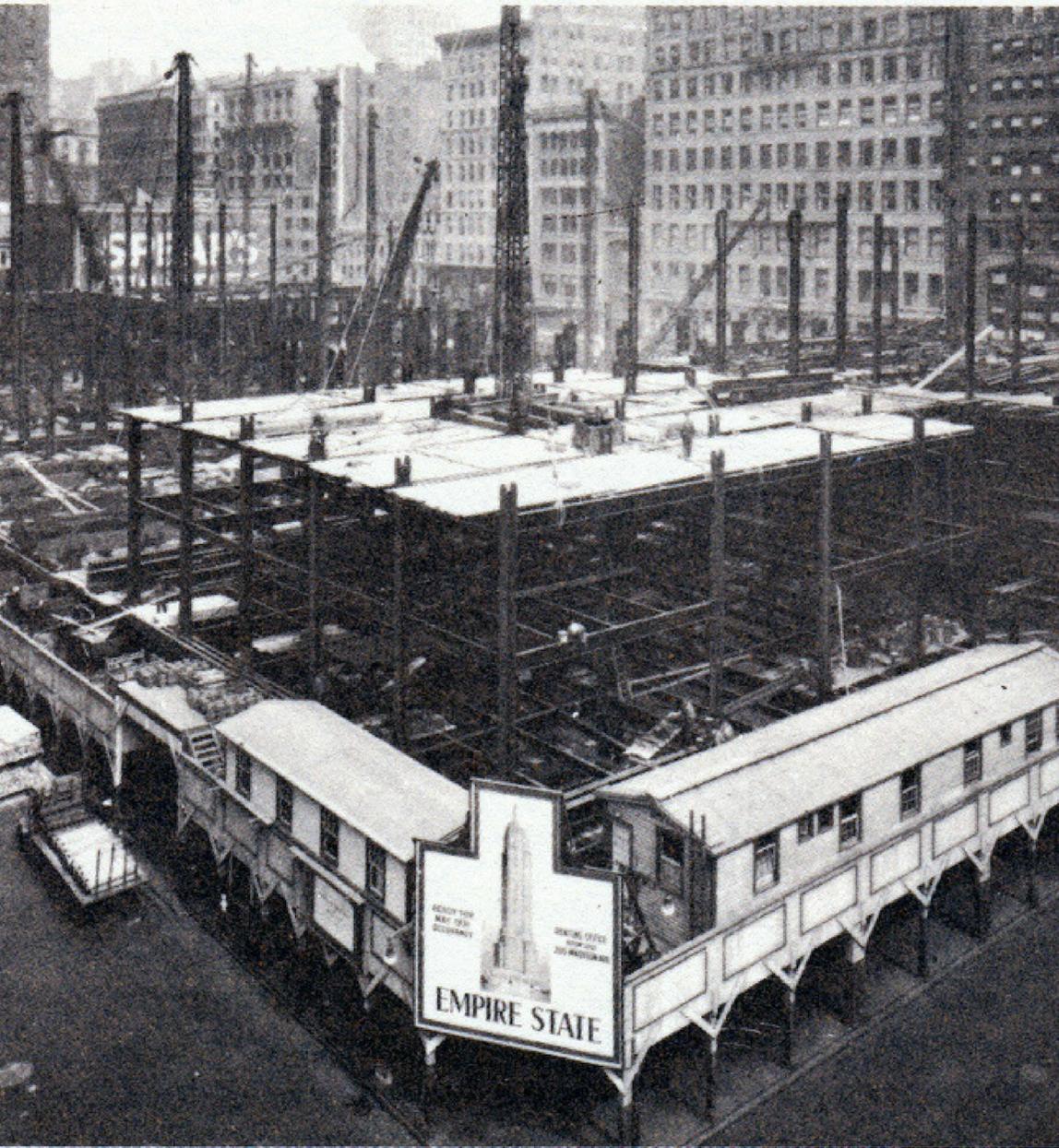


MARK HUBER



Foundations of Data Science

2019 Edition

©2019 Mark Huber
Version 2019-05-19

Contents

Contents	iii
I Dealing with data	1
1 Introduction to Data Science	2
1.1 R	4
2 Scripts and R Markdown files	7
2.1 Scripts	7
2.2 R Markdown	9
2.3 Markdown	10
2.4 Using as a notebook	12
2.5 Knitting a file from the console	13
2.6 Latex	13
3 Graphical Grammars	15
3.1 Visualization in the tidyverse	16
3.2 Aesthetic mappings	16
3.3 What went wrong?	19
3.4 Facets	20
3.5 Using multiple geometries	20
3.6 Bar charts	23
3.7 Transforming coordinates	27
3.8 Putting it all together	28
4 Advanced graphical grammars in the tidyverse	29
4.1 Visualizing marginal distributions with covariation	29
4.2 Correlogram	30
4.3 Diverging bars	32
4.4 Composition plots	33
5 Transforming data	38

5.1	The <code>dplyr</code> package	39
5.2	The <code>filter</code> function	40
5.3	Using <code>arrange</code> to order rows	43
5.4	Using <code>select</code> to pick out variables and string data	44
5.5	Using <code>mutate</code> to create new variables	45
5.6	Logical operators in R	46
5.7	A note about SQL	47
6	Creating summaries of tibbles	48
6.1	Using <code>group_by</code>	49
6.2	Using pipes to avoid intermediate variables	50
6.3	The effect of <code>NA</code> values	52
6.4	Combining groups with <code>filter</code> , <code>select</code> , and <code>mutate</code>	53
6.5	Example: average mileage and displacement by car class	55
7	Exploratory Data Analysis: Variation	58
7.1	Variation	59
7.2	Rare values	63
8	Exploratory Data analysis: Covariation	66
8.1	Categorical and continuous variables	66
8.2	Boxplots	68
8.3	Two categorical variables.	69
8.4	Patterns and modeling	71
II	Preparing data	74
9	Data Import Part I	75
9.1	Comma Separated Files	75
9.2	Parsing vectors	81
10	Data Import Part II	85
10.1	Representing text	85
11	Tidy Data	95
11.1	<code>Spread</code>	97
11.2	<code>Gather</code>	98
11.3	<code>Separate</code>	99
11.4	<code>Unite</code>	100
11.5	Missing Values	101
11.6	Cleaning data	104
12	A mathematical model of data	105

12.1	Sets	105
12.2	Keys	108
12.3	Terminology	110
12.4	History	111
13	Relational data	112
13.1	Left joins	115
13.2	Types of joins	116
13.3	No duplicate keys	116
13.4	Duplicate key values	118
13.5	Defining the factors that make up keys	118
13.6	Merge	120
14	Filtering joins and set operations	121
14.1	Joining over observations	122
14.2	Set operations on tables	123
15	Strings	126
15.1	Helpful string functions	128
15.2	Searching within strings: finite automata	131
16	Regular expressions	134
16.1	Finite Automata for regular expressions	137
16.2	Nondeterministic finite automata	139
17	Using regular expressions	142
17.1	Extracting matches	147
17.2	Keeping our matches	149
17.3	Creating vectors of strings	151
17.4	When <code>stringr</code> is not enough	152
18	Functions that create patterns	153
18.1	Splitting	153
18.2	Transforming other pattern types to regular expressions	155
18.3	Fixed	157
19	Factors	158
19.1	Factors	158
19.2	Package <code>forcats</code>	160
19.3	Ordering the levels	162
19.4	Changing the levels	165
20	Introduction to Structured Query Language (SQL)	170
20.1	Making a connection	171

20.2	SELECT	173
20.3	WHERE	176
20.4	ORDER BY	176
20.5	NULL values and logical operators	178
20.6	Transforming data	179
20.7	LIKE and NOT LIKE	180
20.8	OFFSET	181
20.9	SQL versus the tidyverse	182
21	Joining tables in SQL	184
21.1	Inner Join	185
21.2	Outer Joins	188
21.3	Self Join	189
21.4	Aggregations	190
21.5	GROUP BY	192
21.6	Set operations in SQL	194
III	Program control	195
22	Principles of the tidyverse	196
22.1	Application programming interface	196
22.2	Reusing existing structures	197
22.3	Pipes make code easier	197
22.4	Use functional programming	197
22.5	Designing the API for humans	202
23	Writing Functions in R	204
23.1	Things to keep in mind when writing functions	207
23.2	If and else	208
23.3	Arguments	211
23.4	Arbitrary numbers of arguments	212
IV	Modeling	213
24	Modeling data	214
24.1	Linear models	215
24.2	Using more than one predictor	223
24.3	<code>modelr</code>	224
25	Residuals	228
25.1	Understanding residuals	228
25.2	Notation for models	231

25.3	Continuous and categorical	235
26	Case study: predicting survival on the Titanic	239
26.1	Training data	239
26.2	Building a model	249
27	Machine learning	255
27.1	Supervised learning	257
27.2	Unsupervised learning	263
V	Explorations	265
28	Exploration: introduction to R	266
29	Exploration: Using graphical grammars in the tidyverse	277
29.1	Summary	277
29.2	Scatterplots	277
29.3	Kernel Density plots	279
29.4	Moving the legend around	282
30	Exploration: Transforming data with dplyr	287
31	Exploration: Projects in R and Tibbles	297
32	Exploration: tidying data with tidyr	307
33	Exploration: Relational data in the tidyverse	317
34	Exploration: Working with strings and stringr	324
35	Exploration: MySQL	335
36	Exploration: Modeling Data	344
37	Exploration: Support vector machines with svm	354
	Bibliography	367
	Index	368

Purpose These notes cover a one semester course in the foundations of data science. Students will learn how to import and tidy data in preparation for analysis of the data. They will then learn the basics of data modeling, including how to transform and visualize data. Finally, they will learn how to communicate their findings to the outside world.

Organization The software used in this course is mostly R.

This course follows closely the book *R for Data Science* by Hadley Wickham and Garrett Grolemund [1]. Their book is open access, and can be found at <https://r4ds.had.co.nz/>.

Wickham and Grolemund have a companion book for teaching R, and so their book assumes the reader has some basic knowledge of R, file systems and programming.

This course also assumes that the student is familiar with the basics of programming in a language such as Python. However, no knowledge of R or RStudio is assumed. Therefore right off the bat we begin with a simple introduction to R, followed by a short introduction to R Markdown.

When I teach this course, I spend about two-thirds of my time lecturing, and about one-third on self-guided explorations. I find that ensures that everyone can use packages and R by the end. When I left such things to the homeworks it did not go well.

Most of the lectures are derived from Grolemund & Wickham's book. Some of the explorations also come from the book, others are reifications of blog posts and tutorials that have been posted. The data science community is truly a wonderful place. The amount of sharing that folks do is great, and contributes to the fast pace of growth.

The material in this book covers one semester for me, with each chapter (and exploration) covering about 50 minutes.

Part I

DEALING WITH DATA

Introduction to Data Science

Summary

- A **data scientist** can get data into an effective computer readable form, learn about the data through transformation, visualization, and modeling, and communicate their results to the outside world.
- **R** is a statistical programming language.
- **RStudio** is an IDE for R that allows us to easily use R through the console, scripts, and R Markdown.

Much of these notes follow the excellent text *R for Data Science* by Hadley Wickham and Garrett Golemund. Their book is open access and online and can be found at

<https://r4ds.had.co.nz/>.

Definition 1

Data science consists of the methods and tools for collecting and studying data, with a goal of making informed decisions.

The data sets studied depend on the domain in which we are working. An analyst studying images on the web will have very different data from an economist studying time series data of interest rates.

However, even with all the different types of data there are some common ideas that apply in all situations. The basic tasks that face any data scientist include the following.

1. Get the data into a form that can be read easily by a computer and other humans.
2. Understand what the data is telling us.

3. Communicate what the data says to the rest of the world.

One way of framing this task is with a flowchart. While the term *Big Data* has recently become popular, in fact the data sets studied by statisticians have always been large, starting with census data from centuries ago. Fortunately, today we do not have to deal with millions of forms and slips of paper. Instead we can bring the data into a form that can be analyzed by a computer. This process of making the data computer readable we will call the *import* step.

There are certain conventions in mathematics and statistics. For instance, given variables x and y , the variable x is usually plotted on the horizontal axis, and y is plotted on the vertical. By using this convention, a mathematician makes it easier for their audience to understand new material.

In the same way, there is a standard way to format and present data. Putting the data into a form that follows these conventions we will call *tidying* the data.

Once we have the data in place, we begin the task of understanding it. This takes on several different forms. Our brains are great at picking out visual patterns, and so a popular first step for understanding how data behaves is to use *visualization* tools.

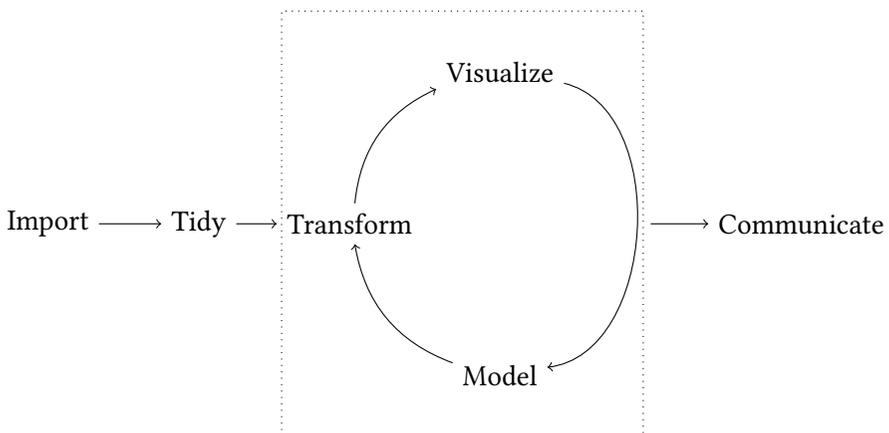
Another aspect of understanding data is building a mathematical *model* of the data. In order to make the tasks of visualization and modeling easier, often we *transform* the data. This can involve picking out the most important parts, or projecting the data onto a plane or curve, or any activity that helps us to make sense of the data set.

These three activities are not done in isolation, but build off of each other. An early visualization might make a researcher realize that their data lies in a subspace of the available variables. By transforming the data by projecting onto that subspace, a new visualization might see further patterns that were hidden in the original data.

A specific model might lead to a better visualization, which gives the user a transformation of the data that needs a new model. And so it goes, one technique feeding into the other until the complexities of the data is understood.

At this point the goal is to be able to communicate what the data tells us to the rest of the world, and *communication* is typically the final step in the process.

This can be summarized as



1.1 R

The programming language for this text will be R. Most data science today is done in R, Python, C, C++, and a few other languages. With proper package support, each is capable on its own of handling most data analysis tasks. R and Python are good places to start because they are interpreted languages that have a console where commands can be tested directly, helping a user to build an intuitive understanding of the language.

Definition 2

R is a programming environment designed for statistical analysis.

The R language interpreter can be downloaded for free from

<https://cloud.r-project.org/>

for Windows, Mac, and Linux. We will also be using an Integrated Development Environment (IDE) for R called RStudio. This IDE is also Open Source, and can be downloaded for free from <https://www.rstudio.com/products/rstudio/download/>.

Definition 3

An **Integrated Development Environment** or IDE is a software program that brings together the tools you need to work with a programming environment effectively.

R fact 1

The most popular IDE for R is RStudio.

We will be using R through RStudio in three ways.

1. **Console.** You can type commands in R directly into the console the same way you can in Python.
2. **Scripts.** Collections of commands for R are called *scripts*, and these have file extension `.R`.
3. **R Markdown.** A markup language uses tags to create a professional looking document. Markdown is a very simple document preparation system, and R Markdown allows the user to easily incorporate R code into their document. These files typically end in extension `.Rmd`.

Let's start by looking at the console.

The R console

Definition 4

A programming language is **interpreted** if the commands of the language are translated into machine code each and every time the commands are executed. A programming language is **compiled** if there is a step where the commands are first translated into machine code. Each time the commands are executed, the compiled machine code is then run.

R fact 2

R is an interpreted language.

Definition 5

A **console** in an interpreted programming language accepts and executes commands one at a time.

When you first start RStudio, in the lower left corner will be the console. In the console you can try out commands individually. The assignment operator in R is `<-`. So for instance the commands

```
x <- 4
y <- 5
x + y
```

returns

```
[1] 9
```

The `[1]` indicates the the result only has one number, the `9` is the actual result.

Some variables are already defined in R. So for instance, if you type

```
cars
```

into the console, it will give you all 50 lines of data from the `cars` data set. To get an idea of what is in this data set, we can use the `head` command to get the first few lines of `cars`.

```
head(cars)
```

gives us the first 7 lines of data, together with the headings for the data, `speed` and `dist`.

The `?` operator opens the help within R. Using

```
?cars
```

in Rstudio opens up the help in the lower right corner window (in the default setup) and tells us that this variable represents speed and stopping distance data for a number of cars from the 1920's.

If we use

```
summary(cars)
```

we are treated to a basic statistical analysis of the data in the `cars` data set.

Definition 6

A **statistic** is any function of the data.

In the cars data summary, we are told (for instance), the minimum speed value among all the cars. This is 4.00 for cars. We are also told the sample mean, which is the sum of the values divided by the number of values. This is 15.4, and is an example of a *measure of central tendency*.

Definition 7

The **sample average** of values x_1, \dots, x_n is

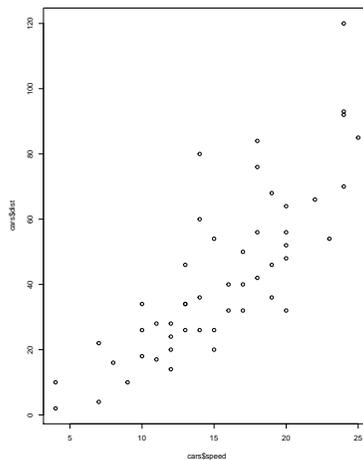
$$\bar{x} = \frac{x_1 + \dots + x_n}{n}.$$

The sample average of the speed of the cars is 15.4, and the average of the stopping time is 42.98. What these statistics do not tell us is how the speed and stopping distance are related.

To understand this, it is helpful to have a way of visualizing the relationship. This simplest thing we can do is just make a plot of the distance values versus the speed values. The `$` operator allows us to pick out specific pieces of a data set. So the following command plots the distance against the speed.

```
plot(cars$speed, cars$dist)
```

The result looks like this:



The beauty of a visualization like this is that it immediately makes apparent the relationship between the speed and the stopping distance: as the car goes faster, the stopping distance tends to be greater.

Scripts and R Markdown files

Summary

- A **script** is a set of commands that can be executed by R. Use a **.R** file extension for scripts.
- A **R Markdown** file can be easily transformed into a standard, professional looking document that includes R code execution. Use a **.Rmd** file extension for these files.
- Inside RStudio, an R Markdown file acts as a **notebook**, where code chunks can be executed individually, and the results displayed.

R is not just a programming language, but also gives a way of presenting results that allow analyses to be checked and modified. This can be accomplished by using *scripts* to record the commands you use, and *R Markdown* to build a publishable work.

2.1 Scripts

Definition 8

In R, a **script** is a collection of commands that you want R to execute.

In RStudio, we can create a file for a script by using the menu command

File ► New File ► R Script.

By default, RStudio will open up a window in the upper left portion of its area. Commands can then be typed into this area. For instance, suppose I put

```
x <- 4
y <- 5
x + y
```

Note that nothing happens, except the lines of the file get numbered 1, 2, and 3 (in the gray area to the left of the code) as you type the lines.

To tell R to execute these lines, we can use the `source` function. There is a shortcut that is very helpful. Above the code there will be a checkbox with the label ‘Source On Save’. When you check this box, every time you save the file, the script will automatically be executed in the console.

Let’s give it a try. First, be sure the checkbox is checked. Then use the menu command

File ► Save

to save the file (there is also a shortcut for this command that is operating system dependent.)

Note that upon saving the file, in the console below the `source` command has been given.

We can check that the commands actually executed by typing

```
x
y
```

directly into the console.

R fact 3

If you do not specify a file extension when saving your file, the default is `.R`, and we will use this convention for R script files throughout.

However, it seems like the `x + y` command did not actually do anything! This is because commands that would normally print in the console do not print when executed as part of a script. A value can be forced to print in a script using the `print` command. Try changing line 3 of your script to

```
print(x + y)
```

and source the script again. This time you should get output

```
[1] 9
```

just as if you had typed `x + y` into the console.

Why you should use scripts Scripts act as a scientific record of your analysis. They record *exactly* what you did and how you got your data.

Now, if someone else (say a researcher following your work) is trying to extend your analysis or apply it to another area, they do not have to guess what exactly you did. They can see each and every step exactly. That is why it is important to keep track of your procedures precisely.

2.2 R Markdown

One of the most important aspects of data analysis is communicating what you found and how you found it to others. Scripts are a good start to this process, and now we consider how you might report your results to others.

You want your communications to have the following good properties.

1. **Complete.** Someone reading your work should be able to replicate what you did.
2. **Compatible.** You want to use a standard format, HTML, pdf, or Markdown to communicate your results so that they can be viewed by the widest possible (perhaps non tech-savvy) audience.
3. **Professional.** You want output that is neat, well-organized, and looks good.

Definition 9

Typesetting is the process of arranging text for publication.

We would like a way of typesetting our results that is pleasing to the eye. This is usually accomplished using a *markup language*. These are a set of commands that allow you to *emphasize* words, add a bit of **color**, start new sections, subsections, and paragraphs. Markup languages also can be used to create a list of bullet points or numbered points.

Definition 10

A **markup language** uses commands to determine the typesetting for a document.

A word processor such as Microsoft Word is often called *WYSIWYG* which stands for *what you see is what you get*. When you type commands into such a program, you directly see what the output will be.

On the other hand, in a markup language you enter simple text that could be typed using only the standard keys on a typewriter. You use commands in order to indicate when a word should be emphasized or is a section heading. The software then takes the result and builds a typeset document for you. Usually you do not see the final result until the software has completed its work.

The most commonly used markup language today is HTML, which stands for *Hypertext Markup Language* and it the language that webpages are usually written in. All major web browsers can interpret and display HTML files.

Definition 11

HTML (Hypertext Markup Language) is the primary markup language used for publishing on websites. It is an interpreted language.

In mathematics and the sciences, another commonly used markup language is \LaTeX , because it is very good at typesetting documents that include mathematics. This ebook was typeset using \LaTeX .

Definition 12

L^AT_EX is a markup language that is extensively used in scientific and mathematical fields. It is a compiled language.

Most word processors have an internal markup language, but since the user usually cannot see it, they cannot directly make changes. The advantage of a markup language is that you can specify what you want to happen in a general sense, and then the language takes care of the details. For instance, if you say you want a new chapter, the markup language will take care of the numbering and table of contents for you without the need for you to intervene and specify exactly the font and style of these types of elements.

2.3 *Markdown*

Often the full control that comes with using a markup language is overkill. For this reason, John Gruber created a light markup language that emphasized ease of use and readability over the ability to do any possible thing. The result was *Markdown*. (Get it? **Markdown** is a lighter version of a **markup** language. That's computer science humor for you in a nutshell.)

Definition 13

Markdown is a markup language that is designed with few commands to be easy to use.

The markdown language has been implemented in many different formats, the one that we will use here is the version implemented by R, called R Markdown. If you want to learn more about how R Studio incorporates R Markdown, go to <https://rmarkdown.rstudio.com/>

We can start a new R markdown file with

File ► New File ► R Markdown

which will create a new document similar to the way that we created a new script.

R fact 4

The default file extension for R Markdown files is `.Rmd`.

Here's an example of an R markdown file.

```
---
title: "Example R Markdown file"
author: "Mark Huber"
date: "February 16, 2019"
output: html_document
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```

```
```
```

```
# Using Headings
```

If you want to indicate that words should be part of a heading, you start the line with the # character.

```
## A section
```

The next level of heading starts a line with ##. There are also headings that start with ### and ####. Simple things like *emphasis* or **bold** text can also be easily created with a Markdown file. To put a word in emphasis, then put a * character on either end of the word. For example `*wow*` in the text looks like *wow* in the final output.

```
## Inline code
```

When you want something to be highlighted as code in R Markdown, you surround the code with left leaning quote marks. So `\`test\`` renders as ``test``.

```
## Code Chunks
```

Longer pieces of code can be included in *code chunks*. These are lines of code that gets executed in the R programming environment. For instance, we can use the following code chunk on the built in variable `'faithful'` to plot the waiting times between eruptions against the length of the eruption.

```
```${r}  
plot(faithful$eruptions, faithful$waiting)
```
```

```
## Knitting an R Markdown file
```

The process of turning the text file into a document is called *knitting* for R Markdown files. If you are working in RStudio, and your text file has an `.Rmd` extension, there will appear a button labeled **Knit** which you can press to create the output file.

```
# Resources
```

You can find out more about R Markdown at <http://rmarkdown.rstudio.com>.

Some key points:

1. The heading at the beginning marked out by `---` is called a YAML header. YAML

stands for **Y**AML **A**in't **M**arkup **L**anguage. This is an example of a recursive acronym. The contents of the header such as **title** and **author** should be self-explanatory. As the acronym tells us, YAML is not a markup language, instead it is considered a *data serialization language*.

2. In the main file, use # to start a new section.
3. Use ## to start a new subsection
4. Use ``` to mark out blocks of code. The `echo=FALSE` parameter indicates that the code block should not be listed in the output, but the results of running the command should be.

Definition 14

Serialization puts data in a form where the data can be read without commands by relying on the order in which data items occur.

Definition 15

YAML (YAML Ain't Markup Language) is a serialization language that is used for the header of an R Markdown file.

Note that in the interface to R Studio there is a button above the file called **Knit**. Press this button to turn the R Markdown file into an HTML file.

Our script in R Markdown

Mark Huber

November 15, 2018

1 R Markdown

This is a document written in R Markdown. Notice that we started a new section very simply by putting a # character. To create a subsection, we can use ##. More examples, templates and instructions for R Markdown can be found at <http://rmarkdown.rstudio.com>.

1.1 This is a subsection

To take an R Markdown document in R Studio and create a new document, use the Knit button. This button takes the Markdown document, and converts it to a different format or markup language. Because in the preamble (called the YAML (YAML is not a markup language) heading) it says output: html_document, the final output created by knit will be in HTML for this document.

1.2 Putting R code inside an R Markdown document

When you click the Knit button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

```
x <- 4
y <- 5
print(x + y)
```

```
## [1] 9
```

2.4 Using as a notebook

Going back to the .Rmd file in RStudio, notice that any code chunk has a little green arrow next to it.

```

26- ...[r code]
27 x <- 4
28 y <- 5
29 print(x + y)
30

```

Pressing that green arrow immediately executes the code chunk in R by running these commands in the console. It then displays the result of the command in the file window.

```

26- ...[r code]
27 x <- 4
28 y <- 5
29 print(x + y)
30

```

[1] 9

This ability to execute code chunks inside of a document is usually called using the file as a *notebook*.

The symbol just to the left of the little green arrow will execute the current code chunk and all the chunks above it, which can be helpful if it depends on previous definitions.

For simple analyses, there is no need to write scripts, only R Markdown files.

2.5 Knitting a file from the console

Just as the `source` command can be used to execute a script using the console, the `render` command can be used from the console to turn a `.Rmd` file into other file types. For instance, typing

```

library(rmarkdown)
render("example.Rmd")

```

into the console will transform the `example.Rmd` file into the file type specified in the YAML header.

2.6 Latex

When you are writing papers and descriptions in a social or physical science, you often need to add in mathematical equations and definitions. The most popular typesetting program in the scientific community for doing this is called \LaTeX . Fortunately, you do not need to learn all of \LaTeX , as R Markdown allows you to use the most important \LaTeX commands directly. For instance, suppose we added to our previous document the following code.

```
## LaTeX examples
```

This is an example of `*inline mathematics*`: `$a^2+b^2=c^2$`. In this type of mathematics, the equation is presented in the middle of a line of text.

The second kind of mathematics is `*display mathematics*`, which is written like

```

\[
a^2 + b^2 = c^2.
\]

```

This is the same statement, but now it appears on its own line of the document.

The result looks like

1.4 LaTeX examples

This is an example of inline mathematics: $a^2 + b^2 = c^2$. This is an example of *inline mathematics*, the mathematics is presented in the middle of a line of text.

The second kind of mathematics is *display mathematics*, which is written like

$$a^2 + b^2 = c^2.$$

This is the same statement, but now it appears on its own line of the document.

Graphical Grammars

Summary

- The **tidyverse** is a collection of packages in R.
- An important part of the **tidyverse** is the **ggplot2** package, which includes commands for a **grammar of graphics**.
 - The grammar of graphics uses the **ggplot** function to create a canvas upon which we will place graphical elements.
 - Various functions that start with **geom_** then are used to place various graphical elements on the canvas.

Putting data into a standard form is often known as *tidying* the data. The advantage of having tidy data is that then standard programs can be used to analyze the data.

Here we will be using a set of packages known collectively as the *tidyverse*.

Definition 16

A **package** or **library** is a collection of functions for a programming environment with a common theme.

The tidyverse actually consists of several packages intended to help visualize, transform, explore, read, and model data. If the tidyverse is not already installed on your system, you can install them with the command

```
install.packages('tidyverse')
```

Once the packages are installed, using

```
library(tidyverse)
```

will load the packages into your current R session so that their functions can be used.

R fact 5

Use `install.packages` to install a particular package to your installation of R.

R fact 6

If a package is installed, you still must use `library('package.name')` before you can call any of its functions in the console or your code.

Note that you should only have to install the packages once, but you will have to load the library every time you wish to use it.

3.1 Visualization in the tidyverse

We start with the fun part: visualization. As we have seen, R has a command called `plot`, but here we will be using a more advanced version called `ggplot`. The gg prefix stands for grammar of graphics, and essentially means that the plotting commands form their own miniature programming language. There are commands for putting the *data* into the plot, and commands for setting the *aesthetics* of the plot.

Definition 17

A **grammar of graphics** is a set of tools for building graphics by adding components and transformations layer by layer.

Let's start by trying this with the cars data from before. If you are working with a new session, remember that you have to load in the tidyverse commands with `tidyverse` so that R knows how to run the functions in the package.

```
ggplot(data=cars) + geom_point(mapping=aes(x=speed, y=dist))
```

We set up two things in this command. The first part, `ggplot(data=cars)` tells R that we are working within the `cars` data set from last time. So we will not need the `$` operators from last time to indicate variables within the data frame.

The second part `geom_point` puts the actual points on the plot, while `aes` tells what aesthetic should be used.

3.2 Aesthetic mappings

Aesthetics tell the `geom` what data to use in building objects.

Definition 18

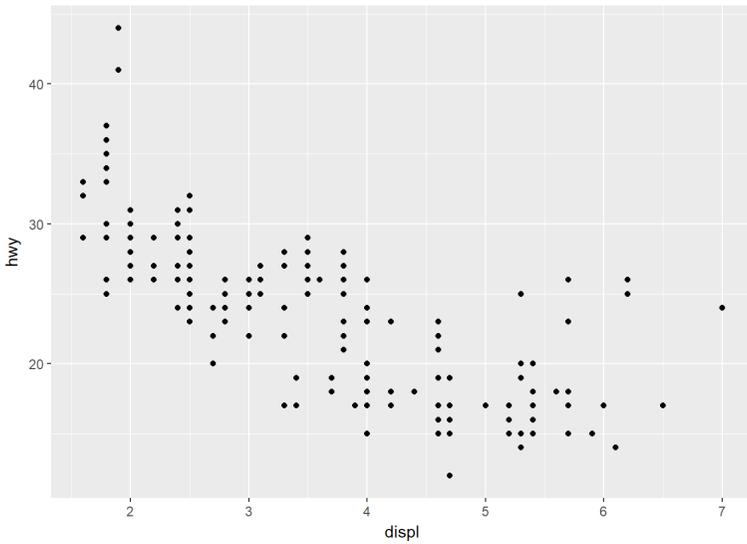
Aesthetic mappings describe how variables in the data are mapped to visual properties

(See section 3.2 of Wickham and Grolemand.) Now let's look at some of the different mappings we can tackle. We will get a feel for this using a variable `ggplot2::mpg` that is built into the `ggplot` package. As usual, use `?mpg` to bring up the help on the package,

where we see that this data set contains mileage information on 234 cars from 38 models spanning 1999 to 2008.

Let's try the same plot from earlier for the `mpg` data set.

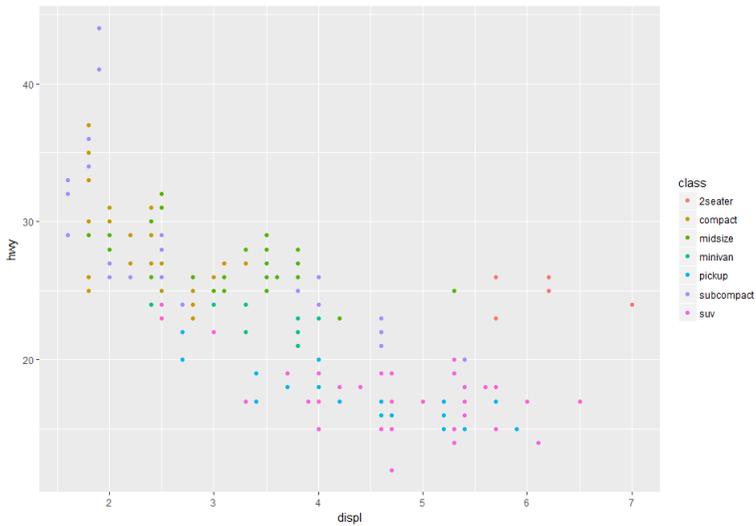
```
ggplot(data=mpg) +
  geom_point(mapping=aes(x = displ, y = hwy))
```



From the data, we see that as the displacement of the engine (essentially the engine size) grows, the highway mileage tends to go down.

However, there is a weird exception among the points. Most of the data clumps together in the same spot, but there are some data points near the right hand side that seem higher than the main body of points. Perhaps those points represent a special type of car? To add that dimension of the data to the graph, We will use color to show the class of the car.

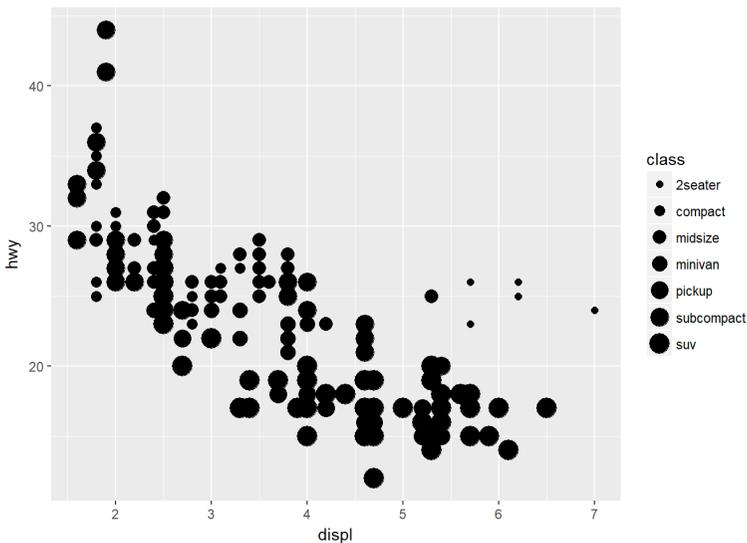
```
ggplot(data=mpg) +
  geom_point(mapping=aes(x = displ, y = hwy, color = class))
```



With the colors in place, it becomes clear that the plots that are off from the rest mostly belong to 2-seater cars.

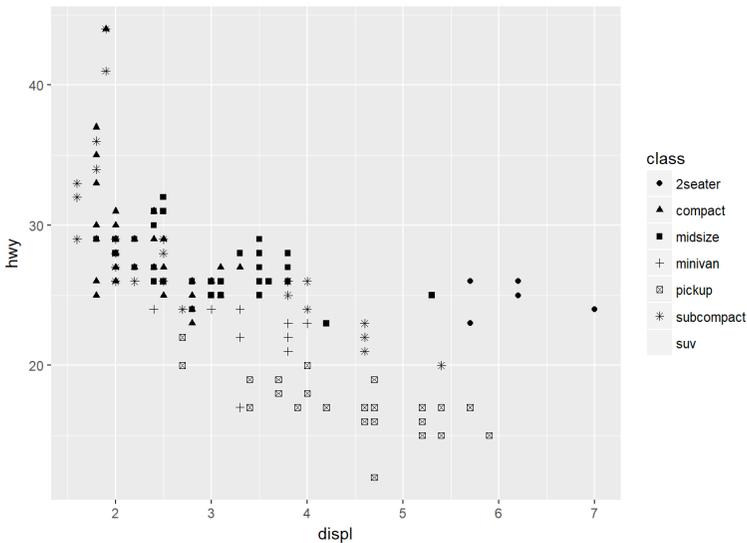
We have lots of choices beyond color here. For instance, we could have used the size of the points to denote the class.

```
ggplot(data=mpg) +  
  geom_point(mapping=aes(x = displ, y = hwy, size = class))
```



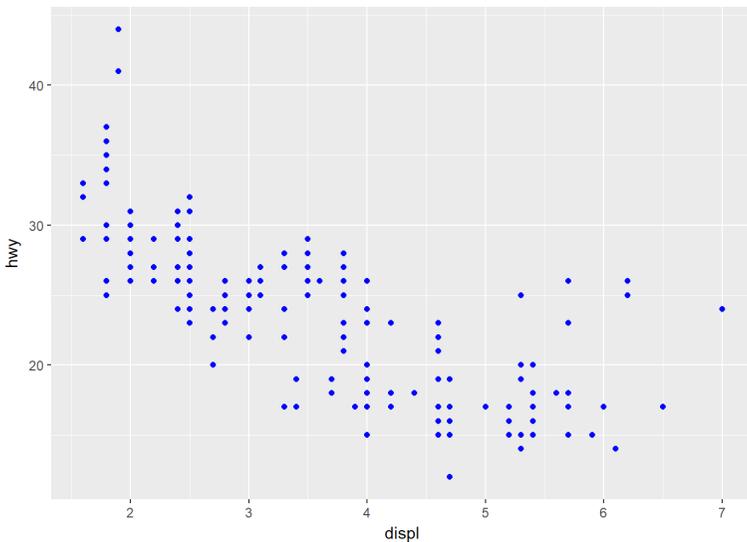
Or you can change the shape of the points by class.

```
ggplot(data=mpg) +  
  geom_point(mapping=aes(x = displ, y = hwy, shape = class))
```



Of course, we can also use the aesthetic to change all the points to the same color.

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy), color = "blue")
```



3.3 What went wrong?

It is very easy for commands in R to go wrong. A misplaced parenthesis or comma and you might get an error message, or even worse is when the command runs without an error, but does not do what you expected it to.

Usually the console in R starts with a `>` character, indicating that it is ready to accept a new line of input. When you forget to close a right parenthesis `)`, the R console will

respond by starting the next line with a + character, indicating that the console wishes for you to add to the previous line and finish your command.

R fact 7

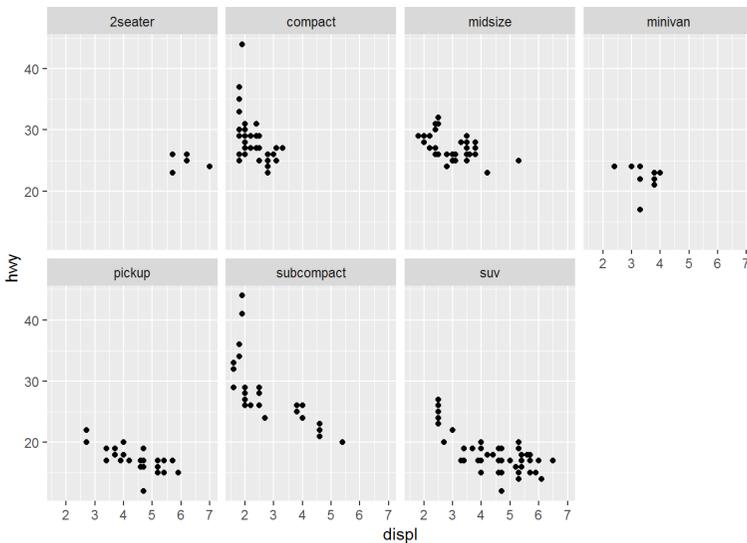
In R you can always get help for a function by using `?function.name`.

3.4 Facets

Previously we used color, size, and shape to tell the different points apart. We can also break the plot into multiple plots using *facets*.

Consider

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_wrap(~ class, nrow = 2)
```

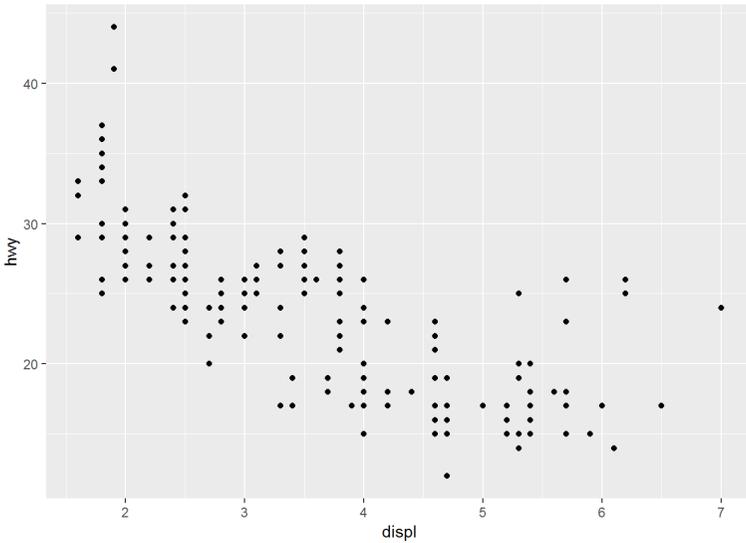


As you see, by using `facet_wrap`, we split the plot into multiple plots based on the class of the vehicle.

3.5 Using multiple geometries

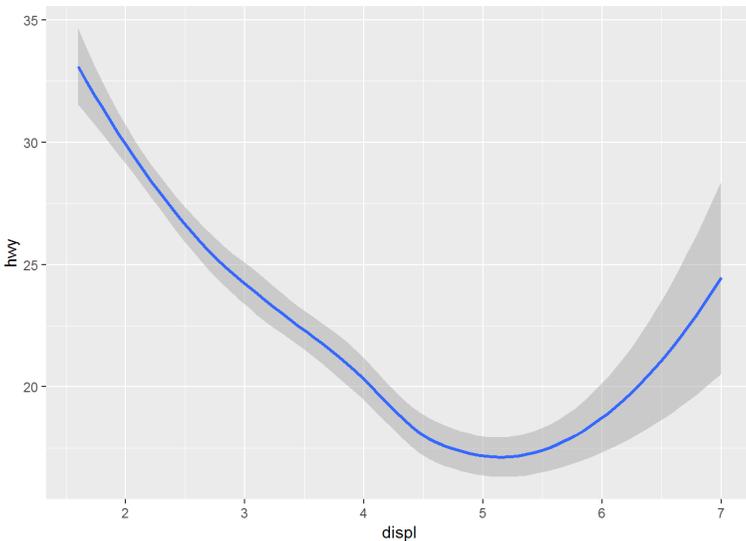
A `geom` is a geometric object, it represents a way of looking at data. In the last chapter, we primarily used the point geom for data. Here each x and y value was represented by a small black dot.

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy))
```



We could replace the points by a smooth line geom that attempts to capture the position of the points.

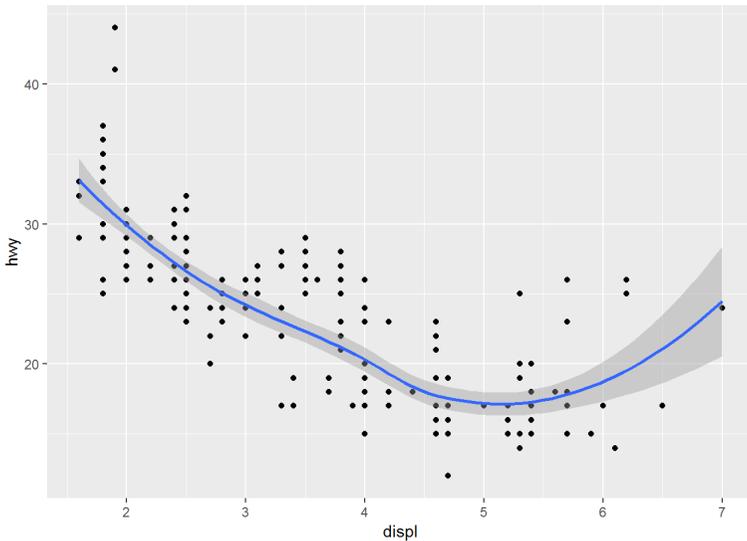
```
ggplot(data = mpg) +  
  geom_smooth(mapping = aes(x = displ, y = hwy))
```



Both the points and the line used a `mapping` argument, but not every aesthetic works with every geom. For instance, the `shape` aesthetic works with points, but not with lines. The `linetype` aesthetic works with lines, but not with points.

These different ways of viewing the data can become even more effective when we put them into the same plot.

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  geom_smooth(mapping = aes(x = displ, y = hwy))
```

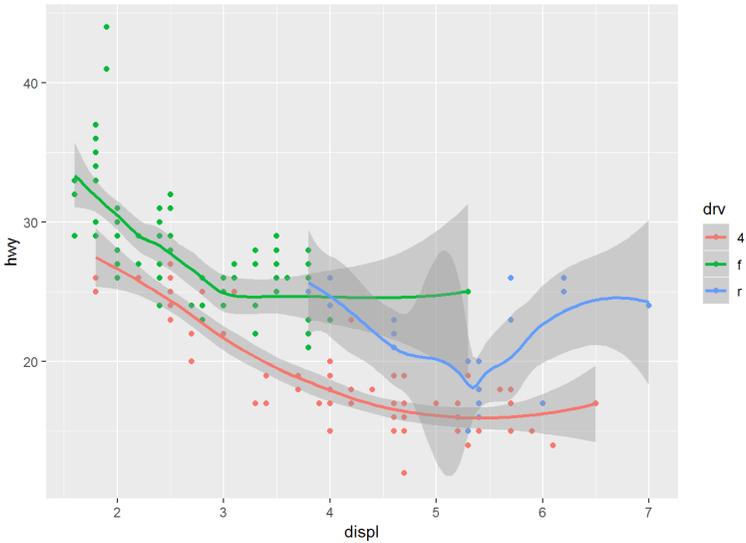


Because both geoms used the same aesthetic, we could place it into the initial `ggplot` and end up with the same plot.

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point() +
  geom_smooth()
```

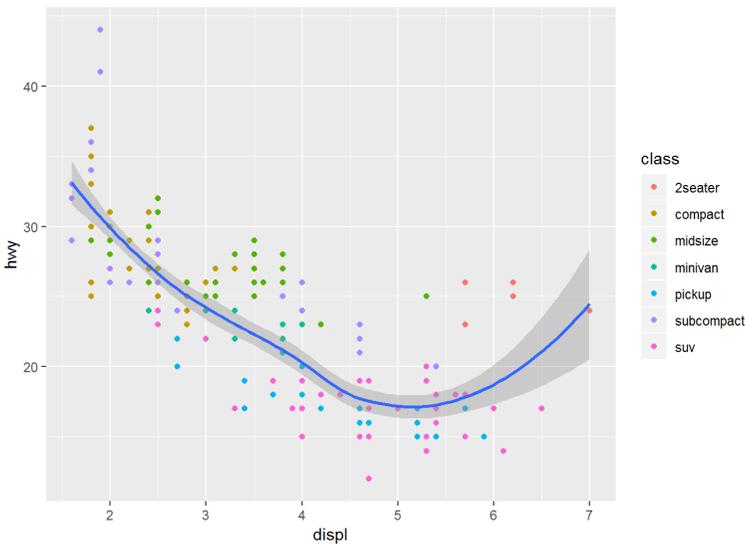
As with the point geoms, we can break down the lines into different classes. For instance, if we break the data into three groups by the type of drive the cars use, we get something like this.

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy, color = drv
  )) +
  geom_point() +
  geom_smooth()
```



We can also apply different aesthetics to the different geoms. For instance, we can color the points by car class and leave the line geom as blue.

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point(mapping = aes(color = class)) +
  geom_smooth()
```

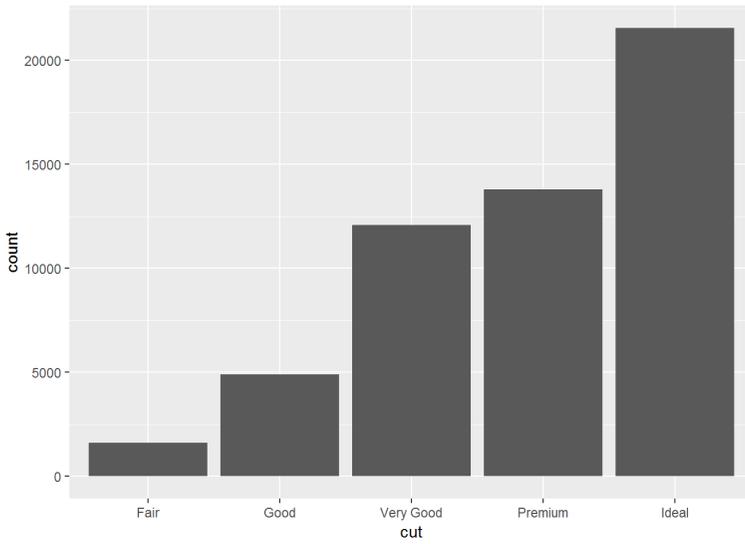


3.6 Bar charts

Another type of geom is `bar`, which as you might have guessed creates a bar chart.

The variable `diamonds` (part of the `ggplot2` package) contains data on almost 54 000 different diamonds. Consider the following bar chart for the data, which shows the various numbers associated with each quality of cut.

```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut))
```



Of course, this particular geom is not just visualizing the data, it is also computing statistics of the data. Each of these counts, for Fair, Good, and so on, is a function of the data. Hence this bar plot is a way of summarizing five statistics of the data at once, in a way that immediately gives us a relative sense of their size.

When you use `?geom_bar`, you are told that one of the parameters of the function is `stat`. In fact, it says that `stat = "count"`, which means that the statistic that the geom is using is the `count` statistic.

R fact 8

If a parameter in a function has the form `parameter = value`, then `value` is the default value given to the parameter if the parameter is not explicitly set by the user.

Every geom has a default statistic that it uses. In the same way, every statistic has a default geom! In the case of `stat_count`, the default geom is `geom_bar`. Hence the following code generates the same bar plot as before.

```
ggplot(data = diamonds) +
  stat_count(mapping = aes(x = cut))
```

To get more detail about a statistic (including its default geom), just use the help (`?statistic`).

As with the last geom, there are plenty of ways to modify the basic defaults. For instance, the following plots the proportion counts rather than the raw counts.

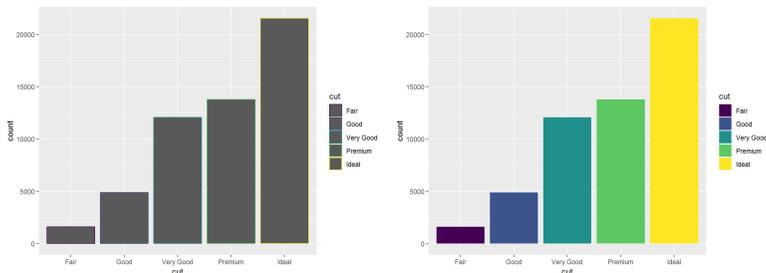
```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, y = ..prop.., group = 1))
```

This modification picks out certain pieces of the `stat_summary` of diamonds to plot.

```
ggplot(data = diamonds) +
  stat_summary(
    mapping = aes(x = cut, y = depth),
    fun.ymin = min,
    fun.ymax = max,
    fun.y = median
  )
```

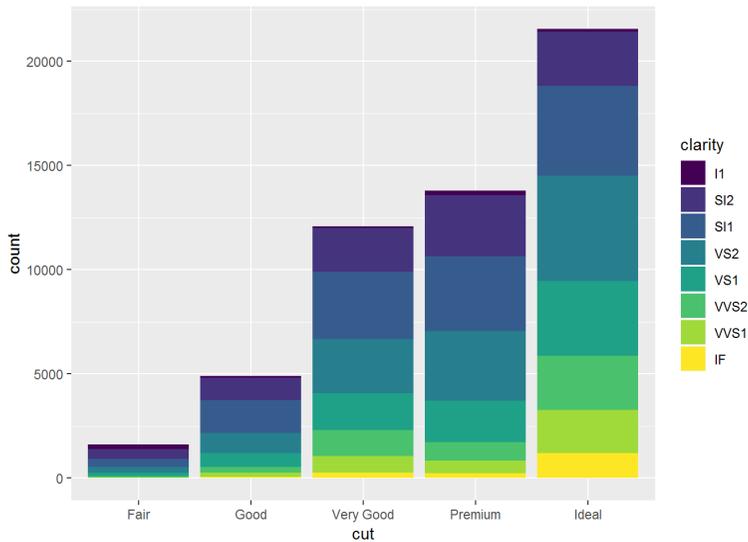
Adding color One thing to note with bar plots. The option `colour` now only colors the borders of the bars. To color the entire bar, use the `fill` option.

```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, colour = cut))
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, fill = cut))
```



Often, we need our bars broken down by another attribute. For instance, suppose for each cut of diamond, we want to know what fraction of each cut corresponds to different levels of clarity. Rather than base our fill color on the cut, base it instead on the clarity.

```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, fill = clarity))
```



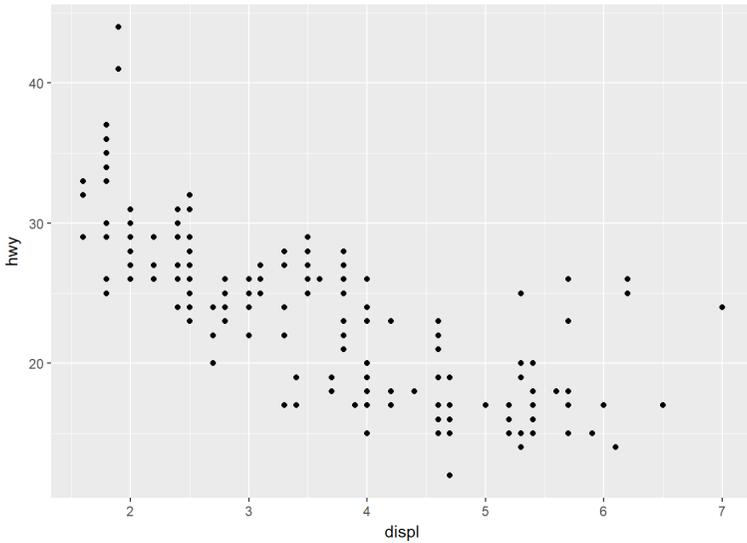
The `geom_bar` has a parameter `position` that defaults to "stack" which is what we saw in the plot above. Note that this stack allows us to effectively look at three dimensions of the data in a two dimensional plot. The color serves as the third dimension, and is an effective way of making a two-dimensional graphic serve as a tool for seeing three-dimensional data.

Other `position` values to try include

- **dodge** This places colored bars side by side for easy relative comparison within groups.
- **fill** This forces each bar to be height 1: that way you can compare the fraction of each type that has the subtype used with the `fill` command.

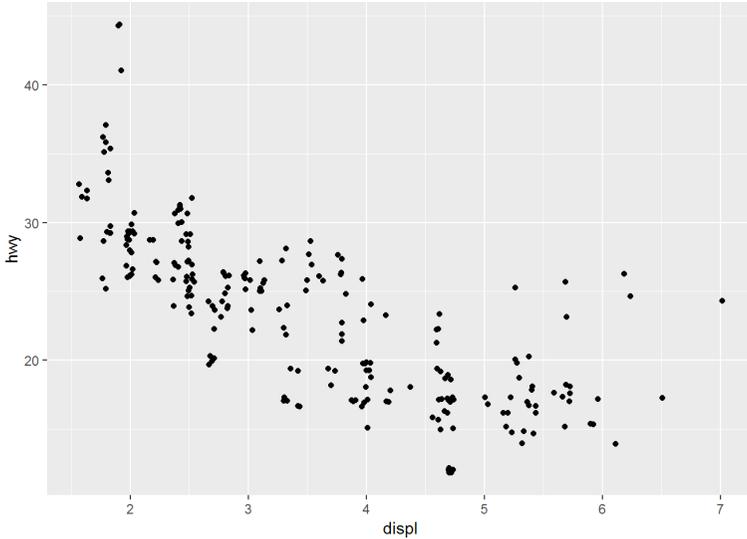
The `position` command can also be used with the scatterplots from earlier. One particularly useful option is `jitter`. This randomly moves the point around, and is helpful when points land right on top of one another.

No jitter:



Now add some jitter:

```
ggplot(data = mpg) +  
  geom_point(mapping = aes(x = displ, y = hwy), position = "jitter")
```



We now have a way of seeing which points actually were hiding multiple points underneath.

3.7 Transforming coordinates

There are some useful functions for dealing with coordinate systems.

- `coord_flip()` flips the x and y axes, which is very helpful in dealing with long label names. Just + this function to your ggplot to flip.

```
ggplot(data = mpg, mapping = aes(x = class, y = hwy)) +
  geom_boxplot() +
  coord_flip()
```

- `coord_quickmap()` sets the aspect ratio to the correct value when your data is coming from map data.

```
states <- map_data("state")
ca <- subset(states, region=="california")

ggplot(ca, aes(long, lat, group = group)) +
  geom_polygon(fill = "gold", color = "black") +
  coord_quickmap()
```

- `coord_polar` Plots points as if they are using polar coordinates.

3.8 Putting it all together

What we have seen is that visualization tools are not just about graphics, but they also calculate statistics from the data set. A good visualization will accomplish several things.

- Pick the aspects of the data set that are important to us.
- Allow us to see multiple dimensions simultaneously on a two dimensional graphic.
- Allow comparisons across different characteristics of our data set.

A general template for `ggplot` can be written as follows.

```
ggplot(data = <DATA>) +
  <GEOM_FUNCTION>(
    mapping = aes(<MAPPINGS>),
    stat = <STAT>,
    position = <POSITION>
  ) +
  <COORDINATE_FUNCTION> +
  <FACET_FUNCTION>
```

Advanced graphical grammars in the tidyverse

Summary There are a lot of more advanced plots that `ggplot` can help with, and many plots that have developed and can be accessed through other packages as well. Some useful plotting capabilities include

- Marginal annotations.
- Correlograms.
- Area charts.
- Diverging bars.
- Composition plots such as pie charts.

For those who are interested, most of these plots can be found at <http://r-statistics.co/Top50-Ggplot2-Visualizations-MasterList-R-Code.html>.

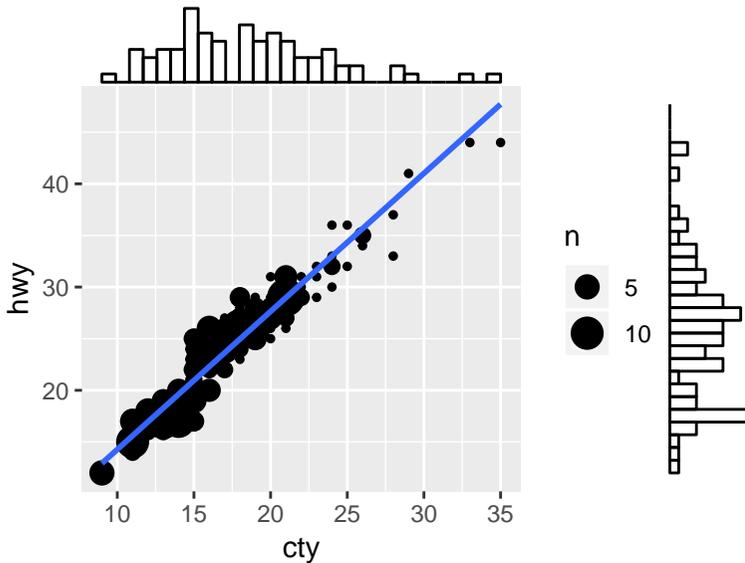
4.1 Visualizing marginal distributions with covariation

When using a scatterplot to visualize relationships between two variables, it helps to understand roughly what the marginal distributions are doing at the same time. With the `ggExtra` package, we can make this happen. Consider the following code. The last call to `ggMarginal` is what puts the extra information on the margins of the plot.

```
# load package and data
# install.packages("ggExtra")
library(ggplot2)
library(ggExtra)
```

```
# Scatterplot
g <- ggplot(mpg, aes(cty, hwy)) +
  geom_count() +
  geom_smooth(method = "lm", se = FALSE)

ggMarginal(g, type = "histogram", fill="transparent")
```



```
# ggMarginal(g, type = "boxplot", fill="transparent")
# ggMarginal(g, type = "density", fill="transparent")
```

4.2 Correlogram

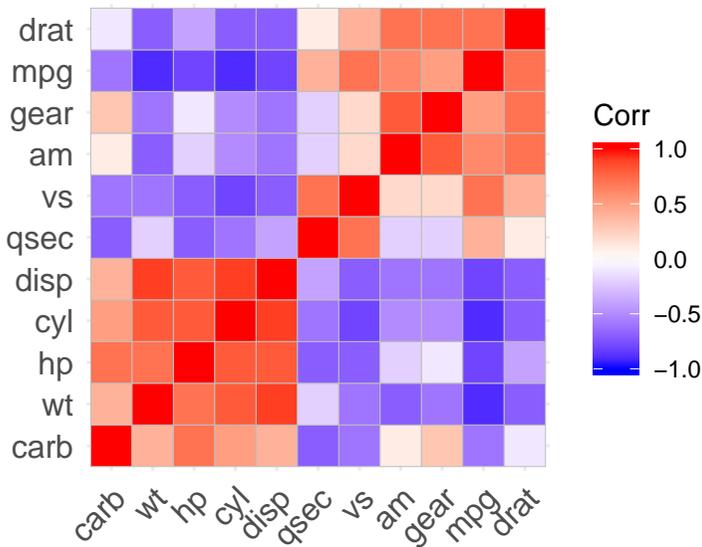
Given a set of variables $\{x_1, \dots, x_k\}$ one can form a k by k correlation matrix whose (i, j) th entry is the correlation between x_i and x_j .

A *correlogram* is a representation of the estimate of the correlation matrix from a data set. The `ggcorrplot` package creates such a correlogram for you.

```
# devtools::install_github("kassambara/ggcorrplot")
library(ggplot2)
library(ggcorrplot)

# Correlation matrix
corr <- round(cor(mtcars), 1)

# Plot
ggcorrplot(corr, hc.order = TRUE)
```



The `hc.order` parameter takes the variables, and tries to place them in groups according to which subset of variables correlates with which subgroup.

```
# install.packages("quantmod")
library(ggplot2)
library(quantmod)

# Compute % Returns
economics$returns_perc <- c(0, diff(economics$psavert)/
  economics$psavert[-length(economics$psavert)])

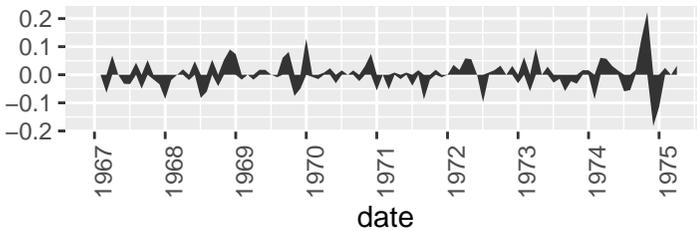
# Create break points and labels for axis ticks
brks <- economics$date[seq(1, length(economics$date), 12)]
lbls <- lubridate::year(economics$date[seq(1,
  length(economics$date), 12)])

# Plot
ggplot(economics[1:100, ], aes(date, returns_perc)) +
  geom_area() +
  scale_x_date(breaks=brks, labels=lbls) +
  theme(axis.text.x = element_text(angle=90)) +
  labs(title="Area Chart",
  subtitle = "Percentage Returns for Personal Savings",
  y="% Returns for Personal savings",
  caption="Source: economics variable")
```

% Returns for Personal savir

Area Chart

Percentage Returns for Personal Savings



Source: economics package

4.3 Diverging bars

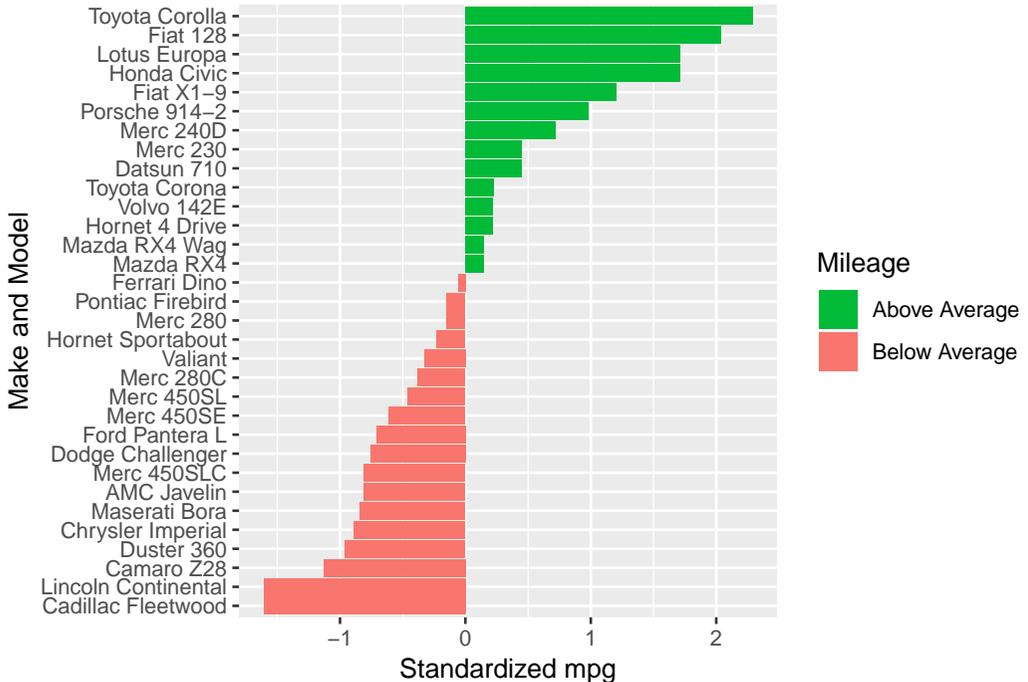
The default statistic for `geom_bar` is `count`, which means that it can only take nonnegative values. By changing the statistic with which it operates, it is possible to tweak it to take both positive and negative values.

- First, change the `stat` parameter in `geom_bar` to `'identity'`.
- Next, we are going to *standardize* our data. That means we subtract off the sample average and divided by the sample standard deviation. The resulting data is often approximately normal.
- The next step is to sort the data so that largest is first, and the smallest (most negative) is last. For this purpose, we will use the `reorder` helper function within the function `aes`.
- We use `scale_fill_manual` to color the positive and negative values differently.
- Finally, we use `coord_flip` to switch from vertical to horizontal bars.

The final code looks as follows:

```
# create rownames as factor
mtcars$car_name <- rownames(mtcars)
# standardize the values
mtcars$mpg_z <- round((mtcars$mpg - mean(mtcars$mpg)) /
  sd(mtcars$mpg), 2)
# create categorical variable
mtcars$mpg_type <- ifelse(mtcars$mpg_z < 0, "below", "above")
# Make the plot
ggplot(mtcars, aes(x = reorder(car_name, mpg_z), y = mpg_z,
  label = mpg_z)) +
  coord_flip() +
```

```
geom_bar(stat = "identity", aes(fill = mpg_type)) +
scale_fill_manual(name = "Mileage",
  labels = c("Above Average", "Below Average"),
  values = c("above" = "#00ba38", "below" = "#f8766d")) +
ylab("Standardized mpg") +
xlab("Make and Model")
```



4.4 Composition plots

Composition plots indicate what values or categorical variables occupy what percentage of the total visually through areas. The most famous of these is the pie chart.

Perhaps surprisingly, there is no built in command in the **ggplot2** package to generate a pie chart. However, this is because a pie chart is really just a bar chart that uses polar coordinates!

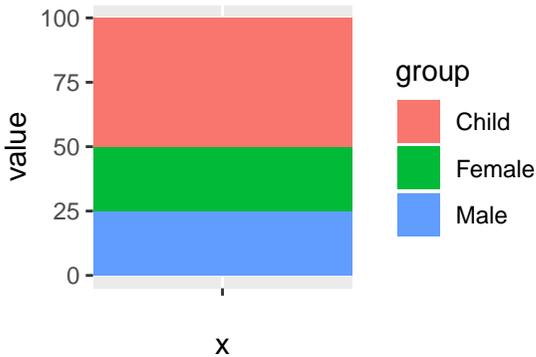
First consider this bar chart. (Example from: <http://www.sthda.com/english/wiki/ggplot2-pie-chart-quick-start-guide-r-software-and-data-visualization> retrieved 31 Jan, 2019.) The first step is to get some data.

```
df <- data.frame (
  group = c("Male", "Female", "Child"),
  value = c(25, 25, 50)
)
head(df)
```

```
##      group value
## 1   Male     25
## 2 Female     25
## 3  Child     50
```

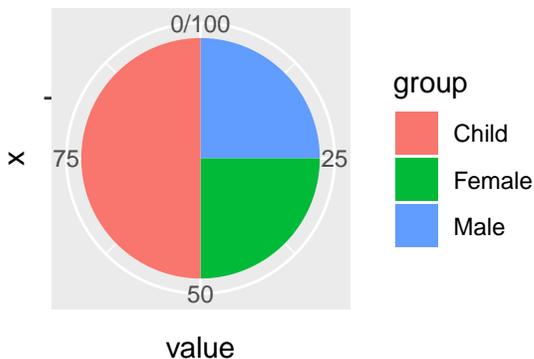
Next we look at a bar chart.

```
library(ggplot2)
# Barplot
bp<- ggplot(df, aes(x = "", y = value, fill=group))+
geom_bar(width = 1, stat = "identity")
bp
```



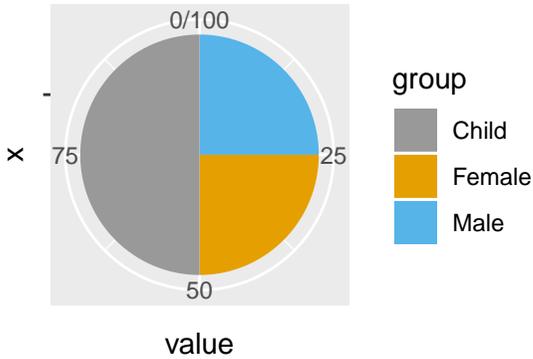
We can turn it into a pie chart simply by doing the same chart in polar coordinates.

```
pie <- bp + coord_polar("y", start = 0)
pie
```

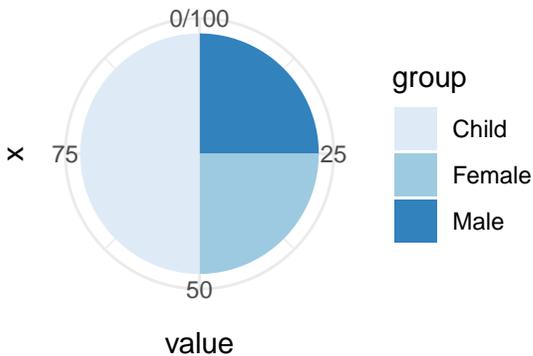


We can adjust the colors of the pie chart manually, or use built in palettes based on our taste.

```
# Use custom color palettes
pie + scale_fill_manual(values =
  c("#999999", "#E69F00", "#56B4E9"))
```



```
pie + scale_fill_brewer(palette = "Blues") +
theme_minimal()
```



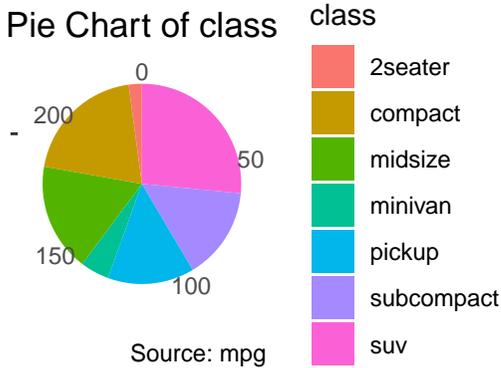
Here's a more sophisticated example.

```
library(ggplot2)
theme_set(theme_classic())

# Source: Frequency table
df <- as.data.frame(table(mpg$class))
colnames(df) <- c("class", "freq")
pie <- ggplot(df,
  aes(x = "", y=freq, fill = factor(class))) +
geom_bar(width = 1, stat = "identity") +
theme(axis.line = element_blank(),
  plot.title = element_text(hjust=0.5)) +
```

```
labs(fill="class",
      x=NULL,
      y=NULL,
      title="Pie Chart of class",
      caption="Source: mpg")
```

```
pie + coord_polar(theta = "y", start=0)
```



Here's the more sophisticated example using a categorical variable.

```
# Source: Categorical variable.
# mpg$class
pie <- ggplot(mpg, aes(x = "", fill = factor(class))) +
  geom_bar(width = 1) +
  theme(axis.line = element_blank(),
        plot.title = element_text(hjust=0.5)) +
  labs(fill="class",
        x=NULL,
        y=NULL,
        title="Pie Chart of class",
        caption="Source: mpg")

pie + coord_polar(theta = "y", start=0)
```

Pie Chart of class



Source: mpg

Transforming data

Summary A tibble is similar to a data frame in *R* but has default behavior that is slightly easier to work with. The `dplyr` package gives us several tools for transforming our tibble, including `filter` for choosing data points with properties, `arrange` for sorting rows by the data values, `select` for picking out variables of the data with certain properties, and `mutate` which allows us to create new variables as functions of existing ones.

Read Chapters 5.1-5.5 of the text.

When we learned about our visualization tools in the last few chapters, our data (such as the map data, diamonds, and mileage data) had been already nicely prepared for us.

In this chapter we will learn about the basic tools used to *transform data* so that we can extract the important pieces that we need for our analysis.

To illustrate these methods, we will use a data set that contains the On-time information for all flights from NYC in 2013.

```
library(nycflights13)
library(tidyverse)
```

loads this data set into *R* together with the tools we will use to transform it.

```
flights
```

```
# A tibble: 336,776 x 19
  year month   day dep_time sched_dep_time dep_delay
  <int> <int> <int>   <int>         <int>         <dbl>
1  2013     1     1     517           515           2
2  2013     1     1     533           529           4
3  2013     1     1     542           540           2
4  2013     1     1     544           545          -1
```

| | | | | | | |
|----|------|---|---|-----|-----|----|
| 5 | 2013 | 1 | 1 | 554 | 600 | -6 |
| 6 | 2013 | 1 | 1 | 554 | 558 | -4 |
| 7 | 2013 | 1 | 1 | 555 | 600 | -5 |
| 8 | 2013 | 1 | 1 | 557 | 600 | -3 |
| 9 | 2013 | 1 | 1 | 557 | 600 | -3 |
| 10 | 2013 | 1 | 1 | 558 | 600 | -2 |

```
# ... with 336,766 more rows, and 13 more variables:
#   arr_time <int>, sched_arr_time <int>, arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
#   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dtm>
```

A *tibble* is an extension of the basic data frame type that is found in *R*, for now the differences between a tibble and a data frame are unimportant. Notice that each row of the tibble contains a single data point, which is itself a vector whose components tell us things like the year of the flight (2013 for every data point), the departure time, the carrier, and other information.

Below the headings are abbreviations like `<int>` and `<dbl>`. These tell us the type of variable we are dealing with.

- `<int>` is an integer valued variable.
- `<dbl>` is a floating point number. It is meant to represent a real number that has been rounded so a value that in fit using 64 bits of precision in a computer. The abbreviation stands for double, since initially, floating point numbers used 32 bits and this uses double that.
- `<chr>` This stands for *character* and is used for strings of characters like "UA" or "AA".
- `<dtm>` This stands for *date and time* and records both the data and current time values for the data point.

To see more of the tibble, we can use the [View](#) function. For instance,

```
View(flights)
```

will put the entire flights variable up into the pane where the scripts and Markdown files live. We can then look at it in its entirety in a fashion similar to a spreadsheet.

5.1 The *dplyr* package

A command in *R* that allows us to apply the same operation to a bunch of different data points is, appropriately enough, called **apply**. The **plyr** function was developed as a faster means of doing certain common tasks for which **apply** was too general. Then **dplyr** was developed to specifically perform those tasks on data frames. By restricting the applicability, the package could be made as fast as possible.

Generally speaking, **dplyr** contains functions that allow us to perform the most common tasks of data management in *R* very quickly. These tasks are as follows.

- **filter** allows us to pick our data points with certain values.
- **arrange** allows us to reorder the data points by their values.
- **select** gives us the ability to pick out data points by their names.
- **mutate** allows us to add new variables as functions of existing variables in the data set.
- **summarize** allows us to summarize the values in the data.

We will cover the first four of these in this chapter: **summarize** is complex enough that we will leave that for the next chapter.

All of these commands work in roughly the same way: the first argument to the command is a data frame, and then the remaining arguments describe what the command should do to the data contained in the data frame.

5.2 The **filter** function

Let's start with **filter**, which (as the name indicates) allows us to filter out the data based on its properties.

Let's say that we want all flights on February 4th. Then we could use

```
feb4 <- filter(flights, month == 2, day == 4)
```

In making this comparison, we used the logical operator **==**, which is true if the numerical expressions on both sides of the **==** are true.

The six common comparison operators are:

| | |
|--------------------------|----|
| greater than | > |
| greater than or equal to | >= |
| less than | < |
| less than or equal to | <= |
| not equal to | != |
| equal to | == |

Recall that floating point numbers are not exact real numbers, and so an issue that comes up is when computations do not give numerical results that are identical. For instance, in a perfect world

```
sqrt(2)^2 == 2
```

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

would return TRUE, in fact it returns FALSE.

In order to deal with this floating point phenomenon, there is a command called `near` to deal with this exact situation. The command

```
near(sqrt(2)^2, 2)
```

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

returns TRUE as desired.

In the command

```
feb4 <- filter(flights, month == 2, day == 4)
```

the comma behaved the way it does in probability expressions, as a logical and.

Of course, sometimes we want to be explicit about our use of logical and. We also sometimes need logical or, and logical negation. These can be obtained as follows

| Logical Operator | Math Notation | in R |
|------------------|------------------------|------------------------|
| logical and | $p \wedge q$ | <code>p & q</code> |
| logical or | $p \vee q$ | <code>p q</code> |
| exclusive or | $p \underline{\vee} q$ | <code>xor(p, q)</code> |

So if I am interested in flights that either left in November or on Dec 25th, I would use

```
filter(flights, (month == 11) | (month == 12 & day == 25))
```

```
# A tibble: 27,987 x 19
  year month   day dep_time sched_dep_time dep_delay
  <int> <int> <int>   <int>         <int>         <dbl>
1  2013    11     1         5           2359             6
2  2013    11     1        35           2250            105
3  2013    11     1       455             500             -5
4  2013    11     1       539             545             -6
5  2013    11     1       542             545             -3
6  2013    11     1       549             600            -11
7  2013    11     1       550             600            -10
8  2013    11     1       554             600             -6
9  2013    11     1       554             600             -6
10 2013    11     1       554             600             -6
# ... with 27,977 more rows, and 13 more variables:
#   arr_time <int>, sched_arr_time <int>, arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
#   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dtm>
```

(Note that here the use of the comma for the logical and would have thrown an error.)

Missing values

One thing that often appears in data is missing values, where a data value is simply not there. For instance, if the recipient of a census survey did not fill out their age, it would appear in the data frame as NA, which stands for *Not Available*.

Such missing data values are impossible to compare with values, and so tend to result in NA when used. For instance, the commands

```
NA > 8
```

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

```
-3 == NA
```

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

```
NA + 0
```

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

```
NA / 2
```

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

```
NA == NA
```

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

all return NA. We have a special command for determine if a value is missing or not:

```
x <- c(NA, 3, NA)
is.na(x)
```

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

Now the `filter` command only returns rows where the condition is `TRUE`. If a data value is either `FALSE` or `NA`, then it is eliminated by the filter. So if you want to keep your missing values as well, you must explicitly ask for `NA` values as well. For instance,

```
df <- tibble(x = c(1, NA, 3))
filter(df, x > 1)
```

```
## # A tibble: 1 x 1
##       x
##   <dbl>
## 1     3
```

does not return the NA value in line 2. Whereas

```
filter(df, is.na(x) | x > 1)
```

```
## # A tibble: 2 x 1
##       x
##   <dbl>
## 1    NA
## 2     3
```

does return lines where either $x > 1$ or the value is **NA**.

5.3 Using *arrange* to order rows

The **arrange** command will take the rows and sort them by numerical value. For instance, the following command arranges the rows from low to high, first by column year, then month, and finally day.

```
arrange(flights, year, month, day)
```

```
# A tibble: 336,776 x 19
  year month   day dep_time sched_dep_time dep_delay
<int> <int> <int>   <int>         <int>         <dbl>
1  2013     1     1     517             515           2
2  2013     1     1     533             529           4
3  2013     1     1     542             540           2
4  2013     1     1     544             545          -1
5  2013     1     1     554             600          -6
6  2013     1     1     554             558          -4
7  2013     1     1     555             600          -5
8  2013     1     1     557             600          -3
9  2013     1     1     557             600          -3
10 2013     1     1     558             600          -2
# ... with 336,766 more rows, and 13 more variables:
#   arr_time <int>, sched_arr_time <int>, arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
#   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dtm>
```

If you want to put the rows in an order from high to low, surround that parameter with the command **desc**. So

```
arrange(flights, desc(dep_delay))
```

```
# A tibble: 336,776 x 19
  year month   day dep_time sched_dep_time dep_delay
<int> <int> <int>   <int>         <int>         <dbl>
1  2013     1     1     517             515           2
2  2013     1     1     533             529           4
3  2013     1     1     542             540           2
4  2013     1     1     544             545          -1
5  2013     1     1     554             600          -6
6  2013     1     1     554             558          -4
7  2013     1     1     555             600          -5
8  2013     1     1     557             600          -3
9  2013     1     1     557             600          -3
10 2013     1     1     558             600          -2
# ... with 336,766 more rows, and 13 more variables:
#   arr_time <int>, sched_arr_time <int>, arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
#   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dtm>
```

arranges the rows so that the largest delays are first, and then the smallest delays will be at the minimum.

5.4 Using *select* to pick out variables and string data

Many data sets have an enormous number of columns, many of which are not of interest in an analysis. The *select* command returns a tibble that only has the targeted columns/variables. For instance,

```
select(flights, year, dep_delay)
```

```
## # A tibble: 336,776 x 2
##   year dep_delay
##   <int> <dbl>
## 1  2013     2
## 2  2013     4
## 3  2013     2
## 4  2013    -1
## 5  2013    -6
## 6  2013    -4
## 7  2013    -5
## 8  2013    -3
## 9  2013    -3
## 10 2013    -2
## # ... with 336,766 more rows
```

returns a new tibble with the same number of rows as the original **flights**, but only two columns: **year** and **dep_delay**.

We can treat variable names as a range with **select** to grab the variables and everything in between.

```
select(flights, year:day)
```

```
## # A tibble: 336,776 x 3
##   year month   day
##   <int> <int> <int>
## 1  2013     1     1
## 2  2013     1     1
## 3  2013     1     1
## 4  2013     1     1
## 5  2013     1     1
## 6  2013     1     1
## 7  2013     1     1
## 8  2013     1     1
## 9  2013     1     1
## 10 2013     1     1
## # ... with 336,766 more rows
```

grabs the variables **year**, **day**, and the **month** variable that is in between them.

We can use **helper functions** within **select** in order to pick out the variable names that match certain criteria. Fortunately, most of these helper functions are self explanatory. For instance,

1. **starts_with("start")** matches all names that begin with “start”.
2. **ends_with("end")** matches all names that end with “end”.
3. **contains("middle")** matches all names that have the string “middle” somewhere inside them.
4. **num_range("a", 1:4)** would match either a1, a2, a3, or a4.

For more general string matching, there is the **matches** command, which uses what are called **regular expressions**. We’ll go into regular expressions in more detail later when we discuss the variable type strings in more detail.

5.5 Using **mutate** to create new variables

One of the great strengths of spreadsheets is their ability to create new columns based on data from the old ones. For instance, if I wish to create a new variable that is the difference of two other ones in my spreadsheet that is very easy to do with a small sheet. When the spreadsheet has 10^5 rows, that becomes much more difficult to do.

For a tibble, that same functionality resides in the `mutate` command. This command adds new variables to the tibble that are created as a function of previous variables. For instance, suppose we start with a smaller tibble that picks out a few variables including those that end with the string “delay”.

```
flights_sml <- select(flights,
  year:day,
  ends_with("delay"),
  distance,
  air_time
)
```

Now we can calculate things like how much time the pilots made up in the air, and what the average speed of the aircraft was.

```
mutate(flights_sml,
  gain = dep_delay - arr_delay,
  speed = distance / air_time * 60
)
```

The gain and speed variables then have values such as:

```
# A tibble: 336,776 x 4
  dep_delay arr_delay gain speed
  <dbl>      <dbl> <dbl> <dbl>
1         2         11    -9  370.
2         4         20   -16  374.
3         2         33   -31  408.
4        -1        -18    17  517.
5        -6        -25    19  394.
6        -4         12   -16  288.
7        -5         19   -24  404.
8        -3        -14    11  259.
9        -3         -8     5  405.
10       -2         8    -10  319.
# ... with 336,766 more rows
```

5.6 Logical operators in R

Note that the logical operators `==`, `&` and `|` are *vector* operators. For instance, consider the command

```
c(2, 1, -6) == c(2, 7, -6)
```

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

Because it looks at each component of the vector and sees if it is a match, the result is a vector of three boolean values. Similarly consider

```
c(TRUE, FALSE, FALSE) & c(TRUE, TRUE, FALSE)
```

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

This also has a vector of three boolean values.

Typically, this is exactly the behavior we want when using **filter** to find data with certain properties.

There are other logical operators, however, **&&** and **||**. These are not vector operators, but only work on the first component.

```
c(TRUE, FALSE, FALSE) && c(TRUE, TRUE, FALSE)
```

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

The output was only a single boolean, based on the first component

These double symbol operators are better for program control using `if` and `while`, and we will discuss these in length later on.

5.7 A note about SQL

The Structured Query Language (SQL) is designed to perform tasks similar to the ones that we looked at in this chapter. Later on, we will see how to build queries from a relational database with SQL that accomplishes the types of tasks that we did here with **dpyrl**.

Chapter 6

Creating summaries of tibbles

Summary The `group_by` function takes the data of a tibble and partitions it into groups. Then the `summarize` command can be used to return summaries that operate on each group.

Read Chapters 5.6-5.7 of the text.

Our last commands for transforming tibbles is `summarize` and `group_by`. The `group_by` command allows us to partition the data into groups. At first, when using this command it appears like our data is unchanged. However, once the data has been partitioned, you can use `summarize` together with functions such as `mean`, `median`, or `arrange` in order to apply them not to the entirety of the data, but instead to within each group in the partition.

R fact 9

You can either use the American English spelling `summarize` or the British English spelling `summarise` for this command.

Basically this collapses a tibble down based on how we perform the summary. As with the previous chapter, we are working with the `flights` tibble and `tidyverse` commands. First we load the libraries:

```
library(nycflights13)
library(tidyverse)
```

Start with the `summarize` command. First we apply `summarize` to `flights` without doing a partition first.

```
summarize(flights)
```

```
## data frame with 0 columns and 0 rows
```

The result is a data frame with 0 rows and 0 columns! That is because we did not tell the command *what* to include in the summary. Let's add a bit more detail.

```
summarize(flights, delay = mean(dep_delay, na.rm = TRUE))
```

```
## # A tibble: 1 x 1
##   delay
##   <dbl>
## 1  12.6
```

The `na.rm` parameter to `mean` is a logical parameter that when `TRUE`, strips out all of the `NA` values in calculating the mean. If we forget to strip out the `NA` values, then we might end up with a `NA` for our final result.

This creates a new tibble from `flights` with a single variable `delay` whose value is the mean of all the `dep_delay` values in the original flights (excluding the `NA` values.)

6.1 Using `group_by`

We can use the `group_by` command to take a tibble and break it down into groups. For instance, consider

```
by_day <- group_by(flights, day)
```

The variable `by_day` is now the same tibble as `flights`, but with 31 extra groups, one for each day. The original flights variable looks like:

```
# A tibble: 336,776 x 19
  year month   day dep_time sched_dep_time dep_delay
<int> <int> <int>   <int>         <int>         <dbl>
1  2013     1     1     517           515           2
2  2013     1     1     533           529           4
3  2013     1     1     542           540           2
4  2013     1     1     544           545          -1
5  2013     1     1     554           600          -6
6  2013     1     1     554           558          -4
7  2013     1     1     555           600          -5
8  2013     1     1     557           600          -3
9  2013     1     1     557           600          -3
10 2013     1     1     558           600          -2
# ... with 336,766 more rows, and 13 more variables:
#   arr_time <int>, sched_arr_time <int>, arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
#   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dtm>
```

Now the grouped variable `by_day`:

```
# A tibble: 336,776 x 19
# Groups:   day [31]
  year month   day dep_time sched_dep_time dep_delay
<int> <int> <int>   <int>         <int>         <dbl>
1  2013     1     1     517             515           2
2  2013     1     1     533             529           4
3  2013     1     1     542             540           2
4  2013     1     1     544             545          -1
5  2013     1     1     554             600          -6
6  2013     1     1     554             558          -4
7  2013     1     1     555             600          -5
8  2013     1     1     557             600          -3
9  2013     1     1     557             600          -3
10 2013     1     1     558             600          -2
# ... with 336,766 more rows, and 13 more variables:
#   arr_time <int>, sched_arr_time <int>, arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
#   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dtm>
```

It looks the same. However, now when we run `summarize` command as before on the grouped variable `by_day`, the result is different:

```
summarize(by_day, delay = mean(dep_delay, na.rm = TRUE))
```

```
## # A tibble: 31 x 2
##   day delay
##   <int> <dbl>
## 1     1  14.2
## 2     2  14.1
## 3     3  10.8
## 4     4   5.79
## 5     5   7.82
## 6     6   6.99
## 7     7  14.3
## 8     8  21.8
## 9     9  14.6
## 10    10  18.3
## # ... with 21 more rows
```

So now the mean of the `dep_delay` variable has been calculated for each *group*. Since there were 31 groups (one for each day), we have 31 means.

6.2 Using pipes to avoid intermediate variables

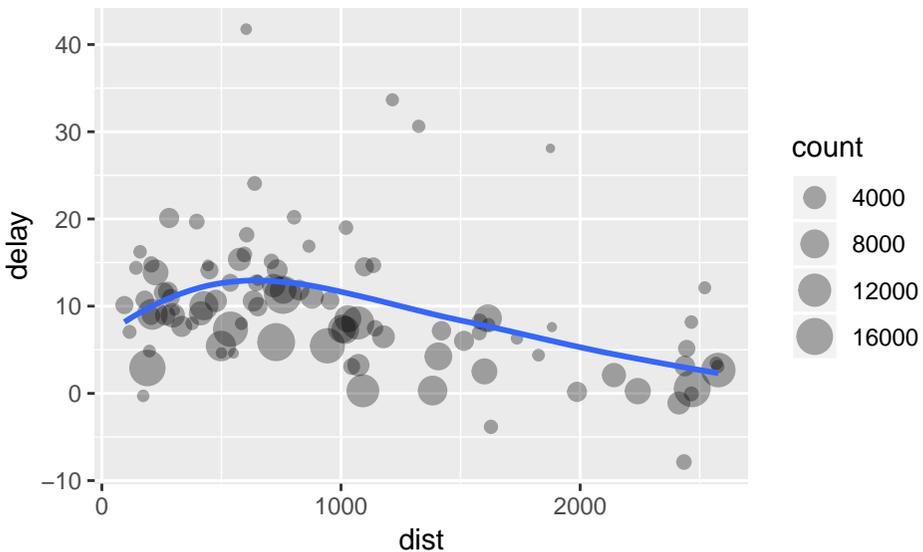
The `by_day` variable is an example of an intermediate variable. We did not really want to create it, but we needed to in order to complete our calculation. If we need to use multiple functions, one after another, we can end up creating a lot of unnecessary intermediate variables. For instance, the following code

```

by_dest <- group_by(flights, dest)
delay <- summarize(by_dest,
  count = n(),
  dist = mean(distance, na.rm = TRUE),
  delay = mean(arr_delay, na.rm = TRUE)
)
delay <- filter(delay, count > 20, dest != "HNL")

# It looks like delays increase with distance up to ~750 miles
# and then decrease. Maybe as flights get longer there's more
# ability to make up delays in the air?
ggplot(data = delay, mapping = aes(x = dist, y = delay)) +
  geom_point(aes(size = count), alpha = 1/3) +
  geom_smooth(se = FALSE)

```



```
#> 'geom_smooth()' using method = 'loess' and formula 'y ~ x'
```

One way to avoid having to create these intermediate variables is with pipes. A pipe takes an output or variable and feeds it into another function. In R, pipes are created by `%>%`. So we can write the same code to generate delay with pipes as

```

delays <- flights %>%
  group_by(dest) %>%
  summarize(
    count = n(),

```

```

  dist = mean(distance, na.rm = TRUE),
  delay = mean(arr_delay, na.rm = TRUE)
) %>%
filter(count > 20, dest != "HNL")

```

Here `flights` is being fed into the `group_by` function and its output is fed directly into the `summarize` function. Then its output is fed directly into the `filter` command.

In general, using pipes makes code easier to read and so should be used in these types of situations when possible.

6.3 The effect of NA values

Suppose that we forgot to take out our NA values from our mean. What would happen? Since the values are unknown, the overall sample average is unknown. Consider the following command.

```

flights %>%
  group_by(year, month, day) %>%
  summarize(mean = mean(dep_delay))

```

```

## # A tibble: 365 x 4
## # Groups:   year, month [?]
##   year month   day mean
##   <int> <int> <int> <dbl>
## 1  2013     1     1    NA
## 2  2013     1     2    NA
## 3  2013     1     3    NA
## 4  2013     1     4    NA
## 5  2013     1     5    NA
## 6  2013     1     6    NA
## 7  2013     1     7    NA
## 8  2013     1     8    NA
## 9  2013     1     9    NA
## 10 2013     1    10    NA
## # ... with 355 more rows

```

Since so many of the variables are unknown, so are the means. Of course, we could have also removed any rows with a NA value for `dep_delay` first, and then done the experiment.

First, we remove the NA values using `filter`.

```

not_cancelled <- flights %>%
  filter(!is.na(dep_delay), !is.na(arr_delay))

```

Then summarize as before.

```
not_cancelled %>%
  group_by(year, month, day) %>%
  summarize(mean = mean(dep_delay))
```

```
## # A tibble: 365 x 4
## # Groups:   year, month [?]
##   year month   day mean
##   <int> <int> <int> <dbl>
## 1  2013     1     1  11.4
## 2  2013     1     2  13.7
## 3  2013     1     3  10.9
## 4  2013     1     4   8.97
## 5  2013     1     5   5.73
## 6  2013     1     6   7.15
## 7  2013     1     7   5.42
## 8  2013     1     8   2.56
## 9  2013     1     9   2.30
## 10 2013     1    10   2.84
## # ... with 355 more rows
```

The variable then looks like

```
not_cancelled
```

```
## # A tibble: 327,346 x 19
##   year month   day dep_time sched_dep_time dep_delay
##   <int> <int> <int>   <int>         <int>         <dbl>
## 1  2013     1     1     517           515             2
## 2  2013     1     1     533           529             4
## 3  2013     1     1     542           540             2
## 4  2013     1     1     544           545            -1
## 5  2013     1     1     554           600            -6
## 6  2013     1     1     554           558            -4
## 7  2013     1     1     555           600            -5
## 8  2013     1     1     557           600            -3
## 9  2013     1     1     557           600            -3
## 10 2013     1     1     558           600            -2
## # ...
```

6.4 Combining groups with *filter*, *select*, and *mutate*

Groups can also be used with the `filter` function. For instance,

```
flights %>%
  group_by(year, month, day) %>%
  filter(rank(desc(arr_delay)) < 5)
```

```
## # A tibble: 1,464 x 19
## # Groups:   year, month, day [365]
##   year month   day dep_time sched_dep_time dep_delay
##   <int> <int> <int>   <int>         <int>         <dbl>
## 1  2013     1     1     848           1835           853
## 2  2013     1     1    1815           1325           290
## 3  2013     1     1    1842           1422           260
## 4  2013     1     1    2343           1724           379
## 5  2013     1     2    1332             904           268
## 6  2013     1     2    1412             838           334
## 7  2013     1     2    1607           1030           337
## 8  2013     1     2    2131           1512           379
## 9  2013     1     3    1834           1540           174
## 10 2013     1     3    2008           1540           268
## # ... with 1,454 more rows ...
```

This has found the four worst arrival delays for each particular day. Note that these worst arrivals are not sorted by `arr_delay`, they appear in the same order as in the original tibble.

Sometimes when we group our tibble, some groups may be too small to be useful. The `n` function helps in these situation. For instance, there are 105 different destinations for the flights:

```
flights %>% group_by(dest)
```

```
## # A tibble: 336,776 x 19
## # Groups:   dest [105]
##   year month   day dep_time sched_dep_time dep_delay
##   <int> <int> <int>   <int>         <int>         <dbl>
## 1  2013     1     1     517           515            2
## 2  2013     1     1     533           529            4
## 3  2013     1     1     542           540            2
## 4  2013     1     1     544           545           -1
## 5  2013     1     1     554           600           -6
## 6  2013     1     1     554           558           -4
## 7  2013     1     1     555           600           -5
## 8  2013     1     1     557           600           -3
## 9  2013     1     1     557           600           -3
## 10 2013     1     1     558           600           -2
## # ... with 336,766 more rows,...
```

By using `n`, we can keep only those destinations with at least 1000 members.

```
flights %>% group_by(dest) %>% filter(n() >= 1000)
```

```
## # A tibble: 320,366 x 19
## # Groups:   dest [58]
```

```
##      year month   day dep_time sched_dep_time dep_delay
##      <int> <int> <int>   <int>         <int>         <dbl>
##  1  2013     1     1     517           515             2
##  2  2013     1     1     533           529             4
##  3  2013     1     1     542           540             2
##  4  2013     1     1     554           600            -6
##  5  2013     1     1     554           558            -4
##  6  2013     1     1     555           600            -5
##  7  2013     1     1     557           600            -3
##  8  2013     1     1     557           600            -3
##  9  2013     1     1     558           600            -2
## 10  2013     1     1     558           600            -2
## # ... with 320,356 more rows,...
```

There were only 58 such destinations with at least 1000 flights to them. We can then take these flights and use `mutate` and `select` on them as well. The following returns the proportion of delay for each group.

```
flights %>%
  group_by(dest) %>%
  filter(n() >= 1000) %>%
  filter(arr_delay > 0) %>%
  mutate(prop_delay = arr_delay / sum(arr_delay)) %>%
  select(year:day, dest, arr_delay, prop_delay)
```

```
## # A tibble: 125,929 x 6
## # Groups:   dest [58]
##      year month   day dest  arr_delay prop_delay
##      <int> <int> <int> <chr>    <dbl>     <dbl>
##  1  2013     1     1 IAH      11  0.000111
##  2  2013     1     1 IAH      20  0.000201
##  3  2013     1     1 MIA      33  0.000235
##  4  2013     1     1 ORD      12  0.0000424
##  5  2013     1     1 FLL      19  0.0000938
##  6  2013     1     1 ORD       8  0.0000283
##  7  2013     1     1 LAX       7  0.0000344
##  8  2013     1     1 DFW      31  0.000282
##  9  2013     1     1 ATL      12  0.0000400
## 10  2013     1     1 DTW      16  0.000116
## # ... with 125,919 more rows
```

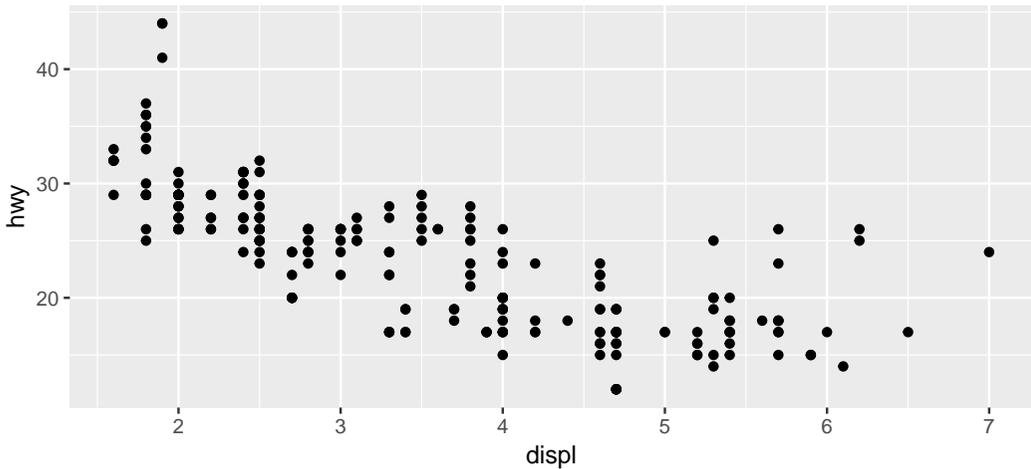
So apparently this third flight to Miami (MIA) was .0235% of all flight delays to that destination.

6.5 Example: average mileage and displacement by car class

We noted in our framework that often the process of visualization will lead us to a transformation. For instance, consider once again our plot of highway mileage versus engine

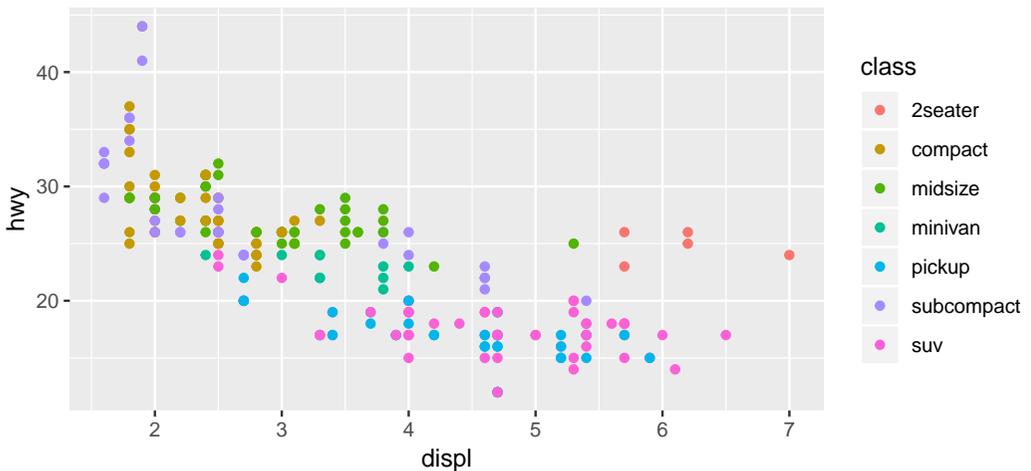
displacement for the mpg data set.

```
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point()
```



We found that breaking the data down by the class of the vehicle made for a much clearer view of what is going on.

```
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point(aes(col = class))
```



That means in doing our analysis, we should work with the data broken down by class. That is exactly what `group_by` does.

```
mpg %>%
  group_by(class) %>%
  summarize(mean(hwy, na.rm = TRUE), mean(displ, na.rm = TRUE))

## # A tibble: 7 x 3
##   class      'mean(hwy) ' 'mean(displ) '
##   <chr>      <dbl>          <dbl>
## 1 2seater      24.8            6.16
## 2 compact     28.3            2.33
## 3 midsize     27.3            2.92
## 4 minivan     22.4            3.39
## 5 pickup      16.9            4.42
## 6 subcompact  28.1            2.66
## 7 suv         18.1            4.46
```

Exploratory Data Analysis: Variation

Summary In Exploratory Data Analysis (EDA), we try to figure out where the data lies and what types of patterns it has. Here we concentrate on **variation**, how to understand the different types of data. The **count** function in **dplyr** is useful here, both for **categorical** and for **numerical** data.

Read Chapter 7.1–7.4 of the text.

When faced with a new data set, the first step is usually what statisticians call Exploratory Data Analysis (EDA). This is when we first try to look at what the data is telling us.

There is no one way to approach EDA, partially because at the beginning, there is no way to know what is going on with your data set. That being said, there are three general areas that we usually start with.

- Center: where is the data located?
- Variation: how does the data vary from its center.
- Covariation: how do two or more variables interact.

First we set up some terminology.

- *Variables* are things that we can measure. They can either be quantitative (numerical) or qualitative (for instance gender).
- *Value* is the state of a variable when measured.
- *Observations* are a set of measurements of a particular variable. These are also referred to as a data point.
- *Tabular data* is a way of organizing the data into a table. We use the term **tidy** to indicate that the variables are set up in the columns, and the rows contain observations.

7.1 Variation

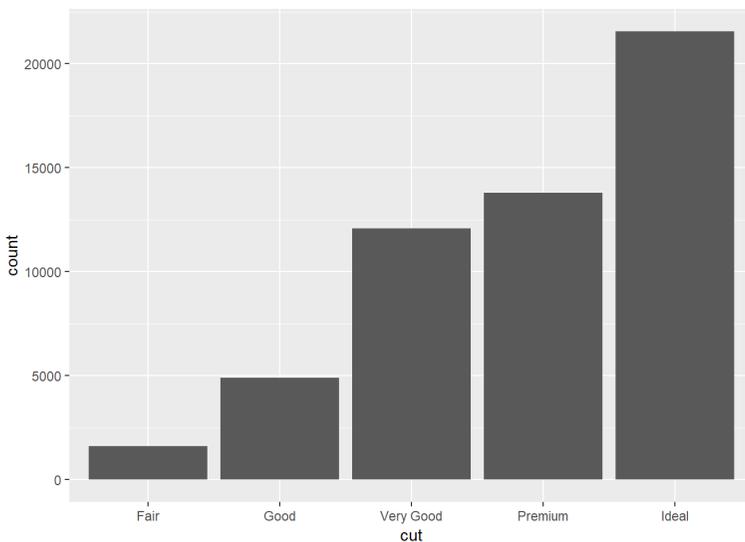
For most experiments, when you measure a variable more than once, you do not obtain the same result. *Variation* represents the fact that you obtain different values when measuring more than once. There are various ways of measuring this variation depending on whether we are dealing with a numerical variable, or a categorical variable.

Definition 19

If a variable takes on only a finite set of values, we call it **categorical**.

For example, the cut of a diamond is either fair, good, very good, premium, or ideal. Using the built in `diamonds` data set, we can illustrate this as follows.

```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut))
```



In this plot, the height of the bar is the number of data points that have this value. The `count` command in the `dplyr` package can be used to extract this data manually.

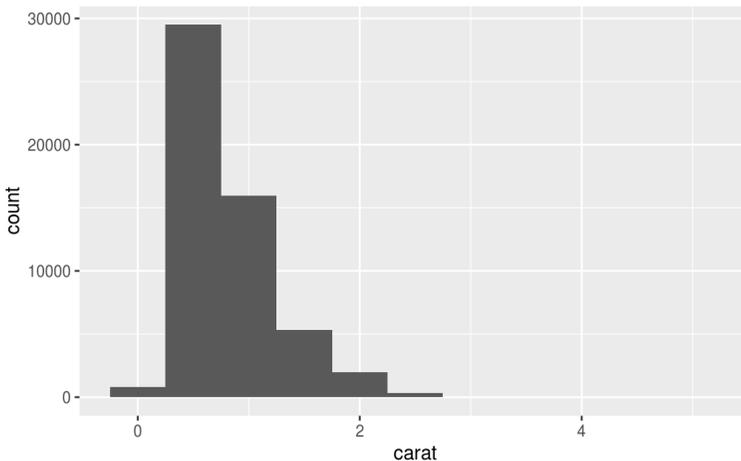
```
> diamonds %>% count(cut)
# A tibble: 5 x 2
  cut          n
<ord>      <int>
1 Fair         1610
2 Good         4906
3 Very Good   12082
4 Premium     13791
5 Ideal       21551
```

Definition 20

If observations are real numbers, call the data **numerical**.

For numerical or *continuous* data, we form our histogram by *binning* the values. We select values $a_1 < a_2 < \dots < a_k$, and add to the count of bar i if the numerical value falls into the interval $(a_i, a_{i+1}]$. If all the intervals have the same width, that is $a_{i+1} - a_i$ is the same for all i , then we call that the *bin width*. If the `geom_histogram` command, we can specify a common bin width for all the bins.

```
ggplot(data = diamonds) +
  geom_histogram(mapping = aes(x = carat), binwidth = 0.5)
```



We can manually compute the counts for these bins with `count` as well.

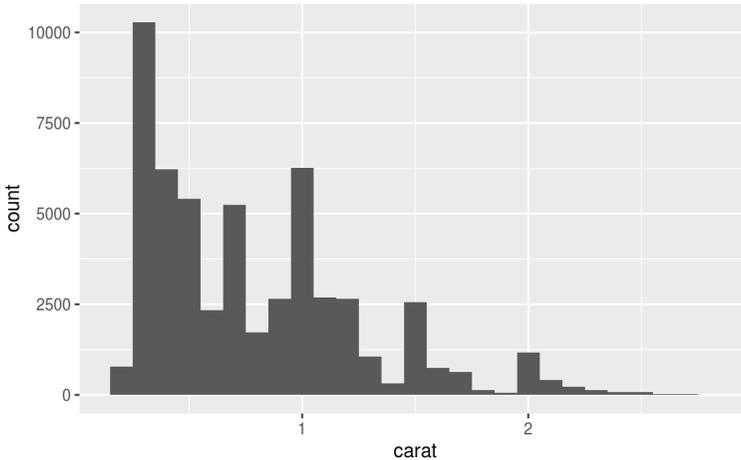
```
> diamonds %>%
+   count(cut_width(carat, 0.5))
# A tibble: 11 x 2
  `cut_width(carat, 0.5)`     n
  <fct>                       <int>
1 [-0.25, 0.25]              785
2 (0.25, 0.75]             29498
3 (0.75, 1.25]             15977
4 (1.25, 1.75]              5313
5 (1.75, 2.25]              2002
6 (2.25, 2.75]               322
7 (2.75, 3.25]               32
8 (3.25, 3.75]                5
9 (3.75, 4.25]                4
10 (4.25, 4.75]                1
11 (4.75, 5.25]                1
```

The *exploratory* part of EDA means that it is important to try different bin widths on different parts of the data in order to try and learn about how it behaves. For instance, suppose we restrict ourselves to the smaller carat results:

```
smaller <- diamonds %>%
  filter(carat < 3)
```

and then play with the bin width.

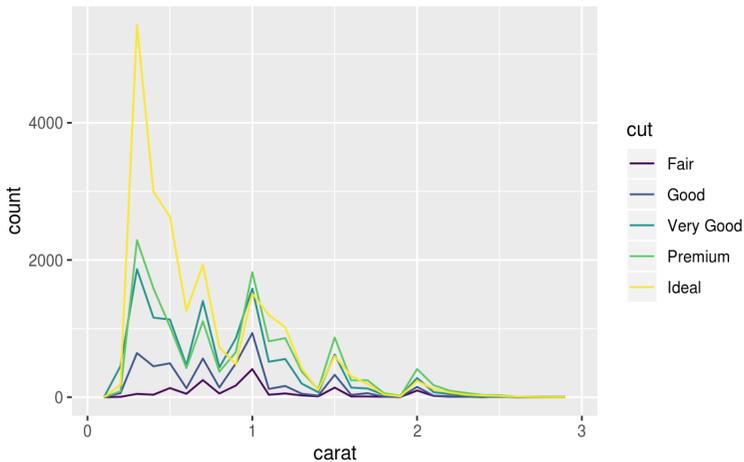
```
ggplot(data = smaller, mapping = aes(x = carat)) +
  geom_histogram(binwidth = 0.1)
```



From this perspective, we can see there are peaks near very low carats, 1 carat, 1.5 carat, and 2 carat diamonds.

We can place several histograms in the same plot, but in this case it can be helpful to use `geom_freqpoly()` rather than `geom_histogram`. This plots the counts using lines rather than bars, which allows us to consider all the different cuts of diamonds simultaneously.

```
ggplot(data = smaller, mapping = aes(x = carat, colour = cut)) +
  geom_freqpoly(binwidth = 0.1)
```



Once we have our plots, what sort of things should we be looking for?

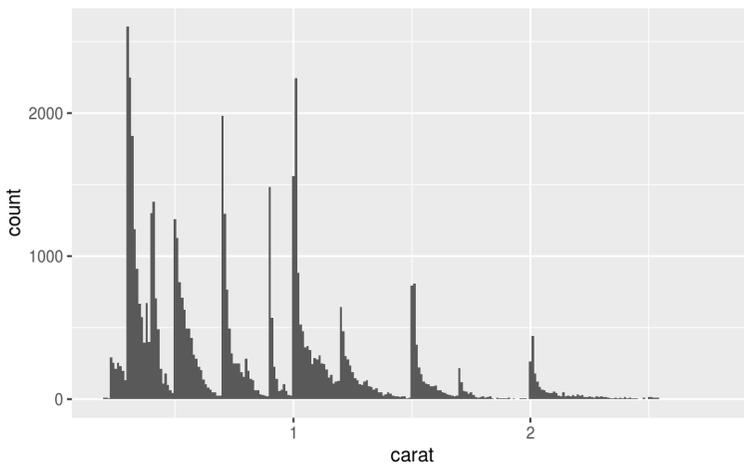
- What are the most common values of the data?
- What values do we not see? Is that reasonable?
- Are there any patterns appearing in the data? What aspects of the data could explain the pattern you see?

From our data, we notice several interesting things

- The carats appear to peak at whole numbers or low denominator fractions.
- The diamonds appear to trail off to the right rather than the left.

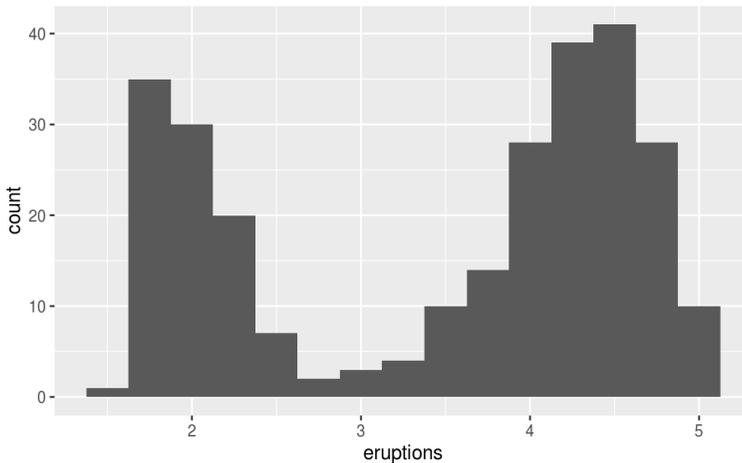
We can see if this pattern holds with a smaller bin width.

```
ggplot(data = smaller, mapping = aes(x = carat, colour = cut)) +
  geom_freqpoly(binwidth = 0.01)
```



Let's look at the histogram for the length of eruptions in Yellowstone.

```
ggplot(data = faithful, mapping = aes(x = eruptions)) +
  geom_histogram(binwidth = 0.25)
```

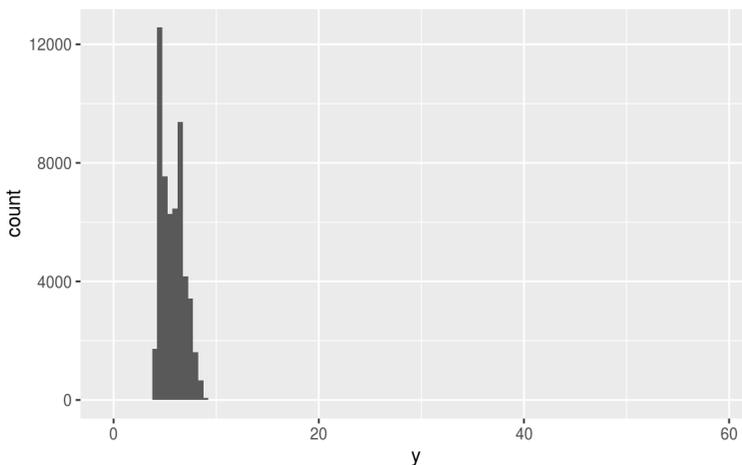


What sort of patterns do you see in this histogram?

7.2 Rare values

Note that we were able to see the patterns in the diamonds only by zooming in on pieces of the data. As another example, let's consider the y variable, which holds the width of the diamond. The view from above of all the widths of the diamonds does not tell us much.

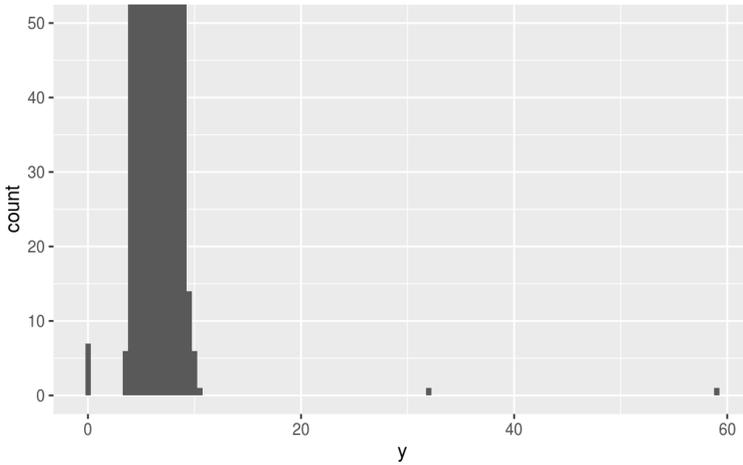
```
ggplot(diamonds) +
  geom_histogram(mapping = aes(x = y), binwidth = 0.5)
```



Unfortunately, the 12000+ count bar completely wipes out the smaller bars that we would otherwise see.

In order to see these rarer values, we can clip the y coordinate to only run from 0 to 50.

```
ggplot(diamonds) +
  geom_histogram(mapping = aes(x = y), binwidth = 0.5) +
  coord_cartesian(ylim = c(0, 50))
```



Aha! We have a bar at 0 one just right of 30, and one just shy of 60. Let's use `filter` to pick out the data corresponding to these values.

```
> diamonds %>%
+   filter(y < 3 | y > 20) %>%
+   select(price, x, y, z) %>%
+   arrange(y)
# A tibble: 9 x 4
  price     x     y     z
  <int> <dbl> <dbl> <dbl>
1  5139     0     0     0
2  6381     0     0     0
3 12800     0     0     0
4 15686     0     0     0
5 18034     0     0     0
6  2130     0     0     0
7  2130     0     0     0
8  2075  5.15 31.8  5.12
9 12210  8.09 58.9  8.06
```

A look at the help for diamonds tells us about the variables. The x , y , and z variables measure the length, width, and depth of the diamonds. So how can these all be 0? That must be a mistake in how the data was recorded.

Line 8 isn't a whole lot better. The variables are measured in mm, and 31.8 mm is more than an inch wide! Would such a diamond only cost \$2075? Not very likely, so again these values are probably errors in the data set.

So what should we do with these types of values that we believe are wrong? We have a choice, we can cut them out entirely, or we can switch them over to NA. This can be accomplished with `mutate`.

```
diamonds2 <- diamonds %>%  
  mutate(y = ifelse(y < 3 | y > 20, NA, y))
```

We used the `ifelse` operator here. The way `ifelse` works is that if the first argument is TRUE, then the value is the second argument. If the first argument is FALSE, then the value is the third argument.

Now when `ggplot2` is used on `diamonds`, it will warn about the missing values.

```
ggplot(data = diamonds2, mapping = aes(x = x, y = y)) +  
  geom_point()  
#> Warning: Removed 9 rows containing missing values (geom_point  
  ).
```

If we modify this last command to `geom_point(na.rm())`, then the warning disappears.

Chapter 8

Exploratory Data analysis: Covariation

Read Sections 7.5–7.8 of the text.

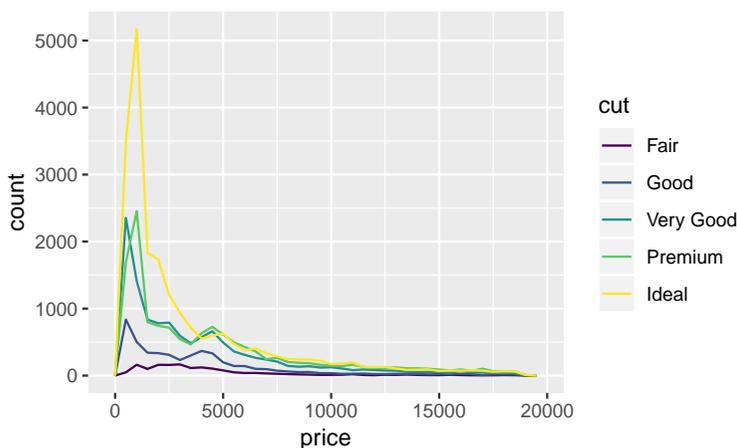
In the last chapter we saw how to calculate variation, which is how far away a variable tends to be from its center. Now we discuss **covariation** which describes how more than one variable interact.

8.1 Categorical and continuous variables

The first case we consider is when dealing with trying to understand the covariation between a categorical (discrete) and a continuous (numerical) random variable.

One method is simply to plot the continuous variable for the different values that the random variable can take on. For example, consider the variable diamonds from package `ggplot2`. We wish to understand the `price` factor versus the `cut`. For each price, we can count how many of each cut fall into that price point. The geom `geom_freqpoly` can accomplish this task.

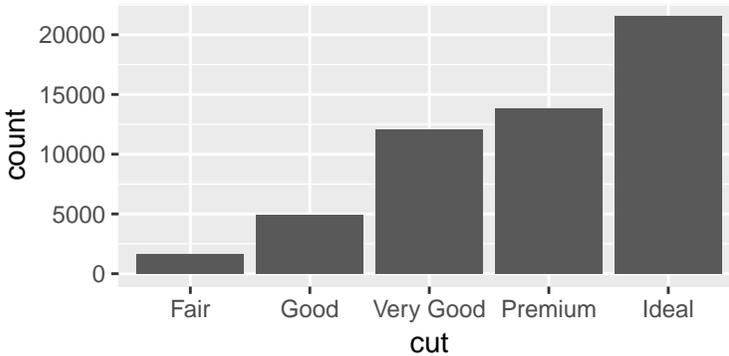
```
library(ggplot2)
ggplot(data = diamonds, mapping = aes(x = price)) +
  geom_freqpoly(mapping = aes(color = cut), binwidth = 500)
```



From the picture, it appears that Ideal cuts have many more diamonds at low counts. But this could simply be because there are in fact many more diamonds altogether in the data set that have ideal cuts.

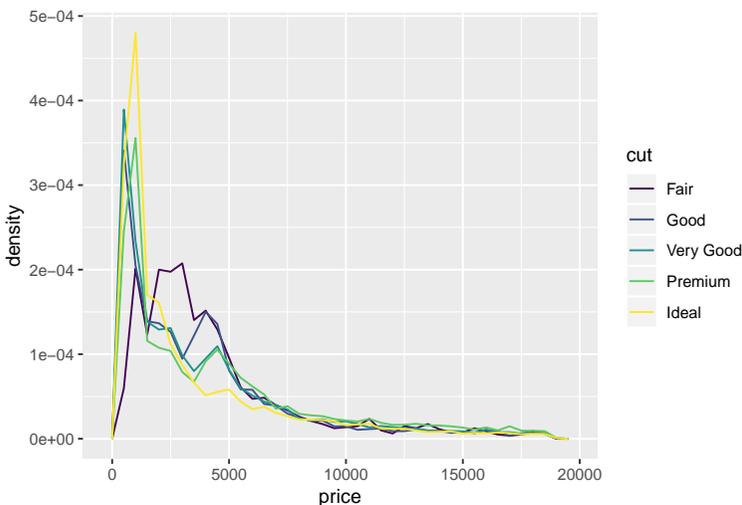
A quick check shows that to be true.

```
ggplot(data = diamonds, mapping = aes(x = cut)) +
  geom_bar()
```



In order to deal with this we need to normalize the data. In other words, we readjust the data so that the area under the frequency curve is one. We accomplish this by giving the aesthetic for variable y the special parameter `..density..`.

```
ggplot(data = diamonds, mapping = aes(x = price, y = ..density..)) +
  geom_freqpoly(mapping = aes(color = cut), binwidth = 500)
```



So there we have it: higher quality cuts of diamonds tend to be cheaper than lower quality cuts.

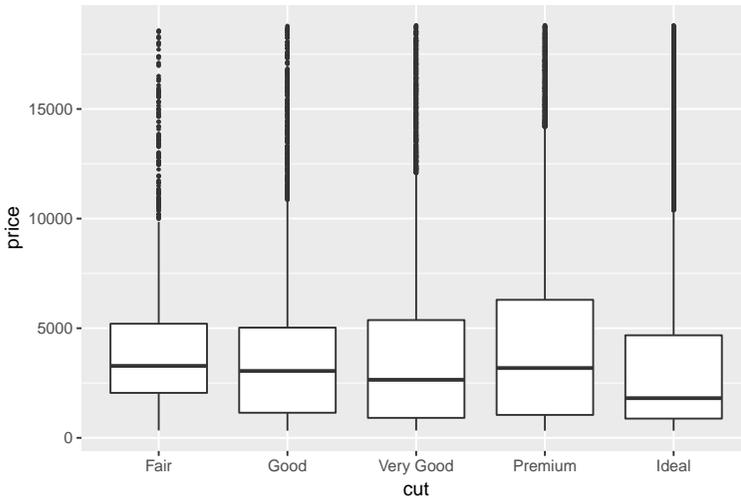
This is frustrating when you see this type of headline in the newspaper, because it is so obviously wrong. So what's going on with the diamonds? Well, one thing is that these are not the only variables involved. Another factor strongly correlated with price is size. If Ideal cut diamonds tend to be smaller, then the overall price on average might be smaller despite the fact that each individual diamond would cost more than the otherwise equivalent diamond with a lesser cut.

These other factors that mess up our attempts to study covariation are called *confounding variables*, and it important to try to keep their effects to a minimum.

8.2 Boxplots

Another way to study the distribution of variables is through the use of **boxplots**. For instance, the boxplots of the price versus cut can be found with `geom_boxplot`.

```
ggplot(data = diamonds, mapping = aes(x = cut, y = price)) +
  geom_boxplot(outlier.size=0.5)
```



The boxplot consists of three parts.

1. In the middle is a box with a horizontal line somewhere in the middle. This line is the *sample median*, a place where about 50% of the data values are above the line and 50% below. Similarly, the top of the boxplot is 75% quantile, where about 75% of the data is below. The bottom of the box is the 25% quantile, so about 25% of the data is below. The distance from the top of the box until the bottom of the box is called the *inquartile range* or *IQR* for short.
2. The points that are farther than $1.5 \cdot \text{IQR}$ from the top or bottom of the box are called *outliers*. Each outlier gets a single dot in the boxplot.

- A *whisker* is a line drawn from the top of the box out to the first outlier, or $1.5 \cdot \text{IQR}$ distance, whichever is longer. A similar whisker is drawn from the bottom of the box.

The 25% quantile is also called the *first quartile*, the median is the *second quartile* and the 75% quantile is the *third quartile*.

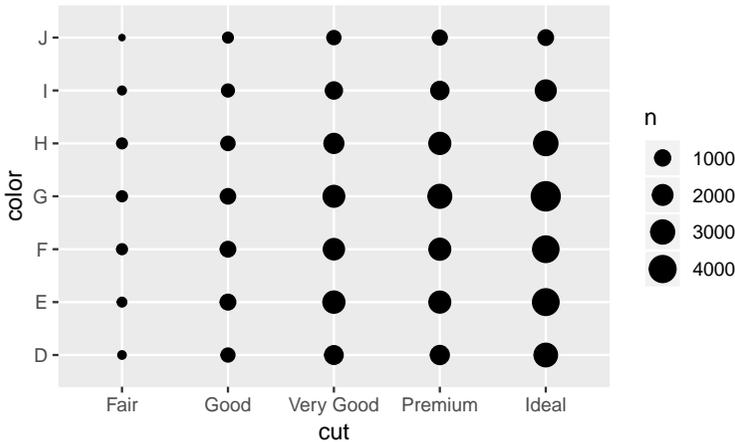
Since the data is always positive here, the bottom whisker stays above 0.

Between the medians, 75% quantile, 25% quantile, and the outliers, it is clear that the pattern seen in the frequency plots is not a fluke: higher quality diamonds really do have lower prices.

8.3 Two categorical variables.

One way to study covariation between categorical variables is to look at the counts for the different pair combinations. The `geom_count` function does the job.

```
ggplot(diamonds) +
  geom_count(aes(cut, color))
```



Again we see the counts of all colors increasing as the quality of the cut increases, but now we detect another pattern: Some colors levels are more common than others, and it seems to be roughly the same pattern across cut.

While `geom_count` is easy, it is also not that pretty. Using the `count` function from `dplyr` directly gives the following.

```
library(tidyverse)
diamonds %>%
  count(color, cut)
```

```
# A tibble: 35 x 3
  color cut          n
```

```

  <ord> <ord>      <int>
1 D      Fair       163
2 D      Good       662
3 D      Very Good 1513
4 D      Premium   1603
5 D      Ideal     2834
6 E      Fair       224
7 E      Good       933
8 E      Very Good 2400
9 E      Premium   2337
10 E     Ideal     3903
# ... with 25 more rows

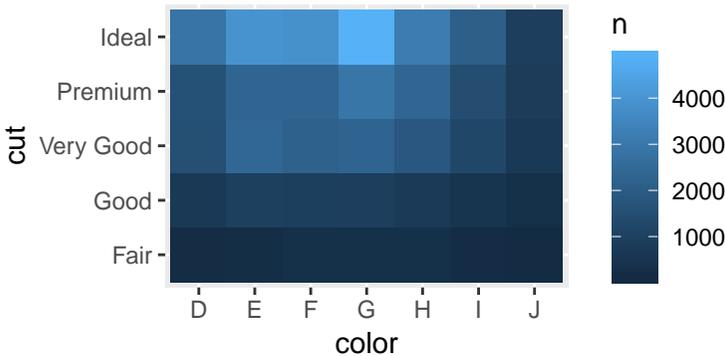
```

We can visualize this with the `geom_tile` function.

```

diamonds %>%
  count(color, cut) %>%
  ggplot(mapping = aes(x = color, y = cut)) +
  geom_tile(mapping = aes(fill = n))

```



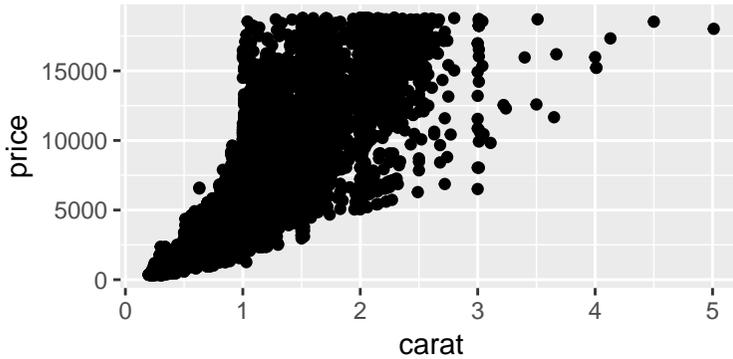
Two continuous variables

Scatterplots are the go-to method for seeing how one variable changes as another does.

```

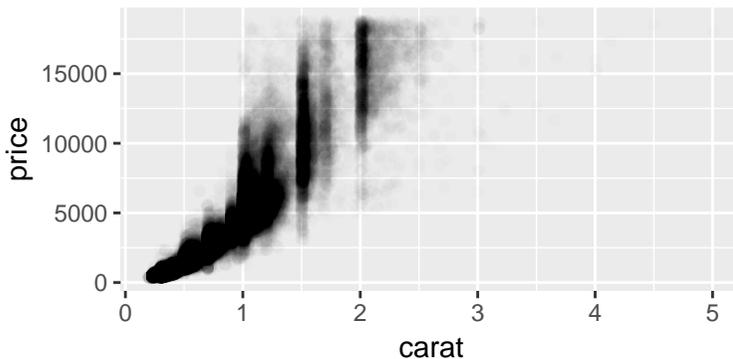
diamonds %>%
  ggplot() +
  geom_point(aes(carat, price))

```



The problem is that we have a lot of points here! To better see what is going on, it helps to make the points somewhat transparent. In the graphics community, transparency is often known as *alpha*, and that is the parameter to change.

```
diamonds %>%
  ggplot() +
  geom_point(aes(carat, price), alpha = 0.01)
```

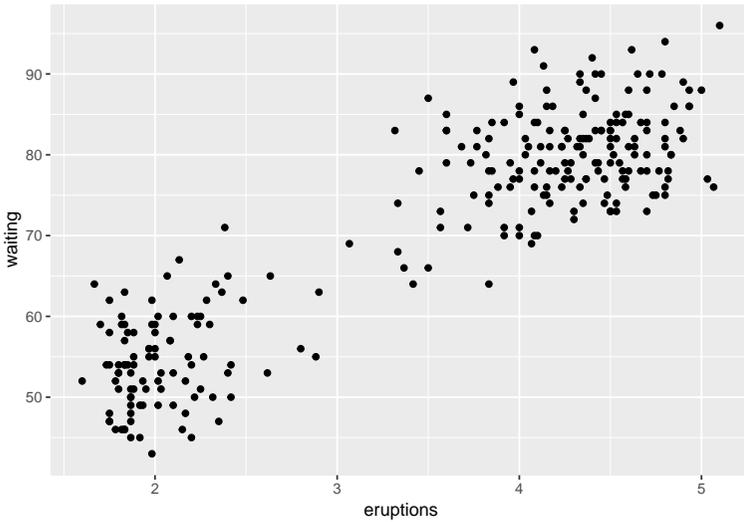


Now we see that same banding around 1, 1.5, and 2 carats that we saw in earlier graphs. We also see that prices tend to concentrate in a particular interval for each carat level.

8.4 Patterns and modeling

Sometimes patterns are easy to spot. A scatterplot of the times between eruptions and the length of eruption of the Old Faithful geyser shows several patterns.

```
ggplot(faithful) +
  geom_point(aes(eruptions, waiting))
```



First, the waiting time until an eruption is positively correlated with the length of the eruption. Second there are definitely two clusters, indicating that there are two types of eruptions.

Patterns are important, because they allow us to make better predictions of variables based on others. This works by using the data to create a **model** of how one or more variables affects the other variables.

Suppose that the relationship between $y = \text{price}$ and $x = \text{carat}$ is polynomial, so of the form $y = x^k$ for some k . Then taking the natural logarithm of both sides gives

$$\ln(y) = k \ln(x).$$

Hence if y varies polynomially in x , then the $\ln(y)$ should have a linear relationship with $\ln(x)$. Linear models are the easiest to find and test for.

We will use the package **modelr** to test this relationship.

```
library(modelr)
mod <- lm(log(price) ~ log(carat), data = diamonds)
summary(mod)
```

Call:

```
lm(formula = log(price) ~ log(carat), data = diamonds)
```

Residuals:

| Min | 1Q | Median | 3Q | Max |
|----------|----------|----------|---------|---------|
| -1.50833 | -0.16951 | -0.00591 | 0.16637 | 1.33793 |

Coefficients:

| Estimate | Std. Error | t value | Pr(> t) |
|----------|------------|---------|----------|
|----------|------------|---------|----------|

```
(Intercept) 8.448661 0.001365 6190.9 <2e-16 ***
log(carat) 1.675817 0.001934 866.6 <2e-16 ***
---
```

```
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
Residual standard error: 0.2627 on 53938 degrees of freedom
Multiple R-squared: 0.933, Adjusted R-squared: 0.933
F-statistic: 7.51e+05 on 1 and 53938 DF, p-value: < 2.2e-16
```

The function `lm` standard for linear model. Without going into the details of the summary, I will just point out here that the very last line saying that the p -value is $2.2 \cdot 10^{-16}$ is considered very strong evidence for a relationship between the two variables.

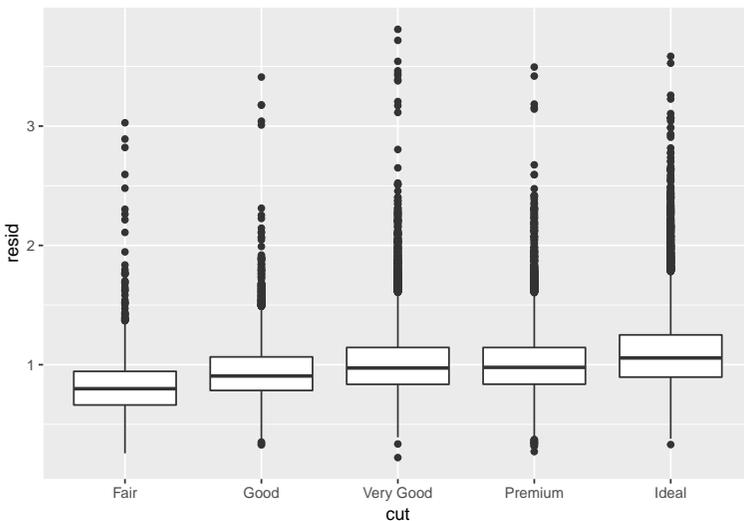
With this model, we can now examine what the price of the diamond is after we have already used our predictive power based on the carat. The difference between the predicted answer and the true answer is called the *residual*.

First let's calculate these residuals, and then exponentiate them (because remember we took the logarithm of the prices earlier.)

```
diamonds2 <- diamonds %>%
  add_residuals(mod) %>%
  mutate(resid = exp(resid))
```

Now we can look at how these residuals behave for different cuts.

```
ggplot(data = diamonds2) +
  geom_boxplot(mapping = aes(x = cut, y = resid))
```



Finally we are now able to see that the price goes up as the cut of the diamond improves. Although not as much as you might think: the carat is far more important to the price than the cut it turns out.

Part II

PREPARING DATA

Data Import Part I

Summary Common ways to organize data in files is as **comma separated files** where each observation is one a separate row, and each variable value is separated by a comma. In **fixed width** files, each variable value gets a fixed number of spaces to record the value.

Read Section 11.1–11.3 of text.

Up until now we've been working with data sets that have been built in to R, but of course the whole point of learning about these packages and functions is so that you can apply them to your own data.

The process of bringing data from a file or website into your computing environment is called *importing* data. Sending data out to a file is *exporting* data.

Definition 21

Data import is the process of reading in data from storage to a programming environment.

Definition 22

Data export is the process of writing data from the programming environment to permanent storage.

Basic R has several tools for data import and export. However, they tend to be slow in practice, and we will be looking primarily at the tools that are part of the tidyverse's **readr** package.

9.1 Comma Separated Files

The most common way of storing data is as a *comma separated file*.

Definition 23

A text document is a **comma separated file** or CSV if it stores data in a table using plain text where the entries for a row are separated by commas.

For example, the USGS keeps the United States Wind Turbine Database (USWTDB) in several formats, one is as a CSV file. Here is a tiny (the full file is 9.7 MB) portion of that CSV file (retrieved 3 Feb 2019 from [https://eerscmap.usgs.gov/uswtodb/data/.](https://eerscmap.usgs.gov/uswtodb/data/))

```
ct_state,t_county,t_fips,p_name,p_year,p_tnum,p_cap,t_manu
CA,Kern County,6029,251 Wind,1987,194,18.43,Vestas
```

There are several functions in **readr** to read comma separated files and their variants.

- **read_csv** reads the standard CSV files.
- **read_csv2** reads a variant of CSV where the symbol between values is a semicolon ; instead of a comma , .
- **read_delim** reads files where the values are separated by any delimiter.
- **read_fwf** reads *fixed width files*. Here each column in the table of data is given using the same number of text characters.
- **read_log** reads Apache style log files. The Apache HTTP Server (which is commonly referred to as just Apache) is a set of tools written in Java for creating and maintaining web servers.

Definition 24

A **fixed width file** gives the same number of spaces in a file for each variable value.

So, for instance, it could be that every eight characters in a row contains the value for a particular variable.

As usual, we'll need the tidyverse library to start.

```
library(tidyverse)
```

All of these functions operate the same way: the first and most important parameter is the file name (with directory) that you are trying to load in. Then come various options that help you parse the file in correctly.

For instance, if the file `uswtodb_v1_3_20190107.csv` was in my working directory under subdirectory `datasets`, I could use the following to load the CSV file into the variable `wind`.

```
wind <- read_csv("datasets/uswtodb_v1_3_20190107.csv")
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   faa_ors = col_character(),
##   faa_asn = col_character(),
##   t_state = col_character(),
##   t_county = col_character(),
##   t_fips = col_character(),
##   p_name = col_character(),
##   t_manu = col_character(),
##   t_model = col_character(),
##   t_img_date = col_character(),
##   t_img_srce = col_character()
## )

## See spec(...) for full column specifications.
```

It creates (by default) a variable that is a tibble.

```
wind %>% select(case_id:t_county)
```

```
## # A tibble: 58,449 x 6
##   case_id faa_ors faa_asn usgs_pr_id t_state t_county
##   <dbl> <chr> <chr> <dbl> <chr> <chr>
## 1 3073438 <NA> <NA> 4979 CA Kern County
## 2 3073442 <NA> <NA> 4989 CA Kern County
## 3 3071562 <NA> <NA> NA CA Kern County
## 4 3073423 <NA> <NA> 4974 CA Kern County
## 5 3072662 <NA> <NA> 5113 CA Kern County
## 6 3004727 <NA> <NA> 5765 CA Kern County
## 7 3071571 <NA> <NA> 5065 CA Kern County
## 8 3073343 <NA> <NA> 4962 CA Kern County
## 9 3071528 <NA> <NA> NA CA Kern County
## 10 3072704 <NA> <NA> 5146 CA Kern County
## # ... with 58,439 more rows
```

The function `read_csv` can also be used to type data directly into a variable.

```
read_csv("
  a, b, c
  1, 2, 3
  4, 5, 6")
```

```
## # A tibble: 2 x 3
##       a     b     c
##   <dbl> <dbl> <dbl>
## 1     1     2     3
## 2     4     5     6
```

A useful character in strings is `\n`, which stands for newline, and allows us to write these types of examples more compactly.

```
read_csv("a, b, c\n1, 2, 3\n4, 5, 6")
```

```
## # A tibble: 2 x 3
##       a     b     c
##   <dbl> <dbl> <dbl>
## 1     1     2     3
## 2     4     5     6
```

Comment lines

One thing to note is that the first line is treated as the variable/factor names. Often, however, the first few lines of a file just contains comments about the file.

```
read_csv("This is a test csv file.
          a, b, c\n1, 2, 3\n4, 5, 6")
```

```
## Warning: 3 parsing failures.
## row col expected actual file
## 1 -- 1 columns 3 columns literal data
## 2 -- 1 columns 3 columns literal data
## 3 -- 1 columns 3 columns literal data
```

```
## # A tibble: 3 x 1
##   'This is a test csv file.'
##   <chr>
## 1 a
## 2 1
## 3 4
```

This tends to throw off the `read_csv` function. So we can force the reader to skip lines at the beginning.

```
read_csv("This is a test csv file.
          a, b, c
          1, 2, 3
          4, 5, 6", skip = 1)
```

```
## # A tibble: 2 x 3
##       a     b     c
##   <dbl> <dbl> <dbl>
## 1     1     2     3
## 2     4     5     6
```

In fact, we can define a comment character so that we can skip any line that starts with that character.

```
read_csv("# This is a test csv file.
         a, b, c
         1, 2, 3
         # Next is the last row.
         4, 5, 6", comment = "#")
```

```
## # A tibble: 2 x 3
##       a     b     c
##   <dbl> <dbl> <dbl>
## 1     1     2     3
## 2     4     5     6
```

Variable names

Some csv files like to jump straight into the data. We can use `col_names = FALSE` so that `read_csv` does *not* treat the first line as the column name.

```
read_csv("# This is a test csv file.
         1, 2, 3
         # Next is the last row.
         4, 5, 6", comment = "#", col_names = FALSE)
```

```
## # A tibble: 2 x 3
##       X1    X2    X3
##   <dbl> <dbl> <dbl>
## 1     1     2     3
## 2     4     5     6
```

R has generously made up names for the tibble, labeling them X1, X2, and X3. Of course, we could also have supplied our own names.

```
read_csv("# This is a test csv file.
         1, 2, 3
         # Next is the last row.
         4, 5, 6", comment = "#",
         col_names = c("Larry", "Moe", "Curly"))
```

```
## # A tibble: 2 x 3
##   Larry Moe  Curly
##   <dbl> <dbl> <dbl>
## 1     1     2     3
## 2     4     5     6
```

One thing about strings: in many programming languages a backslash followed by an `(\n)` indicates a newline of text. So we could get the same file by:

```
read_csv("1, 2, 3\n4, 5, 6",
         col_names = c("Larry", "Moe", "Curly"))
```

```
## # A tibble: 2 x 3
##   Larry Moe  Curly
##   <dbl> <dbl> <dbl>
## 1     1     2     3
## 2     4     5     6
```

Often we need to indicate what in the file indicates that a value is not available. For instance, if we want a period `.` to mean `NA`, we can also set that in the command.

```
read_csv("1, 2, 3\n4, 5, .", na = ".", col_names = FALSE)
```

```
## # A tibble: 2 x 3
##       X1     X2     X3
##   <dbl> <dbl> <dbl>
## 1     1     2     3
## 2     4     5    NA
```

Most of the options that we used here also applies to `read_tsv` (for tab separated values) and `read_fwf` (for fixed width files).

Compared to base R

In base R, the function `read.csv` function performs the same purpose as `read_csv` in `readr`. The `read_csv` improves upon the original in several ways.

- They tend to be much faster at reading files.
- The output is in tibble form.
- Base R functions can depend on the operating system upon which R is running. The `readr` functions operate independently of the OS.

9.2 Parsing vectors

As far as R is concerned, the lines of a comma separated files are strings of characters. So a big part of reading the file is turning strings into integers.

Just like the various **geom** functions add different type of graphics to the canvas, the **parse** functions are the base functions for reading strings of data and turning them into values.

For instance, we can turn strings into integers with the **parse_integer**.

```
parse_integer(c("53", "760", "2343"))
```

```
## [1] 53 760 2343
```

Of course, if our string does not have integers in it, mistakes ensue:

```
parse_integer(c("53", "760", "0.2343"))
```

```
## Warning: 1 parsing failure.
## row col           expected actual
## 3 -- no trailing characters .2343

## [1] 53 760 NA
## attr(,"problems")
## # A tibble: 1 x 4
##   row col expected          actual
##   <int> <int> <chr>          <chr>
## 1     3     NA no trailing characters .2343
```

We can use the **problems** functions to understand what happened with a failure to parse.

```
x <- parse_integer(c("53", "760", "0.2343"))
```

```
## Warning: 1 parsing failure.
## row col           expected actual
## 3 -- no trailing characters .2343
```

```
problems(x)
```

```
## # A tibble: 1 x 4
##   row col expected          actual
##   <int> <int> <chr>          <chr>
## 1     3     NA no trailing characters .2343
```

The most important parse functions are:

| function | purpose |
|---|-------------------------------------|
| <code>parse_logical</code> | parse logical expressions |
| <code>parse_integer</code> | parse integers |
| <code>parse_double</code> | parse real numbers |
| <code>parse_number</code> | parse any type of number |
| <code>parse_character</code> | parse strings |
| <code>parse_factors</code> | parse levels for factors |
| <code>parse_datetime</code> , <code>parse_date</code> , and <code>parse_time</code> | parse various date and time formats |

Dealing with numbers

There are quite a few issues that make parsing number difficult.

- For instance, many countries use the period `.` to separate the integer and fractional parts of a number, while others use the comma `,` to accomplish the same thing. As we have seen, the default behavior of R is to print numbers using `.` for the decimal mark.
- Numbers often are next to special characters that modify them. For instance `$1000` or `10%`.
- In addition to the decimal point, other characters are added to make the number easier to read. For instance, in the United States, a number such as 10^6 might be written `1,000,000`. These grouping characters are different around the world.

To tackle this first problem, `readr` has a parameter `locale` which can be used to change the mark delineating the decimal part of the number. The `locale` function can then be used to create an object suitable for passing to this parameter. This parameter does have defaults. For instance, the following uses the default decimal mark of a period.

```
parse_double("1.23")
```

```
## [1] 1.23
```

If we try the defaults with a different decimal mark, then mistakes happen.

```
parse_double("1,23")
```

```
## Warning: 1 parsing failure.
## row col           expected actual
## 1 -- no trailing characters    ,23
```

```
## [1] NA
## attr(,"problems")
## # A tibble: 1 x 4
##   row   col expected          actual
##   <int> <int> <chr>          <chr>
## 1     1     1    NA no trailing characters ,23
```

But using the `locale` parameter and `locale` function properly can fix this.

```
parse_double("1,23",
             locale = locale(decimal_mark = ","))
```

```
## [1] 1.23
```

The `parse_number` function is intended to help the second problem by stripping out extra modifying characters.

```
parse_number("$100")
```

```
## [1] 100
```

```
parse_number("65 mph")
```

```
## [1] 65
```

```
parse_number("20%")
```

```
## [1] 20
```

```
parse_number("This is the number 342.42.")
```

```
## [1] 342.42
```

Note that the `%` modifier did not actually get applied by `parse_number`: it is up to the user to actually deal with units and their ramifications.

One thing `parse_number` does understand is the grouping character. Again using `locale` and the `locale` function, we can alter what is the grouping symbol.

```
parse_number("$435,274")
```

```
## [1] 435274
```

```
parse_number("$435.274",  
             locale = locale(grouping_mark = "."))
```

```
## [1] 435274
```

Data Import Part II

Read Sections 11.4–11.6 of the text.

10.1 Representing text

We've seen how we can use `parse` functions to bring strings containing numbers into R. However, surprisingly there are also issues with bringing strings of characters into R! This is because of way that characters are encoded in data often varies from system to system.

In order to understand how `parse_character` works, we need to look a bit more deeply at how characters in strings are represented by computers. At the end of the day computers only keep numbers represented by bits in memory, and so we need to have some method of representing (encoding) characters by number.

Numbers in different bases

First a reminder about bits, bytes, binary, decimal, and hexadecimal numbers.

Definition 25

A **bit** is short for a *binary digit*, and is either 0 or 1.

Definition 26

A **byte** consists of 8 bits.

Because bit tends to be too small, most memory sizes in computers are measured in bytes. For instance, in 2019 the computer I am typing this on has 32 GB (gigabytes) of memory, which is $32 \cdot 10^9$ bytes.

For example, 01111001 is a byte. I've written this using base 2, or binary notation. In decimal (base 10) notation, 573 is 5 times 10^2 plus 7 times 10^1 plus 3 times 10^0 . A bit string like 1001 is 1 times 2^3 plus 1 times 2^0 plus 1 times 2^0 , or 9 in decimal notation.

With 8 bits (a byte), one can represent an integer from 0 to 255. With 4 bits (sometimes called a nibble), one can represented a number from 0 to 15. These 16 numbers can also be represented in *hexadecimal* notation.

Definition 27

A **hexadecimal** number is written in base 15.

The digits in hexadecimal use the decimal digits plus the letters a through f for 10 through 15.

| | | | | | | | | | | | | | | | | |
|-------------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| Hexadecimal | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f |
| Decimal | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

Often (but not always) capital letters A through F are used instead of a through f. So to convert 4d to decimal, use

$$4 \cdot 16 + d = 64 + 13 = 77.$$

Having 8 bits, and since each hexadecimal digit can represent 4 bits, a single byte can be represented by a two digit hexadecimal number.

ASCII, ISO LATIN 1, and UTF-8

The simplest standard is the American Standard Code, abbreviated ASCII (pronounced as as-key). In this code, each number from 0 to 127 (which can be represented by 7 bits) stands in for a single character. For instance, 26 is the ampersand character &, and the numbers 65 through 90 represent the capital letters A through Z.

Since a byte contains 8 bits, and so can represent numbers from 0 to $2^8 - 1 = 255$. So what do we do with the extra 128 characters? One way is to use the ISO Latin set of characters, which includes several characters with accents. This character set includes enough characters for complete coverage in 24 language.

A different approach is the UTF-8 standard. Unlike the ASCII and ISO Latin-1 standard, each character here is represented by a variable number of bytes, from 1 to 4.

If the leading bit (most significant) is 0, then the remaining seven bits acts just like the ASCII code. But if the leading bit is 1 (so the number is from 127 through 255), then we also have a second byte in the character. We would write 4d as U+004D. The U+ precedes a character written in UTF-8. So one byte covers U+0000 through U+007F. This format with U+ followed by hexadecimal digits, is called a *Unicode code point*.

If the leading bit is 1 we have a second byte. We use 11 bits to cover characters U+0080 through U+07FF. With three bytes we use 16 bits, and with four bytes we end up with 21 bits covering U+10000 through U+10FFFF. By the time we are at four bytes, we can represent virtually every character used in every language both current and ancient.

UTF in R

In the **readr** package, all the functions by default assume that your data is UTF-8 encoded. If in fact your string uses the ISO Latin 1 standard instead, things will be messed up. As usual, the locale parameter and function comes to the rescue.

As usual, we first load in the tidyverse.

```
library(tidyverse)
```

The `charToRaw` function can be used to obtain the ASCII code of a string reoresented in hexadecimal.

```
charToRaw("Mark Huber")
```

```
## [1] 4d 61 72 6b 20 48 75 62 65 72
```

```
x1 <- "El Ni\xf1o was particularly bad this year"
x2 <- "\x82\xb1\x82\xf1\x82\xc9\x82xbf\x82xcd"
parse_character(x1, locale = locale(encoding = "Latin1"))
```

```
## [1] "El Ni~no was particularly bad this year"
```

```
parse_character(x2, locale = locale(encoding = "Shift-JIS"))
```

```
## [1] "<U+3053><U+3093><U+306B><U+3061><U+306F>"
```

Note the last has encoded the result in UTF-8. Hopefully your document says which encoding it is in. If not, then there is a function `guess_encoding` that will try to figure out the encoding based on the text. It does tend to work better with more text. As input, the function takes the hexadecimal bytes of the file.

```
guess_encoding(charToRaw(x1))
```

```
## # A tibble: 2 x 2
##   encoding confidence
##   <chr>         <dbl>
## 1 ISO-8859-1     0.46
## 2 ISO-8859-9     0.23
```

```
guess_encoding(charToRaw(x2))
```

```
## # A tibble: 1 x 2
##   encoding confidence
##   <chr>         <dbl>
## 1 KOI8-R         0.42
```

Factors

A *factor* in statistics is something that we can measure. (Sometimes it is only used for categorical variables.)

Definition 28

A **level** is a particular value that a factor can take on.

When the **read** functions load in data, they attempt to figure out what the possible levels for each factor is. Of course, they can be wrong about that. The **parse_factor** function can be explicitly told what levels are permitted. They then return an error if it sees a value permitted outside the permitted value.

```
flagcolors <- c("red", "white", "green", "blue")
parse_factor(c("red", "green", "yellow"), levels = flagcolors)

## Warning: 1 parsing failure.
## row col          expected actual
##   3  -- value in level set yellow

## [1] red  green <NA>
## attr(,"problems")
## # A tibble: 1 x 4
##   row col expected          actual
##   <int> <int> <chr>          <chr>
## 1     3  NA value in level set yellow
## Levels: red white green blue
```

Importing dates and times

Unix time, which is also known as POSIX time) is a particular way for describing a particular time. It works by saying the number of seconds that have elapsed since midnight on the first of January in 1970. Several of the R commands are based upon this way of encoding time.

- **parse_datetime** expects to see the date in the ISO8601 format, which puts components from biggest to smallest. That means year first, then month, day, hour, minute, and finally second.

```
parse_datetime("2019-02-09T0829")
```

```
## [1] "2019-02-09 08:29:00 UTC"
```

If you leave off the time, it defaults to midnight.

```
parse_datetime("2019-02-09")
```

```
## [1] "2019-02-09 UTC"
```

- `parse_date` expects to see a year (in four digits) following by `-` or `/`, then the month, then `-` or `/`, and finally the day.

```
parse_date("2019-02-09")
```

```
## [1] "2019-02-09"
```

- `parse_time` takes an hour, minutes, and then optionally seconds, separated by `:`. The time can then be given an `am` or `pm` specification. To use this, it is easier to use the `hms` package in R.

```
library(hms)
parse_time("02:15 pm")
```

```
## 14:15:00
```

```
parse_time("14:15:23")
```

```
## 14:15:23
```

Often when reading in date time data you will be faced with a custom format. You can set up the format for many different possibilities. For example:

```
parse_date("01/02/15", "%m/%d/%y")
```

```
## [1] "2015-01-02"
```

```
parse_date("01/02/15", "%d/%m/%y")
```

```
## [1] "2015-02-01"
```

Characters for the custom dates.

| | |
|--------------|--|
| <i>Year</i> | %Y (4 digit year)
%y (2 digit year) |
| <i>Month</i> | %m (2 digit month)
%b Abbreviated month name such as "Jan"
%B Full name of month such as "January" |
| <i>Day</i> | %d (2 digit day)
%e (can add optional leading space for day) |
| <i>Time</i> | %H Hour given using 0-23
%I Hour given using 0-12 (must also use %p parameter)
%p am/pm indicator
%M minutes
%S integer seconds
%OS real seconds
%Z Time zone
%z Offset from UTC (ex: +0800) |

In addition, **%.** skips any one non-digit character, and **%*** skips any number of non-digits. For **%b** and **%B** you will need to specify the language with locale in order to get the correct month names.

```
parse_date("1 janvier 2015", "%d %B %Y", locale = locale("fr"))

## [1] "2015-01-01"
```

How `read_csv` parses a file

When `readr` tries to read a file, it uses various `parse` functions to read in each data value. The question is: which function should it use for each data type?

The function `guess_parser` function tries to guess at the type of data from reading the first 1000 or so lines in the file. While this works most files, there are exceptional files that can trick this method.

For instance, the first thousand rows might contain a text description. If the rows contain mainly `NA` values, it might read it as a character type instead of integer.

An example of such a type of file is included in the package.

```
challenge <- read_csv(readr_example("challenge.csv"))

## Parsed with column specification:
## cols(
##   x = col_double(),
##   y = col_logical()
## )
```

```
## Warning: 1000 parsing failures.
##   row col                expected      actual

   file
## 1001   y 1/0/T/F/TRUE/FALSE 2015-01-16 'C:/Users/mhuber/
Documents/R/win-library/3.4/readr/extdata/challenge.csv'
## 1002   y 1/0/T/F/TRUE/FALSE 2018-05-18 'C:/Users/mhuber/
Documents/R/win-library/3.4/readr/extdata/challenge.csv'
## 1003   y 1/0/T/F/TRUE/FALSE 2015-09-05 'C:/Users/mhuber/
Documents/R/win-library/3.4/readr/extdata/challenge.csv'
## 1004   y 1/0/T/F/TRUE/FALSE 2012-11-28 'C:/Users/mhuber/
Documents/R/win-library/3.4/readr/extdata/challenge.csv'
## 1005   y 1/0/T/F/TRUE/FALSE 2020-01-13 'C:/Users/mhuber/
Documents/R/win-library/3.4/readr/extdata/challenge.csv'
## .....
.....

## See problems(...) for more details.
```

Lots of problems! Let's try to load in everything as character vectors.

```
challenge2 <- read_csv(readr_example("challenge.csv"),
  col_types = cols(.default = col_character()))
```

By looking at this file in [Viewr](#), we see that after the first 1000 rows, the x values switch to doubles. So we need to alter the way it is loaded in.

You can use the `n_max` parameter to set an upper limit on how many rows are read in so that you can deal with the parsing issues before loading in the rest of the data.

Moreover, the y values start off as `NA`, but the rest are dates and times. Fixing these two allows us to load the file correctly.

```
challenge <- read_csv(
  readr_example("challenge.csv"),
  col_types = cols(
    x = col_double(),
    y = col_date()
  )
)
```

Another approach is to force the reader to look at more data.

```
challenge <- read_csv(readr_example("challenge.csv"), guess_max

## Parsed with column specification:
## cols(
```

```
## x = col_double(),
## y = col_date(format = "")
## )
```

Exporting data

Once we have completed a data analysis, it is sometimes the case that we wish to write out a transformed version of the file. Unsurprisingly, the functions that do so begin with **write**. For instance, we could use

```
write_csv(challenge, "challenge.csv")
```

to put out the file in the current directory.

One thing to note is that the comma separated value format does not keep the variable types, so you will be starting over from scratch in importing the file.

```
challenge
```

```
## # A tibble: 2,000 x 2
##       x y
##   <dbl> <date>
## 1   404 NA
## 2  4172 NA
## 3   3004 NA
## 4    787 NA
## 5     37 NA
## 6  2332 NA
## 7  2489 NA
## 8  1449 NA
## 9  3665 NA
## 10 3863 NA
## # ... with 1,990 more rows
```

```
write_csv(challenge, "challenge-2.csv")
read_csv("challenge-2.csv")
```

```
## Parsed with column specification:
## cols(
##   x = col_double(),
##   y = col_logical()
## )

## Warning: 1000 parsing failures.
## row col expected actual file
## 1001 y 1/0/T/F/TRUE/FALSE 2015-01-16 'challenge-2.csv'
```

```
## 1002   y 1/0/T/F/TRUE/FALSE 2018-05-18 'challenge-2.csv'
## 1003   y 1/0/T/F/TRUE/FALSE 2015-09-05 'challenge-2.csv'
## 1004   y 1/0/T/F/TRUE/FALSE 2012-11-28 'challenge-2.csv'
## 1005   y 1/0/T/F/TRUE/FALSE 2020-01-13 'challenge-2.csv'
## .....
## See problems(...) for more details.

## # A tibble: 2,000 x 2
##       x y
##   <dbl> <lgl>
## 1   404 NA
## 2  4172 NA
## 3  3004 NA
## 4   787 NA
## 5    37 NA
## 6  2332 NA
## 7  2489 NA
## 8  1449 NA
## 9  3665 NA
## 10 3863 NA
## # ... with 1,990 more rows
```

From the message we see that the import lost the date format for the *y* variable. However, there are some nice features of the `write_csv` command.

- It always writes out characters in UTF-8 format.
- The dates are always written out in ISO8601 format.

If you do need a complete copy of your variable, then it is possible to write out your data using RDS, the binary format that R uses.

```
write_rds(challenge, "challenge.rds")
read_rds("challenge.rds")
```

```
## # A tibble: 2,000 x 2
##       x y
##   <dbl> <date>
## 1   404 NA
## 2  4172 NA
## 3  3004 NA
## 4   787 NA
## 5    37 NA
## 6  2332 NA
## 7  2489 NA
## 8  1449 NA
## 9  3665 NA
## 10 3863 NA
## # ... with 1,990 more rows
```

Another option if you want to be able to transfer to other programming languages (such as Python) is to use the **feather** package to save the file.

```
# install.packages{feather}
library(feather)
write_feather(challenge, "challenge.feather")
read_feather("challenge.feather")
```

```
## # A tibble: 2,000 x 2
##       x y
##   <dbl> <date>
## 1   404 NA
## 2  4172 NA
## 3  3004 NA
## 4   787 NA
## 5    37 NA
## 6  2332 NA
## 7  2489 NA
## 8  1449 NA
## 9  3665 NA
## 10 3863 NA
## # ... with 1,990 more rows
```

Other Formats

There are other packages for loading in other data sets.

* **haven** allows us to load in SPSS, Stata, and SAS files.

- **readxl** reads in Microsoft Excel files (.xls and .xlsx).
- **DBL** allows you to run SQL queries against a database and get a data.frame in return.

Tidy Data

Summary In **tidy** data, 1) each row corresponds to an observation, 2) each column corresponds to a variable, and 3) each entry only contains a single value. We have several commands for tidying data.

Tidying data

| | |
|-----------------|--|
| spread | Turns entries into variable names |
| gather | Turns variables into entries (values) |
| separate | When entry is two values, separates into two variables (columns) |
| unite | Combines two variables (columns) into one variable |

We also have some functions for dealing with missing values. A missing value **NA** is **explicit** if we write it out directly in the data table, and **implicit** if we remove the row that has an **NA** value. Also, sometimes **NA** values appear in data to indicate that the entry is the same as the one above it. We have functions to deal with these situations.

Missing values

| | |
|-----------------|--|
| complete | Finds missing observations and explicitly makes them NA |
| fill | Changes NA entries to value of the entry above |

Read Sections 12.1–12.5 of text.

So far we've learned how to visualize data, transform data, and import data. All of these tools expect to be given the data in a tidy form.

Recall that for a statistician, a *variable* or *factor* is something that we can measure. A *level* is the different values that a factor can take on. In tabulating data, often the levels are used for the column names. But this makes the data difficult to analyze. A better way is to have each column correspond to a variable, each row to an observation, and then the entries at each row and column should be a level value.

When this is how the data is organized, we call the data *tidy*.

Definition 29

A dataset is in **tidy** form when

1. Each row corresponds an observation.
2. Each column corresponds to a variable.
3. Each entry only contains a single value.

Fortunately, when these properties

For instance, consider the following variable `table1` that is built in to the **tidyr** package.

```
> table1
# A tibble: 6 x 4
  country    year cases population
  <chr>    <int> <int>    <int>
1 Afghanistan 1999     745  19987071
2 Afghanistan 2000    2666  20595360
3 Brazil      1999   37737  172006362
4 Brazil      2000   80488  174504898
5 China       1999  212258 1272915272
6 China       2000  213766 1280428583
```

Each of the six observations occupies its own row. Each column corresponds to a unique variable, and each of the $24 = 6 \cdot 4$ entries of the table corresponds to a single value.

In contrast, here is the same data, but organized differently. Here

```
> table2
# A tibble: 12 x 4
  country    year type          count
  <chr>    <int> <chr>          <int>
1 Afghanistan 1999 cases           745
2 Afghanistan 1999 population  19987071
3 Afghanistan 2000 cases           2666
4 Afghanistan 2000 population  20595360
5 Brazil      1999 cases           37737
6 Brazil      1999 population  172006362
7 Brazil      2000 cases           80488
8 Brazil      2000 population  174504898
9 China       1999 cases           212258
10 China      1999 population 1272915272
11 China      2000 cases           213766
12 China      2000 population 1280428583
```

Here the two variables `case` and `population` have been turned into a `type` variable, and their values have been put into a `count` variable.

This contains the same information, but it far more difficult to work with.

- This format obscures the fact that there are 6 data points. By taking two different types of data and conflating them, the fact that we are working with countries at various years is lost.
- It becomes more difficult to analyze. If we look just at the mean of the counts the overall result conflates the cases and the population values. A single column (if possible) should always have the same units and be measuring the same thing. Here two different things (with different units) are being measured. In tidy data, the entries in each column are all measuring the same thing.

There are many reasons why the data you encounter in practice is not tidy. The two main reasons are the following.

1. Unless you have trained in analyzing tidy data, you simply might not think to organize your data in a tidy fashion. Despite its simplicity, the tidy principle is not self-apparent.
2. Data is often organized in a way to make recording the data as efficient as possible, not for analysis of the data.

So that means we have to learn tools that take data sets that might be untidy and turn them into tidy data sets.

11.1 Spread

The first tool we will introduce is `spread`, which deals with situations like those found in `table2`. We wish to take the 12 rows, and change them back to the six observations that we know exist by combining rows with different `type` entries.

This function is called `spread` because it will make our table wider (as well as shorter) by removing the `count` variable and introducing an equivalent of a `cases` variable and a `population` variable.

The `spread` function has two main parameters. The first, `key`, is the variable that holds the entries that we will turn into separate variables. The second, `value` is the name of the existing variable with names for each of the variables created under the `key` variable. The result is something like:

```
table2 %>% spread(key = type, value = count)
```

```
## # A tibble: 6 x 4
##   country      year  cases population
##   <chr>      <int> <int>      <int>
## 1 Afghanistan 1999     745  19987071
## 2 Afghanistan 2000    2666  20595360
## 3 Brazil      1999   37737  172006362
## 4 Brazil      2000   80488  174504898
## 5 China       1999  212258  1272915272
## 6 China       2000  213766  1280428583
```

Now each variable has its own column, and each row in each variable contains a value rather than a variable name.

11.2 Gather

We use `spread` when a column contains variable names rather than values. What about when a column name is a value rather than a variable name? In that case, we use `gather`.

This often happens when dealing with numerical data. Suppose we reorganize our table of data in yet another fashion.

```
> table4a
# A tibble: 3 x 3
  country    `1999` `2000`
* <chr>      <int> <int>
1 Afghanistan    745  2666
2 Brazil        37737 80488
3 China         212258 213766
```

You can see that ``1999`` and ``2000`` are not true variable names because there is nothing to measure here. Instead, they are really values that should have been assigned to a variable `year`. Unlike with `spread`, we are going to have to tell `gather` exactly which of the existing variable names are actually values. Then we use `key` to say what the name of the new variable should be, and finally `values` tells us what the name of the new value variable should be.

```
table4a %>% gather(`1999`, `2000`, key = year, value = cases)
```

```
## # A tibble: 6 x 3
##   country    year    cases
##   <chr>      <chr> <int>
## 1 Afghanistan 1999     745
## 2 Brazil      1999  37737
## 3 China       1999 212258
## 4 Afghanistan 2000     2666
## 5 Brazil      2000  80488
## 6 China       2000 213766
```

Of course, you notice that this table does not contain all the information from `table1`. The rest of the data is contained in a variable `table4b`:

```
> table4a
# A tibble: 3 x 3
  country    `1999` `2000`
* <chr>      <int> <int>
1 Afghanistan    745  2666
2 Brazil        37737 80488
3 China         212258 213766
```

After tidying up this data in the same way as for `table4a`, we want to combine the two resulting tables into one single table. A command for combining multiple tables into one is the `left_join` command. When you have your data dispersed over multiple tables, it is called a *relational database*, and we will cover in-depth methods for dealing with this situation later in the text.

For now, we'll use `left_join` to bring things together and recreate our tidy data set.

```
tidy4a <- table4a %>%
  gather('1999', '2000', key = year, value = cases)
tidy4b <- table4b %>%
  gather('1999', '2000', key = year, value = population)
left_join(tidy4a, tidy4b)

## Joining, by = c("country", "year")

## # A tibble: 6 x 4
##   country    year  cases population
##   <chr>      <chr> <int>      <int>
## 1 Afghanistan 1999     745  19987071
## 2 Brazil      1999   37737  172006362
## 3 China       1999  212258  1272915272
## 4 Afghanistan 2000    2666  20595360
## 5 Brazil      2000   80488  174504898
## 6 China       2000  213766  1280428583
```

11.3 Separate

Another problem that can prevent data from being tidy is when the table tries to hold values for two variables inside one entry. For example:

```
> table3
# A tibble: 6 x 3
  country    year rate
* <chr>      <int> <chr>
1 Afghanistan 1999 745/19987071
2 Afghanistan 2000 2666/20595360
3 Brazil      1999 37737/172006362
4 Brazil      2000 80488/174504898
5 China       1999 212258/1272915272
6 China       2000 213766/1280428583
```

This is pretty uncommon, the widespread use of spreadsheets tends to discourage this sort of thing. Still, it is often the case that a name and ID number, or first and last name, get combined into one variable, and we often want to separate entries into their different variables.

In the case of `table3`, the entries under the variable `rate` actually contain two values, not one. We can use the `separate` function to do this. The syntax is straightforward: we provide `separate` with the variable name to split, the new names of the variables, and the delimiting character that separates the values in each entry.

The only problem here is that it choose to treat them as through the entries were character strings rather than the integers that they are. By setting the `convert` to `TRUE`, we tell the function to guess at what type of variable we are dealing with.

```
table3 %>% separate(rate, into = c("cases", "population"))
```

```
## # A tibble: 6 x 4
##   country      year cases population
##   <chr>      <int> <chr>   <chr>
## 1 Afghanistan 1999 745     19987071
## 2 Afghanistan 2000 2666    20595360
## 3 Brazil      1999 37737   172006362
## 4 Brazil      2000 80488   174504898
## 5 China       1999 212258  1272915272
## 6 China       2000 213766  1280428583
```

```
table3 %>% separate(rate, into = c("cases", "population"),
                    convert = TRUE)
```

```
## # A tibble: 6 x 4
##   country      year cases population
##   <chr>      <int> <int>     <int>
## 1 Afghanistan 1999     745     19987071
## 2 Afghanistan 2000     2666    20595360
## 3 Brazil      1999    37737   172006362
## 4 Brazil      2000    80488   174504898
## 5 China       1999   212258  1272915272
## 6 China       2000   213766  1280428583
```

11.4 Unite

The `unite` function does precisely the opposite of `separate`, it brings two variables together into one. This can be useful when, for instance, the century and two digit year within the century have been separated into separate variables. Consider `table5`:

```
> table5
# A tibble: 6 x 4
  country      century year rate
* <chr>      <chr>   <chr> <chr>
1 Afghanistan 19      99    745/19987071
```

| | | | | |
|---|-------------|----|----|-------------------|
| 2 | Afghanistan | 20 | 00 | 2666/20595360 |
| 3 | Brazil | 19 | 99 | 37737/172006362 |
| 4 | Brazil | 20 | 00 | 80488/174504898 |
| 5 | China | 19 | 99 | 212258/1272915272 |
| 6 | China | 20 | 00 | 213766/1280428583 |

The function `unite` also has a straightforward syntax: tell it the new name of the variable, followed by one or more variables you wish to unite.

```
table5 %>% unite(col = year, century, year)
```

```
## # A tibble: 6 x 3
##   country      year  rate
##   <chr>        <chr> <chr>
## 1 Afghanistan 19_99 745/19987071
## 2 Afghanistan 20_00 2666/20595360
## 3 Brazil      19_99 37737/172006362
## 4 Brazil      20_00 80488/174504898
## 5 China       19_99 212258/1272915272
## 6 China       20_00 213766/1280428583
```

To denote the combination, `unite` uses an underscore character (`_`) to separate values. For the year, we don't want that. We can use the parameter `sep` to change the parameter, or eliminate it entirely by giving it a blank string (`" "`).

```
table5 %>% unite(col = year, century, year, sep = " ")
```

```
## # A tibble: 6 x 3
##   country      year  rate
##   <chr>        <chr> <chr>
## 1 Afghanistan 1999 745/19987071
## 2 Afghanistan 2000 2666/20595360
## 3 Brazil      1999 37737/172006362
## 4 Brazil      2000 80488/174504898
## 5 China       1999 212258/1272915272
## 6 China       2000 213766/1280428583
```

11.5 Missing Values

Missing data can be denoted in two different ways.

- *Explicitly*. This is when we give in the value `NA`.
- *Implicitly*. This is when we have an observation missing from the table.

For instance, if our original table had read

```
# A tibble: 5 x 4
  country    year cases population
  <chr>      <int> <int>      <int>
1 Afghanistan 1999     745    19987071
2 Afghanistan 2000    2666    20595360
3 Brazil      1999   37737   172006362
4 Brazil      2000   80488   174504898
5 China       1999  212258  1272915272
```

there is no entry with `country=China` and `year=2000`. That data is *implicitly* missing. Whereas if we had

```
# A tibble: 6 x 4
  country    year cases population
  <chr>      <int> <int>      <int>
1 Afghanistan 1999     745    19987071
2 Afghanistan 2000    2666    20595360
3 Brazil      1999   37737   172006362
4 Brazil      2000   80488   174504898
5 China       1999  212258  1272915272
6 China       2000      NA         NA
```

the data for China in the year 2000 is *explicitly* missing.

We can create a simple data set to illustrate this. Let's consider `table4a` again, but suppose the case data from year 2000 for China was missing.

```
table7 <- table4a
table7[3,3] <- NA
table7
```

```
## # A tibble: 3 x 3
##   country    `1999` `2000`
## * <chr>      <int> <int>
## 1 Afghanistan     745     2666
## 2 Brazil          37737    80488
## 3 China          212258      NA
```

Let's gather the data to change the values 1999 and 2000 from variable names to entries.

```
table7 %>% gather(year, cases, c("1999", "2000"))
```

```
## # A tibble: 6 x 3
##   country    year cases
##   <chr>      <chr> <int>
```

```
## 1 Afghanistan 1999      745
## 2 Brazil      1999    37737
## 3 China       1999   212258
## 4 Afghanistan 2000     2666
## 5 Brazil      2000    80488
## 6 China       2000         NA
```

Note that it automatically created an explicit **NA** entry for China in the year 2000. What if we wanted a table without that last missing observation? To make it implicit we could set `na.rm` to **TRUE** in the call to `gather`.

```
table7 %>% gather(year, cases, c("1999", "2000"),
                 na.rm = TRUE)
```

```
## # A tibble: 5 x 3
##   country    year  cases
##   <chr>      <chr> <int>
## 1 Afghanistan 1999     745
## 2 Brazil      1999   37737
## 3 China       1999  212258
## 4 Afghanistan 2000     2666
## 5 Brazil      2000   80488
```

On the other hand, sometimes we want to make implicit missing values explicit. That is exactly what the `complete` command does. We must tell it the variable combination that we think is hiding implicit missing values.

```
table8 %>% complete(country, year)
```

```
## # A tibble: 6 x 3
##   country    year  cases
##   <chr>      <chr> <int>
## 1 Afghanistan 1999     745
## 2 Afghanistan 2000     2666
## 3 Brazil      1999   37737
## 4 Brazil      2000   80488
## 5 China       1999  212258
## 6 China       2000         NA
```

Another issue we encounter is entries marked **NA** to indicate that the entry should have the same value as the entry above it. This is common in census data, for example, where the surname of one member of the household is often repeated for all members of the household. This type of data can be tidied with the `fill` command.

We also use **NA** to denote entries that match the entry above. For instance, our country data might have been written as in `table9`

```
table9 <- table1
table9[seq(2, 6, by = 2), 1] <- rep(NA, 3)
table9
```

```
## # A tibble: 6 x 4
##   country      year  cases population
##   <chr>      <int> <int>      <int>
## 1 Afghanistan 1999     745  19987071
## 2 <NA>        2000    2666  20595360
## 3 Brazil      1999   37737  172006362
## 4 <NA>        2000   80488  174504898
## 5 China       1999  212258  1272915272
## 6 <NA>        2000  213766  1280428583
```

```
fill(table9, country)
```

```
## # A tibble: 6 x 4
##   country      year  cases population
##   <chr>      <int> <int>      <int>
## 1 Afghanistan 1999     745  19987071
## 2 Afghanistan 2000    2666  20595360
## 3 Brazil      1999   37737  172006362
## 4 Brazil      2000   80488  174504898
## 5 China       1999  212258  1272915272
## 6 China       2000  213766  1280428583
```

11.6 Cleaning data

Even when data is in tidy form, it might end up being somewhat *dirty*. This happens when there are errors in the data file or misnamed variables. Here are two extra tools that are useful for cleaning up such data sets.

- **rename** allows us to efficiently change the name of a variable in a tibble. For example: `rename(oldvariablename = "newvariablename")`
- **str_replace** takes a string and replaces it with another string. This is often used either before or after a call to **separate** or **unite** to prepare the data. For example:

```
mutate(key = str_replace(key, "oldstring", "newstring"))
```

A mathematical model of data

Summary A **set** consists of unordered elements. A **subset** of a set B consists only of elements from B . An **n -tuple** (aka **observation**) is an ordered collection (a_1, \dots, a_n) where each a_i belongs to a specified set A_i . The set of n -tuples is called the **Cartesian product** of the sets A_1, \dots, A_n .

A set of **observations** form a **relation** (aka **data table**). A **relational database** consists of one or more relations. A **key** is a set of variables whose values always uniquely identifies the observation. A key which is artificially created to make a relation is called a **surrogate key**. One key from the relation is designated as the **primary key**.

Now that we understand tidy data, we are ready to build a mathematical model of what exactly a table of data is. To accomplish this, we need to understand the mathematical notion of a *set*.

12.1 Sets

In logic, we can begin with just two undefined terms:

True, False

and every statement is either true or false.

Definition 30

A **logical statement** is either true or false.

Another undefined term is object. The idea of a simple set is that for a particular object, we can say if the set contains that object, or if it does not contain that object.

Definition 31

Call A a **set** if for any object x , the statement “ A contains x ” is a logical statement that evaluates to either true or false.

Definition 32

If for set A , the statement A contains x is true, say that x is an **element** of the set A .

Notation 1

If a is an element of set A , write $a \in A$. If A is a finite set, then we can write out the elements of A by enclosing them in curly braces. Order does not matter for sets, so

$$A = \{a, b, c\} = \{b, a, c\}.$$

By this we mean that $a \in A$, $b \in A$, and $c \in A$ are all true statements, and if $d \neq a$ and $d \neq b$ and $d \neq c$, then $d \in A$ is false.

Some notes.

- If $s_1 \in A$ and $s_2 \in A$ is true, then both s_1 and s_2 are elements of A . There is no notion of order between s_1 and s_2 .
- Either $s \in A$ or $s \notin A$. There is no notion of the *number of times* s appears in A . The sets $\{a, a, b\}$ and $\{a, b\}$ are the same.

Note Sets can be defined as much more complicated objects than as given here. However, for sets applied to data, this is the full generality that we need.

A *subset* A of a set B is a set of elements that also all appear in B . To define this, we need the logical notion of \forall .

Definition 33

Say that statement $q(a)$ is true **for all** $a \in A$ if whenever $a \in A$ is true, $q(a)$ also evaluates to true. Write this as

$$(\forall a \in A)(q(a)).$$

Example 1

For all $x \in \{3, 4, 5\}$, $x^2 > 8$ is a true statement. Write $(\forall x \in \{3, 4, 5\})(x^2 > 8)$.

The statement $(\forall x \in \{1, 2, 3\})(x^2 > 8)$ is a false statement, because there is a value of x in $\{1, 2, 3\}$ (actually either $x = 1$ or $x = 2$) such that $x^2 \leq 8$.

Definition 34

Say that set A is a **subset** of B (write $A \subseteq B$) if

$$(\forall a \in A)(a \in B).$$

Recall that in statistics, we have the notion of a variable (aka factor) that takes on a value that comes from the levels of the factor. In this case, the levels form a set of possibilities, and the variable must come from that set. For instance, if A_i are the levels for variable x_i , then we must have $x_i \in A_i$.

Suppose I take an element from A_1 , one from A_2 , and so on up to A_n , and put them in order. Because they are in order, we use parenthesis “(” and “)” to surround them. The result is an element of the *Cartesian product* of the sets.

Definition 35

Let A_1, \dots, A_n be sets. Then the **Cartesian product** of the sets is written $A_1 \times A_2 \times \dots \times A_n$, and is the set

$$\{(a_1, \dots, a_n) : a_1 \in A_1, \dots, a_n \in A_n\}.$$

Definition 36

An element of the Cartesian product $A_1 \times \dots \times A_n$ is called an **n -tuple**, or an **observation**.

Mathematically, an *observation* or *row* of a table is just an n -tuple, where n is the number of variables in each observation.

Example 2

Let $A_1 = \{a, b\}$ and $A_2 = \{c, d, e\}$. Then the Cartesian product of A_1 and A_2 is

$$A_1 \times A_2 = \{(a, c), (a, d), (a, e), (b, c), (b, d), (b, e)\}.$$

Then (a, c) might be an observation, as might (b, c) .

Definition 37

A **relation**, or **data table** is a subset of the Cartesian product $A_1 \times \dots \times A_n$.

Example 3

For $A_1 = \{a, b\}$ and $A_2 = \{c, d, e\}$, a relation (data table) could be:

| A_1 | A_2 |
|-------|-------|
| a | c |
| b | d |
| b | e |

In R, a `data.frame` or `tibble` are just ways of representing a relation (table) inside the computing environment. Because our data consists of relations in tidy data, we call data represented in this fashion a *relational database*.

Definition 38

In a **relational database**, all data is represented using one or more relations.

Every time we manipulate data with **filter** or **select**, we are creating a new relation. The **select** command creates a relation over a Cartesian product of fewer sets (variables), while the **filter** command creates a relation over the Cartesian product with the same sets but with fewer observations.

A command like **arrange** does not change the relation at all. Instead, it merely affects how the relation is represented inside of the computing environment. It does not affect the mathematics of data, but does affect the computer science aspects of the data. (How the relation is stored.)

12.2 Keys

This definition has two important consequences

- The order of the observations (n -tuples) in the table (relation) does not matter.
- Observations (n -tuples) must be *unique*, that is, they cannot be repeated.

For instance,

| First name | Last name | Party |
|------------|-----------|------------|
| Tammy | Baldwin | Democrat |
| John | Barrasso | Republican |
| Marsha | Blackburn | Republican |

and

| First name | Last name | Party |
|------------|-----------|------------|
| John | Barrasso | Republican |
| Tammy | Baldwin | Democrat |
| Marsha | Blackburn | Republican |

form the same table of data.

Because each observation is unique, there will exist (at least one) subset of variables that uniquely identifies each observation. We call such a subset a *key*. That is, given the values of the observation for the key variable(s), there is only one observation in the relation that has those values. Mathematically, we can write this out as follows.

Definition 39

Consider a relation where each observation $x \in A_1 \times \cdots \times A_n$. A **key** is a subset of variables, $D \subseteq \{1, \dots, n\}$, such that for each $y \in \times_{d \in D} A_d$, there is a unique observation x with $x(D) = y$.

In the last table, the set of variables {First name, Last name} form a key because together they uniquely identify the observation, but {Party} is not a key, since knowing that the observation has Party = Republican is not enough to identify the observation.

Of course, the set of variables {First name, Last name, Party} also forms a key by definition, since knowing the entirety of the observation should uniquely identify the observation.

In fact, for any relation, the set of all variables should serve as a key. On the other hand, if two observations are duplicate, then they do not form a relation. In this case, it is customary to create a *surrogate key* to force the observations to be a relation.

Definition 40

Suppose $T \in (A_1 \times \cdots \times A_n)^m$ (so m ordered observations rather than a set of observations.) Then let $A_{n+1} = \{k_1, \dots, k_m\}$ be any of size m . Let

$$T' = \{(t_1, \dots, t_n, k_j) : j \in \{1, \dots, m\}, (t_1, \dots, t_n) = T(j)\}.$$

Then call $\{A_{n+1}\}$ a **surrogate key**.

For instance, consider the following.

| name | age |
|-------------|------------|
| Smith, John | 47 |
| Smith, John | 47 |

This is not a proper data table because the same observation is repeated in two different rows. That is, the same 2-tuple appears twice in the table. When reading this data, we do not know if there was data entry error involved where the same data was entered twice, or if there were actually two John Smiths in the survey.

In order to fix this problem, it is helpful when collecting data to assign a *key* that allows us to uniquely identify different observations. For instance, if we updated the table to include a patient ID number:

| ID | name | age |
|----|-------------|-----|
| 1 | Smith, John | 47 |
| 2 | Smith, John | 47 |

Here **ID** is a surrogate key that forces each 3-tuple to be unique.

Create a surrogate key in R

If we have a table without a key, we can create a surrogate key using the **mutate** and **row_number** functions. (We can then use **select** with **everything** to move that column to the front.)

```
df <- tibble(name = "Smith, John", age = c(47, 47))
df
```

```
## # A tibble: 2 x 2
##   name      age
##   <chr>    <dbl>
## 1 Smith, John  47.
## 2 Smith, John  47.
```

```
df %>%
  mutate(id = row_number()) %>%
  select(id, everything())
```

```
## # A tibble: 2 x 3
##   id name      age
##   <int> <chr>    <dbl>
## 1     1 Smith, John  47.
## 2     2 Smith, John  47.
```

The primary key

There are usually more than one possible key for any given relation. To be useful in practice, a key should contain as few variables as possible. Each relation is typically associated with a single key known as the *primary key*.

Definition 41

One particular key for a relation is designated as the **primary key**.

12.3 Terminology

Because the terms for our data come from mathematics, statistics, earlier database work and later database work such as SQL, most of the entities in a relational database have more than one name.

The following lists many of these equivalent terms.

| Mathematics | Statistics | Relational Database | SQL |
|--------------------|-------------------|----------------------------|------------|
| set | variable, factor | attribute, field | Column |
| n -tuple | observation | tuple, record | Row |
| relation | data table | relation, base relvar | Table |

When we pull data out of a table, mathematically it is just another relation, but we have different terms in other contexts.

| Mathematics | Relational Database | SQL |
|--------------------|----------------------------|------------------|
| relation | derived relvar | View, result set |

At this point it is worth breaking down our terminology.

12.4 History

These ideas go back to a 1970's paper of Edgar F. Codd, who invented the notion of relational databases while working for IBM. His article *A Relational Model of Data for Large Shared Data Banks* spelled out how data represented by relations should be stored and updated to preserve properties as the database grows in size.

Relational data

Summary If a set of variables from a first table is the primary key for a second table, it is known as a **foreign key** in the first table. Such a foreign key can be used to determine if an observation from one table is related to an observation in another table. Data from the two tables can be combined into one table with various tools.

Bringing data from one table to another

| | |
|-------------------|--|
| inner_join | Keeps observations where the key appears in both tables. |
| left_join | Observations where the key appears in first table. |
| right_join | Observations where the key appears in second table. |
| full_join | Observations where the key appears in either table. |

Read Sections 13.1–13.4 of text.

Often we do not just have one table (relation) of tidy data, but multiple tables that work together to give different information about a subject. In this case, we are dealing with *relational data*, and we utilize tools that work across multiples tables.

The tasks we do are similar to those in single tables.

- *Mutating joins* bring data from one table over to another, matching values from a common variable.
- *Filtering joins* removing observations from one table based on the values in another table.
- *Set operations* are operations such as union and intersect.

Let's go back to the package **nycflights13** that holds the `flights` table of data that we looked at earlier.

```
library(tidyverse)
library(nycflights13)
```

Earlier we looked mainly at the `flights` variable, but there are many others within the library. For instance, `airlines` contains the two letter abbreviations of the airlines from the `flights` variable.

```
airlines
```

```
## # A tibble: 16 x 2
##   carrier name
##   <chr>      <chr>
## 1 9E        Endeavor Air Inc.
## 2 AA        American Airlines Inc.
## 3 AS        Alaska Airlines Inc.
## 4 B6        JetBlue Airways
## 5 DL        Delta Air Lines Inc.
## 6 EV        ExpressJet Airlines Inc.
## 7 F9        Frontier Airlines Inc.
## 8 FL        AirTran Airways Corporation
## 9 HA        Hawaiian Airlines Inc.
## 10 MQ       Envoy Air
## 11 OO       SkyWest Airlines Inc.
## 12 UA       United Air Lines Inc.
## 13 US       US Airways Inc.
## 14 VX       Virgin America
## 15 WN       Southwest Airlines Co.
## 16 YV       Mesa Airlines Inc.
```

This is an example of two tables with different information that support one another. Other examples include

- `airports` gives the codes for each of the airports.
- `planes` tells us about the planes identified by `tailnum`.
- `weather` gives the weather at NYC airports broken down by hour.

Any or none of this information might be useful in a particular analysis of the data in `flights`. Note that `carrier` appears both in `flights` and in `airlines`. It is this variables that allows us to link the two tables together.

Definition 42

Suppose a set of variables in one table is a primary key in another table. Then we call the variable a **foreign key**.

In certain cases, the *name* of the variable might be different in the two tables, but that does affect whether or not we are dealing with a foreign key.

Example 4

In the `planes` variable, `tailnum` is a primary key because each plane with its unique tail number appears exactly once. But in the `flights` variable, `tailnum` is a foreign key. It is not a key in `flights` because a single plane makes more than one flight in the table.

To check if a variable (or set of variables) is a key in a table, each value of the key variable(s) must appear exactly once in the table. We can use `count` to find the number of observations where each key value appears, and then `filter` to find those values that appear more than once.

```
planes %>%
  count(tailnum) %>%
  filter(n > 1)

## # A tibble: 0 x 2
## # ... with 2 variables: tailnum <chr>, n <int>
```

```
flights %>%
  count(tailnum) %>%
  filter(n > 1)

## # A tibble: 3,873 x 2
##   tailnum      n
##   <chr>    <int>
## 1 D942DN         4
## 2 N0EGMQ       371
## 3 N10156       153
## 4 N102UW        48
## 5 N103US        46
## 6 N104UW        47
## 7 N10575       289
## 8 N105UW        45
## 9 N107US        41
## 10 N108UW       60
## # ... with 3,863 more rows
```

The importance of foreign keys is that they allow us to pass information from one table to another.

13.1 Left joins

Keys allow us to properly link up observations across two tables.

To illustrate this, let's make a narrower data set

```
flights2 <- flights %>%
  select(year:day, hour, origin, dest, tailnum, carrier)
flights2
```

```
## # A tibble: 336,776 x 8
##   year month   day   hour origin dest   tailnum carrier
##   <int> <int> <int> <dbl> <chr>  <chr> <chr>  <chr>
## 1  2013     1     1     5 EWR    IAH    N14228 UA
## 2  2013     1     1     5 LGA    IAH    N24211 UA
## 3  2013     1     1     5 JFK    MIA    N619AA AA
## 4  2013     1     1     5 JFK    BQN    N804JB B6
## 5  2013     1     1     6 LGA    ATL    N668DN DL
## 6  2013     1     1     5 EWR    ORD    N39463 UA
## 7  2013     1     1     6 EWR    FLL    N516JB B6
## 8  2013     1     1     6 LGA    IAD    N829AS EV
## 9  2013     1     1     6 JFK    MCO    N593JB B6
## 10 2013     1     1     6 LGA    ORD    N3ALAA AA
## # ... with 336,766 more rows
```

We have the carrier names using their two letter abbreviation. What if we wanted the full name of the carrier? For this, we can use the function `left_join`.

```
flights2 %>%
  select(-origin, -dest, -year) %>%
  left_join(airlines, by = "carrier")
```

```
## # A tibble: 336,776 x 6
##   month   day   hour tailnum carrier name
##   <int> <int> <dbl> <chr>   <chr>  <chr>
## 1     1     1     5 N14228 UA      United Air Lines Inc.
## 2     1     1     5 N24211 UA      United Air Lines Inc.
## 3     1     1     5 N619AA AA      American Airlines Inc.
## 4     1     1     5 N804JB B6      JetBlue Airways
## 5     1     1     6 N668DN DL      Delta Air Lines Inc.
## 6     1     1     5 N39463 UA      United Air Lines Inc.
## 7     1     1     6 N516JB B6      JetBlue Airways
## 8     1     1     6 N829AS EV      ExpressJet Airlines Inc.
## 9     1     1     6 N593JB B6      JetBlue Airways
## 10    1     1     6 N3ALAA AA      American Airlines Inc.
## # ... with 336,766 more rows
```

Let's break down what happened here. The first argument to `left_join` was the `airlines`—that's the name of the table whose information we are seeking to add to `flights`. Next we have the `by` parameter, which tells us the name of the key to use. In this case, we want to use `carrier`, since that is a primary key in `airlines`.

Finally, the variable name was added to the table from the table `airlines`

13.2 Types of joins

Although the left join is the most commonly used, there are different types depending on what you are trying to accomplish when you join the two tables together.

Suppose that Table 1 has a set of key values that we will call (k_1, \dots, k_n) . Table 2 has a set of key values that we will call (ℓ_1, \dots, ℓ_m) .

Definition 43

An **inner join** combines tables by creating a new observation whenever $k_i = \ell_j$.

For an inner join, the key has to appear in both of the Tables. By contrast, an outer join keeps keys that appear in at least one of the tables.

Definition 44

An **outer join** combines tables by creating new observations when the key appears in at least one of the tables. A **left join** occurs when we create observations for each of (k_1, \dots, k_n) , a **right join** occurs when we create observations for (ℓ_1, \dots, ℓ_m) , and a **full join** creates observations for both (k_1, \dots, k_n) and (ℓ_1, \dots, ℓ_m) .

For outer joins, the natural question is: what do we do when we try to add a key value that appears in an observation in one table but not the other? The answer is, we treat this as an implicitly missing value in the other table, and then go ahead and make things explicit.

13.3 No duplicate keys

At this point some examples are in order. Let's create some tibbles to try these out on. Suppose Table 1 has a key that has observation values a, b, and c, while Table 2 has a key with observation values a, b, and d.

```
t1 <- tibble(x = c('a', 'b', 'c'),
             color = c('blue', 'red', 'green'))
t2 <- tibble(x = c('a', 'b', 'd'),
             sound = c('high', 'low', 'middle'))
t1
```

```
## # A tibble: 3 x 2
##   x         color
```

```
##   <chr> <chr>
## 1 a     blue
## 2 b     red
## 3 c     green
```

```
t2
```

```
## # A tibble: 3 x 2
##   x     sound
##   <chr> <chr>
## 1 a     high
## 2 b     low
## 3 d     middle
```

Inner joins only create observations where the factor value appears in both tables:

```
t1 %>% inner_join(t2, by = "x")
```

```
## # A tibble: 2 x 3
##   x     color sound
##   <chr> <chr> <chr>
## 1 a     blue  high
## 2 b     red   low
```

Key *a* and *b* were in both tables, so they appear in the combined table. Now let's try a left join. Remember, this makes sure that all the values from the first table are entered.

```
t1 %>% left_join(t2, by = "x")
```

```
## # A tibble: 3 x 3
##   x     color sound
##   <chr> <chr> <chr>
## 1 a     blue  high
## 2 b     red   low
## 3 c     green <NA>
```

```
t1 %>% right_join(t2, by = "x")
```

```
## # A tibble: 3 x 3
##   x     color sound
##   <chr> <chr> <chr>
## 1 a     blue  high
## 2 b     red   low
## 3 d     <NA> middle
```

```
t1 %>% full_join(t2, by = "x")
```

```
## # A tibble: 4 x 3
##   x     color sound
##   <chr> <chr> <chr>
## 1 a     blue  high
## 2 b     red   low
## 3 c     green <NA>
## 4 d     <NA> middle
```

13.4 Duplicate key values

In the example from before, where we did a left join between `flights` and `airlines`, the `carrier` was a primary key in `airlines`. What if it hadn't been unique, what if the same key value appears more than once?

If a key value appears more than once, then we are unsure what data belongs with that observation. Therefore, we need to create a new observation for each of the possible pairings. For instance, if a key value appears twice in Table 1, and three times in Table 2, then it will appear $2 \cdot 3 = 6$ times in the join.

```
t3 <- tibble(x = c('a', 'a'), y = c(1, 2))
t4 <- tibble(x = c('a', 'a', 'a'), z = c(4, 5, 6))
t3 %>% full_join(t4, by = "x")
```

```
## # A tibble: 6 x 3
##   x     y     z
##   <chr> <dbl> <dbl>
## 1 a     1     4
## 2 a     1     5
## 3 a     1     6
## 4 a     2     4
## 5 a     2     5
## 6 a     2     6
```

(One more note. Because each key value appears in both tables, `full_join`, `left_join`, and `right_join` all give the same result.)

13.5 Defining the factors that make up keys

So far we have been specifying the key to use in the join. However, by default all the join commands will try to work out for themselves what variable (or variables) to use as the key. In this case R will report what variables it tried to use for the join.

Natural joins

For instance, we can try to bring together the `flights` table and `weather` table.

```

flights2 %>% left_join(weather) %>% select(-tailnum)

## Joining, by = c("year", "month", "day", "hour", "origin")

## # A tibble: 336,776 x 17
##   year month   day hour origin dest carrier temp dewp
##   <dbl> <dbl> <int> <dbl> <chr> <chr> <chr> <dbl> <dbl>
## 1 2013     1     1     5 EWR   IAH   UA     39.0  28.0
## 2 2013     1     1     5 LGA   IAH   UA     39.9  25.0
## 3 2013     1     1     5 JFK   MIA   AA     39.0  27.0
## 4 2013     1     1     5 JFK   BQN   B6     39.0  27.0
## 5 2013     1     1     6 LGA   ATL   DL     39.9  25.0
## 6 2013     1     1     5 EWR   ORD   UA     39.0  28.0
## 7 2013     1     1     6 EWR   FLL   B6     37.9  28.0
## 8 2013     1     1     6 LGA   IAD   EV     39.9  25.0
## 9 2013     1     1     6 JFK   MCO   B6     37.9  27.0
## 10 2013     1     1     6 LGA   ORD   AA     39.9  25.0
## # ... with 336,766 more rows, and 8 more variables:
## #   humid <dbl>, wind_dir <dbl>, wind_speed <dbl>,
## #   wind_gust <dbl>, precip <dbl>, pressure <dbl>,
## #   visib <dbl>, time_hour <dtm>

```

Success! The join naturally used five variables as the key in order to determine which observations were the same. It was able to bring in all the variables from `weather` that way. This is called a *natural join*.

Differently named keys

Sometimes a key in one table will have a different name in another table. For instance, in the `flights` there is a variable `dest` that is the three letter code for the airport. In the `airport`, the name of the airport is listed by three letter code under the variable `faa`.

This type of situation can be handled within the `by` parameter using the following syntax.

```

flights2 %>%
  left_join(airports, by = c("dest" = "faa")) %>%
  select(origin:name, -carrier)

## # A tibble: 336,776 x 4
##   origin dest   tailnum name
##   <chr> <chr> <chr> <chr>
## 1 EWR   IAH   N14228 George Bush Intercontinental
## 2 LGA   IAH   N24211 George Bush Intercontinental
## 3 JFK   MIA   N619AA Miami Intl
## 4 JFK   BQN   N804JB <NA>
## 5 LGA   ATL   N668DN Hartsfield Jackson Atlanta Intl
## 6 EWR   ORD   N39463 Chicago Ohare Intl
## 7 EWR   FLL   N516JB Fort Lauderdale Hollywood Intl

```

```
## 8 LGA IAD N829AS Washington Dulles Intl
## 9 JFK MCO N593JB Orlando Intl
## 10 LGA ORD N3ALAA Chicago Ohare Intl
## # ... with 336,766 more rows
```

13.6 Merge

It should be noted that the base function `merge` can accomplish the same tasks as `inner_join`, `left_join`, `right_join`, and `full_join`. The syntax uses the parameters `all.x` and `all.y` to determine which keys should be added.

| Tidyverse | Base R |
|-------------------------------|--|
| <code>inner_join(x, y)</code> | <code>merge(x, y)</code> |
| <code>left_join(x, y)</code> | <code>merge(flights2, airlines, all.x = TRUE)</code> |
| <code>right_join(x, y)</code> | <code>merge(flights2, airlines, all.y = TRUE)</code> |
| <code>full_join(x, y)</code> | <code>merge(flights2, airlines, all.x = TRUE, all.y = TRUE)</code> |

So why learn the tidyverse equivalents? Two main reasons.

- As is often the case, the tidyverse functions tend to be much faster in practice.
- Their naming conforms more closely to the commands in SQL, making that language easier to learn later.

Filtering joins and set operations

Summary There are two more types of joins that are used when we only want to look at observations from one data set for a restricted set of keys. Unlike the earlier joins, these do not add data from the second table, instead, only the observations from the first table are kept.

Combining tables

semi_join Keeps observations where the key appears in the second table.

anti_join Keeps obs. where the key does not appear in the second table.

Another type of operation is **set operations** where the variables for both tables are the same, and a new relation is formed which is either the union, intersection, or set difference of the observations.

Combining tables

union Brings together all observations.

intersect Keeps observations in both tables.

setdiff Keeps observations in first table but not in second.

Two useful commands in base R:

Grabbing observations

head Takes the first few observations in a data table.

intersect Takes the last few observations in a data table.

Read Sections 13.5–13.7 of the text.

Recall that a *primary key* for a relation tells us a set of variables such that knowing the values for those variables uniquely identifies the observation.

That is, a primary key only appears once in the set of observations.

14.1 *Joining over observations*

In an inner join, the matching keys are used to add columns of data from one table to another. But sometimes we are not interested in keeping the new data, we just want to know which of the rows appear in the other table.

As an example of this, suppose we have discovered which destinations had the most flights from New York.

```
top_dest <- flights %>%
  count(dest, sort = TRUE) %>%
  head(10)
top_dest
```

```
## # A tibble: 10 x 2
##   dest      n
##   <chr> <int>
## 1 ORD    17283
## 2 ATL    17215
## 3 LAX    16174
## 4 BOS    15508
## 5 MCO    14082
## 6 CLT    14064
## 7 SFO    13331
## 8 FLL    12055
## 9 MIA    11728
## 10 DCA     9705
```

Suppose that we want to filter out those flights that have only these ten as destinations. If we have just one variable, we could easily build a filter, unfortunately, that becomes difficult as soon as more than one variable is involved.

Instead, we can use the `semi_join` function. This is like an inner join in that it only keeps observations that have a key that appears in both tables, but it does not add the second table's data to the first. For instance:

```
flights %>%
  semi_join(top_dest)
```

```
## Joining, by = "dest"
```

```
## # A tibble: 141,145 x 19
##   year month   day dep_time sched_dep_time dep_delay
##   <int> <int> <int>   <int>         <int>         <dbl>
## 1  2013     1     1     542             540           2
```

```
## 2 2013 1 1 554 600 -6
## 3 2013 1 1 554 558 -4
## 4 2013 1 1 555 600 -5
## 5 2013 1 1 557 600 -3
## 6 2013 1 1 558 600 -2
## 7 2013 1 1 558 600 -2
## 8 2013 1 1 558 600 -2
## 9 2013 1 1 559 559 0
## 10 2013 1 1 600 600 0
## # ... with 141,135 more rows, and 13 more variables:
## #   arr_time <int>, sched_arr_time <int>, arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
## #   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dtm>
```

On the other hand, `anti_join` only keeps keys that *do not* appear in the other table. This lets us know which keys are missing from the first table.

```
flights %>%
  anti_join(planes, by = "tailnum") %>%
  count(tailnum, sort = TRUE)
```

```
## # A tibble: 722 x 2
##   tailnum      n
##   <chr>    <int>
## 1 <NA>    2512
## 2 N725MQ    575
## 3 N722MQ    513
## 4 N723MQ    507
## 5 N713MQ    483
## 6 N735MQ    396
## 7 N0EGMQ    371
## 8 N534MQ    364
## 9 N542MQ    363
## 10 N531MQ    349
## # ... with 712 more rows
```

Anti-joins provide a nice reality check that the variable that we think is a key is actually a key for both tables.

14.2 Set operations on tables

Recall that A is a set if for any x , the statement $x \in A$ evaluates to be either true (T) or false (F).

Many operations on sets can be reduced to logic, so it is useful to have notation for logical operations.

In some cases we are dealing with two tables that have exactly the same set of variables, and we are interested in combining the two tables, or only dealing with information that is one table but not the other. In this case we can use *set operations*.

First, let's review the common set operations.

Definition 45

Given sets A and B the **union** of the two sets is $\{c : c \in A \text{ or } c \in B\}$. Write $A \cup B$ for the union of two sets.

Note that or here means the same as logical or, and so is true if one or the other or both are true.

Definition 46

Given sets A and B , the **intersection** of the two sets is $\{c : c \in A \text{ and } c \in B\}$. Write $A \cap B$, AB , or A, B to mean the intersection of A and B .

Again we are using logical and here, so an element is in the intersection of A and B if it is in both A and B .

Definition 47

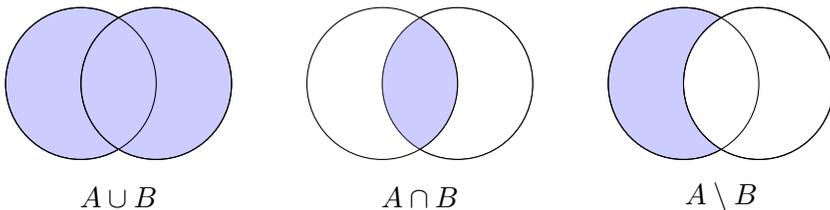
Say that x is in the **complement** of A if $x \notin A$.

Finally, we have the *set difference*.

Definition 48

The **set difference** between A and B is those elements that are in A but not in B . So $\{c : c \in A \text{ and } c \notin B\}$. Write $A \setminus B$.

We can represent these operations pictorially using a Venn Diagram where set A is on the left, and set B is on the right.



So given two tables that each are a set of n -tuples (observations) over the same variables, we can think about taking the union, intersection, and set difference of these using the appropriate functions.

- **union** includes observations from both tables.
- **intersect** includes observations that appear in both tables.

- **setdiff** includes observations in the first table that do not appear in the second table.

Let's whip up an example.

```
df1 <- tribble(
  ~x, ~y,
  "red", "beta",
  "green", "gamma"
)
df2 <- tribble(
  ~x, ~y,
  "red", "beta",
  "yellow", "alpha"
)
```

First we try out **union**:

```
union(df1, df2)
```

```
## # A tibble: 3 x 2
##   x      y
##   <chr> <chr>
## 1 red   beta
## 2 yellow alpha
## 3 green gamma
```

Next **intersect**

```
intersect(df1, df2)
```

```
## # A tibble: 1 x 2
##   x      y
##   <chr> <chr>
## 1 red   beta
```

Finally **setdiff**

```
setdiff(df1, df2)
```

```
## # A tibble: 1 x 2
##   x      y
##   <chr> <chr>
## 1 green gamma
```

It should also be noted that the **union** command is intended to remove any duplicates that appear.

Strings

Summary A **string** is an ordered list of symbols. The **stringr** package contains the tools for dealing with strings in the tidyverse.

String commands

| | |
|-----------------------|--|
| str_length | Returns the number of symbols in the string. |
| str_c | Combines two or more strings into a single string. |
| str_replace_na | Replaces a missing value NA with the string "NA". |
| str_to_upper | Makes all the characters in a string uppercase. |
| str_to_lower | Makes all the characters in a string lowercase. |
| str_sub | Pull out part of a missing string. |

A **regular expression** or **regex** is a sequence of characters used to look for patterns in strings. In R, these expressions can become quite complicated, as they form a particular type of language called a *regular language*.

Read Chapter 14 of the text.

When I was a kid one of my favorite activities was getting all of the Christmas lights out from storage and untangling the giant blob that it had formed itself into. These are typically called a *string* of Christmas lights.

In general, to string something together is to place items on a string, in a particular order. Computer scientists starting using the term almost as early as the first digital computers appeared. In 1944, in *Recursively enumerable sets of positive integers and their decision problems* (<http://www.ams.org/journals/bull/1944-50-05/S0002-9904-1944-08111-1/S0002-9904-1944-08111-1.pdf>) we find the quote

For working purposes, we introduce the letter b, and consider strings of 1's and b's such as 11b1bb1.

By 1958 *A Command Language for Handling Strings of Symbols*, the word string became pretty much how we view it today.

Definition 49

A **string** is an ordered list of symbols from some alphabet.

R fact 10

To indicate a string in R, enclose the symbols with either single quotes (') or double quotes (").

```
s1 <- 'This is a string.'
s2 <- "So is this."
```

```
print(s1)
```

```
## [1] "This is a string."
```

```
print(s2)
```

```
## [1] "So is this."
```

Note that whether you created it with the single quotes or the double quotes, when it prints out it always uses the double quotes. Note that if you want to include a double quote character (") inside your string, you should use single quotes on the outside. Similarly, if you want to include a single quote character (') inside your string, you should use double quotes on the outside.

```
s3 <- "This is a string with an inside 'word'."
s4 <- 'This contains a "quote" in quotes.'
```

```
print(s3)
```

```
## [1] "This is a string with an inside 'word'."
```

```
print(s4)
```

```
## [1] "This contains a \"quote\" in quotes."
```

Notice that for `s4`, the " inside the string was represented as `\`". This is called an *escape character*. To actually see the quote, we can use the `writeLines` function.

```
writeLines(s4)
```

```
## This contains a "quote" in quotes.
```

Definition 50

An **escape character** in a string is a backslash `\` followed by a symbol. Together they have a different meaning inside a string.

Escape characters in R

| | |
|-----------------|-------------------------|
| <code>\'</code> | single quote |
| <code>\"</code> | double quote |
| <code>\n</code> | newline |
| <code>\r</code> | carriage return |
| <code>\t</code> | tab character |
| <code>\u</code> | begin unicode character |

Some older operating systems use the carriage return, but most modern ones use the newline escape character to start a new line. Therefore use of `\n` is recommended.

If you use `\u`, you can follow it with the hexadecimal representation of a Unicode symbol. For instance, to get the degrees symbol, use:

```
"\u00b0"
```

```
## [1] "°"
```

15.1 Helpful string functions

R has some built in commands for dealing with strings, but as usual, we will use the tidyverse alternates. Most of these start with `str_`, which makes them a bit easier to remember. And if you don't remember them in RStudio, you can just start typing `str_` and let autocomplete do its thing.

A simple command to find the length of a string is `str_length`:

```
str_length("abc")
```

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

```
str_length("")
```

```
## [1] 0
```

Combining two strings to make one long string has a special name: we call it *concatenation*.

Definition 51

String concatenation is a binary operator that takes two strings s_1 and s_2 and forms a new string s_3 such that the first $\#(s_1)$ characters in s_3 match s_1 and the last $\#(s_2)$ characters of s_3 matches s_2 . This is written s_1s_2 or sometimes $s_1 + s_2$.

The **concatenation** of two sets of strings S_1 and S_2 consists of

$$S_3 = \{s_1s_2 : s_1 \in S_1, s_2 \in S_2\}.$$

In other words, the concatenation of two sets of strings consists of all the possible ways of concatenating a string from the first set with a string from the second set.

Example 5

If $S_1 = \{ "a", "bt" \}$ and $S_2 = \{ "cd", "ac", "bd" \}$ then

$$S_1S_2 = \{ "acd", "aac", "abd", "btcd", "btac", "btbd" \}.$$

In the **tidyverse**, the string concatenation function is **str_c**.

```
str_c("a", "b", "c")
```

```
## [1] "abc"
```

```
str_c("Cold", " ", "Fusion")
```

```
## [1] "Cold Fusion"
```

Note that combining doesn't work with missing values.

```
str_c("a", NA, "c")
```

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

If we want to convert a missing value **NA** to a string so that we can combine it, we use the **str_replace_na** command.

```
x <- NA
str_c("a", str_replace_na(x), "c")
```

```
## [1] "aNAc"
```

The **str_c** command is vectorized. If you combine a short vector with a long vector, the elements of the short vector will get used as often as necessary to fill the long vector.

```
str_c(c("bad", "jump", "coward", "find"), c("ly", "ing"))

## [1] "badly"      "jumping"    "cowardly"  "finding"
```

Making uppercase and lowercase

Often we are faced with strings that we don't care if they are upper or lower case. The functions `str_to_upper` and `str_to_lower` take care of this for us.

```
test <- c("aBC", "d3&")
str_to_upper(test)
```

```
## [1] "ABC" "D3&"
```

```
str_to_lower(test)
```

```
## [1] "abc" "d3&"
```

The `str_to_title` capitalizes the first letter of each word.

```
str_to_title("And another one bites the dust")
```

```
## [1] "And Another One Bites The Dust"
```

Different languages have different rules for changing from lower to upper case. As always, our friendly `locale` parameter is there to help out.

`str_sub`

You can get part of a string with `str_sub`. The `start` and `end` arguments tell the position of the characters to get from the string.

```
colors <- c("red", "green", "blue")
str_sub(colors, 2, 4)
```

```
## [1] "ed" "ree" "lue"
```

If you use negative numbers, then it counts from the end. So to get the last letter of each string:

```
str_sub(colors, -1, -1)
```

```
## [1] "d" "n" "e"
```

We can also use `str_sub` to assign a subset of a string to be a value. For instance, if you forget about `str_to_title`, to make the first letter of each word uppercase you can use:

```
str_sub(colors, 1, 1) <- str_to_upper(str_sub(colors, 1, 1))
colors

## [1] "Red"    "Green" "Blue"
```

Sorting

We can use `str_sort` (or `str_order`) to put strings in alphabetical order. As with the upper and lower case commands, you can use the `locale` parameter to ensure that you are sorting according to the correct alphabet. For instance, in the Hawaiian alphabet vowels come first, then consonants.

```
greek <- c("beta", "alpha", "iota", "gamma")
str_sort(greek)

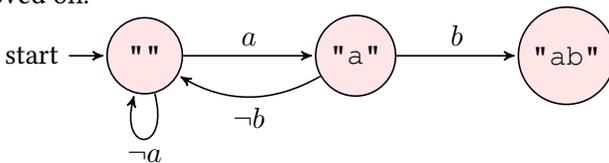
## [1] "alpha" "beta"  "gamma" "iota"
```

```
str_sort(greek, locale = "haw")
```

```
## [1] "alpha" "iota"  "beta"  "gamma"
```

15.2 Searching within strings: finite automata

The next major task we consider is how to search for a pattern within a string. The major rule that we want to enforce (in order to be efficient) is that the string can only be read through once. That is, at each step of our procedure, we get the next character in the string that we are searching and we are unable to look at previous characters once we have moved on.



As the diagram indicates, we start in the " " state. Then we examine the first character of a string. If it is not an a character, then we return to the " " state. But if it is an a , then we move to the "a" state. Next we look at the next character. If it is a b , then we move to the "ab" state, which is a success!

For that reason, we call "ab" a *final state*.

Example 6

If the input string to the above example was "cdagabh", then the set of states the finite automata visits would be:

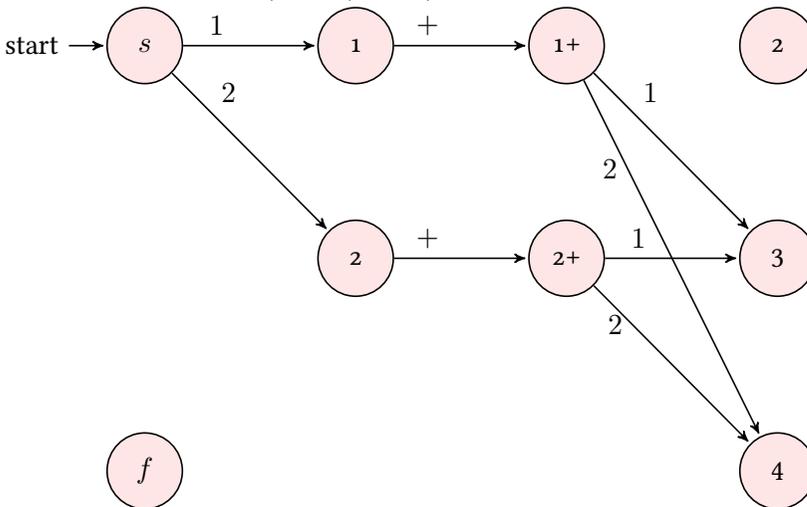
"", "", "", "a", "", "a", "ab",
and then it would stop.

Because there are a finite number of states, this is called a *finite automata*. Formally, a finite automata consists of the states and the rule that tells us how to move from state to state based on the alphabet.

Definition 52

A **finite automaton** consists of a set of states S , an initial state $s \in S$, an alphabet for the input string A , a set of final states $F \subseteq S$, and a rule $r : S \times A \rightarrow S$ that tells us given the state and the next symbol on the input string, to which state we must move next.

A finite automata is a type of computer. For instance, the following finite automata can parse the answer to $1 + 1$, $1 + 2$, $2 + 1$, or $2 + 2$.



Here the rule is if anything other than the listed symbols appear, move back to node f for fail. For instance, if the next symbol from state 1 is anything other than a $+$ symbol, we fail.

This automata does simple addition, but because there are only a finite number of states, it cannot add all integers. We could make this more complex, but no matter how many states we added, it still could not possible implement addition for all of the integers. This is why programming languages such as R have NaN (not a number) types. If a number gets too big (or too small), R will simply output its generic fail message: Nan.

There are patterns that a finite automata can find and patterns that it cannot. In the 1950's, a mathematician named Stephen Kleene proved that finite automata can parse *regular expressions* using a *regular language*. This language is different from any that we have seen so far, and is a very compact but powerful method for describing a finite automata using strings.

Next time we will go into detail about the regular expressions that R can parse.

Regular expressions

Definition 53

Regular expressions (aka **regex** aka **regexp**) is a sequence of characters that define a search pattern within strings.

Regular expressions have been around almost as long as the digital computer. They were invented in the 1950's. Stephen Kleene formalized their description using the notion of a *regular language*, which is quite different from the languages we have used so far. These expressions have the same computing power as a machine that has a finite set of states that responds to inputs. That makes them weaker than a Turing machine (which has an infinite memory), but more powerful than computers with no memory at all that only collate input as it comes in.

Regular expressions became widely used in the Unix operating system. The *grep* command (which stands for **global regular expression print**) became a signature feature of the Unix system.

The simplest kind of regular expression is just a string of alphanumeric characters. We have a match whenever that string is found as a substring of any of our set of strings. For instance, the regex `ta` applied to the names of the `greek` variable from earlier would match in the following way:

beta alpha iota gamma.

In R, you can use the `str_view` function (which uses the package `htmlwidgets`) to look at how these regular expressions work. Consider the following set of strings:

```
greek <- c("alpha", "beta", "gamma", "iota")
```

Matching `"ta"` to this gives

alpha, beta, gamma, iota

Regular expressions have a *wildcard* character that matches any symbol in the string.

Definition 54

The **wildcard** symbol `.` matches any character in a regular expression.

To match `a` followed by any symbol, we would use the regular expression `"a."`, which on the set of strings in `greek` gives

alpha, beta, gamma, iota.

That raises the question, how do we only match `a.` if it is the wildcard character? The answer is to use an escape character `\`. anytime you want to match a period.

That raises another question: if `\` is used for escape characters, how do we match it in a string? Naturally, the answer is to use its escape character, `\\`

In our examples, the expression was matched in any part of the string. In order to *anchor* the expression to the beginning of the string or the end, use `^` to anchor to the beginning, and `$` to anchor to the end of the string. For instance, `"^a"` gives

alpha, beta, gamma, iota,

while `".a$"` gives

alpha, beta, gamma, iota,

If you want the entire string to match the regex exactly, anchor it to both ends. So `"^..t."` gives

alpha, beta, gamma, iota,

Characters that match several symbols

We saw that `.` matches anything (except newline). There are other ways to match more than one, but not all, characters.

| pattern | matches |
|---------------------|---|
| <code>\d</code> | any digit |
| <code>\s</code> | any whitespace such as space, tab, and newline |
| <code>[abc]</code> | Matches a or b or c. |
| <code>[^abc]</code> | Matches any character except a, b, c. |
| <code>[a-z]</code> | Matches any lower case in the Roman alphabet. |
| <code>[A-Z]</code> | Matches any upper case character in the Roman alphabet. |

An important note: remember that inside a regular expression, (which is itself a string) you need to write `\\` to get a single `\`. So you would put something like `\\d` for the wildcard for a digit.

The bracket notation also gives another way of finding wildcard characters. So use `[.]` to search for `.` in a string. This works for

$$\$ \cdot | ? * + () [\{$$

but not

$$] \setminus ^ -$$

There is something similar to logical or. In the context of regular expressions, it is called *alternation*.

Definition 55

If p and q are regular expressions, then $p|q$ is the regular expression that matches either p or q . This is called **alternation**.

Parenthesis

There is an order of operations with regular expressions, but whenever things become unsure or confusing, feel free to add parenthesis to make things clear. For instance,

$$\text{gr(e|a)y}$$

matches either grey or gray.

Repetition

Next we look at how to control how many times a particular expression appears in a string. The default is exactly one. This can be modified using $?$, $+$, or $*$,

| modifier | number of pattern matches |
|------------|----------------------------------|
| $?$ | either no times, or exactly once |
| $+$ | at least once |
| $*$ | zero or more times |
| $\{n\}$ | exactly n |
| $\{n, \}$ | at least n |
| $\{, m\}$ | at most m |
| $\{n, m\}$ | at least n and at most m |

For instance `colou?r` matches both `color` and `colour`.

The $*$ notation turns out to be very useful in theoretical computer science. It is named for the mathematician mentioned earlier who formalized regular expressions, Stephen Kleene.

Definition 56

The **Kleene star** is a unary operator on sets of strings. Given a set of strings S , let S^* be the smallest set such that the empty string is in S^* , each $s \in S$ is in S^* , and for any $s_1, s_2 \in S$, the string concatenation of s_1 and s_2 is in S .

`\begin{example}` Let $S = \{s\}$ where $s = \{\text{"abc"}\}$. Since $s \in S$, the concatenation of s with itself (which is "abcabc") is also in S^* . So is S^4 concatenated with itself three times, and so on. Hence

$$S^* = \{\text{"", "abc", "abcabc", \dots}\}.$$

`\end{example}`

When allowing for repeated expressions, the default behavior is *greedy*, which means that it will try to match as long as possible a string. You can alter this behavior to *lazy* by putting a `?` after the expression.

So `C{2,3}` matches to `MDCCC`LXXXVIII, while `C{2,3}?` matches `MDCC`LXXXVIII.

Repeating wildcard matches

Parenthesis do more than group expressions. Consider an expression of the form $(p)(q)$. Then the *reference* for (p) is 1, since that parenthetical expression appeared first, while the reference for (q) is 2 since that appeared second. (We could have had more parenthesis if we wanted to.)

If a particular string of letters matches p , then from now on `\1` will only match that same string of numbers. So what can we do with that?

Suppose I am interested in discovering strings where the same two letter combination appears twice. For instance, in `banana`, `an` appears twice, as does `pa` in `papaya`. Then we can use $(..)$ to match the first group of two letters. To catch words like this, use the regular expression

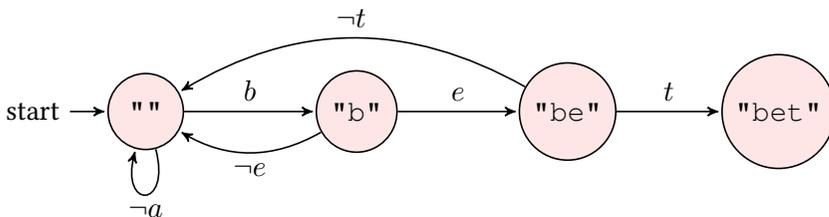
$$(..)\1$$

which says: accept any two characters, then accept only the *same* two characters immediately after. On the `fruit` list of words from the `rcorpora` package gives

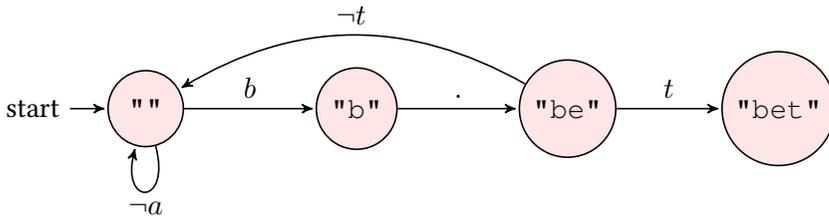
`banana` `coconut` `cucumber` `jujube` `papaya` `salal` `berry`

16.1 Finite Automata for regular expressions

Now let's consider how these regular expressions translate into finite automata. We have now seen how to form the automata for any given string. For instance, the automata for `"bet"` is



Here `"bet"` is the only final node. If we wished to replace the `"e"` in `"bet"` with `"."` that is easy, simply allow any character in its place.



Consider "be+t" = "bee*t"? For instance,

bet_a, beeet_a, bta.

This means that the letters *be* must appear followed by an arbitrary number of *e*'s followed by a *t*. We can use a *directed cycle* in the graph of the finite automata to depict a Kleene star.

Definition 57
 A **directed graph** consists of nodes V connected by edges E . Each edge in E is a 2-tuple of nodes.

Example 7
 For instance, we might have nodes $V = \{A, B, C\}$, and edges $E = \{(A, B), (B, A), (B, C), (C, A)\}$. This can be drawn as follows.

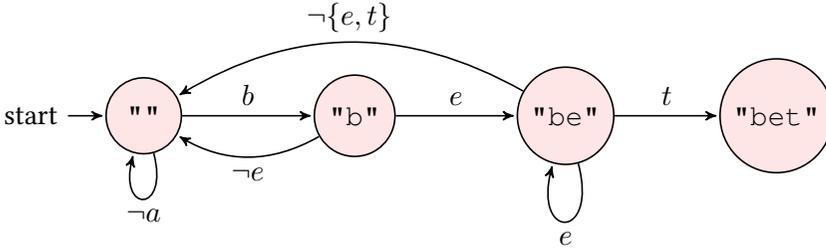
A directed graph with three nodes A, B, and C, each in a pink circle. The edges are:

- A to B
- B to A
- B to C
- C to A

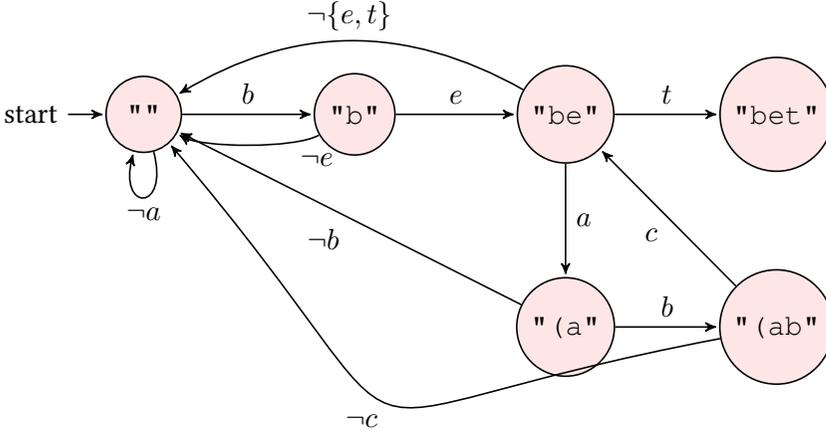
Definition 58
 A **directed cycle** is a n -tuple of nodes v_1, \dots, v_n where $n \geq 2$, $v_1 = v_n$, and for each $i \in \{1, \dots, n - 1\}$, there is a directed edge from v_{i-1} to v_i . The **length** of the cycle is $n - 1$.

Example 8
 In the last graph example, (A, B, A) is a cycle of length 2 and (A, B, C, A) is a cycle of length 4.

For the Kleene star regular expression e^* , the cycle is short, of length 1. This looks like



The length of the cycle will equal the length of the expression the Kleene star is applied to. So for $"b(abc)^*t"$, the automata looks like:



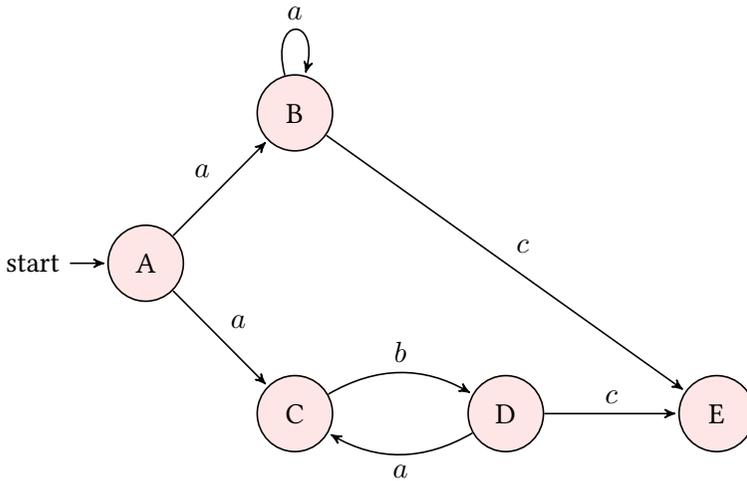
16.2 Nondeterministic finite automata

So far the finite automata that we have been considering are called *deterministic*. Therefore they are sometimes referred to as deterministic finite automata, or **DFA**. From each state, we receive our input character and move to a new state. A *nondeterministic* finite automata, or **NFA** gets two extra possibilities. First, it is possible in an NFA to stay at the current state. Second, it is possible to move not to a single state but to two or more different states simultaneously.

To see why this ability might be useful, consider the regular expression with alternation:

$$(a|ab)^*c$$

When we first see the a character, it might be part of a repeating sequences of a 's, or a repeating sequence of ab 's; those are the alternatives that are possible. Hence we need to be able to travel to 2 different possibilities in order to be able to tell if a path exists.



To simplify the diagram, if in any state we encounter an input that does not lead to an outgoing edge, the match returns to state A . Then E the final node that indicates a match.

What makes this an NFA is from A we see two outgoing nodes marked a . So in some sense the automata takes *both* choices. The final NFA is a success (match) if there exists some path from the start node A to the final node E .

To formally define an NFA,

Definition 59

The **power set** of a finite set S , written 2^S , consists of all subsets of S .

Example 9

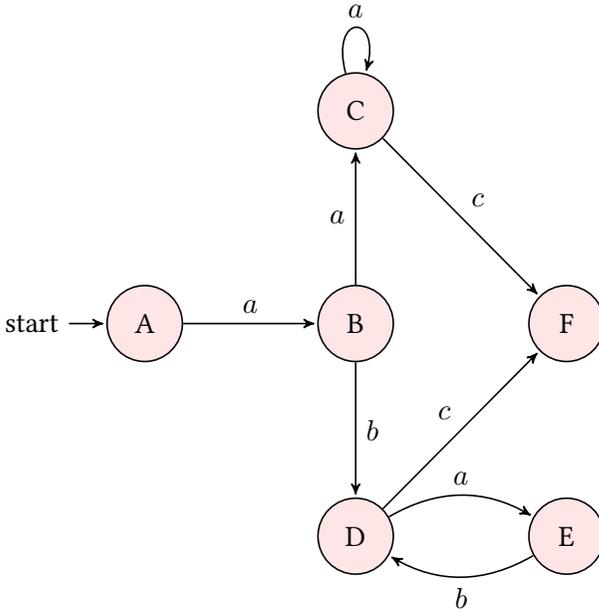
If $S = \{1, 2, 3\}$, then

$$2^S = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}.$$

Definition 60

A **nondeterministic finite automata** (aka **NFA**) consists of a set of states S , an initial state $s \in S$, an alphabet for the input string A , a special symbol ϵ that means do not change the state, a set of final states $F \subseteq S$, and a rule $r : S \times (A \cup \{\epsilon\}) \rightarrow 2^S$ that tells us given the state and the next symbol on the input string, to which set of states we move to next.

It turns out that all NFA's can be converted to DFA's, although the result might use a number of nodes that is exponential in the size of the regular expression. In the case of the NFA above, conversion is easy because after the first a , if the second character is a or b our path is determined.



Because the size of a DFA might be exponentially large in the size of the original regular expression, a seemingly short regular expression might take exponential time to evaluate by a computer! This is the basis for what are called *regular expression denial-of-service attacks*. By feeding a program or service a regular expression that unpacks to be exponentially large, a malicious user (or someone very clumsy) could bring a system to an effective halt while it unpacks the query.

So you do have to be a bit careful using regular expressions!

Using regular expressions

Summary Once we have a regular expression, there are multiple commands to detect matches, count matches, replace matches, and extract matches.

Regular expressions and strings

| | |
|------------------------------|---|
| <code>str_detect</code> | Returns true or false if match found in string. |
| <code>str_subset</code> | Returns strings from vector that have at least one match. |
| <code>str_count</code> | Counts matches in a string. |
| <code>str_extract</code> | Returns first match found in the string. |
| <code>str_extract_all</code> | Returns all the matches found in the string. |
| <code>str_locate</code> | Returns the start and ending characters of the first match. |
| <code>str_locate_all</code> | Returns the start and ending characters of all matches. |
| <code>str_match</code> | Gives the match broken into components. |
| <code>str_match_all</code> | Gives all matches in a string broken up by component. |
| <code>str_replace</code> | Replaces first match with a new string. |
| <code>str_replace_all</code> | Replaces all matches with a new string. |
| <code>seq_along</code> | Vector of numbers from 1 up to the length of the string. |

There are two commands that are used to create vectors of strings.

Vectors of strings

| | |
|----------------------|---|
| <code>apropos</code> | Searches everything in the Global Environment in R. |
| <code>dir</code> | Lists filenames in the working directory. |
| <code>glob2rx</code> | Converts a glob pattern to a regular expression. |

Now that we have regular expressions and all their wonderful finite automata power, how can we use them within R? We have seen that the function `str_view` in the package

`htmlwidgets` can be used to view matches directly, but how do we include matches in a program?

The `str_detect` function does exactly this task. If the string contains a match, then the result is true. Otherwise, it is false.

As an example, consider four Greek letters written out as English words:

```
greek <- str_sort(c("beta", "alpha", "iota", "gamma"))
greek
```

```
## [1] "alpha" "beta" "gamma" "iota"
```

To match "et" to this:

```
str_detect(greek, "et")
```

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

The output indicates that "beta" is the only output that contains exactly these two letters in this order.

Anytime you use any numerical operation on values that are TRUE or FALSE, they automatically get converted to 0 or 1. That means that when you use `sum` or `mean` on the result of `str_detect`, it will calculate the total number of matches, or the percentage number of strings that match respectively.

```
sum(str_detect(greek, "et"))
```

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

```
mean(str_detect(greek, "et"))
```

```
## [1] 0.25
```

As usual in combinatorics, it is often easier to find the negation of something than the original thing. For instance, the `words` data set is a collection of 980 common words in the English language. Suppose that we want to find the words in this set that consists entirely of consonants or y. This could be doable but challenging with a regular expression. An easier approach is to find all words that do not contain a single a, e, i, o, or u and then use `!` to negate this.

First to find words that do not contain a, e, i, o, or u:

```
# Find all words containing at least one vowel, and negate
no_vowels_1 <- !str_detect(words, "[aeiou]")
sum(no_vowels_1)
```

```
## [1] 6
```

To find words that do consist of only consonants or y, we can use the following. Recall that surrounding the regex with `^` and `$` makes it that we match the entire word. The `^` symbol inside the brackets is the negation symbol for regex: it means that we are matching anything that is not a vowel, and the `+` after the brackets means that we are taking words that consists of one or more consonants.

```
# Find all words consisting only of consonants (non-vowels)
no_vowels_2 <- str_detect(words, "[^aeiou]+$")
sum(no_vowels_2)
```

```
## [1] 6
```

We can check if the two vectors are exactly the same with the `identical` function.

```
identical(no_vowels_1, no_vowels_2)
```

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

Logical subsetting

Suppose that we want to pick out those words that do not have an a, e, i, o, or u. Then we could use logical subsetting:

```
words[!str_detect(words, "[aeiou]")]
```

```
## [1] "by" "dry" "fly" "mrs" "try" "why"
```

This is a bit clunky however, so there is a command `str_subset` to avoid this construction:

```
str_subset(words, "[^aeiou]+$")
```

```
## [1] "by" "dry" "fly" "mrs" "try" "why"
```

However, to use `str_subset`, we needed the direct version of the regex.

Strings and tibbles

Oftentimes we are not dealing with strings in isolation, but rather in a tibble. Consider the following tibble.

```
df <- tibble(
  word = words,
  i = seq_along(words)
)
df

## # A tibble: 980 x 2
##   word      i
##   <chr>   <int>
## 1 a         1
## 2 able      2
## 3 about     3
## 4 absolute  4
## 5 accept    5
## 6 account   6
## 7 achieve   7
## 8 across    8
## 9 act       9
## 10 active   10
## # ... with 970 more rows
```

The `seq_along` function is a variant of `seq` that generates a number from 1 up to the length of the argument. So in this case it is equivalent to `i = seq(1:length(words))`.

Anyway, now suppose we want to search with the strings in the tibble variable `words`. We can use `str_detect` within `filter` to make this happen.

```
df %>% filter(!str_detect(words, "[aeiou]"))

## Warning: package 'bindrcpp' was built under R version 3.5.2
## # A tibble: 6 x 2
##   word      i
##   <chr> <int>
## 1 by      123
## 2 dry     249
## 3 fly     328
## 4 mrs     538
## 5 try     895
## 6 why     952
```

Counting matches within a word

The function `str_detect` returns either TRUE or FALSE depending on whether or not the string contains the regex. Sometimes we want more information, such as how many times the string contains the regex. In this case, we can use `str_count`. Recall our set of four Greek letters.

```
greek
```

```
## [1] "alpha" "beta" "gamma" "iota"
```

To count the number of times a appears in each letter, we use:

```
str_count(greek, "a")
```

```
## [1] 2 1 2 1
```

As always, we can use `mutate` to add the information obtained to a tibble.

```
df %>%
  mutate(
    aeiou = str_count(word, "[aeiou]"),
    not_aeiou = str_count(word, "[^aeiou]")
  )
```

```
## # A tibble: 980 x 4
##   word      i aeiou not_aeiou
##   <chr>    <int> <int>     <int>
## 1 a        1     1         0
## 2 able     2     2         2
## 3 about    3     3         2
## 4 absolute 4     4         4
## 5 accept   5     2         4
## 6 account  6     3         4
## 7 achieve  7     4         3
## 8 across   8     2         4
## 9 act      9     1         2
## 10 active  10    3         3
## # ... with 970 more rows
```

A thing to note about the count is that matches never overlap. This goes back to our idea of a regular expression as being equivalent to a finite automata that never looks at previous input. Once we have a match, everything resets to the beginning, and we are starting over from scratch. For instance if we match "aba" to string ababababab, we get

ababababa

For instance:

```
str_count("ababababa", "aba")
```

```
## [1] 2
```

17.1 Extracting matches

The next step is to actually extract the matches when found. This can be illustrated with the Harvard sentences data set (https://en.wikipedia.org/wiki/Harvard_sentences) which is a group of sentences intended to match the frequency of phenomes in English. It is contained in the `stringr` in the variable `sentences`.

```
length(sentences)
```

```
## [1] 720
```

```
head(sentences)
```

```
## [1] "The birch canoe slid on the smooth planks."
## [2] "Glue the sheet to the dark blue background."
## [3] "It's easy to tell the depth of a well."
## [4] "These days a chicken leg is a rare dish."
## [5] "Rice is often served in round bowls."
## [6] "The juice of lemons makes fine punch."
```

Let's take a list of colors, and see which sentences contain at least one of these words. A quick way of creating an `or` regex is to use the `collapse` parameter within `str_c`. This takes a vector and strings and collapses it down to a single string, with the given separator between the string. By using the separator character `|`, we immediately get a useful regex. For our colors:

```
colors <- c("red", "orange", "yellow", "green", "blue", "purple")
color_match <- str_c(colors, collapse = "|")
color_match
```

```
## [1] "red|orange|yellow|green|blue|purple"
```

The `str_detect` can figure out which sentences contain colors, and then the `str_extract` actually tells us which color it was.

```
has_color <- str_subset(sentences, color_match)
matches <- str_extract(has_color, color_match)
head(matches)
```

```
## [1] "blue" "blue" "red" "red" "red" "blue"
```

Note that `str_extract` only returns the first match.

```
str_extract("The blue marker and the red marker", color_match)
```

```
## [1] "blue"
```

If we do want all of the matches, we can use `str_extract_all`. It returns the matches in the form of a *list*, a data structure in R that we have not talked about yet. It is similar to an n -tuple in that a list can contain items of different variable types. In the example below, the list contains two items which are vectors of different lengths.

```
x <- c("The blue marker and the red marker", "green acres")
str_extract_all(x, color_match)
```

```
## [[1]]
## [1] "blue" "red"
##
## [[2]]
## [1] "green"
```

The *matrix* variable type in R can be more intuitive. If we set the parameter `simplify` to `TRUE`, then the result will be placed into a matrix instead of a list.

```
str_extract_all(x, color_match,
                simplify = TRUE)
```

```
##      [,1]      [,2]
## [1,] "blue"   "red"
## [2,] "green"  ""
```

Sometimes we do not want the match extracted, but we want to know *where* in the string the match occurred. The `str_locate` and `str_locate_all` commands accomplish this.

```
str_locate(x, color_match)
```

```
##      start end
## [1,]      5  8
## [2,]      1  5
```

```
str_locate_all(x, color_match)
```

```
## [[1]]
##      start end
## [1,]      5  8
## [2,]     25  27
##
## [[2]]
##      start end
## [1,]      1  5
```

17.2 Keeping our matches

Alternatives can be used to determine entire words as well. Suppose that we want to try to find the nouns in a sentence. Separating words is actually a fairly difficult task (for instance, “a lot” is actually one word), but a simple heuristic is to treat everything separated by a space as a different word.

Finding nouns is even more difficult than finding words, again a simple heuristic is to look for words that follow a ‘, an’ or “the”.

First, the regular expression:

```
noun <- "(a|an|the) ([^ ]+)"
```

Translated this means: look for a, an, or the, followed by a space, followed by a one or more characters that are *not* a space. So stop at the next space.

Now let’s try this on sentences

```
has_noun <- sentences %>%
  str_subset(noun) %>%
  head(10)
has_noun %>%
  str_extract(noun)
```

```
## [1] "the smooth" "the sheet" "the depth" "a chicken" "the
## [6] "the sun" "the huge" "the ball" "the woman" "a l
```

Note that `str_extract` gives us the complete match, while `str_match` will give us each component of the match in a matrix form.

```
has_noun %>%
  str_match(noun)
```

```
##           [,1]           [,2]  [,3]
## [1,] "the smooth" "the" "smooth"
## [2,] "the sheet"  "the" "sheet"
## [3,] "the depth"  "the" "depth"
## [4,] "a chicken"  "a"   "chicken"
## [5,] "the parked" "the" "parked"
## [6,] "the sun"    "the" "sun"
## [7,] "the huge"   "the" "huge"
## [8,] "the ball"   "the" "ball"
## [9,] "the woman"  "the" "woman"
## [10,] "a helps"   "a"   "helps"
```

Just to emphasize, our regular expression is a poor grammarian: it picks up things like “the smooth” which is an adjective, not a noun.

The `extract` function in `tidyr` works much the same way as `str_match` together with a `mutate`. It finds the data and pulls it out into a new column in the tibble.

```
tibble(sentence = sentences) %>%
  extract(
    sentence, c("article", "noun"), "(a|the) ([^ ]+)",
    remove = FALSE
  )
```

```
## # A tibble: 720 x 3
##   sentence                                article noun
##   <chr>                                <chr> <chr>
## 1 The birch canoe slid on the smooth planks. the    smoot
## 2 Glue the sheet to the dark blue background. the    sheet
## 3 It's easy to tell the depth of a well.    the    depth
## 4 These days a chicken leg is a rare dish.  a      chick
## 5 Rice is often served in round bowls.      <NA>   <NA>
## 6 The juice of lemons makes fine punch.     <NA>   <NA>
## 7 The box was thrown beside the parked truck. the    parke
## 8 The hogs were fed chopped corn and garbage. <NA>   <NA>
## 9 Four hours of steady work faced us.       <NA>   <NA>
## 10 Large size in stockings is hard to sell.  <NA>   <NA>
## # ... with 710 more rows
```

As with `str_extract`, there is a form `str_match_all` that pulls out *all* of the matches for a given string.

Replacing matches

The commands `str_replace` and `str_replace_all` find one (or all) matches, and then replace them with the second argument to the command. For example:

```
x <- c("Apple", "Microsoft", "Google")
str_replace(x, "[AEIOUaeiou]", "-")

## [1] "-pple"      "M-crosoft" "G-ogle"
```

```
str_replace_all(x, "[AEIOUaeiou]", "-")

## [1] "-ppl-"      "M-cr-s-ft" "G--gl-"
```

We can use backreferences as part of the replacement. The following swaps the location of the first and second word (as indicated by space.) Recall the regex “`([^\]+)`” picks out non-space characters until it finds a space.

```
sentences %>%
  str_replace("([^\ ]+) ([^\ ]+)", "\\2 \\1") %>%
  head(5)
```

```
## [1] "birch The canoe slid on the smooth planks."
## [2] "the Glue sheet to the dark blue background."
## [3] "easy It's to tell the depth of a well."
## [4] "days These a chicken leg is a rare dish."
## [5] "is Rice often served in round bowls."
```

17.3 Creating vectors of strings

There are a couple commands in R that create vectors of strings. These of course can then be used with any of the commands we’ve learned to search out the strings that are important.

The first such command is `apropos` which searches all of the variables in the Global Environment in R. This can be used to find that function that you know contains a word, but you cannot quite remember what it is.

```
apropos("extract")
```

```
## [1] "extract"      "extract_"      "extract_numeric" "e
## [5] "model.extract" "str_extract"   "str_extract_all"
```

The second such command is `dir`, which creates a vector of strings where each string is the filename of a file in the working directory. The `pattern` parameter takes a regular expression and only returns those filenames that match. For example:

```
head(dir(pattern = "\\*.Rmd$"))
```

This returns any files in the directory that contain “.Rmd” anywhere within the filename.

17.4 When *stringr* is not enough

The package `stringr` contains a couple dozen of the most commonly used functions for dealing with strings, but sometimes more flexibility is needed. At that point, you should turn to `stringi`, which contains several hundred functions related to string operations.

The primary difference in calling functions from `stringi` is that the functions all begin with `stri_` instead of `str_`.

Functions that create patterns

Summary There are other ways of using strings to match patterns.

Pattern matching

| | |
|------------------------|--|
| <code>str_split</code> | Splits a string based upon a separator. |
| <code>regex</code> | Treats a string as a regular expression. |
| <code>glob2rx</code> | Converts a glob pattern for a filename into a regular expression. |
| <code>fixed</code> | Searches for a fixed set of bytes. |

Not all tasks involving strings need the power of regular expressions to accomplish.

18.1 Splitting

Suppose that we want to take a string that uses a separator such as the space or | characters, and break it into its component parts. Then we can use `str_split` to accomplish this. For example:

```
sentences %>%
  head(3) %>%
  str_split(" ")

## [[1]]
## [1] "The"      "birch"    "canoe"    "slid"     "on"
## [6] "the"      "smooth"   "planks."
##
## [[2]]
## [1] "Glue"      "the"      "sheet"    "to"
## [5] "the"      "dark"     "blue"     "background."
##
```

```
## [[3]]
## [1] "It's" "easy" "to" "tell" "the" "depth" "of"
## [8] "a" "well."
```

The result is a list: a collection of varying data types. For instance the first element of the list, denoted `[[1]]` is a vector of 8 strings, while `[[5]]` is a vector of 7 strings.

The `str_` functions that return a list have a parameter `simplify` that can be set to `TRUE` in order to make the result into a matrix.

```
sentences %>%
  head(3) %>%
  str_split(" ", simplify = TRUE)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,] "The" "birch" "canoe" "slid" "on" "the" "smooth"
## [2,] "Glue" "the" "sheet" "to" "the" "dark" "blue"
## [3,] "It's" "easy" "to" "tell" "the" "depth" "of"
##      [,8] [,9]
## [1,] "planks." ""
## [2,] "background." ""
## [3,] "a" "well."
```

The shorter lines get filled with empty strings to make every line of the matrix of equal length. You can also set the maximum number of pieces.

```
fields <- c("Name: Huber: Mark", "Country: US: CA", "Age: 47")
fields %>% str_split(":", n = 2, simplify = TRUE)
```

```
##      [,1] [,2]
## [1,] "Name" "Huber: Mark"
## [2,] "Country" "US: CA"
## [3,] "Age" "47"
```

Note that after the first split since there are a max of two pieces, the remaining gets put all in the second piece regardless of the presence or absence of another “:”.

The `boundary` helper function can also be used to divide things. Consider

```
x <- "This is a lot of monkeys. I find it a bit strange."
```

`str_split` will (without `boundary`) will include the period at the end of the first sentence and the two spaces between monkeys. ' ' and I”.

```
str_split(x, " ") [[1]]
```

```
## [1] "This"      "is"          "a"           "lot"         "of"
## [6] "monkeys."   ""            "I"           "find"        "it"
## [11] "a"          "bit"         "strange."
```

Splitting by words gives:

```
str_split(x, boundary("word")) [[1]]
```

```
## [1] "This"      "is"          "a"           "lot"         "of"
## [6] "monkeys"   "I"           "find"        "it"          "a"
## [11] "bit"       "strange"
```

Although note that it does not recognize a lot ' ' ora bit" as single words.

18.2 Transforming other pattern types to regular expressions

Hidden behind the scenes (strings?) is a command **regex**. A string given as a regular expression is automatically turned into one using the **regex** function. So

```
sum(str_detect(fruit, "berry"))
```

```
## [1] 14
```

is really the same as

```
sum(str_detect(fruit, regex("berry")))
```

```
## [1] 14
```

By explicitly putting in the **regex** function, you can modify how it transforms the string into a regular expression.

- Setting **ignore_case** to **TRUE** means that the string will match either upper or lower case forms.

```
str_replace(c("Apple", "Banana"), regex("a", ignore_case = TRUE
```

```
## [1] "-pple" "B-nana"
```

- For multiline strings, setting **multiline** to **TRUE** will allow **^** and **\$** to match the beginning and end of each line rather than the entire string.

```
x <- "Test 1\nTest 2\nTest 3"
str_extract_all(x, "^Test")[[1]]
```

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

versus

```
str_extract_all(x, regex("^Test", multiline = TRUE))[[1]]
```

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

- Regular expressions are terrible when they get long. The `comments` when set to `TRUE` allow you to make comments. Spaces are ignored, as is everything after the `#` symbol. To make a space actually part of things, it must be escaped with `"`.

```
phone <- regex("
  \\(?:      # optional opening parens
  (\\d{3})  # area code
  [ ] -]?   # optional closing parens, space, or dash
  (\\d{3})  # another three numbers
  [ -]?    # optional space or dash
  (\\d{3})  # three more numbers
", comments = TRUE)
```

```
str_match("514-791-8141", phone)
```

```
##      [,1]      [,2]  [,3]  [,4]
## [1,] "514-791-814" "514" "791" "814"
```

- Setting `dotall` to `TRUE` makes `.` match everything, included the newline character `\n`.

Globs

We saw that all the filenames in the working directory could be brought in using the `dir` command. In fact, filenames have their own pattern matching methods that are very different from regular expressions. These are called *globs*.

Definition 61

A **glob** is a pattern that specifies filename strings. In particular, `*` is often the wildcard character in a glob.

So for instance, `*.Rmd` is a glob that matches all filenames that end in `.Rmd`

If you want to use a glob pattern in `pattern`, the function `glob2rx` converts a glob pattern to a regular expression.

```
head(dir(pattern = glob2rx("*.Rmd")))
```

18.3 Fixed

Another way to match patterns is to use `fixed`. This function looks for a pattern that is a given expression of bytes as a string. For instance

```
fruit %>% head(10) %>% str_replace(fixed("a"), "-")
```

```
## [1] "-pple"          "-pricot"         "-vocado"
## [4] "b-nana"         "bell pepper"    "bilberry"
## [7] "bl-ckberry"     "bl-ckcurrant"   "blood or-nge"
## [10] "blueberry"
```

Why use `fixed` rather than `regex`? In a word: speed. By only having to deal with the simplest type of regular expression, `fixed can be significantly faster than regex`.

Factors

Summary Strings that are used to encode the values measured for a categorical variable are called **levels**, and there are several functions designed to assist in modifying the order and name of levels. Some of these are in base R, while others are in the tidyverse package **forcats**

Factor and level commands

| | |
|---------------------|--|
| factor | Give factor values and permissible levels. |
| unique | Forms levels from unique data values. |
| fct_reorder | Reorder the levels of a factor by one variable. |
| fct_reorder2 | Reorder the levels of a factor by two variables. |
| fct_relevel | Push level to end of the order. |
| fct_recode | Rename levels |
| fct_collapse | Combine levels. |
| fct_lump | Combine all uncommon levels. |

19.1 Factors

Recall that in tidy data, each column corresponds to a variable that is also known as a **factor**. A **factor** is something that can be measured, and the values that each measure can take on are called **levels**. Finally, **categorical** data only takes on a finite set of values. The month, blood type, and religion are typical examples of categorical variables.

Definition 62

In tidy data, a variable can also be called a **factor**

Definition 63

The values that a factor can take on are called **levels**.

Suppose that we record our measurements using a string.

```
x1 <- c("Dec", "Dec", "Apr", "Jan")
```

There are couple problems with this. First it's easy to make a typo that move us outside of the set of months.

```
x2 <- c("Duc", "Dec", "Apr", "Jam")
```

Second, if we sort the values, they don't sort the way we want as months, instead they sort as strings.

```
sort(x2)
```

```
## [1] "Apr" "Dec" "Duc" "Jam"
```

To fix these problems, we can state explicitly what the possible levels are, and how to sort them.

```
months <- c("Jan", "Apr", "Dec")
```

Now we explicitly tell R that these are levels of a factor using the **factor** function.

```
y1 <- factor(x1, levels = months)
y1
```

```
## [1] Dec Dec Apr Jan
## Levels: Jan Apr Dec
```

Now sorting works

```
sort(y1)
```

```
## [1] Jan Apr Dec Dec
## Levels: Jan Apr Dec
```

and if we try to put in something wrong we get an NA value.

```
y2 <- factor(x2, levels = months)
y2
```

```
## [1] <NA> Dec Apr <NA>
## Levels: Jan Apr Dec
```

For certain data sets, it is helpful to have the levels sorted by the order of their first appearance in the data set. This can be accomplished by passing the output of the function `unique` to the `levels` parameter.

```
f1 <- factor(x1, levels = unique(x1))
f1
```

```
## [1] Dec Dec Apr Jan
## Levels: Dec Apr Jan
```

19.2 Package *forcats*

The tidyverse package for dealing with factors and levels is `forcats`, which is an acronym for *categorical data sets*. An acronym is an abbreviation formed from a subset of letters in a phrase that is pronounced as a word. Often acronyms are formed from the initial letters of a phrase. For instance, NORC means the National Opinion Research Centers and is based at the University of Chicago. The other type of acronym that is not formed solely from the initial letters in the phrase is less common, but still used. Examples include NORAD and Ioran.

Speaking of NORC, they conduct something called the General Social Survey (<http://gss.norc.org/>) which has for many years has asked the US population about marriage, age, race, income, religion, and other factors.

The `forcats` packages contains a variable `gss_cat` that is a small sample of the data set from the year 2000.

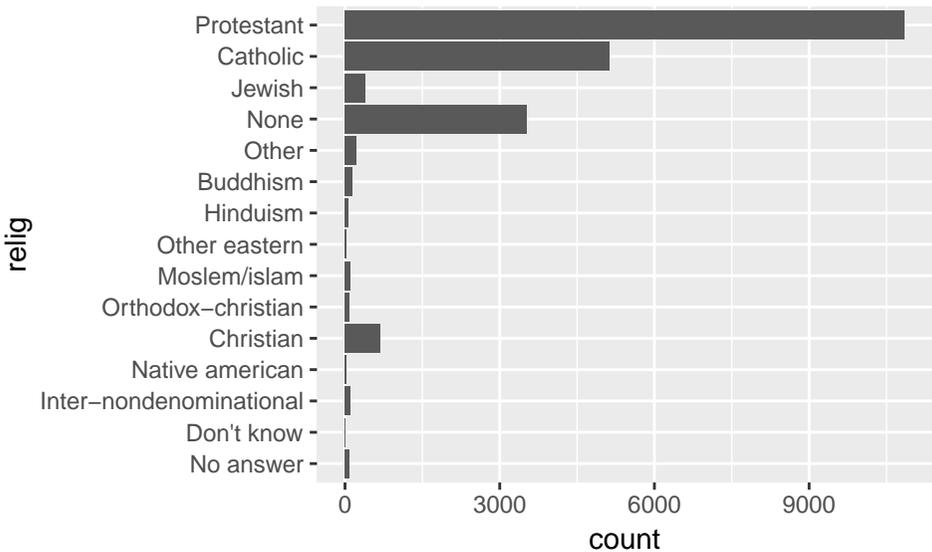
```
gss_cat
```

```
## # A tibble: 21,483 x 9
##   year marital    age race  rincome partyid  relig
##   <int> <fct>    <int> <fct> <fct>    <fct>    <fct>
## 1 2000 Never ma~ 26 White $8000 to~ Ind,near ~ Protes~
## 2 2000 Divorced 48 White $8000 to~ Not str r~ Protes~
## 3 2000 Widowed 67 White Not appl~ Independe~ Protes~
## 4 2000 Never ma~ 39 White Not appl~ Ind,near ~ Orthod~
## 5 2000 Divorced 25 White Not appl~ Not str d~ None
## 6 2000 Married 25 White $20000 ~ Strong de~ Protes~
## 7 2000 Never ma~ 36 White $25000 o~ Not str r~ Christ~
## 8 2000 Divorced 44 White $7000 to~ Ind,near ~ Protes~
## 9 2000 Married 44 White $25000 o~ Not str d~ Protes~
## 10 2000 Married 47 White $25000 o~ Strong re~ Protes~
## # ... with 21,473 more rows
```

This is a tibble, so we can use our panoply of commands to learn more about it.

```
gss_cat %>%
```

```
  ggplot(aes(relig)) +
  geom_bar() +
  coord_flip()
```



Ev-

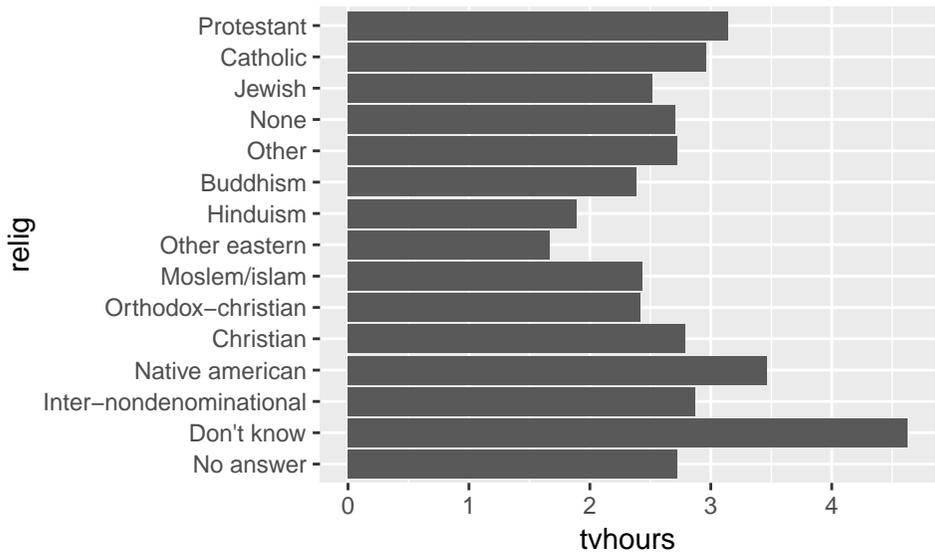
everything on the left hand side is a level for the factor `relig`. Suppose we want to learn about the average age and average number of hours spent watching TV per day across religions.

```
relig_sum <- gss_cat %>%
```

```
  group_by(relig) %>%
  summarize(
    age = mean(age, na.rm = TRUE),
    tvhours = mean(tvhours, na.rm = TRUE),
    n = n()
  )
```

```
relig_sum %>%
```

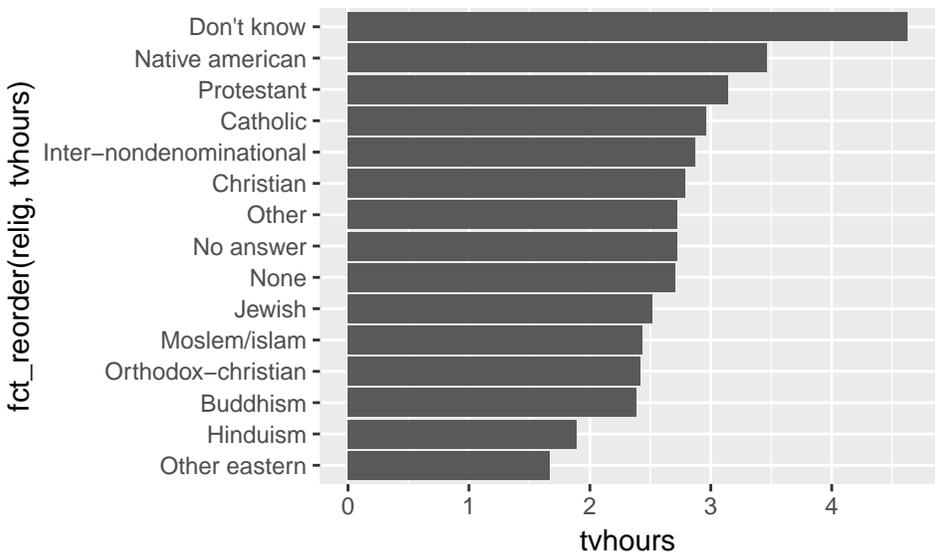
```
  ggplot() +
  geom_bar(aes(relig, tvhours), stat = 'identity') +
  coord_flip()
```



19.3 Ordering the levels

In this case the order of the levels is not exactly helpful. So we want to reorder the factor levels based on the TV hours viewed. The function `fct_reorder` accomplishes exactly this task. For instance:

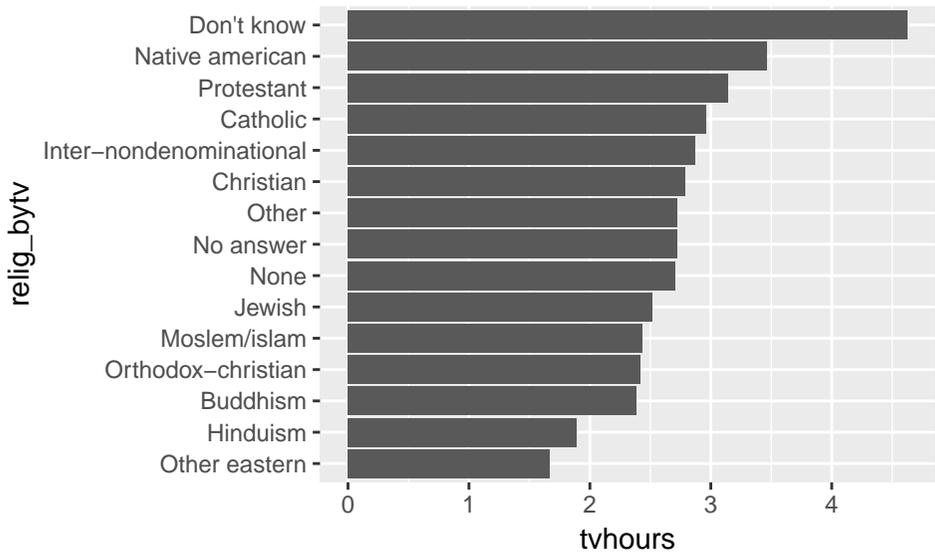
```
relig_sum %>%
  ggplot() +
  geom_bar(aes(fct_reorder(relig, tvhours), tvhours), stat = 'identity') +
  coord_flip()
```



It is much easier with this level ordering to see how much TV viewing hours in 2000 changed with religion.

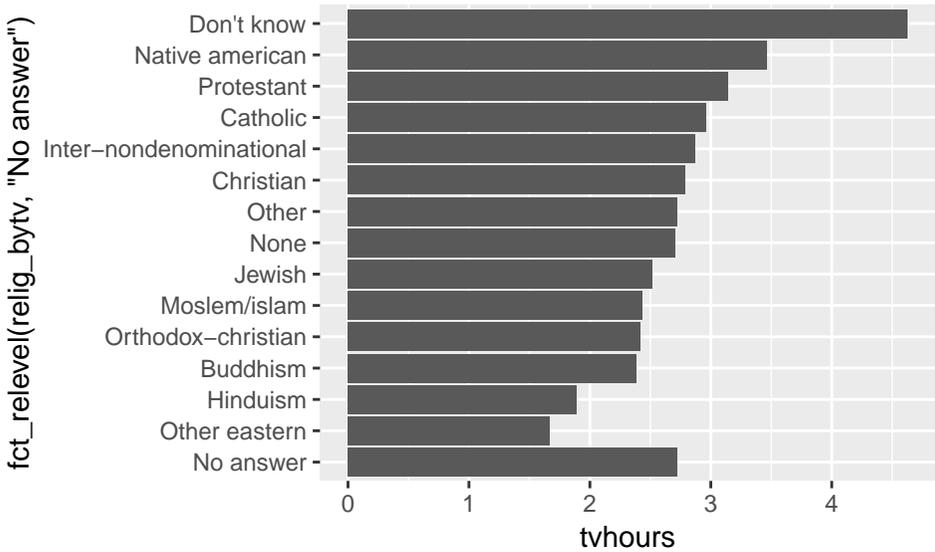
The same reordering could have been accomplished by directly mutating the `relig` factor as well.

```
relig_sum %>%
  mutate(relig_bytv = fct_reorder(relig, tvhours)) %>%
  ggplot() +
    geom_bar(aes(relig_bytv, tvhours), stat = 'identity') +
    coord_flip()
```



Note that a couple of these answers for religion are not like the others. For instance, we have "Don't know", "Other", "None", and "No answer". These of course are not religions in and of themselves. We can move a level to the front of the line using the `fct_relevel` command.

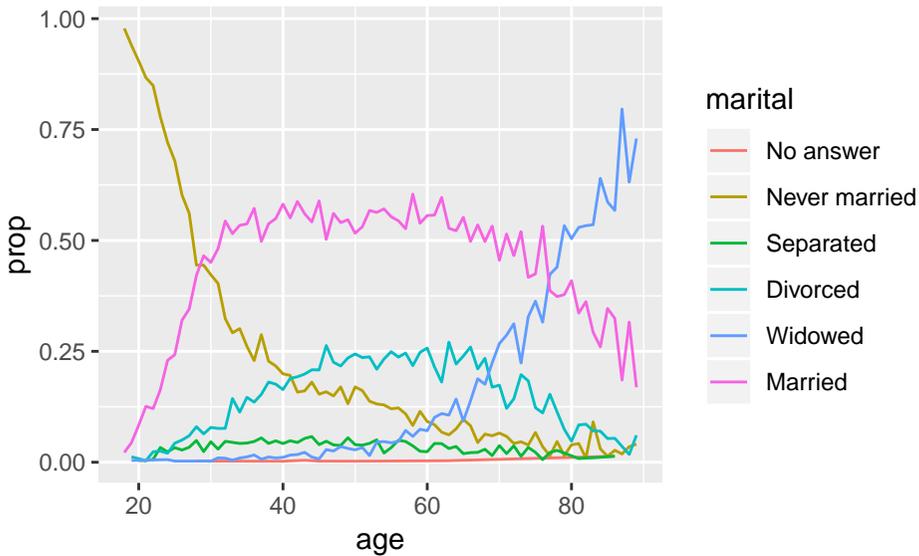
```
relig_sum %>%
  mutate(relig_bytv = fct_reorder(relig, tvhours)) %>%
  ggplot() +
    geom_bar(aes(fct_relevel(relig_bytv, "No answer"), tvhours))
    coord_flip()
```



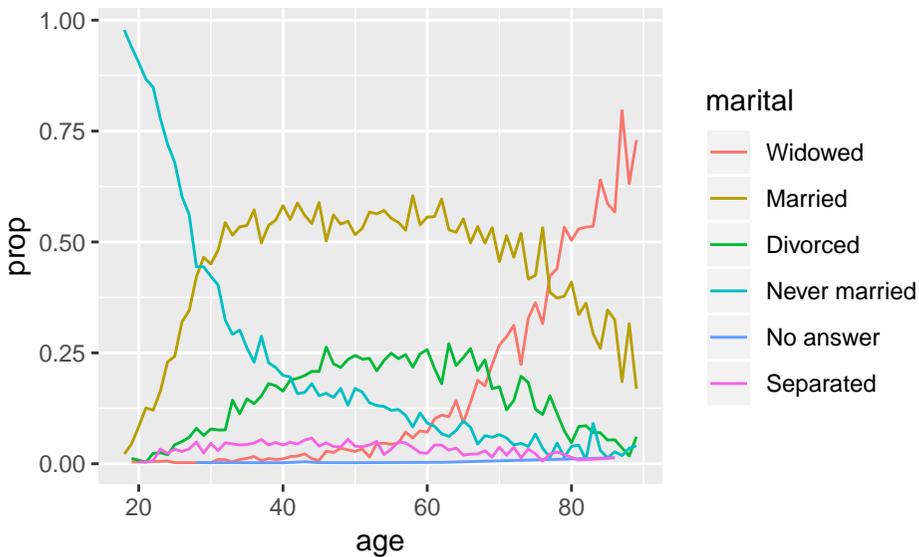
The function `fct_reorder2` can be useful when we are doing line plots in color. This function lines up the lines so that they are ordered by the last value. This makes the lines match up correctly with the labels in the legend, which makes the legend much easier to read.

```
by_age <- gss_cat %>%
  filter(!is.na(age)) %>%
  count(age, marital) %>%
  group_by(age) %>%
  mutate(prop = n / sum(n))

ggplot(by_age, aes(age, prop, color = marital)) +
  geom_line(na.rm = TRUE)
```



```
ggplot(by_age, aes(age, prop, color = fct_reorder2(marital, age)
  geom_line() +
  labs(color = "marital")
```



19.4 Changing the levels

It might be the case that we wish to change the actual names of the levels for clarity or for a particular graphic. The `fct_recode` accomplishes this task. Consider:

```
gss_cat %>% count (partyid)

## # A tibble: 10 x 2
##   partyid          n
##   <fct>          <int>
## 1 No answer        154
## 2 Don't know         1
## 3 Other party      393
## 4 Strong republican 2314
## 5 Not str republican 3032
## 6 Ind,near rep     1791
## 7 Independent      4119
## 8 Ind,near dem     2499
## 9 Not str democrat 3690
## 10 Strong democrat 3490
```

There are three parties hiding in there: Republican, Independent, and Democratic. However, the adjectives come before Republican and Democrat, and after Independent. Moreover, each should be capitalized. We can fix these with a recode.

```
gss_cat %>%
  mutate (partyid = fct_recode (partyid,
    "Republican, strong" = "Strong republican",
    "Republican, weak" = "Not str republican",
    "Independent, near rep" = "Ind,near rep",
    "Independent, near dem" = "Ind,near dem",
    "Democrat, weak" = "Not str democrat",
    "Democrat, strong" = "Strong democrat"
  )) %>%
  count (partyid)
```

```
## # A tibble: 10 x 2
##   partyid          n
##   <fct>          <int>
## 1 No answer        154
## 2 Don't know         1
## 3 Other party      393
## 4 Republican, strong 2314
## 5 Republican, weak  3032
## 6 Independent, near rep 1791
## 7 Independent      4119
## 8 Independent, near dem 2499
```

```
## 9 Democrat, weak          3690
## 10 Democrat, strong       3490
```

Because we did not mention some of the levels (for instance, “No answer”), that level stayed exactly the same as before. We can also use `fct_recode` to combine several labels into 1.

```
gss_cat %>%
  mutate(partyid = fct_recode(partyid,
    "Republican, strong" = "Strong republican",
    "Republican, weak" = "Not str republican",
    "Independent, near rep" = "Ind,near rep",
    "Independent, near dem" = "Ind,near dem",
    "Democrat, weak" = "Not str democrat",
    "Democrat, strong" = "Strong democrat",
    "Other" = "No answer",
    "Other" = "Don't know",
    "Other" = "Other party"
  )) %>%
  count(partyid)
```

```
## # A tibble: 8 x 2
##   partyid          n
##   <fct>          <int>
## 1 Other          548
## 2 Republican, strong 2314
## 3 Republican, weak  3032
## 4 Independent, near rep 1791
## 5 Independent      4119
## 6 Independent, near dem 2499
## 7 Democrat, weak    3690
## 8 Democrat, strong  3490
```

If you wish to collapse multiple levels, an easier to read function to do so is `fct_collapse`. Here we can give each new level a vector of old levels to collapse to.

```
gss_cat %>%
  mutate(partyid = fct_collapse(partyid,
    other = c("No answer", "Don't know", "Other party"),
    rep = c("Strong republican", "Not str republican"),
    ind = c("Ind,near rep", "Independent", "Ind,near dem"),
    dem = c("Not str democrat", "Strong democrat")
```

```
)) %>%  
count(partiid)
```

```
## # A tibble: 4 x 2  
##   partyid     n  
##   <fct>   <int>  
## 1 other     548  
## 2 rep     5346  
## 3 ind     8409  
## 4 dem     7180
```

If you don't want to deal anything but the labels with the largest counts, the `fct_lump` command does this.

```
gss_cat %>%  
mutate(relig = fct_lump(relig)) %>%  
count(relig)
```

```
## # A tibble: 2 x 2  
##   relig           n  
##   <fct>       <int>  
## 1 Protestant 10846  
## 2 Other      10637
```

The most important parameter here is `n`, which says how many groups we wish to end up with.

```
gss_cat %>%  
mutate(relig = fct_lump(relig, n = 10)) %>%  
count(relig, sort = TRUE) %>%  
print(n = Inf) # show all rows of the tibble.
```

```
## # A tibble: 10 x 2  
##   relig           n  
##   <fct>       <int>  
## 1 Protestant 10846  
## 2 Catholic   5124  
## 3 None       3523  
## 4 Christian   689  
## 5 Other       458  
## 6 Jewish      388  
## 7 Buddhism    147
```

| | | | |
|----|----|-------------------------|-----|
| ## | 8 | Inter-nondenominational | 109 |
| ## | 9 | Moslem/islam | 104 |
| ## | 10 | Orthodox-christian | 95 |

Introduction to Structured Query Language (SQL)

Summary SQL, Structured Query Language, is a way to draw out data from a centrally maintained database. It is designed to be written to make command clear while providing much of the same power for selecting and transforming data seen earlier in the tidyverse. Because the tidyverse was written with SQL in mind, many (but not all) of the functions have similar names.

SQL and tidyverse commands

| | | |
|--------------------|--------------------|---|
| SELECT | select | Select a subset of variables/factors. |
| WHERE | filter | Choose observations meeting criteria. |
| ORDER BY | arrange | Order observations by a factor. |
| NULL | NA | Data that is missing or not available. |
| IS NULL | is.na | True if an variable value is NULL. |
| IS NOT NULL | !is.na | True if an variable value is not NULL. |
| & | & | Logical and. |
| OR | | Logical or. |
| NOT | ! | Logical not. |
| AS | mutate | Create new variables from old ones. |
| LIKE | str_extract | Pick out observations involving strings. |
| LIMIT | | Only return the first few values found. |
| OFFSET | | Return few values skipping some as beginning. |

In 1970, Edgar Frank Todd proposed that the data contained in a database should be represented in the form of relations. Following Todd's idea, two researchers at IBM, Raymond Boyce and Donald Chamberlin

developed the language SEQUEL to work with data stored in a relational database at IBM called System R.

Apparently trademark issues intervened and so the name was shortened to SQL, which stands for Structured Query Language.

Of course, this raised an interesting question: should the word SQL be pronounced as *sequel* or as an initialism, that is, *ess-que-ell*. Lots of folks have weighed in on this matter, including Chamberlin himself who still pronounces it *sequel*, and the ISO where it is pronounced *ess-que-ell*.

Today SQL is an ANSI/ISO standard, but there are still several competing versions of the language. Always be sure to download a reference to the version of the dialect of the language you are expected to use, or you could end up with some nasty surprises!

- SQL is often used by websites to access information from a database, making it possible to quickly change the website without modifying the underlying code, merely the data that drives it.
- A version which is quite popular is **MySQL**, which is distributed by Oracle. It has an open source version which allows it to be downloaded and used for free.
- The version we will be using here is SQLite. This is also useable with R Markdown. Instead of putting `{r}` in your code chunks to run R, we use `{sql, connection = db}`, where `db` is the database we are accessing with our query.

We can set up an SQL database in R using the **dbConnect** function in the **DBI** package. This uses a helper function **SQLite** that is part of package **RSQLite**

Much of what we can do with SQL we have already seen how to do in the tidyverse. The format has changed, but the basic tasks remain the same.

20.1 Making a connection

The online platform `data.world` is a social media network for sharing data sets and their analyses. Its name is its URL, that is, you can access it by going to data.world and setting up a free account. To illustrate our commands, we will be using a data set on outcomes from an Austin Animal Center from 2013 to 2017. This data can be found at <https://data.world/cityofaustin/9t4d-g238>.

The idea of using SQL is that the process of maintaining the data should be separate from the process of analyzing the data. That way experts can deal with the problem of storing millions, billions, or trillions of n -tuples (observations), while anyone can quickly draw out the data they need for their analysis.

Much of this chapter follows the SQL tutorial from `data.world` that can be found at <https://docs.data.world/documentation/sql>

The first thing we need to do is to make a connection to our data set.

```
library(DBI)
db = dbConnect(RSQLite::SQLite(), dbname = "data_output/animals")
```

This is a bit different than reading the database into memory, which is what something like `read_csv` does. Instead, `dbConnect` leaves the data where it is, but opens up a pathway to read the data as needed.

Inside the `dbplyr` package are commands for reading the database.

```
library(dbplyr)
```

```
## Warning: package 'dbplyr' was built under R version 3.5.2
```

```
library(dplyr)
```

```
## Warning: package 'dplyr' was built under R version 3.5.2
```

```
##
```

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

```
## The following objects are masked from 'package:dbplyr':
```

```
##
```

```
## ident, sql
```

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

```
##
```

```
## filter, lag
```

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

```
##
```

```
## intersect, setdiff, setequal, union
```

```
src_dbi(db)
```

```
## src:  sqlite 3.22.0 [D:\Dropbox\Work\2019\courses\CSCI036Spr
```

```
## tbls:  austin_animal_center_age_at_outcome, austin_animal_cen
```

```
## austin_animal_center_intakes_by_month, austin_animal_cente
```

```
## austin_animal_center_outcomes_for_animal_type_and_subtype,
```

```
## sqlite_stat4
```

We can see that this database contains four tables,

```
austin_animal_center_intakes
```

```
austin_animal_center_intakes_by_month
```

```
sqlite_state
```

```
sqlite_stat4
```

20.2 SELECT

Earlier we used `select` in the tidyverse to choose particular variable from a tibble. In SQL, the `SELECT` command does exactly the same thing. If we wish to work with all the variables, we use the glob wildcard character `*`.

Consider a data set of animals taken to an animal center in Austin, Texas. For R we usually call our commands functions, for SQL we usually call them a **query**.

For instance, consider the following query.

```
SELECT name, intake_type
FROM austin_animal_center_intakes
```

Table 20.1: *Displaying records 1 - 10*

| name | intake_type |
|---------|-----------------|
| Scamp | Stray |
| Scamp | Public Assist |
| Bri-Bri | Stray |
| Tyson | Public Assist |
| Jo Jo | Public Assist |
| Oso | Owner Surrender |
| Oso | Public Assist |
| Dottie | Stray |
| Manolo | Owner Surrender |
| Manolo | Owner Surrender |

This returns the two requested variables from the `austin_animal_center_intakes` data set. Note that `*` serves as a wildcard here: so `SELECT * FROM austin_animal_center_intakes` would return all variables.

Now, the `SELECT` and `FROM` commands were captialized in the previous commands. Strictly speaking, this is not necessary, as SQL is case-insensitive. That is because SQL was created in the days before it was common to allow upper and lower case within computer commands. That being said, the modern convention in SQL is to capitalize keywords like this. It turns out it helps greatly when reading the code to know the functions from the parameters.

Suppose that we want to rename one or more of the variables. Then we can use the `AS` keyword to change things in our output. The query.

```
SELECT name AS Name,
       intake_type AS Type
FROM austin_animal_center_intakes
```

Table 20.2: *Displaying records 1 - 10*

| Name | Type |
|---------|-----------------|
| Scamp | Stray |
| Scamp | Public Assist |
| Bri-Bri | Stray |
| Tyson | Public Assist |
| Jo Jo | Public Assist |
| Oso | Owner Surrender |
| Oso | Public Assist |
| Dottie | Stray |
| Manolo | Owner Surrender |
| Manolo | Owner Surrender |

This changes the name of the variables in the result to Name and Type respectively.

If you want to use more than one word for the titles, just use left-apostrophe to delineate strings.

```
SELECT name AS Name,
       intake_type AS 'Intake Type'
FROM austin_animal_center_intakes
```

Table 20.3: *Displaying records 1 - 10*

| Name | Intake Type |
|---------|-----------------|
| Scamp | Stray |
| Scamp | Public Assist |
| Bri-Bri | Stray |
| Tyson | Public Assist |
| Jo Jo | Public Assist |
| Oso | Owner Surrender |
| Oso | Public Assist |
| Dottie | Stray |
| Manolo | Owner Surrender |
| Manolo | Owner Surrender |

Now suppose that we want to collect together all the different types of animals. By adding the keyword `DISTINCT` to the `SELECT` command, we collapse all the different results with the same data into one (compare to [group_by](#). For instance,

```
SELECT DISTINCT animal_type
FROM austin_animal_center_intakes
```

Table 20.4: 5 records

| animal_type |
|-------------|
| Dog |
| Cat |
| Other |
| Bird |
| Livestock |

This indicates that only five values appear in the column `animal_type`.

Once `DISTINCT` is invoked it only returns observations with unique values, no matter how many columns you include. For instance, consider

```
SELECT DISTINCT animal_type,
sex_upon_intake,
age_upon_intake
FROM austin_animal_center_intakes
```

Table 20.5: Displaying records 1 - 10

| animal_type | sex_upon_intake | age_upon_intake |
|-------------|-----------------|-----------------|
| Dog | Neutered Male | 10 years |
| Dog | Neutered Male | 7 years |
| Cat | Intact Female | 16 years |
| Dog | Neutered Male | 11 years |
| Dog | Spayed Female | 7 years |
| Dog | Intact Male | 3 years |
| Dog | Spayed Female | 2 years |
| Dog | Neutered Male | 9 years |
| Dog | Spayed Female | 1 year |
| Other | Unknown | 3 years |

This returns 539 query results out of the original 75947 animal intakes. The very first line is

Dog Neutered Male 10 years

which means that this particular combination will not be repeated in the table.

20.3 WHERE

For R we used `filter` to pick out rows satisfying certain characteristics, for SQL we use `WHERE` to accomplish similar tasks.

For instance, suppose that we wish to list all animals in the data set that are cats. We could use

```
SELECT year,
       month,
       count,
       animal_type
FROM austin_animal_center_intakes_by_month
WHERE animal_type = "Cat"
```

Table 20.6: Displaying records 1 - 10

| year | month | count | animal_type |
|------|-------|-------|-------------|
| 2013 | 10 | 542 | Cat |
| 2013 | 11 | 436 | Cat |
| 2013 | 12 | 331 | Cat |
| 2014 | 1 | 335 | Cat |
| 2014 | 2 | 269 | Cat |
| 2014 | 3 | 353 | Cat |
| 2014 | 4 | 566 | Cat |
| 2014 | 5 | 901 | Cat |
| 2014 | 6 | 821 | Cat |
| 2014 | 7 | 881 | Cat |

Note that in SQL the logical equals operator is a single equals sign `=`, and not two equals signs as in most languages.

20.4 ORDER BY

We used `arrange` to put rows in order by a specified column. In SQL, the command is just called `ORDER BY`, and uses alphabetical order.

For instance,

```
SELECT year,
       month,
       count,
       animal_type
FROM austin_animal_center_intakes_by_month
```

```
WHERE animal_type = "Cat"
ORDER BY year, month
```

Table 20.7: Displaying records 1 - 10

| year | month | count | animal_type |
|------|-------|-------|-------------|
| 2013 | 10 | 542 | Cat |
| 2013 | 11 | 436 | Cat |
| 2013 | 12 | 331 | Cat |
| 2014 | 1 | 335 | Cat |
| 2014 | 2 | 269 | Cat |
| 2014 | 3 | 353 | Cat |
| 2014 | 4 | 566 | Cat |
| 2014 | 5 | 901 | Cat |
| 2014 | 6 | 821 | Cat |
| 2014 | 7 | 881 | Cat |

We can convert this to descending order by adding `DESC` to the command.

```
SELECT year,
       month,
       count,
       animal_type
FROM austin_animal_center_intakes_by_month
WHERE animal_type = "Cat"
ORDER BY year DESC, month DESC
```

Table 20.8: Displaying records 1 - 10

| year | month | count | animal_type |
|------|-------|-------|-------------|
| 2017 | 12 | 100 | Cat |
| 2017 | 11 | 427 | Cat |
| 2017 | 10 | 513 | Cat |
| 2017 | 9 | 656 | Cat |
| 2017 | 8 | 565 | Cat |
| 2017 | 7 | 669 | Cat |
| 2017 | 6 | 895 | Cat |
| 2017 | 5 | 914 | Cat |
| 2017 | 4 | 565 | Cat |
| 2017 | 3 | 353 | Cat |

20.5 NULL values and logical operators

The equivalent of NA in R is called NULL. By default, the ORDER BY command puts a NULL value at the end. To put these values first, simply add NULLS FIRST at the end of the ORDER BY line.

There are also logical operators in SQL, similar to those in R. The logical and is AND, logical or is OR, and logical not is NOT.

If we wished to look at data for cats and dogs where either the type was a stray or an owner surrender, we would use the following:

```
SELECT animal_type,
       intake_type,
       Intake_condition,
       age_upon_intake
FROM austin_animal_center_intakes
WHERE (animal_type = "Cat" OR animal_type = "DOG")
      AND (intake_type = "Stray" OR intake_type = "Owner Surrender")
```

Table 20.9: *Displaying records 1 - 10*

| animal_type | intake_type | intake_condition | age_upon_intake |
|-------------|-----------------|------------------|-----------------|
| Cat | Stray | Normal | 16 years |
| Cat | Stray | Normal | 1 month |
| Cat | Owner Surrender | Normal | 10 years |
| Cat | Owner Surrender | Normal | 9 months |
| Cat | Stray | Normal | 10 months |
| Cat | Owner Surrender | Sick | 15 years |
| Cat | Stray | Normal | 7 years |
| Cat | Stray | Normal | 3 years |
| Cat | Owner Surrender | Normal | 1 month |
| Cat | Owner Surrender | Normal | 1 month |

To check if a data value is null, we use the IS NULL expression. Similarly, to test if a data value is not null, we use the IS NOT NULL expression.

We could use AND to find data between two values, or we could use BETWEEN. For instance,

```
SELECT year,
       month,
       animal_type,
       COUNT
FROM austin_animal_center_intakes_by_month
```

```
WHERE count BETWEEN 900 AND 2000
ORDER BY month
```

Table 20.10: *Displaying records 1 - 10*

| year | month | animal_type | count |
|------|-------|-------------|-------|
| 2014 | 5 | Cat | 901 |
| 2014 | 5 | Dog | 966 |
| 2015 | 5 | Cat | 1009 |
| 2015 | 5 | Dog | 988 |
| 2016 | 5 | Cat | 921 |
| 2016 | 5 | Dog | 1020 |
| 2017 | 5 | Cat | 914 |
| 2015 | 6 | Cat | 1103 |
| 2015 | 6 | Dog | 1014 |
| 2014 | 7 | Dog | 926 |

This finds all data points with values between 900 and 2000 inclusive of the endpoints.

20.6 Transforming data

It often is the case that we wish to transform data when presenting it to the user. The standard arithmetic operators $+$, $-$, $*$, and $/$ behave exactly the way you would expect. For instance, if I wanted to transform the `age_in_days` variable to years, I could use

```
SELECT monthyear,
       animal_type,
       outcome_type,
       (age_in_days / 365) AS `Years Old`
FROM austin_animal_center_age_at_outcome
```

Table 20.11: *Displaying records 1 - 10*

| monthyear | animal_type | outcome_type | Years Old |
|-----------|-------------|-----------------|-----------|
| 2014-03 | Dog | Return to Owner | 6.668493 |
| 2014-12 | Dog | Return to Owner | 7.454795 |
| 2015-11 | Cat | Return to Owner | 16.252055 |
| 2015-03 | Dog | Return to Owner | 11.972603 |
| 2015-04 | Dog | Return to Owner | 7.638356 |
| 2014-09 | Dog | Return to Owner | 2.668493 |
| 2014-01 | Dog | Euthanasia | 2.002740 |
| 2014-01 | Cat | Euthanasia | 15.013699 |

| monthyear | animal_type | outcome_type | Years Old |
|-----------|-------------|-----------------|-----------|
| 2014-01 | Dog | Return to Owner | 3.005479 |
| 2014-01 | Dog | Return to Owner | 2.013699 |

If I wanted to pull out all the data with age at least 8 years, and then sort by the age, I could use

```
SELECT monthyear,
       animal_type,
       outcome_type,
       (age_in_days / 365) AS 'Years Old'
FROM austin_animal_center_age_at_outcome
WHERE (age_in_days / 365) > 8
ORDER BY age_in_days
```

Table 20.12: *Displaying records 1 - 10*

| monthyear | animal_type | outcome_type | Years Old |
|-----------|-------------|-----------------|-----------|
| 2014-02 | Dog | Return to Owner | 8.002740 |
| 2015-01 | Dog | Return to Owner | 8.002740 |
| 2015-05 | Dog | Return to Owner | 8.005479 |
| 2014-01 | Dog | Transfer | 8.005479 |
| 2014-02 | Dog | Return to Owner | 8.005479 |
| 2014-02 | Cat | Euthanasia | 8.005479 |
| 2014-02 | Cat | Euthanasia | 8.005479 |
| 2014-03 | Dog | Euthanasia | 8.005479 |
| 2014-03 | Dog | Return to Owner | 8.005479 |
| 2014-03 | Dog | Euthanasia | 8.005479 |

20.7 LIKE and NOT LIKE

Suppose we want to find all the data such that the breed ends with the word “wolfhound”. To accomplish this (and similar tasks), we use the `LIKE` command.

The `LIKE` command uses two wildcards. The first wildcard is the percentage sign, `%`, and stands in for any number of characters. So for example `%test` would match `unfairtest` or `fair test`, but not `test case`.

The other wildcard is the underscore symbol `_`, and matches a single character. So `t_st` would match `test` or `tkst`, but not `tests`. You can use more than one `_` if you want more than one wildcard in your search.

For instance,

```

SELECT sex_upon_outcome,
       outcome_type,
       outcome_subtype,
       breed
FROM   austin_animal_center_outcomes
WHERE  animal_type = "Dog"
       AND breed LIKE "%wolfhound%"
ORDER BY monthyear

```

Table 20.13: Displaying records 1 - 10

| sex_upon_outcome | outcome_type | outcome_subtype | breed |
|------------------|-----------------|-----------------|-------------------------------|
| Neutered Male | Transfer | Partner | Irish Terrier/Irish Wolfhound |
| Spayed Female | Adoption | NA | Irish Wolfhound Mix |
| Neutered Male | Transfer | Partner | Irish Wolfhound/Great Pyrene |
| Neutered Male | Adoption | NA | Irish Wolfhound Mix |
| Neutered Male | Adoption | NA | Catahoula/Irish Wolfhound |
| Intact Female | Return to Owner | NA | Irish Wolfhound/Great Dane |
| Neutered Male | Transfer | Partner | Irish Wolfhound/Australian Sh |
| Neutered Male | Return to Owner | NA | Irish Wolfhound Mix |
| Intact Male | Return to Owner | NA | Irish Wolfhound Mix |
| Intact Female | Transfer | Partner | Irish Wolfhound Mix |

This query matches any name that contains “wolfhound” anywhere inside the text of the breed.

20.8 OFFSET

For a table that is very large, a query could take a very large amount of time. The `LIMIT` keyword allows you to limit the number of results obtained. So

```

SELECT DISTINCT animal_type,
               sex_upon_intake,
               age_upon_intake
FROM   austin_animal_center_intakes
LIMIT 10

```

Table 20.14: Displaying records 1 - 10

| animal_type | sex_upon_intake | age_upon_intake |
|-------------|-----------------|-----------------|
| Dog | Neutered Male | 10 years |
| Dog | Neutered Male | 7 years |

| animal_type | sex_upon_intake | age_upon_intake |
|-------------|-----------------|-----------------|
| Cat | Intact Female | 16 years |
| Dog | Neutered Male | 11 years |
| Dog | Spayed Female | 7 years |
| Dog | Intact Male | 3 years |
| Dog | Spayed Female | 2 years |
| Dog | Neutered Male | 9 years |
| Dog | Spayed Female | 1 year |
| Other | Unknown | 3 years |

only returns the first 10 rows out of 539.

What if we wanted values 11 through 20 instead of 1 through 10? We could just use `OFFSET 10` to get them. That is,

```
SELECT found_location, intake_type
FROM austin_animal_center_intakes
LIMIT 10
OFFSET 10
```

Table 20.15: *Displaying records 1 - 10*

| found_location | intake_type |
|--|-----------------|
| Austin (TX) | Owner Surrender |
| 1111 W 34Th St in Austin (TX) | Public Assist |
| 12705 Lamplight Village in Austin (TX) | Wildlife |
| 6103 Manor Rd in Austin (TX) | Stray |
| 2318 Post Oak Rd. in Travis (TX) | Stray |
| Stassney & Westgate in Austin (TX) | Stray |
| 12900 Carillon Way in Manor (TX) | Stray |
| Anderson Mill Rd And Olson Dr in Austin (TX) | Stray |
| 6720 Quinton in Austin (TX) | Stray |
| Verbank Villa Dr & Ringsby Rd in Austin (TX) | Stray |

This table is rows 11 through 20 of the data. `OFFSET 20` would give rows 21 through 30, and so on.

20.9 SQL versus the tidyverse

Our commands so far!

| SQL | tidyverse |
|--------------------|----------------|
| SELECT | select |
| WHERE | filter |
| ORDER BY | arrange |
| DESC | desc |
| NULL | NA |
| = | == |
| AND | & |
| OR | |
| NOT | ! |
| IS NULL | is.na |
| IS NOT NULL | !is.na |

```
library(DBI)
```

```
## Warning: package 'DBI' was built under R version 3.4.4
```

```
db = dbConnect(RSQLite::SQLite(), dbname = "data_output/sales.s
```

Joining tables in SQL

Summary SQL was designed to use relational databases, and so has many commands for drawing data from multiple tables.

Joining tables in SQL

| | |
|------------------|---|
| JOIN | Bring two tables together. |
| OUTER | Modifies 'JOIN' to be an outer join. |
| LEFT | Modifies 'OUTER' to be a left outer join. |
| UNION | Union of observations from tables with same variables. |
| INTERSECT | Intersection of observations from tables with same variables. |
| MINUS | Set difference of observations from tables with same variables. |

Aggregating and grouping data can also be done in SQL.

Aggregation and groups in SQL

| | |
|---------------------|---|
| SUM | Adds together the non NULL values. |
| COUNT | Counts non NULL values. |
| AVG | Averages non NULL values. |
| MIN | Minimum of non NULL values. |
| MAX | Maximum of non NULL values. |
| GROUP_CONCAT | concatenate strings. |

So far we have been working with one table (relation) at a time, but the point of having more than one table is that we should have the ability to collect data from multiple tables to get the report that we are after.

21.1 Inner Join

Recall that an Inner Join brings together those observations where a particular value of a column is equal in both tables.

In the table `sales_teams`, each value of `sales_agent` appears only once, since each agent is part of only one team. Therefore it is a *key* in this table.

```
SELECT sales_agent, manager
FROM sales_teams
```

Table 21.1: *Displaying records 1 - 10*

| sales_agent | manager |
|--------------------|------------------|
| Anna Snelling | Dustin Brinkmann |
| Cecily Lampkin | Dustin Brinkmann |
| Versie Hillebrand | Dustin Brinkmann |
| Lajuana Vencill | Dustin Brinkmann |
| Moses Frase | Dustin Brinkmann |
| Jonathan Berthelot | Melvin Marxen |
| Marty Freudenburg | Melvin Marxen |
| Gladys Colclough | Melvin Marxen |
| Niesha Huffines | Melvin Marxen |
| Darcel Schlecht | Melvin Marxen |

The table `sales_pipeline` tells us what deals a particular agent has in pipeline. In this table each agent might be working on more than one deal at a time.

```
SELECT account, sales_agent
FROM sales_pipeline
```

Table 21.2: *Displaying records 1 - 10*

| Account | Sales_Agent |
|----------|-----------------|
| Cancity | Moses Frase |
| Isdom | Darcel Schlecht |
| Cancity | Darcel Schlecht |
| Codehow | Moses Frase |
| Hatfan | Zane Levy |
| Ron-tech | Anna Snelling |
| J-Texon | Vicki Laflamme |
| Cheers | Markita Hansen |
| Zumgoity | Niesha Huffines |

| Account | Sales_Agent |
|---------|----------------|
| NA | James Ascencio |

So `sales_agent` is *not* a key in this table. Since it is a key in `sales_teams`, it provides a foreign key for that table. This simplest kind of inner join can be accomplished just by using the **WHERE** command.

```
SELECT sales_teams.manager,
        sales_pipeline.sales_agent,
        sales_pipeline.account
FROM sales_teams, sales_pipeline
WHERE (sales_pipeline.sales_agent = sales_teams.sales_agent)
AND sales_pipeline.deal_stage = "Won"
```

Table 21.3: *Displaying records 1 - 10*

| manager | Sales_Agent | Account |
|------------------|-----------------|------------|
| Dustin Brinkmann | Moses Frase | Cancity |
| Melvin Marxen | Darcel Schlecht | Isdom |
| Melvin Marxen | Darcel Schlecht | Cancity |
| Dustin Brinkmann | Moses Frase | Codehow |
| Summer Sewald | Zane Levy | Hatfan |
| Dustin Brinkmann | Anna Snelling | Ron-tech |
| Celia Rouche | Vicki Laflamme | J-Texon |
| Celia Rouche | Markita Hansen | Cheers |
| Melvin Marxen | Niesha Huffines | Zumgoity |
| Dustin Brinkmann | Anna Snelling | Bioholding |

Let's break down what happened.

- In **FROM**, we have two tables `sales_teams` and `sales_pipeline`. So we are able to draw data from both.
- In **SELECT**, we can now include variables from either `sales_teams`, `sales_pipeline`, or both. We use a `tablename.variablename` formulation for such selections.
- In **WHERE**, we indicate how the join works by declaring which of the variables from the two tables must be equal, **AND** and extra condition on data from one of the tables.

The **AS** can be used to relabel the variables for extra readability. It can also be used to relabel the tables. Note that although the **AS** in the **FROM** command relabels the tables, we can use the relabels earlier in the **SELECT** command.

```
SELECT teams.manager,
        pipeline.sales_agent AS agent,
        pipeline.account
FROM sales_teams AS teams, sales_pipeline AS pipeline
WHERE (pipeline.sales_agent = teams.sales_agent)
AND pipeline.deal_stage = "Won"
```

Table 21.4: *Displaying records 1 - 10*

| manager | agent | Account |
|------------------|-----------------|------------|
| Dustin Brinkmann | Moses Frase | Cancity |
| Melvin Marxen | Darcel Schlecht | Isdom |
| Melvin Marxen | Darcel Schlecht | Cancity |
| Dustin Brinkmann | Moses Frase | Codehow |
| Summer Sewald | Zane Levy | Hatfan |
| Dustin Brinkmann | Anna Snelling | Ron-tech |
| Celia Rouche | Vicki Laflamme | J-Texon |
| Celia Rouche | Markita Hansen | Cheers |
| Melvin Marxen | Niesha Huffines | Zumgoity |
| Dustin Brinkmann | Anna Snelling | Bioholding |

Using **WHERE** in this way to make a join works, but SQL does also have an explicit **JOIN** command so we can separate conceptually the join from the filtering condition. The **USING** keyword then explicitly tells us what variable to use in bring the tables together.

```
SELECT DISTINCT teams.manager,
        pipeline.sales_agent AS agent,
        pipeline.account
FROM sales_teams AS teams
        JOIN sales_pipeline AS pipeline USING (sales_agent)
WHERE pipeline.deal_stage = "Won"
```

Table 21.5: *Displaying records 1 - 10*

| manager | agent | Account |
|------------------|-----------------|---------|
| Dustin Brinkmann | Moses Frase | Cancity |
| Melvin Marxen | Darcel Schlecht | Isdom |

| manager | agent | Account |
|------------------|-----------------|------------|
| Melvin Marxen | Darcel Schlecht | Cancity |
| Dustin Brinkmann | Moses Frase | Codehow |
| Summer Sewald | Zane Levy | Hatfan |
| Dustin Brinkmann | Anna Snelling | Ron-tech |
| Celia Rouche | Vicki Laflamme | J-Texon |
| Celia Rouche | Markita Hansen | Cheers |
| Melvin Marxen | Niesha Huffines | Zumgoity |
| Dustin Brinkmann | Anna Snelling | Bioholding |

What if our foreign key had a different name in the second table? Then instead of using **USING**, we would join the tables with **ON** which allows us to specify names.

```
SELECT DISTINCT teams.manager,
  pl.sales_agent AS agent,
  pl.account
FROM sales_teams AS teams
JOIN sales_pipeline AS pl
ON pl.sales_agent = teams.sales_agent
WHERE pl.deal_stage = "Won"
```

Table 21.6: *Displaying records 1 - 10*

| manager | agent | Account |
|------------------|-----------------|------------|
| Dustin Brinkmann | Moses Frase | Cancity |
| Melvin Marxen | Darcel Schlecht | Isdom |
| Melvin Marxen | Darcel Schlecht | Cancity |
| Dustin Brinkmann | Moses Frase | Codehow |
| Summer Sewald | Zane Levy | Hatfan |
| Dustin Brinkmann | Anna Snelling | Ron-tech |
| Celia Rouche | Vicki Laflamme | J-Texon |
| Celia Rouche | Markita Hansen | Cheers |
| Melvin Marxen | Niesha Huffines | Zumgoity |
| Dustin Brinkmann | Anna Snelling | Bioholding |

21.2 Outer Joins

Inner joins only return observations where the value for a particular column appears in both tables. Outer joins return observations where the value for a particular column appears at least once. For **left outer joins** the value has to appear in the left table, for **right outer joins** it must appear in the right, and for **full outer joins** it could appear in

either table.

In SQL, **OUTER** is a keyword that modifies **JOIN**, which can then be further modified by **LEFT**, **RIGHT**, or **FULL**.

For instance, to do a left outer join on the tables:

```
SELECT DISTINCT sales_teams.sales_agent AS agent,
               sales_pipeline.deal_stage
FROM sales_teams
LEFT OUTER JOIN sales_pipeline
ON sales_teams.sales_agent = sales_pipeline.sales_agent
WHERE sales_pipeline.deal_stage = "In Progress"
```

Table 21.7: *Displaying records 1 - 10*

| agent | Deal_Stage |
|--------------------|-------------|
| Anna Snelling | In Progress |
| Cecily Lampkin | In Progress |
| Versie Hillebrand | In Progress |
| Lajuana Vencill | In Progress |
| Moses Frase | In Progress |
| Jonathan Berthelot | In Progress |
| Marty Freudenburg | In Progress |
| Gladys Colclough | In Progress |
| Niesha Huffines | In Progress |
| Darcel Schlecht | In Progress |

A left outer join is appropriate here since we are not interested in all of the sales agents, only those that have a deal at some stage.

The use of right outer joins is rare: of course any right outer join can be written as a left outer join simply by swapping the order of the two tables. As of 2019-03-22, right outer joins are not supported by SQLite in R Markdown, so you have to use the swap trick when working in this format.

21.3 Self Join

The need for a self join arises when the values of a key for a table are used as entries in another column. For instance, suppose that I have a variable which is the employee ID. Then this ID number might be used to also indicate a manager.

```
SELECT *
FROM employees
```

Table 21.8: Displaying records 1 - 10

| emp_id | name | mgr_id | Regional_Office | Status |
|--------|--------------------|--------|-----------------|---------|
| 10001 | Anna Snelling | 10036 | Central | Current |
| 10002 | Cecily Lampkin | 10036 | Central | Current |
| 10003 | Versie Hillebrand | 10036 | Central | Current |
| 10004 | Lajuana Vencill | 10036 | Central | Current |
| 10005 | Moses Frase | 10036 | Central | Current |
| 10006 | Jonathan Berthelot | 10037 | Central | Current |
| 10007 | Marty Freudenburg | 10037 | Central | Current |
| 10008 | Gladys Colclough | 10037 | Central | Current |
| 10009 | Niesha Huffines | 10037 | Central | Current |
| 10010 | Darcel Schlecht | 10037 | Central | Current |

A self join can be used to pull out the name of each manager of each employee.

```
SELECT emp.name AS employee,
       mgr.name AS manager
FROM employees AS emp
JOIN employees AS mgr ON emp.mgr_id = mgr.emp_id
```

Table 21.9: Displaying records 1 - 10

| employee | manager |
|--------------------|------------------|
| Anna Snelling | Dustin Brinkmann |
| Cecily Lampkin | Dustin Brinkmann |
| Versie Hillebrand | Dustin Brinkmann |
| Lajuana Vencill | Dustin Brinkmann |
| Moses Frase | Dustin Brinkmann |
| Jonathan Berthelot | Melvin Marxen |
| Marty Freudenburg | Melvin Marxen |
| Gladys Colclough | Melvin Marxen |
| Niesha Huffines | Melvin Marxen |
| Darcel Schlecht | Melvin Marxen |

21.4 Aggregations

To apply functions to variable values in the tidyverse, we used `mutate` or `summarize`. In the SQL framework, these are called *aggregations*.

| | |
|---------------------|--------------------------------------|
| SUM | Adds together the non 'NULL' values. |
| COUNT | Counts non 'NULL' values. |
| AVG | Averages non 'NULL' values. |
| MIN | Minimum of non 'NULL' values. |
| MAX | Maximum of non 'NULL' values. |
| GROUP_CONCAT | Concatenate strings. |

For example, to get the total monetary value of deals closed:

```
SELECT SUM(close_value)
FROM sales_pipeline
```

Table 21.10: 1 records

| |
|-------------------------|
| <u>SUM(close_value)</u> |
| 10005534 |

The total number of won deals:

```
SELECT COUNT(*)
FROM sales_pipeline
WHERE deal_stage = "Won"
```

Table 21.11: 1 records

| |
|-----------------|
| <u>COUNT(*)</u> |
| 4238 |

The average value of the deals won:

```
SELECT AVG(close_value)
FROM sales_pipeline
WHERE sales_pipeline.deal_stage = "Won"
```

Table 21.12: 1 records

| |
|-------------------------|
| <u>AVG(close_value)</u> |
| 2360.90939122227 |

The smallest deal won:

```
SELECT MIN(close_value)
FROM sales_pipeline
WHERE sales_pipeline.deal_stage = "Won"
```

Table 21.13: 1 records

| MIN(close_value) |
|------------------|
| 38 |

21.5 GROUP BY

The **GROUP BY** command in SQL performs the same function as **group_by** in the tidyverse: it partitions the observations by the values of a particular variable. For instance, to find the average deal size for each sales agents, we could use:

```
SELECT sales_agent,
       AVG(close_value)
FROM sales_pipeline
WHERE sales_pipeline.deal_stage = "Won"
GROUP BY sales_agent
ORDER BY AVG(close_value) DESC
```

Table 21.14: Displaying records 1 - 10

| Sales_Agent | AVG(close_value) |
|--------------------|------------------|
| Elease Gluck | 3614.938 |
| Darcel Schlecht | 3304.338 |
| Rosalina Dieter | 3269.486 |
| Daniell Hammack | 3194.991 |
| James Ascencio | 3063.207 |
| Rosie Papadopoulos | 2950.885 |
| Wilburn Farren | 2866.182 |
| Reed Clapper | 2827.974 |
| Donn Cantrell | 2821.899 |
| Corliss Cosme | 2806.907 |

Once you join data from another table, you can equally well group by the adding data. So if we wanted average deal by manager:

```

SELECT sales_teams.manager,
       AVG(sales_pipeline.close_value)
FROM   sales_teams
       JOIN sales_pipeline ON (sales_teams.sales_agent = sales_
WHERE  sales_pipeline.deal_stage = "Won"
GROUP BY sales_teams.manager

```

Table 21.15: 6 records

| manager | AVG(sales_pipeline.close_value) |
|------------------|---------------------------------|
| Cara Losch | 2354.269 |
| Celia Rouche | 2629.339 |
| Dustin Brinkmann | 1465.011 |
| Melvin Marxen | 2553.209 |
| Rocco Neubert | 2837.258 |
| Summer Sewald | 2372.886 |

For filtering observations, we used **WHERE**, but if we want to use these filtered observations within a **GROUP BY**, we need to surround the **WHERE** with a **FILTER**. For instance, to get the number of deals won that had a value greater than 1000, we could use

```

SELECT sales_agent,
       COUNT(sales_pipeline.close_value) AS total,
       COUNT(sales_pipeline.close_value)
FILTER(WHERE(sales_pipeline.close_value > 1000)) AS 'over 1000'
FROM   sales_pipeline
WHERE  sales_pipeline.deal_stage = "Won"
GROUP BY sales_pipeline.sales_agent

```

To filter observations *after* aggregation has occurred, we need the **HAVING** keyword.

```

SELECT sales_agent,
       COUNT(sales_pipeline.close_value) AS 'number won'
FROM   sales_pipeline
WHERE  sales_pipeline.deal_stage = "Won"
GROUP BY sales_pipeline.sales_agent
HAVING COUNT(sales_pipeline.close_value) > 200

```

Table 21.16: 4 records

| Sales_Agent | number won |
|-----------------|------------|
| Anna Snelling | 208 |
| Darcel Schlecht | 349 |
| Kary Hendrixson | 209 |
| Vicki Laflamme | 221 |

21.6 Set operations in SQL

In the tidyverse, the set operations **union**, **intersect** and **setdiff** find the union, intersection, and set difference respectively of observations that belong to different tables, but have the same variables. The corresponding commands in SQL are **UNION**, **INTERSECT**, and **MINUS**.

Up until now, we have been using sQL queries that only have one **SELECT** command. Each time we use the **SELECT** command it creates a table. We can then use the set operations to combine these tables.

For instance, suppose that international accounts were located in one table, and domestic in another. You could use two **SELECT** commands to put the data from both tables into the same form, then **UNION** to combine them.

```
SELECT intl_accounts.account,
       intl_accounts.office_location AS location
FROM intl_accounts
UNION
SELECT accounts.account,
       "USA" AS location
FROM accounts
```

Table 21.17: Displaying records 1 - 10

| account | location |
|------------------|-------------|
| AWOLEX | USA |
| Acme Corporation | USA |
| Betasoloin | USA |
| Betatech | Kenya |
| Betatech | USA |
| Bioholding | Philippines |
| Bioholding | USA |
| Bioplex | USA |
| Blackzim | USA |
| Bluth Company | USA |

Part III

PROGRAM CONTROL

Principles of the tidyverse

Summary

One way of viewing the tidyverse is as a collection of functions that keep common principles in mind. Such a collection is also known as an *API*.

22.1 Application programming interface

Definition 64

An **API** (application programming interface) is a collection of functions and tools that allow the creation of applications that access other base functionality.

For instance, there are API's for

- accessing the operating system,
- accessing a graphics card,
- accessing a hard drive,

and in the case of the tidyverse,

- accessing the base functions of R.

The purpose of an API is to make life easier for both the programmer who must create code, the updater who must maintain the code, and the end user who receives the output of the code.

In the case of the tidyverse, Hadley Wickham had four principles in mind when creating the packages therein.

1. It should reuse existing data structures.
2. Compose simple functions with pipes.
3. The API should embrace functional programming.

4. It should be designed with humans in mind.

Let's look at each of these principles in turn.

22.2 Reusing existing structures

Data has been collected for millennia. While a census count of France in the 17th century might only be of interest to historians, data from ten or even a hundred years ago is often still of great importance today.

Therefore it is important that any tools work with systems for organizing information that already exist. In the context of R, that means that any new tools should work within the context of the *data frame*, which is the primary data type for storing data in R.

That is why the *tibble*, the preferred data storage form in the tidyverse extends the data frame rather than replacing it. Any package in the tidyverse can take as an argument either a tibble or a data frame,

22.3 Pipes make code easier

No one can hold a complex series of transformations entirely in their heads. Pipes give us a semantic way of breaking down such a series into their component parts. That way we can handle large tasks one step at a time.

So what does this mean when we begin writing functions of our own? There are a couple things to keep in mind

- Keep functions simple. That means it should have as few inputs as possible, and return only one thing. That makes it easy to chain functions together using pipes.
- Function names should be verbs when possible. That makes the piped code easier to read. The function `filter` filters out observations, `select` selects variables, and so on.

Of course, these are guidelines, not hard and fast rules. Most of the `geom_` functions in `ggplot2` for instance, are nouns rather than verbs, because they are adding a particular thing to the canvas.

22.4 Use functional programming

This is a big one, and so will take some explanation. There are two main types of programming paradigms.

1. *Imperative programming* Here the focus of the programmer is how to modify the state of the system in order to accomplish a task. Most commands are destructive, they remove an existing portion of the state and replace it with a new one. The Turing machine is the canonical example of imperative programming.
2. *Functional programming* Here the focus is on listing the transformations needed to get from the current state to the final state. Commands are non-destructive, they

indicate what to do to the existing state to move it in the desired direction. The Lambda Calculus is the canonical example of functional programming.

Note that neither of these paradigms is “right” or “better”. Instead, they have different strengths and weaknesses that encourage the user to think about their problem and write code to solve it in different ways.

Most procedural and object-oriented languages are imperative. On the other hand, since a statistic is a function of the data, many statistical analyses have a clearer form when written as a functional program.

So what makes a language functional? There are several properties that a functional language must have.

- Functions are mathematical functions, also known as *pure*.
- Variables are *immutable*, meaning they cannot be changed once assigned. This leads to *referential transparency*, where each variable name returns a unique value.
- Recursion is used for loops.
- Functions are First-Class and can also be Higher-Order. This means you can pass functions as input to other functions.

Functions are mathematical functions

In imperative programming, a **function** can either be like a mathematical function (for example: $y = x^2$) or it can be a set of commands that alters the state of the system in a destructive fashion.

Definition 65

In a programming language, a function is **pure** if it always produces the same output with the same input, and if there are no side-effects. That is, it does not change the value of the input variables or any global state.

In functional programming, functions are all just mathematical functions. Consider:

```
y <- function(x)
  return(x^2)
```

This code in R incorporates the function $y(x) = x^2$. It returns one thing, the output of the function.

Note that R is designed to assist in this type of programming by only allowing the return of a single object.

Variables are immutable

In a functional program, it is not permitted to change the value of a variable! Once you have assigned a variable, you cannot change its value.

Definition 66

Say that variables in a programming language are **immutable** if they can only be assigned once.

This is again to bring variables in line with how they are used in mathematics. For instance, if I write

$$y = x^2$$

$$y = -|x| - 2,$$

that does not make any sense, as y cannot be both of those things simultaneously.

Of course, R does *not* enforce variable immutability. It happily allows you to change the values of variables within a function or use functions to change the variable values.

So if you are going to use this principle, you will have to do it yourself. That means writing code like

```
x <- 4
y1 <- x*x
y2 <- 3*y1 + 2
y3 <- -y2
```

instead of

```
x <- 4
y <- x*x
y <- 3*x + 2
y <- -y
```

So why make variables immutable? It leads to a great advantage of functional programming called *referential transparency*

Definition 67

A programming language has **referential transparency** if every assigned variable has the same value throughout the program.

In other words, a particular name only references one value of the variable. That means that when writing code, you never have to worry about the same variable being used in two different ways, or the same function name being used for two different functions. You are guaranteed that the result will always stay the same.

This prevents you from accidentally changing the value of a variable and then expecting it to be the same as it was before. Or if you are collaborating in writing code on a large project, it prevents you from changing a variable in one part of the code that you are working on, thereby breaking code that your collaborator had finished.

Recursion is used instead of loops

But wait a minute, one of the most common constructions in programming languages is the *loop*, which executes a series of commands more than once. For instance, consider the following snippet of C code:

```
#include <stdio.h>

int main () {

    int a, s = 0;

    /* for loop execution */
    for( a = 10; a < 20; a = a + 1 ){
        s = s + a;
    }
    printf("%d\n",s);

    return 0;
}
```

I don't want to get too much into the details of this code, but I will say that this code calculates $\sum_{a=10}^{19} a = 145$. It does this by keeping track of the sum at each stage of the computation, and changing the variable *a* at each step. So what can go wrong? Well, suppose that I had a bug in my code:

```
#include <stdio.h>

int main () {

    int a, s = 0;

    /* for loop execution */
    for( a = 10; a < 20; a = a + 1 ){
        s = s + a;
        a = a - 1;
    }
    printf("%d\n",s);

    return 0;
}
```

}

Inside the for loops, the value of `a` is being reduced by one at each step, so in the execution of the for loop, it undoes the addition of 1 to `a`. This code will never stop, it will run forever!

That's bad! Remember that functional languages cannot change the values of variables once assigned, so they use the notation of *recursion* to solve this problem.

Definition 68

A function is defined **recursively** if it includes itself in the definition.

Let's see how we could build that same for loop using recursion. To do this, let's make the function a bit more general. Say that

$$s(n) = \sum_{a=10}^n a.$$

Then mathematically we can define $s(n)$ recursively as follows:

$$\begin{aligned} s(10) &= 10 \\ s(n) &= n + s(n - 1) \quad \text{when } n > 10. \end{aligned}$$

Note that like in induction proof, we have a *base case* $s(n) = 10$ and a *recursive case* $s(n) = n + s(n - 1)$. Now we build code where the function calls itself:

```
s <- function(n) {
  if (n == 10)
    return(10)
  else
    return(n + s(n - 1))
}

s(19)
```

```
## [1] 145
```

Note that we never had to redefine a variable in this program! Now, that being said, R does have a **for** loop, which we will see later on. However, it is partially recursive, in the sense that what happens inside a particular execution of the for loop stays in the for loop. That means that it is not possible to create the bug that we saw in C where the for loop variable was altered resulting in an infinite loop.

Functions are first-class and higher order

In R, we have seen that we use the assignment operator `<-` to assign the function to a particular name. So functions and variables are really the same type of object. This is what we mean by a function being *first-class*.

Definition 69

In a programming language, a function is **first-class** if it is treated like any other variable.

Since it is treated like any other variable, we can take functions as input to another function, and return functions as results. We call functions like these **higher-order**.

Definition 70

A function which takes a function as input, or returns a function as output, it called **higher-order**.

We have seen this behavior with the **ggplot** function, which takes a parameter **mapping** which is set equal to a function **aes** with its own parameters.

Functional programming and data science

So that's functional programming in a nutshell.

- R itself is not a fully functional language, but it incorporate enough features of functional languages that it is possible to do functional programming. Sticking to this paradigm is very helpful both for code readability and in large collaborative projects.
- Functional programming fits in very nicely with the data science view that we are transforming our data to make patterns obvious. Many languages such as Haskell used in data science are fully functional languages for this reason.

22.5 Designing the API for humans

The last principle for the design of the tidyverse is that it will be used by humans. We have not talked much about computational complexity in this course. That is partially because that would lead us deeper into the algorithm for accomplishing tasks than we plan to go here, but also because in practice most of the difficulty of data analysis comes from the human time, not the computer time.

Therefore, it is essential that you make your analysis as transparent as possible to humans, sometimes even at the cost of making the code slower.

This also informs the choice of function names. For instance, the geometry functions all begin with **geom_**. This makes them easier for people to remember, and also has the added benefit of making the autocomplete more powerful, as a user can scroll through a set of possibilities in order to decide what is appropriate.

In naming your functions, do not be afraid to have a lengthy name if the description power of the name is needed. Save short names for functions that will be used very often, and then overall your code will be much easier to read and use by others.

Writing Functions in R

Summary Users of R can write their own functions that then operate on an equal basis with the built-in functions.

Functions and conditional execution

| | |
|------------------|--|
| function | Make a function. |
| if | Execute the next command if the condition is true. |
| else | Execute the next command if the condition is false. |
| all | True if every element of a vector is true. |
| any | True if any element of a vector is true. |
| identical | True if variables are the same type and equal. |
| near | True if floating point variables are close in value. |
| switch | Switch among several different commands based on a variable. |

So far we've been relying on the commands and functions built into the tidyverse to accomplish tasks. But there will eventually come a time when you just doesn't exist a tool that does what you need it to do. At this point, you can write your own function!

The rule of thumb for writing your own functions is as follows. If you plan to use a particular bit of code three or more times, write it as a function. This will make your code more readable, and avoid unnecessary repetition.

As an example, consider the following:

```
set.seed(123456) # make same random choices every time
df <- tibble::tibble(
  a = rnorm(5),
  b = rnorm(5),
  c = rnorm(5),
  d = rnorm(5)
```

```

)
dfs <- df

dfs$a <- (df$a - min(df$a, na.rm = TRUE)) /
  (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
dfs$b <- (df$b - min(df$b, na.rm = TRUE)) /
  (max(df$b, na.rm = TRUE) - min(df$a, na.rm = TRUE))
dfs$c <- (df$c - min(df$c, na.rm = TRUE)) /
  (max(df$c, na.rm = TRUE) - min(df$c, na.rm = TRUE))
dfs$d <- (df$d - min(df$d, na.rm = TRUE)) /
  (max(df$d, na.rm = TRUE) - min(df$d, na.rm = TRUE))

dfs

## # A tibble: 5 x 4
##       a         b         c         d
##   <dbl> <dbl> <dbl> <dbl>
## 1 0.456  0.441 0.0515 0.330
## 2 0.0303 0.608 0.      0.
## 3 0.      1.02  0.462  0.693
## 4 0.170  0.558 1.00    1.00
## 5 1.00    0.     0.947  0.539

```

This code generates independent, identically distributed (iid) standard normal random variables as toy data to play with. We then rescale the data values so that they all lie between 0 and 1. There's several problems with this.

- First, it's hard to read. Even if you understand what you are doing when you write the code, it is not clear that you will understand six months or a year from now.
- Second, it's repetitive. That not only is poor style, but writing essentially the same thing down four times makes mistakes easy to slip in to just one out of the four.

To write the function, we first identify what the input to the function should be. In this case, the only thing that changes from line to line is `df$a`, `df$b`, `df$c`, `df$d`. So we really only need one input which is the variable vector of values that we are changing.

We write a function in R by assigning it to a variable name. The keyword **function** is followed by parenthesis that enclose the input to the function. The value that we wish to return is then passed back to the user using a **return** command.

The other thing to remember is that you can group commands together using curly braces, `{` and `}`. Everything between these symbols will be executed in turn, one after the other. Since most functions need more than one command, most times the function will start with `{` and end with `}`. The result is something like this.

```
scale01 <- function(x) {
  a <- min(x, na.rm = TRUE)
  b <- max(x, na.rm = TRUE)
  return((x - a)/(b - a))
}
```

Let's try it out on some examples:

```
scale01(c(-5, 0, 5))
```

```
## [1] 0.0 0.5 1.0
```

```
scale01(c(0, 1, 2, 3, 4, 5))
```

```
## [1] 0.0 0.2 0.4 0.6 0.8 1.0
```

Seems to be doing the right thing!

Now that we have a function, we can use it the way we would any other function to mutate our tibble.

```
df %>%
  mutate(a = scale01(a), b = scale01(b), c = scale01(c),
         d = scale01(d))
```

```
## # A tibble: 5 x 4
##       a      b      c      d
##   <dbl> <dbl> <dbl> <dbl>
## 1 0.456  0.430 0.0515 0.330
## 2 0.0303 0.594 0.      0.
## 3 0.      1.00  0.462  0.693
## 4 0.170  0.544 1.00    1.00
## 5 1.00    0.     0.947  0.539
```

One of the big advantages of using functions is that if we decide later to alter the function, we only have to change the function in one place. For instance, if we want to rescale from 0 to 100 percent instead of from 0 to 1, we modify the function as follows:

```
scale01 <- function(x) {
  a <- min(x, na.rm = TRUE)
  b <- max(x, na.rm = TRUE)
  return(100*(x - a)/(b - a))
}
```

Now our output would be

```
df %>%
  mutate(a = scale01(a), b = scale01(b), c = scale01(c),
         d = scale01(d))

## # A tibble: 5 x 4
##       a      b      c      d
##   <dbl> <dbl> <dbl> <dbl>
## 1  45.6  43.0   5.15  33.0
## 2   3.03 59.4    0.    0.
## 3    0.  100.   46.2  69.3
## 4  17.0  54.4  100.  100.
## 5  100.    0.   94.7  53.9
```

23.1 Things to keep in mind when writing functions

There are several things to think about when writing your own function.

- Don't make function names too long. If you have a function like `scale.one.hundred.numbers.in.vector` you will make your code too long to read and unwieldy to use.
- Don't make function names too short. If you call your function `f` you won't be able to easily remember what it does.
- In general, your function names should be verbs when possible, and the inputs should be nouns.
- Pick a style: separate words by `_` (as the tidyverse does) or `.` (as most of the base R functions do) or by using lower and upper case as `functionName`. But do not mix styles within your function or your code will quickly become unreadable!
- Do comment your code when you can. Remember that any line after the `#` symbol is not executed by `rsoft`. Use that to fill out and explain what's going on with your code.

```
scale01 <- function(x) {
  # This function scales data to lie between 0 and 1
  a <- min(x, na.rm = TRUE)
  b <- max(x, na.rm = TRUE)
  return(100*(x - a)/(b - a))
}
```

- Don't comment on the function of basic commands in R. Your audience will know what `min`, `max`, `sum`, and `mean` do. The following is too much:

```
scale01 <- function(x) {
  a <- min(x, na.rm = TRUE) # find the min value of x
  b <- max(x, na.rm = TRUE) # find the max value of x
  return(100*(x - a)/(b - a))
}
```

23.2 If and else

Sometimes you only want commands to execute if a certain condition is true. That's when you want to use the `if` command. Consider the following.

```
sqrt_abs <- function(a) {
  if (a > 0)
    return(sqrt(a))
  else
    return(sqrt(-a))
}
```

Let's try it out!

```
sqrt_abs(4)
```

```
## [1] 2
```

```
sqrt_abs(-9)
```

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

From this example, you can see how the form of the keywords `if` and `else` work.

- Put the condition inside parentheses. Here `(a > 0)` is the condition under which the `if` statement executes the following command.
- After the command you want executed when the condition is true, you can then (optionally) place a `else` keyword. After the `else`, put the command that you want to execute if the condition is *not* true.
- Remember that if you want more than one command to execute after an `if` statement, just enclose them in curly braces.

Two common errors to watch out for:

- The condition in the **if** command must evaluate to be either **TRUE** or **FALSE**. If it evaluates to a vector of possibilities, you will get an error.

```
sqrt_abs(c(4, -9))
```

```
## Warning in if (a > 0) return(sqrt(a)) else return(sqrt(-a)) :
## the condition has length > 1 and only the first element will
## be used
```

```
## Warning in sqrt(a) : NaNs produced
```

```
## [1] 2 NaN
```

- You will also get an error in this function if you pass something that is not a number.

```
sqrt_abs("a")
```

```
## Error in sqrt(a) : non-numeric argument to mathematical function
```

Previously we used `|` for logical or and `&` for logical and to apply to vectors. Inside an **if** statement, we use `||` and `&&`, which only return the first element of a vector of **TRUE** and **FALSE** values.

Now the equality testing operator `==` is also vectorized, so it will also return more than one **TRUE/FALSE** value. You can use the command `any` to require that at least one equality works, or `all` to require that at every equality works. We can use this to rewrite our function:

```
sqrt_abs <- function(a) {
  if (any(a < 0))
    a <- abs(a)
  return(sqrt(a))
}
```

```
sqrt_abs(c(4, -9))
```

```
## [1] 2 3
```

An alternative to the equality operator is **identical**, which only matches if its inputs are the same value and the same type. For instance, by default `0` is a floating point number, while `0L` is an integer. So:

```
identical(0, 0)
```

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

```
identical(0L, 0)
```

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

When doing floating point computations, we can use **near** to detect numbers that are very close together.

```
sqrt(2)^2 == 2
```

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

```
near(sqrt(2)^2, 2)
```

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

Multiple conditions

The basic **if** works with a condition with two choices: **TRUE** or **FALSE**. To deal with more than two possibilities, one thing we can do is chain **if** statements together.

```
three_choices <- function(a) {
  if (a < 4) {
    if (a < 2)
      print("Less than 2.")
    else
      print("At least 2 but less than 4.")
  } else
    print("4 or greater.")
}
```

```
three_choices(1)
```

```
## [1] "Less than 2."
```

```
three_choices(3)
```

```
## [1] "At least 2 but less than 4."
```

```
three_choices(5)
```

```
## [1] "4 or greater."
```

Another way is to break into choices based on the value of a variable with the **switch** command. For example:

```
four_choices <- function(a, op) {
  switch(op,
    double = 2*a,
    square = a^2,
    absolute = abs(a),
    cube = a^3,
    stop("Unknown operation.")
  )
}
```

```
four_choices(4, "cube")
```

```
## [1] 64
```

```
four_choices(4, "triple")
```

```
## Error in four_choices(4, "triple"): Unknown operation.
```

Notice the code style that we are using. We put the leading `{` on the end of the line it starts, but the ending `}` goes at the beginning of a line. It stays on its own line with nothing except possibly an **else** keyword following it.

Everything inside a **function** is indented two spaces, and then inside an **if** is two more spaces, and so on.

23.3 Arguments

For a function arguments, we can assign a *default* value that is given to the function if we do not assign a value. For instance, we can modify our previous parameter:

```
four_choices <- function(a, op = "double") {
  switch(op,
    double = 2*a,
    square = a^2,
    absolute = abs(a),
    cube = a^3,
    stop("Unknown operation.")
  )
}
```

```
)
}

four_choices(4)
```

```
## [1] 8
```

Note the order we put the arguments in. It is good form to put arguments with default values after those that do not. Usually arguments that take in data come first, while those that modify what the function does come later. Often arguments with defaults are called parameters.

When naming your arguments and parameters, try to use the same names as base R functions or tidyverse functions when possible. If setting a parameter to **TRUE** in your function removes all the **NA** values first, then call the parameter `na.rm` as in **mean**.

23.4 Arbitrary numbers of arguments

Some commands in R take an arbitrary number of arguments. For instance:

```
min(5, 4, 3)
```

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

```
min(5, 4, 3, 2, 1)
```

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

To do this in your own code, we use a special argument name `...` that is three dots in a row. This can then be passed to other functions to deal with an unknown number of parameters.

```
str_alternate <- function(...)
  return(str_c(..., sep = "|"))

str_alternate("red", "green", "blue")
```

```
## [1] "red|green|blue"
```

Part IV

MODELING

Modeling data

Summary A **model** is a simplification of the real world that helps us make decisions and predictions. The variables used to make predictions are called **prediction** variables, while the variables we are trying to predict are called **response** variables. If these are numerical, the difference between a prediction and the response is called the **error** or **residual**.

Modeling

| | |
|------------------------|--|
| optim | Optimizes a function. |
| data_grid | Creates grid of unique prediction values. |
| add_predictions | Gives predictions based upon a model. |
| add_residuals | Gives the residuals from the predictions and response variables. |

You’ve tidied and cleaned your data, done some visualizations to see what’s going on, and think you have a pattern. What next? One of the key pieces of data science is *modeling* your data. A model is an simplification of the real world that allows us to predict outcomes.

For instance, a map is a model that is very useful.

A map does not try to incorporate every building, every tree, every bush. It does not record the markings on the middle of a road because they do not matter to the purpose of the map. That is because the location of the roads is useful when planning a route, while knowing the trees on the side of the road are not.

Instead, a model seeks to capture what is important about the thing that it is modeling. It allows the user to make informed decisions, and understand what is important in the data, and what is not.

This notion is captured by a famous quote of George Box: “All models are wrong, but some are useful.” No model completely captures the nature of the thing it is modeling. A

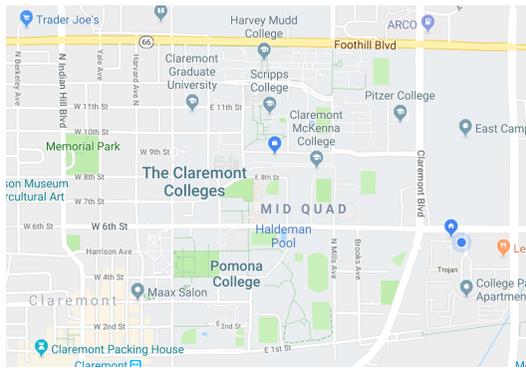


Figure 24.1: A map of the Claremont College

useful model strips away unnecessary information to leave you with a bare bones skeleton that tries to capture the essence of what it going on.

24.1 Linear models

The simplest relationship between two observed variables is linear. Sometimes this relationship is formed by definition. Since an inch is 2.54 centimeters, one's height in inches varies linearly with one's height in centimeters.

Other times the connection is more tenuous: does one's performance on a test vary linearly with the amount of time one studies? This simple model can be written as

$$y = c_0 + c_1x.$$

Here we have two *parameters*, c_0 and c_1 . Of course, such a line never fits real data precisely, so instead we could write

$$y_i = c_0 + c_1x_i + \epsilon_i,$$

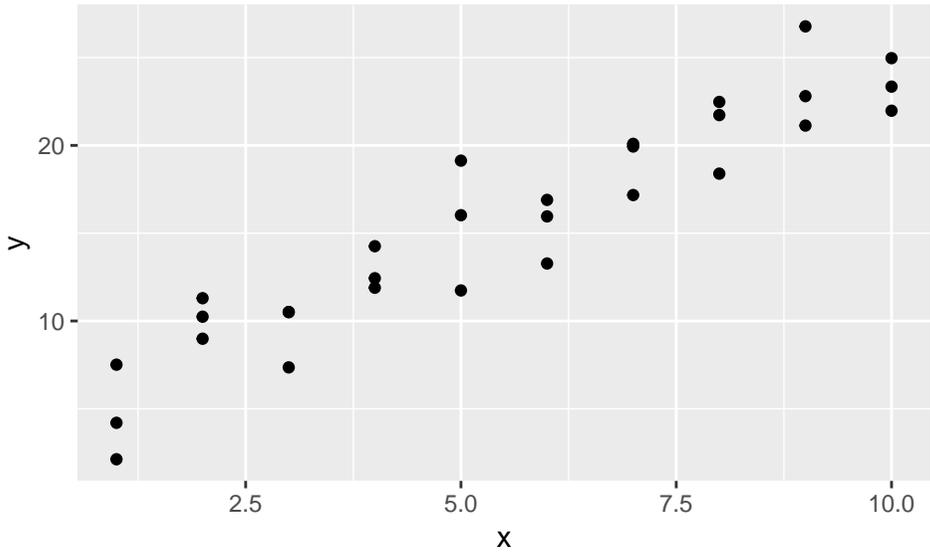
where (x_i, y_i) is the i th observation, and ϵ_i is what is called an *error* or *random effect*. We model ϵ_i as a *random variable*. A random variable is a variable about which we have partial information. So we don't know it exactly, but we do know something about it. For instance, we might model ϵ_i as having equal chance of being positive or negative.

In the `modelr` package, there is a simulated dataset `sim1`. Let's take a look at this data:

```
# install.packages("modelr")
library(modelr)
```

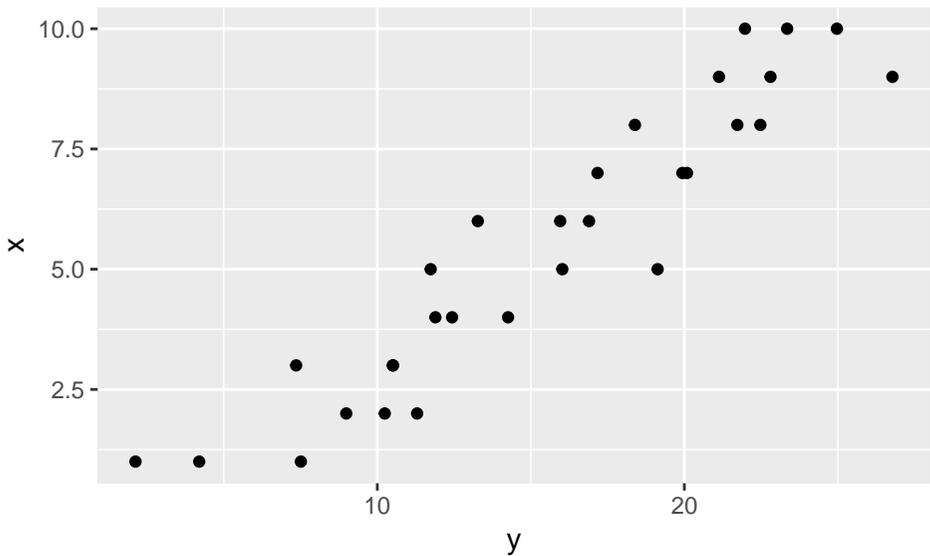
```
## Warning: package 'modelr' was built under R version 3.4.4
```

```
ggplot(sim1, aes(x, y)) +
  geom_point()
```



Now that's what we've been talking about: the y_i values look like they are varying roughly linearly on the x_i values. But here's an important note: don't get too caught up in the x versus y distinction. If I had swapped the axis of x and y that vary linearly, we would also have a linear relationship!

```
ggplot(sim1, aes(y, x)) +
  geom_point()
```



That being said, note that there does exist an asymmetry in the data: for each value x can take on, y can take on multiple values. And the values that the x_i can take on are positive integers.

```
sim1

## # A tibble: 30 x 2
##       x     y
##   <int> <dbl>
## 1     1  4.20
## 2     1  7.51
## 3     1  2.13
## 4     2  8.99
## 5     2 10.2
## 6     2 11.3
## 7     3  7.36
## 8     3 10.5
## 9     3 10.5
## 10    4 12.4
## # ... with 20 more rows
```

Therefore, in this case it makes more sense to model $y_i = c_0 + c_1x_i + \epsilon_i$. This raises the question: what should c_0 and c_1 be?

A simple way of doing this would be to choose the line that passes through the middle y value when $x = 1$, and the middle y value when $x = 10$.

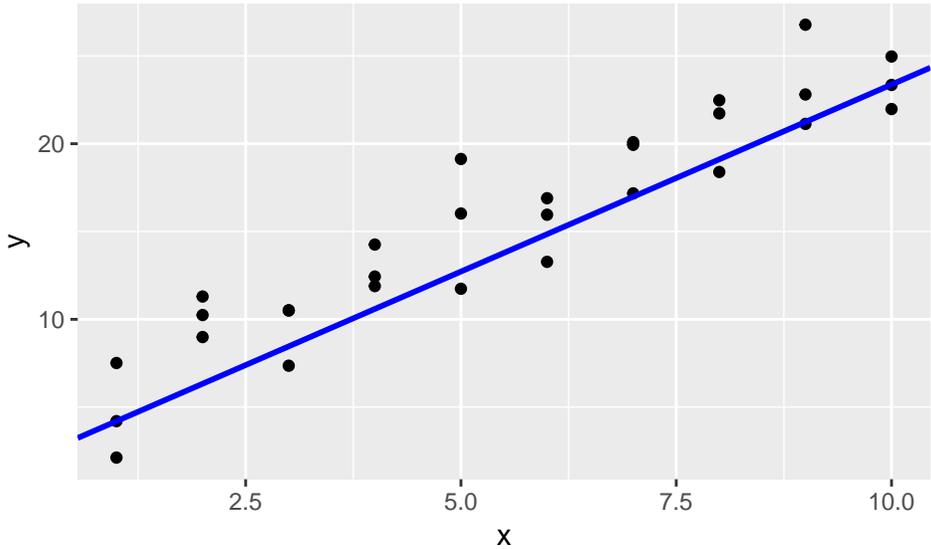
```
sim1 %>%
  filter(x == 1 | x == 10) %>%
  arrange(y)
```

```
## Warning: package 'bindrcpp' was built under R version 3.4.4

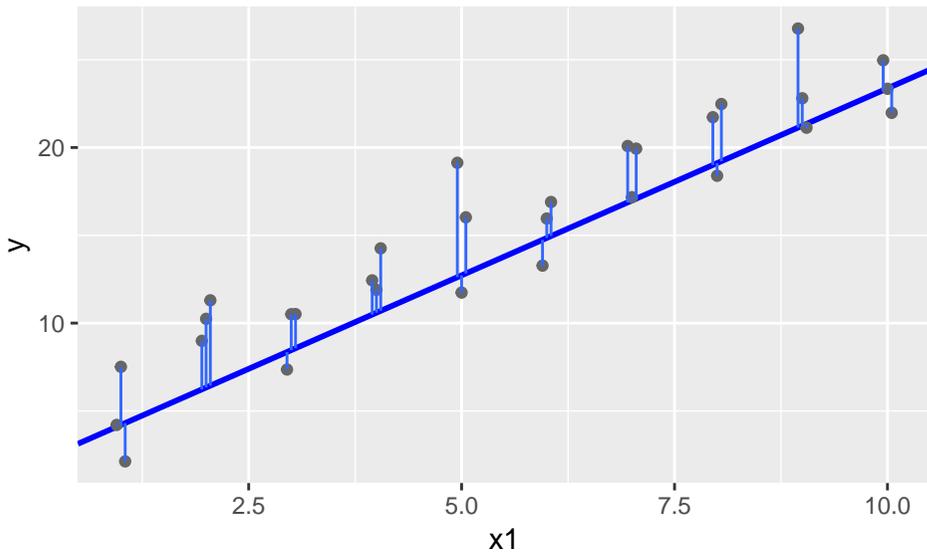
## # A tibble: 6 x 2
##       x     y
##   <int> <dbl>
## 1     1  2.13
## 2     1  4.20
## 3     1  7.51
## 4    10 22.0
## 5    10 23.3
## 6    10 25.0
```

So this line is $y_i = (x - 1)(23.34 - 4.2)/(10 - 9) + 4.2 = 2.12x + 2.08$. We can put this on the plot of the data with the `geom_abline` function.

```
sim1 %>%
  ggplot(aes(x, y)) +
  geom_point() +
  geom_abline(aes(intercept = 2.08, slope = 2.12),
             col = "blue", lwd = 1)
```



Looks okay, but maybe a little lower than we would think. One way to see how well we are doing is to measure the errors ϵ_i .



Definition 71

Given a model $y_i = c_0 + c_1x_i + \epsilon_i$, call $\hat{y}_i = c_0 + c_1x_i$ the **prediction**, y_i the **response**, and ϵ_i the **error** or **residual**.

In an idea world, all the errors would be zero and the prediction would exactly match the response. This never happens in real data though: too many ways for the measurement to go wrong or for the model to not exactly capture the process.

So we instead just try to make the magnitude of the error as small as possible. Without going into too much detail, the most common way of modeling the error is an a *normal random variable* centered at 0. This leads to something in statistics called the *maximum likelihood estimate*, or MLE. For us, the important thing about the MLE is that it occurs at the place where the sum of of the squares of the residuals are as small as possible.

Definition 72

In a model, the choice of parameters than minimizes the sum of the squares of the errors is called the **least squares** fit.

To find these errors, let's make a function that is our model that takes a tibble or data frame with variable x , and returns a prediction given the slope and intercept of a linear model.

```
modell <- function(c, data) {
  return(c[1] + c[2] * data$x)
}
modell(c(2.07, 2.13), sim1)
```

```
## [1] 4.20 4.20 4.20 6.33 6.33 6.33 8.46 8.46 8.46
## [10] 10.59 10.59 10.59 12.72 12.72 12.72 14.85 14.85 14.85
## [19] 16.98 16.98 16.98 19.11 19.11 19.11 21.24 21.24 21.24
## [28] 23.37 23.37 23.37
```

Those are the predicted values. Now let's measure the sum of the squares of the error, when the response values are in variable y .

```
sum_square_error <- function(mod, data) {
  diff <- data$y - modell(mod, data)
  return(sum(diff^2))
}
sum_square_error(c(2.07, 2.13), sim1)
```

```
## [1] 226.0735
```

So let's see if we can make this smaller by pushing the line up a little bit.

```
sum_square_error(c(2.1, 2.13), sim1)
```

```
## [1] 223.0058
```

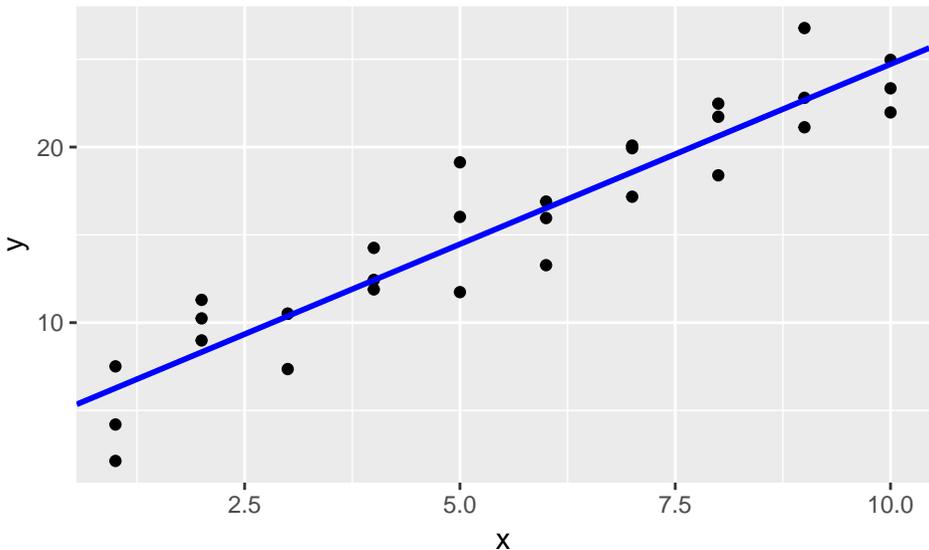
Better! By playing around with the intercept and slope, we can make this smaller and smaller. In fact, R has a built in function to do exactly that. It is called **optim**, and it works using a method called *Newton-Raphson*. When we try it would on our data:

```
optimal_coef <- optim(c(2.07, 2.13), sum_square_error, data = s
optimal_coef$par
```

```
## [1] 4.220708 2.051585
```

So our slope was pretty good to start with, but our intercept needed to be quite a bit higher! Plotting this line gives:

```
sim1 %>%
  ggplot(aes(x, y)) +
  geom_point() +
  geom_abline(aes(intercept = 4.22, slope = 2.05),
             col = "blue", lwd = 1)
```



Note that we passed the function **sum_square_error** as a parameter to **optim**. That's Functional Programming in action!

The square root of the sum of squares of a vector is also called the L_2 -distance (or Euclidean distance). Another way of measuring the distance is the L_1 -distance, which is the sum of the absolute values of the vector. With this, we get a slightly different line:

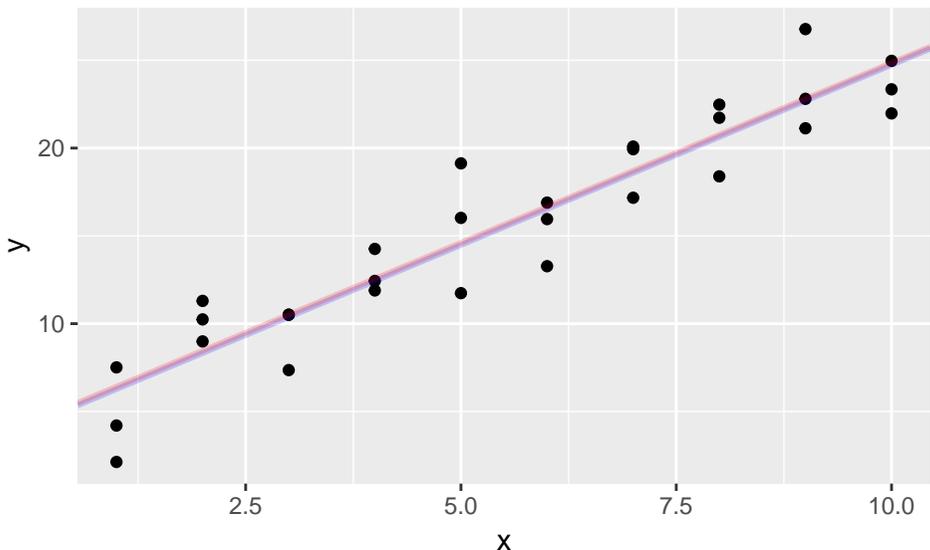
```
sum_abs_error <- function(mod, data) {
  diff <- data$y - modell(mod, data)
  return(sum(abs(diff)))
}

optimal_coef2 <- optim(c(2.07, 2.13), sum_abs_error,
  data = sim1)

optimal_coef2$par
```

```
## [1] 4.364849 2.048918
```

```
sim1 %>%
  ggplot(aes(x, y)) +
  geom_point() +
  geom_abline(aes(intercept = optimal_coef$par[1],
    slope = optimal_coef$par[2]), col = "blue", lwd = 1,
    alpha = 0.2) +
  geom_abline(aes(intercept = optimal_coef2$par[1],
    slope = optimal_coef2$par[2]), col = "red", lwd = 1,
    alpha = 0.2)
```



In this case, the two lines are pretty much on top of each other, but if I have an outlier, a point far away from the group, it will drag the L_2 -distance (least squares) line much more than the L_1 -distance line.

```
simlaug <- sim1 %>% union(tibble(x = 10, y = 60))

optimal_coef3 <- optim(c(2.07, 2.13), sum_square_error,
                      data = simlaug)

optimal_coef3$par
```

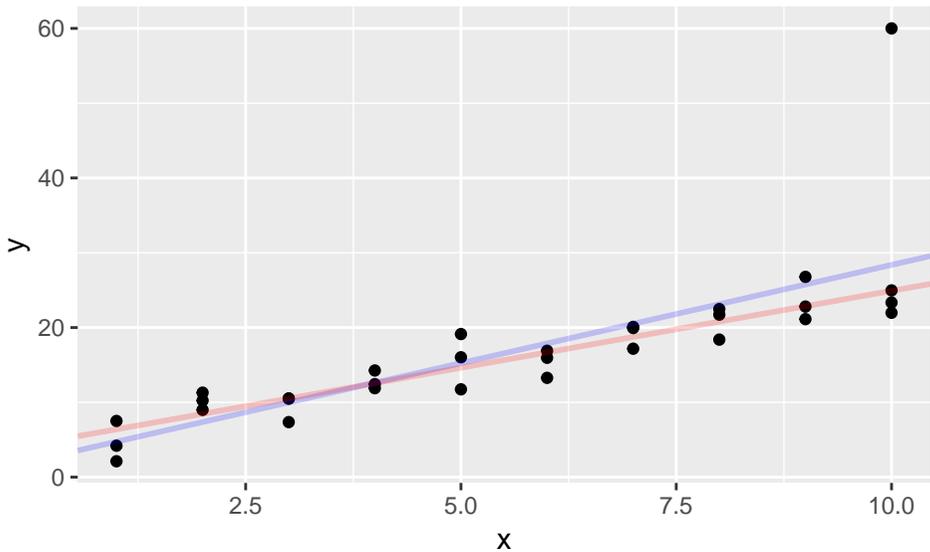
```
## [1] 2.113694 2.626334
```

```
optimal_coef4 <- optim(c(2.07, 2.13), sum_abs_error,
                      data = simlaug)

optimal_coef4$par
```

```
## [1] 4.340730 2.056957
```

```
simlaug %>%
  ggplot(aes(x, y)) +
  geom_point() +
  geom_abline(aes(intercept = optimal_coef3$par[1],
                  slope = optimal_coef3$par[2]), col = "blue",
              lwd = 1, alpha = 0.2) +
  geom_abline(aes(intercept = optimal_coef4$par[1],
                  slope = optimal_coef4$par[2]), col = "red",
              lwd = 1, alpha = 0.2)
```



It is important to note that neither of these lines are “right”, but they are both useful in making predictions. The least squares line is used most often because it is easier to calculate, but for small problems, the least absolute value line is less influenced by outliers.

24.2 Using more than one predictor

It is actually pretty rare to have a single prediction variable. If we have n different such predictor variables, our linear model can be expanded to

$$y = c_0 + c_1x_1 + \dots + c_nx_n + \epsilon.$$

We do not have to use `optim` to deal with this more complicated state. Instead, we can use the `lm` command to fit a linear model. The `~` character is used to designate the model. The variable to be predicted goes on the left of `~`, and the variables used to predict go on the right. So for this model, we can use:

```
sim1_mod <- lm(y ~ x, data = sim1)
coef(sim1_mod)
```

```
## (Intercept)          x
##    4.220822    2.051533
```

Currently, the only type of error minimization that R can do is least squares. This is because there exists a deterministic algorithm for calculating the values exactly. This is different from `optim`, which is slower, but applies to a wide variety of models.

24.3 *modelr*

The package **modelr** contains several functions designed to test how well a model fits data. The first is **data_grid**, which takes a tibble as an argument and returns all combinations of the prediction variables. For instance,

```
grid <- sim1 %>%
  data_grid(x)
grid

## # A tibble: 10 x 1
##       x
##   <int>
## 1     1
## 2     2
## 3     3
## 4     4
## 5     5
## 6     6
## 7     7
## 8     8
## 9     9
## 10    10
```

There are 10 different unique values for x , and so that is our starting grid.

Once we have our model (such as created through **lm**), we can add it to our grid with **add_predictions**

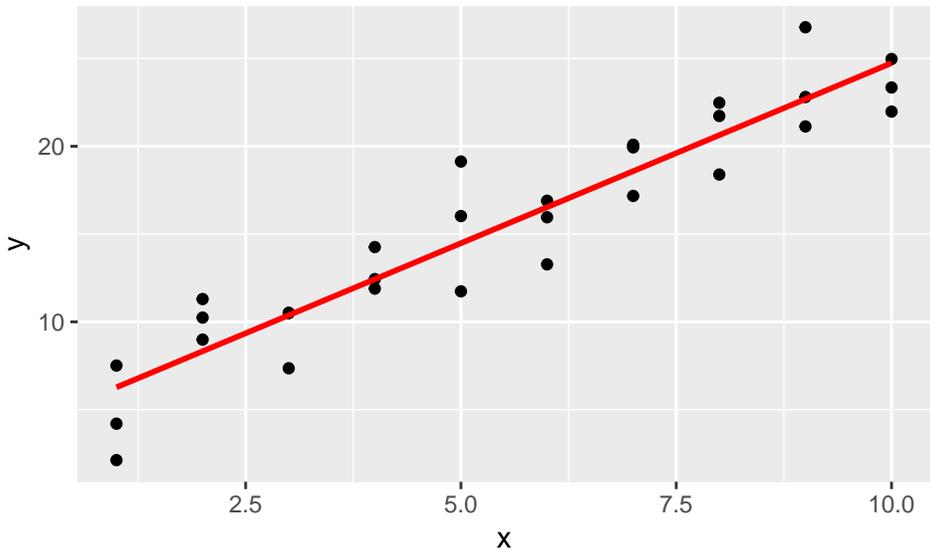
```
grid_pred <- grid %>%
  add_predictions(sim1_mod)
grid_pred

## # A tibble: 10 x 2
##       x pred
##   <int> <dbl>
## 1     1  6.27
## 2     2  8.32
## 3     3 10.4
## 4     4 12.4
## 5     5 14.5
## 6     6 16.5
## 7     7 18.6
## 8     8 20.6
```

```
## 9      9 22.7
## 10     10 24.7
```

Now we have predictions for several x values. Now let's add these predicted values to the plot of the data.

```
ggplot(sim1, aes(x)) +
  geom_point(aes(y = y)) +
  geom_line(aes(x = grid_pred$x, y = grid_pred$pred), data = gr
```

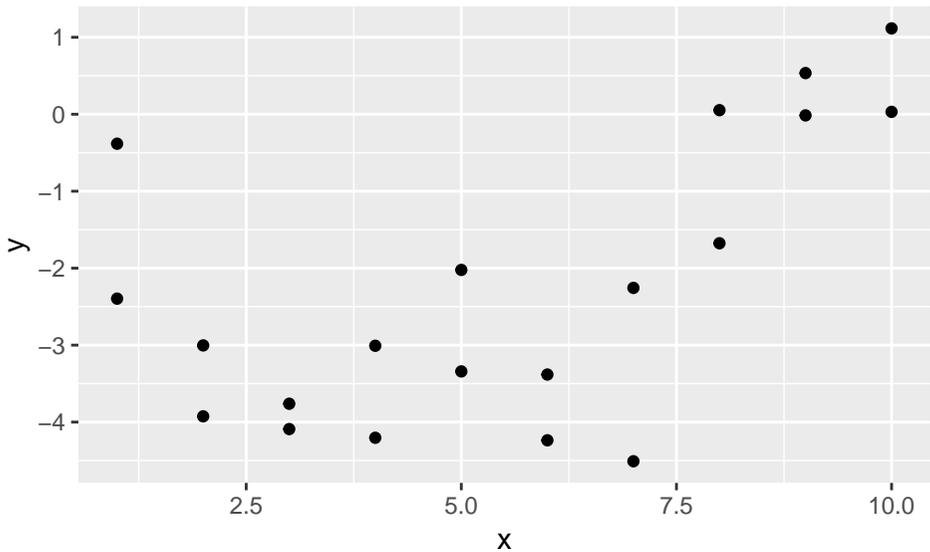


Hooray! We got exactly what we did earlier with `geom_abline`. But in actuality, this way of doing things is far more general. If we had used a more complicated model, then this would also have worked for predictions.

For instance, let's generate some data from the model

$$y = 0.5 - 0.8x + 1.2x^2 + \epsilon.$$

```
set.seed(123456)
x <- 1:10
sim5 <- tibble(x = c(x, x)) %>% mutate(x_square = x^2) %>% mutata
sim5 %>%
  ggplot() +
  geom_point(aes(x, y))
```



Now suppose we try to fit this model to the data:

```
sim5_mod <- lm(y ~ x + x_square, data = sim5)
coef(sim5_mod)
```

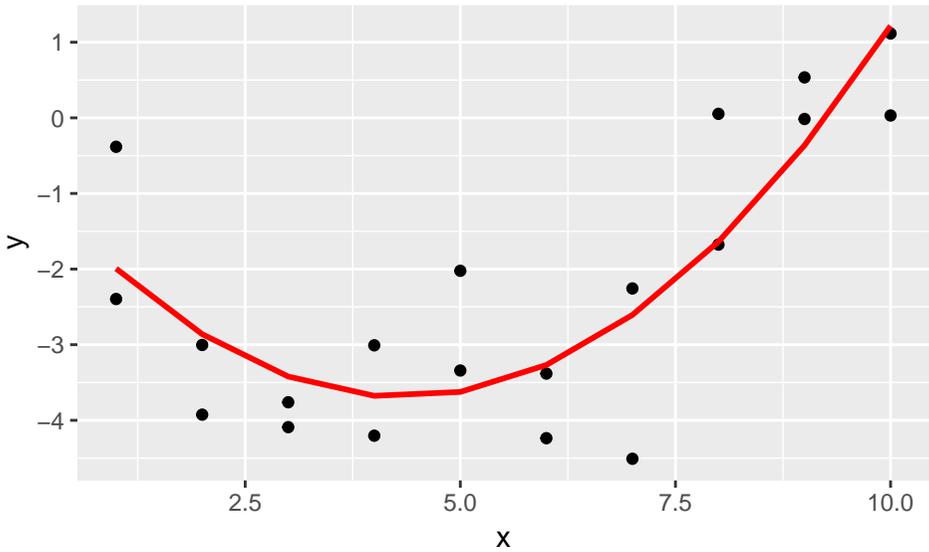
```
## (Intercept)          x      x_square
## -0.8217856  -1.3251656   0.1528941
```

```
grid5_pred <-
  tibble(x = 1:10, x_square = (1:10)^2) %>%
  add_predictions(sim5_mod)
grid5_pred
```

```
## # A tibble: 10 x 3
##       x x_square  pred
##   <int>   <dbl> <dbl>
## 1     1     1.0 -1.99
## 2     2     4.0 -2.86
## 3     3     9.0 -3.42
## 4     4    16.0 -3.68
## 5     5    25.0 -3.63
## 6     6    36.0 -3.27
## 7     7    49.0 -2.61
## 8     8    64.0 -1.64
## 9     9    81.0 -0.364
## 10    10   100.0  1.22
```

Now let's plot the original data and the predictions.

```
ggplot(sim5, aes(x)) +  
  geom_point(aes(y = y)) +  
  geom_line(aes(x = grid5_pred$x, y = grid5_pred$pred), data = c
```



Residuals

Summary

The package `modelr` gives us tools for seeing models in action.

| Modeling | |
|---------------------------------|---|
| <code>add_residuals</code> | Gives the residuals from predictors and the model. |
| <code>gather_residuals</code> | Bring together residuals from more than one model for comparison. |
| <code>gather_predictions</code> | Bring together predictions from more than one model for comparison. |
| <code>model_matrix</code> | The matrix X for a given model. |

25.1 Understanding residuals

Once we have a prediction, we can turn to studying our *residuals*, the difference between our predictions from our model, and our true value.

Let's create our model for our simulated data again:

```
library(tidyverse)
library(modelr)

## Warning: package 'modelr' was built under R version 3.5.2

sim1_mod <- lm(y ~ x, data = sim1)
coef(sim1_mod)

## (Intercept)          x
## 4.220822      2.051533
```

Now let's look at how off our predictions are from our model using `add_residuals`.

```

sim1 <- sim1 %>%
  add_residuals(sim1_mod)
sim1

## # A tibble: 30 x 3
##       x     y   resid
##   <int> <dbl> <dbl>
## 1     1   4.20 -2.07
## 2     1   7.51  1.24
## 3     1   2.13 -4.15
## 4     2   8.99  0.665
## 5     2  10.2   1.92
## 6     2  11.3   2.97
## 7     3   7.36 -3.02
## 8     3  10.5   0.130
## 9     3  10.5   0.136
## 10    4  12.4   0.00763
## # ... with 20 more rows

```

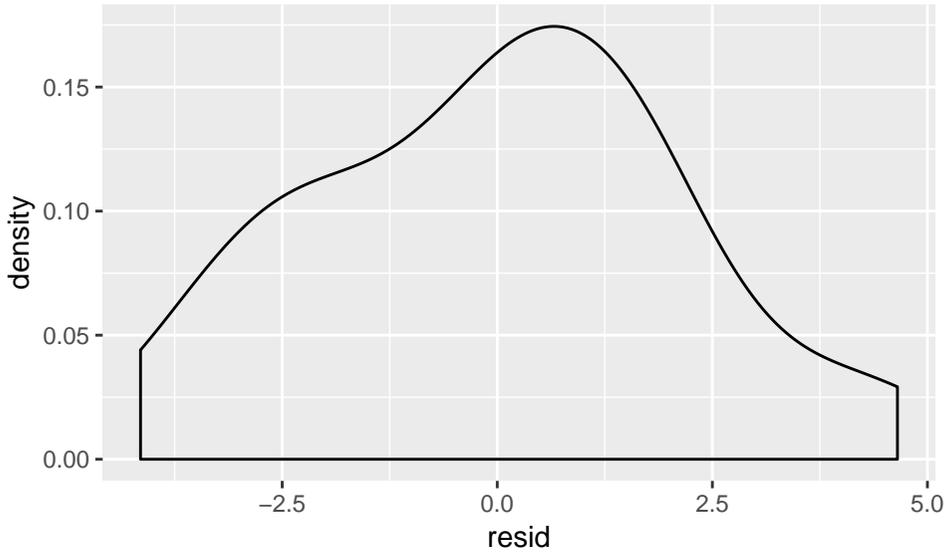
If our model is working, we would expect some of the residuals to be positive, and some to be negative. The least squares line that we fit is designed to make the mean of the residuals equal to 0.

```
mean(sim1$resid)
```

```
## [1] 5.121872e-15
```

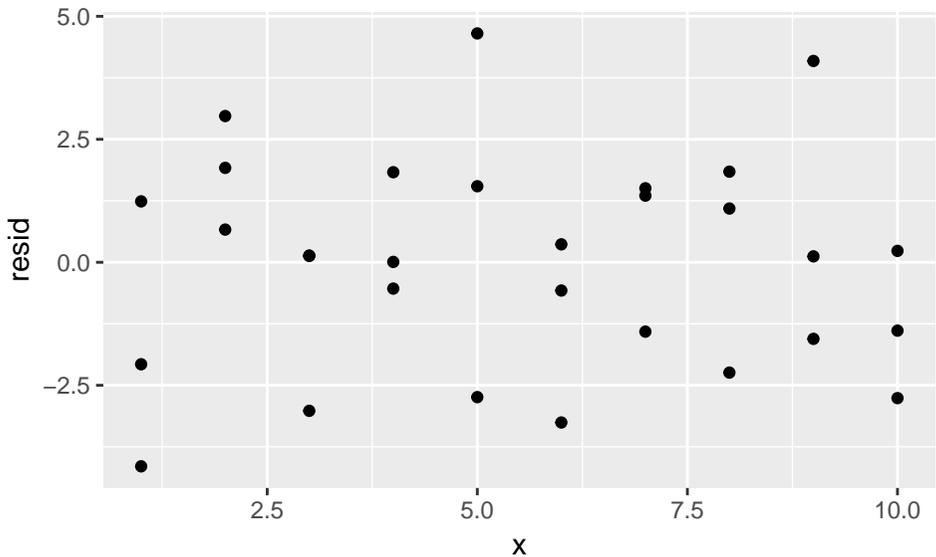
We would also expect them to be spread out. We can look at a kernel density plot or a histogram to get an idea of what they are like.

```
sim1 %>% ggplot() + geom_density(aes(resid))
```



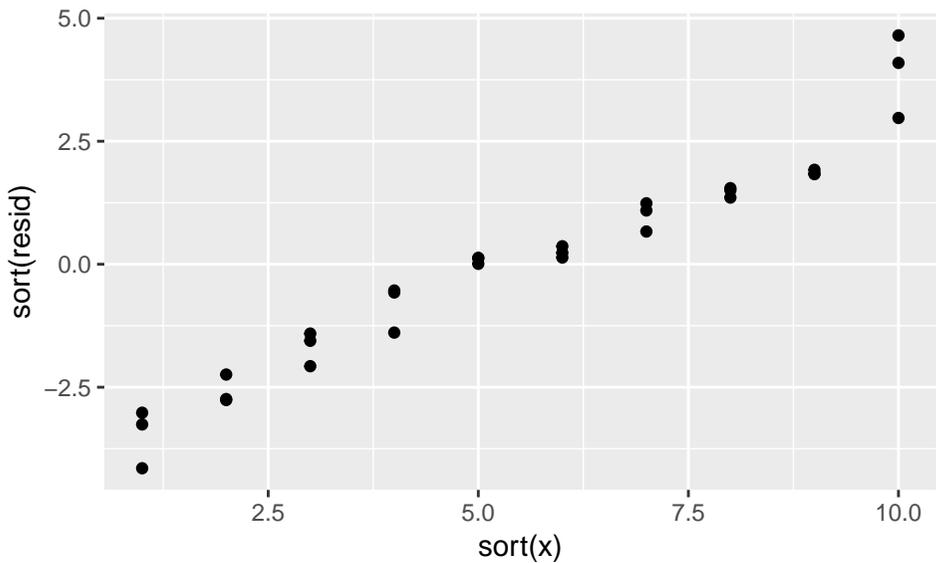
If our model is correct, then the residuals should all be independent of each other, there should not be a pattern. A scatterplot can show this:

```
sim1 %>% ggplot(aes(x, resid)) + geom_point()
```



If there had been a pattern, it might have looked something like this instead:

```
sim1 %>% ggplot(aes(x = sort(x), y = sort(resid))) + geom_point
```



Those values are not independent of each other!

25.2 Notation for models

We use statistical model notation to quickly describe how the response varies with the predictors in the model. For instance,

$$y \sim x$$

is the same as the model

$$y = c_0 + c_1x + \epsilon.$$

The model

$$y \sim x_1 + x_2 + x_1x_2$$

gives the model

$$y = c_0 + c_1x_1 + c_2x_2 + c_3x_1x_2 + \epsilon.$$

We usually say that we have three predictors, x_1 , x_2 , and an interaction predictor x_1x_2 . The constant term is usually not considered a predictor.

Definition 73

The process of choosing parameter values for a model based on the data is called **fitting** a model.

If we have n observations of the predictors and the response, then we can also write the model using linear algebra notation. In this notation, we write

$$Y = XC + \epsilon.$$

Hence c is a vector with p entries. The matrix X has n rows and $p + 1$ columns. This gives a column for each of the p predictors plus the constant predictor. The vector Y is then a vector of length n .

To see how this matrix X might be formed, consider the following data set:

```
df <- tribble(
  ~y, ~x1, ~x2,
  4, 2.3, 5.2,
  5, 1.1, 6.9,
  6, -1.4, 2.6
)
```

Then we can see the model matrix for a constant and x_1 predictor as follows.

```
model_matrix(df, y ~ x1)
```

```
## # A tibble: 3 x 2
##   `(Intercept)`    x1
##         <dbl> <dbl>
## 1             1  2.3
## 2             1  1.1
## 3             1 -1.4
```

Note that the first column is just 1's. This indicates that no matter what the x_1 value is, the first column just returns the constant term of the model. We can make the matrix more complicated by adding a second predictor:

```
model_matrix(df, y ~ x1 + x2)
```

```
## # A tibble: 3 x 3
##   `(Intercept)`    x1    x2
##         <dbl> <dbl> <dbl>
## 1             1  2.3  5.2
## 2             1  1.1  6.9
## 3             1 -1.4  2.6
```

Finally, we add the interaction term:

```
model_matrix(df, y ~ x1 + x2 + x1 * x2)
```

```
## # A tibble: 3 x 4
##   `(Intercept)`    x1    x2 `x1:x2`
##         <dbl> <dbl> <dbl>   <dbl>
```

```
## 1      1    2.3    5.2    12.0
## 2      1    1.1    6.9     7.59
## 3      1   -1.4    2.6    -3.64
```

It is represented by either $x_1 * x_2$ or $x_1 : x_2$. This is known as Wilkinson-Rodgers notation [2].

What about if the predictor is a categorical rather than a numerical value. For instance, consider the following dataset.

```
df <- tribble(
  ~gender, ~response,
  "male", 4.3,
  "female", 2.1,
  "male", 5.6
)
```

Because the `gender` variable is categorical, when creating the model matrix, it gets translated to either 0 or 1 depending on if it is male or female. It appends the level `male` to the end of `gender` to form `gendermale` to indicate that a 1 means that the gender level is male.

```
model_matrix(df, response ~ gender)
```

```
## # A tibble: 3 x 2
##   `(Intercept)` gendermale
##   <dbl>         <dbl>
## 1         1           1
## 2         1           0
## 3         1           1
```

(Note that there is no point in creating a `genderfemale` variable. Since the sum of the two vectors would be 1, the two columns together with the constant column would be *linearly dependent*. Therefore all information needed is contained in the one variable.)

Consider the `sim2` dataset from the `modelr` package.

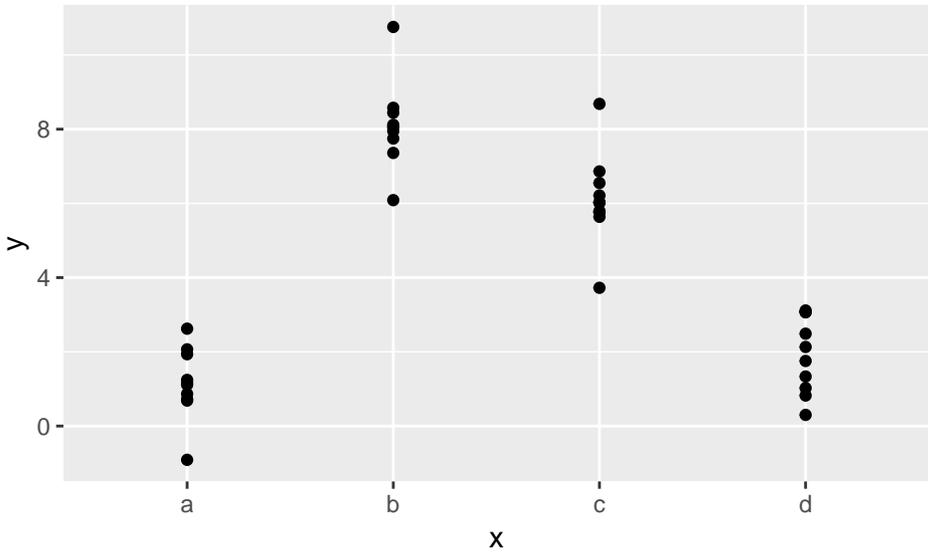
```
print(sim2, n = 5)
```

```
## # A tibble: 40 x 2
##   x      y
##   <chr> <dbl>
## 1 a     1.94
## 2 a     1.18
## 3 a     1.24
```

```
## 4 a      2.62
## 5 a      1.11
## # ... with 35 more rows
```

The variable `x` has four levels, `a`, `b`, `c`, and `d`.

```
sim2 %>% ggplot() +
  geom_point(aes(x, y))
```



Because this is categorical data, a model will just try to fit the average value for each of the levels.

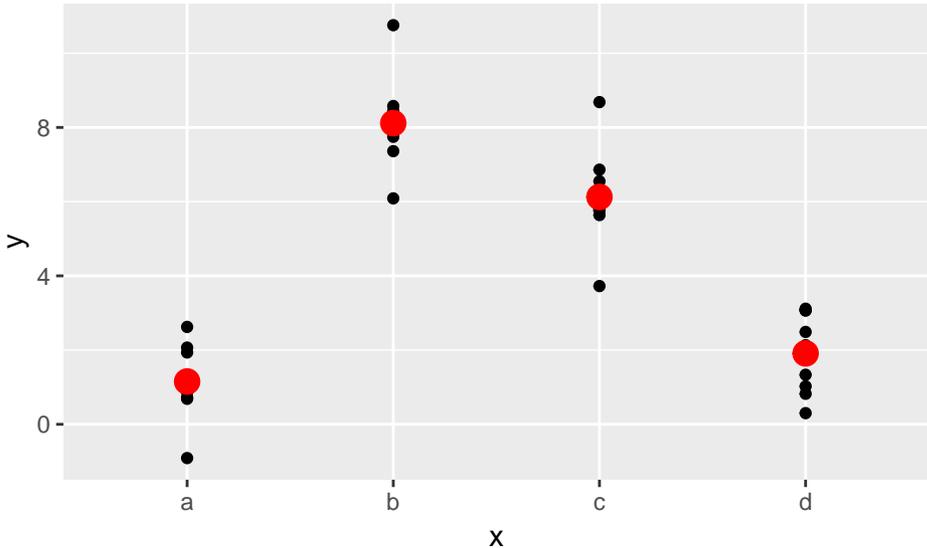
```
mod2 <- lm(y ~ x, data = sim2)

grid <- sim2 %>%
  data_grid(x) %>%
  add_predictions(mod2)
grid
```

```
## # A tibble: 4 x 2
##   x      pred
##   <chr> <dbl>
## 1 a      1.15
## 2 b      8.12
## 3 c      6.13
## 4 d      1.91
```

It turns out that the sample average of the values minimizes the sum of the squares of the distance from the prediction. We can see this in the following plot:

```
ggplot(sim2, aes(x)) +
  geom_point(aes(y = y)) +
  geom_point(data = grid, aes(y = pred), color = "red", size =
```



Note that if you try to predict the value where there are no observations, you will get an error message:

```
tibble(x = "e") %>%
  add_predictions(mod2)
```

```
## Error in model.frame.default(Terms, newdata, na.action = na.
```

25.3 Continuous and categorical

So what happens if you have both a continuous variable and a categorical variable in a model? The dataset `sim3` contains such a situation.

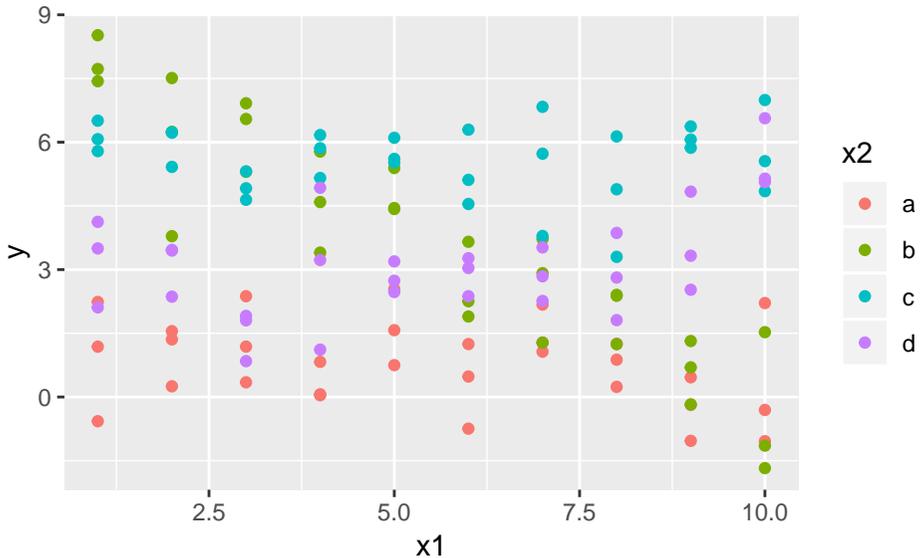
```
print(sim3, n = 5)
```

```
## # A tibble: 120 x 5
##   x1 x2     rep      y     sd
##   <int> <fct> <int> <dbl> <dbl>
## 1     1 a         1 -0.571     2
## 2     1 a         2  1.18     2
```

```
## 3      1 a          3  2.24      2
## 4      1 b          1  7.44      2
## 5      1 b          2  8.52      2
## # ... with 115 more rows
```

Let's visualize the data with a scatterplot colored based on the value in `x2`:

```
sim3 %>% ggplot() +
  geom_point(aes(x1, y, col = x2))
```



Here `x1` is numerical, and `x2` is categorical. Consider fitting a model both with and without interactions.

```
mod1 <- lm(y ~ x1 + x2, data = sim3)
mod2 <- lm(y ~ x1 * x2, data = sim3)
```

The `gather_predictions` function can be used to get predictions from both models simultaneously. We will first build a grid of `x1` and `x2` values, then bring the predictions from the models together.

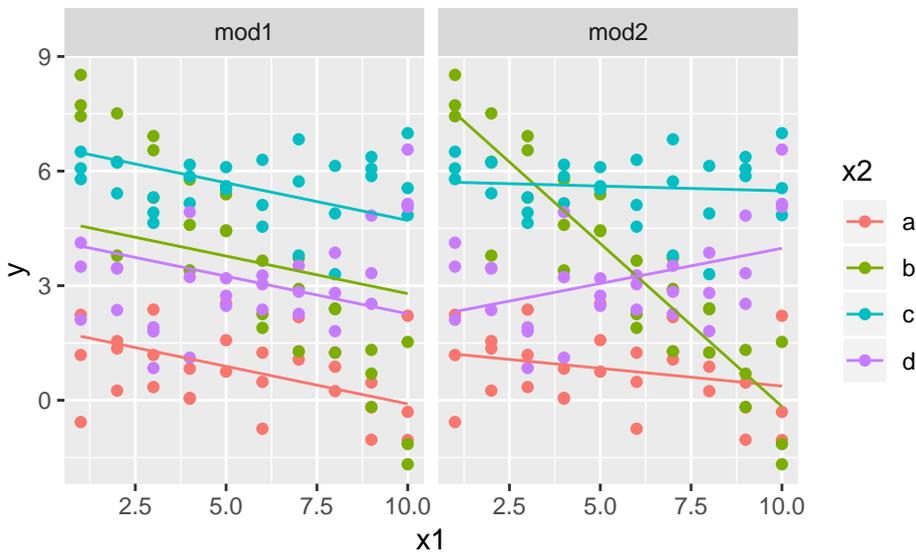
```
grid <- sim3 %>%
  data_grid(x1, x2) %>%
  gather_predictions(mod1, mod2)
print(grid, n = 5)
```

```
## # A tibble: 80 x 4
##   model    x1 x2      pred
```

```
##   <chr> <int> <fct> <dbl>
## 1 mod1     1 a      1.67
## 2 mod1     1 b      4.56
## 3 mod1     1 c      6.48
## 4 mod1     1 d      4.03
## 5 mod1     2 a      1.48
## # ... with 75 more rows
```

We can use two facets to look at these predictions for the two models.

```
ggplot(data = sim3, aes(x1, y, color = x2)) +
  geom_point() +
  geom_line(data = grid, aes(y = pred)) +
  facet_wrap(~ model)
```



Now we can see the effect of having an interaction versus not having an interaction. The slope of each line in the first model is the same: the x_2 variable effectively just changes the y -intercept.

On the other hand, when there is interaction between the x_1 and x_2 variable, the slope is allowed to change for each value of x_2 . This second model is clearly better, as the fact that the slopes change so dramatically to fit the data means that the slope should be changing with the value in x_2 .

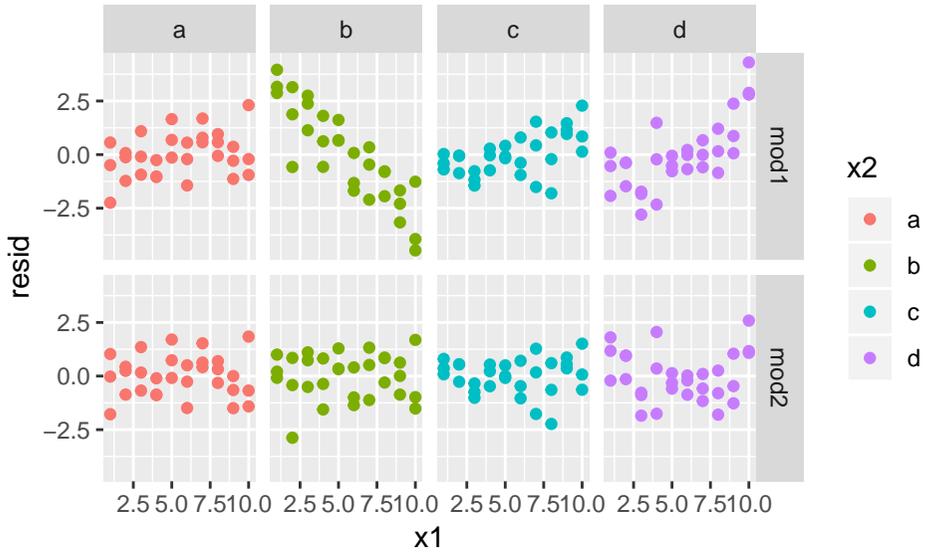
A look at the residuals bears this out. The following scatterplots use facets to break it down by level and model.

```

sim3 <- sim3 %>%
  gather_residuals(mod1, mod2)

ggplot(sim3, aes(x1, resid, col = x2)) +
  geom_point() +
  facet_grid(model ~ x2)

```



The second model (`mod2`) has residuals that look independent, while the residuals in `mod1` look like they have a strong pattern to them. Therefore one should use `mod2`.

Case study: predicting survival on the Titanic

Summary

When we are building a model, we often split the observations into a **training set** that we use to fit parameters, and a **test set** that we use to verify that the model is accurately predicting the response that we are after.

Sources

This chapter is based upon a blog post of Andrew Kinsman found at <https://www.kaggle.com/varimp/a-mostly-tidyverse-tour-of-the-titanic>.

26.1 Training data

A good model is useful in predicting the value of a variable given the value of the predictor variables. In order to test a particular model for data, one common method is to break data into a *training* set and a *testing* set.

Definition 74

Given a dataset, the **training set** is the subset of the data that is used to fit a model.

Definition 75

Given a dataset, the **testing set** is the subset of the data used to determine how good the model is at predicting observations that were not used to fit the model.

The training and testing set are disjoint observations.

Now, *Kaggle* is an online repository for data science that is owned by Google. It allows users to post data sets, and occasionally has competitions to see who can model data in the best way possible. In one particular competition, they challenged participants to predict survival on the Titanic. In 1912, the Titanic hit an iceberg and sank on its maiden voyage. Most of the passengers did not survive.

First let's download the training and test data for the problem.

```

library(tidyverse)
train <- read_csv("../datasets/titanic/train.csv")
test <- read_csv("../datasets/titanic/test.csv")
train

## # A tibble: 891 x 12
##   PassengerId Survived Pclass Name Sex Age SibSp
##   <dbl> <dbl> <dbl> <chr> <chr> <dbl> <dbl>
## 1 1 0 3 Brau~ male 22 1
## 2 2 1 1 Cumi~ fema~ 38 1
## 3 3 1 3 Heik~ fema~ 26 0
## 4 4 1 1 Futr~ fema~ 35 1
## 5 5 0 3 Alle~ male 35 0
## 6 6 0 3 Mora~ male NA 0
## 7 7 0 1 McCa~ male 54 0
## 8 8 0 3 Pals~ male 2 3
## 9 9 1 3 John~ fema~ 27 0
## 10 10 1 2 Nass~ fema~ 14 1
## # ... with 881 more rows, and 5 more variables:
## # Parch <dbl>, Ticket <chr>, Fare <dbl>,
## # Cabin <chr>, Embarked <chr>

```

The factor `Survived` is an indicator: it equals 1 if the person survived the sinking, and 0 if the person did not. Our goal is to predict with as high an accuracy as possible whether a particular passenger survived or not based on the values of the other variables.

The difference between the `test` data and the `train` data is that the `test` data does not contain the variable `Survived`. We can check that quickly with the `setdiff` function:

```
setdiff(names(train), names(test))
```

```
## [1] "Survived"
```

```
setdiff(names(test), names(train))
```

```
## character(0)
```

The first thing to do is to look at the `Survived` data. What percentage of passengers survived? For indicator data, we can get that by looking at the mean of the values.

```
train %>% summarize(SurvivalRate = mean(Survived) * 100)
```

```
## Warning: package 'bindrcpp' was built under R version
## 3.5.2

## # A tibble: 1 x 1
##   SurvivalRate
##         <dbl>
## 1           38.4
```

Only 38.3% of passengers in the training set survived. So our best prediction for whether or not a passenger survived if we knew nothing about that passenger is just to predict that the passenger did not. Let's try that with the test data.

```
baseline_solution <- test %>%
  select( PassengerId ) %>%
  mutate( Survived = 0 )
write_csv( baseline_solution,
           "../datasets/titanic/baseline_model.csv" )
```

At this point, one could submit this prediction back to Kaggle, and they would give it a score 0.62679, indicating that the choice of `Survived` was correct for 62.679% of the passengers in the test data set.

That is our baseline: any prediction, any model that we try to build has to do at least as well as the prediction that doesn't use any information whatsoever! Now that we know what we need to beat, let's bring the data together with a full join.

```
titanic <- full_join( train, test )
```

```
## Joining, by = c("PassengerId", "Pclass", "Name", "Sex", "Age")
```

One thing you might know about ship evacuations from that era is the phrase 'Women and children first'. So it seems natural to try to predict survival based on gender. First we will make sure that the `Sex` variable is actually a factor with levels.

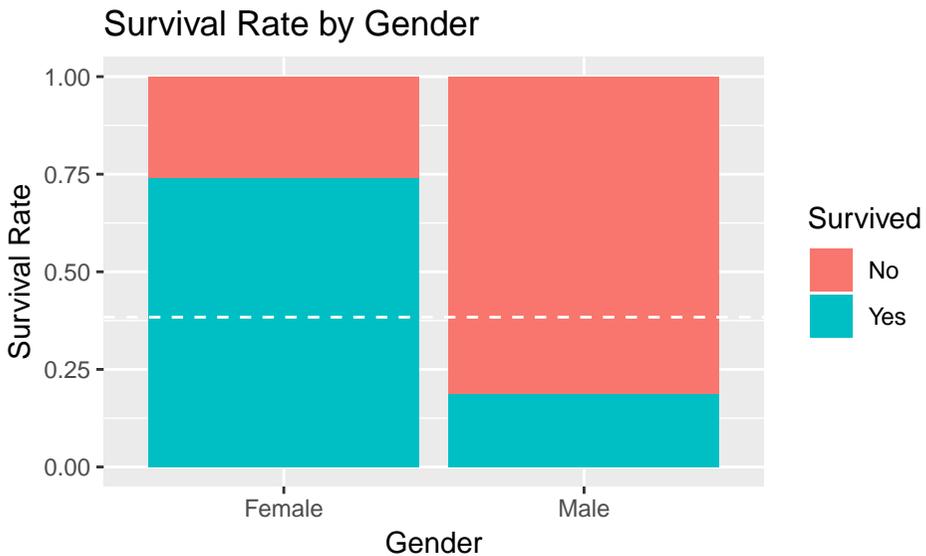
```
titanic <- titanic %>%
  mutate( Gender = fct_recode( Sex, "Female" = "female",
                              "Male" = "male" ) )
```

Next we treat `Survival` as categorical rather than numerical. Since it currently is not, first we turn it into a factor, then recode the levels.

```
titanic <- titanic %>%
  mutate(Survived = factor(Survived)) %>%
  mutate(Survived = fct_recode(Survived, "No" = "0",
                               "Yes" = "1"))
```

Now let's look at survival by gender:

```
titanic %>%
  filter(!is.na(Survived)) %>%
  ggplot() +
  geom_bar(aes(Gender, fill = Survived),
           position = "fill") +
  ylab("Survival Rate") +
  geom_hline(yintercept = mean(train$Survived),
            col = "white", lty = 2) +
  ggtitle("Survival Rate by Gender")
```



Wow, women really did survive more than men! Here's our new model: if you were a woman, you survived, if you were a man, you did not.

Next, let's take a look at the names of passengers.

```
train %>% select(Name) %>% print(n = 5)
```

```
## # A tibble: 891 x 1
##   Name
##   <chr>
```

```
## 1 Braund, Mr. Owen Harris
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer)
## 3 Heikkinen, Miss. Laina
## 4 Futrelle, Mrs. Jacques Heath (Lily May Peel)
## 5 Allen, Mr. William Henry
## # ... with 886 more rows
```

That is an interesting combination of data. We have the last name, the title, followed by the first name, followed by a middle name if they have one, followed by the maiden name of the person if applicable. Let's tidy this up a bit by pulling out the title of the passengers.

```
library(stringr)
titanic <- titanic %>%
  mutate(Title = str_extract(Name, "[a-zA-Z ]+")) %>%
  mutate(Title = str_replace(Title, " ", ""))
```

Let's count the number of times each title appears.

```
titanic %>%
  group_by(Title) %>%
  summarize(count = n()) %>%
  arrange(desc(count))
```

```
## # A tibble: 18 x 2
##   Title      count
##   <chr>      <int>
## 1 Mr          757
## 2 Miss        260
## 3 Mrs         197
## 4 Master       61
## 5 Dr           8
## 6 Rev          8
## 7 Col          4
## 8 Major        2
## 9 Mlle         2
## 10 Ms           2
## 11 Capt         1
## 12 Don          1
## 13 Dona         1
## 14 Jonkheer    1
## 15 Lady         1
## 16 Mme          1
```

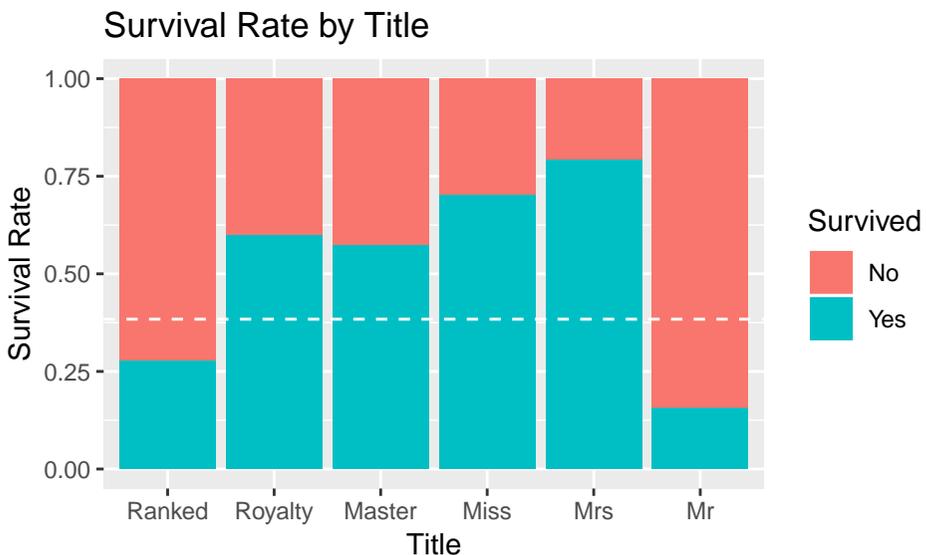
```
## 17 Sir 1
## 18 the Countess 1
```

There is a sharp dropoff in frequency from the first four titles to the remaining 14. By inspection, some language translation, and a little research, we can break down the titles into fewer categories using functions from `forcats`.

```
titanic <- titanic %>%
  mutate(Title = factor(Title)) %>%
  mutate(Title = fct_collapse(Title,
    "Miss" = c("Mlle", "Ms"),
    "Mrs" = "Mme",
    "Ranked" = c("Major", "Dr", "Capt", "Col", "Rev"),
    "Royalty" = c("Lady", "Dona", "the Countess", "Don",
      "Sir", "Jonkheer")))
```

Let's take a look at survival based on title:

```
titanic %>%
  filter(!is.na(Survived)) %>%
  ggplot(aes(x = Title, fill = Survived)) +
  geom_bar(position = "fill") +
  ylab("Survival Rate") +
  geom_hline(yintercept = 0.3838,
    col = "white", lty = 2) +
  ggtitle("Survival Rate by Title")
```



So Royalty (big surprise) those with Master in their title, and the titles associated with women were more likely to survive.

Missing Data

So far we haven't looked much at missing data. We can use `map_dbl` to locate the places in numerical data where there are missing values.

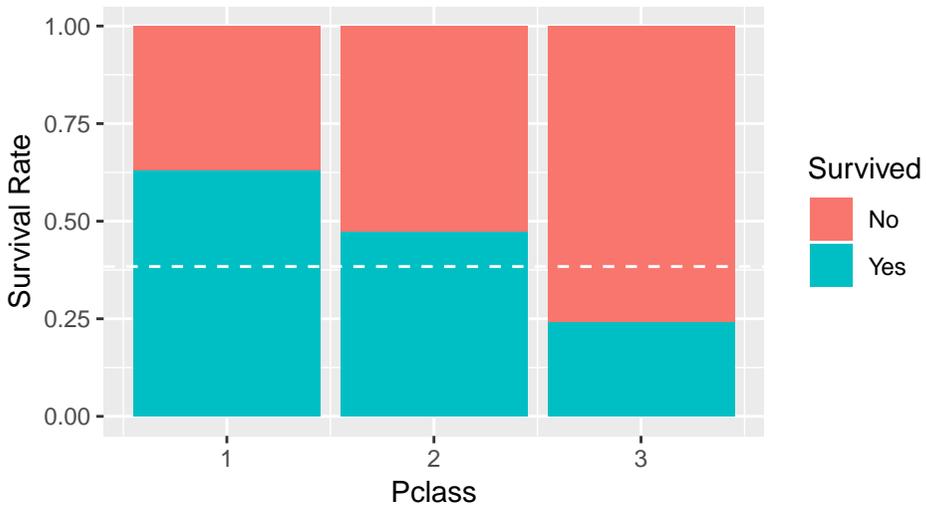
```
titanic %>% map_dbl(~sum(is.na(.)))
```

```
## PassengerId      Survived      Pclass      Name
##           0           418           0           0
##           Sex           Age           SibSp           Parch
##           0           263           0           0
##           Ticket       Fare           Cabin      Embarked
##           0             1           1014           2
##           Gender       Title
##           0             0
```

The 418 Survived missing data are of course coming from our test data. The main values missing among the remaining information are the Cabin and the age, with one Fare missing and two Embarked. With so many missing values, we should be wary of the Cabin number in our model. Especially since we have the passenger class (`Pclass`) data for all passengers. Speaking of which, let's see how the passengers fared by class of their ticket.

```
titanic %>%
  filter(!is.na(Survived)) %>%
  ggplot(aes(x = Pclass, fill = Survived)) +
  geom_bar(position = "fill") +
  ylab("Survival Rate") +
  geom_hline(yintercept = 0.3838, col = "white", lty = 2) +
  ggtitle("Survival Rates by Passenger Class")
```

Survival Rates by Passenger Class



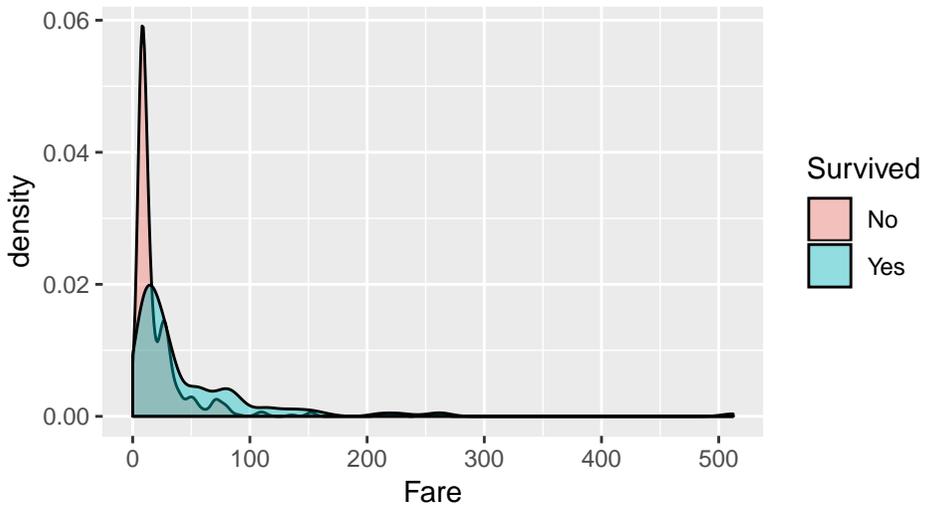
Not exactly a huge surprise there.

Fare

Let's look at survival versus fare. Now unlike the variables we've tried so far, fare is numerical. So instead of breaking down survival by color, we can use a kernel density plot to try to visualize what is going on.

```
titanic %>%
  filter(!is.na(Survived)) %>%
  ggplot(aes(x = Fare, fill = Survived)) +
  geom_density(alpha = 0.4) +
  ggtitle("Density Plot of Fare related to Survival")
```

Density Plot of Fare related to Survival



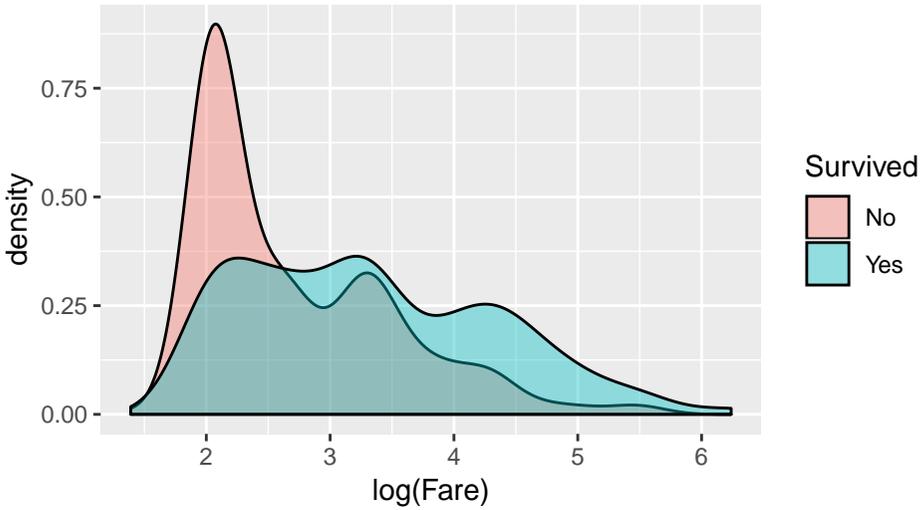
Notice that the Fare goes out far to the right. This is emblematic of *heavy-tailed data*. The price of a first class ticket can be orders of magnitude higher than a typical third class ticket. This type of behavior never shows up in light-tailed data such as the normal distribution.

One way to deal with this is to consider the logarithm of the Fare values. This will convert heavy-tailed data back to light-tailed data.

```
titanic %>%  
  filter(!is.na(Survived)) %>%  
  ggplot(aes(x = log(Fare), fill = Survived)) +  
    geom_density(alpha = 0.4) +  
    ggtitle("Density Plot of Fare related to Survival")
```

```
## Warning: Removed 15 rows containing non-finite values  
## (stat_density).
```

Density Plot of Fare related to Survival

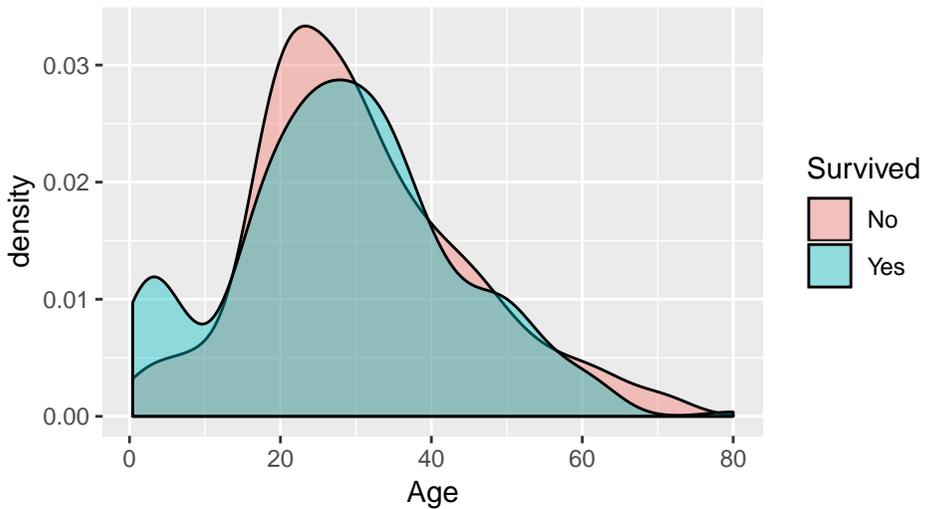


This echoes what we saw earlier with passenger class: travelers with cheap tickets were not first in the lifeboats. Were the old and young treated better?

```
titanic %>%  
  filter(!is.na(Survived)) %>%  
  ggplot(aes(x = Age, fill = Survived)) +  
    geom_density(alpha = 0.4) +  
    ggtitle("Density Plot of Age related to Survival")
```

```
## Warning: Removed 177 rows containing non-finite values  
## (stat_density).
```

Density Plot of Age related to Survival



Age is not a heavy tailed value. And while it looks like the young had a mild advantage in survival, great age did not confer much benefit.

26.2 Building a model

In order to build a model, we will use *conditional inference tree*. Any type of tree model works by partitioning the data space into pieces to better find those regions where survival is high. For instance, a simple partition might be by passenger class, or by gender. The tree uses non-parametric statistical tests in order to decide upon which factor to split the space next.

First let's turn variables that we think might be important into factors, and pull out our training data.

```
titanic <- titanic %>%
  mutate(Sex = factor(Sex))

train1 <- titanic %>%
  filter(!is.na(Survived))

train1 <- train1 %>%
  mutate(Survived = factor(Survived))
```

The choice of model created is random, and so we will set the seed at the beginning to ensure that the same random choices are made each call. As noted earlier, we will be using $\log(\text{Fare})$ rather than the Fare data directly. The function `cforest` for building this model is in the `party` package. Let's start with a simple model that predicts based on Fare.

```
set.seed(123456)
library(partykit)
cf_model1 <- cforest(Survived ~ Fare, data = train1)
```

To test this model, we will use the functions from the `modelr` package.

```
library(modelr)
```

Now we can use `add_predictions` to make predictions

```
train1_pred <- train1 %>%
  add_predictions(cf_model1)
```

```
## Warning in model.frame.default(object$predictf, data =
## newdata, na.action = na.pass, : variable 'Survived' is
## not a factor
```

```
train1_pred %>%
  mutate(right = (Survived == pred)) %>%
  summarize(mean(right))
```

```
## # A tibble: 1 x 1
##   `mean(right)`
##           <dbl>
## 1           0.749
```

So just by using Age, we have beaten our baseline and improved to 74.85% (at least on the training data.) Can we do better? Let's put a few more predictors in our model.

```
cf_model2 <- cforest(
  Survived ~ Sex + Age + Fare + Pclass + Title,
  data = train1
)
```

Now we add those predictions:

```
train1_pred2 <- train1 %>%
  add_predictions(cf_model2)
```

```
## Warning in model.frame.default(object$predictf, data =
## newdata, na.action = na.pass, : variable 'Survived' is
## not a factor
```

```
train1_pred2
```

```
## # A tibble: 891 x 15
##   PassengerId Survived Pclass Name Sex Age SibSp
##   <dbl> <fct> <dbl> <chr> <fct> <dbl> <dbl>
## 1 1 No 3 Brau~ male 22 1
## 2 2 Yes 1 Cumi~ fema~ 38 1
## 3 3 Yes 3 Heik~ fema~ 26 0
## 4 4 Yes 1 Futr~ fema~ 35 1
## 5 5 No 3 Alle~ male 35 0
## 6 6 No 3 Mora~ male NA 0
## 7 7 No 1 McCa~ male 54 0
## 8 8 No 3 Pals~ male 2 3
## 9 9 Yes 3 John~ fema~ 27 0
## 10 10 Yes 2 Nass~ fema~ 14 1
## # ... with 881 more rows, and 8 more variables:
## #   Parch <dbl>, Ticket <chr>, Fare <dbl>,
## #   Cabin <chr>, Embarked <chr>, Gender <fct>,
## #   Title <fct>, pred <fct>
```

Then see how well we did:

```
train1_pred2 %>%
  mutate(right = (Survived == pred)) %>%
  select(PassengerId, Survived, pred, right) %>%
  summarize(mean(right))

## # A tibble: 1 x 1
##   `mean(right)`
##   <dbl>
## 1 0.860
```

Woohoo! We're up to 86.30%, at least on our training set. Of course, the model parameters were fitted to the training set, which means that it is unlikely to do as well when applied to the test data set. Let's try it on our test set. First let's draw back out our test set.

```
test1 <- titanic %>%
  filter(is.na(Survived))
test1

## # A tibble: 418 x 14
##   PassengerId Survived Pclass Name Sex Age SibSp
```

```
##           <dbl> <fct>           <dbl> <chr> <fct> <dbl> <dbl>
##  1             892 <NA>             3 Kell~ male   34.5     0
##  2             893 <NA>             3 Wilk~ fema~  47       1
##  3             894 <NA>             2 Myle~ male   62       0
##  4             895 <NA>             3 Wirz~ male   27       0
##  5             896 <NA>             3 Hirv~ fema~  22       1
##  6             897 <NA>             3 Sven~ male   14       0
##  7             898 <NA>             3 Conn~ fema~  30       0
##  8             899 <NA>             2 Cald~ male   26       1
##  9             900 <NA>             3 Abra~ fema~  18       0
## 10            901 <NA>             3 Davi~ male   21       2
## # ... with 408 more rows, and 7 more variables:
## #   Parch <dbl>, Ticket <chr>, Fare <dbl>,
## #   Cabin <chr>, Embarked <chr>, Gender <fct>,
## #   Title <fct>
```

Now we make our predictions.

```
test1_pred2 <- test1 %>%
  select(-Survived) %>%
  add_predictions(cf_model2)
```

```
## Warning in model.frame.default(object$predictf, data =
## newdata, na.action = na.pass, : variable 'Survived' is
## not a factor
```

```
test1_pred2
```

```
## # A tibble: 418 x 14
##   PassengerId Pclass Name      Sex      Age SibSp Parch
##         <dbl> <dbl> <chr> <fct> <dbl> <dbl> <dbl>
##  1             892     3 Kell~ male   34.5     0     0
##  2             893     3 Wilk~ fema~  47     1     0
##  3             894     2 Myle~ male   62     0     0
##  4             895     3 Wirz~ male   27     0     0
##  5             896     3 Hirv~ fema~  22     1     1
##  6             897     3 Sven~ male   14     0     0
##  7             898     3 Conn~ fema~  30     0     0
##  8             899     2 Cald~ male   26     1     1
##  9             900     3 Abra~ fema~  18     0     0
## 10            901     3 Davi~ male   21     2     0
## # ... with 408 more rows, and 7 more variables:
```

```
## # Ticket <chr>, Fare <dbl>, Cabin <chr>,
## # Embarked <chr>, Gender <fct>, Title <fct>,
## # pred <fct>
```

```
cf_model2_solution <- test1_pred2 %>%
  select( PassengerId, Survived = pred ) %>%
  mutate( Survived = fct_recode( Survived, "1" = "Yes",
                                "0" = "No" ) )

cf_model2_solution
```

```
## # A tibble: 418 x 2
##   PassengerId Survived
##   <dbl> <fct>
## 1         892 0
## 2         893 0
## 3         894 0
## 4         895 0
## 5         896 1
## 6         897 0
## 7         898 1
## 8         899 0
## 9         900 1
## 10        901 0
## # ... with 408 more rows
```

Now write it out to a file:

```
write_csv( cf_model2_solution,
           "../datasets/titanic/cf_model2.csv" )
```

When submitted to Kaggle, this returns a score of 77.511%.

Further thoughts

When you are constructing a model, there are two primary goals you are looking for.

1. Accuracy of predictions (small residuals).
2. Simplicity of the model.

These are usually at cross purposes. The more complicated the model, the closer you can fit your data in your training set. However, this can lead to *overfitting*, when you have too many parameters in your models. These types of models tend to be brittle, in the sense that

the model, when faced with data outside of the carefully modeled observations, completely falls apart.

How to know when you've overfitted? There are statistical tests that you can use, but essentially it boils down to experience and an awareness that you should always keep in mind: make a model as sophisticated as it needs to be to capture the features of the data you are interested in, and not more complex than that.

Machine learning

Summary

Machine learning is the term for algorithms that learn from the data how to build a model.

Modeling

| | |
|------------|---|
| lm | Create a linear model. |
| rlm | Creates a robust linear model less sensitive to outliers. |
| svm | Creates a support vector machine model. |
| glm | With <code>family = "binomial"</code> , does logistic regression. |

Introduction

So far we have been using a traditional approach to modeling and predicting. Visualize and try to understand the data, and then build an explicit model that picks out the factors that are most important in getting the response.

Machine learning is an alternate approach to this traditional procedure.

Definition 76

Machine learning is the area of computer science that deals with algorithms designed to learn from datasets how to accomplish various tasks.

A good machine learning algorithm will improve its results as more data is fed into the system. We often say that the algorithms *gain experience* as they *learn* from the data. There are still models in machine learning, but they are designed to be much more flexible than the classic linear models. Often machine learning algorithms go beyond just fitting parameters to deciding the more basic question of which factors serve as the best predictor variables in the first place.

The four main tasks of machine learning are the following.

1. *Classification*. In this case, our response is a categorical variable with a finite set of possibilities. We want to know which possible outcome is the best class for our observation.

2. *Prediction.* This applies when our response is numerical. Here we are usually trying to minimize some measure of how far away our prediction is from the true answer.
3. *Density estimation.* Often our observations do not have factors spread over all possible values, but instead concentrate in a particular area. The goal here is to understand where the density of input values is highest.
4. *Pattern recognition.* Our observations are in general very high dimensional. For instance, a photograph might have 16 million pixels. A 16 million dimensional model is too much to handle! Instead, we look for lower dimensional behavior within the set of model. This allows us to project the data onto a much lower dimensional data set.

Remember in its most basic form, a model is a mathematical function:

$$(y_1, y_2, \dots, y_k) = f(x_1, x_2, \dots, x_p) + \epsilon.$$

In machine learning, the function f is created partially by looking at the data itself.

There are two types of machine learning.

Definition 77

In **supervised learning** we have a **training set** that has labeled data. For each set of possible predictors, the output is known in the training set.

Definition 78

In **unsupervised learning** there is no labeled dataset. The goal is learn about the data solely from the data values themselves.

Some common methods for supervised learning include:

- Decision Trees and Random Forests
- Linear Regression
- Logistic Regression
- Boosting
- Support Vector Machines
- Bayesian Classifiers/Bayesian Networks
- Neural Networks
- Deep learning

Some common methods for unsupervised learning include:

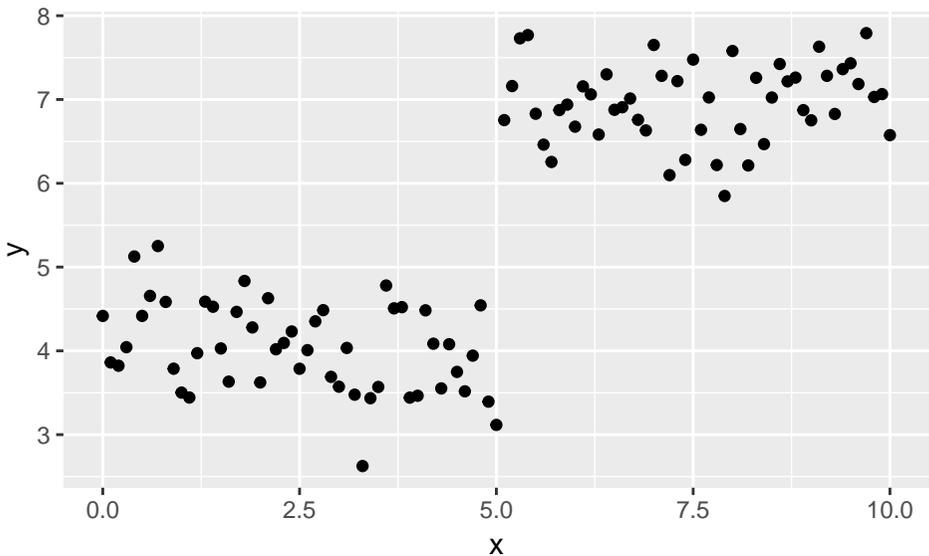
- Clustering
- Anomaly detection
- Topic modeling
- Neural Networks

27.1 Supervised learning

Let's break down some of these methods into more detail.

Decision Trees and Random Forests

In this method, the goal is to split the state space of inputs into two equal parts, where the response variable is as similar as possible to each other over the space.



In the data above, there is a clear break in the y values for $x < 5$ and $x \geq 5$. On either side of this split, the y values are much closer to one another.

Definition 79

Each node of a **decision tree** breaks the input space into two pieces where the response is closer to its center in each of the two trees than in the overall space.

In the toy example above, the decision tree idea works very well. Unfortunately, in real data it can be prone to overfitting. To solve this problem, a *random forest* works by performing the decision tree process multiple times. Each time a tree is created, a bit of randomness is intentionally injected into the tree process.

The set of trees together is called a random forest. When a user then inputs a new observation and wants a prediction of the response, each tree is run separately. The final

result can then be found by looking at the level most commonly seen for categorical response, or the average of the predictions from each tree for a numerical response.

Definition 8o

A **random forest** is a collection of decision trees where at each step in their formation, some random choices were made.

It is well known that a survey of a group of individuals can sometimes perform better than a single expert on a topic. Each individual member of the group has less information about the subject, but some are biased high and some are biased low as to the true answer. By averaging their information, the result is often better than a single person whose biases are unknown. This phenomenon sometimes goes by the name *wisdom of the crowd*.

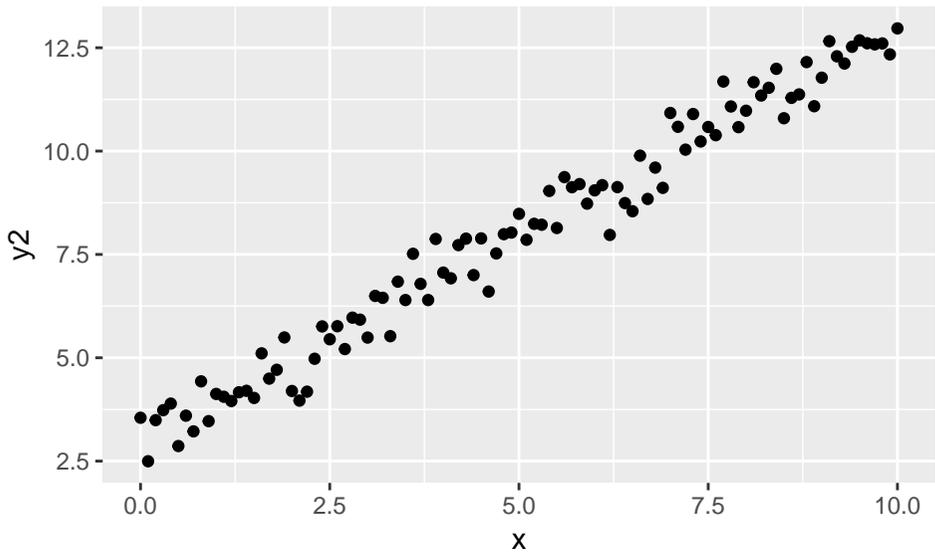
In the same way, each individual decision tree in the random forest might be biased towards some response or another based on how it was created. However, since the process has been done randomly multiple times, some trees will be biased in a certain way while some trees will be biased in the opposite way. Together their average is close to the true value.

Linear regression

The nice thing about random forests is that we need to know very little about the structure of the data in order to make accurate predictions. If we do no more about the structure of the data, then a linear regression model might be in order.

For instance, consider the following data set.

```
y2 <- 3 + x + rnorm(length(x), 0, 0.5)
tibble(x, y2) %>%
  ggplot(aes(x, y2)) +
  geom_point()
```



A decision tree would have to break this down into many pieces to get an accurate read, while a simple linear model does better with only a slope parameter and y -intercept parameters.

Definition 81

Suppose we have n observations and p predictor variables. Call the column vector of the response variable values Y . Then form a model matrix whose $i, (j + 1)$ th entry is the value of the j th predictor variable in the i th observation, and whose $i, 1$ entry is always 1. Then the model

$$Y = X\beta + \epsilon,$$

where Y is an n by 1 matrix, X is an n by $p + 1$ matrix, β is a $p + 1$ by 1 matrix, and ϵ is an n by 1 matrix. This is called a **linear model** of the data, and β are called the **parameters** of the model.

One reason that linear models are widely used is that if our goal is the minimize the sum of the squares of the ϵ , it is possible to solve for the β values exactly given Y and X . If there are many observations and predictors, these computations can still take a long time, however, it is still possible to approximate the values of β that is the best fit.

Logistic regression

Linear models do better with numerical data and decision trees do better with categorical data. Is there a way to use linear models for categorical data? Consider the best least squares fit for the following data where the response is either 0 or 1.

```
library(modelr)
y3 <- (runif(length(x)) < (.3 * (x < 5) + 0.7 * (x > 5))) + 0
```

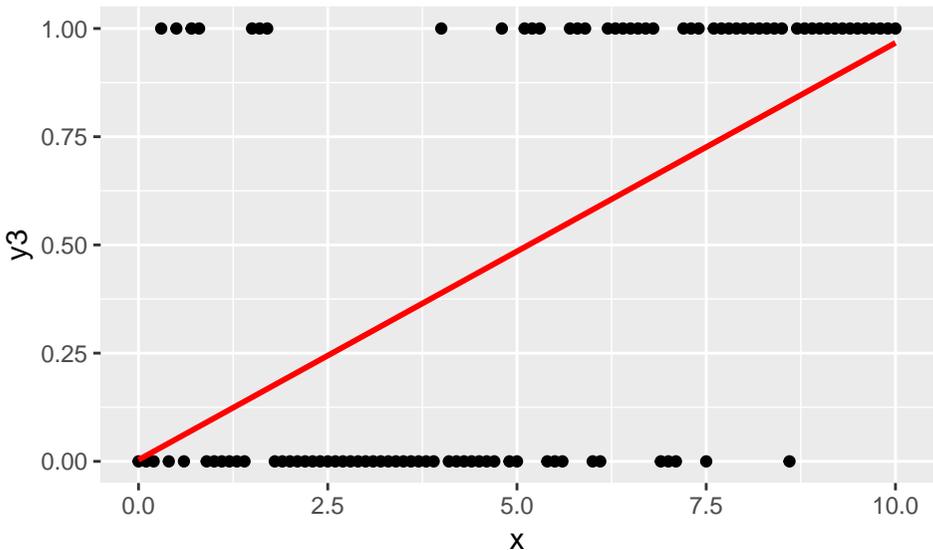
```

mod1 <- lm(y3 ~ x)
df3 <- tibble(x, y3)

df3_pred <- df3 %>%
  add_predictions(mod1)

tibble(x, y3) %>%
  ggplot(aes(x, y3)) +
  geom_point() +
  geom_line(data = df3_pred, aes(x, pred), color = "red",
           lwd = 1)

```



Instead of directly modeling the data using a linear model, we let p be the probability that the data is 1, $1 - p$ be the probability that it is 0.

Definition 82

If the probability of one outcome is p , and the probability of another outcome is $1 - p$, then the **logit function** is the logarithm of the odds of the first outcome to the second outcome. That is,

$$\text{logit}(p) = \log\left(\frac{p}{1-p}\right).$$

Note that $\text{logit}(p)$ can be any positive or negative real number. The idea of logistic regression is to model $\text{logit}(p)$ as

$$\text{logit}(p) = \log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p.$$

In R, we can create such a model with the `glm`, generalized linear models function. First we make sure that R knows that we are dealing with a categorical variable by making the data a factor.

```
df4 <- df3 %>%
  mutate(y = factor(y3))
```

```
## Warning: package 'bindrcpp' was built under R version 3.5.2
```

```
mod_logit <- glm(y ~ x, data = df4, family = "binomial")
```

Now we create the new model

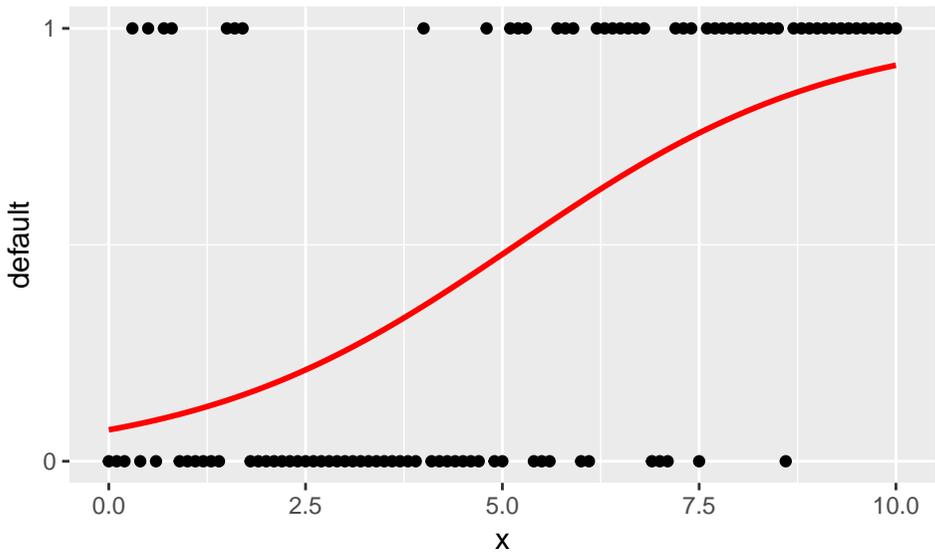
```
df4_pred <- df3 %>%
  data_grid(x) %>%
  add_predictions(mod_logit, type = "response")
# mutate(pred = predict(mod_logit, newdata = ., type = 'response'))

df4_pred
```

```
## # A tibble: 101 x 2
##       x     pred
##   <dbl> <dbl>
## 1     0  0.0725
## 2    0.1 0.0759
## 3    0.2 0.0794
## 4    0.3 0.0831
## 5    0.4 0.0869
## 6    0.5 0.0909
## 7    0.6 0.0950
## 8    0.7 0.0993
## 9    0.8 0.104
## 10   0.9 0.109
## # ... with 91 more rows
```

Graphing this is a bit tricky. We use `as.numeric` to convert the labels 0 and 1 to numbers. The numbers will be 1 and 2 (if we had three levels the numbers would be 1, 2, and 3.) So we subtract 1 to get them back to 0 and 1.

```
ggplot(data = df4, aes(x)) +
  geom_point(aes(y = as.numeric(y) - 1)) +
  geom_line(data = df4_pred, aes(x, pred), color = "red", lwd = 2) +
  scale_y_continuous('default', breaks = 0:1)
```



A simple prediction method is then: if $p \geq 0.5$ predict 1, otherwise predict 0. Note that this is very close to what the decision tree would do for this data set.

Other methods

There are many other approaches as well to supervised learning.

Boosting

Boosting tries to improve methods such as logistic regression by repeating them over multiple levels of prediction. We begin with some weak classifier that does not overfit. Logistic regression is often used for this.

At this point, if we just look at the predictions for our classifier on our training data set, we have made mistakes. So build a second classifier that gives more weight to the observations where we were mistaken. This gives us a second classifier.

We can repeat this process to obtain a family of classifiers. Now, just as in the random forests, we can run each of these classifiers to obtain a family of predictions. Then go with the prediction that is most popular among the family.

Support Vector Machines

A support vector machine classifies by trying to split the data into two groups using a hyperplane. In some cases, this is very easy, and the hyperplane can be used directly.

In other cases, the groups are separated by a curve. In order to deal with data like this, we need to develop a *feature*, a function of the predictors that gives a new predictor. Then we apply the hyperplane to this new feature.

Definition 83

A **feature** is a new predictor whose value is a function of other predictors.

Bayesian Classifiers/Bayesian Networks

Suppose we know given which class we are in, the probability of certain predictor values appearing. Then *Bayes Rule* allows us to turn this around: given that we say certain predictor values, what is the chance that we are in a particular class.

These methods are in some sense the gold standard of classification. However, applying Bayes' Rule to complex models is computationally very expensive.

So often instead of a general model we make simplifying assumptions. The most basic assumption is that the predictors are conditionally independent given the class of the observation. That is, once we know which class we are in, all the predictors are independent of each other! Although this is a very powerful assumption, it can actually give models that are very useful for making predictions.

Neural Networks

A neural network tries to understand how the input and output variables of a model connect through a *graph*. In mathematics, a graph consists of *nodes* (also called vertices) that are connected by *edges*. Inputs to nodes become outputs to other nodes through the weights on the edges.

This process was inspired by biological neurons, which fire to other neurons when they are excited. Although mathematical neural networks are somewhat different from this process, the name has stuck, which is why they are called neural networks.

By using the data in observations, the weights on the edges are fine-tuned to make the final output of the graph equal to the input.

Deep Learning

Neural networks turned out to be difficult to use in complex situations. However, by first broadening the classifications to larger groups, a neural network works quite well. Then these broader classes can be refined. Do this three or four times, and we can get back to the original classes that we had in mind.

This is the basic idea behind *deep learning*. Use a nested set of neural networks (or other models) to slowly move from the input space to the output space. This idea has proved especially effective in areas such as computer vision, speech recognition, and natural language processing where the data naturally lends itself to large groups, then smaller more refined groups.

27.2 Unsupervised learning

In unsupervised learning, we are not given any labeled data. It is up to the algorithm to construct the classes that the observations fall into.

Clustering

An example of this type of learning is *cluster analysis* or more simply, *clustering*. Here the goal is to determine which observations in the sample space are close together to one another.

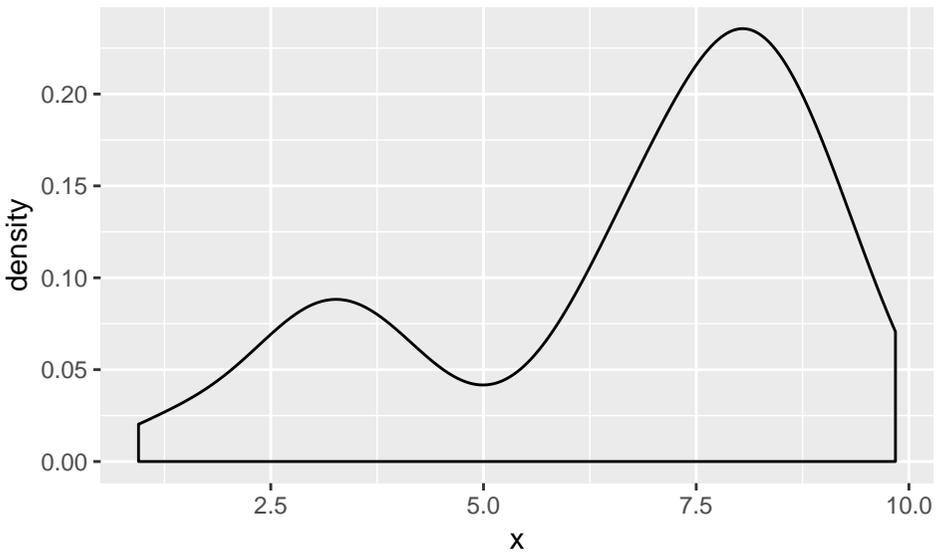
As with most models, there is no one “right clustering”. Instead, different clustering algorithms will achieve different results. In the end, the question is whether or not the clustering is useful for the purposes that the person analyzing the data is trying to achieve.

For example, a simple way to cluster is to calculate for observations which are numerical n -tuples the distance between each pair of points. Then for a given threshold value, connect any pairs of points whose distance is below that threshold. The number of clusters equals the number of points when the threshold is 0, and decreases as the threshold is raised.

Another type of clustering is based upon *kernel density estimation*. Here a normal density is placed on top of each point in the data set and then added up. For instance, suppose we have some x values drawn using a normal distribution centered at 3, and others draw using a normal distribution centered at 8. Then the kernel density plot might look as follows.

```
x <- c(rnorm(50) + 3, rnorm(150) + 8)
```

```
ggplot() +  
  geom_density(aes(x))
```



This kernel density plot has two local maxima, which separated the data points into two clusters. More generally this idea is known as *density-based clustering*.

Part V

EXPLORATIONS

Chapter 28

Exploration: introduction to R

Summary

This lab is an introduction to the R programming environment using RStudio, with a goal of learning about how to submit commands, write scripts, and create documents using R Markdown. In this lab, you will learn about

- The console: where you can directly enter commands into the R programming environment.
- Scripts: a list of instructions that you can give R.
- R Markdown: a light markup language that you can use to include R code in your typeset documents.
- \LaTeX : how to add mathematics to your R Markdown document using \LaTeX .
- Packages and libraries: how to expand what R can do through packages and libraries.
- pdf output: how to have R Markdown create a .pdf file instead of HTML output.
- notebooks: how to use your R Markdown file as a notebook
- functions: how to write basic functions in R Markdown

Instructions

This lab is a tutorial for using the R programming environment. It does not assume any prior knowledge about R. You are welcome to ask questions of myself or your neighbors during the lab. When you are done, please answer the questions at the end of the lab and turn it in. If you have not finished the lab by the end of the period, please complete the lab outside of the time and turn it in the next class period.

The console

- Begin by starting RStudio on either your desktop computer or laptop if you brought it. If you have not yet installed R on your laptop, you can obtain R from <https://www.r-project.org/> and RStudio from <https://www.rstudio.com/>.
- Once you start RStudio, you will see various windows called **panes**. One of these panes will have a tab marked **Console**. By default it appears at the left pane or lower left pane. Type the following commands into the console and hit return.

```
x <- 8
y <- -6
x + y
```

- The `<-` that appears in the commands is the **assignment operator** in R. The command `x <- 8` tells R that moving forward, we want to assign the value of 8 to `x`. In the upper right pane there is a tab labeled **Environment**. In this pane there should be a part labeled Values that now tells us that `x` is 8.
- The second command assigned the value -6 to `y`. The last command then added the two numbers together, and printed the result to the console. The output should look like

```
[1] 2
```

- The `[1]` part indicates that the result is a vector, and on this line the first component of the vector is given. Of course, this result is a number, that is, a vector of length 1. So there only is the first component! To see what happens with a result that is a vector with more components, try

```
seq(100, 1, by=-1)
```

which generates a length 100 sequence of numbers starting at 100, going down to 1 and changing by -1 at each step.

- Any command, function or variable built in to R has help information that can be accessed by putting a question mark in front of the command. Try

```
?seq
```

to get to the help for the `seq` function.

- The `seq` command creates a variable that is a vector. Often we just want a vector like `(1, 2, 3, 4, 5)`. A shorthand for the command `seq(a, b, by=1)` is `1:5`. Try

```
1:100
```

- You can also manually create your own vectors. Try

```
x <- c(6, 2, 3)
```

to get a vector of length 3. Most calculator commands such as + (addition), - (subtraction), * (multiplication), / (division), ^ (raising to a power) operate on each component at a time. Try

```
x <- c(-1, 2, 3)
2 * x
x^2
x^(1 / 2)
```

- Note that the first result for the last command was NaN which stands for *Not a Number*. This is the result since the square root of -1 is not a real number.
- The most often used statistical operations on a vector are `length` (how many numbers in the vector are there), `sum` (adds the numbers), `mean` (adds the numbers then divides by the length of the vector), and `sd` (find the sample standard deviation of the numbers). Try

```
mean(x)
sd(x)
```

to get the mean and sample standard deviation of $(-1, 2, 3)$.

Questions

1. What command generates a sequence from 2 to 50 changing by 2 at each step?
2. What commands would assign the value 7 to `z`, 8 to `w`, and then print their product?
3. What command would find the square of the numbers $(-1, 4, 2)$?

Scripts

A *program* or a *script* is a list of instructions given to a computer. Originally, the term program was reserved for programming languages such as FORTRAN, while the script was reserved for commands given to an operating system such as Unix.

Over time, the distinction has become muddled, although there exist folks who feel very strongly about what set apart a script from a program. R is considered a programming environment, but a set of commands given to R is usually called a script.

By using scripts to perform your work, you keep an exact record of what you did. This is very helpful when it comes time to report your results, for collaborations or to share your work with others. Moreover, it is much easier to check a script and correct any errors than to try to do things perfectly through the console.

- To open your first script in R, use the menu option

File → New File → R Script

- This will open up a pane in the upper left corner (by default) in your RStudio window labeled *Untitled1*. Try typing the following commands in the pane:

```
a <- 5
b <- 1
a + b
```

- Now these commands are ready to go, all they need is your go ahead to run. This can be accomplished by using the `source` command in R. However, with RStudio there is another way. Below the tab reading *Untitled1*, there should be a checkbox labeled *Source on Save*. Check this box. Now, whenever you save your script, the commands will automatically be sourced to the console.
- Give it a try. Check the *Source on Save* checkbox, and then use

File → Save

(or Ctrl-S on Windows machines) to save the file. It will ask for a name, use `script1`. In the console you will see a command similar to `source("~/script1.R")` appear. That indicates that the script has been sourced to the console. (Note that RStudio has automatically added the standard `.R` extension to your script name. This is the common practice for R script files.)

- Now check the upper right pane: it looks like the variables `a` and `b` have been assigned properly, but the value of `a+b` was not printed in the console. That is because when commands like this are run as a script, they do not automatically generate output. Try changing the last line to

```
print (a+b)
```

and source your script again. Now it should return `[1] 6` when sourced.

Questions

- Write a script that assign the numbers 1 to 10 to `x`, and then assigns the square of each number in `x` to `y`

R Markdown

Scripts allow you to precisely record your analyses, R Markdown allows you to create professional looking reports that include the use of R code.

In general, a **markup language** allows you to use a text file to create a document that will be typeset by an appropriate application. For instance, \LaTeX is an extremely powerful markup language known for its ability in typesetting mathematics, and hypertext markup language (HTML) is the standard for typesetting webpages.

Markdown is a markup language that (as the down part of the name implies) was intended to be extremely simple to use. **R Markdown** is an implementation of markdown that is designed specifically for working with the R programming environment.

- We can get started with R Markdown in a fashion similar to when we wrote our first script. Use

File → New File → R Markdown

to open a new R Markdown file. The first thing is a window will pop that asks for a title, and asks you to choose one of three publishing options, HTML, pdf, or Word. In fact, both these options can be changed later, so don't feel that you are locking yourself in by choosing at this stage. For now, title your new document **Lab 1** and use HTML.

- In the newly created pane (entitled *Untitled 1*), the first six lines are put in by default using a form which is called a **serialization language**. The difference between this and a markup language is that it is the syntax of the lines that determines the result. You cannot just (for instance) start a new section in a serialization language. For instance, you can see a line that begins `title:` and `author:`.
- This particular format is called YAML, which stands for *YAML Ain't Markup Language*. This is a perfect illustration of a *recursive acronym* because it contains itself in its own abbreviation. The YAML heading is followed by three strange lines of code. These set certain defaults for how code will be displayed in your document, and you don't have to worry about them for now.

- Next you will see a line that reads

```
## R Markdown
```

The two `##` symbols (read as number sign or hashtag) indicate that R Markdown is a section heading. Since there are two `#` symbols, it is a level 2 header. To see how this works in practice, try pressing the Knit button that appears in the pane right below the list of files that are open. First RStudio will ask you to save your file, use **markdown1** for now. RStudio will automatically add the standard `.Rmd` file extension to whatever name you give it. Next, RStudio will knit together your file and open it in a new window.

- This is an HTML document that you can print to pdf or use on a webpage or send to a collaborator.
- One of the next things you'll see is something called a **code chunk**. Such a chunk starts with `"`, ends with `"`, and the command in between (`summary(cars)`). In the HTML generated by knit, you see the command against a grey box, and then the consequences of the command in a white box.
- Try changing the `{r cars}` part of the line to `{r cars, echo=FALSE}` and reknit the document. You can see that it still shows the output generated by the command, but does not copy the command itself into the HTML document. Now change it to `{r cars, results="hide"}`. You can see that now we see the code itself, but not the results of the code.
- If you change it to `{r card, include=FALSE}`, then neither the code nor the results of the code will appear, but the code will still be evaluated.
- The next code chunk in the default R Markdown file is a plotting command. If you do not like the default size of the plot generated, the `fig.width` and `fig.height` commands come in handy. Try altering the code chunk options to read `{r pressure, echo=FALSE, fig.height=4, fig.width=8}` and see what happens to the size of the figure created.

Mathematics and \LaTeX

Mathematics can be added to an R Markdown format using \LaTeX .

- There are two types of mathematics that you can create. The first is called **inline mathematics** and to create it, you surround the math with dollar signs. Try adding

```
This is inline math $a + b$.
```

anywhere in the `markdown1` document. Knit together to see the result.

- The other type of mathematics is **display style**. To create this type of math, we do something like

```
\[
a + b.
\]
```

Try adding this line to the `markdown` document and `knit` to see the result.

- Most of the symbols and notation of mathematics can be created in \LaTeX by using the backslash, `\` followed by the command. For instance, less than or equal to is `\leq`, so

```
$a \leq b$
```

creates $a \leq b$. Most of the Greek letters are in \LaTeX , and can be modified in various ways. So

```
$$\bar{\mu}$$
```

yields $\bar{\mu}$ for instance.

Libraries

Libraries are a way to expand what the R programming environment can do by adding in new functions, commands, and variables. For instance, we can add the ability to knit together your document from the console. Using libraries are a two step process:

1. First make sure that the library is part of the software installation of R. This can be accomplished using the command `install.packages("nameofpackage")`. This only needs to be done once for your R installation. Once you've installed a package, it can be used on your computer until you uninstall R.
 2. Second, each time you start the program R, you need to load the package into R. This can be done with the command `library(nameofpackage)`. Note that in the `install.packages` command we used quotes, and in this command we do not.
- Let's practice this by using the library `rmarkdown`, which will add the ability to render from the console.

```
install.packages("rmarkdown")
```

If the package is installed already, this will try to update the package to the latest version.

- To begin, type the command

```
library(rmarkdown)
```

into the console. This will load the package `rmarkdown` into the R programming environment so that we can use it moving forward.

- At this point, you can use the `render` command to knit the document together:

```
render("~/markdown1.Rmd")
```

pdf output

Up until now, we have been working with HTML, which is great for websites, not so great for reports, articles, and books. These types of documents typically are broken up into pages, while HTML tends to create one long document.

A different markup language is \LaTeX , and once its compiler is installed, we can use R Markdown to create `.pdf` files with pages.

- In order for R Markdown to knit to a `.pdf` file, we need to have a version of \TeX installed. In R, the easiest way to do this is to use `tinytex`, but the installation procedure does not always work. Here's what to do. First, use the commands

```
install.packages("tinytex")  
library(tinytex)  
is_tinytex()
```

If the answer is `FALSE`, then we can install `tinytex` with

```
install_tinytex()
```

- At this point, hopefully you have a working \TeX installation that R Markdown can connect with. Otherwise, just skip this section of the Lab.
- Now we need to tell R Markdown to make a `.pdf` rather than an HTML document. In the YAML heading, change the last line to

```
output: pdf_document
```

- If you knit the document now, it will tell the application **pandoc** to create an HTML file. It is possible that you will need to have RStudio load a package when creating this file. R Markdown utilizes the `pdflatex` compiler to do this. Therefore, once you change the output to `pdf` format, if you are familiar with \LaTeX commands, then you can add any such code and it should compile. For instance, try adding

```
\begin{center}
This is a test of pdf document mode.
\end{center}
```

and see what happens when you compile.

Questions - R

5. See if you can recreate the following pdf document:

A short report on the Nile

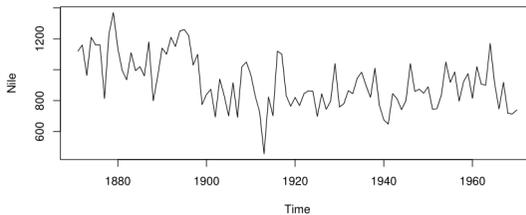
John Smith

January 24, 2019

The River Nile

Consider the annual flow of the river Nile as Aswen from 1871-1970, as measured in hundreds of millions of meters cubed.

```
plot(Nile)
```



If we let \bar{x} denote the average flow, then

$$\bar{x} = 919.35$$

Notebooks

R Markdown files also operate as a **notebook**. A notebook is a list of commands that include code chunks that can be evaluated individually to see the results of just that part of the file.

- Go back to your `markdown1.Rmd` file. If you press the little green play arrow to the right of the `cars` code chunk, then the `summary(cars)` command in the chunk will be evaluated. This will immediately display the results below the code chunk. This allows you to play around with code on a chunk-by-chunk basis so you don't have to knit the whole thing together every time you make a change.
- In the subwindow displaying the result of the code chunk, in the upper right corner is a tiny `x` that you can press to close the subwindow again.

- Often your code chunks will depend up code chunks above the one you are looking at. To the left of the green play arrow is another button that when pressed, not only evaluates that code chunk, but also every code chunk above it. This can be helpful when your current code chunk depends on the higher up chunks being evaluated.

Functions

Commands like `sum` and `mean` are actually functions in R. The last thing we will discuss is how to create your own functions.

- First, just as with numbers, we assign functions to a name using the assignment operator `<-`. The difference between a function and a set of commands is that a function has *input variables* and can *return* a single variable. Consider the following function in R.

```
add <- function(a = 2, b = 4) {
  s <- a + 2*b
  return(s)
}
```

- Now that this function has been defined, we can use it in commands.

```
add(5, 6)
```

```
## [1] 17
```

- The two input variables here are `a` and `b`. The return variable is `s`. Note that I gave default values for the parameters `a` and `b`. So if I don't specify those parameters, then they get their default values. Also, I do not have to give the variables in order, I can rearrange them by specifying the name. Try the following to see how these ideas work in practice.

```
add(5)
add(b=2)
add(4, -10)
add(b=-10, a=4)
```

6. Create a function that takes two inputs `x` and `y`, and returns x^y .

Useful Links

Some links that you might find useful as you learn R, R Markdown, and \LaTeX .

- Basic R cheat sheet: <https://www.rstudio.com/wp-content/uploads/2016/10/r-cheat-sheet-3.pdf>
- Reference for R Markdown: https://rmarkdown.rstudio.com/authoring_basics.html
- Reference for \LaTeX symbol commands: https://oeis.org/wiki/List_of_LaTeX_mathematical_symbols
- Find a $LaTeX$ symbol by drawing it: <http://detexify.kirelabs.org/classify.html>

Exploration: Using graphical grammars in the tidyverse

29.1 Summary

In this lab you will learn how to create many of the common visualizations using **ggplot2**.

29.2 Scatterplots

Our code will always have two parts. In the first part we set up the data that is being used, and in the second part we create the plot.

- The first plot is intended to introduce you to several of the options available for a scatterplot.

```
# If necessary, you might need to install the package ggplot2 w
# install.packages("ggplot2")
options(scipen=999) # turn-off scientific notation like 1e+48
library(ggplot2)
theme_set(theme_bw()) # pre-set the bw theme.

# Scatterplot
ggplot(midwest, aes(x=area, y=poptotal)) +
  geom_point(aes(col=state, size=popdensity)) +
  geom_smooth(method = "loess", se = FALSE) +
  labs(subtitle="Area Vs Population",
       y="Population",
       x="Area",
       title="Scatterplot",
       caption = "Source: midwest")
```

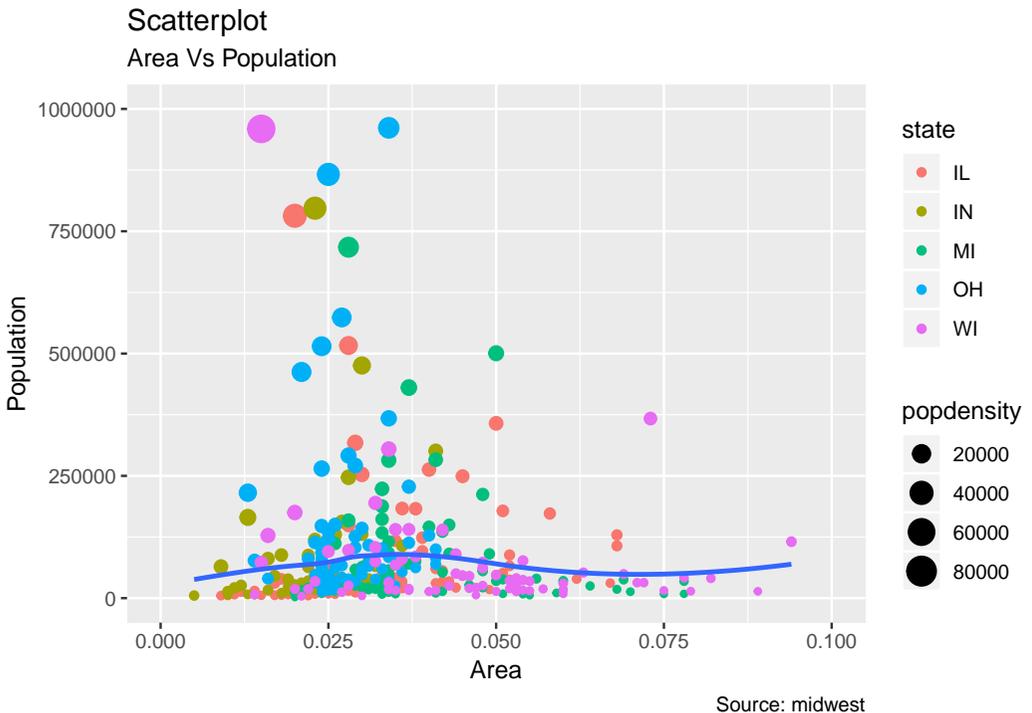
Questions

1. What is the title of the plot created?

2. Looking at the plot, in which state is the county with the highest population located?
3. What does `se = FALSE` do?

Let's modify the plot by only keeping counties under a million, and areas from 0 to 0.1. To do this, modify the `ggplot` function call as

```
ggplot(midwest, aes(x=area, y=poptotal)) +
  geom_point(aes(col=state, size=popdensity)) +
  geom_smooth(method="loess", se=F) +
  xlim(c(0, 0.1)) +
  ylim(c(0, 10^6)) +
  labs(subtitle="Area Vs Population",
       y="Population",
       x="Area",
       title="Scatterplot",
       caption = "Source: midwest")
```



- Now let's circle some of the values that are very high, greater than 800,000. This can be done with the `geom_encircle` command from the `ggalt` package.
- First we need to load in the package, and select the points to be encircled.

```
# install.packages("ggalt")      (Might need to install the p
library(ggalt)
midwest_select <- midwest[midwest$poptotal > 8*10^5, ]
```

- Next, modify the plot by encircling the selected values.

```
ggplot(midwest, aes(x=area, y=poptotal)) +
  geom_point(aes(col=state, size=popdensity)) +
  geom_smooth(method="loess", se=F) +
  geom_encircle(aes(x=area, y=poptotal),
               data=midwest_select,
               color="red",
               size=2,
               expand=0.04) + # encircle xlim(c(0, 0.1)) +
  ylim(c(0, 10^6)) +
  labs(subtitle="Area Vs Population",
       y="Population",
       x="Area",
       title="Scatterplot",
       caption = "Source: midwest")
```

Questions

4. Try encircling the points with populations between $3 \cdot 10^5$ and $5 \cdot 10^5$.

29.3 Kernel Density plots

Given data from a distribution, it is helpful to have a way of estimating the density of the distribution. This is called a *kernel density plot*. The word kernel here refers to the type of smoothing. The default is to use a normal distribution, which is why our plots will look kind of like gentle mountains.

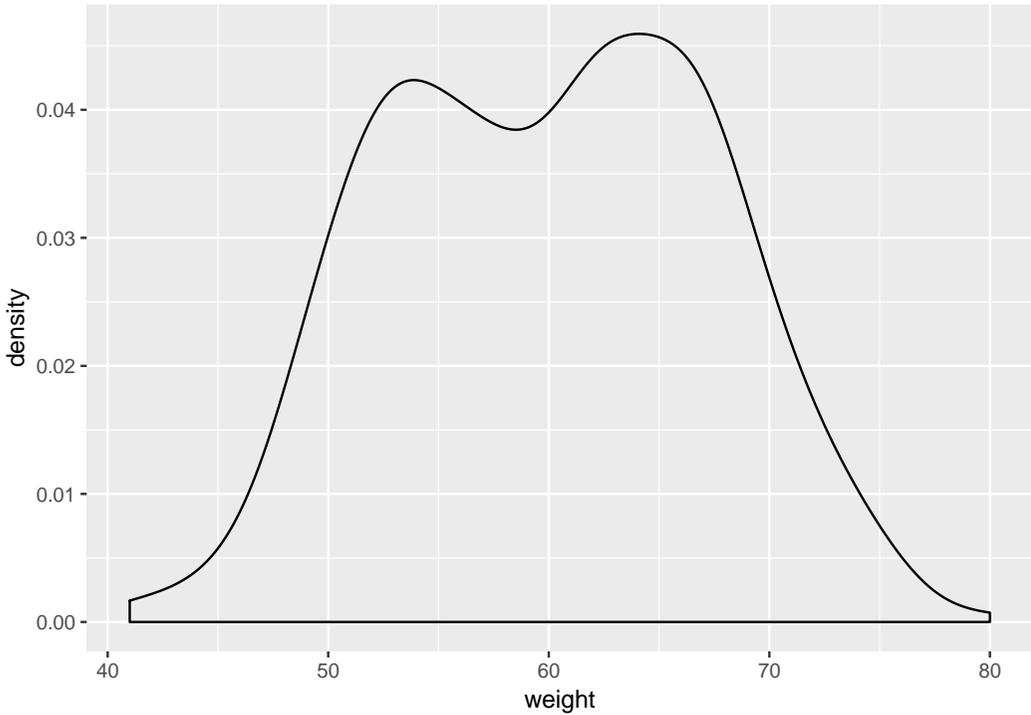
- To see how this can be accomplished with `ggplot2`, first let us create some random data.

```
set.seed(1234)
df <- data.frame (
  sex=factor(rep(c("F", "M"), each=200)),
  weight=round(c(rnorm(200, mean=55, sd=5),
                 rnorm(200, mean=65, sd=5)))
)
head(df)
```

```
##   sex weight
## 1   F     49
## 2   F     56
## 3   F     60
## 4   F     43
## 5   F     57
## 6   F     58
```

- This simulates 200 draws for the weight of male subjects, and 200 draws for the weights of female subjects.
- The basic density plot in ggplot2 is called `geom_density()`. Try

```
library(ggplot2)
p <- ggplot(df, aes(x = weight)) +
  geom_density()
p
```



- We can also add in a vertical line indicating the mean.

```
p + geom_vline(aes(xintercept = mean(weight)),
```

- As with most `geom` functions, the `color` parameter changes the color of the line, while `fill` changes the color of the area under the line.

```
ggplot(df, aes(x = weight)) +
  geom_density(color = "darkblue", fill = "lightblue")
```

- This mix of normals is hiding the difference in average weight between men and women. To break out the data, we need only declare that the two groups should be treated separately in the plot:

```
ggplot(df, aes(x = weight, color = sex)) +
  geom_density()
```

Questions

5. Use

```
values <- rexp(100, rate=2)
df <- data.frame(values)
```

to generate some random data in the variable `df`. Write code to plot a kernel density estimate and a vertical line at the sample mean.

6. Use the following code to examine the mpg of various cars using differing numbers of cylinders.

```
library(ggplot2)
theme_set(theme_classic())

# Plot
g <- ggplot(mpg, aes(cty))
g + geom_density(aes(fill=factor(cyl)), alpha=0.8) +
  labs(title="Density plot",
        subtitle="City Mileage Grouped by Number of cylinders",
        caption="Source: mpg",
        x="City Mileage",
        fill="# Cylinders")
```

From your plot, which variable has more spread, the mpg with 5 cylinders, or the mpg with 6 cylinders?

29.4 Moving the legend around

By default the legend is on the right hand side, but can be moved or eliminated with the `theme` function.

```
p + theme(legend.position="top")
p + theme(legend.position="bottom")
p + theme(legend.position="none") # Remove legend
```

Questions

- Using the `theme` function, change the aspect ratio of the plot to 4:3.

Animating charts

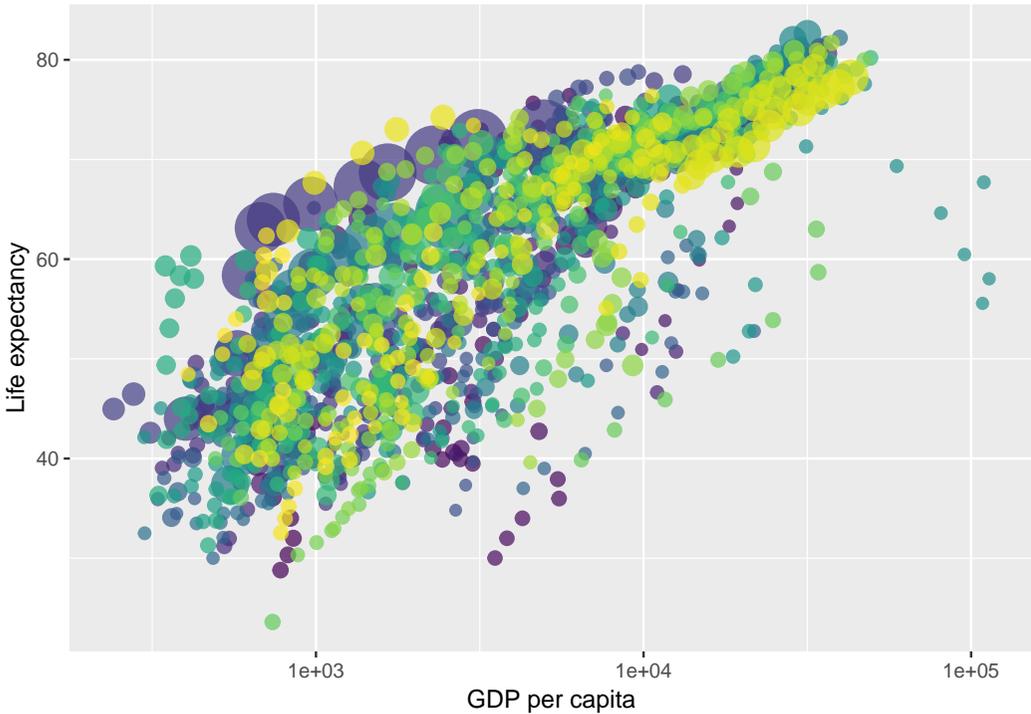
In the example above we plotted the point for each county using area and population. We then used the color to indicate the state, and the size to indicate a fourth variable. Hence in our two dimensional plot, we were effectively visualizing four dimensions.

We can add a fifth dimension to our chart using time, that is, by creating an animation. When possible, the time being displayed in R should be a time dimension.

- For instance, consider the following code which makes a static plot using data from the package `gapminder`:

```
library(gapminder)

p <- ggplot(
  gapminder,
  aes(x = gdpPercap, y=lifeExp, size = pop, colour = country)
) +
  geom_point(show.legend = FALSE, alpha = 0.7) +
  scale_color_viridis_d() +
  scale_size(range = c(2, 12)) +
  scale_x_log10() +
  labs(x = "GDP per capita", y = "Life expectancy")
p
```



- Add in the `gganimate` library so that we can use it.

```
# install.packages("gganimate")
library(gganimate)
```

- Now by using the function `transition_time` in the `gganimate` package we can show how the life expectancy versus GDP changes over time. Note that you might need to install packages `gifski` and `png` before running this code. Note that it can take quite a bit of time to build an animation such as this.

```
p + transition_time(year) +
  labs(title = "Year: {frame_time}")
```

- If you right click and save the image that you have created, then be sure to give it an `.gif` extension so that programs will know how to handle it.
- The functions that we have learned about can be used with the `ggplot2` function. For instance, to create multiple facets with the animation, we could replace the above code with

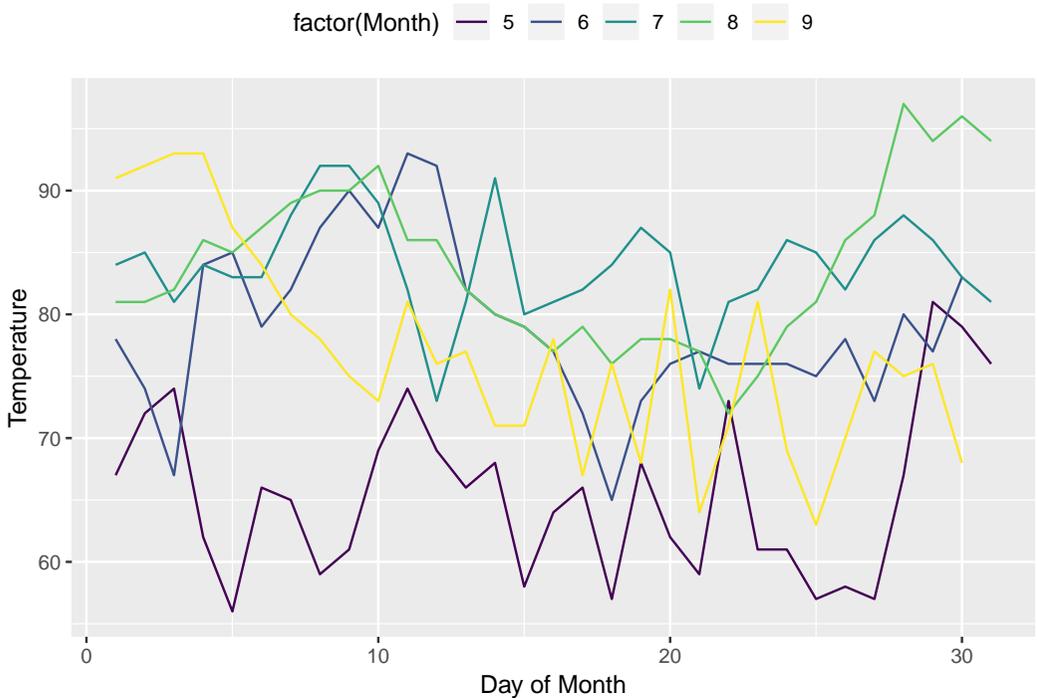
```
p + facet_wrap(~continent) +
  transition_time(year) +
  labs(title = "Year: {frame_time}")
```

- The purpose of a wake is to show how the points are moving by leaving smaller versions of themselves behind them. The result looks as follows.

```
p + transition_time(year) +
  labs(title = "Year: {frame_time}") +
  shadow_wake(wake_length = 0.1, alpha = FALSE)
```

- We can also use animation to let data gradually appear. For instance, consider loading in some air quality data.

```
p <- ggplot(
  airquality,
  aes(Day, Temp, group = Month, color = factor(Month))
) +
  geom_line() +
  scale_color_viridis_d() +
  labs(x = "Day of Month", y = "Temperature") +
  theme(legend.position = "top")
p
```



- If we wanted to draw this data more slowly, we could use the `transition_reveal` function.

p + transition_reveal (Day)

Exploration: Transforming data with *dplyr*

Summary In this lab you will learn how to manipulate data using the `dplyr` package. Most of the content of this lab came from the `dplyr` website at: <https://dplyr.tidyverse.org/>

- Start by loading in the `dplyr` library (installing the package first if necessary.)

```
# install.packages("dplyr")  
library(dplyr)
```

- The `dplyr` package contains tools for manipulating data contained in a `data.frame` or `tibble`. Let's look at the `starwars` variable.

```
starwars
```

```
## # A tibble: 87 x 13  
##   name height mass hair_color skin_color eye_color  
##   <chr> <int> <dbl> <chr> <chr> <chr>  
## 1 Luke~ 172 77 blond fair blue  
## 2 C-3PO 167 75 <NA> gold yellow  
## 3 R2-D2 96 32 <NA> white, bl~ red  
## 4 Dart~ 202 136 none white yellow  
## 5 Leia~ 150 49 brown light brown  
## 6 Owen~ 178 120 brown, gr~ light blue  
## 7 Beru~ 165 75 brown light blue  
## 8 R5-D4 97 32 <NA> white, red red  
## 9 Bigg~ 183 84 black light brown  
## 10 Obi~ 182 77 auburn, w~ fair blue-gray
```

```
## # ... with 77 more rows, and 7 more variables:
## #   birth_year <dbl>, gender <chr>, homeworld <chr>,
## #   species <chr>, films <list>, vehicles <list>,
## #   starships <list>
```

The data in this tibble consists of some of the characters that appear in the Star Wars movies. Since it is 87 by 13, there are 87 data values, and 13 variables (also called factors).

Select

We might not be interested in all the variables, and the `select` function allows us to only look at the variables that are important. For instance, if we only wanted the name, mass, species, and homeworld, we could use

```
select(starwars, name, mass, species, homeworld)
```

The result is a tibble that just contains the 4 variables listed. We can also use helper functions like `starts_with`, `ends_with`, and `contains`. Try

```
select(starwars, ends_with("color"))
```

to see the variables that end with the string "color", and

```
select(starwars, contains("a"))
```

to see those variables that have the string "a" somewhere in the name.

The first parameter we pass to `select` is the name of the variable, but it is also possible to use **pipes** to accomplish the same task. The following command pipes the variable `starwars` into the `select` function:

```
starwars %>% select(contains("a"))
```

Questions

1. Give a command that gives the data from `starwars` that has the factors name, gender, and homeworld.
2. Give a command that returns the factors of `starwars` that contains an "e". How many such factors are there?

Filter

First, let's search for the droid characters. To find the droids that we are looking for, try

```
starwars %>%
  filter(species == "Droid")

## # A tibble: 5 x 13
##   name height mass hair_color skin_color eye_color
##   <chr> <int> <dbl> <chr>         <chr>         <chr>
## 1 C-3PO  167    75 <NA>         gold          yellow
## 2 R2-D2   96    32 <NA>         white, bl~ red
## 3 R5-D4   97    32 <NA>         white, red red
## 4 IG-88  200   140 none         metal         red
## 5 BB8     NA     NA none         none         black
## # ... with 7 more variables: birth_year <dbl>, gender <chr>,
## #   homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

Of course this search was practically instantaneous because there are so few rows of data. In practice, there are often more data rows than variables. So it can be helpful to insert a `select` function before the `filter` function. We then connect the `select` function to the `filter` function with a pipe.

```
starwars %>%
  select(name, mass, species, gender) %>%
  filter(species == "Droid")

## # A tibble: 5 x 4
##   name    mass species gender
##   <chr> <dbl> <chr>  <chr>
## 1 C-3PO    75 Droid  <NA>
## 2 R2-D2    32 Droid  <NA>
## 3 R5-D4    32 Droid  <NA>
## 4 IG-88   140 Droid  none
## 5 BB8      NA Droid  none
```

Logical operators

We can also use filters to search for more than one characteristic with the `&` logical operator. This represents logical and, which is true only if both of the expressions are true. So `TRUE & TRUE` equals `TRUE`, `FALSE & TRUE` is `FALSE`, `TRUE & FALSE` is `FALSE`, and `FALSE & FALSE` is `FALSE`.

Try

```
starwars %>%
  select(name, mass, species, gender) %>%
  filter(species == "Droid" & mass > 50)
```

```
## # A tibble: 2 x 4
##   name    mass species gender
##   <chr> <dbl> <chr>  <chr>
## 1 C-3PO     75 Droid  <NA>
## 2 IG-88    140 Droid  none
```

to find the droids that have mass greater than 50.

The logical operator `|` is true if either one (or both) of the expressions it connects is true. So `TRUE | TRUE` equals `TRUE`, `FALSE | TRUE` is `TRUE`, `TRUE | FALSE` is `TRUE`, and `FALSE | FALSE` is `FALSE`. Try

```
starwars %>%
  select(name, mass, species, gender) %>%
  filter(species == "Droid" | mass == 136)
```

```
## # A tibble: 7 x 4
##   name          mass species gender
##   <chr>        <dbl> <chr>  <chr>
## 1 C-3PO          75 Droid  <NA>
## 2 R2-D2          32 Droid  <NA>
## 3 Darth Vader   136 Human  male
## 4 R5-D4          32 Droid  <NA>
## 5 IG-88        140 Droid  none
## 6 Tarfful       136 Wookiee male
## 7 BB8           NA Droid  none
```

This should pick up the well known Darth Vader and the less well-known Tarfful, Wookiee general during the Clone Wars.

You will notice that some of the droids are missing values for factors. For instance, BB8 does not have either a mass value. To find these entries `NA` (which stands for not available) values, we can use the `is.na` function. Try

```
starwars %>%
  select(name, mass, species, gender) %>%
  filter(is.na(mass) & species == "Droid")
```

```
## # A tibble: 1 x 4
##   name    mass species gender
##   <chr> <dbl> <chr>  <chr>
## 1 BB8     NA Droid  none
```

to find all the data where the height is not available.

Another useful logical operator in this context is `!`, which means **not**. So the following will tell us the droids where the mass does not equal NA.

```
starwars %>%
  select(name, mass, species, gender) %>%
  filter(!is.na(mass) & species == "Droid")
```

```
## # A tibble: 4 x 4
##   name      mass species gender
##   <chr> <dbl> <chr>   <chr>
## 1 C-3PO     75 Droid   <NA>
## 2 R2-D2     32 Droid   <NA>
## 3 R5-D4     32 Droid   <NA>
## 4 IG-88    140 Droid   none
```

Logical operators are evaluated from left to right. So for instance,

```
starwars %>%
  select(name, mass, species, gender) %>%
  filter(species == "Droid" & mass > 100 | mass < 40)
```

```
## # A tibble: 6 x 4
##   name                mass species      gender
##   <chr>              <dbl> <chr>      <chr>
## 1 R2-D2                32 Droid      <NA>
## 2 R5-D4                32 Droid      <NA>
## 3 Yoda                 17 Yoda's species male
## 4 IG-88               140 Droid      none
## 5 Wicket Systri Warrick 20 Ewok       male
## 6 Ratts Tyerell        15 Aleena    male
```

For Wicket, it was false that his species is a droid, and false that his mass is greater than 100. So the first two clauses become false. But the final mass value is less than 40, and `FALSE | TRUE` evaluates to `TRUE`.

If instead we are interested in only those droids who have mass greater than 100 or mass less than 40, then

```
starwars %>%
  select(name, mass, species, gender) %>%
  filter(species == "Droid" & (mass > 100 | mass < 40))
```

does the job.

Questions

3. Create a command to find the characters who are female. How many are there in the data?
4. Create a tibble from the variable `starwars` that has the factors `name`, `gender`, `hair_color`, and `homeworld`, and only characters with blond hair from Tatooine.

Mutate

Mutate alters a tibble by adding an extra variable that can be some function of other variables. For instance, suppose we are interested in how the mass varies with height. We could compute the ratio as follows.

```
starwars %>%
  select(name, mass, height) %>%
  mutate(massweightratio = mass/height)
```

```
## # A tibble: 87 x 4
##   name                mass height massweightratio
##   <chr>                <dbl> <int>         <dbl>
## 1 Luke Skywalker      77    172          0.448
## 2 C-3PO                75    167          0.449
## 3 R2-D2                32     96          0.333
## 4 Darth Vader        136    202          0.673
## 5 Leia Organa         49    150          0.327
## 6 Owen Lars          120    178          0.674
## 7 Beru Whitesun lars  75    165          0.455
## 8 R5-D4                32     97          0.330
## 9 Biggs Darklighter  84    183          0.459
## 10 Obi-Wan Kenobi     77    182          0.423
## # ... with 77 more rows
```

Note that if either the mass or the height variable is NA, then their ratio will also be NA

```
starwars %>%
  select(name, mass, height) %>%
  mutate(massweightratio = mass/height) %>%
  filter(is.na(massweightratio))
```

```
## # A tibble: 28 x 4
##   name          mass height massweighratio
##   <chr>        <dbl> <int>         <dbl>
## 1 Wilhuff Tarkin    NA    180            NA
## 2 Mon Mothma       NA    150            NA
## 3 Arvel Crynyd     NA     NA            NA
## 4 Finis Valorum    NA    170            NA
## 5 Rugor Nass       NA    206            NA
## 6 Ric Olie         NA    183            NA
## 7 Watto           NA    137            NA
## 8 Quarsh Panaka    NA    183            NA
## 9 Shmi Skywalker   NA    163            NA
## 10 Bib Fortuna     NA    180            NA
## # ... with 18 more rows
```

Questions

5. Currently the mass is in kilograms. Create a new variable where the mass is measured in pounds by multiplying by 2.20462.
6. How many pounds does Darth Vader weigh?
7. What happens if you try to add a categorical variable like `hair_color` to `height`?

Arrange

Another way to transform the data is through the `arrange` function. This sorts the data by a particular variable so we can learn about the highest or lowest values. The following sorts the variable by mass.

```
starwars %>%
  select(name, mass, height) %>%
  arrange(mass)
```

As you can see, this arranges the data from low mass to high mass.

When you `arrange` based on a categorical variable like `hair_color`, it sorts things alphabetically.

```
starwars %>%
  select(name, hair_color, mass, height) %>%
  arrange(hair_color)
```

If we want to reverse the sort, we use the helper function `desc`. By putting this around the variable name, we reverse the order of the sorting.

```
starwars %>%
  select(name, hair_color, mass, height) %>%
  mutate(massweightratio = mass/height) %>%
  arrange(desc(hair_color))
```

```
## # A tibble: 87 x 5
##   name          hair_color  mass height massweightratio
##   <chr>         <chr>    <dbl> <int>         <dbl>
## 1 Yoda          white     17     66           0.258
## 2 Ki-Adi-Mundi white     82    198           0.414
## 3 Dooku         white     80    193           0.415
## 4 Jocasta Nu    white     NA     167           NA
## 5 Captain Phasma unknown    NA     NA           NA
## 6 Darth Vader   none     136    202           0.673
## 7 IG-88         none     140    200           0.7
## 8 Bossk         none     113    190           0.595
## 9 Lobot         none     79     175           0.451
## 10 Ackbar       none     83     180           0.461
## # ... with 77 more rows
```

To break ties, we just add another variable to the `arrange` function.

```
starwars %>%
  select(name, hair_color, mass, height) %>%
  mutate(massweightratio = mass/height) %>%
  arrange(desc(hair_color), mass)
```

```
## # A tibble: 87 x 5
##   name          hair_color  mass height massweightratio
##   <chr>         <chr>    <dbl> <int>         <dbl>
## 1 Yoda          white     17     66           0.258
## 2 Dooku         white     80    193           0.415
## 3 Ki-Adi-Mundi white     82    198           0.414
## 4 Jocasta Nu    white     NA     167           NA
## 5 Captain Phasma unknown    NA     NA           NA
## 6 Ratts Tyerell none     15     79           0.190
## 7 Sebulba       none     40    112           0.357
## 8 Dud Bolt      none     45     94           0.479
```

```
## 9 Wat Tambor      none      48      193      0.249
## 10 Sly Moore      none      48      178      0.270
## # ... with 77 more rows
```

Note that entries with NA will appear last whether you are arranging either in ascending or descending order.

Questions

8. Which of the characters in the `starwars` variable are the tallest?

9. What is the characters in the `starwars` variable weighs the least?

group_by and summarize

The `group_by` function takes a tibble and partitions the data based on a particular variable. For instance, `group_by(species)` breaks the tibble into 9 groups based on the species of the character.

```
starwars %>%
  group_by(species)
```

Now `group_by` by itself does not do anything, what it does is allow other functions to work on the groups. For instance, the `summarize` function operates by group.

```
starwars %>%
  group_by(species) %>%
  summarise(
    mass = mean(mass, na.rm = TRUE)
  )
```

The `n` function here can be useful in counting the number of data points in each group.

```
starwars %>%
  group_by(species) %>%
  summarise(
    n = n(),
    mass = mean(mass, na.rm = TRUE)
  )
```


Exploration: Projects in R and Tibbles

Summary You can use good habits involving file directories and R projects in order to make finding information easier later on. We will also show how to save figures and data sets in individual files.

We then discuss some of the commands that apply to tibbles, the extension of the `data.frame` variable type in the tidyverse. Content for this lab was drawn from Chapters 8 and 9 of Grolemund and Wickham, *R for Data Science* <https://r4ds.had.co.nz/>.

The Workspace

- Open up RStudio and in the console, type

```
x <- 2
```

Now close RStudio.

- You will immediately be greeted with a question asking if you want to save your workspace image before closing. Saving your workspace image will save your variable definitions in the console. So the next time you open up RStudio, everything is as you left it.
- Click on save. This will create a new file with an extension `.RData` Now reopen RStudio, click on

File → Open File...

and navigate to the `.RData` file that you just saved. (Unless you changed your directory earlier, it should still be in your home directory.) Open it up. The value `x = 2` should now be back in your Global Environment. Great!

- Seems like a good feature, right? Well, it actually can lead to some bad habits, and I'm going to recommend that you *not* save your Workspace each time you leave your session. Instead, we'll use a different method that leads to better overall habits.

Script and R Markdown

When we started the course, we said that typically the last thing we will do as a data scientist is to communicate our results to other. But even though its the last thing we do, it helps to prepare for it from the very start. That means instead of doing your analysis in the console, use scripts and R Markdown files for your analytical work.

- Work done in the console can be difficult to reproduce later. It is also difficult to make sure that you haven't made an error in typing parameters or commands somewhere down the line. By putting all your commands in your analysis in a script or R Markdown file, you are automatically keeping a record of what you did and accomplished.
- Suppose you have a figure in your report or paper that needs changing a month or two later. If you have the commands recorded in a script that created the figure, then all you have to do is go back and alter one or two things. But if not, you have to start from scratch. And it is very possible that you will forget exactly how you created the figure in the first place, making it impossible to make your changes.
- This also makes sense if you are doing a preliminary analysis of data. If you get more data later, incorporating that into a preset workflow will be a breeze. Starting from scratch will be painful.
- Everytime you Source a script or Knit an R Markdown file, you are also saving it. That means that in order to make progress in your analysis, you are automatically keeping a record of what you have done! That's a good habit to get into.

File Directories

Directories are used to organized the many files on your computer. If all your files were in one place, things would quickly become a mess.

- If you look at the console pane, just beneath the word console is a directory. This is your *working directory*. If you cannot find it, try

```
getwd()
```

Your working directory might be as short as "`~/`". We'll talk more about that directory in a minute

Questions

1. What is your current working directory?

There are two different ways of handling directories, based on whether you have a Windows or a Mac/Unix machine. In fact, they have slightly different conventions.

1. In windows, the file directory symbol is a backward slash `\`, while for Mac/Linux it is a forward slash `/`. But, when you are using R, you should always use the forward slash.
2. Remember `~`? When used as a file directory, this points to your home directory. In Mac and Linux, this is the highest directory for your username. In Windows, this is your Documents directory.
3. You can also use what are called absolute paths. In Windows, you start an absolute path using either the drive letter (such as `C:`) or two backslashes if you are getting data from a server (for example `\\myserver`). In Mac/Linux, the absolute paths start with a slash, for example `/home/mhuber`. Despite just giving you this information, I recommend never using absolute paths. The reason is that it makes it difficult to send and share your projects with other people. By using `~`, it is much easier to write scripts and markdown files that work across multiple computers.

Questions

2. What's the home directory for a Windows computer?
3. What's the home directory for a Mac/Linux computer if your username is `smile42`?

RStudio projects

It makes sense when working on a large project to keep all your files in one place. This means

- input data
- R Scripts
- R Markdown files
- Figures and analytical results

To help you out in this endeavor, RStudio has a method called a project. To start a project, use

File → New Project...

You will be asked if you want to create a new directory, use an existing directory, or use a version control system (such as Git). Just create a new directory for now.

- You will be asked to name the new directory and where you want it to be a subdirectory of. You can call it something like "datascience" and put it under your home directory.
- Once you create your project, RStudio gives you a blank slate upon which to work. In the lower right hand pane, on the Files tab, it should say `datascience.Rproj` since that is your project.
- Try opening a new script called `script1` in your console, and type a few commands. Now close RStudio. Open RStudio, Your `script1.R` tab should still be there.

Questions

4. Create a new project `datascience2` in the existing directory `~/datascience`. Create a new script file `script2.R`. Now switch back to the project `datascience` and note that it has the scripts associated with the first project you made.

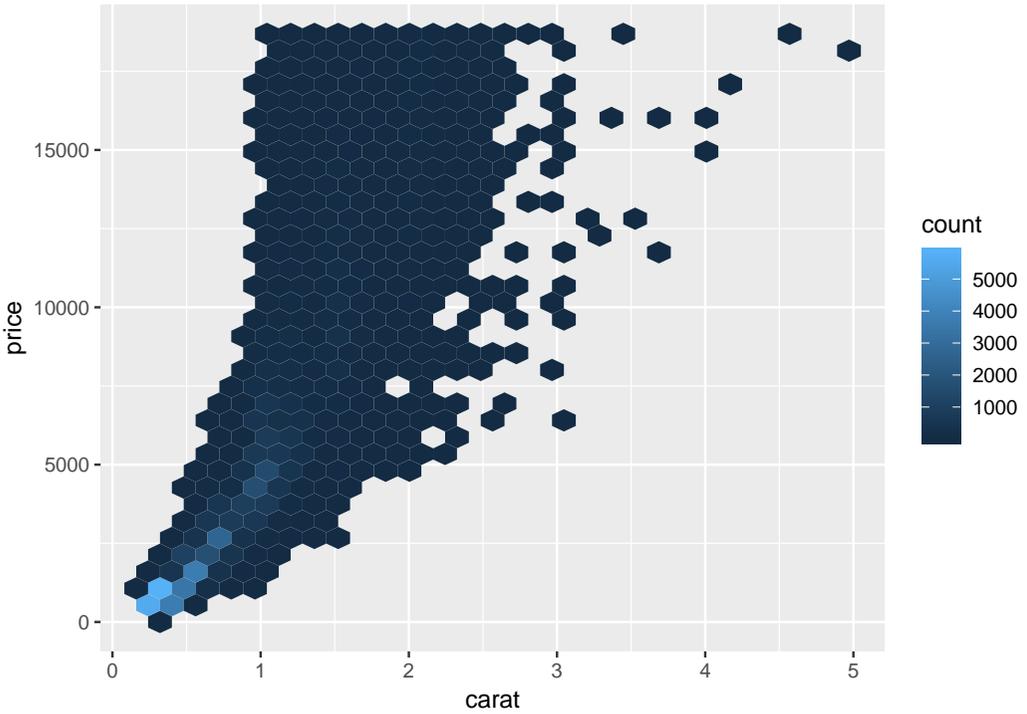
Saving files for a project

Now that you have a project, you can use it to store the output of your scripts (and markdown files).

- Go back to your `datascience` project and `script1.R`. Inside the script, put

```
library(tidyverse)

ggplot(diamonds, aes(carat, price)) +
  geom_hex()
```



```
ggsave("diamonds.pdf")
```

```
write_csv(diamonds, "diamonds.csv")
```

- Now use either the File Explorer (in Windows) or the Finder (in Mac) or whatever GUI or command window you are using in Linux to navigate to the folder `~/datascience`.
- You should find files `diamond.pdf` and `diamonds.csv` there. Open them up and see if they were what you expected.

Question

5. While `diamonds.pdf` is open, try running `script1` again. What error message do you get?
6. The bins in `geom_hex` are hexagonally shaped. Try changing `geom_hex` to `geom_bin2d`. What shape are the bins now?

Tibbles

When R was built, the `data.frame` data type was the primary way that data could be stored. In the tidyverse, the `data.frame` has been upgrade to a `tibble`, which has some nice properties.

- Begin by loading in the tidyverse package.

```
library(tidyverse)
```

- Type `iris` into the console. Since `iris` is a `data.frame`, it tries to list the entire variable. Now try in the console:

```
as_tibble(iris)
```

The result is much more nicely formatted.

We can use the function `head` to only print the first few lines of a `data.frame`. But `data.frames` by default will change the character type of strings. So for instance consider the built in variable `letters` in R:

```
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m"
## [14] "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

If we put it in a data frame, the strings get turned into levels for the factor `x`.

```
head(data.frame(x = letters))
```

Whereas if we make it a tibble, the values in `x` stay as strings (`<chr>` type):

```
tibble(x = letters)
```

Unlike data frames, tibbles can use weirder variable names that are not valid for `data.frame` variables. Try

```
tb <- tibble(
  `:)` = "smile",
  ` ` = "space",
  `2000` = "number"
)
tb
```

to see an unusual tibble.

You can load a tibble either column by column:

```
df <- tibble(
  x <- runif(5),
  y <- rnorm(5)
)
```

or you can load by row using the `tribble` (short for transposed tibble) function:

```
tb <- tribble(
  ~x, ~y, ~z,
  #--/--/-----
  "a", 2, 3.6,
  "b", 1, 8.5
)
tb
```

```
## # A tibble: 2 x 3
##   x         y         z
##   <chr> <dbl> <dbl>
## 1 a         2     3.6
## 2 b         1     8.5
```

Note that we use `~x` to indicate that `x` is the name of this particular variable.

If we are unsure if we are dealing with a tibble, `is_tibble` can be used to check.

```
is_tibble(tb)
```

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

```
df <- data.frame(x = c(1,2,3))
is_tibble(df)
```

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

Printing tibbles

If you actually need to print an entire tibble, just use the `print` function. The parameter `n` controls the number of lines, and `width` the number of columns

```
print(as_tibble(iris), n = 15, width = 40)
```

You can also load the tibble into its own pane with the `View` function:

```
View(as_tibble(iris))
```

When printing out tibbles, sometimes it helps to have the variables on the rows, and the first few data values in each row. The `glimpse` function does exactly this.

```
glimpse(as_tibble(iris))
```

```
## Observations: 150
## Variables: 5
## $ Sepal.Length <dbl> 5.1, 4.9, 4.7, 4.6, 5.0, 5.4, 4.6, 5.0,
## $ Sepal.Width <dbl> 3.5, 3.0, 3.2, 3.1, 3.6, 3.9, 3.4, 3.4,
## $ Petal.Length <dbl> 1.4, 1.4, 1.3, 1.5, 1.4, 1.7, 1.4, 1.5,
## $ Petal.Width <dbl> 0.2, 0.2, 0.2, 0.2, 0.2, 0.4, 0.3, 0.2,
## $ Species <fct> setosa, setosa, setosa, setosa, setosa,
```

Selecting rows and columns

As with a `data.frame`, you can use `$` to pull out the value of a particular column/variable.

```
tb$y
```

You can also use double square brackets to pull out rows. For instance, since `y` was variable number 2:

```
tb[[2]]
```

Backward compatibility

Some older R functions might not work with tibbles, in which case you need to convert back to a `data.frame`. Try

```
as.data.frame(df)
```

to see how that is done.

Questions

7. Give two different commands that yield the `z` variable in `tb`.

Plotting data

(This part of the lab taken from <https://monashbioinformaticsplatform.github.io/r-more/topics/tidyverse.html>.)

When plotting categorical versus categorical data, the tile geometry can be used to get an idea of how categories interact with one another.

Download the file `fastqc.csv` and place it in your working directory. Use

```
bigtab <- read_csv("~/fastqc.csv")
```

to input the data into the tibble `bigtab`.

Lets see how the grade changes based on the `test` and `file` categorical variables.

```
ggplot(bigtab, aes(x = file, y = test, color = grade)) +  
  geom_point()
```

The tile geom is better as displaying this type of graph.

```
ggplot(bigtab, aes(x = file, y = test, fill = grade)) +  
  geom_tile()
```

This is better, but still is not professional quality. There's still some problems.

- The file names along the horizontal style overlap.
- The vertical axis names have the first names alphabetically at the bottom.
- We don't need the gray background behind the graph anymore.
- Grid lines would help in reading the data,

Using the `factor` function allows us to accomplish this

```
# y axis plots from bottom to top, so reverse
y_order <- sort(unique(bigtab$test), decreasing = TRUE)
bigtab$test <- factor(bigtab$test, levels = y_order)

x_order <- unique(bigtab$file)
bigtab$file <- factor(bigtab$file, levels = x_order)

# Give PASS the color green and FAIL the color red
color_order <- c("FAIL", "WARN", "PASS")
bigtab$grade <- factor(bigtab$grade, levels = color_order)

myplot <- ggplot(bigtab, aes(x = file, y = test, fill = grade))
```

```
geom_tile(color = "black", size = 0.5) + # Black border on tiles
scale_fill_manual(                        # Colors, as color hex codes
  values=c("#ee0000", "#ffee00", "#00aa00")) +
labs(x = "", y = "", fill = "") +        # Remove axis labels
coord_fixed() +                          # Square tiles
theme_minimal() +                        # Minimal theme, no grid lines
theme(panel.grid = element_blank(),      # No underlying grid lines
  axis.text.x = element_text(           # Vertical text on x axis
    angle=90, vjust=0.5, hjust=0))
```

Now you can just use `myplot` to see your plot.

Chapter 32

Exploration: tidying data with tidyr

The content of this lab are based on Chapter 12.6 of *R for Data Science* by Hadley Wickham and Garrett Grolemund.

Tidy data

Data is said to be **tidy** if it satisfies:

1. Each row contains an observation.
2. Each column contains a variable.
3. Each entry (row and column) contains a single value.

In this lab we will practice tidying data using the elements of the **tidyr** package.

- As usual, start by loading in the tidyverse:

```
# install.packages("tidyverse")  
library(tidyverse)
```

The WHO Tuberculosis Data Set

- The data set we'll be using here comes from the World Health Organization (WHO) and is their Global Tuberculosis Report from 2014. It is located in the variable 'who' in the 'tidyr' package which is part of the 'tidyverse' group of packages.

Take a look with

```
who
```

Questions

1. How many observations are there?
2. How many variables are there?

The Data Dictionary

- There are many variables that seem to actually be values rather than variables. For instance, consider ‘new_sp_m014’ and ‘new_sp_m1524’. They appear to both be ‘NA’ for most of the first few rows. Lets take a look at the first instances where they are not ‘NA’:

```
who %>% filter(!is.na(new_sp_m014))
```

```
## Warning: package 'bindrcpp' was built under R version 3.4.4
```

- At this point, we would need to consult a *data dictionary*, a description of what is inside the data set, in order to understand what the data variables are telling us. In this case, the first three letters are “new” for new cases of TB, and “old” for old cases of TB.

Questions

3. Does the data set contain any old cases of TB?

- Then there should be an underscore, followed by two or three letters indicating the type of TB case it is. **rel** for relapse **ep** for extrapulmonary TB **sn** TB that cannot be detected by a pulmonary smear (sn stands for smear negative) **sp** cases that can be detected by a pulmonary smear (smear positive)
- Next there should be another underscore, followed by **f** for female patients and **m** for male patients. Finally, there is a number which indexes the age of the patients in the group. For instance, **3544** indicates that the patients are 35-44 years old.
- Okay, so there’s a lot to take in here, but the most important thing is that the variable names are being used to encode data values. They should not be in the variable names, they should be an entry which indicates what type of patient we are dealing with.

Questions

- The relapse patients should (if they are following their naming scheme), have variable names that start `new_rel`. Are there variables whose name start with `new_rel`, or is there a typo in the data set?

Gather

- Let's begin by taking the variable names that are encoding data properties, and turn them into entries in a column. For now, we'll give that column the name `key`. Fortunately, all the variables that start with `new` are together sequentially, so we can use our `:` notation to indicate which variables are which.

```
who1 <- who %>%
  gather(new_sp_m014:newrel_f65, key = "key", value = "cases", 1)
who1
```

Questions

- How many new cases of TB among males aged 0-14 years where smear positive in Afganistan in 2006?

Fixing value strings

- As we saw in an earlier question, `new_rel` has been written `newrel` in some of the data entries. Fixing typos like this is not part of tidying data, it is part of *cleaning* data. We will talk much more about how to change strings later on in the course, but for now let's just `mutate` the entries in our `key` variable to change `newrel` to `new_rel` using the `str_change` function.

```
who2 <- who1 %>%
  mutate(key = str_replace(key, "newrel", "new_rel"))
```

Those relapse observations will be towards the end of the data set, so let's look at the last observations:

```
who2[76000:76036, ] %>% select(country, year, key, cases)
```

```
## # A tibble: 37 x 4
##   country          year key          cases
##   <chr>            <int> <chr>      <int>
## 1 Rwanda           2013 new_rel_f65  131
## 2 Saint Kitts and Nevis 2013 new_rel_f65    0
## 3 Saint Lucia       2013 new_rel_f65    0
## 4 Samoa            2013 new_rel_f65    2
## 5 Sao Tome and Principe 2013 new_rel_f65    6
## 6 Saudi Arabia      2013 new_rel_f65   98
## 7 Serbia           2013 new_rel_f65  170
## 8 Seychelles       2013 new_rel_f65    2
## 9 Sierra Leone     2013 new_rel_f65  128
## 10 Singapore       2013 new_rel_f65  130
## # ... with 27 more rows
```

Questions

6. How many female relapse cases did Saudia Arabia have in 2013?

- In tidy data each entry contains only one value, but each key variable is actually containing four pieces of information. We can break each single entry into four entries by using `separate`.

```
who3 <- who2 %>%
  separate(key, c("new", "type", "genderage"), sep = "_")
```

- Well, we broke it into three variables anyway, since the last two were not separated by an underscore, gender and age are still intermixed. The last split needed takes the first character and splits it off into gender:

```
who4 <- who3 %>%
  separate(genderage, c("gender", "age"), sep = 1)
who4 %>% select(iso2, year, new, gender, age)
```

```
## # A tibble: 76,046 x 5
##   iso2  year new  gender age
##   <chr> <int> <chr> <chr> <chr>
## 1 AF    1997 new  m     014
## 2 AF    1998 new  m     014
```

```
## 3 AF      1999 new    m      014
## 4 AF      2000 new    m      014
## 5 AF      2001 new    m      014
## 6 AF      2002 new    m      014
## 7 AF      2003 new    m      014
## 8 AF      2004 new    m      014
## 9 AF      2005 new    m      014
## 10 AF     2006 new    m      014
## # ... with 76,036 more rows
```

- One could argue that the new variable is unnecessary since all the entries are new. To verify this, use `count` to find the number of distinct entries in the `new` variable.

```
who4 %>% count(new)
```

```
## # A tibble: 1 x 2
##   new      n
##   <chr> <int>
## 1 new    76046
```

Questions

7. Use `count` to see how many observations come from the country of Andorra.

- Since `new` isn't giving us any information let's remove it. Also, the variables `iso2` and `iso3` are just other ways of identifying the county, so let's remove them as well.

```
who5 <- who4 %>% select(-new, -iso2, -iso3)
```

That's it, our data is now tidy!

Pew data set on religion and income

The Pew research center gathers data from a variety of sources. One such is a data set on the religion of various income levels. Download the file `pew.txt` from the web site to your working directory, and read it into R. This file is an example of a *tab-delimited* file where the values are all separated by tab characters. You can get a tab by using the escape character `"\t"`. So the command to read the file into a tibble is

```
pew <- read_delim("pew.txt", delim = "\t")
```

```
## Parsed with column specification:
## cols(
##   religion = col_character(),
##   `<$10k` = col_double(),
##   `$10-20k` = col_double(),
##   `$20-30k` = col_double(),
##   `$30-40k` = col_double(),
##   `$40-50k` = col_double(),
##   `$50-75k` = col_double(),
##   `$75-100k` = col_double(),
##   `$100-150k` = col_double(),
##   `>150k` = col_double(),
##   `Don't know/refused` = col_double()
## )
```

- This is not an uncommon way to see data organized, as we saw this in the Tuberculosis data set as well. Here there are two primary variables of interest, income and religion. And so income is placed along one axis (in this case columns) while religion is placed along the other (rows). It's a very intuitive way of putting data, but it is not tidy!

Questions

8. Among those surveyed, which income level had the most responses among Catholics?
9. What command would you use to tidy the data?

UN Migrant stock total

Let's go back to the UN now, and consider the number of migrants by country.

- Download the spreadsheet `UN_MigrantStockTotal_2015.xlsx` from the website and place it in your working directory.
- Open the file using a viewer that can read `.xlsx` files. Move to Table 1. Note that the first fourteen lines are given over to a picture, title of the report, and a copyright notice.

- In lines 15 and 16, some of the columns contain variable names, while other contain three sets of the years from 1990 through 2015 (by 5 year intervals.) By clicking on cell F15, you see it reads “International migrant stock at mid-year (both sexes)”. Cell L14 gives male migrant stock, and R14 gives female migrant stock.
- Blank entries have two dots (..) in them. So we have to be sure to tell the reader to treat these types of entries as NA.
- Therefore when we load it in, we must make sure that we eliminate the first fifteen rows. First, make sure the package `readxl` is installed. This package is considered part of the tidyverse, but is not one of the core packages that is automatically read in with the tidyverse. The current version of `readxl` is 1.3.0. If you are using an older version of the package, then some of the commands below might not work!

```
# install.packages("readxl")
library(readxl)
```

Now we load in the sheet labeled “Table 1”, skipping the first 15 rows.

```
ms <- read_excel("UN_MigrantStockTotal_2015.xlsx", sheet = "Table 1", skip = 15)
```

```
## New names:
## * ` ` -> `..1`
## * ` ` -> `..2`
## * ` ` -> `..3`
## * ` ` -> `..4`
## * ` ` -> `..5`
## * ... and 18 more
```

```
ms
## # A tibble: 265 x 23
##   ..1 ..2 ..3 ..4 ..5 `1990..6` `1995..7` `2000..8`
##   <dbl> <chr> <chr> <dbl> <chr> <dbl> <dbl> <dbl>
## 1 1 WORLD <NA> 900 <NA> 152563212 160801752 1727033
## 2 2 Deve~ (b) 901 <NA> 82378628 92306854 1033753
## 3 3 Deve~ (c) 902 <NA> 70184584 68494898 693279
## 4 4 Leas~ (d) 941 <NA> 11075966 11711703 100778
## 5 5 Less~ <NA> 934 <NA> 59105261 56778501 592441
## 6 6 Sub~-~ (e) 947 <NA> 14690319 15324570 137165
## 7 7 Afri~ <NA> 903 <NA> 15690623 16352814 148003
## 8 8 East~ <NA> 910 <NA> 5964031 5022742 48447
## 9 9 Buru~ <NA> 108 B R 333110 254853 1256
```

```
## 10      10 Como~ <NA>      174 B          14079      13939      137
## # ... with 255 more rows, and 15 more variables:
## #   `2005..9` <dbl>, `2010..10` <dbl>, `2015..11` <dbl>,
## #   `1990..12` <dbl>, `1995..13` <dbl>, `2000..14` <dbl>,
## #   `2005..15` <dbl>, `2010..16` <dbl>, `2015..17` <dbl>,
## #   `1990..18` <dbl>, `1995..19` <dbl>, `2000..20` <dbl>,
## #   `2005..21` <dbl>, `2010..22` <dbl>, `2015..23` <dbl>
```

- Oops, we see some problems right away. First, the first column is redundant, it only records the line of the file. More seriously, we don't want all the years, we only want to study the male set of years for now. The new standard for excel files is to end each variable name with `..` followed by the number of the column in the original file. This makes it easy for `select` to pick out specific columns. The following picks out columns 2, 4, and 12 through 17.

```
ms2 <- ms %>% select(`..2`, `..4`, `1990..12`:`2015..17`)
```

- We're now in better shape. Let's get the first two variables named properly with the `rename` command.

```
ms3 <- ms2 %>% rename(Area = `..2`, Country_code = `..4`)
```

Questions

10. What would the command have been if we had wanted to rename the `..2` variable `Region`?

- Now let's take the year variables and turn them into entries.

```
ms4 <- ms3 %>% gather(`1990..12`:`2015..17`, key = year, value = )
```

- So far so good, but now the year has the extra column information hanging off of it. Let's get rid of it with `separate` and `select`.

```
ms5 <- ms4 %>% separate(year, into = c("year", "excelcol")) %>%
ms5
```

- That works, but leaves the year as a character string. Let's fix that by using setting the `convert` parameter to `true` in `separate`.

```
ms5 <- ms4 %>% separate(year, into = c("year", "excelcol"), con
ms5
```

- Let's indicate that these are the numbers for male migrants.

```
ms6 <- ms5 %>% mutate("gender" = "male")
ms6
```

- Is this data tidy at this point? Well, yes, and no. You see, some of the “observations” are actually regions such as Eastern Africa rather than countries. So technically we should have a `region` variable, a `continent` variable, and a `developed` variable. The country codes for these regions are all 900 or later, so for now to get a tidy data set we simply remove these fake observations.

```
ms7 <- ms6 %>% filter(Country_code < 900)
ms7
```

- The data is now tidy! Of course, we did not have to use all the intermediary variables, we could have just done this in one fell swoop:

```
ms_tidy <- ms %>%
  select('..2', '..4', '1990..12':'2015..17') %>%
  rename(Area = '..2', Country_code = '..4') %>%
  gather('1990..12':'2015..17', key = year, value = migrants) %>%
  separate(year, into = c("year", "excelcol")) %>% select(-excelcol)
  mutate("gender" = "male") %>%
  filter(Country_code < 900)
ms_tidy
```

```
## # A tibble: 1,392 x 5
##   Area          Country_code year  migrants gender
##   <chr>          <dbl> <chr>    <dbl> <chr>
## 1 Burundi         108 1990    163267 male
## 2 Comoros          174 1990     6717 male
## 3 Djibouti         262 1990    64242 male
## 4 Eritrea           232 1990     6228 male
## 5 Ethiopia         231 1990   607284 male
## 6 Kenya           404 1990   160852 male
## 7 Madagascar       450 1990    13348 male
## 8 Malawi            454 1990   546520 male
## 9 Mauritius         480 1990     1763 male
## 10 Mayotte          175 1990     8780 male
## # ... with 1,382 more rows
```

- Remember that one of our goals is to make our analysis process as transparent as possible. By keeping a record of how we tidied the data, we allow ourselves the possibility of improvement in the future, or for others to collaborate with us more easily.

Test your knowledge (try if you have time)

11. Now try creating a variable that tidies the data for female migrants as well as the table for male migrants.

Chapter 33

Exploration: Relational data in the tidyverse

Summary

In this lab you will be taking a look at drawing data together from more than one table.

Bringing data together

- We'll start with a small toy data set that describes the band members of the Beatles and Rolling Stones. They are included as part of the package `dplyr`. So let's load in the tidyverse to start.

```
library(tidyverse)
```

```
## -- Attaching packages -----  
  
## v ggplot2 3.1.0      v purrr  0.2.5  
## v tibble  2.0.1      v dplyr  0.7.8  
## v tidyr   0.8.2      v stringr 1.3.1  
## v readr   1.3.1      v forcats 0.3.0  
  
## -- Conflicts -----  
## x dplyr::filter() masks stats::filter()  
## x dplyr::lag()    masks stats::lag()
```

There are three tables, `band_members`, `band_instruments`, and `band_instruments2`.

Questions

1. Take a look at `band_members`. Recall that a *key* is a single variable or set of variables that once the values are known, the observation is known. What variable or set of variables form a key for this set?

2. Which variables do not form a key for this data table?

Checking keys

- To double check your answers above, let's count the number of times each value appears in each variable:

```
band_members %>% count (name)
```

```
## # A tibble: 3 x 2
##   name      n
##   <chr> <int>
## 1 John      1
## 2 Mick      1
## 3 Paul      1
```

```
band_members %>% count (band)
```

```
## # A tibble: 2 x 2
##   band      n
##   <chr> <int>
## 1 Beatles  2
## 2 Stones   1
```

Questions

3. For a variable to be a key, what is the largest the variable `n` can be in the `count(variablename)` result?

Mutating joins

- Recall that a *mutating join* adds variables from one table to another. Let's try the four types of mutating joins with these tables. First the *inner join*.

```
band_members %>% inner_join(band_instruments)

## Joining, by = "name"
## # A tibble: 2 x 3
##   name band    plays
##   <chr> <chr>   <chr>
## 1 John  Beatles guitar
## 2 Paul  Beatles bass
```

Question

4. What variable was used as the foreign key for `band_members`?

5. Why wasn't Keith included in the `inner_join`?

- The next three joins are all types of *outer joins*. First the left join:

```
band_members %>% left_join(band_instruments)

## Joining, by = "name"
## # A tibble: 3 x 3
##   name band    plays
##   <chr> <chr>   <chr>
## 1 Mick  Stones <NA>
## 2 John  Beatles guitar
## 3 Paul  Beatles bass
```

- Next the right join:

```
band_members %>% right_join(band_instruments)

## Joining, by = "name"
## # A tibble: 3 x 3
##   name band    plays
##   <chr> <chr>   <chr>
## 1 John  Beatles guitar
## 2 Paul  Beatles bass
## 3 Keith <NA>   guitar
```

Questions

6. Explain why there is a missing value in the `right_join` table.

- Now try the full join:

```
band_members %>% full_join(band_instruments)

## Joining, by = "name"
## # A tibble: 4 x 3
##   name band plays
##   <chr> <chr> <chr>
## 1 Mick Stones <NA>
## 2 John Beatles guitar
## 3 Paul Beatles bass
## 4 Keith <NA> guitar
```

Filtering joins

- The *filtering joins* do not add variables to a table, rather, they select based on the presence or absence of the variable in the other table. For instance, consider the `semi_join`:

```
band_members %>% semi_join(band_instruments)

## Joining, by = "name"
## # A tibble: 2 x 2
##   name band
##   <chr> <chr>
## 1 John Beatles
## 2 Paul Beatles
```

Questions

7. Are the variables in the `semi_join` taken from the left table or the right table?

8. Why did Mick not appear in the `semi_join`?

- The next type of filtering join is an *anti join*. Let's try this out:

```
band_members %>% anti_join(band_instruments)
```

```
## Joining, by = "name"
```

```
## # A tibble: 1 x 2
```

```
##   name band
```

```
##   <chr> <chr>
```

```
## 1 Mick Stones
```

9. What table would be the union of the observations in the `semi_join` and the `anti_join` tables?

Changing variable names

- Now the variable `band_instruments2` contains the same information as `band_instruments`, but the first variable name is now different. To successfully use our joins on this table, we need to tell R what variables to compare.

```
band_members %>% full_join(band_instruments2, by = c("name" = "name"))
```

```
## # A tibble: 4 x 3
```

```
##   name band plays
```

```
##   <chr> <chr> <chr>
```

```
## 1 Mick Stones <NA>
```

```
## 2 John Beatles guitar
```

```
## 3 Paul Beatles bass
```

```
## 4 Keith <NA> guitar
```

Questions

10. Suppose we had used `by = c("band" = "artist")` in the above command by mistake. How many missing values would appear in the result?

Working with non keys

- The join functions will still run when we do not use keys, but you will end up with many more rows. Because the functions will not know which value is the ‘right’ observation, it is forced to include all possibilities. Enter the following tables:

```
t1 <- tibble(name = c('AZI-3', '2-1B', 'R2-Q5', 'AZI-3'),
             occupation = c('med', 'surgery', 'astromech', 'pro
t2 <- tibble(name = c('AZI-3', 'R2-Q5'), location = c('Kamino',
```

- Now consider what happens when we use name to add location data to t1:

```
t1 %>% full_join(t2, by = "name")
```

Questions

11. How many times does AZI-3 appear in the join?
12. Now suppose that AZI-3 appears twice in the data table we are joining:

```
t3 <- tibble(name = c('AZI-3', 'AZI-3', 'R2-Q5'), location = c(
```

How many times does AZI-3 appear in the full join of t1 and t3 by name?

Darwin's Finch data

During his famous trip to the Galápagos Islands, Darwin recorded the presence or absence of several species of finches during his trip. The file `darwins_finches.xlsx` records this data.

- Download `darwins_finches.xlsx` into the working directory.
- Load Sheet1 from this table into a variable `finch`.

```
library(readxl)
finch <- read_excel("darwins_finches.xlsx", skip = 1)
```

- Load Sheet2 from this table into a variable `island.names`.

```
island.names <- read_excel("darwins_finches.xlsx", sheet = "Sheets")
```

- Tidy the data from `finch`, into a table `finch_tidy`. In `finch_tidy` there should be a new variable `island.codes`.

```
finch_tidy <- finch %>% gather(key = island.codes, value = presence)
```

- Use a mutating join to create a new tibble `finch_names` which has both the presence/absence data and the names of the island in it.

```
finch_name <- finch_tidy %>% left_join(island.names, by = c("island.codes", "island.names"))
```

Question

13. Give a command to find out how many species of finch are found on each island.

Flights from the New York area

Now let's try some of these ideas with a real data set.

- Begin by loading in the `nycflights13` package.

```
library(nycflights13)
```

- One of the data tables in the `nycflights13` package is `weather`, which contains the temp, dewpoint, humidity, and `wind_dir` for each hour of every day in 2013 recorded at Newark Airport.

Questions

14. Add the data from the `weather` table to the `flights` table to create a new table `flighttemp`.
15. Give a command to find the mean temperature of the flights by month.
16. Plot these temperatures as bars with a different color for each month.

Chapter 34

Exploration: Working with strings and stringr

Summary: Strings

- `str_view` shows in the Viewer panel the result of a match.
- `str_subset` only keeps those strings where there is a match.
- `str_extract` pulls out the match from the string.
- `str_match` pulls out matches to each regular expression in parentheses inside the larger regular expression.
- `str_split` splits strings based upon matches in the regular expression.
- The helper function `boundary("words")` can be useful in taking out words from sentences.
- Wildcards and repetition symbols can greatly expand the ability to create new patterns.
- `regex` is implicitly called by many `str_` functions. When you call it explicitly, you can use parameters to give much greater power for how the string is transformed to a regular expression.
- Globbs are a type of pattern typically used for filename matching. `glob2rx` can convert a glob to a regular expression.

Viewing strings

- The `stringr` package is part of the tidyverse.

```
library(tidyverse)
```

- The `str_view` command gives us the ability to see all the matches highlighted in a vector of strings. Try the following.

```
s1 <- c("abc", "bcd", "cde")
str_view(s1, "b")
```

Escape characters

- Escape characters are tricky to use in regular expressions. For instance, to find a left parenthesis, we first must use the escape formulation `\(`. But if we pass `\(` as part of a string to a regular expression, it will just pass the `(` and not the escape part!
- So we need to first pass a backslash, and then put a left parenthesis. The escape character for a backslash is `\\`, which means the regular expression becomes `\\(`. For instance,

```
s2 <- "(2 + (8 + x))*3"
str_view(s2, "\\(")
```

- Note that it only located the first left parenthesis in the string, and ignored the second one. In order to highlight all of the matches, we can use `str_view_all`.

```
s2 <- "(2 + (8 + x))*3"
str_view_all(s2, "\\(")
```

- Many of the functions in `stringr` come in pairs in this fashion: `str_function` for doing something on the first match, and `str_function_all` for doing the same thing on all matches in the string.

Questions - strings

1. Give a command to view all the dollar signs in a string?

Wildcard characters

- Some characters are *wildcards*. In card games, when you designate a card or group of cards *wild*, that means that you can substitute the card for any other card in the deck. In regular expressions wildcard characters stand for certain character in the string. For instance, `\d` will match any digit.

```
str_view_all(s2, "\\d")
```

- You can form your own wildcards using square brackets []. If you put characters in square brackets, they will match any of the characters in the brackets, Try

```
str_view_all(s2, "[x23\\+]" )
```

- If we use a – in brackets, it matches any character *between* the endpoints, including the endpoints themselves. Try

```
str_view_all(s2, "[2-7]" )
```

- If you use a dot ., that matches any symbol. Try

```
str_view_all(s2, "\\d.")
```

- This matches any digit that has a character following it. Note that the character which follows becomes part of the match.

Questions - strings

2. What regular expression matches a left parenthesis followed by a digit?

str_extract, str_extract_all, and str_subset

- We can use `str_extract` to return strings which have the expression in them. The regular expression “w.” matches the letter w followed by any character. Try

```
r1 <- c("white", "red", "willow", "owl", "few", "tough")
str_match(r1, "w.")
```

- The NA values indicate that no match was found. In the third string “willow”, it found the first match “wi” but then quit before finding “wy”. As with many of the string commands, use the `_all` formation to get all matches.

```
str_extract_all(r1, "w.")
```

- The character (0) is just another way of saying the empty string “”. We can eliminate these by only considering strings with a match using `str_subset`.

```
rg1 <- "w."
r1 %>% str_subset(rg1) %>% str_extract_all(rg1)
```

- We can put the results in a matrix rather than a list by using `simplify = TRUE`. Try

```
r1 %>% str_subset(rg1) %>% str_extract_all(rg1, simplify =
```

Questions - strings

3. What command would extract matches to the letter e followed by any letter from a to z?

str_match and *str_match_all*

- Sometimes we want the string that matches together with the individual pieces that made up the match. That is what `str_match` is for. The canonical application is phone numbers. First let's get some examples of phone numbers

```
r2 <- c("202 456-1111", "202-224-3121", "(909) 621-8088")
```

- Now we want a regular expression that finds the area code, the three digit local code, and the last four numbers for our problem.

```
rg2 <- "([2-9][0-9]{2})[-. ]([0-9]{3})[-. ]([0-9]{4})"
```

- Notice there are three expressions in parenthesis. So `str_match` will create a matrix with four columns, the last three columns correspond to the wildcard matches for the three parenthesis. The `{2}` means the last pattern should repeat 2 times. There is also `+` which means repeat 1 or more times, `?` which means repeat 0 times or 1 time, and `*` (the **Kleene star**) which means repeat 0 or more times.

```
r2 %>% str_match(rg2)
```

- Note that our third phone number doesn't match the pattern. We would need a more advanced regular expression to deal with this type of input.

Questions - strings

4. How many columns will the matrix resulting from regular expression $([a-z]^+)-([A-Z]^+)$ have?

Anchor symbols

- The \wedge symbol in regular expression matches only the beginning of words. It *anchors* the pattern to the beginning. To get all strings in the `words` variable that begin with the letter "w", try

```
rg3 <- "^w"
words %>% str_subset(rg3) %>% str_view(rg3)
```

- To get words where the end matches, use a dollar sign. Try

```
rg5 <- "w$"
words %>% str_subset(rg5) %>% str_view(rg5)
```

- To match both the beginning and the end of the string, we need to use both.

```
rg5 <- "^w[a-z]*w$"
words %>% str_subset(rg5) %>% str_view(rg5)
```

Questions - strings

5. How many words in the `words` variable end in `ay`?
6. What words in the `words` variable start with the letter `m` and end with `g`?

Splitting strings

- Consider the following string:

```
x1 <- c("This is a string.", "Another string.")
```

- This string can be split into multiple pieces using `str_split`.

```
str_split(x1, pattern = " ")
```

```
## [[1]]
## [1] "This"      "is"         "a"          "string."
##
## [[2]]
## [1] "Another" "string."
```

- Note that this creates a *list*, which is a combination of data types of different lengths. (Contrast with a *tibble* or *data frame*, where each observation comes from the same Cartesian product $A_1 \times \dots \times A_n$.) You access elements of a list using double brackets.

```
str_split(x1, pattern = " ")[[1]]
```

```
## [1] "This"      "is"         "a"          "string."
```

- Alternatively, we can split the string into a matrix by setting `simplify = TRUE` in the call to `str_split`.

```
str_split(x1, pattern = " ", simplify = TRUE)
```

```
##      [,1]      [,2]      [,3] [,4]
## [1,] "This"    "is"      "a"   "string."
## [2,] "Another"  "string." ""    ""
```

- Note that for the shorter strings, null strings are used to fill out the columns of the matrix. By default the pattern is treated as a regular expression. For example, try

```
x3 <- "$43.25 $56.25 $4.03"
str_split(x3, pattern = "\\$")
```

```
## [[1]]
## [1] ""          "43.25" "56.25" "4.03"
```

- Note we had to use two backslashes to turn into a single backslash, which then combined with the dollar sign to form the character escape dollar sign, which is what we were looking for in the string.

Questions - strings

7. Give a command for separating “ab cd edf” using space bar as the separator symbol.

8. Give a command for separating “ab|cd|edf” using the vertical bar as the separator symbol. Remember that the vertical bar needs escape character `\|`, and that to get a `\` requires `\\` in the regular expression.

9. What would the command be to turn “ab|cd|edf” into a matrix? How many rows and columns does the resulting matrix have?
 - You can also set the maximum number of pieces that split breaks the string into. Consider:

```
fields <- c("Name: Huber: Mark", "Country: US: CA", "Age: 47")
fields %>% str_split(":", n = 2, simplify = TRUE)
```

```
##      [,1]      [,2]
## [1,] "Name"   "Huber: Mark"
## [2,] "Country" "US: CA"
## [3,] "Age"    "47"
```

- Note that after the first split since there are a max of two pieces, the remaining gets put all in the second piece regardless of the presence or absence of another `':'`. The helper function `boundary` can also be used to split strings. For instance, consider

```
x2 <- c("A string. Another string.")
str_split(x2, " ")
```

```
## [[1]]
## [1] "A"      "string." "" "Another" "string."
```

- With this split, the period (.) gets attached to the word it is ending. Often we are just interested in the word itself. This is usually the case when we are doing an analysis of a text. We can use `boundary("word")` to indicate that we wish to split the string into words.

```
str_split(x1, boundary("word"))
```

```
## [[1]]
## [1] "This" "is" "a" "string"
##
## [[2]]
## [1] "Another" "string"
```

Regular expressions

- Consider the fruit list of strings:

```
fruit
```

Questions - strings

10. How many fruits are listed in the `fruit` variable?

- We can search for the berries within the fruit using a *regular expression*. The simplest type of regular expression is just a string itself.

```
str_detect(fruit, "berry")
```

```
## [1] FALSE FALSE FALSE FALSE FALSE TRUE TRUE FALSE FALSE
## [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE
## [23] FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE
## [34] FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE
## [45] FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE
## [56] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [67] FALSE FALSE FALSE TRUE FALSE FALSE TRUE FALSE FALSE
## [78] FALSE FALSE FALSE
```

- If you use a mathematical operation on TRUE/FALSE data, the TRUE values will be converted to 1 and the FALSE values will be converted to 0. (This is called the *indicator function*.) So for instance,

```
sum(str_detect(fruit, "berry"))
```

```
## [1] 14
```

tells us the number of fruits in the list with “berry” in them.

The function `mean` applied to data that is 0 or 1 is the sum of the data (aka the numbers of 1’s) divided by the number of data points, and so gives the percentage of 1’s in the set. Applied to TRUE/FALSE vectors, it gives us the percentage of true answers.

Questions

11. What percentage of terms in the list of strings `sentences` contains the word “the” at least once?

Transforming other pattern types to regular expressions

- When you pass a string along as the regular expression, it actually goes through a helper function `regex` by default. Try

```
sum(str_detect(fruit, "berry"))
```

and

```
sum(str_detect(fruit, regex("berry")))
```

to verify that they return the same result.

- If you want to use the string as is rather than a regular expression, you can use `fixed`. This is especially helpful when dealing with characters that otherwise require escape characters. For instance, try

```
x3
str_split(x3, pattern = regex("\\$"))
str_split(x3, pattern = fixed("$"))
```

Questions - strings

12. Give a command for separating "ab|cd|edf" using the vertical bar as the separator symbol using the `fixed` helper function.
 - Aside from avoiding complicated backslash expressions, the primary purpose of using `fixed` is for speed: because it concentrates on finding a particular case of a regular expression, `fixed` can be quite a bit faster than using `regex`.

Using `regex` explicitly

- By calling `regex` explicitly, it is possible to use parameters to modify the way in which it transforms patterns into regular expressions. For instance, try

```
x4 <- "Test 1\nTest 2\nTest 3\n"
cat(x4)
str_extract_all(x4, "^Test")[[1]]
```

- Even though `x` as a string contains four lines, a search for "Test" only finds one instance. By setting the parameter `multiline = TRUE` in an explicit call to `regex`, we can force `str_extract` to treat each line separately.

```
str_extract_all(x4, regex("^Test", multiline = TRUE))
```

Questions - strings

13. Suppose the variable `x5` is given by:

```
x5 <- "the heart\nthen brain\na spleen and the kidney\n"
```

Give a command to locate all the sentence fragments that start with "the".

- When a regular expression becomes long, it can be difficult to parse. The basic regular expressions in R do not have a comment character, but by using the parameter `comments = TRUE`, we can use the `#` character to mark off comments that are not part of the regular expression. Consider the following regular expression for reading a phone number.

```

phone <- regex("
  \\(?:      # optional opening parens
  (\\d{3})  # area code
  [ ] -]?   # optional closing parens, space, or dash
  (\\d{3})  # another three numbers
  [ -]?    # optional space or dash
  (\\d{3})  # three more numbers
", comments = TRUE)

str_match("514-791-8141", phone)
str_match("(514)791-8141", phone)

```

Questions - strings

14. Give three more forms of the phone number above that match the regular expression.

Globs

- The vector of strings that consists of all the filenames in the working directory is created by the `dir` command. Filenames have their own pattern matching methods that are very different from regular expressions. These are called *globs*. A *glob* is a pattern that specifies filename strings. In particular, `*` is often the wildcard character in a glob.
- So for instance, `*.Rmd` is a glob that matches all filenames that end in `.Rmd`. If you want to use a glob pattern with parameter `pattern`, the function `glob2rx` converts a glob pattern to a regular expression.
- For instance, the following locates all files that end in `.Rmd`.

```
head(dir(pattern = glob2rx("*.Rmd")))
```

Questions - strings

15. What command finds all files that end in `.R`?

Chapter 35

Exploration: MySQL

To practice working with SQL queries, we will be using a relational dataset that is contained in the Relational Dataset Repository. We will be accessing this using a dialect of SQL called **MySQL**. This is an open source version of SQL.

Despite how it looks, the My in MySQL is not the English word My, instead one of the creators named it after his oldest daughter whose name is My. Later, Oracle bought the trademark MySQL, and so to keep the project open it was renamed to MariaDB because Maria is the name of another daughter. The DB was added because there also used to be a storage system named Maria. This storage system is now named Aria to avoid confusion.

Anyway, the point of this is that in order for R to use MySQL commands, we load in a package called `RMariaDB`.

```
# install.packages("RMariaDB")
library(RMariaDB)
```

Accessing SQL tables remotely

Now we can form a database connection to the Relational Dataset Repository. This connection needs a host name (given by a URL), a username, a password, and a *port*, which allows the data to be sent. Fortunately the site has a guest account set up that anyone can access.

```
# Connect to my-db as defined in ~/.my.cnf
con <- dbConnect(RMariaDB::MariaDB(),
  host = "relational.fit.cvut.cz",
  username = "guest",
  password = "relational",
  port = "3306",
  dbname = "northwind")
```

In your Global Environment, you should at this point have a new variable `con` which is of type Formal class `MariaDBConnection`.

To list the tables that are part of this relational database, use the following command:

`dbListTables` (con)

The Northwind database is a fictional set of data for Northwind Trading Co. intended to learn SQL commands.

Questions

1. How many tables are there in the Northwind database?

Up until now, all of our code chunks start with three backticks: ```` followed by `{r}`. The `{r}` tells R Markdown to use R to evaluate the code. If instead we use `{sql, connection = con}`, then this tells R to treat the code chunk as an SQLite query to the database defined in `con`.

Much of this part of the lab is based upon a tutorial available at <https://www.webucator.com/tutorial/learn-sql/index.cfm>.

Start with the following query

```
SELECT *
FROM Employees
```

Whitespace does not matter in SQL queries, so an equivalent command is:

```
SELECT * FROM Employees
```

We format the commands the way we do simply to make them easier to read.

We can also send SQL queries in R using the function `dbSendQuery` command to send the query to the database, and then the `dbFetch` command to get the results of the query back to the user.

```
df <- dbSendQuery(con, "SELECT * FROM Employees")
dbFetch(df)
```

Questions

2. How many employees are there in the company?
3. How many variables are there in the relation `Employees`?

To select only the `LastName` and `TitleofCourtesy` factors, try

```
SELECT LastName, TitleofCourtesy
FROM Employees
```

To order by last name we can use the `ORDER BY` keyword:

```
SELECT LastName, TitleofCourtesy
FROM Employees
ORDER BY LastName
```

Once you have used `SELECT` to create column names, you can also order by the columns by number:

```
SELECT LastName, TitleofCourtesy
FROM Employees
ORDER BY 2
```

The keyword `DESC` reverses the order:

```
SELECT LastName, TitleofCourtesy
FROM Employees
ORDER BY 2 DESC
```

Question

4. Which employee comes first when ordered by first name?

WHERE

`WHERE` allows us to pick out observations that meet certain criteria.

```
SELECT Title, FirstName, LastName
FROM Employees
WHERE Title = "Sales Representative"
```

Table 35.1: 6 records

| Title | FirstName | LastName |
|----------------------|-----------|-----------|
| Sales Representative | Nancy | Davolio |
| Sales Representative | Janet | Leverling |
| Sales Representative | Margaret | Peacock |
| Sales Representative | Michael | Suyama |
| Sales Representative | Robert | King |
| Sales Representative | Anne | Dodsworth |

Use `<>` for does not equals, so to get all the non-Sales Representatives, try:

```
SELECT Title, FirstName, LastName
FROM Employees
WHERE Title <> "Sales Representative"
```

Table 35.2: 3 records

| Title | FirstName | LastName |
|--------------------------|-----------|----------|
| Vice President, Sales | Andrew | Fuller |
| Sales Manager | Steven | Buchanan |
| Inside Sales Coordinator | Laura | Callahan |

Ordering note: The `WHERE` command must precede the `ORDER BY` command in an SQL query.

Concatentation

Suppose that we have multiple factors that contain string values that we wish to bring together. In the tidyverse, we used `unite` for this purpose, but in MySQL, we use the `CONCAT` keyword. Try:

```
SELECT CONCAT(FirstName, " ", LastName) AS Name
FROM Employees
```

Table 35.3: 9 records

| Name |
|------------------|
| Nancy Davolio |
| Andrew Fuller |
| Janet Leverling |
| Margaret Peacock |

 Name

Steven Buchanan
 Michael Suyama
 Robert King
 Laura Callahan
 Anne Dodsworth

Arithmetic

The usual arithmetic commands apply. Suppose that if freight has a cost at least \$500.00, then it is taxed at 10%. The following gives the taxed freight amount.

```
SELECT OrderID, Freight AS `Freight Cost`, Freight * 1.1 AS `Taxed Freight`
FROM Orders
WHERE Freight >= 500
```

Questions

- Suppose that freight with cost over \$1000 is taxed at 12%. Create a query to find a table with the taxed freight cost that only contains those observations where the tax applies.

Grouped data

We can use COUNT, SUM, AVG, MIN and MAX to analyze data.

```
SELECT MAX(freight) AS max_freight,
       MIN(freight) AS min_freight,
       AVG(freight) AS avg_freight
FROM Orders
```

Table 35.4: 1 records

| max_freight | min_freight | avg_freight |
|-------------|-------------|-------------|
| 1007.64 | 0.02 | 78.2442 |

We can find the total number of employees with COUNT:

```
SELECT COUNT(EmployeeID) AS num_emp
FROM Employees
```

By grouping employees, any function applied to them will be applied group by group. For instance, to count the number of employees from each city:

```
SELECT City, COUNT(EmployeeID) AS num_emp
FROM Employees
GROUP BY City
```

Table 35.5: 5 records

| City | num_emp |
|----------|---------|
| Kirkland | 1 |
| London | 4 |
| Redmond | 1 |
| Seattle | 2 |
| Tacoma | 1 |

Questions

6. Create an SQL query that returns for each product in the `Order Details` table, the number of times that product was ordered. Note that for tables names like this with a space in them, you must surround the name by backticks (‘) to indicate that it is all one name.

To filter observations by group, use the `HAVING` keyword. For instance, to find all the cities which have more than one employee:

```
SELECT City, COUNT(EmployeeID) AS num_emp
FROM Employees
GROUP BY City
HAVING COUNT(EmployeeID) > 1
```

Table 35.6: 2 records

| City | num_emp |
|---------|---------|
| London | 4 |
| Seattle | 2 |

Note that there is a specific order that keywords must have in an SQL query: 1. SELECT 2. FROM 3. WHERE 4. GROUP BY 5. HAVING 6. ORDER BY

So WHERE filters observations before the GROUP BY, while HAVING filters afterwards. So in order to find the number of cities employing more than 1 Sales Representative:

```
SELECT City, COUNT(EmployeeID) AS num_emp
FROM Employees
WHERE Title = "Sales Representative"
GROUP BY City
HAVING COUNT(EmployeeID) > 1
```

Table 35.7: 1 records

| City | num_emp |
|--------|---------|
| London | 3 |

Any time you put the keyword DISTINCT in front of a factor name, it only returns the levels of that factor, that is, distinct values that the factor takes on. You can use this to count, for instance, the number of times each city appears in a table.

```
SELECT COUNT(DISTINCT City) AS num_city
FROM Employees
```

Subqueries

So far we've just worked with one query. But you can use a query within a query to further refine a query. For instance, the following command finds the CustomerID associated with OrderID 10290.

```
SELECT CustomerID
FROM Orders
WHERE OrderID = 10290
```

Table 35.8: 1 records

| CustomerID |
|------------|
| COMMI |

Now suppose we want the name of the company with that CustomerID. We could use this result within a WHERE query to find it:

```

SELECT CompanyName
FROM Customers
WHERE CustomerID = (SELECT CustomerID
                    FROM Orders
                    WHERE OrderID = 10290)

```

Table 35.9: 1 records

| CompanyName |
|-----------------|
| Comrcio Mineiro |

In order for this type of construction to be valid, the subquery must return a single column.

Questions

7. What is the name of the contact at the company that placed order with ID number 10292?

Joins

Another way to bring data from one table to another is by using *joins*. For instance, suppose we have the employee and order ID from each order.

What we would like is the *name* of each employee that gives that order. So we join the name variable from the `Employees` table to the `Orders` table. Since we are working with more than one table, to indicate a factor we use the form `. to do this`.

```

SELECT Employees.EmployeeID, Employees.FirstName,
       Employees.LastName, Orders.OrderID, Orders.OrderDate
FROM Employees
JOIN Orders ON (Employees.EmployeeID = Orders.EmployeeID)
ORDER BY Orders.OrderDate

```

Using aliases for the table names can make queries more readable:

```

SELECT emp.EmployeeID, emp.FirstName,
       emp.LastName, ord.OrderID, ord.OrderDate
FROM Employees AS emp
JOIN Orders AS ord ON (emp.EmployeeID = ord.EmployeeID)
ORDER BY ord.OrderDate

```

By default, a join in SQL is an *inner join*. For *outer joins* we explicitly put the word `OUTER` in front of the `JOIN` keyword.

```
SELECT COUNT(DISTINCT e.EmployeeID) AS numEmployees,
       COUNT(DISTINCT c.CustomerID) AS numCompanies,
       e.City, c.City
FROM Employees AS e
     LEFT JOIN Customers c ON (e.City = c.City)
GROUP BY e.City, c.City
ORDER BY numEmployees DESC
```

Unions

We can use `UNION` to combine reports as long as they have the same variables. For instance, if we want the name and phone number of all our shippers, customers, and suppliers, we could use `UNION` to get it.

```
SELECT CompanyName, Phone
FROM Shippers
UNION
SELECT CompanyName, Phone
FROM Customers
UNION
SELECT CompanyName, Phone
FROM Suppliers
ORDER BY CompanyName
```

Questions

8. Create a report showing the contact name and phone numbers for all employees, customers, and suppliers.

Chapter 36

Exploration: Modeling Data

Summary

In this lab you will learn how to model data using the base R commands and the `modelr` package.

Source

The content of this lab comes from Chapter 24 on Model building from *R for Data Science* by Wickham and Golemund (<https://r4ds.had.co.nz/>).

Modeling data

- Start by loading in the `modelr` library (installing the package first if necessary.)

```
# install.packages("modelr")  
library(modelr)
```

- We'll also then add the `tidyverse`.

```
library(tidyverse)
```

Diamond prices

- Consider the dataset `diamonds` that is built in to the `tidyverse`. We can get a look at the price versus quality through a boxplot approach. Try the following.

```
ggplot(diamonds, aes(cut, price)) + geom_boxplot()  
ggplot(diamonds, aes(color, price)) + geom_boxplot()  
ggplot(diamonds, aes(clarity, price)) + geom_boxplot()
```

- Recall that the line in the middle of the boxplot is the median of the values.

Questions

1. What color has the highest median price?
2. What clarity has the highest median price?
 - You might be surprised to learn that color J is considered the worst color for a diamond (slightly yellow), and I₁ is considered the worst quality since it indicates that there exist inclusions visible to the naked eye.
 - This is a perfect example of where there is another variable that confounds our ability to predict price: the weight of the diamond, as measured by `carat`. The weight is simultaneously the most important factor in the price of the diamond, and poorer color diamonds also tend to be larger. We can visualize this with a hex plot.

```
diamonds %>% ggplot(aes(carat, price)) +
  geom_hex(bins = 50)
```

- The graph shows that as the carat increases, the price of the diamonds increase as well. How can we fit a linear model to this data? One thing to note is that if there is a polynomial relationship between y and x , then

$$y = c_0x^{c_1}$$

for constants c_0 and c_1 . If we take the logarithm base 2 of both sides, then

$$\lg(y) = \lg(c_0) + c_1 \lg(x).$$

In other words, if x and y have a *polynomial* relationship, then $\lg(x)$ and $\lg(y)$ have a *linear* relationship. Let's see if that holds here:

```
diamonds2 <- diamonds %>%
  mutate(lg_price = log2(price), lg_carat = log2(carat))
```

- Now let's graph the log data:

```
diamonds2 %>%
  ggplot(aes(lg_carat, lg_price)) +
  geom_hex(bins = 50)
```

- That looks much more linear!. Let's make a linear model out of it:

```
mod_diamond <- lm(lg_price ~ lg_carat, data = diamonds2)
coef(mod_diamond)
```

```
## (Intercept)    lg_carat
## 12.188841     1.675817
```

Questions

- Write the above fitted model in the form

$$y = c_0 x^{c_1}.$$

- At this point, the predictions that we get are for the linear (log-log) model. To overlay the original data on top of that, we need to take the inverse of the log base 2, which is raising 2 to the value.

```
diamonds3 <- diamonds2 %>%
  filter(carat <= 2.5)

grid <- diamonds3 %>%
  data_grid(carat = seq_range(carat, 20)) %>%
  mutate(lg_carat = log2(carat)) %>%
  add_predictions(mod_diamond, "lg_price") %>%
  mutate(price = 2 ^ lg_price)
```

Now for the actual plot.

```
diamonds3 %>% ggplot(aes(carat, price)) +
  geom_hex(bins = 50) +
  geom_line(data = grid, color = "red", size = 1)
```

- The model starts off strong, but as a certain point, the red prediction line rises above all known prices. This indicates that the model breaks down as the carat size grows past about 2.3.
- As usual, we graph the residuals to see if they show a pattern. First calculate the residuals.

```
diamonds3 <- diamonds3 %>%
  add_residuals(mod_diamond, "lg_resid")
```

Then plot the results.

```
diamonds3 %>% ggplot() +
  geom_hex(aes(lg_carat, lg_resid), bins = 50)
```

- Now let's go back to our original boxplots, this time with the residuals.

```
ggplot(diamonds3, aes(cut, lg_resid)) + geom_boxplot()
ggplot(diamonds3, aes(color, lg_resid)) + geom_boxplot()
ggplot(diamonds3, aes(clarity, lg_resid)) + geom_boxplot()
```

Questions

4. Which of the clarity classes has the lowest (worst) residuals?
5. Which of the color classes has the lowest (worst) residuals?
6. Which of the cut classes has the highest (best) residuals?

Including color, cut, and clarity in the model

- Now let's add the properties of color, cut, and clarity to the model.

```
mod_diamond2 <- lm(lg_price ~ lg_carat + color + cut + clar
  data = diamonds3)
```

We'll pass along the model to `data_grid` in order to get a good sampling of possibilities.

```
grid <- diamonds2 %>%
  data_grid(cut, .model = mod_diamond2) %>%
  add_predictions(mod_diamond2)
grid
```

```
## # A tibble: 5 x 5
##   cut      lg_carat color clarity  pred
##   <ord>      <dbl> <chr> <chr>   <dbl>
## 1 Fair      -0.515 G     VS2     11.2
## 2 Good      -0.515 G     VS2     11.3
## 3 Very Good -0.515 G     VS2     11.4
## 4 Premium   -0.515 G     VS2     11.4
## 5 Ideal     -0.515 G     VS2     11.4
```

- Now let's take a look at the residuals:

```
diamonds3 <- diamonds3 %>%
  add_residuals(mod_diamond2, "lg_resid2")
```

And let's plot them

```
diamonds3 %>% ggplot() +
  geom_hex(aes(lg_carat, lg_resid2), bins = 50)
```

- Overall we have a pretty good model at this point. However, there are still some cases where the log-residuals are either very large or very small, so it is not capturing all situations.

The flights data set

- Now let's consider an analysis of the `flights` data from the package `nycflights13`. First let's load in the data, and the package `lubridate` in order to handle the data entries.

```
library(nycflights13)
library(lubridate)
```

- Next let's break down the number of flights by date.

```
daily <- flights %>%
  mutate(date = make_date(year, month, day)) %>%
  group_by(date) %>%
  summarize(n = n())
daily
```

```
## # A tibble: 365 x 2
##   date      n
##   <date>   <int>
```

```
## 1 2013-01-01 842
## 2 2013-01-02 943
## 3 2013-01-03 914
## 4 2013-01-04 915
## 5 2013-01-05 720
## 6 2013-01-06 832
## 7 2013-01-07 933
## 8 2013-01-08 899
## 9 2013-01-09 902
## 10 2013-01-10 932
## # ... with 355 more rows
```

- We can graph the data to look for a pattern. First we create a new factor `weekday` based on the `weekday` using the `wday` function.

```
weekday <- daily %>%
  mutate(wday = wday(date, label = TRUE))
```

- Now we can look at a boxplot of number of flights by day of the week.

```
weekday %>% ggplot(aes(wday, n)) +
  geom_boxplot()
```

- We strongly see the effect of the weekend. Most fliers are traveling for business, so very few leave on a Saturday. Because this is categorical data, when we fit a model it will just use the mean of the data for the prediction. First fit the model and add the predictions.

```
mod <- lm(n ~ wday, data = weekday)
```

```
grid <- weekday %>%
  data_grid(wday) %>%
  add_predictions(mod, "n")
```

- Now we add the predictions to the model plot.

```
weekday %>% ggplot(aes(wday, n)) +
  geom_boxplot() +
  geom_point(data = grid, color = "red", size = 4)
```

- Now that we have predictions, we can look at the residuals to try and identify any remaining patterns that need to be modeled.

```
weekday <- weekday %>%
  add_residuals(mod)
```

- For the plot, try

```
weekday %>%
  ggplot(aes(date, resid)) +
  geom_ref_line(h = 0) +
  geom_line()
```

- There's definitely a pattern there! For one thing, there are some spikes in the data. Let's take a closer look at those.

```
weekday %>%
  filter(resid < -100)
```

```
## # A tibble: 11 x 4
##   date           n wday  resid
##   <date>       <int> <ord> <dbl>
## 1 2013-01-01     842 Tue   -109.
## 2 2013-01-20     786 Sun   -105.
## 3 2013-05-26     729 Sun   -162.
## 4 2013-07-04     737 Thu   -229.
## 5 2013-07-05     822 Fri   -145.
## 6 2013-09-01     718 Sun   -173.
## 7 2013-11-28     634 Thu   -332.
## 8 2013-11-29     661 Fri   -306.
## 9 2013-12-24     761 Tue   -190.
## 10 2013-12-25    719 Wed   -244.
## 11 2013-12-31    776 Tue   -175.
```

- You can see, July 4th (and July 5th) in the date, along with Thanksgiving, Christmas, and Christmas Eve. There also seems to be more flights in the summer in general and fewer in winter. We can use the `geom_smooth` function to give a local estimate for this behavior.

```
weekday %>%
  ggplot(aes(date, resid)) +
  geom_ref_line(h = 0) +
  geom_line(colour = "grey50") +
  geom_smooth(se = FALSE, span = 0.20)
```

- To get a better idea of what's going on, let's concentrate on the Saturday flights. First let's plot them over the course of the year.

```
weekday %>%
  filter(wday == "Sat") %>%
  ggplot(aes(date, n)) +
    geom_point() +
    geom_line() +
    scale_x_date(NULL, date_breaks = "1 month", date_labels = "%b %d")
```

- The pattern is clear—people fly much more in the summer months (perhaps because of school vacation), less in the Spring and even less in the Fall with a spike at the edges of Christmas vacation. Since things appear to be school driven, let's break up our data into Spring, Summer, and Fall. First we create a function that calculates the term:

```
term <- function(date) {
  cut(date,
      breaks = ymd(20130101, 20130605, 20130825, 20140101),
      labels = c("spring", "summer", "fall")
  )
}
```

- Next we apply it to our data.

```
weekday <- weekday %>%
  mutate(term = term(date))
```

- Now we can graph our data broken up by term.

```
weekday %>%
  filter(wday == "Sat") %>%
  ggplot(aes(date, n, color = term)) +
    geom_point(alpha = 1/3) +
    geom_line() +
    scale_x_date(NULL, date_breaks = "1 month", date_labels = "%b %d")
```

- To see if this new factor is useful, let's look at the box plots broken down by term.

```
weekday %>%
  ggplot(aes(wday, n, color = term)) +
  geom_boxplot()
```

- Definitely some term effects going on there. But does it help the model? Let's add in term and see how the model changes.

```
mod1 <- lm(n ~ wday, data = weekday)
mod2 <- lm(n ~ wday * term, data = weekday)
```

- Put the residuals from both models together.

```
weekday <- weekday %>%
  gather_residuals(without_term = mod1, with_term = mod2)
```

- Now plot them.

```
weekday %>%
  ggplot() +
  geom_line(aes(date, resid, color = model), alpha = 0.75)
```

- There's a bit of difference, but not as much as one might have hoped.

Questions

7. How would you create a boxplot as earlier for the new residuals, but with a facet for each term instead of a different color?

Fitting a spline

- In the last section, we used our knowledge of school terms to induce an extra factor in the data. As an alternative, we could use an automatic method to fit the data. One such approach uses *splines*. First load in the library

```
# install.packages("splines")
library(splines)
```

- Next let's fit a spline to the data. Instead of the basic `lm` (linear models) function, we will use `rlm` which stands for *robust linear models*. This uses a more advanced method of determining coefficients called an M estimator. It tends to ignore outliers automatically, so can be a useful tool for not letting days like the Fourth of July dominate our estimate. The `ns` function uses a natural spline to try to match what is happening across days.

```
mod <- MASS::rlm(n ~ wday * ns(date, 5), data = weekday)
```

- With that in place, let's go ahead and look at the predictions.

```
weekday <- weekday %>%  
  data_grid(wday, date = seq_range(date, n = 13)) %>%  
  add_predictions(mod, "pred")
```

```
weekday %>%  
  ggplot(aes(date, pred, color = wday)) +  
    geom_line() +  
    geom_point()
```

- This is part of the issue: we have a strong pattern on Saturday flights that seemingly is not replicated on the other days of the week where travel is more consistent.

Chapter 37

Exploration: Support vector machines with *svm*

Summary

In this lab you will learn how to classify data using a Support Vector Machine

Source

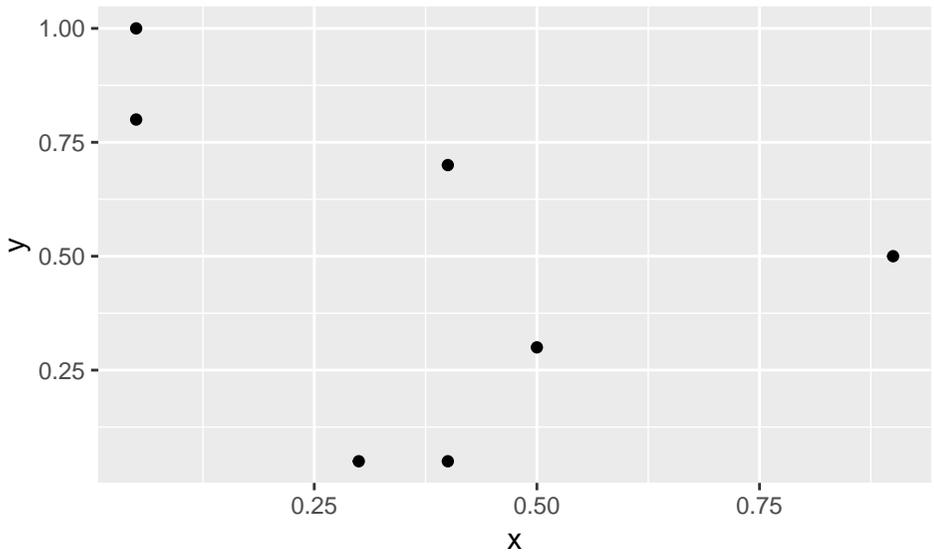
This lab is based upon a blog post at <https://www.datacamp.com/community/tutorials/support-vector-machines-r>.

Support Vector Machines

- *Support vector machine* or *svm* is an approach to supervised learning in order to classify data. Let's start simple. Suppose I have the following data set.

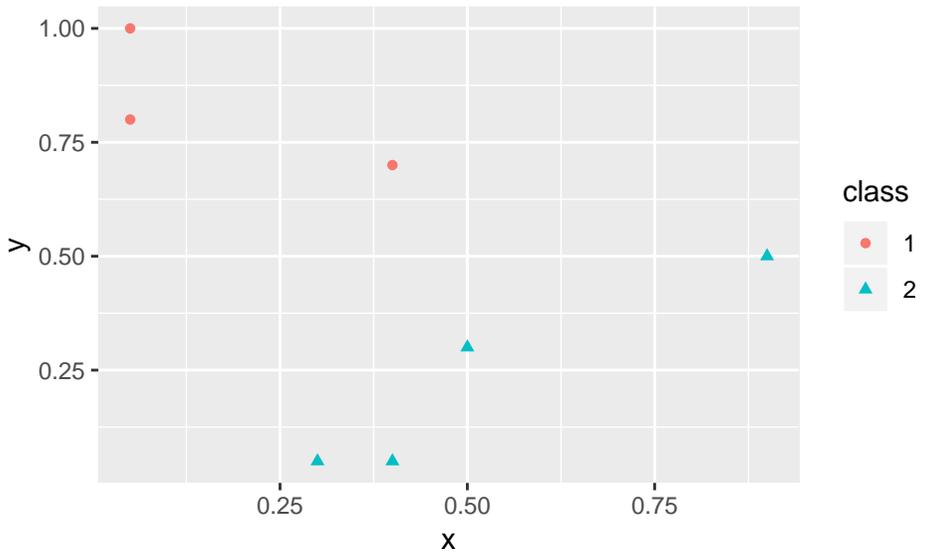
```
library(tidyverse)
df <- tibble(x = c(0.05, 0.4, 0.05, 0.9, 0.4, 0.5, 0.3),
             y = c(1, 0.7, 0.8, 0.5, 0.05, 0.3, 0.05))

df %>% ggplot() +
  geom_point(aes(x, y))
```



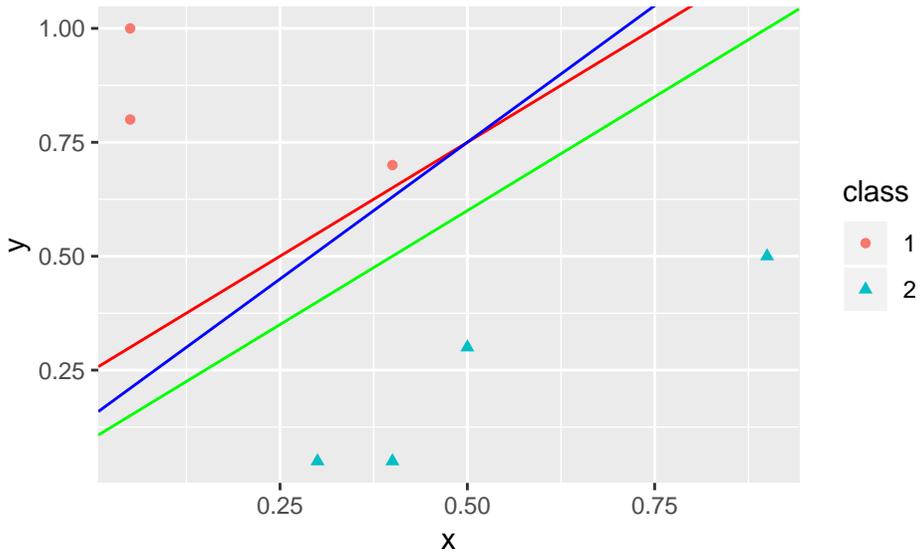
- In supervised learning, we are given labels for our data. So for instance, suppose that I know that the points in the upper left are of one type, and in the lower right are another.

```
df <- df %>%  
  mutate(class = factor(c(1, 1, 1, 2, 2, 2, 2)))  
  
g <- df %>%  
  ggplot() +  
    geom_point(aes(x, y, shape = class, col = class))  
g
```



- An svm works by drawing a *hyperplane* between the points in one class and the other class. In three dimensions, a hyperplane is just a plane. In two dimensions, a hyperplane is a line. So here are three possible svm's for this data.

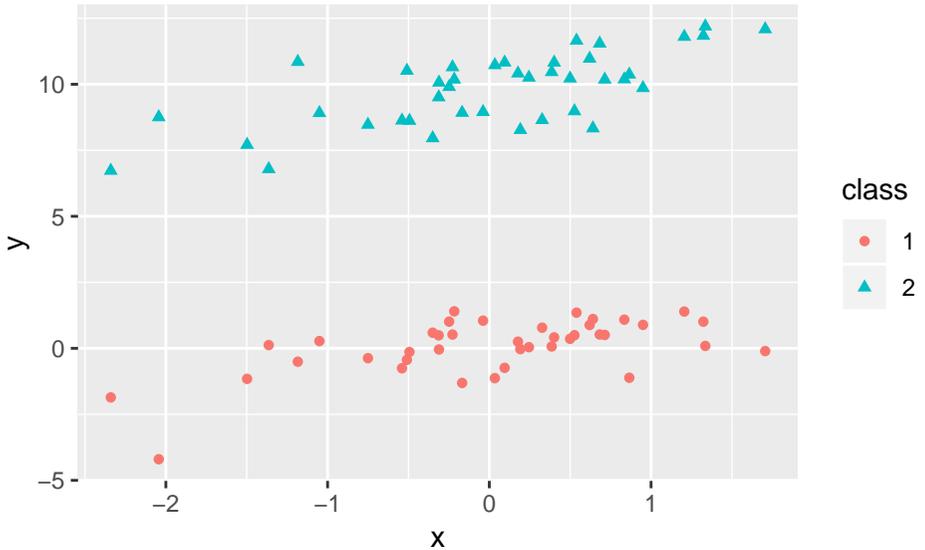
```
g +
  geom_abline(aes(intercept = 0.25, slope = 1),
             color = "red") +
  geom_abline(aes(intercept = 0.1, slope = 1),
             color = "green") +
  geom_abline(aes(intercept = 0.15, slope = 1.2),
             color = "blue")
```



- All three of those lines separate the two classes. Let's try it with some randomly generated data. Try

```
set.seed(10111)
n <- 40
r <- rnorm(n)
df <- tibble(
  x = c(r, r),
  y = c(r + rnorm(n), r + 10 + rnorm(40)),
  class = factor(c(rep("1", n), rep("2", n)))
)

ggplot(df) +
  geom_point(aes(x, y, shape = class, color = class))
```



- We will load in the `e1071` package which contains a function `svm`.

```
# install.packages("e1071")
library(e1071)
```

```
## Warning: package 'e1071' was built under R version 3.4.4
```

Now for the model.

```
mod_svm <- svm(class ~ x + y, data = df, kernel = "linear")
print(mod_svm)
```

```
##
## Call:
## svm(formula = class ~ x + y, data = df, kernel = "linear")
##
##
## Parameters:
##   SVM-Type:  C-classification
## SVM-Kernel:  linear
##       cost:  1
##       gamma: 0.5
##
## Number of Support Vectors:  4
```

- The predictions are probably very good. At this point we will bring in `modelr` to get our predictions

```
library(modelr)
```

```
## Warning: package 'modelr' was built under R version 3.4.
```

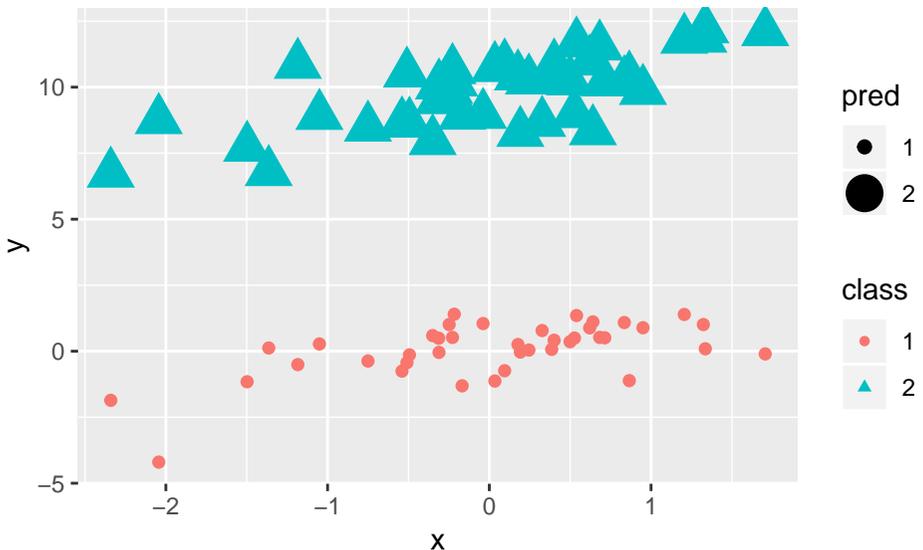
```
df %>%
```

```
  add_predictions(mod_svm) %>%
```

```
  ggplot() +
```

```
    geom_point(aes(x, y, shape = class, color = class, size = pred))
```

```
## Warning: Using size for a discrete variable is not advised
```



Questions

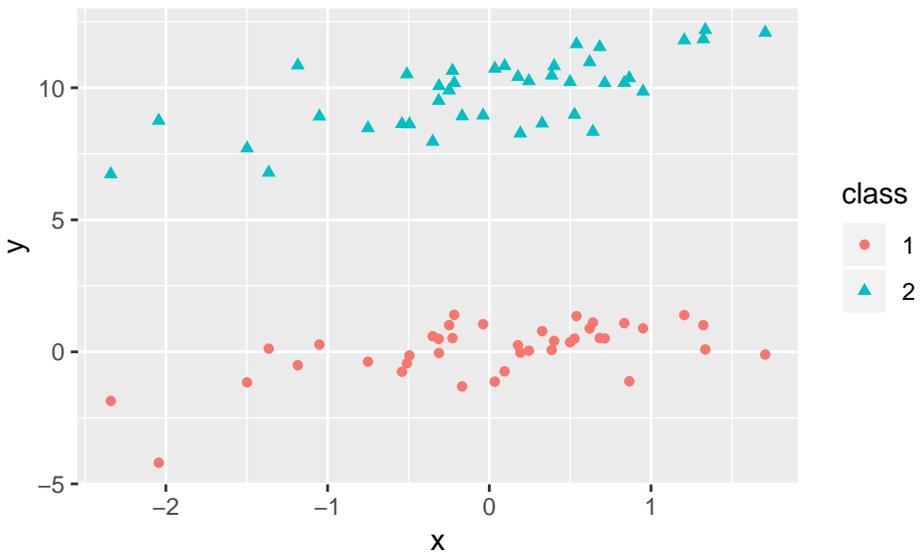
1. What is the accuracy rate of the predictions on the training data?
- We did so well on this data table because the points of the clusters were so far apart. What if it had been closer together?

```

set.seed(123456)
n <- 40
r <- rnorm(n)
df2 <- tibble(
  x = c(r, r),
  y = c(r + rnorm(n), r + 2 + 2 * rnorm(40)),
  class = factor(c(rep("1", n), rep("2", n)))
)

ggplot(df2) +
  geom_point(aes(x, y, shape = class, color = class))

```



- Note that *no* basic svm can perfectly classify this data. No line cleanly separates the triangles and the dots. We can try to get close, however.

```

mod_svm2 <- svm(class ~ x + y, data = df2, kernel = "linear")
print(mod_svm2)

```

```

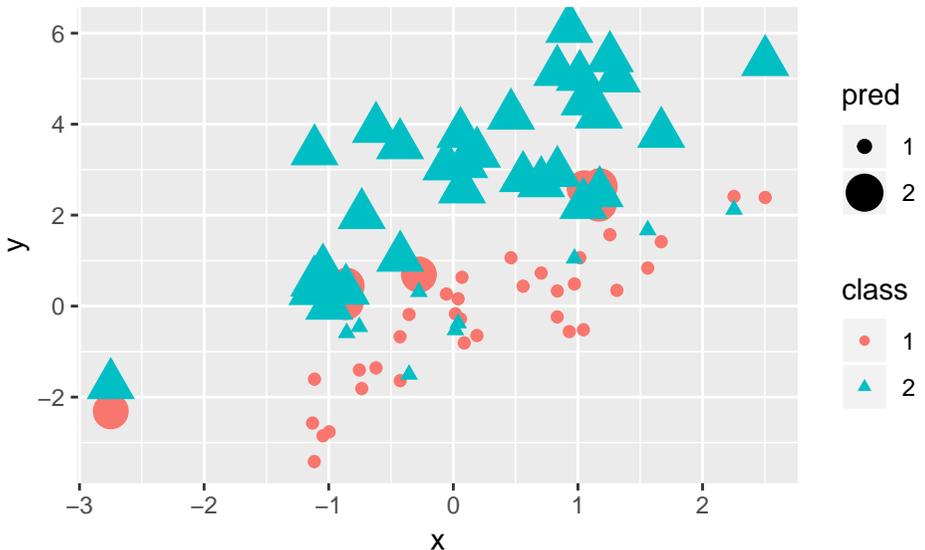
##
## Call:
## svm(formula = class ~ x + y, data = df2, kernel = "linear")
##
##
## Parameters:
##   SVM-Type:  C-classification

```

```
## SVM-Kernel: linear
## cost: 1
## gamma: 0.5
##
## Number of Support Vectors: 41
```

```
library(modelr)
df2 %>%
  add_predictions(mod_svm2) %>%
  ggplot() +
    geom_point(aes(x, y, shape = class,
                  color = class, size = pred))
```

```
## Warning: Using size for a discrete variable is not advised
```



- It did its best, but you see some small triangles and some large circles there. Each one of those is a failure.

Questions

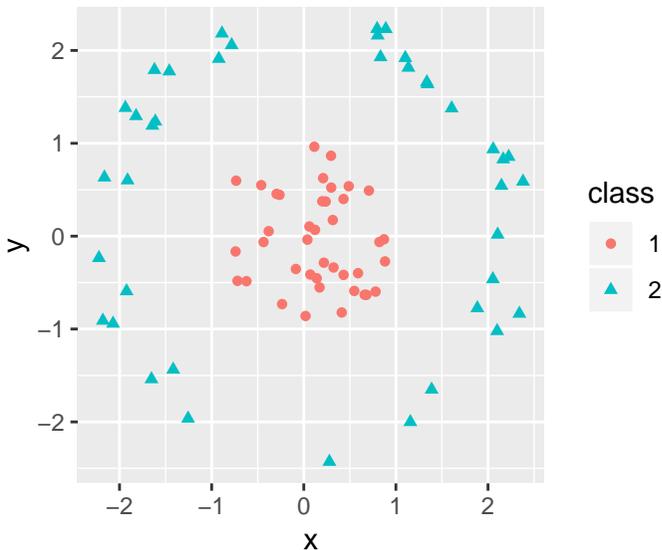
2. Find the percentage of predictions that were correct.

Features

- Of course, the situation could be even worse. Suppose your data looked like this:

```
# One cluster in center, one ring around it
set.seed(123456)
n <- 40
th1 <- 2 * pi * runif(n)
r1 <- sqrt(runif(n))
th2 <- 2 * pi * runif(n)
r2 <- 0.5 * runif(n) + 2
df3 <- tibble(
  x = c(r1 * cos(th1), r2 * cos(th2)),
  y = c(r1 * sin(th1), r2 * sin(th2)),
  class = factor(c(rep("1", n), rep("2", n)))
)

ggplot(df3) +
  geom_point(aes(x, y, shape = class, color = class)) +
  coord_fixed()
```



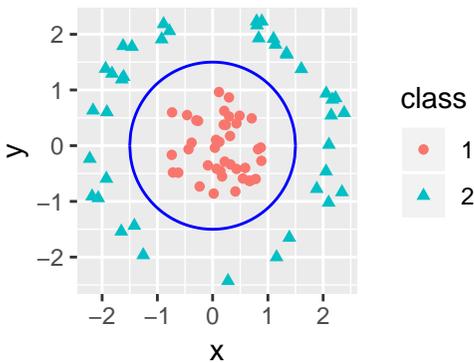
- It's very easy for the human eye to pick out the clusters here, but any straight line is doomed to either put too much together or too little. The solution is to create an artificial *feature*. A feature is an extra predictor that is a function of the other predictors. In this case, suppose we look at

$$z = \sqrt{x^2 + y^2}.$$

Then an equation like $z \geq 2$ (a hyperplane in the new space) looks like this on the plot:

```
x1 <- seq(-1, 1, by = 0.01)
circle <- tibble(
  x = x1,
  y = 1.5 * sqrt(1 - x1^2)
)

ggplot(df3) +
  geom_point(aes(x, y, shape = class, color = class)) +
  geom_line(data = circle, aes(1.5 * x, y),
            color = "blue") +
  geom_line(data = circle, aes(1.5 * x, -y),
            color = "blue") +
  coord_fixed()
```



- Let's add our phantom coordinate and run the svm again:

```
df3_z <- df3 %>%
  mutate(z = sqrt(x^2 + y^2))

mod_svm3 <- svm(class ~ x + y + z, data = df3_z,
                kernel = "linear")
print(mod_svm3)
```

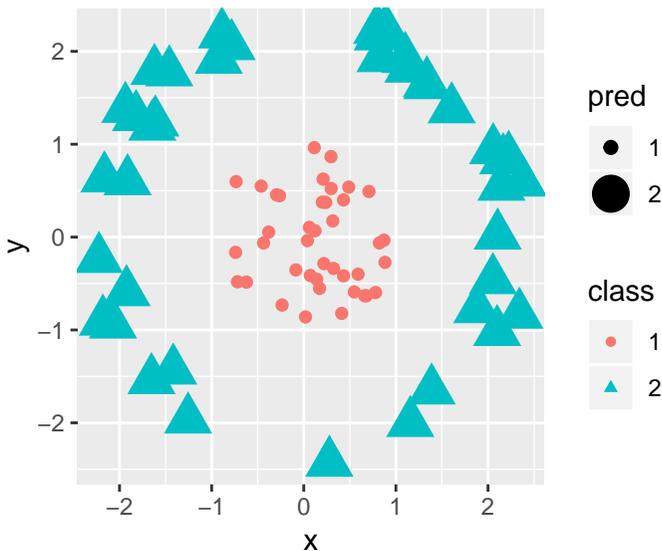
```
##
```

```
## Call:
## svm(formula = class ~ x + y + z, data = df3_z, kernel =
##
##
## Parameters:
##   SVM-Type:  C-classification
##   SVM-Kernel: linear
##     cost:  1
##   gamma:  0.3333333
##
## Number of Support Vectors:  4
```

Let's see how we did!

```
df3_z %>%
  add_predictions(mod_svm3) %>%
  ggplot() +
    geom_point(aes(x, y, shape = class, color = class,
                  size = pred)) +
    coord_fixed()
```

Warning: Using size for a discrete variable is not advised



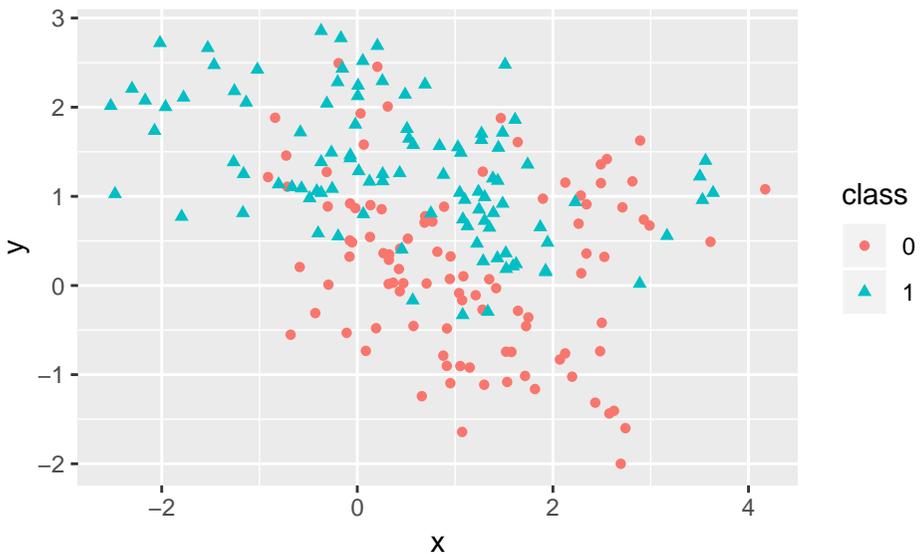
Once again, we get 100% accuracy.

Non-linear SVM

- We can try more general things than the feature addition that we did before. Consider a data set from *The Elements of Statistical Learning* by Hastie, Tibshirani, and Friedman. After downloading the file from the course website, place it in the working directory. It is an .rda file, so loads directly into a data frame in R.

```
load(file = "ESL.mixture.rda")
df4 <- tibble(
  x = ESL.mixture$x[, 1],
  y = ESL.mixture$x[, 2],
  class = factor(ESL.mixture$y)
)

df4 %>%
  ggplot() +
  geom_point(aes(x, y, color = class, shape = class))
```



- This is a bit like our second example, so we'll use a radial svm to model the points.

```
mod_svm4 <- svm(class ~ ., data = df4, scale = FALSE,
  kernel = "radial", cost = 5)
```

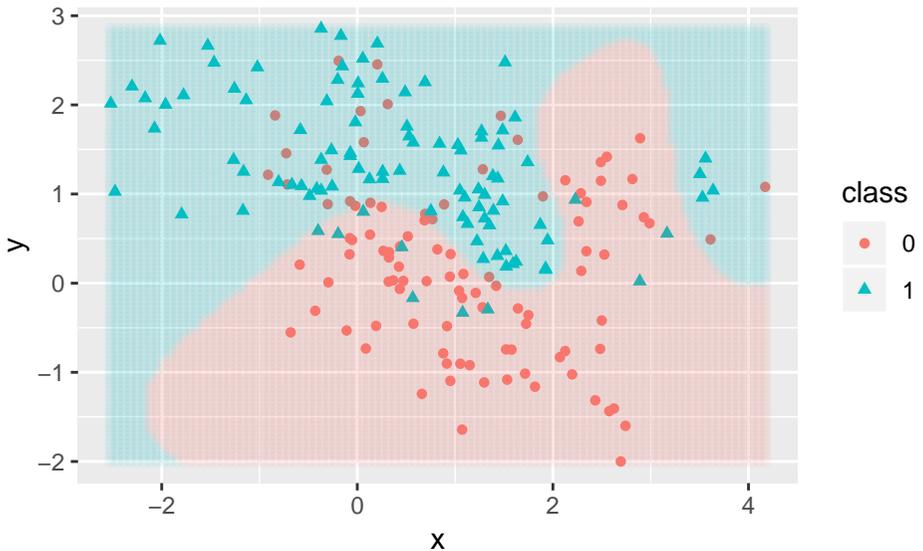
To see how the area of separation works, create a grid of points, and shade them according to the prediction

```

grid <- df4 %>%
  data_grid(x = seq_range(x, 100),
            y = seq_range(y, 100)) %>%
  add_predictions(mod_svm4)

ggplot() +
  geom_point(data = df4, aes(x, y, color = class,
                             shape = class)) +
  geom_point(data = grid, aes(x, y, color = pred),
            alpha = 0.05)

```



- The radial allows the region of separation to swerve.

Questions

3. The cost parameter is a penalty factor for getting points wrong. Change the cost to 1. Describe what happens to the red region.
4. What percentage of the points are correct with cost = 5? With cost = 1?

Bibliography

- [1] Hadley Wickham and Garrett Golemund. *R for Data Science*. O'Reilly, 2017. ([document](#))
- [2] G. N. Wilkinson and C. E. Rogers. Symbolic description of factorial models for analysis of variance. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 22(3):392–399, 1973. [25.2](#)

Index

- R, 4
- n -tuple, 107
- L^AT_EX, 10

- alternation, 136

- bit, 85
- byte, 85

- Cartesian product, 107
- comma separated file, 76
- complement, 124
- concatenation, 129

- data science, 2
- data table, 107
- decision tree, 257
- DFA, 139
- directed cycle, 138
- directed graph, 138

- element, 106
- error, 219
- escape character, 128

- facets, 20
- factor, 158
- feature, 262
- finite automaton, 132
- first-class function, 202
- fitting, 231

- glob, 156
- grammar of graphics, 16

- help in R, 20
- hexadecimal, 86
- higher-order function, 202

- immutable variables, 199
- install.packages, 16

- key, 109, 113
- Kleene star, 136

- least squares, 219
- length, 138
- level, 88
- levels, 158
- library, 16
- linear model, 259
- logical statement, 105
- logit function, 260

- machine learning, 255
- markup language, 9
- MySQL, 171

- NFA, 139, 140
- nondeterministic finite automata, 140

- observation, 107

- parameters, 259
- power set, 140
- prediction, 219
- pure functions, 198

- random forest, 258
- referential transparency, 199

regex, [134](#)
regexp, [134](#)
regular expressions, [134](#)
relation, [107](#)
relational database, [108](#)
residual, [219](#)
response, [219](#)

script, [7](#)
set, [106](#)
set difference, [124](#)
SQL, [171](#)
string, [127](#)
string concatenation, [129](#)
Structured Query Language, [171](#)
subset, [107](#)
supervised learning, [256](#)

testing set, [239](#)
tidy, [96](#)
training set, [239](#), [256](#)

unsupervised learning, [256](#)

wildcard for regular expressions, [135](#)