$ . On the face of it, it appears our regression model is doing a decent job of estimating the true model.

With this information in hand we can draw the regression line as so:

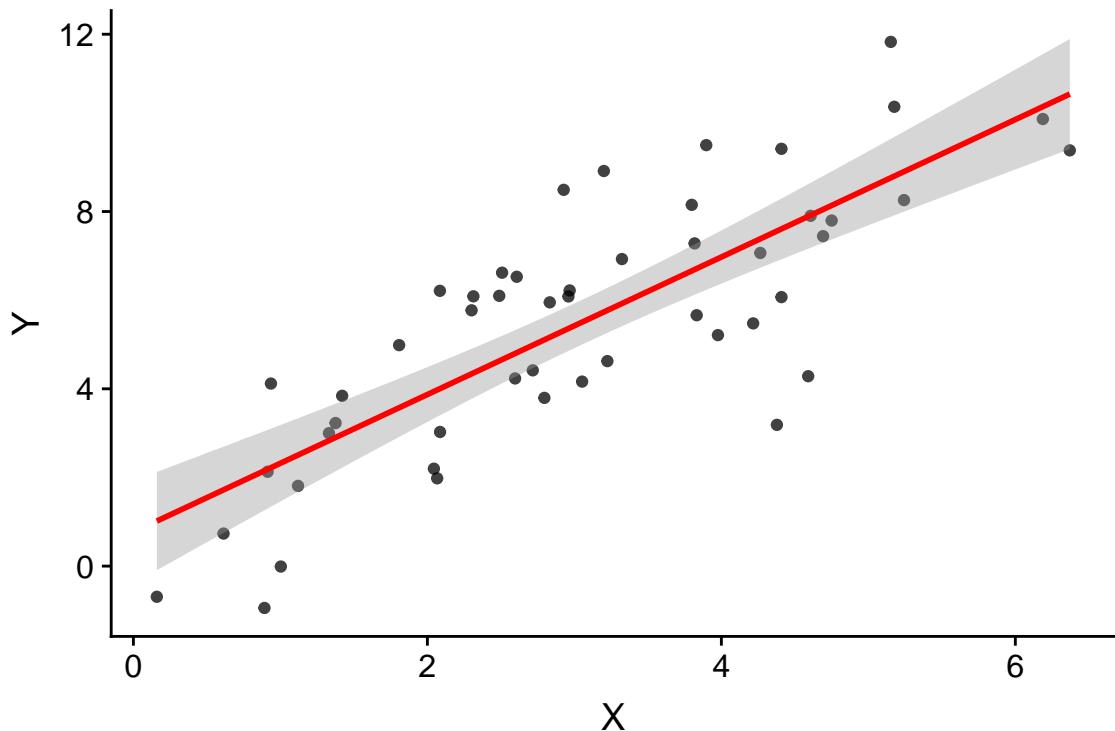
```
ggplot(df.xy, aes(x = X, y = Y)) +
  geom_point(alpha=0.7) +      # observed data
```

```
geom_abline(slope = fit.xy$coefficients[[2]],
            intercept = fit.xy$coefficients[[1]],
            color='red', alpha=0.5)
```



Since linear model fitting is a fairly common task, the ggplot library includes a geometric mapping, `geom_smooth`, that will fit a linear model for us and generate the corresponding regression plot.

```
ggplot(df.xy, aes(x = X, y = Y)) +
  geom_point(alpha = 0.75) +
  geom_smooth(method="lm", color = 'red')
```



By default, `geom_smooth` draws confidence intervals for the regression model (the shaded gray area around the regression line). Note that confidence intervals for a linear regression model are wider far away from the mean values of  $X$  and  $Y$ .

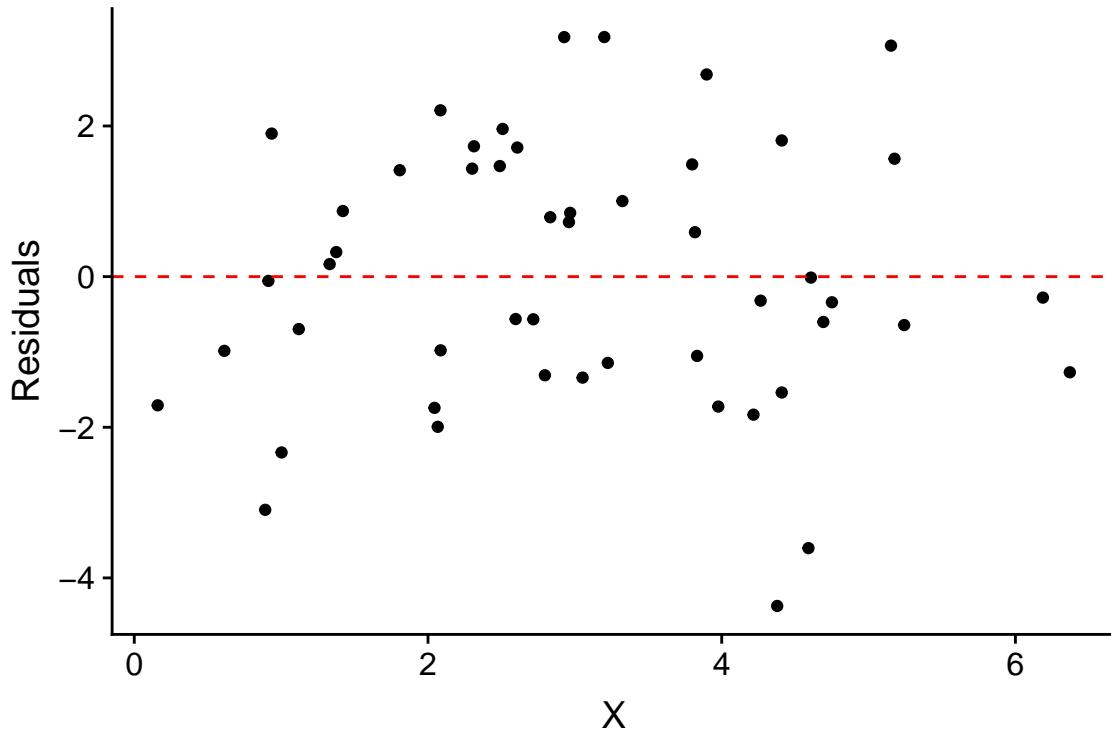
## 22.7 Residuals

Residuals are the difference between the observed values of  $Y$  and the predicted values. You can think of residuals as the proportion of  $Y$  unaccounted for by the model.

$$\text{residuals} = Y - \hat{Y}$$

The previous figure showed the residuals as dashed lines connected the observed and predicted values. A common way to depict the residuals, is to plot the predictor values versus the corresponding residual value, like so:

```
ggplot(df.xy, aes(x = X)) +
  geom_point(aes(y = fit.xy$residuals)) +
  geom_hline(yintercept = 0, color = 'red', linetype = "dashed") +
  labs(x = "X", y = "Residuals")
```



When the linear regression model is appropriate, residuals should be normally distributed, centered around zero and should show no strong trends or extreme differences in spread (variance) for different values of  $X$ .

## 22.8 Regression as sum-of-squares decomposition

Regression can be viewed as a decomposition of the sum-of-squared deviations..

$$ss(Y) = ss(\hat{Y}) + ss(\text{residuals})$$

Let's check this for our example:

```
> ss.Y <- sum((Y - mean(Y))^2)
> ss.Yhat <- sum((fit.xy$fitted.values - mean(Y))^2)
> ss.residuals <- sum(fit.xy$residuals^2)
> ss.Y
#> [1] 412.6367
> ss.Yhat + ss.residuals
#> [1] 412.6367
```

## 22.9 Variance “explained” by a regression model

We can use the sum-of-square decomposition to understand the relative proportion of variance “explained” (accounted for) by the regression model.

We call this quantity the “Coefficient of Determination”, designated  $R^2$ .

$$R^2 = \left(1 - \frac{SS_{residuals}}{SS_{total}}\right)$$

For this particular example we can estimate  $R^2$  as follows:

```
R2 <- 1.0 - (ss$residuals/ss$Y)
R2
#> [1] 0.6345462
```

In this particular example, we find our linear model accounts for about 63% of the variance in  $Y$ . Note that the coefficient of determination is also reported when you apply the `summary` function to a linear model.

## 22.10 Broom: a library for converting model results into data frames

The model fit object we got back when we used the `lm` function to carry out linear regression, carries lots of useful information it isn’t a particularly “tidy” way to access the data. The R package Broom converts “statistical analysis objects from R into tidy data frames, so that they can more easily be combined, reshaped and otherwise processed with tools like ‘dplyr’, ‘tidyr’ and ‘ggplot2’. The discussion of Broom below is drawn from the Introduction to Broom

If you haven’t already done so, install the `broom` package before proceeding.

```
library(broom)
```

There are three `broom` functions that are particularly useful for our purposes. They are:

1. `tidy` – constructs a data frame that summarizes the model’s statistical findings.
2. `augment` – add columns to the original data that was modeled. This includes predictions, residuals, and cluster assignments.
3. `glance` – construct a concise one-row summary of the model.

### 22.10.1 broom::tidy

`tidy` applied to a regression model object returns a table giving the estimated coefficients and other information about the uncertainty of those estimates and corresponding p-values. For now we’re just interested in the estimates, the other values will be described in detail when we get to statistical inference.

```
tidy(fit.xy)
#> # A tibble: 2 x 5
#>   term      estimate std.error statistic  p.value
#>   <chr>     <dbl>     <dbl>     <dbl>    <dbl>
#> 1 (Intercept) 0.7688    0.5755    1.336  1.879e- 1
#> 2 X           1.551     0.1699    9.129  4.581e-12
```

### 22.10.2 broom::augment

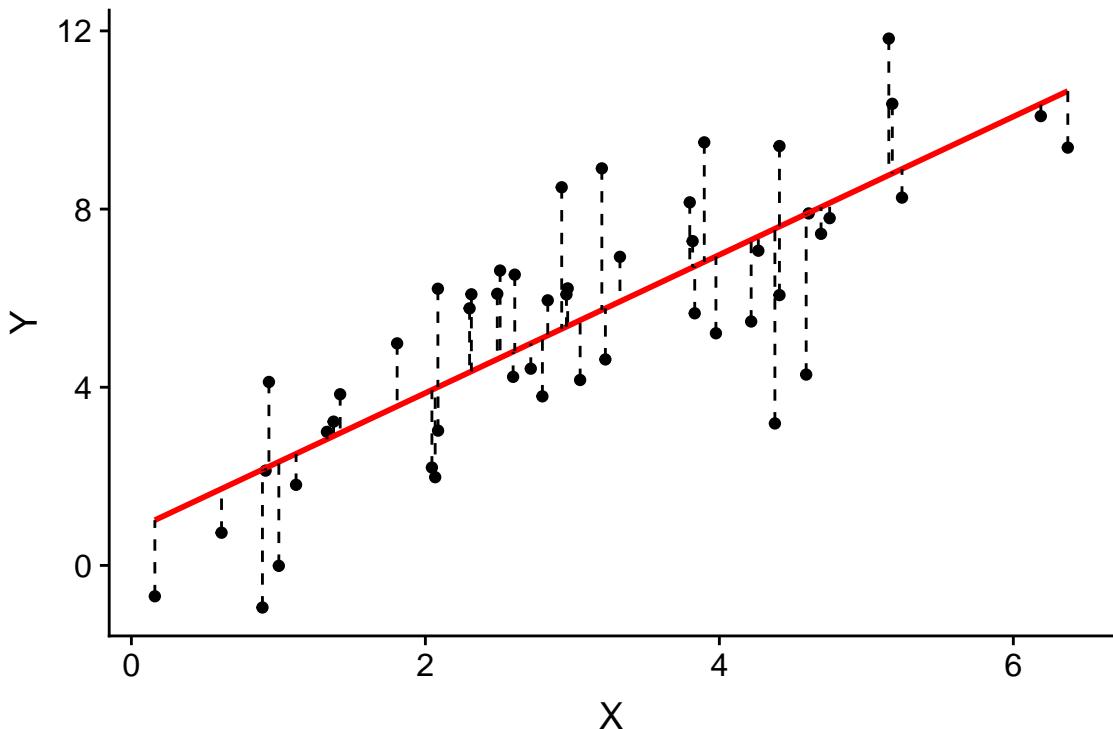
`augment` creates a data frame that combines the original data with related information from the model fit.

```
df.xy.augmented <- augment(fit.xy, df.xy)
head(df.xy.augmented)
#> # A tibble: 6 x 9
#>   X     Y .fitted .se.fit .resid    .hat .sigma .cooksdi
#>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 2.086  3.026  4.005 0.2993 -0.9787 0.02851 1.785 4.606e-3
#> 2 1.120  1.810  2.506 0.4126 -0.6967 0.05418 1.788 4.679e-3
#> 3 1.808  4.985  3.573 0.3276 1.412  0.03415 1.779 1.162e-2
#> 4 1.375  3.228  2.902 0.3790 0.3255 0.04573 1.791 8.469e-4
#> 5 1.003  -0.009990 2.325 0.4285 -2.335  0.05846 1.757 5.720e-2
#> 6 0.8916 -0.9444  2.152 0.4440 -3.096  0.06276 1.729 1.090e-1
#> # ... with 1 more variable: .std.resid <dbl>
```

Now, in addition to the  $X$  and  $Y$  variables of the original data, we have columns like `.fitted` (value of  $Y$  predicted by the model for the corresponding value of  $X$ ), `.resid` (difference between the actual  $Y$  and the predicted value), and a variety of other information for evaluating model uncertainty.

One thing we can do with this “augmented” data frame is to use it to better visualize and explore the model. For example, if we wanted to generate a figure highlighting the deviations from the model using vertical lines emanating from the regression line, we could do something like this:

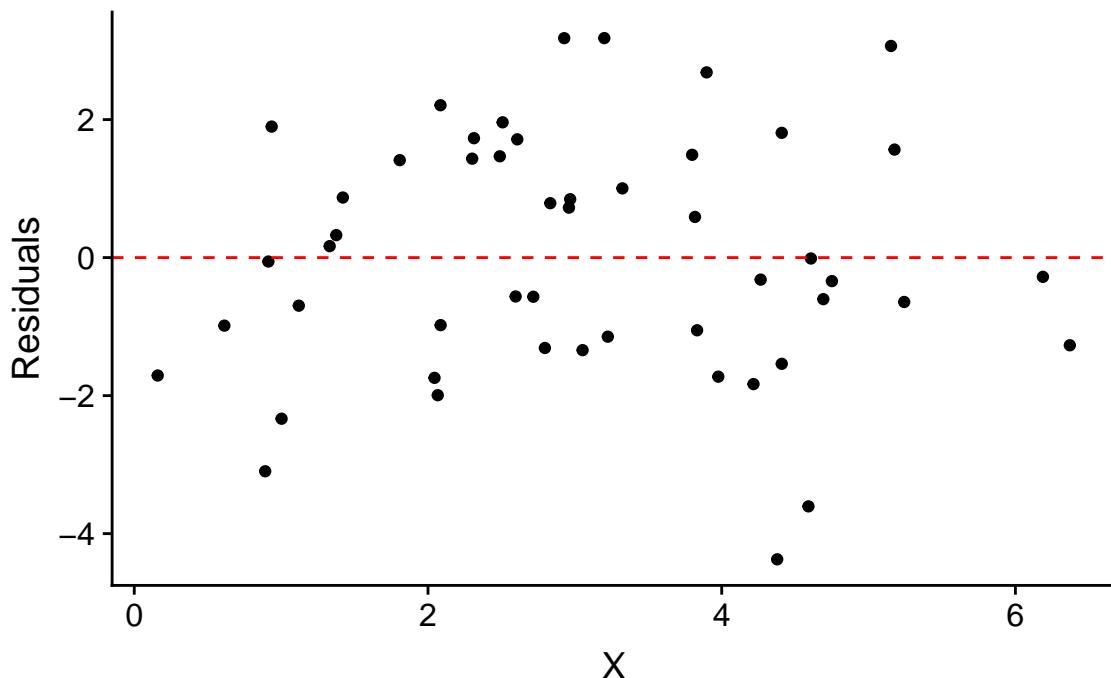
```
ggplot(df.xy.augmented, aes(X, Y)) +
  geom_point() + geom_smooth(method="lm", color="red", se=FALSE) +
  geom_segment(aes(xend = X, yend = .fitted), linetype="dashed")
```



As another example, we can recreate our residual plot using the augmented data frame as so:

```
ggplot(df.xy.augmented, aes(X, .resid)) +
  geom_point() +
  geom_hline(yintercept = 0, color = "red", linetype='dashed') +
  labs(y = "Residuals", title = "Residual plot for synthetic data example.")
```

### Residual plot for synthetic data example.



#### 22.10.3 `broom::glance`

`glance()` provides summary information about the goodness of fit of the model. Most relevant for our current discussion is the column giving the coefficient of determination (`r.squared`):

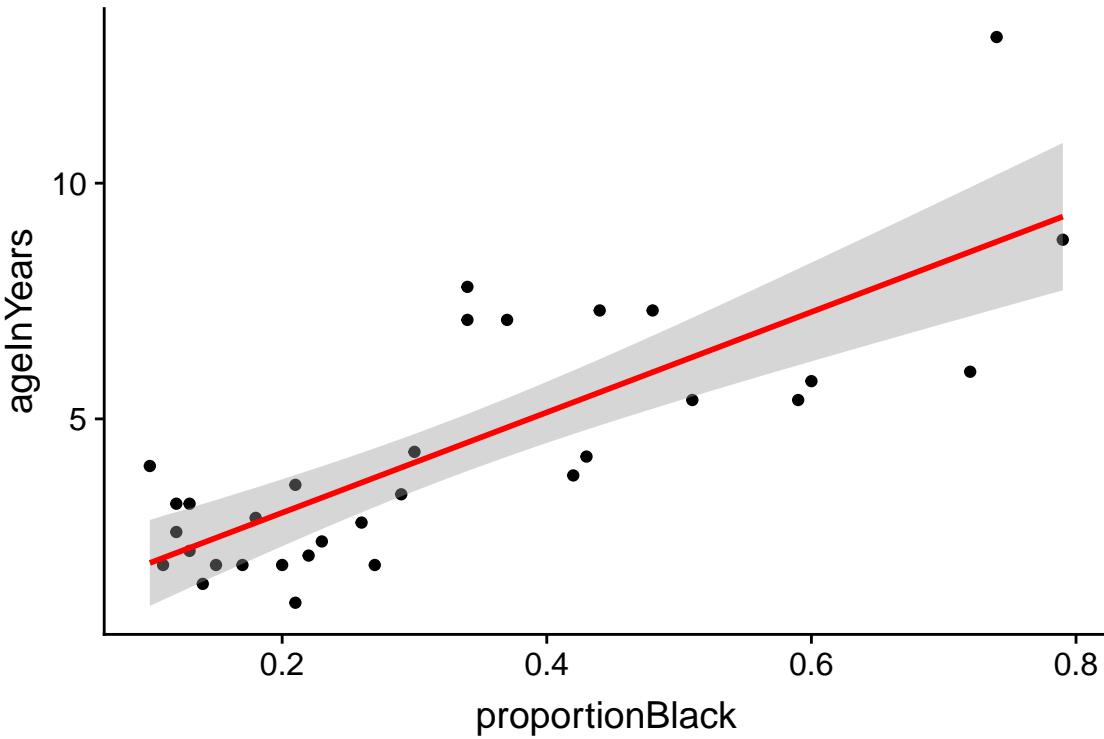
```
glance(fit.xy)
#> # A tibble: 1 x 11
#>   r.squared adj.r.squared sigma statistic  p.value    df logLik    AIC
#> *     <dbl>         <dbl>  <dbl>      <dbl> <int>  <dbl> <dbl>
#> 1     0.6345        0.6269 1.7712     83.34 4.581e-12     2 -98.55 203.1
#> # ... with 3 more variables: BIC <dbl>, deviance <dbl>, df.residual <int>
```

## 22.11 Example: Predicting lion age based on nose color

Having walked through a simulation example, let's now turn to a real world data set. A study by Whitman et al. (2004) showed that the amount of black coloring on the nose of male lions increases with age, and suggested that this might be used to estimate the age of unknown lions. To establish the relationship between these variables they measured the black coloring on the noses of male lions of known age (represented as a proportion), giving the bivariate relationship (and fitted model) shown below:

```
lions <- read_csv("https://github.com/bio304-class/bio304-course-notes/raw/master/datasets/ABD-lion-nos

ggplot(lions, aes(x = proportionBlack, y = ageInYears)) +
  geom_point() +
  geom_smooth(method="lm", color = 'red')
```



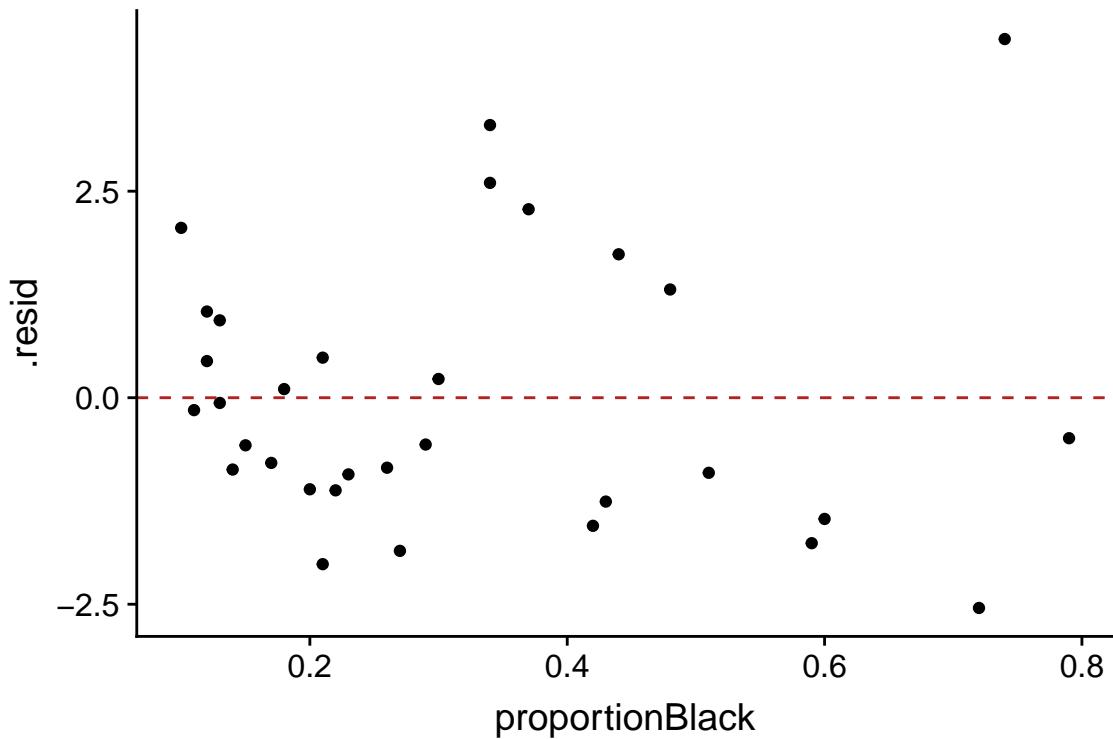
By eye, the linear model looks like a pretty good fit. Let's take a look at the quantitative values of the regression model, using the various *Broom* functions to produce nice output.

```
lion.model <- lm(ageInYears ~ proportionBlack, data = lions)
tidy(lion.model)
#> # A tibble: 2 x 5
#>   term        estimate std.error statistic    p.value
#>   <chr>      <dbl>     <dbl>     <dbl>      <dbl>
#> 1 (Intercept) 0.8790    0.5688    1.545  0.1328
#> 2 proportionBlack 10.65     1.510     7.053 0.00000007677

glance(lion.model)
#> # A tibble: 1 x 11
#>   r.squared adj.r.squared sigma statistic  p.value    df logLik   AIC   BIC
#> *   <dbl>         <dbl>    <dbl>     <dbl>      <dbl>   <dbl> <dbl> <dbl>
#> 1   0.6238        0.6113  1.669     49.75 7.677e-8     2 -60.76 127.5 131.9
#> # ... with 2 more variables: deviance <dbl>, df.residual <int>
```

We then augment our data set with information from the model fit and plot a residual plot:

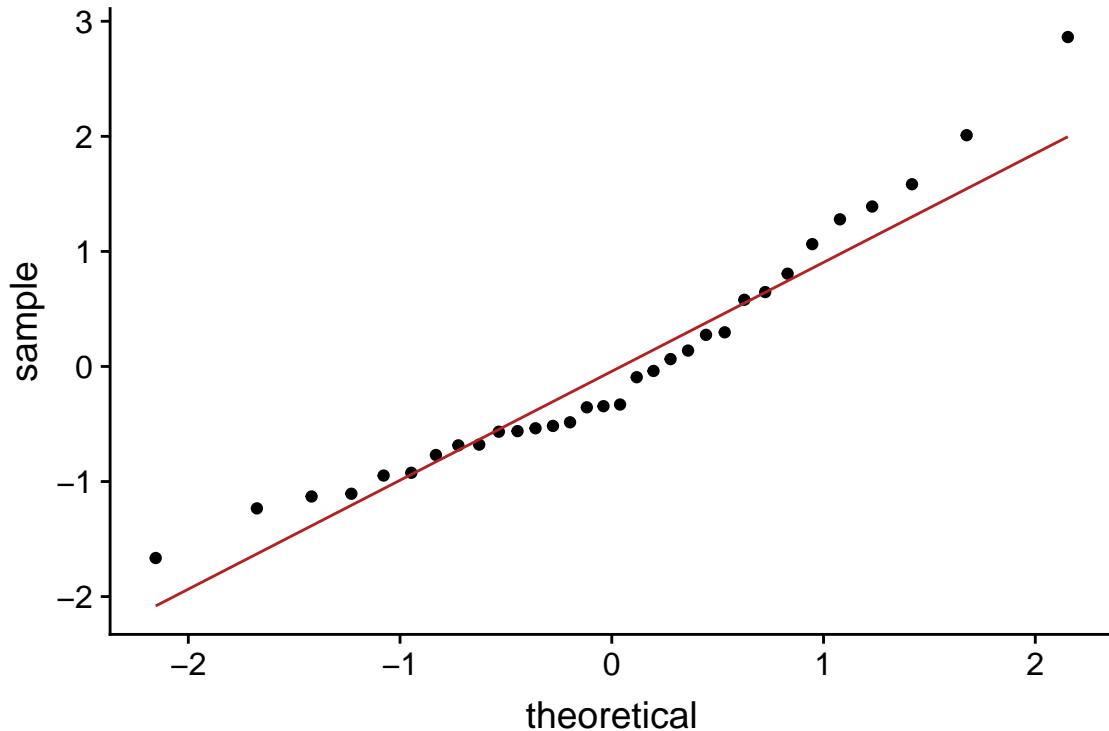
```
lions.augmented <- augment(lion.model, lions)
ggplot(lions.augmented, aes(proportionBlack, .resid)) +
  geom_point() +
  geom_hline(yintercept = 0, color="firebrick", linetype="dashed")
```



From this plot, there may be some indication of greater variance of residuals for larger values of the predictor variable.

Let's check how normal the residuals look using a QQ-plot. Here we construct the QQ-plot using "standardized residuals" which are just z-scores for the residuals.

```
ggplot(lions.augmented, aes(sample = .std.resid)) +  
  geom_qq() +  
  geom_qq_line(color="firebrick")
```



Based on the QQ-plot, the residuals seem to diverge somewhat from a normal distribution, as there's noticeable curvature in the QQ-plot. When we test for the normality of the residuals using Shapiro-Wilk's test for normality, we fail to reject the null hypothesis of normality at a significance threshold of  $\alpha = 0.05$ :

```
shapiro.test(lions.augmented$resid)
#>
#> Shapiro-Wilk normality test
#>
#> data: lions.augmented$resid
#> W = 0.93879, p-value = 0.0692
```

Even though we failed to reject the null hypothesis of normality for the residuals, but the P-value is very close to significance, suggesting some caution in applying the linear model.

# Chapter 23

## Multiple regression

### 23.1 Libraries to install

We'll be using several new packages for this class session. Install the following packages via one of the standard install mechanisms:

- `HistData` – provides the `GaltonFamilies` example data sets we'll work with
- `plot3D` – for generating 3D plots
- `rgl` – NOTE: On OS X, `rgl` requires you to install a program called XQuartz. XQuartz can be downloaded from the XQuartz Home Page. If you're on a Mac, install XQuartz before installing `rgl`. You may have to reboot your computer after installing XQuartz.

### 23.2 Review of bivariate regression

Recall the model for bivariate least-squares regression. When we regress  $Y$  and  $X$  we're looking for a linear function,  $f(X)$ , for which the following sum-of-squared deviations is minimized:

$$\sum_{i=1}^n (y_i - f(x_i))^2$$

The general form a linear function of one variable is a line,

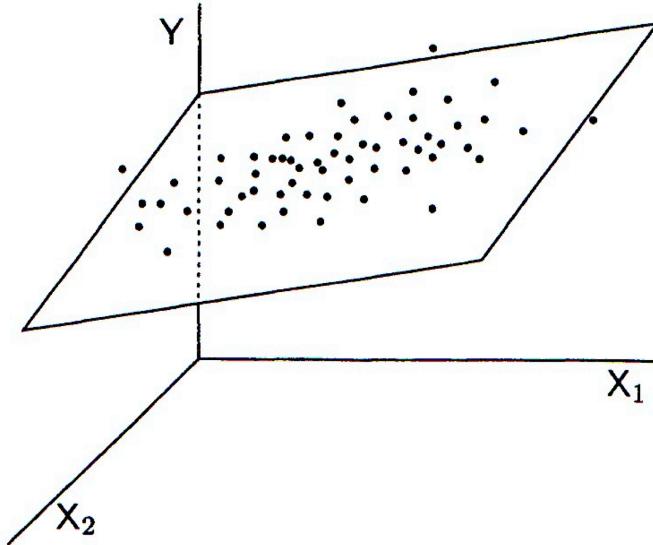
$$\hat{Y} = f(x) = a + bX$$

where  $b$  is the slope of the line and  $a$  is the intercept.

### 23.3 Multiple regression

The idea behind multiple regression is almost exactly the same as bivariate regression, except now we try and fit a linear model for  $Y$  using multiple explanatory variables,  $X_1, X_2, \dots, X_m$ . That is we're looking for a linear function,  $f(X_1, X_2, \dots, X_m)$  that minimizes:

$$\sum_{i=1}^n (y_i - f(x_1, x_2, \dots, x_m))^2$$



**Figure 4.1:** The regression of Y onto X<sub>1</sub> and X<sub>2</sub> as a scatterplot in variable space.

Figure 23.1: Multiple regression, two predictor variables

A linear function of more than one variable is written as:

$$\hat{Y} = f(X_1, X_2, \dots, X_m) = a + b_1X_1 + b_2X_2 + \dots + b_mX_m$$

Where  $a$  is the intercept and  $b_1, b_2, \dots, b_m$  are the **regression coefficients**.

### 23.3.1 Geometrical interpretation

Geometrically the regression coefficients have the same interpretation as in the bivariate case – slopes with respect to the corresponding variable. When there are two predictor variables, the linear regression is geometrically a plane in 3-space, as shown in the figure below. When there are more than two predictor variables, the regression solution is a hyper-plane.

Mathematically, the best fitting regression coefficients,  $b_1, b_2, \dots, b_m$ , are found using linear algebra. Since we haven't covered linear algebra in this course, I will omit the details. Conceptually the thing to remember is that the regression coefficients are related to the magnitude of the standard deviations of the predictor variables and the covariances between the predictor and outcome variables.

### 23.3.2 Coefficient of determination for multiple regression

As in bivariate regression, the coefficient of determination ( $R^2$ ) provides a measure of the proportion of variance in the outcome variable (Y) “explained” by the predictor variables (X<sub>1</sub>, X<sub>2</sub>, ...).

## 23.4 Interpreting Multiple Regression

Here are some things to keep in mind when interpreting a multiple regression:

- In most cases of regression, causal interpretation of the model is not justified.
- Standard bivariate and multiple regression assumes that the predictor variables ( $(X_1, X_2, \dots)$ ) are observed without error. That is, uncertainty in the regression model is only associated with the outcome variable, not the predictors.
- Comparing the size of regression coefficients only makes sense if all the predictor (explanatory) variables have the same scale
- If the explanatory variables ( $X_1, X_2, \dots, X_m$ ) are highly correlated, then the regression solution can be “unstable” – a small change in the data could lead to a large change in the regression model.

## 23.5 Libraries

```
library(tidyverse)
library(cowplot)
library(broom)
library(GGally)
library(plot3D)
library(rgl)
```

## 23.6 Example data set: mtcars

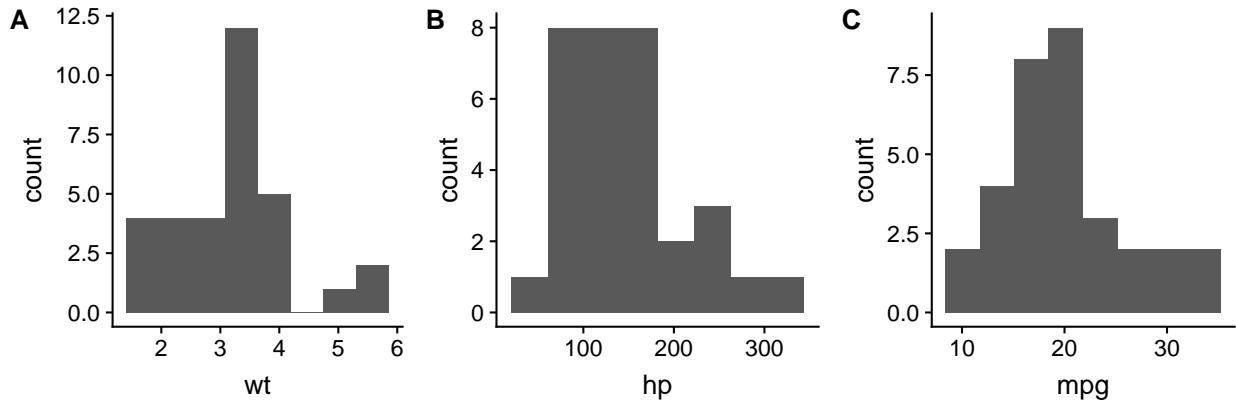
The `mtcars` dataset contains information on fuel consumption and ten other aspects of car design (see `?mtcars` for more info). We'll use multiple regression to model the relationship between fuel consumption (`mpg`) and a vehicle's weight (`wt`) and horsepower (`hp`).

## 23.7 Visualizing and summarizing the variables of interest

Before carrying out any regression model it's always a good idea to start out with visualizations of the individual variables first.

```
hist.wt <- ggplot(mtcars, aes(wt)) + geom_histogram(bins=8)
hist.hp <- ggplot(mtcars, aes(hp)) + geom_histogram(bins=8)
hist.mpg <- ggplot(mtcars, aes(mpg)) + geom_histogram(bins=8)

plot_grid(hist.wt, hist.hp, hist.mpg, nrow = 1, labels = c("A", "B", "C"))
```



Let's also create some quick data summaries for our variables:

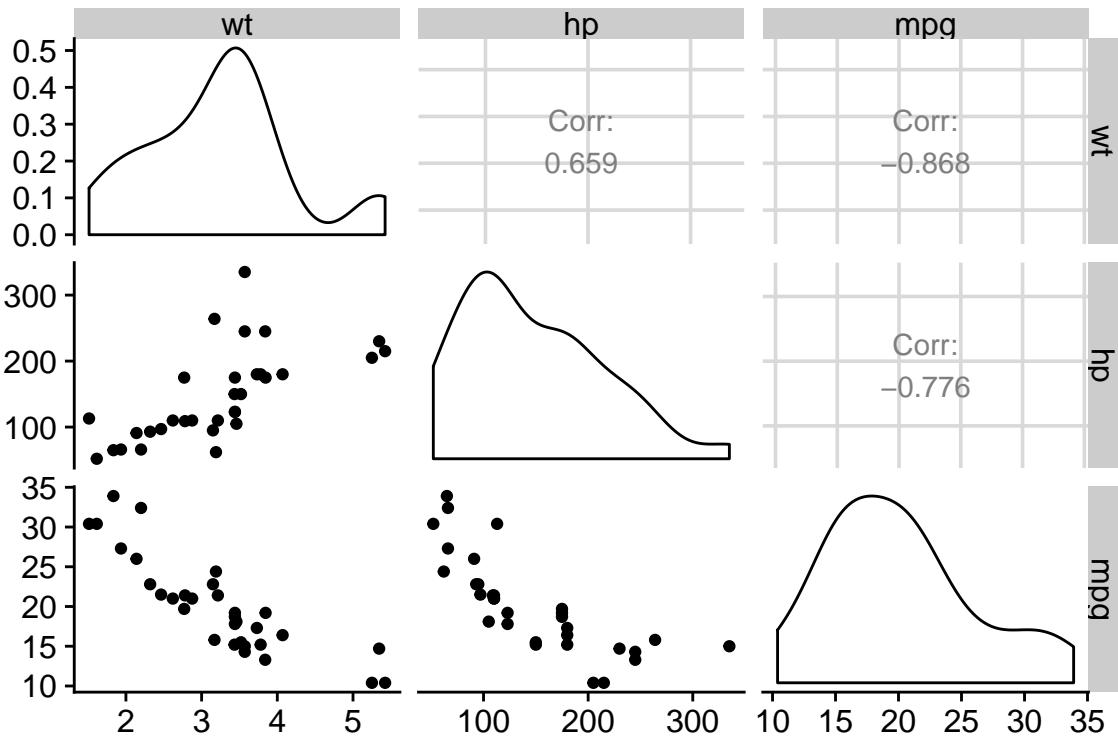
```
mtcars.subset <-  
  mtcars %>%  
  select(wt, hp, mpg)  
  
summary(mtcars.subset)
#>      wt          hp          mpg
#> Min.   :1.513   Min.   :52.0    Min.   :10.40
#> 1st Qu.:2.581   1st Qu.:96.5    1st Qu.:15.43
#> Median :3.325   Median :123.0   Median :19.20
#> Mean   :3.217   Mean   :146.7   Mean   :20.09
#> 3rd Qu.:3.610   3rd Qu.:180.0   3rd Qu.:22.80
#> Max.   :5.424   Max.   :335.0   Max.   :33.90
```

And a correlation matrix to summarize the bivariate associations between the variables:

```
cor(mtcars.subset)
#>      wt          hp          mpg
#> wt   1.0000000  0.6587479 -0.8676594
#> hp   0.6587479  1.0000000 -0.7761684
#> mpg -0.8676594 -0.7761684  1.0000000
```

We can use the `GGally::ggpairs()` function, which we've seen previously, to create a visualization of the bivariate relationships:

```
ggpairs(mtcars.subset)
```



From the scatter plots and correlation matrix we see that weight and horsepower are positively correlated, but both are negatively correlated with fuel economy. This jives with our intuition – bigger cars with more powerful engines generally get lower gas mileage.

## 23.8 3D plots

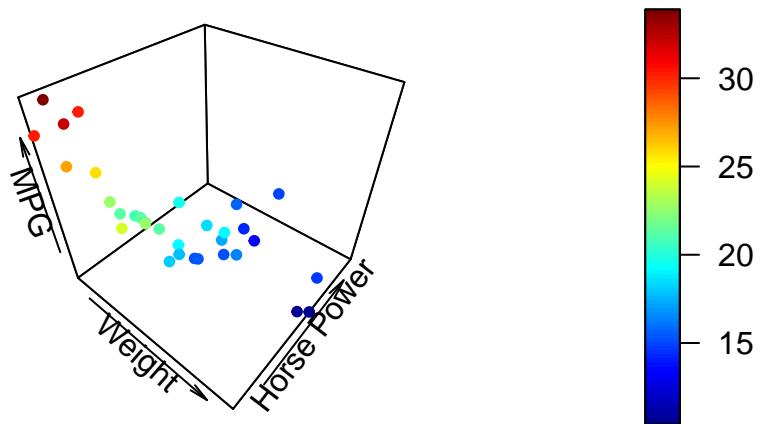
Since we're building a model that involves three variables, it makes sense to look at a 3D plot. `ggplot2` has no built-in facilities for 3D scatter plots so we'll use a package called `plot3D`. `plot3D` follows the plotting conventions of the base R-graphics capabilities, so we can't build up figures in layers in the same way we do in `ggplot`. Instead we pass all the formatting arguments to a single function call.

To create a 3D scatter plot we can use the `plot3D::points3D` function. The argument `pch` sets the type of plotting character to use in the plot (for a graphical key of the available plotting characters see this link).

```
library(plot3D)

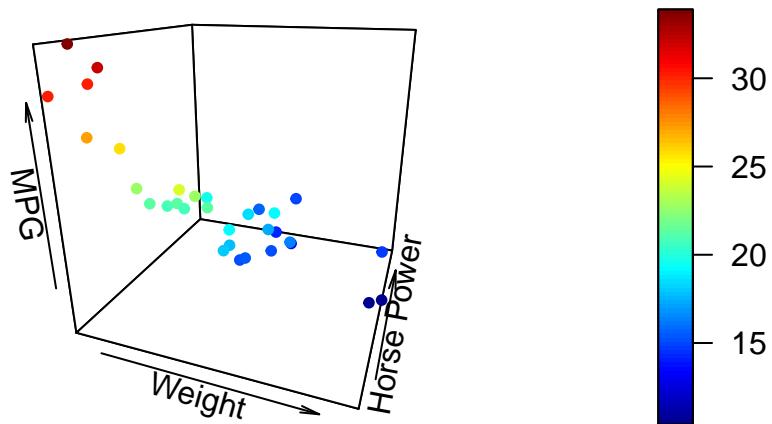
# create short variable names for convenience
wt <- mtcars$wt
hp <- mtcars$hp
mpg <- mtcars$mpg

points3D(wt, hp, mpg,
         xlab = "Weight", ylab = "Horse Power", zlab = "MPG",
         pch = 20)
```



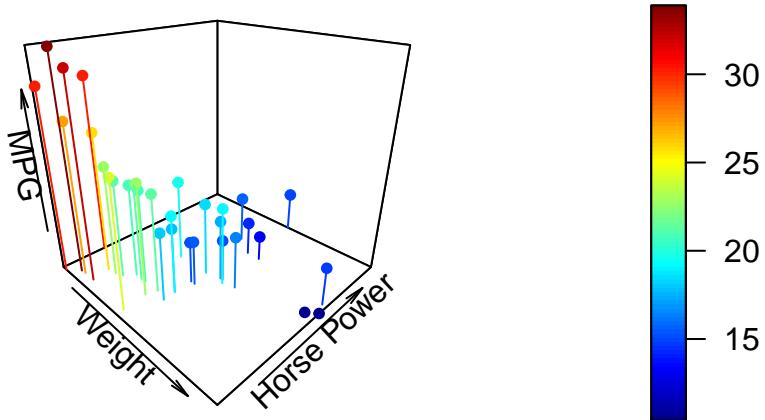
We can change the angle of the 3D plot using the arguments `theta` and `phi` which change the “azimuthal direction” and “colatitude” (inclination angle). See the wikipedia page on spherical coordinate systems for more explanation of this values.

```
points3D(wt, hp, mpg,
         xlab = "Weight", ylab = "Horse Power", zlab = "MPG",
         pch = 20,
         theta = 20, phi = 20  # these set viewing angle
)
```



If you want the points to have a uniform color specify a single color in the `col` argument. Here we also add vertical lines to the plot using the `type` argument and show

```
points3D(wt, hp, mpg,
         xlab = "Weight", ylab = "Horse Power", zlab = "MPG",
         pch = 20,
         theta = 45, phi = 25,
         type = "h")
```



For more examples of how you can modify plots generated with the `plot3D` package see this web page.

## 23.9 Fitting a multiple regression model in R

Using the `lm()` function, fitting multiple regression models is a straightforward extension of fitting a bivariate regression model.

```
fit.mpg <- lm(mpg ~ wt + hp, data = mtcars.subset)
summary(fit.mpg)
#>
#> Call:
#> lm(formula = mpg ~ wt + hp, data = mtcars.subset)
#>
#> Residuals:
#>    Min     1Q Median     3Q    Max
#> -3.941 -1.600 -0.182  1.050  5.854
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept) 37.22727   1.59879  23.285 < 2e-16 ***
#> wt          -3.87783   0.63273  -6.129 1.12e-06 ***
#> hp          -0.03177   0.00903  -3.519  0.00145 **
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 2.593 on 29 degrees of freedom
#> Multiple R-squared:  0.8268, Adjusted R-squared:  0.8148
#> F-statistic: 69.21 on 2 and 29 DF,  p-value: 9.109e-12
```

As was the case for bivariate regression, the `broom` package functions `tidy`, `glance`, and `augment` can be useful for working with the results from fitting the mode.

```
tidy(fit.mpg)
#> # A tibble: 3 x 5
#>   term      estimate std.error statistic p.value
#>   <chr>     <dbl>    <dbl>     <dbl>    <dbl>
#> 1 (Intercept) 37.23     1.599     23.28  2.565e-20
#> 2 wt        -3.878     0.6327    -6.129  1.120e- 6
#> 3 hp       -0.03177    0.009030  -3.519  1.451e- 3

glance(fit.mpg)
#> # A tibble: 1 x 11
#>   r.squared adj.r.squared sigma statistic p.value    df logLik   AIC
#> *     <dbl>          <dbl>  <dbl>     <dbl>    <int>  <dbl>  <dbl>
#> 1     0.8268         0.8148 2.593    69.21 9.109e-12     3 -74.33 156.7
#> # ... with 3 more variables: BIC <dbl>, deviance <dbl>, df.residual <int>

mtcars.subset.augmented <-
  augment(fit.mpg, mtcars.subset)
```

## 23.10 Visualizing the regression plane

For multiple regression on two predictor variables we can visualize the plane of best fit but adding it as a surface to our 3D plot.

```

# Create a regular grid over the range of wt and hp values
grid.lines = 10

wt.grid <- seq(min(wt), max(wt), length.out = grid.lines)
hp.grid <- seq(min(hp), max(hp), length.out = grid.lines)
wthp.grid <- expand.grid(x = wt.grid, y = hp.grid)

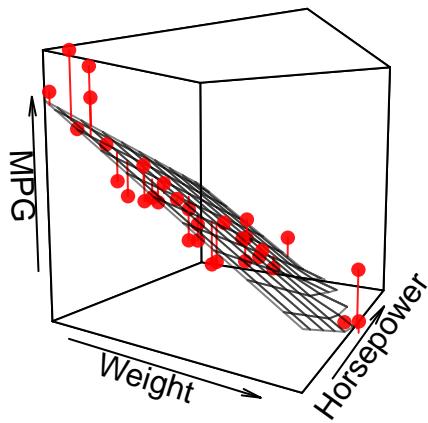
grid.df <- data.frame(wt = wthp.grid[,1], hp = wthp.grid[,2])

# Predicted mpg at each point in grid
mpg.grid <- matrix(predict(fit.mpg, newdata = grid.df),
                     nrow = grid.lines, ncol = grid.lines)

# Predicted mpg at observed
mpg.predicted <- predict(fit.mpg)

# scatter plot with regression plane
points3D(wt, hp, mpg,
          pch = 16, theta = 30, phi = 5,
          col = "red", alpha=0.9,
          xlab = "Weight", ylab = "Horsepower", zlab = "MPG",
          surf = list(x = wt.grid,
                      y = hp.grid,
                      z = mpg.grid,
                      facets = NA,
                      fit = mpg.predicted,
                      col = "black", alpha = 0.5)
        )

```



### 23.11 Interactive 3D Visualizations Using OpenGL

The package `rgl` is another package that we can use for 3D visualization. `rgl` is powerful because it lets us create interactive plots we can rotate and zoom in/out on.

Once you've installed and loaded `rgl` try the following code.

```
# create 3D scatter, using spheres to draw points
plot3d(wt, hp, mpg,
       type = "s",
       size = 1.5,
       col = "red")

# only need to include this line if using in a markdown document
rglwidget()
## PhantomJS not found. You can install it with webshot::install_phantomjs(). If it is installed, please
```

We can add a 3d plane to our plot, representing the multiple regression model, with the `rgl.planes()` function as shown below.

```
coefs <- coef(fit.mpg)
b1 <- coefs["wt"]
b2 <- coefs["hp"]
c <- -1
a <- coefs["(Intercept)"]
plot3d(wt, hp, mpg,
       type = "s",
       size = 1.5,
       col = "red")
```

```
rgl.planes(b1, b2, c, a, alpha = 0.9, color = "gray")
rglwidget()
```

## 23.12 Examining the residuals

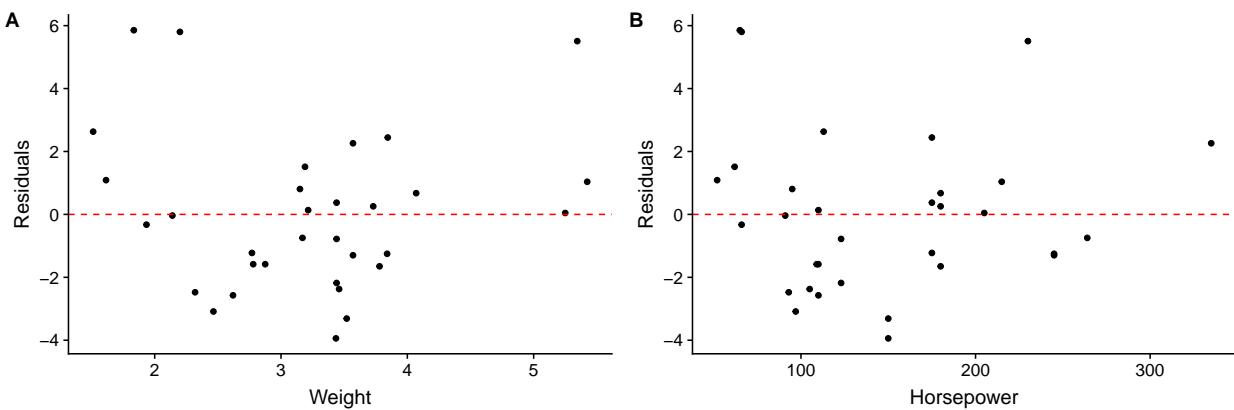
Residual plots are useful for multiple regression, just as they were for bivariate regression.

First we plot the residuals versus each of the predictor variable individually.

```
wt.resids <-
  mtcars.subset.augmented %>%
  ggplot(aes(x = wt, y = .resid)) +
  geom_point() +
  geom_hline(yintercept = 0, linetype = 'dashed', color='red') +
  labs(x = "Weight", y = "Residuals")

hp.resids <-
  mtcars.subset.augmented %>%
  ggplot(aes(x = hp, y = .resid)) +
  geom_point() +
  geom_hline(yintercept = 0, linetype = 'dashed', color='red') +
  labs(x = "Horsepower", y = "Residuals")

plot_grid(wt.resids, hp.resids, labels=c("A", "B"))
```



And now we plot the residuals in 3D space, with a plane parallel to the xy-plane (wt, hp-plane) representing the plane about which the residuals should be homogeneously scattered if the assumptions of the linear regression model hold.

```
.resid <- mtcars.subset.augmented$.resid

# coefficients for plane perpendicular to
# xy-axis intercepting the z-axis at 0
b1 <- 0
b2 <- 0
c <- -1
a <- 0

plot3d(wt, hp, .resid,
       type = "s",
```

```
size = 1.5,  
col = "red",  
aspect = c(2,2,1))  
rgl.planes(b1, b2, c, a, alpha = 0.9, color = "gray")  
rglwidget()
```

# Chapter 24

## Logistic regression

Logistic regression is used when the dependent variable is discrete (often binary). The explanatory variables may be either continuous or discrete.

Examples:

- Whether a gene is turned off ( $=0$ ) or on ( $=1$ ) as a function of levels of various proteins
- Whether an individual is healthy ( $=0$ ) or diseased ( $=1$ ) as a function of various risk factors.
- Whether an individual died ( $=0$ ) or survived ( $=1$ ) some selective event as a function of behavior, morphology, etc.

We model the binary response variable,  $Y$ , as a function of the predictor variables,  $X_1, X_2$ , etc as :

$$P(Y = 1|X_1, \dots, X_p) = f(\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_p X_p)$$

So we're modeling the *probability of the state of  $Y$  as a function of a linear combination of the predictor variables.*

For logistic regression,  $f$  is the logistic function:

$$f(z) = \frac{e^z}{1 + e^z} = \frac{1}{1 + e^{-z}}$$

Therefore, the bivariate logistic regression is given by:

$$P(Y = 1|X) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 X)}}$$

Note that  $\beta_0$  here is akin to the intercept in our standard linear regression.

### 24.1 A web app to explore the logistic regression equation

To help you develop an intuition for the logistic regression equation, I've developed a small web app, that allows you to explore how the shape of the regression curve responds to changes in the regression coefficients  $\beta_0$  and  $\beta_1$ . Open the app in another browser window and play with the sliders that control the coefficients  $B_0$  and  $B_1$ . In the assignment associated with today's class you'll be asked to answer some specific questions based on this app.

## 24.2 Titanic data set

titanic.csv contains information about passengers on the Titanic. Variables in this data set include information such as sex, age, passenger class (1st, 2nd, 3rd), and whether or not they survived the sinking of the ship (0 = died, 1 = survived).

```
library(tidyverse)
library(broom)
library(cowplot)
library(ggthemes)

titanic <- read_csv("http://bit.ly/bio304-titanic-data")
names(titanic)
#> [1] "pclass"      "survived"     "name"        "sex"         "age"
#> [6] "sibsp"       "parch"       "ticket"      "fare"        "cabin"
#> [11] "embarked"    "boat"        "body"        "home.dest"
```

## 24.3 Subsetting the data

We've all heard the phrase, "Women and children first", so we might expect that the probability that a passenger survived the sinking of the Titanic is related to their sex and/or age. Let's create separate data subsets for male and female passengers.

```
male <- filter(titanic, sex == "male")
female <- filter(titanic, sex == "female")
```

## 24.4 Visualizing survival as a function of age

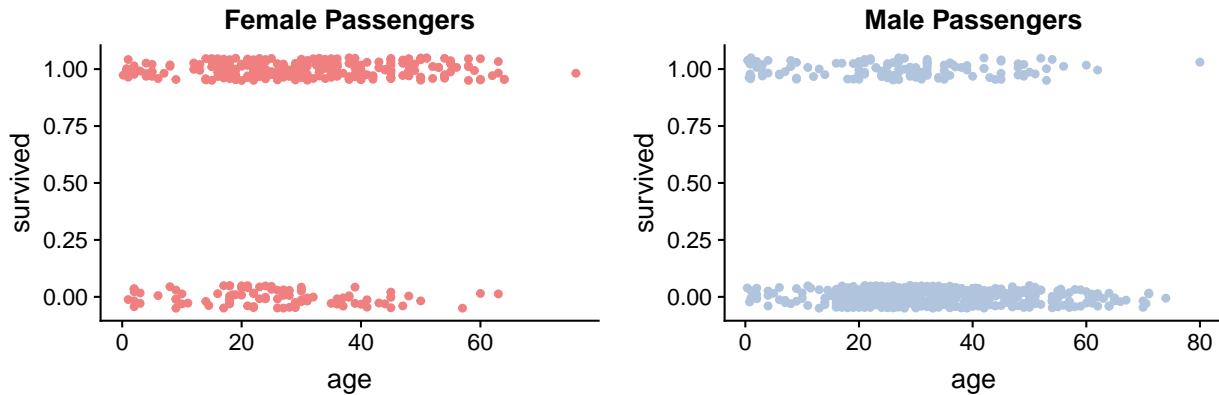
Let's create visualizations of survival as a function of age for the male and female passengers.

```
fcolor = "lightcoral"
mcolor = "lightsteelblue"

female.plot <- ggplot(female, aes(x = age, y = survived)) +
  geom_jitter(width = 0, height = 0.05, color = fcolor) +
  labs(title = "Female Passengers")

male.plot <- ggplot(male, aes(x = age, y = survived)) +
  geom_jitter(width = 0, height = 0.05, color = mcolor) +
  labs(title = "Male Passengers")

plot_grid(female.plot, male.plot)
```



The jittered points with Y-axis value around one are passengers who survived, the point jittered around zero are those who died.

## 24.5 Fitting the logistic regression model

The function `glm` (generalized linear model) can be used to fit the logistic regression model (as well as other models). Setting the argument `family = binomial` gives us logistic regression. Note that when fitting the model the dependent variable needs to be numeric, so if the data is provided as Boolean (logical) TRUE/FALSE values, they should be converted to integers using `as.numeric()`.

First we fit the regression for the female passengers.

```
fit.female <- glm(survived ~ age, family = binomial, female)
tidy(fit.female)
#> # A tibble: 2 x 5
#>   term      estimate std.error statistic p.value
#>   <chr>     <dbl>     <dbl>     <dbl>    <dbl>
#> 1 (Intercept) 0.4934    0.2542    1.941  0.05226
#> 2 age        0.02252   0.008535   2.638  0.008342
```

The column “estimate” gives the coefficients of the model. The “intercept” estimate corresponds to  $B_0$  in the logistic regression equation, the “age” estimate corresponds to the coefficient  $B_1$  in the equation.

Now we repeat the same step for the male passengers.

```
fit.male <- glm(survived ~ age, family = binomial, male)
tidy(fit.male)
#> # A tibble: 2 x 5
#>   term      estimate std.error statistic p.value
#>   <chr>     <dbl>     <dbl>     <dbl>    <dbl>
#> 1 (Intercept) -0.6608    0.2248    -2.939  0.003290
#> 2 age        -0.02376   0.007276   -3.266  0.001092
```

Notice that the female coefficients are both positive, while the male coefficients are negative. We’ll visualize what this means in terms of the model below.

## 24.6 Visualizing the logistic regression

To visualize the logistic regression fit, we first use the `predict` function to generate the model predictions about probability of survival as a function of age.

```
ages <- seq(0, 75, 1) # predict survival for ages 0 to 75

predicted.female <- predict(fit.female,
                           newdata = data.frame(age = ages),
                           type = "response")

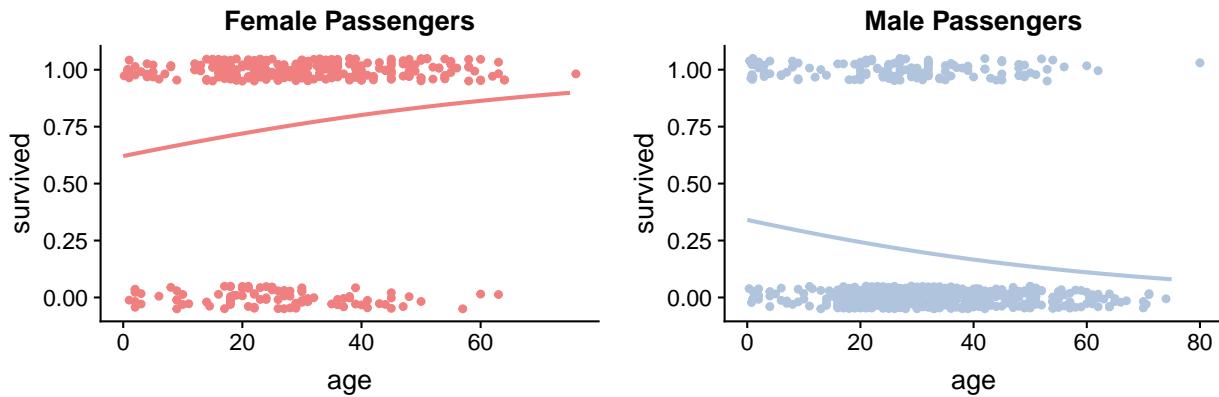
predicted.male <- predict(fit.male,
                           newdata = data.frame(age = ages),
                           type = "response")
```

Having generated the predicted probabilities of survival we can then add these prediction lines to our previous plot using `geom_line`.

```
female.logistic.plot <- female.plot +
  geom_line(data = data.frame(age = ages, survived = predicted.female),
            color = fcolor, size = 1)

male.logistic.plot <- male.plot +
  geom_line(data = data.frame(age = ages, survived = predicted.male),
            color = mcolor, size = 1)

plot_grid(female.logistic.plot, male.logistic.plot)
```

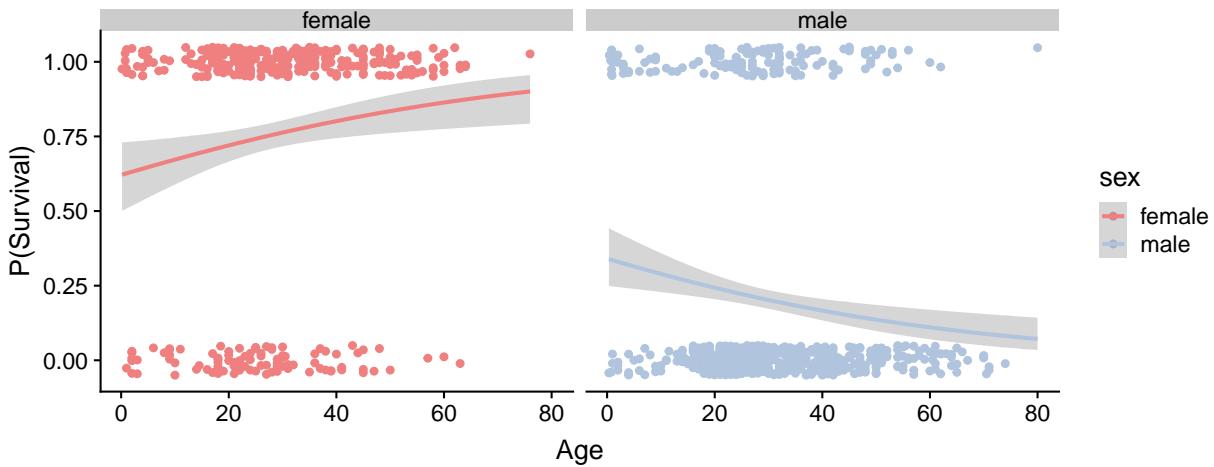


We see that for the female passengers, the logistic regression predicts that the probability of survival *increases* with passenger age. In contrast, the model fit to the male passengers suggests that the probability of survival decreases with passenger age. For the male passengers, the data is consistent with “children first”; for female passengers this model doesn’t seem to hold. However, there are other factors to consider as we’ll see below.

### 24.6.1 Quick and easy visualization

Here’s an alternative “quick and easy” way to generate the plot above using the awesome power of ggplot. The downside of this approach is we don’t generate the detailed information on the model, which is something you’d certainly want to have in any real analysis.

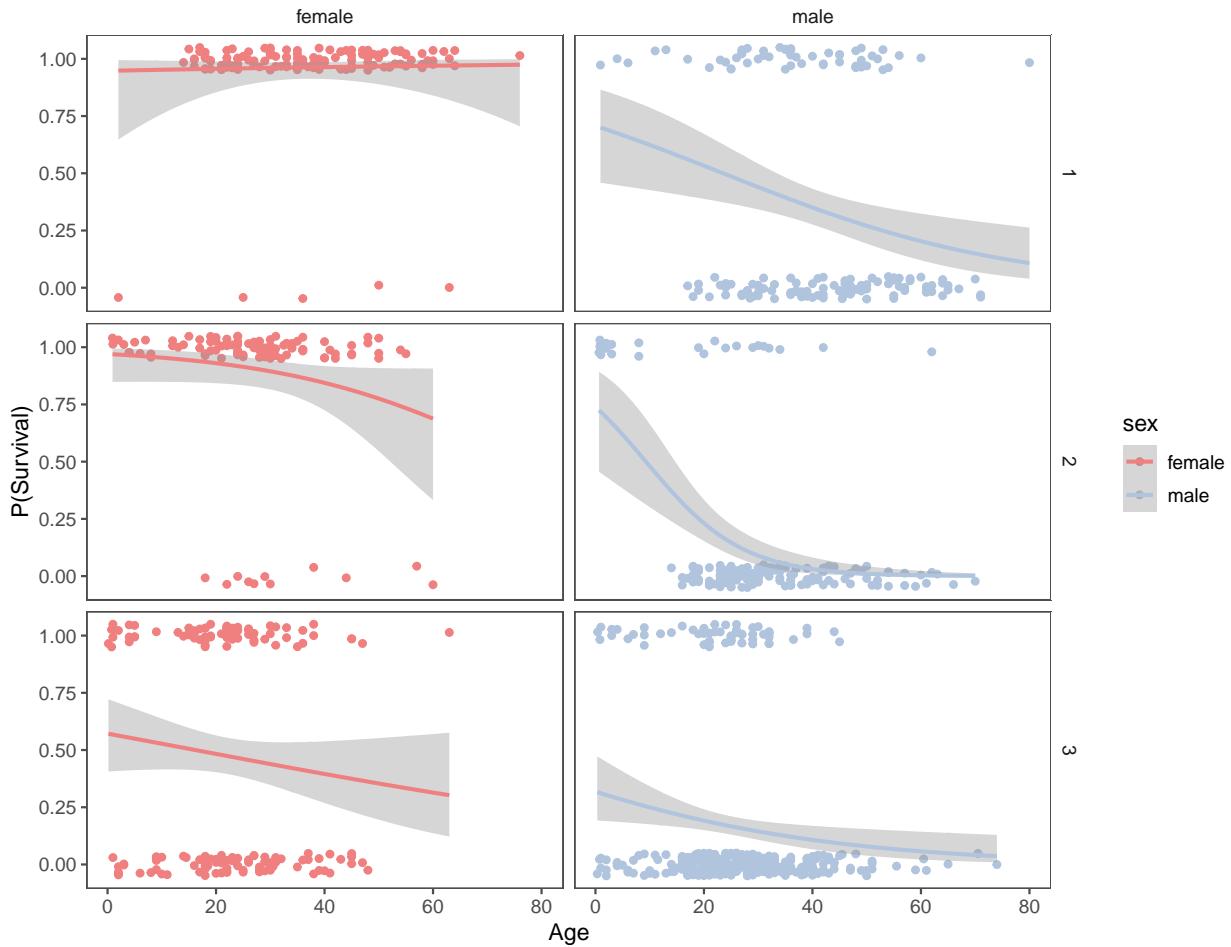
```
ggplot(titanic, aes(x=age, y=survived, color=sex)) +
  geom_jitter(width = 0, height = 0.05) +
  geom_smooth(method="glm", method.args = list(family="binomial")) +
  labs(x = "Age", y = "P(Survival)") +
  facet_wrap(~ sex) +
  scale_color_manual(values = c(fcolor, mcolor))
```



## 24.7 Impact of sex and passenger class on the models

In our previous analysis we considered the relationship between survival and age, conditioned (facted) on passenger sex. In a complex data set like this one, it is often useful to condition on multiple variables simultaneously. Lets extend our visualization to look at the regression faceted on both class and sex, using `facet_grid`:

```
ggplot(titanic, aes(x=age, y=survived, color=sex)) +
  geom_jitter(width = 0, height = 0.05) +
  geom_smooth(method="glm", method.args = list(family="binomial")) +
  labs(x = "Age", y = "P(Survival)") +
  facet_grid(pclass ~ sex) +
  scale_color_manual(values = c(fcolor, mcolor)) +
  theme_few()
```



Having conditioned on both sex and ticket class, our figure now reveals a much more complex relationship between age and survival. Almost all first class female passengers survived, regardless of age. For second class female passengers, the logistic regression suggests a very modest decrease in survival with increasing age. The negative relationship between age and survival is stronger still for third class females. Male passengers on the other hand show a negative relationship between sex and survival, regardless of class, but the models suggest that there are still class specific differences in this relationship.

## 24.8 Fitting multiple models based on groupings use `dplyr::do`

In the figure above we used `ggplot` and `facet_grid` to visualize logistic regression of survival on age, conditioned on both sex and class. What if we wanted to calculate the terms of the logistic regressions for each combination of these two categorical variables? There are three passenger classes and two sexes, meaning we'd have to create six data subsets and fit the model six times if we used the same approach we used previously. Luckily, `dplyr` provides a powerful function called `do()` that allows us to carry out arbitrary computations on grouped data.

There are two ways to use `do()`. The first way is to give the expressions you evaluate in `do()` a name, in which case `do()` will store the results in a column. The second way to use `do()` is for the expression to return a data frame.

In this first example, the model fits are stored in the `fits` column. When using `do()` you can refer to the groupings using a period (.):

```
grouped.models <-
  titanic %>%
  group_by(sex, pclass) %>%
  do(fits = glm(survived ~ age, family = binomial, data = .))

grouped.models
#> # Source: local data frame [6 x 3]
#> # Groups: <by row>
#>
#> # A tibble: 6 x 3
#>   sex     pclass fits
#>   <chr>    <int> <list>
#> 1 female      1 <S3: glm>
#> 2 female      2 <S3: glm>
#> 3 female      3 <S3: glm>
#> 4 male        1 <S3: glm>
#> 5 male        2 <S3: glm>
#> 6 male        3 <S3: glm>
```

Notice that the “fits” column doesn’t explicitly print out the details of the model. The object returned by `glm()` can’t be simply represented as text string (it’s a list), so we see a placeholder string that tells us that there is data here represented a `glm` object. However, we can access the the columns with the fits just like any other variable:

```
# get the summary of the second logistic regression (Female, 2nd Class)
tidy(grouped.models$fits[[2]])
#> # A tibble: 2 x 5
#>   term      estimate std.error statistic  p.value
#>   <chr>      <dbl>     <dbl>      <dbl>      <dbl>
#> 1 (Intercept) 3.491     0.9037     3.863 0.0001119
#> 2 age       -0.04502    0.02548    -1.767 0.07730
```

Now we illustrate the second approach to using `do()`. When no name is provided, `do()` expects its expression to return a data frame. Here we use the `broom::tidy()` function to get the key results of each fit model into a data frame:

```
titanic %>%
  group_by(sex, pclass) %>%
  do(tidy(glm(survived ~ age, family = binomial, data = .)))

#> # A tibble: 12 x 7
#> # Groups:   sex, pclass [6]
#>   sex     pclass term      estimate std.error statistic  p.value
#>   <chr>    <int> <chr>      <dbl>     <dbl>      <dbl>      <dbl>
#> 1 female      1 (Intercept) 2.896     1.238     2.339 0.01932
#> 2 female      1 age        0.009599  0.03263    0.2942 0.7686
#> 3 female      2 (Intercept) 3.491     0.9037     3.863 0.0001119
#> 4 female      2 age       -0.04502    0.02548    -1.767 0.07730
#> 5 female      3 (Intercept) 0.2887    0.3412     0.8461 0.3975
#> 6 female      3 age       -0.01783   0.01361    -1.310 0.1901
#> 7 male        1 (Intercept) 0.8803    0.5275     1.669 0.09514
#> 8 male        1 age       -0.03743   0.01276    -2.934 0.003351
#> 9 male        2 (Intercept) 1.042     0.5990     1.740 0.08181
#> 10 male       2 age       -0.1122    0.02492    -4.502 0.000006724
#> 11 male       3 (Intercept) -0.7613   0.3418     -2.228 0.02591
#> 12 male       3 age       -0.03392   0.01339    -2.534 0.01129
```

Using this approach we get a nice data frame showing the logistic regression coefficients, and associated statistics (standard error, P-values, etc) for the regression of survival on age, for each combination of sex and class.