R Programming Workshop

Institute for Computational and Mathematical Engineering
Stanford University
Andreas Santucci
Summer 2021

Contents

1 Background Reading

1.1 Introduction

1.1.1 History

R is a powerful open-source tool for statistical computing. It's earliest origins date back to Bell Labs in the 70's via S (programming language). The motivation was for 'S' was to move away from calling Fortran routines and offer an interactive environment for statistical analysis. In the early 90's a "different implementation" of S by Ross I. and Robert G. started gaining traction.

1.1.2 Why R?

- R is excellent for *statistical analyses*.

  Methods are built in for (non)-linear regression modeling, time-series analysis, classification, clustering, and plotting. For these, we don't even need to import a package.

- R is *interactive*, and enables us to explore data and refine models iteratively.

  R is an *interpreted* language, i.e. we don't need to compile our programs into machine instructions before executing our code. We can simply enter code into the console and the result appears!

- R is *flexible*, allowing procedural, object-oriented, or even functional programming approaches.

  The practical implication is that we can run models and perform matrix arithmetic at a high level of abstraction, allowing us to think about the actual problem at hand.

- The active R development community makes the program *extensible*.

  Aside from the core R development team, we have Dirk Eddelbuettel on `Rcpp` and performance computing in R, Hadley Wickham who focuses on redesigning R code to be more user friendly via the `tidyverse`, and Matt Dowle leading the `data.table` development, just to name a few.

I have been greeted by many individuals who are surprised at what I have accomplished by programming in R. Unsurprisingly, those who are most aghast are typically lesser practiced in this language.

1.2 Setting Up R

We start by downloading and installing the latest version of R on our machine. With this, we can start programming in R, albeit not in a friendly environment.

1.2.1 R Studio

I recommend starting with a popular interface for programming in R, R Studio. Note that one must download both R *and* R-Studio (the latter doesn't include the former). The R-Studio interface consists of a *layout* of several windows:

- **Console:** This is where you type commands.
- **Editor:** This is where you write your code and save it to a text file.
- **Workspace/History:** What values exist in memory, and what's been executed historically.
- **Files/Plots/Packages/Help:** Allows graphical navigation of folders, display of visualizations, methods for installing and loading packages, and also a help utility.

1.2.2 Emacs Speaks Statistics

A more old-school, extensible approach relies on ESS. See documentation. Most notably, to start R we use `M-x R RET`. We can send a single line of code from a text file to our ESS process via `C-<RET>`, or alternatively we could use `C-c C-c` to run not an entire block or region of code and then step to the next.

1.2.3 10k Foot Overview

How can we load, visualize, and model data? Let's learn to work with the `dplyr` package. It takes a minute to install, but it's going to make our lives *much* simpler down the road when learning how to manipulate data in R. From within an R-console (or within R-Studio), run the following command:

```
# This first line of code is required only 1x when you
# need to install a package for the *first* time.
install.packages("dplyr")
# This next line of code is required *every* time you open
# up R and want to *use* the package.
require(dplyr)
```

When you've successfully run the commands above, you should see some output from the console showing that the package was loaded successfuly. If you get an error, please e-mail: `jgaeb@stanford.edu` describing the nature of the error message you ran into.

Now that you've loaded the `dplyr` package, loading data is as easy as π: try pasting the following into your console (you'll have to probably retype the `~` character that appears in the third line (linear model), since they tend not to paste over correctly from a `pdf` into a text editor); also, make sure that the line-breaks (or new-lines) get pasted / broken-apart correctly.

```
webSite <- 'https://web.stanford.edu/~hastie/ElemStatLearn/datasets/ozone.data'
# Load the data into memory, fit a linear model, summarize results.
read.csv(webSite, sep = '\t', header = TRUE) %>%
  lm(formula = ozone ~ radiation) %>%
  summary
```

The above created an output summary (with coefficients and p-values!) describing a *linear-model* fit when we regressed the variable `ozone` as a function of `radiation`. The `%<%` operator takes the *output* from the *previous* command and *pipes* it into the *input* of the *subsequent* command. It's pretty impressive that R can pull data in from the web, fit a model, and print meaningful results in such a syntactically concise way.

1.3 Help and Documentation

R has fantastic built-in documentation. Simply typing `help(<function_name>)` will pull up a help-page for a particular function, e.g. `help(read.csv)`. A prefix short hand for `help()` is the `?` operator, e.g. try inputting `?read.csv` into console.

1.3.1 Course Outline

We're going to summarize the extensive notes captured in Hadley Wickham's Intro to R and walk through the parts that feel most essential to getting started! In general, we're going to start by learning how to plot data, then we will learn how to transform our data, and finally we'll *combine* these two techniques to answer meaningful questions about our data.

2 Data Visualization

Let's cover some basic data visualization, using `ggplot2`. We'll skip over the base plotting utils in this section, although they are pretty good. We'll work with the `mpg` dataset, included in the `ggplot2` package.

```
install.packages("ggplot2")
```

Again, the above command only needs to be run 1x per machine you plan to use the package on. However, each time you load a new R-session and plan to use the package, you must call

```
require(ggplot2)
```

With this package, we get access to the `mpg` dataset, which we'll be working with in this section.

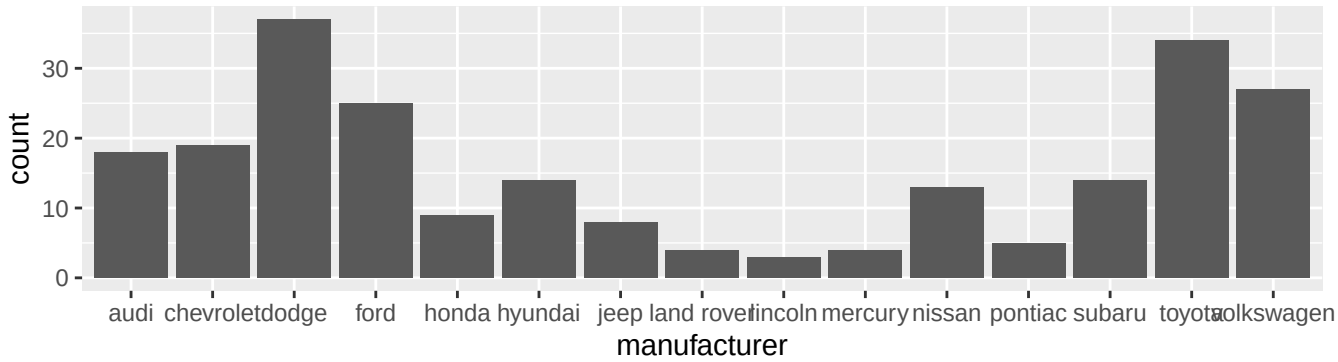
```
head(mpg)
```

```
## # A tibble: 6 x 11
##   manufacturer model displ  year   cyl trans      drv     cty   hwy fl    class
##   <chr>        <chr> <dbl> <int> <int> <chr>      <chr> <int> <int> <chr> <chr>
## 1 audi         a4      1.8  1999     4 auto(l5)   f        18    29 p     compa~
## 2 audi         a4      1.8  1999     4 manual(m5) f        21    29 p     compa~
## 3 audi         a4      2    2008     4 manual(m6) f        20    31 p     compa~
## 4 audi         a4      2    2008     4 auto(av)   f        21    30 p     compa~
## 5 audi         a4      2.8  1999     6 auto(l5)   f        16    26 p     compa~
## 6 audi         a4      2.8  1999     6 manual(m5) f        18    26 p     compa~
```

2.1 Univariate distributions

What if we want to visualize the distribution of car manufacturers in our dataset? We can use the `table()` command to get a numeric summary (try this in the console!), but how about a visual?

```
ggplot(data = mpg, aes(x = manufacturer)) +
    geom_bar()
```



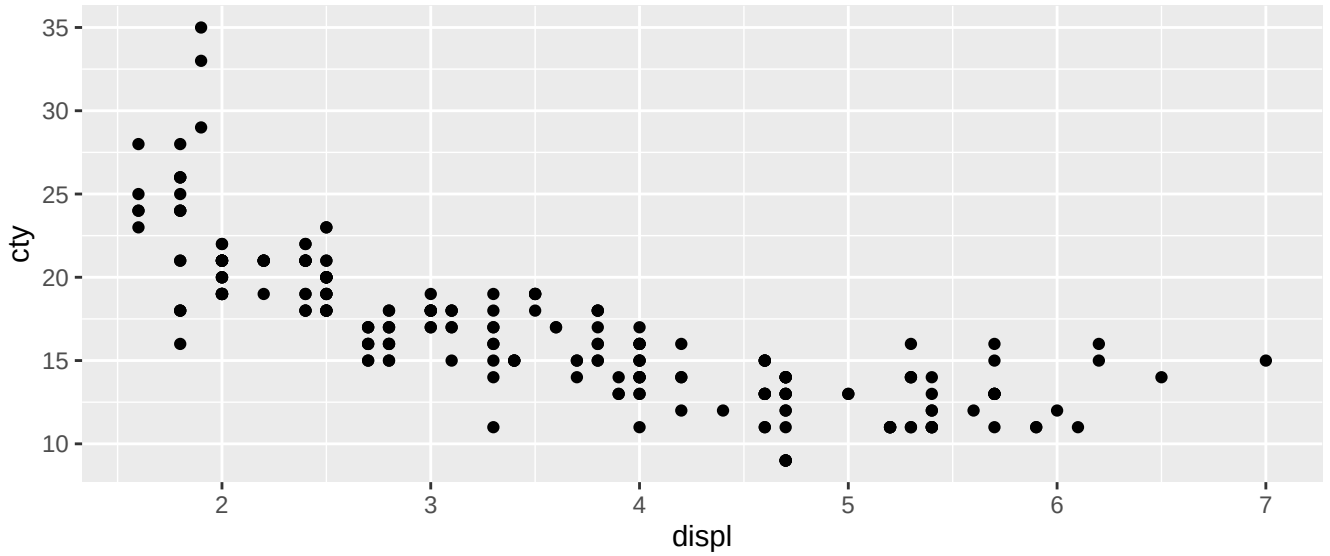
The x-axis describes the manufacturer, and the `geom_bar()` plotting method defaults to plotting raw counts, i.e. the number of of observations for each manufacturer in our dataset.

Exercise We can also visualize histograms quite easily, simply replacing the geometry used to plot. I.e. we can replace `geom_bar()` with a different geometry, e.g. `geom_histogram()` or `geom_density()`. Try plotting both a histogram and a density plot for the city miles-per-gallon, described by variable `cty`.

2.2 Bivariate distributions

Let's start with a simple visualization of engine size against fuel efficiency in the city.

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

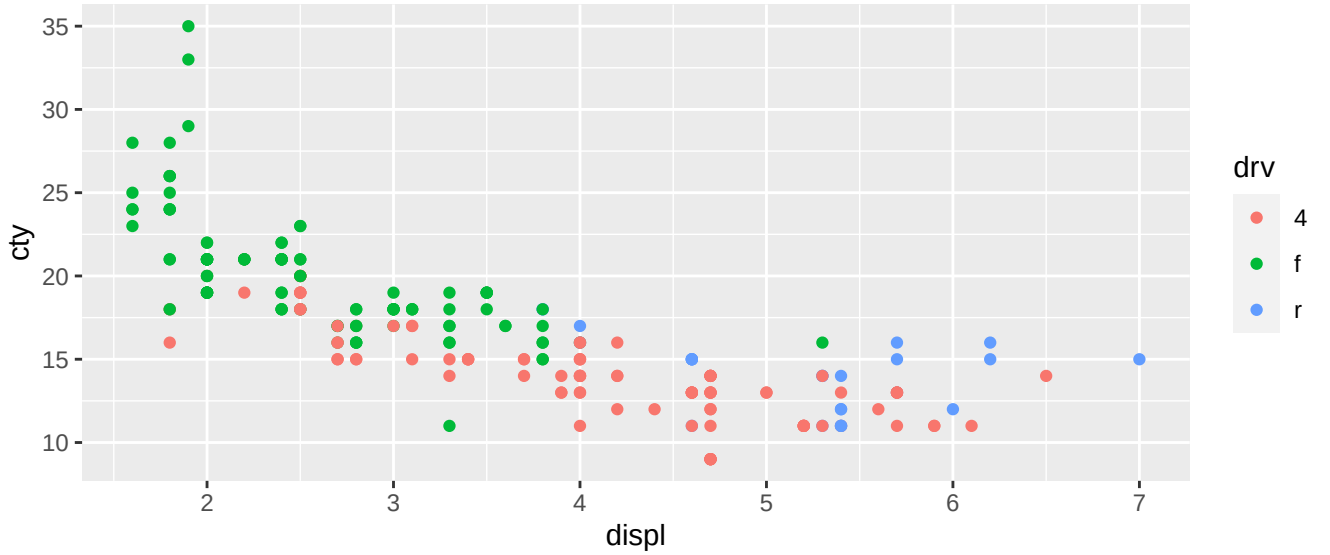


There is a strong negative relationship between displacement and fuel efficiency, which we would expect.

2.2.1 Adding Additional Dimensions to your Plots

What about whether the car is front, rear, or all-wheel drive. Would that affect mileage? We can add a *third* dimension to our plot by colour each data point according to whether the cars drivetrain type.

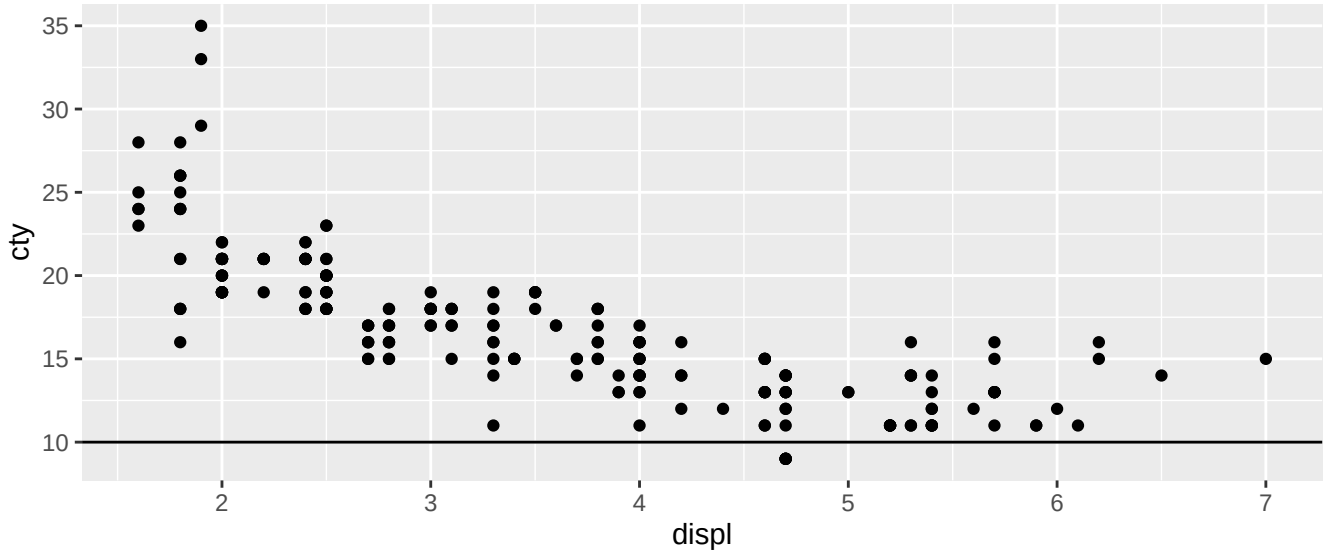
```
ggplot(data = mpg, mapping = aes(x = displ, y = cty, colour = drv)) +
    geom_point()
```



It becomes clear from this plot that the interaction of `displacement` and `city-mileage` is predictive of the drivetrain type. I.e. cars with really low displacement and high MPG are always front-wheel drive, whereas cars with lower fuel-economy can be partitioned into two groups in part based on displacement: cars with lower fuel-economy that are *also* high displacement tend to be rear-wheel drive, for example.

Preliminary Exercise Let's walk through a simple exercise. Suppose we wanted to use a graphical summary to learn approximately how many cars in our dataset have less than 10 miles-per-gallon in the city. One way we could answer this question is with a *univariate* distribution (i.e. a histogram); we saw how to do this above in the exercise in the plotting univariate distributions section. How does this result conflict with what we see if we count the number of points falling below the y-intercept corresponding to 10 MPG in the plot below?

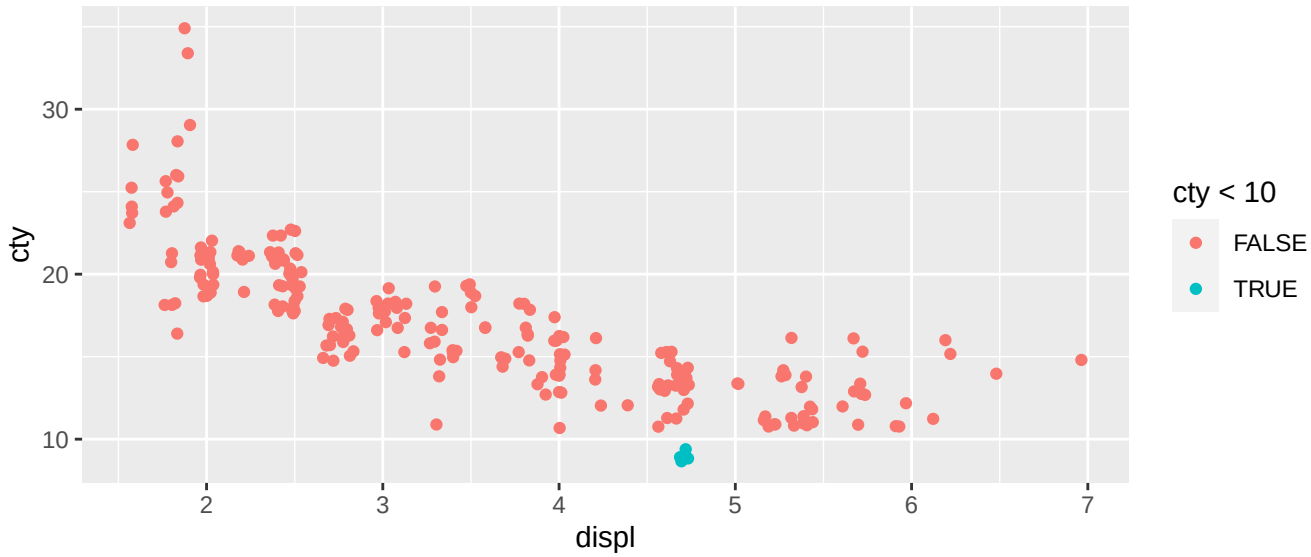
```
ggplot(data = mpg, mapping = aes(x = displ, y = cty)) +
    geom_point() +
    geom_hline(yintercept = 10)
```



Doesn't it look like there's only 1 data-point where the car has less than 10 miles-per-gallon in the city?

Adding Noise to Make the Picture More Clear It turns out that there's actually a lot of overlapping datapoints in our scatterplot above. We can apply some random noise to make the plot a bit easier to read by by using a `geom_jitter` geometry.

```
ggplot(data = mpg, mapping = aes(x = displ, y = cty, colour = cty < 10)) +
    geom_jitter()
```

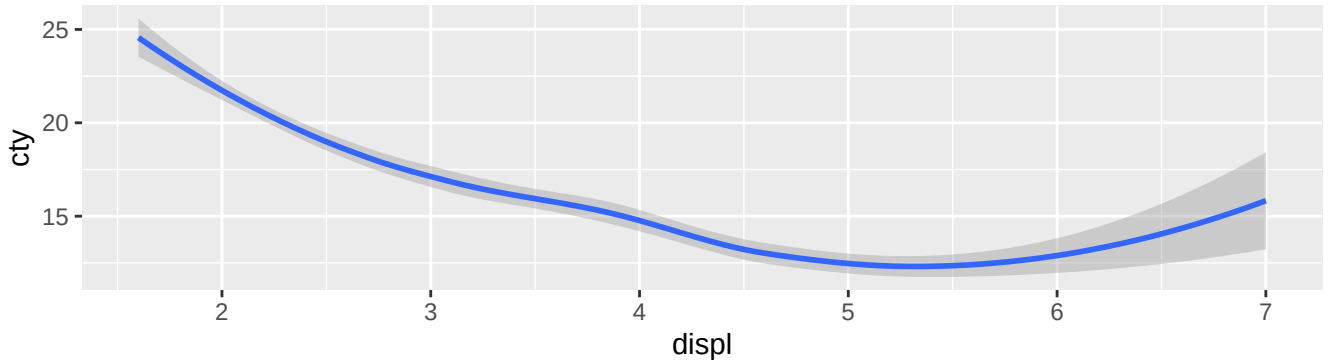


Here it is much easier to see that, for example, there are several cars with fewer than 10 city miles-per-gallon, all having just under 5 liters displacement. Notice that we created an expression "on-the-fly" and used it to colour out points... cool!

2.3 Visualizing Smoothed Relationships with Modeling Techniques

With `ggplot2`, it's actually really easy to visualize smoothed relationships derived from fitted models. For example, consider the following plot:

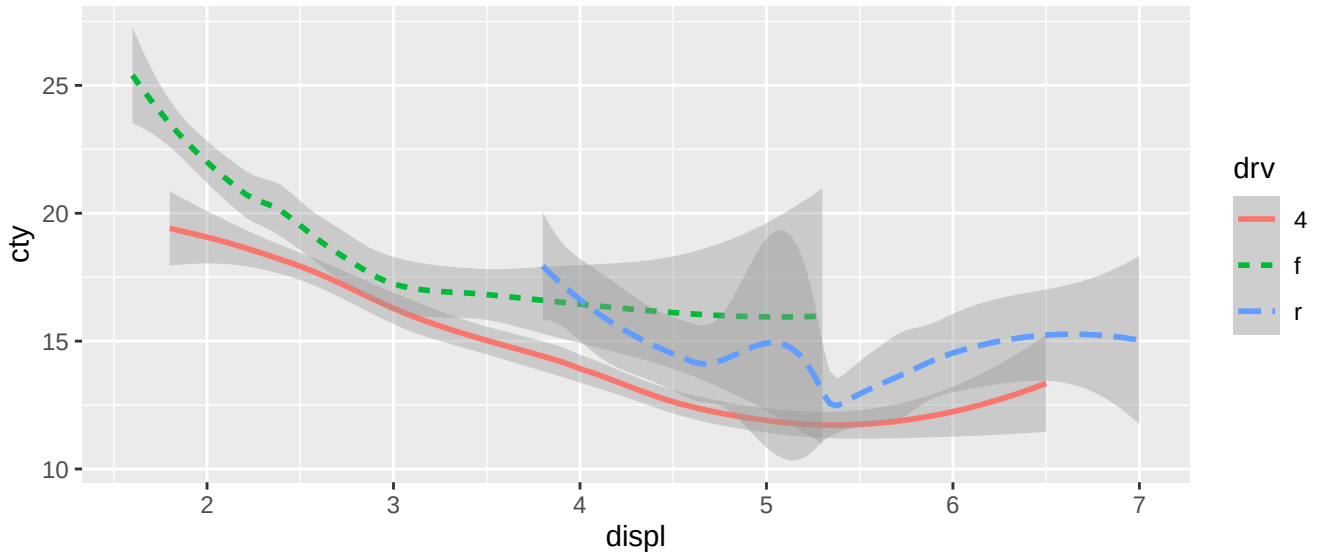
```
ggplot(mpg, aes(x = displ, y = cty)) +
    geom_smooth()
```



We've basically plotted a smoothed average alongside confidence intervals. Notice a couple things: (i) there is a (perhaps surprising) quadratic relationship between the two variables, and (ii) the variance of our estimates increass as we increase displacement (since there are fewer observations to back each estimate).

Fitting separate models based on another variable It's pretty amazing really, that we can in fact fit a separate relationship between two variables according to a third.

```
ggplot(mpg, aes(x = displ, y = cty, colour = drv, linetype = drv)) +
    geom_smooth()
```



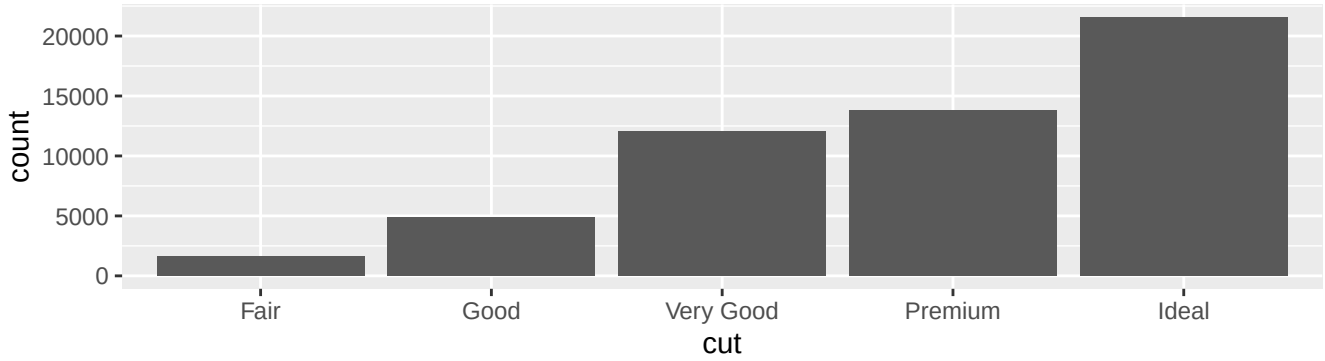
2.4 Statistical Transformations when Plotting

Some `geom_*`'s compute variables (such as counts and proportions) automatically when we call them, and we can specify which variable to plot in the aesthetic. I.e. it's as though we have access to the computed variable even though it doesn't appear in our raw data! For example, there's a dataset called `diamonds` that describes various attributes of gems. For reference, this is what our data looks like:

```
## # A tibble: 3 x 10
##   carat cut     color clarity depth table price     x     y     z
##   <dbl> <ord>   <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  0.23 Ideal   E     SI2      61.5    55   326  3.95  3.98  2.43
## 2  0.21 Premium E     SI1      59.8    61   326  3.89  3.84  2.31
## 3  0.23 Good    E     VS1      56.9    65   327  4.05  4.07  2.31
```

We could plot out a simple bar-plot: how many *times* each cut of diamond appears in our data?

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



This code is actually equivalent to the following

```
ggplot(diamonds, aes(x = cut)) +
    geom_bar(stat = "count")
```

Realize that we never computed a statistic called `count`, but that it was calculated for us automatically and we were able to use it in our plot.

Plotting proportions in bar charts But maybe the actual frequency is less important (e.g. suppose we're working with down-sampled data in order to fit it in memory), and all we care about are *proportions* instead. Well, if one were to look for help on the `geom_bar()` command by typing `?geom_bar()` into their R console, and then searching for "computed variables", we would learn that `proportion` is also a computed statistic. I.e. there is some advanced syntax (we won't cover in the workshop, but mentioned here, for example) that lets us use these computed proportions. Perhaps a more interpretable tool to use is the `after_stat` utility:

```
ggplot(diamonds, aes(x = cut, y = after_stat(count / sum(count)))) + geom_bar()
```

Here, we've basically supplied the statistical transformation manually ourselves (i.e. take the `count` of how often something appears in our data, and divide it by the sum of `counts` to get a proportion).

3 Revisiting a Couple of Basics in R

3.1 As (an extensible) Calculator

R replaces our graphing calculators. Simply write an expression into the console and hit `<return>`. E.g.

```
10^2 + 36

## [1] 136
```

Exercise Compute the difference between 2021 and the year you graduated from your most recent educational program. Divide this by the difference between 2021 and the year you were born. Multiply the resulting value by 100 to determine the percentage of your life that you have been enjoying the fruits of your labor. You may find the use of parentheses helpful to disambiguate order of operations.

I was born in 1989 and graduated in 2017, and so the expression for me looks as follows:

```
(2021 - 2017) / (2021 - 1989) * 100
```

3.1.1 Variables

We can also give values a name. When we attach an identifier to a value, we realize a *variable*. E.g.

```
a <- 4
```

If you're using R-Studio, you'll now notice that `a` appears in the workspace window. We can also ask `R` what value `a` takes on, simply by typing `a` followed by `<return>` in the command window.

```
a * 5  # We can perform calculations on a variable, e.g.

## [1] 20
```

If we specify a new value for `a`, we lose the old value.

```
a <- a + 10
a

## [1] 14
```

We can view all the objects in our workspace by typing `ls()`.

Exercise Repeat the previous exercise, calculating the percentage of your life since graduating, this time taking several steps to arrive at the final result. Assign each intermediate result to a variable, whose name can be one of your choosing (but is required to begin with a letter). I recommend using the following variable names: `nYearsSinceGraduation`, `Age`. The resulting expression then becomes

```
nYearsSinceGraduation / age * 100
```

3.1.2 Scalars and Vectors

Numerical values are represented via scalars (single numbers, zero dimensional), vectors (a column of numbers, one dimensional), and matrices (a table of data, two dimensional). In our previous example, the `a` we defined was a *scalar*. To define a *vector* of values, we use the `c()` command which stands for *concatenate*. E.g.

```
vals <- c(4, 7, 10)
```

We can also create a sequence of consecutive integers using the `:` operator.[1] E.g. inputting `0:9` into your console will return a length 10 vector of ordered digits.

3.1.3 (Using) Functions

What if we want to compute the average of the elements in `vals` above? We could manually type

```
(4 + 7 + 10) / 3
```

but this would be error prone and impracticable for more interesting calculations. Common tasks are *automated* into *functions*: given input(s), they have a sequence of computations which yield an output. Some functions are available in base R, some are available in packages on CRAN, and if all else fails we can always write our own. To compute an arithmetic average, we simply type

```
mean(vals)  # Outputs the scalar value 7, of course!
```

Within the parentheses, we've specified the *function arguments* (or parameters). For the `mean` function, it requries an `x` vector for which we will compute an arithmetic average.

Exercise Compute the sum of `4, 5, 8, 11` by first combining the elements into a single vector and then using the function `sum`.

```
myVector <- c(4, 5, 8, 11)
sum(myVector)
```

Exercise What's the sum of consecutive integers between 1 and 100 inclusive? Hint: `help(":")`.[2]

3.2 Built-in Statistical Functions

What's so great about R is that it has statistical functions built-in and easily accessible. E.g. if we type

```
rnorm(10)
```

we create a vector of ten draws from a standard normal distribution. If we run the same command again, we'll see ten *different* random numbers. To draw from a non-standard normal with mean $\mu = 10$ and standard deviation $\sigma = 3$, we may execute `rnorm(n = 4, mean = 10, sd = 3)`, where here we've specified our arguments by name to disambiguate our intent (recommended). We remark that in R-Studio, if we simply type `rnorm(` within the command window, followed by pressing `TAB`, we will be prompted with the possible arguments to the function.[3]

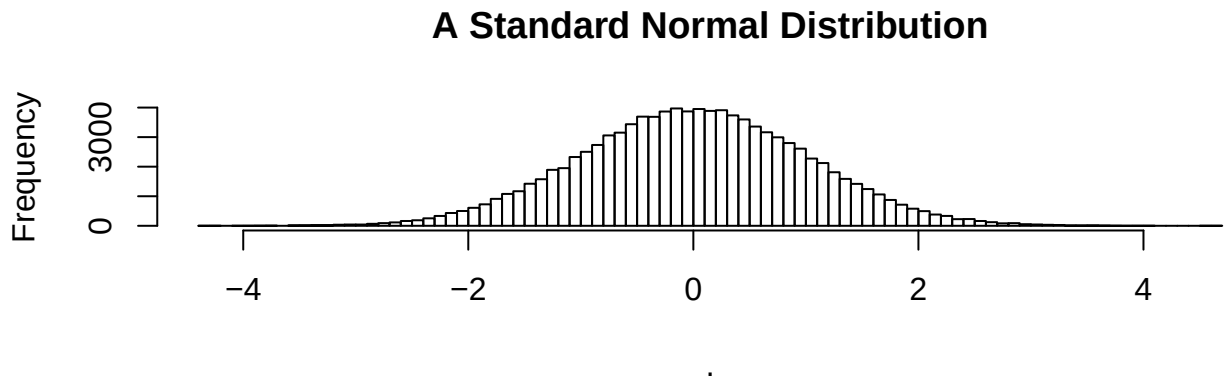
[1] If we want non-consecutive but equi-spaced values, we can use `seq`, e.g. `seq(0, pi, length.out = 5)` generates five evenly spaced values between zero and π.

[2] Do you remember the formula for the sum of consecutive integers between 1 and N as a function of N? Hint: for any $n \in \mathbb{N}$, what is the result of the computation `1:n + n:1`? What does each value take on, and how many values are there?

[3] Emacs and ESS have similar functionality built-in.

Exercise Draw 100 random numbers from a normal distribution, assign the result to a variable `x`. Then, plot the result using `plot(x)`. Note: we could use `ggplot2` to do this, but we'd have to cover how to construct a `data.frame` first and that's left for the next section!

```
rnorm(1e5) %>%
    hist(breaks = 100, main = "A Standard Normal Distribution")
```

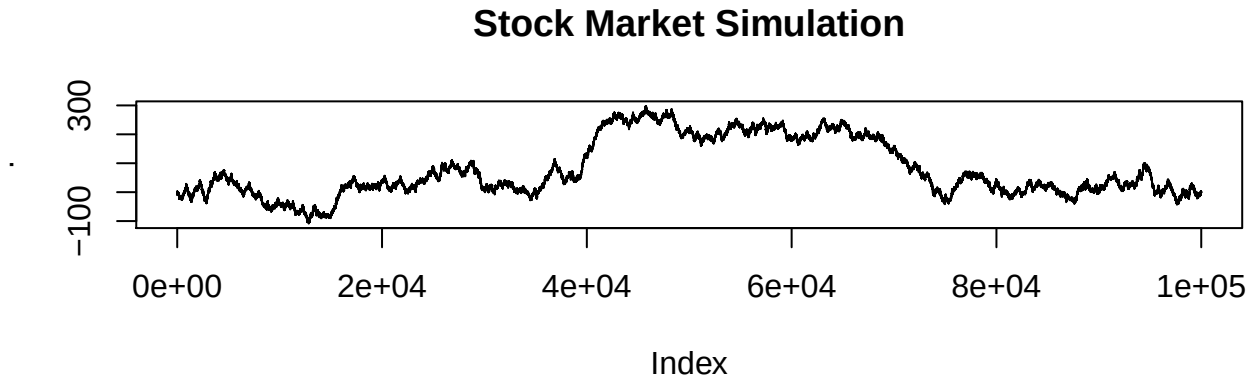


.

```
## # In ggplot2 (and using data.frame's):
## df <- data.frame(x = rnorm(1e5))
## ggplot(df, aes(x)) + geom_histogram(bins = 100)
```

Exercise Apply the `cumsum` function to generate an auto-correlated time series.

```
rnorm(1e5) %>% cumsum %>% plot(type = "l", main = "Stock Market Simulation")
```




```
## # Again, if we wanted to use ggplot2 we'd have to use data.frame's. Not disadvantageous.
## df <- data.frame(x_var = 1:1e5, y_var = cumsum(rnorm(1e5)))
## ggplot(df, aes(x = x_var, y = y_var)) + geom_line()
```

3.3 Assignment Operators

In R, there are two ways you can assign a variable, either using `<-` or `=`. You may have noticed in one of the above function calls to `rnorm()` or `plot()`, that we specified our arguments using the `=` operator.

```
# But it's true that in R, to assign a variable,  you can write either write:
variable <- rnorm(10)
variable = rnorm(10)  # <-- But! Writing this is also valid!
```

We prefer to use `<-` when assigning variables. Now, there's a difference when it comes to initializing function *arguments*: here, you should *always* use `=`. I.e.

```
sample(x = 100)  # Randomly permutes the first 100 positive integers
```

If we had used `<-` instead, this would have actually *created* a variable called `x` and then passed this variable into the function, with the side-effect of the variable `x` persisting after the function call. You can see this if you use `ls()`, which lists out objects in the (global) namespace.

```
rm(list=ls())  # Clear workspace (don't save this in your scripts).
result <- sample(x <- 10)
ls()

## [1] "result" "x"
```

We now have a variable that exists in our global workspace whose name is `x` and value is 10. We probably didn't want that: to create a separate variable. We most likely just wanted to pass in an integer literal 10 to the function as an argument. This is admittedly a quirk with R; to understand it:

- When you call a function, the *arguments* are evaluated first before the function procedure is executed.
- When you write `x <- 10`, you are passing an *expression* as argument, in particular the expression is an assignment operation `x <- 10`, so this gets evaluated first in global scope.

Takeaway: use `<-` only when assigning variables in global scope, not initializing function arguments.

3.4 Scripts

One advantage of R is in *reproducibility*. I.e. instead of using point-and-click GUI's to navigate spreadsheet calculations, for example, we can embed these instructions into code which is reusable. We store a sequence of R expressions in `.R` files which we call *scripts*. It's possible to run the entirety of another script from within R by using the `source()` function.

Exercise Make a file titled `firstRScript.R` in your *current* working directory. Request that R generate a hundred random numbers uniformly at random from an interval $[a, b]$ for any $a, b \in \mathbb{R}$ of your choosing. *Hint:* see `?runif` for help on how to draw values uniformly at random from an interval. Assign the resulting draws to a variable, and plot the result using `plot()`. Then, run this script a couple times, using `source()` from within an R console.

```
plot(runif(100, -1e3, 1e3))
# df <- data.frame(x = 1:100, y = runif(100, -1e3, 1e3))
# ggplot(df, aes(x, y)) + geom_point()
```

4 Data Frames

The canonical data structure for statistical data analysis is a `data.frame`. Unlike Python, `data.frames` are built-in to R, and unlike Matlab, columns of a `data.frame` may be referenced by name so that we don't need to remember their position. A `data.frame` is simply a (named) *list* of vectors, each the same length. Usually, we obtain a `data.frame` by loading data into R.

4.1 Using `read.csv` to read in spreadsheet data

Let's go to `data.fivethirtyeight.com` and scroll down or search for a data-set called Foul Balls; download the file and unzip `foul_balls.zip`, and then extract the `.csv` file. You can do this all manually or you can do it programmatically, in fact. If we navigate through the web UI, we see that there is a copy of the data stored at `https://raw.githubusercontent.com/fivethirtyeight/data/master/foul-balls/foul-balls.csv`.

We can simply do:

```
download.file(
  url = file.path(
    "https://raw.githubusercontent.com",
    "fivethirtyeight/data/master/foul-balls",
    "foul-balls.csv"
  ),
  destfile = "~/Downloads/foul-balls.csv"
)
```

```
data <- read.csv("~/Downloads/foul-balls.csv")
```

Exercise There's another function called `read.table()`, which also allows you to read in tabular data (of which comma separated value files happen to be a special type). Can you find a way to read in the data using `read.table()`? See `?read.table`, and in particular think about how you might choose the `sep` argument to be different from the default of what `read.table()` provides.

4.2 Indexing into Data.Frames

We can easily index into rows and columns just like we index into matrices:

```
# Request the first 3 rows using *slicing*.
data %>% slice(1:3)
```

```
##             matchup  game_date type_of_hit exit_velocity predicted_zone
## 1 Mariners vs Twins 2019-05-18      Ground            NA              1
## 2 Mariners vs Twins 2019-05-18         Fly            NA              4
## 3 Mariners vs Twins 2019-05-18         Fly          56.9              4
##   camera_zone used_zone
## 1           1         1
## 2          NA         4
## 3          NA         4
```

```
# Request the first 3 rows and the first 3 columns.
data %>% slice(1:3) %>% select(1:3)

##             matchup  game_date type_of_hit
## 1 Mariners vs Twins 2019-05-18      Ground
## 2 Mariners vs Twins 2019-05-18         Fly
## 3 Mariners vs Twins 2019-05-18         Fly

# Request rows 2 and 5 as well as two columns by name.
data %>% slice(c(2,5)) %>% select(game_date, type_of_hit)

##    game_date type_of_hit
## 1 2019-05-18         Fly
## 2 2019-05-18         Fly
```

Using `operator$` to Extract Columns Since a `data.frame` is a `list` of columns (each of the same length), we can also use `operator$` to extract columns (i.e. elements) from our `data.frame` (which is a list). E.g. let's *count* (using `table()` the number of entries of each type in the `matchup` column.

```
table(data$matchup)

##
##            A's vs Astros           Braves vs Mets          Brewers vs Mets
##                      109                       73                       85
## Dodgers vs Diamondsbacks        Mariners vs Twins         Orioles vs Twins
##                       86                      100                      113
##      Phillies vs Marlins       Pirates vs Brewers          Rangers vs Jays
##                       75                      111                       87
##       Yankees vs Orioles
##                       67
```

Here, we've indexed into our data and in particular we've extracted a *column* using `operator$`. In `dplyr`, this would look like the following expression, where we remark that a `data.frame` is returned (not a named vector).

```
data %>% group_by(matchup) %>% count()
```

4.3 Techniques for Data Inspection

One of the first things I like to do is simply inspect our data. We can look at the first few rows by using the `head()` command:

```
head(data, n = 4)

##             matchup  game_date type_of_hit exit_velocity predicted_zone
## 1 Mariners vs Twins 2019-05-18      Ground            NA              1
## 2 Mariners vs Twins 2019-05-18         Fly            NA              4
## 3 Mariners vs Twins 2019-05-18         Fly          56.9              4
```

```
## 4 Mariners vs Twins 2019-05-18         Fly          78.8              1
##   camera_zone used_zone
## 1           1         1
## 2          NA         4
## 3          NA         4
## 4           1         1
```

We immediately notice, for example, that we have some missing values in the fields `exit_velocity` and `camera_zone`. We can use `dim()` to request the *dimensions* of our data. The `summary()` command is also useful to tabulate statistics on each column.

```
dim(data)
str(data)
summary(data)
```

Exercise: How many *entries* (or cells, speaking in spreadsheet terms) are in our `data.frame`? Hint: use the functions `nrow()` and `ncol()`.

Visualizing an entire `data.frame` We can even plot out all of our data at once, making a pairwise scatter plot relation.

```
plot(data)
```

4.4 Re-order rows of a `data.frame` via `dplyr::arrange()`

Unlike *sort*ing a vector, `data.frame`s typically have row-re*order*ing operations performed on them. Accordingly, instead of `sort()` we use `order()` (in Base R); in the `tidyverse` or with `dplyr` we can simply use `arrange`.

```
data %>% arrange(matchup)
```

We can order in descending order by using an argument!

```
data %>% arrange(matchup, decreasing = TRUE) %>% {rbind(head(., n = 2), tail(., n = 2))}
```

```
##                matchup  game_date type_of_hit exit_velocity predicted_zone
## 1        A's vs Astros 2019-06-02         Fly          82.6              5
## 2        A's vs Astros 2019-06-02         Fly            NA              1
## 905 Yankees vs Orioles 2019-03-31         Fly          71.4              5
## 906 Yankees vs Orioles 2019-03-31         Fly          86.5              4
##     camera_zone used_zone
## 1            NA         5
## 2             1         1
## 905          NA         5
## 906          NA         4
```

4.5 Data Transformations

There are some common data transformations that you'll find yourself wanting to perform when carrying out statistical analyses. Note that we'll cover some more advanced operations like reshaping and aggregating data in a later section; see .

4.5.1 Filtering rows using `filter()`

It's really simple to filter out rows by using predicate conditions. E.g.

```
filter(data, type_of_hit == "Ground") %>% head(n = 5)

##             matchup  game_date type_of_hit exit_velocity predicted_zone
## 1 Mariners vs Twins 2019-05-18      Ground            NA              1
## 2 Mariners vs Twins 2019-05-18      Ground            NA              1
## 3 Mariners vs Twins 2019-05-18      Ground            NA              1
## 4 Mariners vs Twins 2019-05-18      Ground          76.2              2
## 5 Mariners vs Twins 2019-05-18      Ground          96.2              4
##   camera_zone used_zone
## 1           1         1
## 2           1         1
## 3           1         1
## 4          NA         2
## 5          NA         4
```

Notice that it's simple enough to add in multiple predicate conditions, simply spearate them with a comma!

```
filter(data, type_of_hit == "Ground", exit_velocity > 90) %>% head(n = 5)

##             matchup  game_date type_of_hit exit_velocity predicted_zone
## 1 Mariners vs Twins 2019-05-18      Ground          96.2              4
## 2 Mariners vs Twins 2019-05-18      Ground         100.8              5
## 3 Mariners vs Twins 2019-05-18      Ground          92.1              5
## 4 Mariners vs Twins 2019-05-18      Ground          96.8              5
## 5 Mariners vs Twins 2019-05-18      Ground         100.7              4
##   camera_zone used_zone
## 1          NA         4
## 2           5         5
## 3          NA         5
## 4           5         5
## 5           4         4
```

Of course, you can also use an "or" condition via the logical-or operator... something of the flavor

```
filter(df, predicate_1 | predicate_2)
```

Once you realize this works, of course you can always rewrite our logical `and` statements explicitly in the penultimate example:

```
filter(data, type_of_hit == "Ground" & exit_velocity > 90) %>% head(n = 5)
```

```
##             matchup  game_date type_of_hit exit_velocity predicted_zone
## 1 Mariners vs Twins 2019-05-18      Ground          96.2              4
## 2 Mariners vs Twins 2019-05-18      Ground         100.8              5
## 3 Mariners vs Twins 2019-05-18      Ground          92.1              5
## 4 Mariners vs Twins 2019-05-18      Ground          96.8              5
## 5 Mariners vs Twins 2019-05-18      Ground         100.7              4
##   camera_zone used_zone
## 1          NA         4
## 2           5         5
## 3          NA         5
## 4           5         5
## 5           4         4
```

4.6 Adding New Variables to a `data.frame`

Adding a new variable to a `data.frame` can be done in a variety of different and equivalent ways.

```
# Add a column by using integer position.
# Note that if we do this, the new variable name
# is of the vorm Vk, where k is an integer describing
# the column number.
data[, ncol(data) + 1] <- data$game_date

head(data[, c("game_date", "V8")], n = 3)
```

```
##    game_date         V8
## 1 2019-05-18 2019-05-18
## 2 2019-05-18 2019-05-18
## 3 2019-05-18 2019-05-18
```

```
# We can also (more readably) assign to a new column
# and give it an interpretable name at the same time.
data[, "zone_gt_2"] <- data[, "used_zone"] > 2
data[1:5, "zone_gt_2"]
```

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

```
# We can also use our technique of using operator\£ to
# grab columns.
data$logged_exit_velocity <- log(data$exit_velocity, base = 2)
data %>%
  filter(!is.na(exit_velocity)) %>%
  select(ends_with("exit_velocity")) %>%
  head(n = 3)
```

```
##   exit_velocity logged_exit_velocity
## 1          56.9             5.830357
## 2          78.8             6.300124
## 3          74.8             6.224966
```

4.6.1 `dplyr::mutate()`

There's also the `dplyr` way, which is to use `mutate()`.

```
data %>%
    mutate(transform_of_zones = predicted_zone + sqrt(camera_zone) + used_zone^2) %>%
    select(matchup, predicted_zone, camera_zone, used_zone, transform_of_zones) %>%
    filter(!is.na(transform_of_zones)) %>%
    arrange(transform_of_zones, decreasing = TRUE) %>%
    head(n = 3)

##             matchup predicted_zone camera_zone used_zone transform_of_zones
## 1 Mariners vs Twins              1           1         1                   3
## 2 Mariners vs Twins              1           1         1                   3
## 3 Mariners vs Twins              1           1         1                   3
```

Note that the mutation adds a variable to an existing `data.frame`, but not in place. There are many useful transformations on data we can apply, e.g. creating lagged variables.

```
data %>%
    group_by(matchup) %>%
    mutate(
      last_game_played = lag(game_date),
      cumulative_velocity = cumsum(ifelse(is.na(exit_velocity), 0, exit_velocity))
    ) %>%
    select(matchup, game_date, last_game_played, exit_velocity, cumulative_velocity) %>%
    head(n = 4)

## # A tibble: 4 x 5
## # Groups:   matchup [1]
##   matchup           game_date  last_game_played exit_velocity cumulative_veloci˜
##   <chr>             <fct>      <fct>                    <dbl>              <dbl>
## 1 Mariners vs Twins 2019-05-18 <NA>                      NA                  0
## 2 Mariners vs Twins 2019-05-18 2019-05-18                NA                  0
## 3 Mariners vs Twins 2019-05-18 2019-05-18                56.9               56.9
## 4 Mariners vs Twins 2019-05-18 2019-05-18                78.8              136.
```

4.7 Tabulating two categorical fields

We might already be familiar that we can use `table()` to get a count of the number of times each value in the input appears. But we can also do this with more than just one argument! Here, we showcase using two arguments. There is one field called `predicted_zone` which describes a prediction for where the foul ball will end up, and another field called `camera_zone` which describe where the ball *actually* ended up, as observed by a camera. How good are our predictions? We can make a confusion matrix using `table()`.

```
confusion_matrix <- table(predicted = data$predicted_zone,
                          observed = data$camera_zone)
confusion_matrix[1:3, c(1:2, 4)]  # Just look at the first few rows and columns...
```

```
##           observed
## predicted   1   2   4
##         1 240   0   0
##         2   1   3   7
##         3   0   0   0
```

How do we read this table? When we predict, say, zone 2, it turns out that we once observed the ball land in zone one, three times in zone two, and seven times in zone four.

Exercise: Can you count the number of times that we got the "right" answer, programatically? I don't mean a rate, simply the sum total number of times that our prediction matched the observed outcome. Hint: consider using `sum()` and `diag()`.

Accuracy We see that we have fairly decent accuracy.

```
accuracy <- sum(diag(confusion_matrix)) / sum(confusion_matrix)
paste0("Accuracy = ", round(accuracy * 100, 2), "%")

## [1] "Accuracy = 96.95%"
```

4.8 Imputation

What if we want to replace our NA's with some data? There are many ways to do data imputation, but let's just cover the most basic syntax. Perhaps we want to fill is missing values for `exit_velocity` with the average exit-velocity. To do this:

```
missing_idx <- which(is.na(data$exit_velocity))
data[missing_idx, "exit_velocity"] <- mean(data$exit_velocity, na.rm = TRUE)
```

Note that there are several important steps going on here:

- We first create an integer vector describing the *indices* of missing elements in the `exit_velocity` vector. We've used `is.na()` to return a vector of Booleans, and we've used `which()` to tell us *which* of the Booleans evaluated to `TRUE`.
- When we make a replacement to our data.frame, we have to specify which rows and which column we are modifying. In this case, the syntax `data[missing_idx, ''exit_velocity'']` signifies that we're going to assign values to the missing entries in our column of interest.
- When we impute with the average value, we've used the argument `na.rm = TRUE` to calculate the average among the non-missing values of our input. If we removed this, it would default to `na.rm = FALSE`, and then we'd get an `NA` back as answer.

We could more easily do the above using `tidyimpute`... but that's for another day!

Exercise Can you replace `camera_zone` missing values with the most frequently recurring `camera_zone` value? Hint: see functions: `table()`, `which.max()`, `names()`, and `as.integer()`.

4.9 Other Ways of Getting `data.frames` Into R

There are a myriad of ways to get data into R, including things like reading from DataBase connections, but we'll mention some of the more handy methods.

4.9.1 Remote Files

Note that the argument (input) to the `read.csv` or `read.table` commands don't have to be local files, they can be remote files as well. We simply input the URL instead of the file-path. For example, the baseball data is also available on GitHub at `https://raw.githubusercontent.com/fivethirtyeight/data/master/foul-balls/foul-balls.csv`.

```
link  <- paste0("https://raw.githubusercontent.com/",
                "fivethirtyeight/data/master/foul-balls/",
                "foul-balls.csv")
fouls <- read.csv(link)
```

4.9.2 Copy Paste

One of my favorite ways of loading data into R is the copy-paste method, which can vary by platform.

```
name height weight
Andreas 70 174
Kristoffer 74 145
Monika 63 125
```

```
data <- read.delim("clipbloard")     # Works on Windows, Linux
data <- read.delim(pipe("pbpaste")) # Works on iOS.
```

To work with the code above, remember to add `sep = " "` argument to the `read.delim` command to inform R that the delimiter between columns is a space.

5 Aggregating and Reshaping Data

5.1 Aggregation of Data

There are ways to do this using base R only, but we leave this for an appendix.

5.1.1 Summarizing Multiple Columns

Although the `aggregate` function does have syntax to support applying *one* function to multiple columns (with one or more grouping columns), it's not that flexible: we can't apply different functions to different columns of our data. For that, we could *either* make two different calls to our `aggregate` function, and then merge the results, *or* we could use an R package that facilitates aggregation of data.

```
# install.packages("dplyr")
require(dplyr)
fouls %>%
    group_by(type_of_hit) %>%
    summarize(avg = mean(exit_velocity, na.rm = TRUE),
              std =   sd(exit_velocity, na.rm = TRUE))
```

```
## # A tibble: 5 x 3
##   type_of_hit        avg   std
##   <fct>            <dbl> <dbl>
## 1 Batter hits self  69.4  7.92
## 2 Fly               76.7 10.8
## 3 Ground            74.4 14.6
## 4 Line              82.2 17.1
## 5 Pop Up            74.3  5.99
```

5.2 Reshaping Data

It's often the case that you want to reshape your data in order to facilitate analysis or plotting.

Melting a `data.frame` into Long Format Let's take the dataset we just created in the last example, and see how we can reshape it from its current *wide* format into a *long* format. To do this, we'll make use of the `data.table` package.

```
install.packages("reshape2")
```

```
# Use "gather" to go from wide to long.
long <- fouls %>%
    group_by(type_of_hit) %>%
    # Ie we only executed summarize, we'd get one column per variable.
    summarize(avg = mean(exit_velocity, na.rm = TRUE),
              std =   sd(exit_velocity, na.rm = TRUE)) %>%
    # If we add a 'melt' command, we can collect values into a common column.
    reshape2::melt(id.vars = "type_of_hit")
print(long)
```

```
##         type_of_hit variable     value
## 1  Batter hits self      avg 69.380000
## 2               Fly      avg 76.737027
## 3            Ground      avg 74.402564
## 4              Line      avg 82.178571
## 5            Pop Up      avg 74.273913
## 6  Batter hits self      std  7.918775
## 7               Fly      std 10.810123
## 8            Ground      std 14.582302
## 9              Line      std 17.082776
## 10           Pop Up      std  5.992715
```

Casting a data to wide format If we want, we could undo the operation with a `spread` command to widen our data, using `pivot_wider` in `tidyr`, but I find the syntax pretty confusing. I prefer the lesser of two evils and choose `reshape2::dcast`.

```r
# Now do the opposite, go from long to wide via pivot_wider.
# For this, I personally prefer reshape2 package.
wide <- reshape2::dcast(long, type_of_hit ~ variable)
print(wide)
```

```
##        type_of_hit      avg       std
## 1 Batter hits self 69.38000  7.918775
## 2              Fly 76.73703 10.810123
## 3           Ground 74.40256 14.582302
## 4             Line 82.17857 17.082776
## 5           Pop Up 74.27391  5.992715
```

6 Putting It All Together

Let's take a second to go over a simple statistical analysis, to put what we've learned together and see an R analysis wholistically.

6.1 Data Collection, Ingestion

It starts with data collection. Here, we show how you can actually use a `download.file()` function to fetch a `.zip` file from the web, then programatically `unzip()` it.

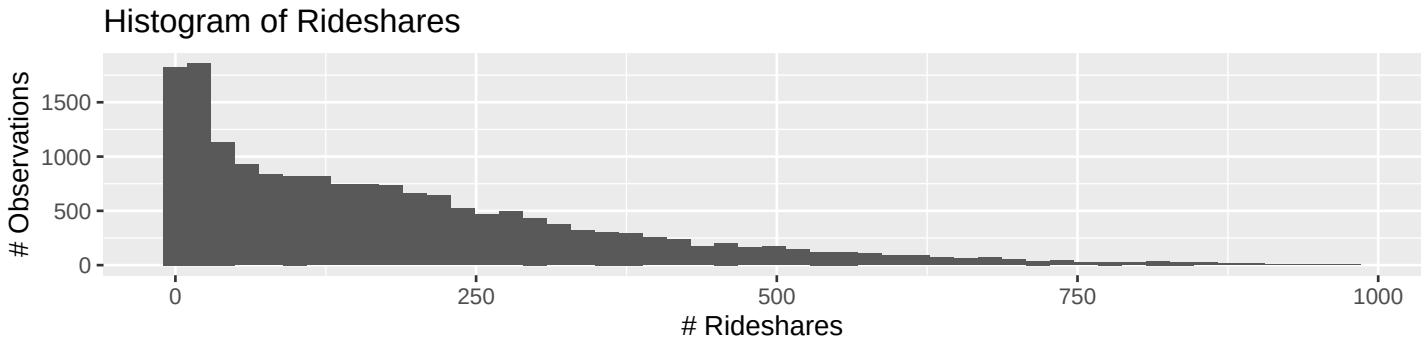
```r
# Download a zip file of bike-share data.
download.file(paste0("http://archive.ics.uci.edu/ml/machine-learning-databases/00275/",
                     "Bike-Sharing-Dataset.zip"),
              destfile = "bikesharing.zip")
# Unzip the contents and create a corresponding data directory. Load, and inspect.
unzip(zipfile = "bikesharing.zip", exdir = "bikesharing")
bshare <- read.csv("bikesharing/hour.csv")
bshare %>% slice(1:3) %>% select(1:8)
```

```
##   instant     dteday season yr mnth hr holiday weekday
## 1       1 2011-01-01      1  0    1  0       0       6
## 2       2 2011-01-01      1  0    1  1       0       6
## 3       3 2011-01-01      1  0    1  2       0       6
```

6.2 Data Visualization

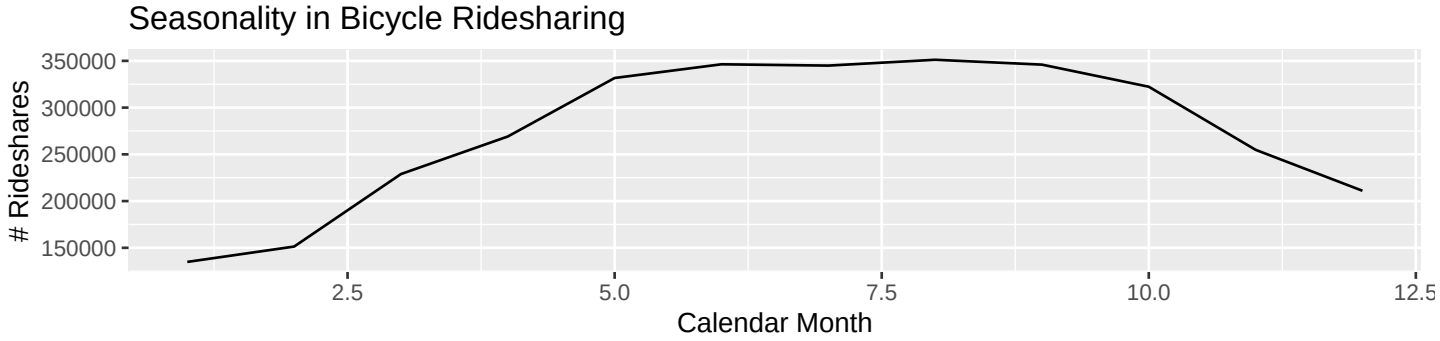
What's the univariate distribution of rideshares? See `?hist` for documentation.

```r
ggplot(bshare, aes(x = cnt)) +
    geom_histogram(bins = 50) +
    labs(title = "Histogram of Rideshares",
         x = "# Rideshares",
         y = "# Observations")
```



The fact that the number of rideshares has non-trivial range which spans orders of magnitude suggests that a *log-transform* may be appropriate in a linear model. Our date-variable is already pulled apart with year, month, day, hour and metafeatures such as holiday or workday.

```
bshare %>%
    group_by(mnth) %>%
    summarise(ttl = sum(cnt)) %>%
    ggplot(aes(x = mnth, y = ttl)) +
    geom_line() +
    labs(title = "Seasonality in Bicycle Ridesharing",
         x = "Calendar Month",
         y = "# Rideshares")
```



6.3 A First Model

We get excited, seeing that there is a relationship between rideshares and calendar month. Let's try building a haphazard *linear-model* using `lm()`, which estimates the marginal effects of each variable on a response under some assumptions, which are details of interest for a different workshop.

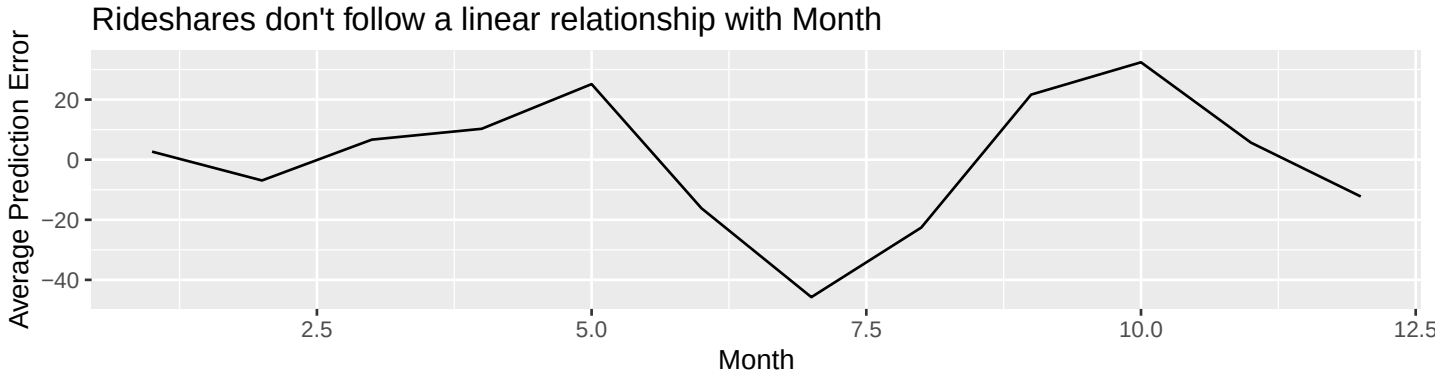
```
m <- lm(cnt ~ mnth + hr + holiday + weekday + workingday + temp + hum + windspeed,
        data = bshare)
```

In this linear model, we've specified that we want to regress (or model) the response variable of counts as a function of the month, hour-of-day, holiday indicator, weekday level, workingday indicator, temperature, humidity, and windspeed. There's potentially useful output here if we inspect `summary(m)`, but not quite yet because we have *violated* the linear modeling assumptions. Note that in the `formulae` notation, when we write `+` we're not really saing add the variables together, we're rather saying to simpy include them.

6.4 Checking Linear Model Assumptions

Garbage-in, garbage-out, however: our linear model assumptions were violated.

```
# Determine if any non-linear relationships were left out
# of the month feature.
bshare %>%
    mutate(residual = resid(m)) %>%
    group_by(mnth) %>%
    summarise(mean_residual = mean(residual)) %>%
    ggplot(aes(x = mnth, y = mean_residual)) +
    geom_line() +
    labs(title = "Rideshares don't follow a linear relationship with Month",
         x = "Month",
         y = "Average Prediction Error")
```



Yikes! We totally ignored seasonality, even though it looked like our `month` variable was statistically significant. The problem lies within the formatting of our data: the `month` variable was left as `integer`, wherein we assume that for example there is a linear relationship in ridesharing with respect to calendar month, which doesn't really make sense. Let's try again with a *factor* coding. See `?factor`, and `?I`.

```
m <- lm(cnt ~ I(factor(mnth)) + hr + holiday + weekday + workingday + temp + hum
        + windspeed, data = bshare)

# Verify non-linear relationships have been explicitly modeled.
bshare %>%
    mutate(residual = resid(m)) %>%
    group_by(mnth) %>%
    summarise(mean_residual = mean(residual)) %>%
    summarise(all(mean_residual < 1e-7))

## # A tibble: 1 x 1
##   `all(mean_residual < 1e-07)`
##   <lgl>
## 1 TRUE
```

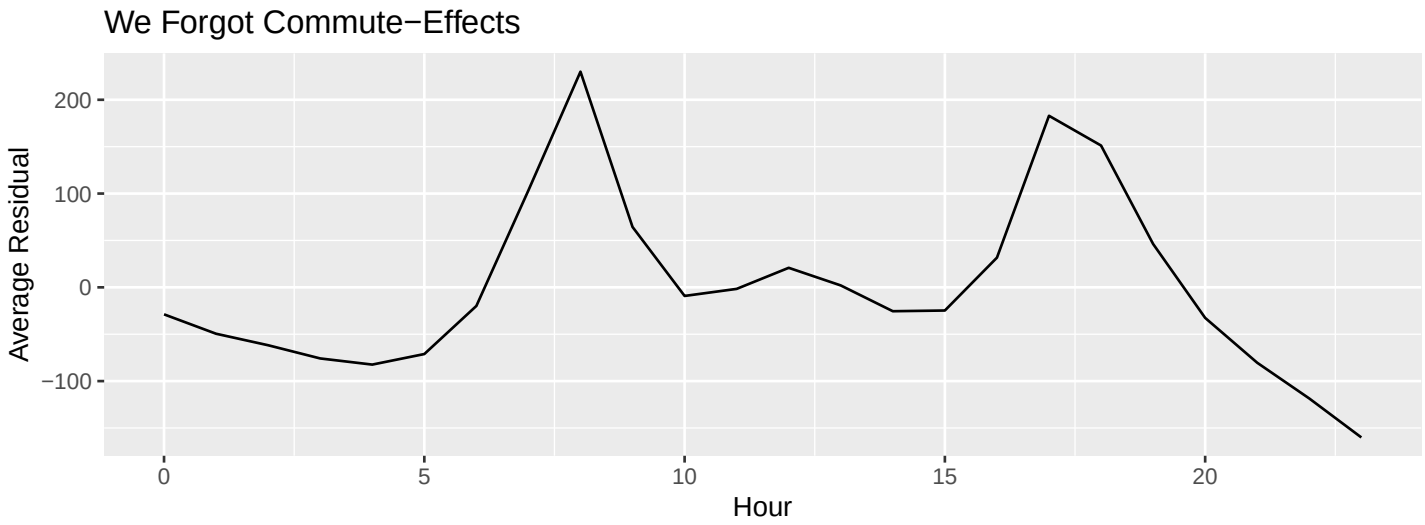
Ah, there we go; much better! Here, we've encoded an indicator variable for each month, leaving out the first level by default (January in this case).

6.5 Iterative Model Refinement

What about our `hour` variable? It's also significant. But the same problem was made! There isn't expected to be a linear relation between hour of day and number of bicycle rideshares. There's more than likely different fixed effects, with a possibly sinusoidal pattern as a function of daylight. Easiest is to simply

encode fixed effects for each hour via a factor, as before. In fact, we can use a similar code to verify the same is true for `weekday`, so we take care of that as well.

```
# Very that our linearity assumption was violated; we see clear commute and
# day-of-week effects.
bshare %>%
    mutate(residual = resid(m)) %>%
    group_by(hr) %>%
    summarise(mean_residual = mean(residual)) %>%
    ggplot(aes(x = hr, y = mean_residual)) +
    geom_line() +
    labs(title = "We Forgot Commute-Effects",
         x = "Hour",
         y = "Average Residual")
```



We've clearly left unexplained variation in the response on the table. Replacing factors for relevant integer-coded variables, including `workingday`.

```
m <- lm(cnt ~ I(factor(mnth)) + I(factor(hr)) + I(factor(weekday)) +
        I(factor(workingday)) + holiday + temp + hum + windspeed, data = bshare)
# summary(m)  <-- This prints useful output but its a bit verbose.
```

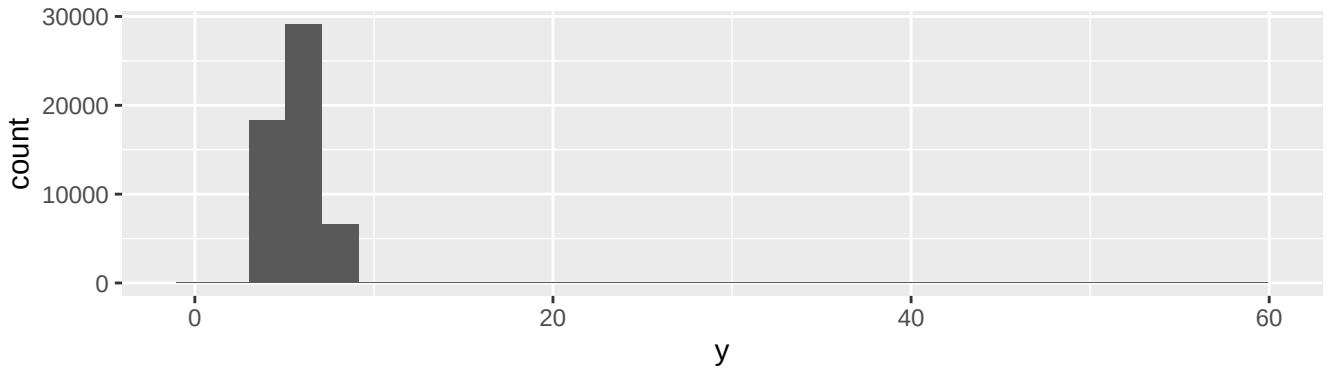
Do you notice that in the above model summary, one of the coefficients is listed as Not Available (i.e. `NA`)? To learn why, take our Stats workshop![4]

6.6 Applied Example: Handling Outliers

When we're analyzing data, we sometimes come across outliers in our dataset. They can usually be spotted quite easily in univariate distributions by large "gaps" in the x-axis without any mass on the y-axis.

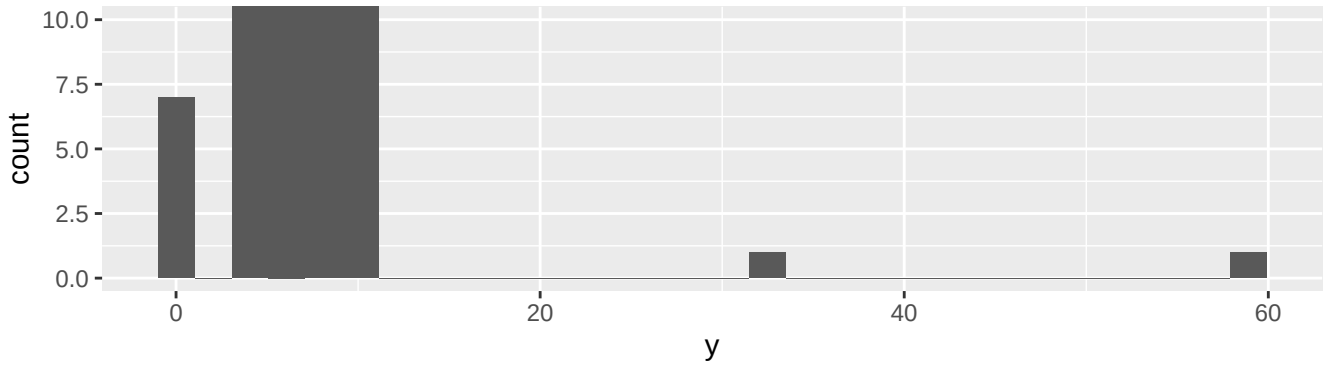
```
ggplot(diamonds, aes(x = y)) +
    geom_histogram()
```

[4] In short: collinearity.



Similar to our introductory plotting example where we couldn't see exactly what was happening in the data without a modification to our plot, the same is true here.

```
ggplot(diamonds, aes(x = y)) +
    geom_histogram() +
    # Truncate the y-axis to [0, 10] to help see the mass at x = 0, 32, and 59.
    coord_cartesian(ylim = c(0, 10))
```

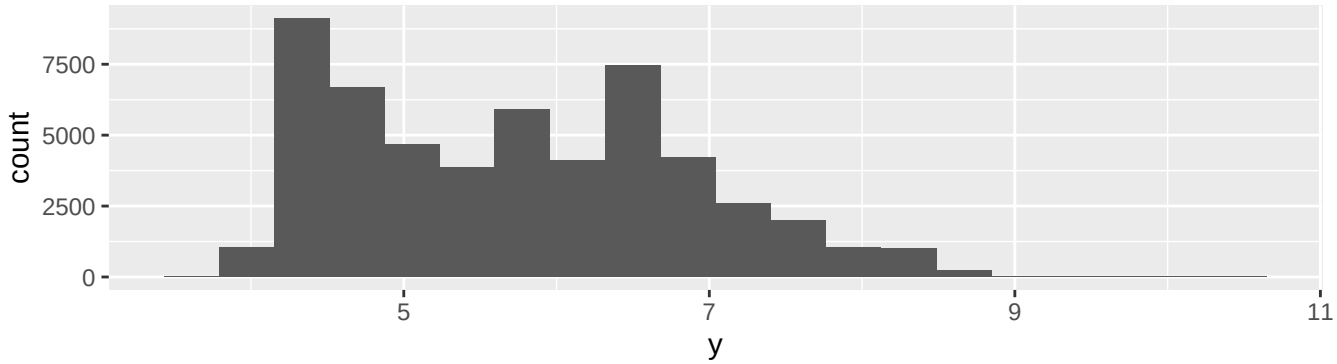


We've used the `coord_cartesian()` function in order to truncate the y-axis without discarding any data. Now we can see that there are more than one observations on the x-axis at the points 0, 32, and 59.

6.6.1 Replacing Outliers with Missing Values

One technique to handling outliers, rather than throwing out the observations entirely, is to retain the remaining feature values for the observation but replace the outlier feature value with a missing-value. We can easily do this using an `ifelse()` statement within a call to `mutate()`:

```
diamonds %>%
    mutate(y = ifelse(y < 3 | y > 20, NA, y)) %>%
    ggplot(aes(x = y)) +
    geom_histogram(bins = 20)
```



Notice that the range that the variable `y` takes on is now much more constrained. This can help avoid spurious results when taking averages, performing statistical analyses, or even machine learning.

7 What Next?

This course is a simple introduction, and we hope to have sparked an appetite for learning more about statistical computing (in R)! We've include some resources in the bibliography below.

Nailing the Basics I found that R in a Nutshell was a great thorough reference.

Packages in R To install a package for the first time (after becoming aware of its existence through a web search or word of mouth), we may type `install.packages("package_name")` to download and install the binary files required to call upon the package contents. We then must type `library("package_name")` at the start of each R session in which we wish to use said package. If you wish to see a listing of available objects that come exported with a package, try `help(package = "<package_name>")`; from there, you can inquire for further help on individual function `help(package_name::function_name)`.

Tidyverse Recently popular in the R programming community, the Advanced R book by Hadley Wickham (click the table-of-contents in the upper left corner to access full course contents) seeks to further enable interactive statistical programming. The packages rely on the `magrittr` package to pipe computations forward in a more readable manner. (For a more introductory but also highly recommended guide to the `tidyverse`, check out R for Data Science!)

Visualization The package `ggplot2`, also by Hadley, provides a grammar of graphics approach to data visualization. From there, you can easily learn animations via `gganimate`, or even plotting geospatial (mapping) data via `ggmap`, e.g. see this fun example here.

Performant Computing In a completely different direction, one of my favorite packages in R is `data.table`, authored by Matt Dowle. It supports assignment by reference, and allows us to compute on larger data sets more efficiently. Advantages we can already see based on examples above: each call to `aggregate()` can be replaced by a `group-by` operation within the `data.table`, where we can take advantage of scoping to forego typing `dt$` constantly to select variables. If you're interested in performant computing in R, CRAN has an entire page dedicated to this subject, pointing to libraries geared toward computing with large data efficiently.

Pipelines / Reproducibility If you're writing a collection of R scripts that are all related, you may consider bundling them into your own R package. You can then add unit-tests easily via `testthat`, and have these run automatically each time you rebuild your package. If you want to be really fancy, you can

integrate these unit tests in with your Git workflow via Continuous Integration, where we emphasize the aspect of having the tests re-run with each new commit (e.g. Travis).

Interfacing with a Database There are a myriad of ways that R can interface directly with databases. This can be nice in that you can write SQL queries programmatically in R and then pull the results directly into memory. See database queries with R.

8 Appendix

8.1 Essential Data Structures

8.1.1 Vector (Operations)

We've seen how we can create vectors using `c()`, `:`, and functions like `seq` or `rnorm`.

```
u <- c(1, 2, 4, 8, 16)
v <- seq(from = 0, to = 1, length.out = length(u))  # (0, 1/4, 1/2, 3/4, 1)
```

Exercise: We used the argument `length.out` within the `seq()` function call which was used to create vector `v` above. Create a vector called `odds` which generates all non-negative odd values less than 100. Hint: see `?seq` and specifically the argument `by`.

Indexing into Vectors using Extraction Operator There is a handy extraction operator that allows us to grab particular elements from a vector in a variety of ways, and this is useful.

```
u[3]        # R is 1-indexed.

## [1] 4

u[c(3, 5)]  # Grabs the third and fifth element from 'u'.

## [1]  4 16

u[3] <- 0   # We may access and overwrite individual elems.
```

For details on this operator, see: `help("[")`.

Exercise: Replace the 10th odd number with a zero, then calculate the sum of the resulting vector. What answer do you get?

```
rbind(u, v) # We can also row-bind two vectors to get a matrix; see below.

##   [,1] [,2] [,3] [,4] [,5]
## u    1 2.00  0.0 8.00   16
## v    0 0.25  0.5 0.75    1

u + v       # Vector addition is performed element-wise.

## [1]  1.00  2.25  0.50  8.75 17.00
```

Vector Arithmetic The last expression demonstrates vector arithmetic behaves in the obvious way, i.e. element-wise.

Boolean Filtering Note that we can also use a logical predicate condition to filter a vector, e.g.

```
vals <- rnorm(n = 100)
negs <- vals[vals < 0]  # Only keep 'vals' that are negative.
summary(negs)           # Notice the maximum is negative, by construction.

##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -2.21470 -0.90948 -0.57097 -0.66069 -0.18673 -0.01619
```

In the example above, we've first created a vector of 100 normally distributed random variables. Then, we've created a logical filter by indexing into our `vals` using extraction operator `[`. In particular, the input we feed to the extraction operator is a *Boolean* vector describing whether each element in `vals` is negative.

```
head(vals < 0)  # Notice that vals < 0 returns a Boolean vector.

## [1]  TRUE FALSE  TRUE FALSE FALSE  TRUE

data <- cbind(original = vals, predicate = vals < 0)
head(data)

##        original predicate
## [1,] -0.6264538         1
## [2,]  0.1836433         0
## [3,] -0.8356286         1
## [4,]  1.5952808         0
## [5,]  0.3295078         0
## [6,] -0.8204684         1
```

We remark that a matrix must be homogeneous in the type of data stored, which is why our Boolean vector of Trues and Falses got converted to 1's and 0's respectively.

Ifelse We might have previously run into `if() {} else {}` statements before. These work when the predicate condition to the `if()` statement is a scalar value (i.e. length one vector). If you want to apply `if-else` filtering on a vector, you probably want R's builtin `ifelse`, is just like `if() {} else{}` in other languages except it is *vectorized*. Example:

```
data <- 1:10
ifelse(data < 5, 0, data)

##  [1]  0  0  0  0  5  6  7  8  9 10
```

Here, we first created a Boolean vector using the expression `data < 5`. Then, we said: *if* the condition is *true*, emit a zero, else emit the corresponding value in the `data` vector.

Exercise: Use your vector of non-negative `odd` numbers less than 100 from the previous exercise. Replace all odd numbers less than 50 with half their value.

8.1.2 Matrices

A matrix (in R) is just a vector with attributes (number of rows, number of columns, and a corresponding stride-length). To define a `matrix` object, we use the aptly named function, `matrix()`.

```
mat <- matrix(data = 1:9, nrow = 3)
print(mat)

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

Exercise Place the numbers `31` through `60` in a vector p. Place these same numbers in a matrix, call it Q. *Hint:* examine your *workspace* window *after* creating these objects. How are they similar, or different?

```
p <- 31:60
Q <- matrix(31:60, nrow = 5, ncol = 6)
```

Matrix Extraction We can access elements using the extraction operator '`[`' just as with vectors. I.e. we access element `i,j` of a matrix `mat` via `mat[i,j]`.

Exercise What is the value of the 15th element in p, created in the previous exercise? Print this to console. Now, print the element in the 5th row and 3rd column of matrix Q, also created previously.

Selecting entire rows or columns from a matrix If you wish to select all of row `i`, we simply omit the column index (e.g. `mat[i, ]`), and conversely if we wish to select an entire column.

Exercise: What is the mean value of the second column of our matrix `Q` created above? Although it's easy to eyeball, please calculate the expression using R. What about the mean value of the second row? Again, same principle in coding the solution to verify what's immediately obvious from eyeballing the data.

Functions and Matrices Many functions also work with matrices as argument. E.g. we can calulate the sum total of elements in a matrix using `sum(mat)`.

Exercise What is the *mean* value in vector p? What about the *mean* value in matrix Q? Observe the result if we try computing other basic statistics, such as `min()`, `max()`, or `median()`. Does it make sense to you why these functions work with matrices?

Efficient Operations on Matrices R has many built-in functions which make matrix operations simple and performant. E.g. we can calculate column-sums or column-means via `colSums()` and `colMeans()` respectively.

Exercise What are the column-sums of Q? What about the column-means? Hint: see `?colSums`.

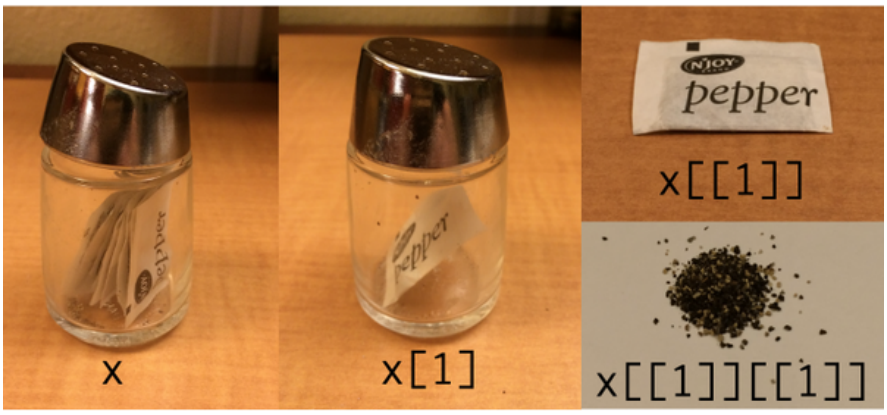
8.1.3 Lists

Lists are *the* fundamental data-structure in R. In general, the elements of a list *don't* have to be the same length. And unlike matrices, the elements don't have to be of the same *type* either.

```
l <- list(one = 1, two = 1:2, five = seq(0, 1, length.out = 5))

## $one
## [1] 1
##
## $two
## [1] 1 2
##
## $five
## [1] 0.00 0.25 0.50 0.75 1.00

l$five + 10

## [1] 10.00 10.25 10.50 10.75 11.00

l$newVar <- sin(l$five)  # We can assign a new element to a list on the fly.
```

Indexing into (possibly nested) lists can be difficult for beginner R users. See a satirical post.

```
# Create some salt and pepper packages.
# (Each contains a different number of particles and ratio of salt-pepper)
sap <- list(package_A = list(serv_one = rbinom(20, 1, prob = 50/100),
                             serv_two = rbinom(14, 1, prob = 30/100)),
            package_B = rbinom(28, 1, prob = 55/100),
            package_C = rbinom(36, 1, prob = 40/100))
```



Note that the `rbinom(n, size, prob)` above generates random variables from a binomial distribution where `n` is the number of observations, `size` is the number of trials within each observation, and `prob` is the probability of success on each trial.

Exercise Discuss the (non-rigorous) relationship between the salt-and-pepper graphic and the list `sap` (created above) with your neighbor. What's the difference between `x[1]` and `x[[1]]` in the graphic (and correspondingly in our `sap` object)?

More on List Extraction We emphasize that using the '`[[`' extract operator for lists, there are two methods of access: by name and by position. I.e. `sap[["package_C"]]` is equivalent to `sap[[3]]` since we happened to place `package_C` in the third element of our list `sap`.

Exercise Write down three different ways of accessing the third element of our list `l` created above. Verify that they produce an identical result. Hint: one of them uses `[[` with an integer argument, another uses `[[` with a string argument, and the third uses the name of the element with the `$` extraction operator.

Exercise Replace the `0.25` with a `0` in the vector of (five) elements that are named "five" in our list `l` above. Hint: you may do this in a variety of different ways based on what we've seen in the course so far.

8.2 Control Flow

This next section will be particularly useful to those entirely new to programming.

8.2.1 Functions

We can define our own functions in R using the following syntax:

```
Greeting <- function(name, salutation = "Hello", punctuation = "!") {
    paste0(salutation, ", ", name, punctuation)
}
```

The function `Greeting` requires a `name` argument, but has default arguments set for the leading `salutation` and the trailing `punctuation` mark. We can override these by supplying the arguments explicitly. hold

```
Greeting("Andreas",  salutation = "god morgen")
Greeting("Santucci", salutation = "buon giorno")

## [1] "god morgen, Andreas!"
## [1] "buon giorno, Santucci!"
```

It's always the case that the *last expression* we use in the definition of a function acts as the return or output. We can also specify explicitly the return value; i.e. we could have defined our function as:

```
Greeting <- function(name, salutation = "Hello", punctuation = "!") {
    return(paste0(salutation, ", ", name, punctuation))
}
```

Exercise Write a function to compute the value of $2x + 3$.

```
MyFunction <- function(x) {}
```

Exercise What happens if you input a vector to your function? Can you predict what the result will be? *Hint:* vectors and recycling.

```
MyFunction(1:10)
```

Exercise Write a function to compute the number of days since you were born. Your function should adhere to the following prototype; you may find the helper function `Sys.Date()` handy, and also note that you can perform arithmetic on dates.

```
DaysSinceBorn <- function(birthdate = as.Date("1989-09-04")) {}
```

Examples from Statistics E.g. we know that we can define R-Squared for non-linear models by looking at the square of the correlation coefficient between the response y and fitted values $\hat{y}$.

```
RSQU <- function(y, x)    cor(y, x)^2               # Univariate case.
RSQM <- function(y, yhat) 1 - sum((y - yhat)^2) /   # OLS: 1 - RSS / TSS.
                              sum((y - mean(y))^2)
RSQG <- function(y, yhat) cor(y, yhat)^2            # General model.
```

Don't worry if you haven't seen some of these definitions before; we hope perhaps one is familiar.

Comparisons The expression `a > b` for two comparable objects will return a `TRUE` if `a` is in fact strictly larger than `b`, and `FALSE` otherwise.

```
x <- c(1:10)
x > 8
```

Exercise Write a function called `CheckAgainstThreshold(x, K=1)` which, given an input x and a threshold K (defaulting to unit value), return the result of $x > K$. Try calling it with inputs of zero, one, and two individually (keeping the threshold at $K = 1$). Additionally, pass a vector of inputs `c(0, 1, 2)`; can you explain the result to your neighbor?

Exercise* Write a function called `CheckID` which accepts a birthdate as argument and returns a Boolean indicating whether the corresponding person is at least K years of age, for K defaulting to 18. *Hints:* (i) use `as.Date()` to cast a string to a `Date` object, and (ii) using the function `Sys.date()` may be helpful, (iii) use the rule of thumb that there are 365.25 days in a year.

```
CheckID <- function(birthdate, K = 18) {
    # Your job is to fill in a sub-routine here.
    # Return true only if birthdate at least K years.
}
```

Indexing via Booleans We've seen previously that we can index into an object (vector, matrix, list) via individual indices. E.g. `vec[i]` or `mat[i,j]`. This will work for any i, j that are in-bounds and of the same length; we exract a sub-matrix of dimension `length(i)` $\times$ `length(j)`. We've also seen in the previous section that the `>` operator can be used to compare a vector against a single value, returning a vector of boolean values as a result.

We can also subset into a vector or matrix using logical conditions. E.g.

```r
set.seed(20210813)
x <- runif(300)    # Sample from [0,1] uniformly at random.
idx <- x < 1/2     # In expectation, 1/2 of these evaluate TRUE.
# Not run: x[idx] (Return elements of 'x' for which 'idx' is TRUE.)
```

Function `which()` and `operator==()` You also may find the function `which()` to be helpful when working with boolean vectors: it returns the indices of the elements whose values are `true`. Lastly, we mention that the logical negation operator `!` can invert boolean relations.

```r
v = c(1,3,5,7,8)
which(v%%2 == 0)

## [1] 5
```

```r
which(v%%2 != 0)

## [1] 1 2 3 4
```

```r
sum(idx)
mean(idx)


## [1] 144
## [1] 0.48
```

Boolean Arithmetic You may find it interesting that we can perform arithmetic on booleans. I.e. `sum(idx)` and `mean(idx)` yield the count and proportion of values which satisfy the predicate. I.e. we count `FALSE` as zero and `TRUE` as unit valued.

Exercise Do the following expressions make sense? Discuss with your neighbor what is happening.

```r
FALSE + TRUE == 1L
TRUE  + TRUE == 2L
mean(c(FALSE, TRUE))
# What is the value of the following expression, in expectation?
mean(rnorm(1e3) < 0)
```

We'll revisit another application of Boolean arithmetic when we cover `data.frames`.

8.3 For-Loops and Apply Functionals

Let's continue using our foul-balls dataset.

```
link  <- paste0("https://raw.githubusercontent.com/",
                "fivethirtyeight/data/master/foul-balls/",
                "foul-balls.csv")
data <- read.csv(link)
```

8.3.1 Control Flow with Data.Frames

What if we wanted to work with a complete set of data, rather than an object that contains some missing values? Recall that if we use `str()`, this command will inform us of how many missing values are in each column. We could also ask the same question a different way, by *looping* over our columns and summing over the number of missing values.

```
for (column in colnames(data))
    print(paste("Column", column, "has",
                sum(is.na(data[[column]])),
                "missing values"))
```

```
## [1] "Column matchup has 0 missing values"
## [1] "Column game_date has 0 missing values"
## [1] "Column type_of_hit has 0 missing values"
## [1] "Column exit_velocity has 326 missing values"
## [1] "Column predicted_zone has 0 missing values"
## [1] "Column camera_zone has 513 missing values"
## [1] "Column used_zone has 0 missing values"
```

Let's break this down a bit.

- The function `colnames()` returns the column-names for a `data.frame`.
- The syntax `for (var in x)` creates a local variable called `var` whose value gets updated each *iteration* to reflect the next element in `x`.
- We then have a `print` statement such that we can inspect the output.
- We use `paste` such that we can combine some text and some numbers.
- Lastly, we use `sum(is.na())` to count the number of missing values in a column. In particular, `is.na()` returns a logical vector of `TRUE`s and `FALSE`s, and `sum()` takes this as input and counts how many values are `TRUE`.

Storing Results From `for`-loop What if we didn't want to print the results to console, we just wanted to *concatenate* the results into a vector? I.e. we suppose we only want to store the number of missing values in each column, and we don't care about printing out this data; we just want to store it in a variable. We might write something like:

```
result <- c()
for (column in colnames(data))
    result <- c(result, sum(is.na(data[[column]])))
```

At each iteration, we've appended an entry to our `result` describing the number of missing entries in the corresponding column of our `data.frame`.

Pre-Allocating Storage Constantly re-sizing an output vector can be a little bit computationally expensive. It's cheaper to first pre-allocate the vector, then fill it with data. I.e. instead of appending to an ever-growing vector, we place values into the appropriate positions of a pre-allocated array.

```
result <- vector(mode = "numeric", length = ncol(data))
for (i in seq_along(data))
    result[i] <- sum(is.na(data[[i]]))
```

Here, we've used `seq_along(data)`, which is just like getting a vector `1:ncol(data)`; we then place values into our result at the appropriate spot by using `result[i]` to access the `i`th element of the `result` vector. Similarly, calling `data[[i]]` gets us the ith column of our `data.frame`.

Exercise: Calculate the number of unique elements in each column. You may want to consider using the functions `length()` and `unique()`. You can use either an append or a pre-allocate strategy.

An easier way: `sapply` There's an easier way to wrap up a for-loop + computation. This pattern appears so often in R that the language authors created a shorthand for the pattern. It's called the `apply` family of functions.

```
sapply(colnames(data), function(colname)
    sum(is.na(data[[colname]])))
```

```
##        matchup      game_date    type_of_hit  exit_velocity predicted_zone
##              0              0              0            326              0
##    camera_zone      used_zone
##            513              0
```

The resulting output is a *named* vector, where each value in the vector describes the number of missing elements in the column.

Exercise: Use an `sapply` functional to calculate the unique number of elements in each column.

8.3.2 Apply Functionals

We can apply a function to each column of our data.frame.

```
NUnique <- function(x) length(unique(x))
uniques <- sapply(fouls, NUnique)
```

`sapply` iterates over its first argument (the "input"), and applies the *function* provided as second argument to each of the inputs. In this case, the input is a set of columns, so we iterate over each column and apply our function `NUnique` one column at a time.

Exercise Write a function which only returns the numeric columns in a data.frame. You should use either a `for`-loop or an `sapply` to iterate over the columns of the data.frame and check whether each is a numeric column or not. You can use the function `is.numeric()` for this purpose. We remark that as a final step, you'll need to subset your data.frame using one of the techniques we've covered previously. Your function should adhere to the following prototype:

```
NumericCols <- function(df) {}
```

You can ue the foul-balls dataset to test your function!

Exercise Use this function to transform all of the numeric columns according to a transformation of your choosing. Examples include $f(x) = log(x)$, $f(x) = sin(x)$, $f(x) = x + 10$. You should first encapsulate your function into a definition, then use a `for`-loop or an `sapply()` to apply the function to each column returned by `NumericCols()`. Optionally, you can not only create transformed columns but assign them to new columns in your existing data.frame. You may find it helpful to use the objects:

```
require(magrittr)  # We load this package so we can use %>%.
cols <- sapply(fouls, is.numeric) %>%
    Filter(f = identity) %>%
    names()
transformed_cols = paste("transformed", cols)
```

Here, we've used a couple of interesting tactics. The first is that we've started to use the *pipe* operator `%>%` in order to pass the output from one operation as the first input input argument into the subsequent operation. The second is that we've filtered the result from the first line using a trivial identity function, i.e. `is.numeric()` returned a boolean of `TRUE`s and `FALSE`s, and we are simply filtering out the values which are `FALSE`. Lastly, ever since we called `is.numeric()`, we've been dealing with a *named*-vector, and so as a last step we extract the names for which the previous operation(s) returned `TRUE`, i.e. we extract the names for the numeric columns!

This is not a contrived example; lots of times in machine learning we want to apply a transformation such as *log* to a set of columns if they describe random variables with support spanning the positive half-space, or we might apply a `logit` transform on columns which span the range of $[0, 1]$.

Function Composition We've seen several examples of function composition in these notes, most recently: `length(unique())`. There is a functional programming package called `purrr` that allows us to compose functions together!

```
require(purrr)
sapply(fouls, compose(length, unique))

##        matchup      game_date    type_of_hit  exit_velocity predicted_zone
##             10             10              5            320              7
##    camera_zone      used_zone
##              8              7
```

We mention this simply becuase it's syntactic sugar and it's cool, not because there are performance benefits.

8.3.3 Loading Multiple Files at a Time

What if we have a folder that contains a bunch of files, all related to each other, such as financial time series data broken apart by year. Consider BLS Employment and Wages data, available by *quarter*. We download a zip file for 2018 from `https://www.bls.gov/cew/downloadable-data-files.htm` (scroll down a bit and you'll see links for "CSVs by Area"). We can do this programatically:

```
download.file(
  file.path(
    "https://data.bls.gov/cew/data/files/2018/csv/",
    "2018_qtrly_by_area.zip"
  ),
  "~/Downloads/2018.q1-q4.by_area.zip"
)
unzip("~/Downloads/2018.q1-q4.by_area.zip",
      exdir = "~/Downloads/2018.q1-q4.by_area")
```

Now, the resulting directory contains over 4k `csv` files! How are we going to process them all? We can use an `apply` functional and pass it a `read.csv` function as argument! This allows us to read each file. To combine them, we can row-bind them together.

```
# There are 4,428 files in the directory!
fnames <- list.files(path = "~/Downloads/2018.q1-q4.by_area/2018.q1-q4.by_area",
                     full.names = TRUE)
d <- lapply(fnames[3:5], read.csv) %>%  # Load several files...
    rbindlist()                         # ...and "stack" them together.
```

The function `lapply()` works similarly as `sapply()`, returning a list (which explains the "l") by applying a given function to elements of a list. The function `rbindlist()` takes in a *list* of `data.frame`'s and row-binds them together into one larger `data.frame` object.

Exercise Instead of using `lapply()` and `rbindlist()`, can you come up with a way to create the same result using a `for`-loop and `rbind()`? Hint: first create an empty-data.frame, then iterate over a few file-names (be careful not to try and load more than a handful at once!) and for each: load it into memory using `read.csv()`, and then use `rbind()` to row-bind your (initially empty) data.frame to the newly read in data. Second hint: what happens when you `rbind()` an empty data.frame with a non-empty one? Can this simplify the logic of your `for`-loop?

8.3.4 Aggregation of Data

Another common theme is data aggregation. Base R can handle this, but there are packages that make it easier. Let's first quickly review the `aggregate` *functional*, which takes a function used to summarize (or aggregate) the as argument! The syntax looks as follows:

```
aggregate(column you want to summarize,
          a _list_ of columns you want to group by,
          function you want to apply,
          optional arguments for the function)
```

To see this in action, let's look at the average exit velocity as a function of the type-of-hit. Since there are `NA` values in the vector `exit_velocity`, we use `na.rm = TRUE` to ignore these observations when calculating our average.

```
aggregate(
  x = fouls[["exit_velocity"]],
  by = list(fouls$type_of_hit),
  FUN = mean,
```

```
  na.rm = TRUE
)
```

```
##            Group.1        x
## 1 Batter hits self 69.38000
## 2              Fly 76.73703
## 3           Ground 74.40256
## 4             Line 82.17857
## 5           Pop Up 74.27391
```

You could also do this same thing with a `for`-loop, but the above is syntactically more concise.

Aggregating by Multiple Columns You might be wondering, why does this `aggregate` function require a *list* of columns for the by argument? Well, this actually works to our advantage when we want to group-by multiple columns. E.g.

```
aggregate(
  x = fouls[["exit_velocity"]],
  by = fouls[, c("type_of_hit", "predicted_zone")],
  FUN = mean,
  na.rm = TRUE
)
```

```
##         type_of_hit predicted_zone         x
## 1  Batter hits self              1  69.38000
## 2              Fly              1  74.83968
## 3           Ground              1  67.46296
## 4             Line              1  69.10000
## 5           Pop Up              1  74.27391
## 6              Fly              2  73.39268
## 7           Ground              2  67.36471
## 8              Fly              3  72.09756
## 9           Ground              3  69.80000
## 10            Line              3  67.10000
## 11             Fly              4  76.68333
## 12          Ground              4  81.74643
## 13            Line              4  94.22222
## 14             Fly              5  78.72400
## 15          Ground              5  78.36944
## 16            Line              5  84.58571
## 17             Fly              6 100.85000
## 18             Fly              7  98.66000
```

Exercise: Discuss with your working group *why* we can provide a `data.frame` with multiple columns in place of a list within the `by` argument of the `aggregate` function above. Hint: what's the relationship between a `data.frame` and a `list`?

`data.table` package What if we want to calculate the mean *and* standard-deviation for each type of hit? For this, I personally find `data.table` package to really shine.

```
# install.packages("data.table")
require(data.table)
setDT(fouls)  # Convert the data.frame to a data.table
fouls[, .(avg = mean(exit_velocity, na.rm = TRUE),
          std =   sd(exit_velocity, na.rm = TRUE)),
      by = type_of_hit]

##         type_of_hit      avg       std
## 1:           Ground 74.40256 14.582302
## 2:              Fly 76.73703 10.810123
## 3:             Line 82.17857 17.082776
## 4: Batter hits self 69.38000  7.918775
## 5:           Pop Up 74.27391  5.992715
```

There are some details on the syntax that are well beyond the scope of the course, but I would argue it's relatively easy to read.

8.4 Linear Modeling

8.4.1 Regression

Let's introduce some basic modeling techniques. We start with linear regression. In R, the way to construct a linear model is using the command `lm()`, it accepts a formula and a `data.fame`.

```
m <- lm(formula = cty ~ displ, data = mpg)
```

Here, we've fit a regression where we explain the variation in city miles-per-gallon as a function of displacement. We should expect a negative marginal effect, as per our plot above.

```
summary(m)

##
## Call:
## lm(formula = cty ~ displ, data = mpg)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -6.3109 -1.4695 -0.2566  1.1087 14.0064
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  25.9915     0.4821   53.91   <2e-16 ***
## displ        -2.6305     0.1302  -20.20   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.567 on 232 degrees of freedom
## Multiple R-squared:  0.6376,Adjusted R-squared:  0.6361
## F-statistic: 408.2 on 1 and 232 DF,  p-value: < 2.2e-16
```

Exercise: What's the interpretation of our `displ` coefficient in the model that we fit above?

Model Diagnositics R has a lot of facilities for diagnosing linear models, for example, we can run

```
plot(m)
```

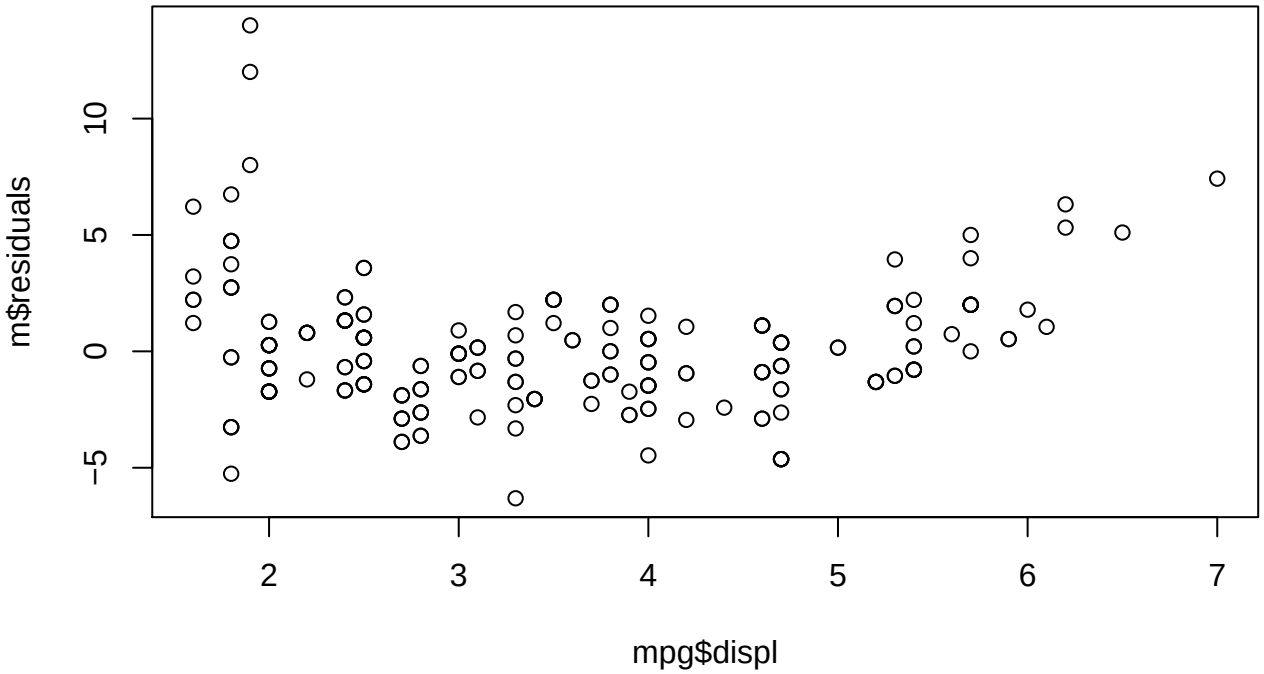
to get a sequence of summary plots describing our model-fit; this allows us to check if our modeling assumptions were satisfied.

Prediction What if we want to impute values on unseen observations? We can use the `predict()` function!

```
df <- data.frame(displ = seq(min(mpg$displ), max(mpg$displ), length.out = 20))
head(df)

##      displ
## 1 1.600000
## 2 1.884211
## 3 2.168421
## 4 2.452632
## 5 2.736842
## 6 3.021053

df$predicted_cty <- predict(object = m, newdata = df)
head(df)

##      displ predicted_cty
## 1 1.600000      21.78270
## 2 1.884211      21.03509
## 3 2.168421      20.28748
## 4 2.452632      19.53986
## 5 2.736842      18.79225
## 6 3.021053      18.04464

# plot(df, type = 'l')
```

Nothing interesting happens in the plot, because we've simply fit a linear model with a single feature, so the plot is just a downward sloping line (with slope equal to the coefficient obtained from the linear model. I.e., if we were to plot out our *residuals* as a function of our feature, we'd see a non-linear trend, which indicates that our model specification wasn't completely accurate.

```
plot(x = mpg$displ, y = m$residuals)
```



Notice there is a quadratic relationship "leftover", i.e. that was not explained by our model: the residuals tend to be higher when displacement is low (and high) and smaller (or even negative) for moderately sized values of displacement.

Exercise The function `plot` takes in an x and y argument, and with these you can create a scatter plot. Create a scatter plot of the predicted vs. used zone. Since many of the points may overlap ontop of each other, use `jitter` to add some random noise to your inputs

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

This is just like using `geom_jitter()` with `ggplot2`.

Adding Features to a Linear Model But we observed a slightly non-linear relationship when we plotted our raw data. What if we add a quadratic term? to account for the trend we saw in the original graph, and the takeaways we took from our residuals plot above.

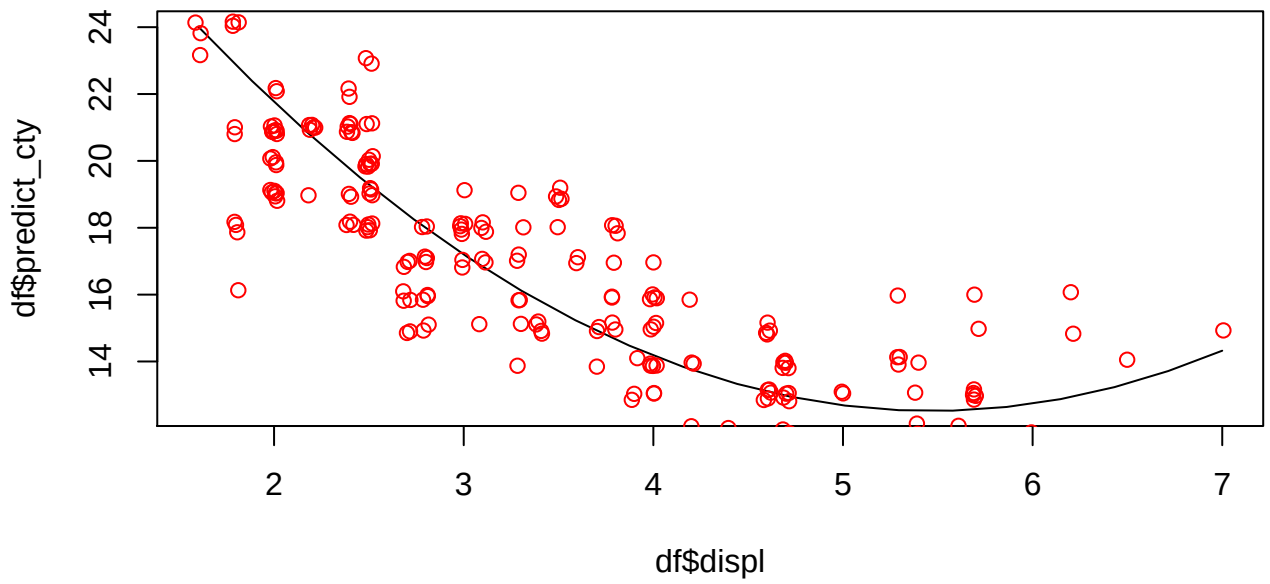
```
m2 <- lm(cty ~ displ + I(displ^2), data = mpg)
```

The new model has improved explanatory power, as can be seen by comparing the diagnostic plots via `plot(m); plot(m2)`.

```
df$predict_cty <- predict(m2, df)
```

Notice that we didn't have to create a squared explanatory variable, even though one is included in our model; R is smart enough to figure out how to create one for us given an appropriate input (i.e. a data.frame that has all the same features as used in the original model). Let's plot this trend we've fit and then also overlay the original data to eyeball how we'll we've fit the data:

```
plot(df$displ, df$predict_cty, type = "l")
points(x = jitter(mpg$displ), y = jitter(mpg$cty), col = "red")
```



Additional ways to add Interactions and Crosses You can use : in a formula to specify an *interaction* and you can use * to specify a cross. What does this mean?

- If you say `y ~ var1:var2`, you're saying that the marginal effect of the response depends on the interaction between the two variables.
- If you use `y ~ var1*var2`, then this is shorthand for writing `y ~ var1 + var2 + var1:var2`, where we are allowing for differing slope coefficients within each of `var1`, `var2`, and their interaction separately.

To learn more about this type of thing you would simply call `?lm` in the console.

Factor variables in modeling Factor variables get a bad reputation from those who are new to R programming, but they are incredibly useful. When used as a feature in a model, a string or factor variable with k different categories automatically gets treated as $k-1$ indicator variables. E.g. we can regress city-mpg as a function of manufacturer.

```
m3 <- lm(cty ~ manufacturer, data = mpg)
```

Note that we've calculated coefficients relative to our baseline, which in this cause is the first manufacturer in lexicographic (alphabetical) order, i.e. all coefficients describe the marginal effect of switching from Audi to another manufacturer:

```
coef(m3)
```

```
##            (Intercept)  manufacturerchevrolet      manufacturerdodge
##              17.6111111             -2.6111111             -4.4759760
##       manufacturerford      manufacturerhonda    manufacturerhyundai
##             -3.6111111              6.8333333              1.0317460
##       manufacturerjeep manufacturerland rover    manufacturerlincoln
##             -4.1111111             -6.1111111             -6.2777778
##    manufacturermercury     manufacturernissan    manufacturerpontiac
##             -4.3611111              0.4658120             -0.6111111
##     manufacturersubaru     manufacturertoyota manufacturervolkswagen
##              1.6746032              0.9183007              3.3148148
```

Exercise: We observed that the intercept term for city miles-per-gallon seems to depend on whether the car is front, rear, or four-wheel drive. Can you construct a model which incorporates this information? How does the interpretation of the coefficients match the intuition described by the plot we made above?

Exercise: Perhaps the relationship between city mileage and displacement depends on the manufacturer. Can you encode this information into a model?

8.4.2 Binary Classification

What about a simple logistic regression? Let's consider a dataset from `https://archive.ics.uci.edu/ml/machine-learning-databases/00426/`.

```
download.file(
  url = file.path("https://archive.ics.uci.edu/ml",
                  "machine-learning-databases/00426",
                  "Autism-Adult-Data%20Plus%20Description%20File.zip"),
  destfile = "~/Downloads/autism.zip"
)
unzip("~/Downloads/autism.zip", files = "Autism-Adult-Data.arff", exdir = "~/Downloads")
```

To load this data, we need the `foreign` package, since the data is in a Weka attributational relational file format (i.e. a special file type).

```
require(foreign)
data <- read.arff("~/Downloads/Autism-Adult-Data.arff")
str(data)
```

```
## 'data.frame': 704 obs. of  21 variables:
##  $ A1_Score       : Factor w/ 2 levels "0","1": 2 2 2 2 2 2 1 2 2 2 ...
##  $ A2_Score       : Factor w/ 2 levels "0","1": 2 2 2 2 1 2 2 2 2 2 ...
##  $ A3_Score       : Factor w/ 2 levels "0","1": 2 1 1 1 1 2 1 2 1 2 ...
##  $ A4_Score       : Factor w/ 2 levels "0","1": 2 2 2 2 1 2 1 2 1 2 ...
##  $ A5_Score       : Factor w/ 2 levels "0","1": 1 1 2 1 1 2 1 1 2 1 ...
##  $ A6_Score       : Factor w/ 2 levels "0","1": 1 1 1 1 1 1 1 1 1 2 ...
##  $ A7_Score       : Factor w/ 2 levels "0","1": 2 1 2 2 1 2 1 1 1 2 ...
##  $ A8_Score       : Factor w/ 2 levels "0","1": 2 2 2 2 2 2 2 1 2 2 ...
##  $ A9_Score       : Factor w/ 2 levels "0","1": 1 1 2 1 1 2 1 2 2 2 ...
```

```
##  $ A10_Score      : Factor w/ 2 levels "0","1": 1 2 2 2 1 2 1 1 2 1 ...
##  $ age            : num  26 24 27 35 40 36 17 64 29 17 ...
##  $ gender         : Factor w/ 2 levels "f","m": 1 2 2 1 1 2 1 2 2 2 ...
##  $ ethnicity      : Factor w/ 11 levels "Asian","Black",..: 11 4 4 11 NA 7 2 11 11 1 ...
##  $ jundice        : Factor w/ 2 levels "no","yes": 1 1 2 1 1 2 1 1 1 2 ...
##  $ austim         : Factor w/ 2 levels "no","yes": 1 2 2 2 1 1 1 1 1 2 ...
##  $ contry_of_res  : Factor w/ 67 levels "Afghanistan",..: 65 14 57 65 23 65 65 44 65 10 ...
##  $ used_app_before: Factor w/ 2 levels "no","yes": 1 1 1 1 1 1 1 1 1 1 ...
##  $ result         : num  6 5 8 6 2 9 2 5 6 8 ...
##  $ age_desc       : Factor w/ 1 level "18 and more": 1 1 1 1 1 1 1 1 1 1 ...
##  $ relation       : Factor w/ 5 levels "Health care professional",..: 5 5 3 5 NA 5 5 3 5 1
##  $ Class/ASD      : Factor w/ 2 levels "NO","YES": 1 1 2 1 1 2 1 1 1 2 ...
```

Let's pretend we're interested in classifying the outcome of `A10_Score`, and we can use some of the features in our dataset to explain the variation in the response. Since `A10_Score` is a *binary* outcome, we might consider a *logistic* regression. A logistic regression is a special type of a generalized linear model, so we use the command `glm()`, where in particular we specify that we want a binomial family i.e. a logistic link function.

```
data <- na.omit(data)
m <- glm(A10_Score ~ age + gender + ethnicity + jundice + austim + contry_of_res,
         family = "binomial", data)
```

We can still do things that we did before with our data, like call `plot()` on the model-object to view diagnostics, call `summary()` to view coefficients and statistical conclusions drawn, and use the `predict()` function to make predictions on new data.

Calibration How well do our predictions explain our data? Our logistic *regression* spits out a probability in the interval $[0, 1]$, but any given outcome (i.e. label) is either a zero *or* a one. So how can we evaluate the performance of our classfier? One way is to check its calibration.

What is calibration? If a classifier spits out a prediction of 0.4, then we would hope that about 40% of such predictions should indeed have a "positive" label. If it turns out that only 10% or say 80% of predictions have a positive label, we would not say our model is calibrated. By using `type = ''response''`, we ask our `predict()` method to give us probabilities as predictions; don't forget this!

```
data$prediction <- predict(m, data, type = "response")
```

Now let's aggregate our data.

```
aggregate(x = as.integer(as.character(data$A10_Score)),
          by = list(prediction = round(data$prediction, 1)),
          FUN = mean, na.rm = TRUE)
```

```
##    prediction         x
## 1         0.0 0.0000000
## 2         0.2 1.0000000
## 3         0.3 0.2500000
## 4         0.4 0.4479167
## 5         0.5 0.4692308
```

```
## 6         0.6 0.6300000
## 7         0.7 0.6793893
## 8         0.8 0.7761194
## 9         0.9 0.9047619
## 10        1.0 1.0000000
```

It looks like we did a "bad" job at predicting the lower-tail. But how much data was there in the tails? One technique we can use is bucketizing our data into quantiles!

8.4.3 Quantiles and Discretizing Data

We use `cut()` and `quantile()`.

```
breaks <- quantile(data$prediction, probs = seq(0, 1, by = 0.1),
                   na.rm = TRUE)
data$bucket <- cut(data$prediction, breaks, include.lowest = TRUE)
```

Now, we've assigned each observation to one of ten quantiles. This means there's approximately an even number of observations in each bucket.

Exercise: Can you verify that we have bucketed our data such that there is an equal amount of support in each bucket? Hint: see `prop.table()`. If you'd like, after using this function you can try calling `plot()` on the output to get a visual.

Re-Aggregating Data Let's now re-aggregate our data.

```
outcomes <-
    aggregate(x = as.integer(as.character(data$A10_Score)),
              by = list(prediction = data$bucket),
              FUN = mean, na.rm = TRUE)
outcomes
```

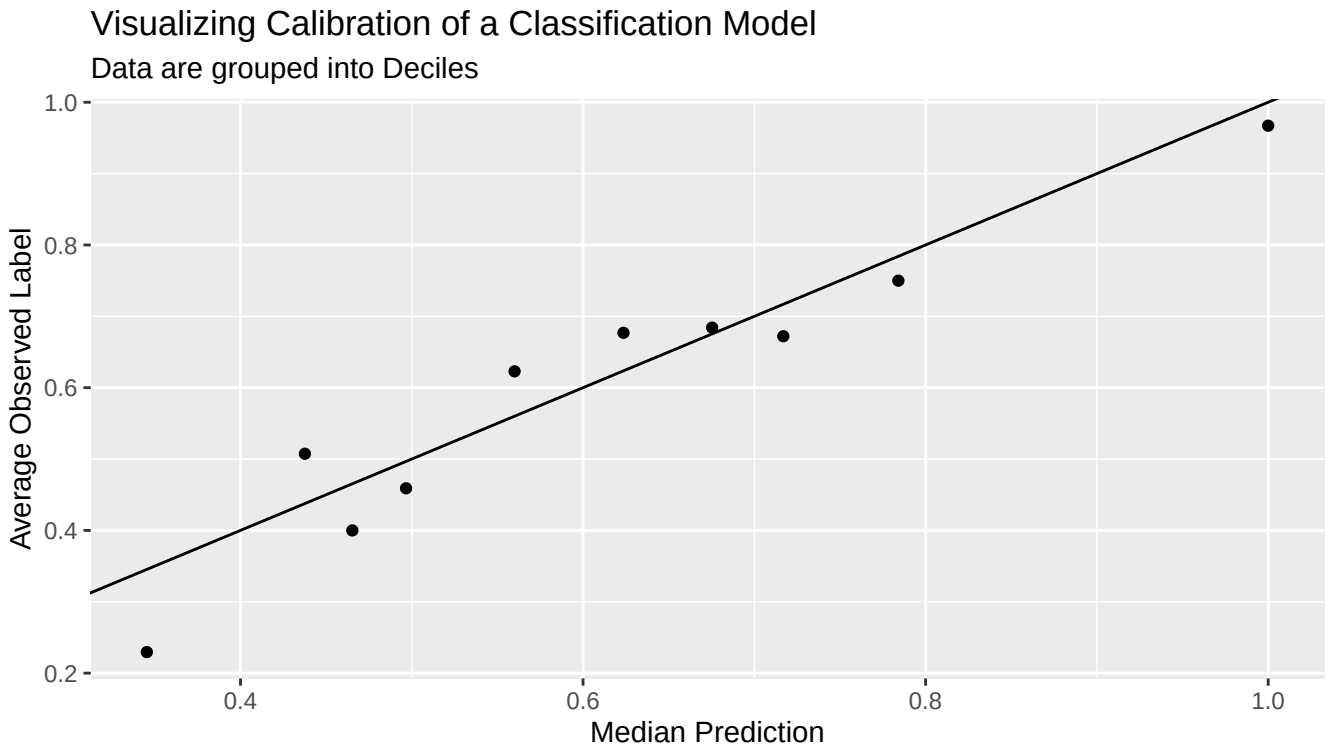
```
##          prediction         x
## 1  [1.38e-08,0.414] 0.2295082
## 2     (0.414,0.449] 0.5074627
## 3     (0.449,0.476] 0.4000000
## 4     (0.476,0.535] 0.4590164
## 5     (0.535,0.592] 0.6229508
## 6     (0.592,0.657] 0.6769231
## 7     (0.657,0.699] 0.6842105
## 8     (0.699,0.749] 0.6721311
## 9     (0.749,0.829] 0.7500000
## 10        (0.829,1] 0.9672131
```

Visualizing Median Predictions against Mean Outcomes It's a little bit hard to see what's going on, let's visualize our data by plotting the mean outcome against the median prediction within each bucket.

```
median_predictions <-
    aggregate(x = data$prediction,
              by = list(prediction = data$bucket),
              FUN = median, na.rm = TRUE)
summ <- merge(outcomes, median_predictions,
              by = "prediction",
              suffixes = c("_outcome", "_median_prediction"))
summ
```

```
##          prediction x_outcome x_median_prediction
## 1     (0.414,0.449] 0.5074627           0.4375913
## 2     (0.449,0.476] 0.4000000           0.4652986
## 3     (0.476,0.535] 0.4590164           0.4966645
## 4     (0.535,0.592] 0.6229508           0.5600122
## 5     (0.592,0.657] 0.6769231           0.6236088
## 6     (0.657,0.699] 0.6842105           0.6753622
## 7     (0.699,0.749] 0.6721311           0.7169160
## 8     (0.749,0.829] 0.7500000           0.7840766
## 9         (0.829,1] 0.9672131           1.0000000
## 10 [1.38e-08,0.414] 0.2295082           0.3453151
```

```
ggplot(summ, aes(x = x_median_prediction, y = x_outcome)) +
    geom_point() +
    geom_abline(slope = 1, intercept = 0) +
    labs(title = "Visualizing Calibration of a Classification Model",
         subtitle = "Data are grouped into Deciles",
         x = "Median Prediction",
         y = "Average Observed Label")
```



Here we have a model that is not too unreasonably calibrated.

9 Practice Exercises

9.1 Built-in Constants

R has a few built-in constants that are really handy. See: `?letters` for a listing. Also realize that you can create `data.frames` with objects in memory using syntax that looks as follows:

```
df <- data.frame(variable1 = 1:10,
                 variable2 = rnorm(10))
```

Exercise Create a `data.frame` where the first column is labeled "id" and contains the first 12 letters of the lower-case alphabet, the second column should be labeled "month" and contain the 12 calendar months in temporal order, and a third column should contain some random noise drawn from a distribution of your choosing (e.g. see `?rpois`, `?rexp`).

Exercise The `summary()` command is very powerful, especially when applied to `data.frames`. Try calling `summary()` on various objects like the `data.frame` you just created in the previous exercise.

9.2 First Principles Statistics

Exercise Make a script which constructs three random (standard) normal vectors of length 1,000. Call these vectors `x`, `y`, and `z`. Now, create a `data.frame` with columns a, b, and c which are respectively defined as x, $x + y$, and $x + y + z$.

Exercise Additionally, print out the (scalar-valued) variance of each column from the previous exercise one at a time to console, using the built-in function `var()`. Can you explain the (statistical) result?

Exercise Using the same `data.frame` created in the first exercise of this sub-section, try plotting in the following way.

```
# Assuming we have (only!) columns "a", "b", "c" in our `df`.
m <- reshape2::melt(df)  # <-- Automatically creates long data.frame.
m <- m %>% group_by(variable) %>% mutate(idx = seq_along(value))
ggplot(m, aes(x = value, fill = variable)) +
    geom_histogram(bins = 50) +
    facet_wrap(~variable, nrow = 3)
```

How does this match your statistical intuition?

9.3 Creating New Variables in `data.frames`

Exercise Try creating a `data.frame` with three numeric variables: `x`, `y`, and `z`; they can all be random noise if you wish, e.g.

```
df <- data.frame(x = rnorm(6), y = rpois(6, lambda = 1), z = rexp(6))
```

Now, try creating a new variable, call it *alpha*, which takes on a transformation of other data in data.frame `df`. E.g. $\alpha = x + y^2 + sin(z)$. Remember that if you want to refer to (either a new or already existing) column that you can use either "$" or '[[' syntax, or you can also use `mutate()`.

9.4 Boolean Arithmetic

The daily utility of boolean arithmetic is indispensable. I.e. want to know the proportion of GPA's above 3.5, segmented by under-graduate major? Simply create a boolean vector comparing GPA to 3.5, and use it as an input argument to `mean()`... all of this well be used within an argument to `summarise()`. Start with:

```
majors <- c("bio", "chem", "math", "english", "statistics")
df <- data.frame(major = sample(majors, size = 100, repl = TRUE),
```

9.5 Operations on Filtered `data.frames`

Suppose we have a `data.frame` as follows.

```
d <- data.frame(i = letters,
                x = runif(26),
                y = rnorm(26),
                z = rnorm(26))
d %>% slice(c(1:3, nrow(d)))

##   i         x           y          z
## 1 a 0.8781658 -0.06164547  0.6915121
## 2 b 0.5339206 -0.30203217 -0.2170951
## 3 c 0.8504338 -1.74261209 -1.5348000
## 4 z 0.5533647 -1.09411243 -0.1333596

# Pull out the first, third, and last row by ID.
d %>% filter(i %in% c("a", "c", "z"))

##   i         x           y          z
## 1 a 0.8781658 -0.06164547  0.6915121
## 2 c 0.8504338 -1.74261209 -1.5348000
## 3 z 0.5533647 -1.09411243 -0.1333596
```

Exercise We could also do things like, perform conditional correlation between two columns; i.e. for observations satisfying the boolean condition (in this case, if the ID is one of "a", "c", or "z") we take a correlation. This will involve two steps: one call to `filter()` and another to `summarise()` where within the latter we make a call to `cor()`.

9.6 Plotting

Colours and Density Plots Using the `mpg` dataset, can you add a colour layer describing the `drv` (drivetrain-type) to a density plot of `cty` miles-per-gallon? See `?geom_density()`.

In addition to using the *colour* layer, in a separate plot try replacing `colour` with `fill` and see what happens. Hint: the optional argument `alpha` may be helpful; the syntax to use it is something like `geom_density(alpha = 1/2)`.

Box and Whisker Plots Using our `foul`-balls dataset, we can also wonder if exit velocity depends on the type of hit. Perhaps a box-and-whisker plot is appropriate here. Hint: if you want to use `ggplot2`, consider the `geom_boxplot()` geometry; it's also worth seeing what happens if you feed in `type_of_hit` as the explanatory (independent) variable in a basic plot routine (i.e. as first argument)

9.7 Working with Strings and Dates

Try creating a *date* vector: R has handy built-in's for this!

```
dates <- seq.Date(from = as.Date("2021-08-01"),
                  to   = as.Date("2021-12-31"),
                  by   = "1 day")
```

We used the `as.Date()` function to create a date object from a string. The `seq.Date()` function is a specialized method of the more general `seq()` that is designed to create date-sequences.

Exercise First, install/load either `data.table` package or `lubridate` to get access to the `month()` function. Call the function `month()` on the `dates` object above. What do you get? What happens if you call `table()` on the *resulting* output? Can you explain the result to your neighbor?

Exercise Given a year (specified as an integer), can you write a function to determine if it is a leap-year? Hint: can you find a way to use the `paste()` function to combine an integer `year` with string describing a month and day to create a date object? E.g.

```
year <- "2012"
beg_date <- as.Date(paste(year, "01-01", sep = "-"))
```

Follow-up hints: consider using the functions `seq.Date()` and `length()`.

Exercise (1) Try creating a vector of strings, where each string contains a first and a last name (separated by a space). E.g.

```
names <- c("Andreas Santucci", "Ada Lovelace")
df <- data.frame(name = names)
```

(2) Use the function `separate()` with argument `sep` chosen judiciously to *split* the first and last names apart; note that you'll have to use the `into` argument, I recommend using `into = c("first", "last")`. (3) Inspect the output, what is the class of the object and what is its structure?

Exercise Although there is a handy `year()` and `month()` function built-in to `data.table` and `lubridate` packages, there isn't an equivalent `day()` function. Use `strsplit()` and `do.call(rbind, .)` on the `dates` object we created above in order to create a matrix where the year, month, and day attributes are separated from each other into their own columns. If you want to, you can also place the `dates` object into a `data.frame` as a variable, and then use `tidyr::separate()` with argument `split` chosen appropriately. This latter approach may be more applicable to real world scenaios where you have dates inside of a `data.frame`, as opposed to dealing with a standalone dates object.

9.8 Case Study

Let's try a more practical case study example. There's a datset called `starwars` that's already loaded into your workspace when you load the `dplyr` package.

```
head(starwars)
```

Exercise: Imputing a String Notice that there are a few missing values for hair color. Find out, using verbs `dplyr::group_by()`, `dplyr::select()`, and `dplyr::count()`, and `dplyr::arrange()`. Now that you've identified what the most common value is for this variable, try "imputing" `NA` entries with this value. Hint: consider `dplyr::mutate()` and `ifelse()` alongside `is.na()`.

Exercise: Spotting Outliers There are outliers in the `birth_year` variable. Can you come up with a technique to spot them, e.g. creating a graphic that visually identifies them? Once you identify them, can you create a new variable that's Boolean valued that describes whether `birth_year` is an outlier? Or perhaps another variable that is the `log()` transform of the `birth_year`...

Exercise: Splitting Names Suppose you were tasked with splitting names into "first" and "last" components. The way to do this is to use `tidyr::separate()`, in general, however, there are some special considerations to pay attention to with this dataset.

- Some `name`s have more than 2 components. What should we do with these? Let's arbitrarily decide to lump all but the first component together and call that the "last" name. To accomplish this, consider using the arguments `extra = 'merge'` and `fill = 'right'`.
- Some `name`s don't have spaces to split on, they perhaps have dashes. Can we handle this in one call to `separate()`? The answer is yes, we just need to set `sep = ' |-'`, which means: "split on either a space *or* a dash".

Note that you'll have to be careful when porting the above code into your script/console as the quotation-marks aren't properly displayed in the pdf.

Exercise: Spotting multi-tonal skin colors As usual, the operations we seek to perform can be easily carried out in base-R but there are wrappers written by Hadley Wickham's supporting team.

```
install.packages("stringr")
```

1. Use `select()` and `unique()` to get a glimpse of what the distinct types of skin color we are observing in our raw data.
2. Use `mutate()` to create a new variable that is Boolean valued and describes whether there is more than one skin-tone. Hint: use `str_detect()`.

10 Base-R Exercises

Exercises marked with `(*)` may be slightly more difficult.

1. You are given an input of numeric values x, and are told that instead of corrupted observations being reported as missing, they are instead reported as the numeric value `999`. E.g. a sample input could be `x = c(1, 4, 2, 999, 7, 11, 999, 999, 12)`. Use `ifelse` to create a new object which replaces any instance of `999` with `NA`, which is R's internal representation for "not available" or missing data. Then, use `sum()`, `!`, and `is.na()` to count the number of non-missing entries provided in the input.

2. You are given an input of consecutive integers between 1 and N inclusive, except exactly *one element* is missing. Your goal is to write a function which returns the missing element. E.g. `FindMissing(c(1,2,4,5))` should return `3`. *Hints:* try solving several ways: (i) by using operators `which()`, `%in%`, and `!`, (ii) by using `setdiff()`, `range()`, and `seq()`, and (iii) using triangular numbers.

3. Make a vector of consecutive integers from 1 to 100 inclusive. Write a `for`-loop which runs through the whole vector, multiplying elements less than 5 or larger than 90 by 10 and all other elements by 1/10. Then, rewrite your above solution using what you learn from examining `?ifelse`.

4. You are given an input sequence x and an output sequence $f(x)$. Write a function which returns the value of x corresponding to the maximum of $f(x)$. I.e. implement an `argmax` function that is of zeroth-order. *Hint:* see `?which.max`, and use the result in conjuction with an `'['` operator.

5. You are given a single string with a listing of words. Return a table describing the frequency of word-occurrences. *Hint:* you'll find the functions `strsplit()`, `unlist()`, and `table()`, helpful.

6. You are given a folder (working directory) with consumption data stored in `.csv` files which you need to load into memory. There are 12 files, one for each calendar month of a particular year. Your goal is to use `list.files()` and a `for()` loop to iteratively `rbind()` the `data.frame`'s together. You'll need to take care to add a month describing the calendar month during the process, since the raw data won't include this. To get started, run the following code to create 12 sample data files in a new directory.

```
dir.create("calendar_months_data", showWarnings = FALSE)
for (i in 1:12) {
    d <- data.frame(id = letters, y  = rnorm(26))
    write.csv(d, file = paste0("calendar_months_data/month_", i, ".csv"),
              row.names = FALSE)
}
```

   Once you solve the problem iteratively using `rbind()`, try to understand what the following expression does. `l <- lapply(fnames, read.csv); do.call(rbind, l)`. See `?sapply()` which replaces a `for`-loop, and `do.call()`.

7. **(*)** Rewrite our data generation process used in the above question as follows: first write a function `Create(i)` which takes an integer i and effectively replaces the body of our `for`-loop above. Then, realize that we can simply write `for (i in 1:12) Create(i)` to achieve the same result! Now, look at `?sapply` and figure out how to use it to achieve again the identical result.

8. **(*)** You are again given a single string with a listing of words. Return a frequency count of all n-grams occurring, for arbitrary n. We define an n-gram as a contiguous sequence of n words. *Hint:* start with $n = 2$.

9. (*) Given an input stream 1,000 digits long, find the greatest product of five consecutive digits. (The input is given as a single string. Your output should be a single scalar.)

10. (*) Given an integer K, return a description of the number of days in each of the twelve calendar months; some years are leap years, in which case February will have 29 days. *Hint:* see `?seq.Date`. An ideal solution will create a `data.frame` with three columns: one with the year, another with the month-name, and a third with the number of days in the month (and year).

11. (*) You are given an urn with an unlimited number of balls in it, where you are told that 4/10 of them are red, 1/2 are green, and the remaining 1/10 are blue. You draw a sequence of balls until you draw one which is blue. How many draws do you expect to take before realizing a success? *Hint:* you can solve this problem using built-in statistical functions if you understand the probability distribution we are drawing from, and alternatively you can also solve it via simulation. You may find `?replicate` to be helpful if you choose the latter approach.

12. (*) What's more likely, the sequence of coin tosses `HTH` or `HHT`? Support your answer with a simulation in R. This answer can also be answered using theory from Markov chains.

13. (*) What's the difference between `sapply()` and `lapply()` exactly? Can you make one behave like the other? *Hint:* see `head(sapply)` and also `?sapply`.

14. (*) Why would you use `mapply()`? As a trivial example, consider re-implementing element-wise vector addition in terms of a single (and simple) call to `mapply` where we supply `FUN = "+"`. *Hint:* see `help("+")` to learn the names of the arguments the addition operator expects.

References

[1] Wickham, Hadley Introduction to R 2017, O'Reilly

[2] Adler, Joseph. R in a Nutshell 2012, O'Reilly

[3] Brauer, Torfs. *A (very) short introduction to R.* 2014, CRAN.

[4] Burns, Patrick. The R Inferno 2011

[5] Free resources for learning R 2016, Stack Exchange

[6] Abelson and Susman Structure and Interpretation of Computer Programs MIT, 2005

[7] An Introduction to R CRAN, July 2018

[8] Dirk Eddelbuettel CRAN Task View: High-Performance & Parallel Computing with R July 30, 2018

[9] Hilary Parker Writing an R Package from scratch April 29, 2014

[10] Jonathan Taylor Data Science 101 Spring 2018