Biological Data Science with R

Stephen D. Turner, Ph.D.

2019-01-01

Table of contents

Preface

This book was written as a companion to a series of courses I taught at the University of Virginia introducing the essentials of biological data science with R:

1. UVA Biomedical Sciences Graduate Program BIMS8382: **bims8382.github.io**.
2. UVA Health Sciences Library Biological Data Science Workshops: **stephen-turner.github.io/workshops**.
3. UVA Translational Health Research Institute of Virginia (THRIV) Scholars program Biological Data Science course: **thriv.github.io**.

While this book was written with the accompanying live instruction in mind, this book can be used as a self-contained self study guide for quickly learning the essentials need to get started with R. The BDSR book and accompanying course introduces methods, tools, and software for reproducibly managing, manipulating, analyzing, and visualizing large-scale biological data using the R statistical computing environment. This book also covers essential statistical analysis, and advanced topics including survival analysis, predictive modeling, forecasting, and text mining.

This is not a *"Tool X"* or *"Software Y"* book. I want you to take away from this book and accompanying course the ability to use an extremely powerful scientific computing environment (R) to do many of the things that you'll do *across study designs and disciplines* – managing, manipulating, visualizing, and analyzing large, sometimes high-dimensional data. Regardless of your specific discipline you'll need the same computational know-how and data literacy to do the same kinds of basic tasks in each. This book might show you how to use specific tools here and there (e.g., DESeq2 for RNA-seq analysis (Love, Huber, and Anders 2014), ggtree for drawing phylogenetic trees (Yu et al. 2017), etc.), but these are not important – you probably won't be using the same specific software or methods 10 years from now, but you'll still use the same underlying data and computational foundation. That is the point of this series – to arm you with a basic foundation, and more importantly, to enable you to figure out how to use *this tool* or *that tool* on your own, when you need to.

This is not a statistics book. There is a short chapter on essential statistics using R in Chapter 8 but this short chapter offers neither a comprehensive background on underlying theory nor in-depth coverage of implementation strategies using R. Some general knowledge of statistics and study design is helpful, but isn't required for going through this book or taking the accompanying course.

There are no prerequisites to this book or the accompanying course. However, each chapter involves lots of hands-on practice coding, and you'll need to download and install required softwar and download required data. See the setup instructions in Appendix A.

Acknowledgements

This book is partially adapted from material developed from the courses I taught above, some co-taught with VP (Pete) Nagraj, from 2015-2019. The material for this course was adapted from and/or inspired by Jenny Bryan's STAT545 course at UBC (Bryan 2019), Software Carpentry (Wilson 2014) and Data Carpentry (Teal et al. 2015) courses, David Robinson's *Variance Explained* blog (Robinson 2015), the ggtree vignettes (Yu 2022) *Tidy Text Mining with R* (Silge and Robinson 2017), and likely many others.

Part I

Core Curriculum

1 Basics

This chapter introduces the R environment and some of the most basic functionality aspects of R that are used through the remainder of the book. This section assumes little to no experience with statistical computing with R. This chapter introduces the very basic functionality in R, including variables, functions, and importing/inspecting data frames (tibbles).

1.1 RStudio

Let's start by learning about RStudio. **R** is the underlying statistical computing environment. **RStudio** is a graphical integrated development environment (IDE) that makes using R much easier.

- **Options:** First, let's change a few options. We'll only have to do this once. Under *Tools… Global Options…*:
  - Under *General*: Uncheck "Restore most recently opened project at startup"
  - Under *General*: Uncheck "Restore .RData into workspace at startup"
  - Under *General*: Set "Save workspace to .RData on exit:" to Never.
  - Under *General*: Set "Save workspace to .RData on exit:" to Never.
  - Under *R Markdown*: Uncheck "Show output inline for all R Markdown documents"
- Projects: first, start a new project in a new folder somewhere easy to remember. When we start reading in data it'll be important that the *code and the data are in the same place.* Creating a project creates an Rproj file that opens R running *in that folder.* This way, when you want to read in dataset *whatever.txt*, you just tell it the filename rather than a full path. This is critical for reproducibility, and we'll talk about that more later.
- Code that you type into the console is code that R executes. From here forward we will use the editor window to write a script that we can save to a file and run it again whenever we want to. We usually give it a `.R` extension, but it's just a plain text file. If you want to send commands from your editor to the console, use `CMD+Enter` (`Ctrl+Enter` on Windows).
- Anything after a `#` sign is a comment. Use them liberally to *comment your code.*

1.2 Basic operations

R can be used as a glorified calculator. Try typing this in directly into the console. Make sure you're typing into into the editor, not the console, and save your script. Use the run button, or press `CMD`+`Enter` (`Ctrl`+`Enter` on Windows).

```
2+2
```

```
[1] 4
```

```
5*4
```

```
[1] 20
```

```
2^3
```

```
[1] 8
```

R Knows order of operations and scientific notation.

```
2+3*4/(5+3)*15/2^2+3*4^2
```

```
[1] 55.6
```

```
5e4
```

```
[1] 50000
```

However, to do useful and interesting things, we need to assign *values* to *objects*. To create objects, we need to give it a name followed by the assignment operator `<-` and the value we want to give it:

```
weight_kg <- 55
```

`<-` is the assignment operator. Assigns values on the right to objects on the left, it is like an arrow that points from the value to the object. Mostly similar to `=` but not always. Learn to

use `<-` as it is good programming practice. Using `=` in place of `<-` can lead to issues down the line. The keyboard shortcut for inserting the `<-` operator is `Alt-dash`.

Objects can be given any name such as `x`, `current_temperature`, or `subject_id`. You want your object names to be explicit and not too long. They cannot start with a number (`2x` is not valid but `x2` is). R is case sensitive (e.g., `weight_kg` is different from `Weight_kg`). There are some names that cannot be used because they represent the names of fundamental functions in R (e.g., `if`, `else`, `for`, see here for a complete list). In general, even if it's allowed, it's best to not use other function names, which we'll get into shortly (e.g., `c`, `T`, `mean`, `data`, `df`, `weights`). In doubt check the help to see if the name is already in use. It's also best to avoid dots (`.`) within a variable name as in `my.dataset`. It is also recommended to use nouns for variable names, and verbs for function names.

When assigning a value to an object, R does not print anything. You can force to print the value by typing the name:

```
weight_kg
```

```
[1] 55
```

Now that R has `weight_kg` in memory, we can do arithmetic with it. For instance, we may want to convert this weight in pounds (weight in pounds is 2.2 times the weight in kg).

```
2.2 * weight_kg
```

```
[1] 121
```

We can also change a variable's value by assigning it a new one:

```
weight_kg <- 57.5
2.2 * weight_kg
```

```
[1] 127
```

This means that assigning a value to one variable does not change the values of other variables. For example, let's store the animal's weight in pounds in a variable.

```
weight_lb <- 2.2 * weight_kg
```

and then change `weight_kg` to 100.

```
weight_kg <- 100
```

What do you think is the current content of the object `weight_lb`? 126.5 or 220?

You can see what objects (variables) are stored by viewing the Environment tab in Rstudio. You can also use the `ls()` function. You can remove objects (variables) with the `rm()` function. You can do this one at a time or remove several objects at once. You can also use the little broom button in your environment pane to remove everything from your environment.

```
ls()
rm(weight_lb, weight_kg)
ls()
weight_lb # oops! you should get an error because weight_lb no longer exists!
```

Exercise 1

What are the values after each statement in the following?

```
mass <- 50              # mass?
age  <- 30              # age?
mass <- mass * 2        # mass?
age  <- age - 10        # age?
mass_index <- mass/age  # massIndex?
```

1.3 Functions

R has built-in functions.

```
# Notice that this is a comment.
# Anything behind a # is "commented out" and is not run.
sqrt(144)
```

```
[1] 12
```

```
log(1000)
```

```
[1] 6.91
```

Get help by typing a question mark in front of the function's name, or `help(functionname)`:

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

Note syntax highlighting when typing this into the editor. Also note how we pass *arguments* to functions. The `base=` part inside the parentheses is called an argument, and most functions use arguments. Arguments modify the behavior of the function. Functions some input (e.g., some data, an object) and other options to change what the function will return, or how to treat the data provided. Finally, see how you can *next* one function inside of another (here taking the square root of the log-base-10 of 1000).

```
log(1000)
```

```
[1] 6.91
```

```
log(1000, base=10)
```

```
[1] 3
```

```
log(1000, 10)
```

```
[1] 3
```

```
sqrt(log(1000, base=10))
```

```
[1] 1.73
```

> Exercise 2
>
> See `?abs` and calculate the square root of the log-base-10 of the absolute value of `-4*(2550-50)`. Answer should be 2.

1.4 Tibbles (data frames)

There are *lots* of different basic data structures in R. If you take any kind of longer introduction to R you'll probably learn about arrays, lists, matrices, etc. We are going to skip straight to the data structure you'll probably use most – the **tibble** (also known as the data frame). We use tibbles to store heterogeneous tabular data in R: tabular, meaning that individuals or observations are typically represented in rows, while variables or features are represented as columns; heterogeneous, meaning that columns/features/variables can be different classes (on variable, e.g. age, can be numeric, while another, e.g., cause of death, can be text).

We'll learn more about tibbles in Chapter 2.

2 Tibbles

There are *lots* of different basic data structures in R. If you take any kind of longer introduction to R you'll probably learn about arrays, lists, matrices, etc. Let's skip straight to the data structure you'll probably use most – the **data frame**. We use data frames to store heterogeneous tabular data in R: tabular, meaning that individuals or observations are typically represented in rows, while variables or features are represented as columns; heterogeneous, meaning that columns/features/variables can be different classes (on variable, e.g. age, can be numeric, while another, e.g., cause of death, can be text).

This chapter assumes a basic familiarity with R (see Chapter 1).

Recommended reading: Review the *Introduction* (10.1) and *Tibbles vs. data.frame* (10.3) sections of the ***R for Data Science* book**. We will initially be using the `read_*` functions from the **readr** package. These functions load data into a *tibble* instead of R's traditional data.frame. Tibbles are data frames, but they tweak some older behaviors to make life a little easier. These sections explain the few key small differences between traditional data.frames and tibbles.

2.1 Our data

The data we're going to look at is cleaned up version of a gene expression dataset from Brauer et al. Coordination of Growth Rate, Cell Cycle, Stress Response, and Metabolic Activity in Yeast (2008) *Mol Biol Cell* 19:352-367. This data is from a gene expression microarray, and in this paper the authors are examining the relationship between growth rate and gene expression in yeast cultures limited by one of six different nutrients (glucose, leucine, ammonium, sulfate, phosphate, uracil). If you give yeast a rich media loaded with nutrients except restrict the supply of a *single* nutrient, you can control the growth rate to any rate you choose. By starving yeast of specific nutrients you can find genes that:

1. **Raise or lower their expression in response to growth rate**. Growth-rate dependent expression patterns can tell us a lot about cell cycle control, and how the cell responds to stress. The authors found that expression of >25% of all yeast genes is linearly correlated with growth rate, independent of the limiting nutrient. They also found that the subset of negatively growth-correlated genes is enriched for peroxisomal functions, and positively correlated genes mainly encode ribosomal functions.

2. **Respond differently when different nutrients are being limited**. If you see particular genes that respond very differently when a nutrient is sharply restricted, these genes might be involved in the transport or metabolism of that specific nutrient.

You can download the cleaned up version of the data here. The file is called **brauer2007_tidy.csv**. Later on we'll actually start with the original raw data (minimally processed) and manipulate it so that we can make it more amenable for analysis.

2.2 Reading in data

2.2.1 dplyr and readr

There are some built-in functions for reading in data in text files. These functions are *read-dot-something* – for example, `read.csv()` reads in comma-delimited text data; `read.delim()` reads in tab-delimited text, etc. We're going to read in data a little bit differently here using the readr package. When you load the readr package, you'll have access to very similar looking functions, named *read-underscore-something* – e.g., `read_csv()`. You have to have the readr package installed to access these functions. Compared to the base functions, they're *much* faster, they're good at guessing the types of data in the columns, they don't do some of the other silly things that the base functions do. We're going to use another package later on called dplyr, and if you have the dplyr package loaded as well, and you read in the data with readr, the data will display nicely.

First let's load those packages.

```
library(readr)
library(dplyr)
```

If you see a warning that looks like this: `Error in library(packageName) : there is no package called 'packageName'`, then you don't have the package installed correctly. See the setup chapter (Appendix A).

2.2.2 `read_csv()`

Now, let's actually load the data. You can get help for the import function with `?read_csv`. When we load data we assign it to a variable just like any other, and we can choose a name for that data. Since we're going to be referring to this data a lot, let's give it a short easy name to type. I'm going to call it `ydat`. Once we've loaded it we can type the name of the object itself (`ydat`) to see it printed to the screen.

```
ydat <- read_csv(file="data/brauer2007_tidy.csv")
ydat
```

```
# A tibble: 198,430 x 7
   symbol systematic_name nutrient  rate expression bp                    mf
   <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                 <chr>
 1 SFB2   YNL049C         Glucose   0.05      -0.24 ER to Golgi transport  mole~
 2 <NA>   YNL095C         Glucose   0.05       0.28 biological process un~ mole~
 3 QRI7   YDL104C         Glucose   0.05      -0.02 proteolysis and pepti~ meta~
 4 CFT2   YLR115W         Glucose   0.05      -0.33 mRNA polyadenylylatio~ RNA ~
 5 SSO2   YMR183C         Glucose   0.05       0.05 vesicle fusion*        t-SN~
 6 PSP2   YML017W         Glucose   0.05      -0.69 biological process un~ mole~
 7 RIB2   YOL066C         Glucose   0.05      -0.55 riboflavin biosynthes~ pseu~
 8 VMA13  YPR036W         Glucose   0.05      -0.75 vacuolar acidification hydr~
 9 EDC3   YEL015W         Glucose   0.05      -0.24 deadenylylation-indep~ mole~
10 VPS5   YOR069W         Glucose   0.05      -0.16 protein retention in ~ prot~
# i 198,420 more rows
```

Take a look at that output. The nice thing about loading dplyr and reading in data with readr is that data frames are displayed in a much more friendly way. This dataset has nearly 200,000 rows and 7 columns. When you import data this way and try to display the object in the console, instead of trying to display all 200,000 rows, you'll only see about 10 by default. Also, if you have so many columns that the data would wrap off the edge of your screen, those columns will not be displayed, but you'll see at the bottom of the output which, if any, columns were hidden from view. If you want to see the whole dataset, there are two ways to do this. First, you can click on the name of the data.frame in the **Environment** panel in RStudio. Or you could use the `View()` function (*with a capital V*).

```
View(ydat)
```

2.3 Inspecting data.frame objects

2.3.1 Built-in functions

There are several built-in functions that are useful for working with data frames.

- Content:
  - `head()`: shows the first few rows
  - `tail()`: shows the last few rows

- Size:
  - `dim()`: returns a 2-element vector with the number of rows in the first element, and the number of columns as the second element (the dimensions of the object)
  - `nrow()`: returns the number of rows
  - `ncol()`: returns the number of columns
- Summary:
  - `colnames()` (or just `names()`): returns the column names
  - `str()`: structure of the object and information about the class, length and content of each column
  - `summary()`: works differently depending on what kind of object you pass to it. Passing a data frame to the `summary()` function prints out useful summary statistics about numeric column (min, max, median, mean, etc.)

```
head(ydat)
```

```
# A tibble: 6 x 7
  symbol systematic_name nutrient  rate expression bp                      mf
  <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                   <chr>
1 SFB2   YNL049C         Glucose   0.05      -0.24 ER to Golgi transport   mole~
2 <NA>   YNL095C         Glucose   0.05       0.28 biological process unk~ mole~
3 QRI7   YDL104C         Glucose   0.05      -0.02 proteolysis and peptid~ meta~
4 CFT2   YLR115W         Glucose   0.05      -0.33 mRNA polyadenylylation* RNA ~
5 SSO2   YMR183C         Glucose   0.05       0.05 vesicle fusion*         t-SN~
6 PSP2   YML017W         Glucose   0.05      -0.69 biological process unk~ mole~
```

```
tail(ydat)
```

```
# A tibble: 6 x 7
  symbol systematic_name nutrient  rate expression bp                      mf
  <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                   <chr>
1 DOA1   YKL213C         Uracil     0.3       0.14 ubiquitin-dependent pr~ mole~
2 KRE1   YNL322C         Uracil     0.3       0.28 cell wall organization~ stru~
3 MTL1   YGR023W         Uracil     0.3       0.27 cell wall organization~ mole~
4 KRE9   YJL174W         Uracil     0.3       0.43 cell wall organization~ mole~
5 UTH1   YKR042W         Uracil     0.3       0.19 mitochondrion organiza~ mole~
6 <NA>   YOL111C         Uracil     0.3       0.04 biological process unk~ mole~
```

```
dim(ydat)
```

```
[1] 198430      7
```

```
names(ydat)
```

```
[1] "symbol"          "systematic_name" "nutrient"        "rate"
[5] "expression"      "bp"              "mf"
```

```
str(ydat)
```

```
spc_tbl_ [198,430 x 7] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
 $ symbol         : chr [1:198430] "SFB2" NA "QRI7" "CFT2" ...
 $ systematic_name: chr [1:198430] "YNL049C" "YNL095C" "YDL104C" "YLR115W" ...
 $ nutrient       : chr [1:198430] "Glucose" "Glucose" "Glucose" "Glucose" ...
 $ rate           : num [1:198430] 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 ...
 $ expression     : num [1:198430] -0.24 0.28 -0.02 -0.33 0.05 -0.69 -0.55 -0.75 -0.24 -0.16
 $ bp             : chr [1:198430] "ER to Golgi transport" "biological process unknown" "prot
 $ mf             : chr [1:198430] "molecular function unknown" "molecular function unknown"
 - attr(*, "spec")=
  .. cols(
  ..   symbol = col_character(),
  ..   systematic_name = col_character(),
  ..   nutrient = col_character(),
  ..   rate = col_double(),
  ..   expression = col_double(),
  ..   bp = col_character(),
  ..   mf = col_character()
  .. )
 - attr(*, "problems")=<externalptr>
```

```
summary(ydat)
```

```
   symbol          systematic_name      nutrient             rate
Length:198430      Length:198430      Length:198430      Min.   :0.050
Class :character   Class :character   Class :character   1st Qu.:0.100
Mode  :character   Mode  :character   Mode  :character   Median :0.200
                                                         Mean   :0.175
                                                         3rd Qu.:0.250
                                                         Max.   :0.300
```

```
  expression          bp                 mf
Min.   :-6.50   Length:198430      Length:198430
1st Qu.:-0.29   Class :character   Class :character
Median : 0.00   Mode  :character   Mode  :character
Mean   : 0.00
3rd Qu.: 0.29
Max.   : 6.64
```

2.3.2 Other packages

The `glimpse()` function is available once you load the **dplyr** library, and it's like `str()` but its display is a little bit better.

```
glimpse(ydat)
```

```
Rows: 198,430
Columns: 7
$ symbol          <chr> "SFB2", NA, "QRI7", "CFT2", "SSO2", "PSP2", "RIB2", "V~
$ systematic_name <chr> "YNL049C", "YNL095C", "YDL104C", "YLR115W", "YMR183C",~
$ nutrient        <chr> "Glucose", "Glucose", "Glucose", "Glucose", "Glucose",~
$ rate            <dbl> 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, ~
$ expression      <dbl> -0.24, 0.28, -0.02, -0.33, 0.05, -0.69, -0.55, -0.75, ~
$ bp              <chr> "ER to Golgi transport", "biological process unknown",~
$ mf              <chr> "molecular function unknown", "molecular function unkn~
```

The **skimr** package has a nice function, skim, that provides summary statistics the user can skim quickly to understand your data. You can install it with `install.packages("skimr")` if you don't have it already.

```
library(skimr)
skim(ydat)
```

Table 2.1: Data summary

Name	ydat
Number of rows	198430
Number of columns	7
Column type frequency:	
character	5

numeric	2
Group variables	None

Variable type: character

skim_variable	n_missing	complete_rate	min	max	empty	n_unique	whitespace
symbol	47250	0.76	2	9	0	4210	0
systematic_name	0	1.00	5	9	0	5536	0
nutrient	0	1.00	6	9	0	6	0
bp	7663	0.96	7	82	0	880	0
mf	7663	0.96	11	125	0	1085	0

Variable type: numeric

skim_variable	n_missing	complete_rate	mean	sd	p0	p25	p50	p75	p100	hist
rate	0	1	0.18	0.09	0.05	0.10	0.2	0.25	0.30	
expression	0	1	0.00	0.67	-6.50	-0.29	0.0	0.29	6.64	

2.4 Accessing variables & subsetting data frames

We can access individual variables within a data frame using the `$` operator, e.g., `mydataframe$specificVariable`. Let's print out all the gene names in the data. Then let's calculate the average expression across all conditions, all genes (using the built-in `mean()` function).

```
# display all gene symbols
ydat$symbol
```

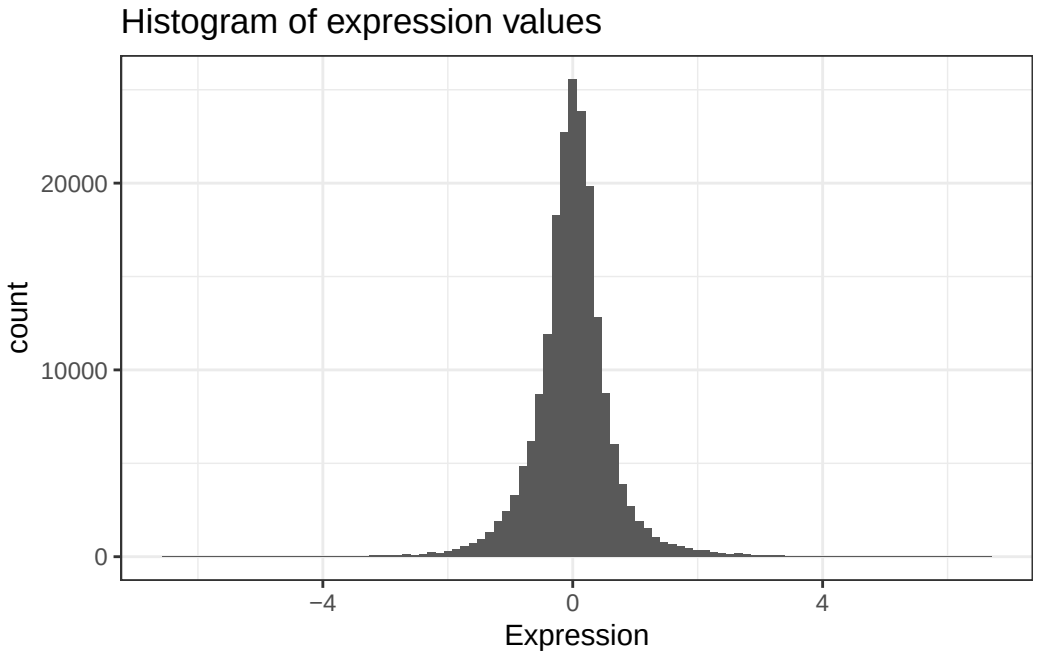
```
 [1] "SFB2"     NA         "QRI7"     "CFT2"     "SSO2"     "PSP2"
 [7] "RIB2"     "VMA13"    "EDC3"     "VPS5"     NA         "AMN1"
[13] "SCW11"    "DSE2"     "COX15"    "SPE1"     "MTF1"     "KSS1"
[19] NA         NA         "YAP7"     NA         "YVC1"     "CDC40"
[25] NA         "RMD1"     "PCL6"     "AI4"      "GGC1"     "SUL1"
[31] "RAD57"    NA         "PER1"     "YHC3"     "SGE1"     "HNM1"
[37] "SWI1"     "NAM8"     NA         "BGL2"     "ACT1"     NA
[43] "SFL1"     "OYE3"     "MMP1"     "MHT1"     "SUL2"     "IPP1"
```

```
 [49] "CWP1"     "SNF11"    "PEX25"    "ELO1"     NA         "CDC13"
 [55] "FKH1"     "SWD1"     NA         "HOF1"     "HOC1"     "BNI5"
 [61] "CSN12"    "PGS1"     "MLP2"     "HRP1"     NA         "SEC39"
 [67] "ECM31"    NA         NA         "ADE4"     "ABC1"     "DLD2"
 [73] "PHA2"     NA         "HAP3"     "MRPL23"   NA         NA
 [79] "MRPL16"   NA         NA         NA         NA         "AI3"
 [85] "COX1"     NA         "VAR1"     "COX3"     "COX2"     "AI5_BETA"
 [91] "AI2"      NA         NA         "GPI18"    "COS9"     NA
 [97] NA         "PRP46"    "XDJ1"     "SLG1"     "MAM3"     "AEP1"
[103] "UGO1"     NA         "RSC2"     "YAP1801"  "ZPR1"     "BCD1"
[109] "UBP10"    "SLD3"     "RLF2"     "LRO1"     NA         "ITR2"
[115] "ABP140"   "STT3"     "PTC2"     "STE20"    "HRD3"     "CWH43"
[121] "ASK10"    "MPE1"     "SWC3"     "TSA1"     "ADE17"    "GFD2"
[127] "PXR1"     NA         "BUD14"    "AUS1"     "NHX1"     "NTE1"
[133] NA         "KIN3"     "BUD4"     "SLI15"    "PMT4"     "AVT5"
[139] "CHS2"     "GPI13"    "KAP95"    "EFT2"     "EFT1"     "GAS1"
[145] "CYK3"     "COQ2"     "PSD1"     NA         "PAC1"     "SUR7"
[151] "RAX1"     "DFM1"     "RBD2"     NA         "YIP4"     "SRB2"
[157] "HOL1"     "MEP3"     NA         "FEN2"     NA         "RFT1"
[163] NA         "MCK1"     "GPI10"    "APT1"     NA         NA
[169] "CPT1"     "ERV29"    "SFK1"     NA         "SEC20"    "TIR4"
[175] NA         NA         "ARC35"    "SOL1"     "BIO2"     "ASC1"
[181] "RBG1"     "PTC4"     NA         "OXA1"     "SIT4"     "PUB1"
[187] "FPR4"     "FUN12"    "DPH2"     "DPS1"     "DLD1"     "ASN2"
[193] "TRM9"     "DED81"    "SRM1"     "SAM50"    "POP2"     "FAA4"
[199] NA         "CEM1"
 [ reached getOption("max.print") -- omitted 198230 entries ]
```

```
#mean expression
mean(ydat$expression)
```

```
[1] 0.00337
```

Now that's not too interesting. This is the average gene expression across all genes, across all conditions. The data is actually scaled/centered around zero:



We might be interested in the average expression of genes with a particular biological function, and how that changes over different growth rates restricted by particular nutrients. This is the kind of thing we're going to do in the next section.

Exercise 1

1. What's the standard deviation expression (hint: get help on the `sd` function with `?sd`).
2. What's the range of rate represented in the data? (hint: `range()`).

2.5 BONUS: Preview to advanced manipulation

What if we wanted show the mean expression, standard deviation, and correlation between growth rate and expression, separately for each limiting nutrient, separately for each gene, for all genes involved in the leucine biosynthesis pathway?

```
ydat |>
  filter(bp=="leucine biosynthesis") |>
  group_by(nutrient, symbol) |>
  summarize(mean=mean(expression), sd=sd(expression), r=cor(rate, expression))
```

nutrient	symbol	mean	sd	r
Ammonia	LEU1	-0.82	0.39	0.66
Ammonia	LEU2	-0.54	0.38	-0.19
Ammonia	LEU4	-0.37	0.56	-0.67
Ammonia	LEU9	-1.01	0.64	0.87
Glucose	LEU1	-0.55	0.41	0.98
Glucose	LEU2	-0.39	0.33	0.90
Glucose	LEU4	1.09	1.01	-0.97
Glucose	LEU9	-0.17	0.35	0.35
Leucine	LEU1	2.70	1.08	-0.95
Leucine	LEU2	0.28	1.16	-0.97
Leucine	LEU4	0.80	1.06	-0.97
Leucine	LEU9	0.39	0.18	-0.77
Phosphate	LEU1	-0.43	0.27	0.95
Phosphate	LEU2	-0.26	0.19	0.70
Phosphate	LEU4	-0.99	0.11	0.24
Phosphate	LEU9	-1.12	0.53	0.90
Sulfate	LEU1	-1.17	0.34	0.98
Sulfate	LEU2	-0.96	0.30	0.57
Sulfate	LEU4	-0.24	0.43	-0.60
Sulfate	LEU9	-1.24	0.55	0.99
Uracil	LEU1	-0.74	0.73	0.89
Uracil	LEU2	0.18	0.13	-0.07
Uracil	LEU4	-0.65	0.44	0.77
Uracil	LEU9	-1.02	0.91	0.94

Neat eh? We'll learn how to do that in the advanced manipulation with dplyr section (Chapter 3).

3 Data Manipulation with dplyr

Data analysis involves a large amount of janitor work – munging and cleaning data to facilitate downstream data analysis. This chapter demonstrates techniques for advanced data manipulation and analysis with the split-apply-combine strategy. We will use the dplyr package in R to effectively manipulate and conditionally compute summary statistics over subsets of a "big" dataset containing many observations.

This chapter assumes a basic familiarity with R (Chapter 1) and data frames (Chapter 2).

Recommended reading: Review the *Introduction* (10.1) and *Tibbles vs. data.frame* (10.3) sections of the ***R for Data Science* book**. We will initially be using the `read_*` functions from the **readr** package. These functions load data into a *tibble* instead of R's traditional data.frame. Tibbles are data frames, but they tweak some older behaviors to make life a little easier. These sections explain the few key small differences between traditional data.frames and tibbles.

3.1 Review

3.1.1 Our data

We're going to use the yeast gene expression dataset described on the data frames chapter in Chapter 2. This is a cleaned up version of a gene expression dataset from Brauer et al. Coordination of Growth Rate, Cell Cycle, Stress Response, and Metabolic Activity in Yeast (2008) *Mol Biol Cell* 19:352-367. This data is from a gene expression microarray, and in this paper the authors are examining the relationship between growth rate and gene expression in yeast cultures limited by one of six different nutrients (glucose, leucine, ammonium, sulfate, phosphate, uracil). If you give yeast a rich media loaded with nutrients except restrict the supply of a *single* nutrient, you can control the growth rate to any rate you choose. By starving yeast of specific nutrients you can find genes that:

1. **Raise or lower their expression in response to growth rate**. Growth-rate dependent expression patterns can tell us a lot about cell cycle control, and how the cell responds to stress. The authors found that expression of >25% of all yeast genes is linearly correlated with growth rate, independent of the limiting nutrient. They also

found that the subset of negatively growth-correlated genes is enriched for peroxisomal functions, and positively correlated genes mainly encode ribosomal functions.

2. **Respond differently when different nutrients are being limited**. If you see particular genes that respond very differently when a nutrient is sharply restricted, these genes might be involved in the transport or metabolism of that specific nutrient.

You can download the cleaned up version of the data here. The file is called **brauer2007_tidy.csv**. Later on we'll actually start with the original raw data (minimally processed) and manipulate it so that we can make it more amenable for analysis.

3.1.2 Reading in data

We need to load both the dplyr and readr packages for efficiently reading in and displaying this data. We're also going to use many other functions from the dplyr package. Make sure you have these packages installed as described on the setup chapter (Appendix A).

```
# Load packages
library(readr)
library(dplyr)

# Read in data
ydat <- read_csv(file="data/brauer2007_tidy.csv")

# Display the data
ydat

# Optionally, bring up the data in a viewer window
# View(ydat)
```

```
# A tibble: 198,430 x 7
   symbol systematic_name nutrient  rate expression bp                     mf
   <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                  <chr>
 1 SFB2   YNL049C         Glucose   0.05      -0.24 ER to Golgi transport  mole~
 2 <NA>   YNL095C         Glucose   0.05       0.28 biological process un~ mole~
 3 QRI7   YDL104C         Glucose   0.05      -0.02 proteolysis and pepti~ meta~
 4 CFT2   YLR115W         Glucose   0.05      -0.33 mRNA polyadenylylatio~ RNA ~
 5 SSO2   YMR183C         Glucose   0.05       0.05 vesicle fusion*        t-SN~
 6 PSP2   YML017W         Glucose   0.05      -0.69 biological process un~ mole~
 7 RIB2   YOL066C         Glucose   0.05      -0.55 riboflavin biosynthes~ pseu~
 8 VMA13  YPR036W         Glucose   0.05      -0.75 vacuolar acidification hydr~
 9 EDC3   YEL015W         Glucose   0.05      -0.24 deadenylylation-indep~ mole~
```

```
10 VPS5   YOR069W         Glucose   0.05      -0.16 protein retention in ~ prot~
# i 198,420 more rows
```

3.2 The dplyr package

The dplyr package is a relatively new R package that makes data manipulation fast and easy. It imports functionality from another package called magrittr that allows you to chain commands together into a pipeline that will completely change the way you write R code such that you're writing code the way you're thinking about the problem.

When you read in data with the readr package (`read_csv()`) and you had the dplyr package loaded already, the data frame takes on this "special" class of data frames called a `tbl` (pronounced "tibble"), which you can see with `class(ydat)`. If you have other "regular" data frames in your workspace, the `as_tibble()` function will convert it into the special dplyr `tbl` that displays nicely (e.g.: `iris <- as_tibble(iris)`). You don't have to turn all your data frame objects into tibbles, but it does make working with large datasets a bit easier.

You can read more about tibbles in Tibbles chapter in *R for Data Science* or in the tibbles vignette. They keep most of the features of data frames, and drop the features that used to be convenient but are now frustrating (i.e. converting character vectors to factors). You can read more about the differences between data frames and tibbles in this section of the tibbles vignette, but the major convenience for us concerns **printing** (aka displaying) a tibble to the screen. When you print (i.e., display) a tibble, it only shows the first 10 rows and all the columns that fit on one screen. It also prints an abbreviated description of the column type. You can control the default appearance with options:

- `options(tibble.print_max = n, tibble.print_min = m)`: if there are more than n rows, print only the first m rows. Use `options(tibble.print_max = Inf)` to always show all rows.
- `options(tibble.width = Inf)` will always print all columns, regardless of the width of the screen.

3.3 dplyr verbs

The dplyr package gives you a handful of useful **verbs** for managing data. On their own they don't do anything that base R can't do. Here are some of the *single-table* verbs we'll be working with in this chapter (single-table meaning that they only work on a single table – contrast that to *two-table* verbs used for joining data together, which we'll cover in a later chapter).

1. `filter()`

2. `select()`
3. `mutate()`
4. `arrange()`
5. `summarize()`
6. `group_by()`

They all take a data frame or tibble as their input for the first argument, and they all return a data frame or tibble as output.

3.3.1 filter()

If you want to filter **rows** of the data where some condition is true, use the `filter()` function.

1. The first argument is the data frame you want to filter, e.g. `filter(mydata, ....`
2. The second argument is a condition you must satisfy, e.g. `filter(ydat, symbol == "LEU1")`. If you want to satisfy *all* of multiple conditions, you can use the "and" operator, `&`. The "or" operator `|` (the pipe character, usually shift-backslash) will return a subset that meet *any* of the conditions.

- `==`: Equal to
- `!=`: Not equal to
- `>`, `>=`: Greater than, greater than or equal to
- `<`, `<=`: Less than, less than or equal to

Let's try it out. For this to work you have to have already loaded the dplyr package. Let's take a look at LEU1, a gene involved in leucine synthesis.

```
# First, make sure you've loaded the dplyr package
library(dplyr)

# Look at a single gene involved in leucine synthesis pathway
filter(ydat, symbol == "LEU1")
```

```
# A tibble: 36 x 7
   symbol systematic_name nutrient  rate expression bp                  mf
   <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>               <chr>
 1 LEU1   YGL009C         Glucose   0.05      -1.12 leucine biosynthesis 3-isop~
 2 LEU1   YGL009C         Glucose   0.1       -0.77 leucine biosynthesis 3-isop~
 3 LEU1   YGL009C         Glucose   0.15      -0.67 leucine biosynthesis 3-isop~
 4 LEU1   YGL009C         Glucose   0.2       -0.59 leucine biosynthesis 3-isop~
 5 LEU1   YGL009C         Glucose   0.25      -0.2  leucine biosynthesis 3-isop~
```

```
 6 LEU1   YGL009C         Glucose   0.3        0.03 leucine biosynthesis 3-isop~
 7 LEU1   YGL009C         Ammonia   0.05      -0.76 leucine biosynthesis 3-isop~
 8 LEU1   YGL009C         Ammonia   0.1       -1.17 leucine biosynthesis 3-isop~
 9 LEU1   YGL009C         Ammonia   0.15      -1.2  leucine biosynthesis 3-isop~
10 LEU1   YGL009C         Ammonia   0.2       -1.02 leucine biosynthesis 3-isop~
# i 26 more rows
```

```
  # Optionally, bring that result up in a View window
  # View(filter(ydat, symbol == "LEU1"))

  # Look at multiple genes
  filter(ydat, symbol=="LEU1" | symbol=="ADH2")
```

```
# A tibble: 72 x 7
   symbol systematic_name nutrient  rate expression bp                    mf
   <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                 <chr>
 1 LEU1   YGL009C         Glucose   0.05      -1.12 leucine biosynthesis 3-isop~
 2 ADH2   YMR303C         Glucose   0.05       6.28 fermentation*        alcoho~
 3 LEU1   YGL009C         Glucose   0.1       -0.77 leucine biosynthesis 3-isop~
 4 ADH2   YMR303C         Glucose   0.1        5.81 fermentation*        alcoho~
 5 LEU1   YGL009C         Glucose   0.15      -0.67 leucine biosynthesis 3-isop~
 6 ADH2   YMR303C         Glucose   0.15       5.64 fermentation*        alcoho~
 7 LEU1   YGL009C         Glucose   0.2       -0.59 leucine biosynthesis 3-isop~
 8 ADH2   YMR303C         Glucose   0.2        5.1  fermentation*        alcoho~
 9 LEU1   YGL009C         Glucose   0.25      -0.2  leucine biosynthesis 3-isop~
10 ADH2   YMR303C         Glucose   0.25       1.89 fermentation*        alcoho~
# i 62 more rows
```

```
  # Look at LEU1 expression at a low growth rate due to nutrient depletion
  # Notice how LEU1 is highly upregulated when leucine is depleted!
  filter(ydat, symbol=="LEU1" & rate==.05)
```

```
# A tibble: 6 x 7
  symbol systematic_name nutrient   rate expression bp                    mf
  <chr>  <chr>           <chr>     <dbl>      <dbl> <chr>                 <chr>
1 LEU1   YGL009C         Glucose    0.05      -1.12 leucine biosynthesis 3-isop~
2 LEU1   YGL009C         Ammonia    0.05      -0.76 leucine biosynthesis 3-isop~
3 LEU1   YGL009C         Phosphate  0.05      -0.81 leucine biosynthesis 3-isop~
4 LEU1   YGL009C         Sulfate    0.05      -1.57 leucine biosynthesis 3-isop~
5 LEU1   YGL009C         Leucine    0.05       3.84 leucine biosynthesis 3-isop~
6 LEU1   YGL009C         Uracil     0.05      -2.07 leucine biosynthesis 3-isop~
```

```
# But expression goes back down when the growth/nutrient restriction is relaxed
filter(ydat, symbol=="LEU1" & rate==.3)
```

```
# A tibble: 6 x 7
  symbol systematic_name nutrient   rate expression bp                    mf
  <chr>  <chr>           <chr>     <dbl>      <dbl> <chr>                 <chr>
1 LEU1   YGL009C         Glucose     0.3       0.03 leucine biosynthesis 3-isop~
2 LEU1   YGL009C         Ammonia     0.3      -0.22 leucine biosynthesis 3-isop~
3 LEU1   YGL009C         Phosphate   0.3      -0.07 leucine biosynthesis 3-isop~
4 LEU1   YGL009C         Sulfate     0.3      -0.76 leucine biosynthesis 3-isop~
5 LEU1   YGL009C         Leucine     0.3       0.87 leucine biosynthesis 3-isop~
6 LEU1   YGL009C         Uracil      0.3      -0.16 leucine biosynthesis 3-isop~
```

```
# Show only stats for LEU1 and Leucine depletion.
# LEU1 expression starts off high and drops
filter(ydat, symbol=="LEU1" & nutrient=="Leucine")
```

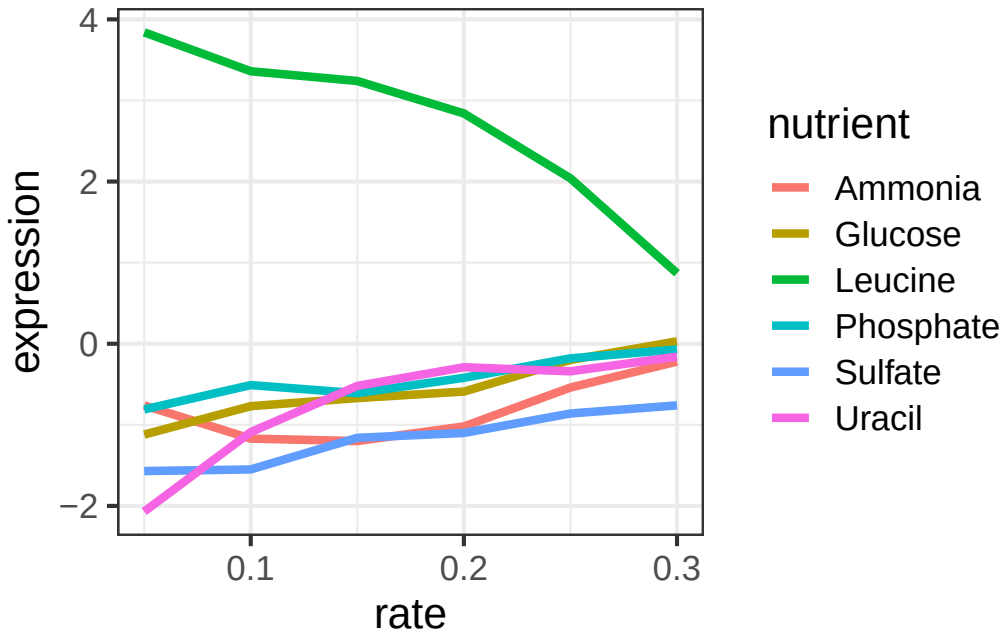
```
# A tibble: 6 x 7
  symbol systematic_name nutrient  rate expression bp                    mf
  <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                 <chr>
1 LEU1   YGL009C         Leucine   0.05       3.84 leucine biosynthesis 3-isopr~
2 LEU1   YGL009C         Leucine   0.1        3.36 leucine biosynthesis 3-isopr~
3 LEU1   YGL009C         Leucine   0.15       3.24 leucine biosynthesis 3-isopr~
4 LEU1   YGL009C         Leucine   0.2        2.84 leucine biosynthesis 3-isopr~
5 LEU1   YGL009C         Leucine   0.25       2.04 leucine biosynthesis 3-isopr~
6 LEU1   YGL009C         Leucine   0.3        0.87 leucine biosynthesis 3-isopr~
```

```
# What about LEU1 expression with other nutrients being depleted?
filter(ydat, symbol=="LEU1" & nutrient=="Glucose")
```

```
# A tibble: 6 x 7
  symbol systematic_name nutrient  rate expression bp                    mf
  <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                 <chr>
1 LEU1   YGL009C         Glucose   0.05      -1.12 leucine biosynthesis 3-isopr~
2 LEU1   YGL009C         Glucose   0.1       -0.77 leucine biosynthesis 3-isopr~
3 LEU1   YGL009C         Glucose   0.15      -0.67 leucine biosynthesis 3-isopr~
4 LEU1   YGL009C         Glucose   0.2       -0.59 leucine biosynthesis 3-isopr~
5 LEU1   YGL009C         Glucose   0.25      -0.2  leucine biosynthesis 3-isopr~
6 LEU1   YGL009C         Glucose   0.3        0.03 leucine biosynthesis 3-isopr~
```

Let's look at this graphically. Don't worry about what these commands are doing just yet - we'll cover that later on when we talk about ggplot2. Here's I'm taking the filtered dataset containing just expression estimates for LEU1 where I have 36 rows (one for each of 6 nutrients $\times$ 6 growth rates), and I'm *piping* that dataset to the plotting function, where I'm plotting rate on the x-axis, expression on the y-axis, mapping the value of nutrient to the color, and using a line plot to display the data.

```
library(ggplot2)
filter(ydat, symbol=="LEU1") |>
  ggplot(aes(rate, expression, colour=nutrient)) + geom_line(lwd=1.5)
```



Look closely at that! LEU1 is *highly expressed* when starved of leucine because the cell has to synthesize its own! And as the amount of leucine in the environment (the growth *rate*) increases, the cell can worry less about synthesizing leucine, so LEU1 expression goes back down. Consequently the cell can devote more energy into other functions, and we see other genes' expression very slightly raising.

Exercise 1

1. Display the data where the gene ontology biological process (the `bp` variable) is "leucine biosynthesis" (case-sensitive) *and* the limiting nutrient was Leucine. (Answer should return a 24-by-7 data frame – 4 genes $\times$ 6 growth rates).
2. Gene/rate combinations had high expression (in the top 1% of expressed genes)?

Hint: see `?quantile` and try `quantile(ydat$expression, probs=.99)` to see the expression value which is higher than 99% of all the data, then `filter()` based on that. Try wrapping your answer with a `View()` function so you can see the whole thing. What does it look like those genes are doing? Answer should return a 1971-by-7 data frame.

3.3.1.1 Aside: Writing Data to File

What we've done up to this point is read in data from a file (`read_csv(...)`), and assigning that to an object in our *workspace* (`ydat <- ...`). When we run operations like `filter()` on our data, consider two things:

1. The `ydat` object in our workspace is not being modified directly. That is, we can `filter(ydat, ...)`, and a result is returned to the screen, but `ydat` remains the same. This effect is similar to what we demonstrated in our first session.

```
# Assign the value '50' to the weight object.
weight <- 50

# Print out weight to the screen (50)
weight

# What's the value of weight plus 10?
weight + 10

# Weight is still 50
weight

# Weight is only modified if we *reassign* weight to the modified value
weight <- weight+10
# Weight is now 60
weight
```

2. More importantly, the *data file on disk* (`data/brauer2007_tidy.csv`) is *never* modified. No matter what we do to ydat, the file is never modified. If we want to *save* the result of an operation to a file on disk, we can assign the result of an operation to an object, and `write_csv` that object to disk. See the help for `?write_csv` (note, `write_csv()` with an underscore is part of the **readr** package – not to be confused with the built-in `write.csv()` function).

```r
# What's the result of this filter operation?
filter(ydat, nutrient=="Leucine" & bp=="leucine biosynthesis")

# Assign the result to a new object
leudat <- filter(ydat, nutrient=="Leucine" & bp=="leucine biosynthesis")

# Write that out to disk
write_csv(leudat, "leucinedata.csv")
```

Note that this is different than saving your *entire workspace to an Rdata file*, which would contain all the objects we've created (weight, ydat, leudat, etc).

3.3.2 select()

The `filter()` function allows you to return only certain *rows* matching a condition. The `select()` function returns only certain *columns*. The first argument is the data, and subsequent arguments are the columns you want.

```r
# Select just the symbol and systematic_name
select(ydat, symbol, systematic_name)
```

```
# A tibble: 198,430 x 2
   symbol systematic_name
   <chr>  <chr>
 1 SFB2   YNL049C
 2 <NA>   YNL095C
 3 QRI7   YDL104C
 4 CFT2   YLR115W
 5 SSO2   YMR183C
 6 PSP2   YML017W
 7 RIB2   YOL066C
 8 VMA13  YPR036W
 9 EDC3   YEL015W
10 VPS5   YOR069W
# i 198,420 more rows
```

```r
# Alternatively, just remove columns. Remove the bp and mf columns.
select(ydat, -bp, -mf)
```

```
# A tibble: 198,430 x 5
   symbol systematic_name nutrient  rate expression
   <chr>  <chr>           <chr>    <dbl>      <dbl>
 1 SFB2   YNL049C         Glucose   0.05      -0.24
 2 <NA>   YNL095C         Glucose   0.05       0.28
 3 QRI7   YDL104C         Glucose   0.05      -0.02
 4 CFT2   YLR115W         Glucose   0.05      -0.33
 5 SSO2   YMR183C         Glucose   0.05       0.05
 6 PSP2   YML017W         Glucose   0.05      -0.69
 7 RIB2   YOL066C         Glucose   0.05      -0.55
 8 VMA13  YPR036W         Glucose   0.05      -0.75
 9 EDC3   YEL015W         Glucose   0.05      -0.24
10 VPS5   YOR069W         Glucose   0.05      -0.16
# i 198,420 more rows
```

```
# Notice that the original data doesn't change!
ydat
```

```
# A tibble: 198,430 x 7
   symbol systematic_name nutrient  rate expression bp                    mf
   <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                 <chr>
 1 SFB2   YNL049C         Glucose   0.05      -0.24 ER to Golgi transport  mole~
 2 <NA>   YNL095C         Glucose   0.05       0.28 biological process un~ mole~
 3 QRI7   YDL104C         Glucose   0.05      -0.02 proteolysis and pepti~ meta~
 4 CFT2   YLR115W         Glucose   0.05      -0.33 mRNA polyadenylylatio~ RNA ~
 5 SSO2   YMR183C         Glucose   0.05       0.05 vesicle fusion*        t-SN~
 6 PSP2   YML017W         Glucose   0.05      -0.69 biological process un~ mole~
 7 RIB2   YOL066C         Glucose   0.05      -0.55 riboflavin biosynthes~ pseu~
 8 VMA13  YPR036W         Glucose   0.05      -0.75 vacuolar acidification hydr~
 9 EDC3   YEL015W         Glucose   0.05      -0.24 deadenylylation-indep~ mole~
10 VPS5   YOR069W         Glucose   0.05      -0.16 protein retention in ~ prot~
# i 198,420 more rows
```

Notice above how the original data doesn't change. We're selecting out only certain columns of interest and throwing away columns we don't care about. If we wanted to *keep* this data, we would need to *reassign* the result of the `select()` operation to a new object. Let's make a new object called `nogo` that does not contain the GO annotations. Notice again how the original data is unchanged.

```
# create a new dataset without the go annotations.
nogo <- select(ydat, -bp, -mf)
```

```
nogo
```

```
# A tibble: 198,430 x 5
   symbol systematic_name nutrient  rate expression
   <chr>  <chr>           <chr>    <dbl>      <dbl>
 1 SFB2   YNL049C         Glucose   0.05      -0.24
 2 <NA>   YNL095C         Glucose   0.05       0.28
 3 QRI7   YDL104C         Glucose   0.05      -0.02
 4 CFT2   YLR115W         Glucose   0.05      -0.33
 5 SSO2   YMR183C         Glucose   0.05       0.05
 6 PSP2   YML017W         Glucose   0.05      -0.69
 7 RIB2   YOL066C         Glucose   0.05      -0.55
 8 VMA13  YPR036W         Glucose   0.05      -0.75
 9 EDC3   YEL015W         Glucose   0.05      -0.24
10 VPS5   YOR069W         Glucose   0.05      -0.16
# i 198,420 more rows
```

```
# we could filter this new dataset
filter(nogo, symbol=="LEU1" & rate==.05)
```

```
# A tibble: 6 x 5
  symbol systematic_name nutrient   rate expression
  <chr>  <chr>           <chr>     <dbl>      <dbl>
1 LEU1   YGL009C         Glucose    0.05      -1.12
2 LEU1   YGL009C         Ammonia    0.05      -0.76
3 LEU1   YGL009C         Phosphate  0.05      -0.81
4 LEU1   YGL009C         Sulfate    0.05      -1.57
5 LEU1   YGL009C         Leucine    0.05       3.84
6 LEU1   YGL009C         Uracil     0.05      -2.07
```

```
# Notice how the original data is unchanged - still have all 7 columns
ydat
```

```
# A tibble: 198,430 x 7
   symbol systematic_name nutrient  rate expression bp                    mf
   <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                 <chr>
 1 SFB2   YNL049C         Glucose   0.05      -0.24 ER to Golgi transport  mole~
 2 <NA>   YNL095C         Glucose   0.05       0.28 biological process un~ mole~
 3 QRI7   YDL104C         Glucose   0.05      -0.02 proteolysis and pepti~ meta~
```

```
 4 CFT2   YLR115W         Glucose   0.05      -0.33 mRNA polyadenylylatio~ RNA ~
 5 SSO2   YMR183C         Glucose   0.05       0.05 vesicle fusion*        t-SN~
 6 PSP2   YML017W         Glucose   0.05      -0.69 biological process un~ mole~
 7 RIB2   YOL066C         Glucose   0.05      -0.55 riboflavin biosynthes~ pseu~
 8 VMA13  YPR036W         Glucose   0.05      -0.75 vacuolar acidification hydr~
 9 EDC3   YEL015W         Glucose   0.05      -0.24 deadenylylation-indep~ mole~
10 VPS5   YOR069W         Glucose   0.05      -0.16 protein retention in ~ prot~
# i 198,420 more rows
```

3.3.3 mutate()

The `mutate()` function adds new columns to the data. Remember, it doesn't actually modify the data frame you're operating on, and the result is transient unless you assign it to a new object or reassign it back to itself (generally, not always a good practice).

The expression level reported here is the log_2 of the sample signal divided by the signal in the reference channel, where the reference RNA for all samples was taken from the glucose-limited chemostat grown at a dilution rate of 0.25 h^{-1}. Let's mutate this data to add a new variable called "signal" that's the actual raw signal ratio instead of the log-transformed signal.

```
mutate(nogo, signal=2^expression)
```

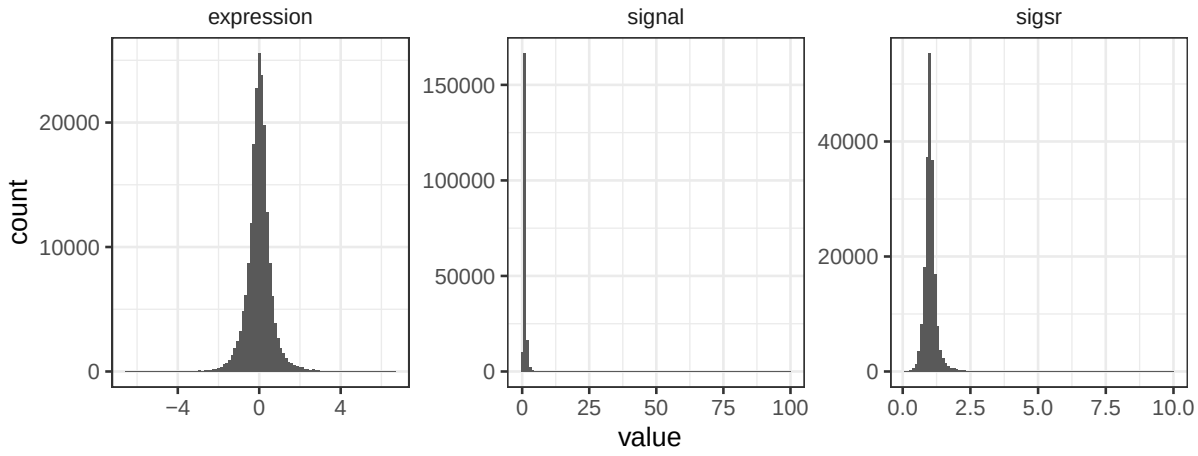
Mutate has a nice little feature too in that it's "lazy." You can mutate and add one variable, then continue mutating to add more variables based on that variable. Let's make another column that's the square root of the signal ratio.

```
mutate(nogo, signal=2^expression, sigsr=sqrt(signal))
```

```
# A tibble: 198,430 x 7
   symbol systematic_name nutrient  rate expression signal sigsr
   <chr>  <chr>           <chr>    <dbl>      <dbl>  <dbl> <dbl>
 1 SFB2   YNL049C         Glucose   0.05      -0.24  0.847 0.920
 2 <NA>   YNL095C         Glucose   0.05       0.28  1.21  1.10
 3 QRI7   YDL104C         Glucose   0.05      -0.02  0.986 0.993
 4 CFT2   YLR115W         Glucose   0.05      -0.33  0.796 0.892
 5 SSO2   YMR183C         Glucose   0.05       0.05  1.04  1.02
 6 PSP2   YML017W         Glucose   0.05      -0.69  0.620 0.787
 7 RIB2   YOL066C         Glucose   0.05      -0.55  0.683 0.826
 8 VMA13  YPR036W         Glucose   0.05      -0.75  0.595 0.771
 9 EDC3   YEL015W         Glucose   0.05      -0.24  0.847 0.920
10 VPS5   YOR069W         Glucose   0.05      -0.16  0.895 0.946
# i 198,420 more rows
```

Again, don't worry about the code here to make the plot – we'll learn about this later. Why do you think we log-transform the data prior to analysis?

```
library(tidyr)
mutate(nogo, signal=2^expression, sigsr=sqrt(signal)) |>
  gather(unit, value, expression:sigsr) |>
  ggplot(aes(value)) + geom_histogram(bins=100) + facet_wrap(~unit, scales="free")
```



3.3.4 arrange()

The `arrange()` function does what it sounds like. It takes a data frame or tbl and arranges (or sorts) by column(s) of interest. The first argument is the data, and subsequent arguments are columns to sort on. Use the `desc()` function to arrange by descending.

```
# arrange by gene symbol
arrange(ydat, symbol)
```

```
# A tibble: 198,430 x 7
   symbol systematic_name nutrient  rate expression bp                    mf
   <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                 <chr>
 1 AAC1   YMR056C         Glucose   0.05        1.5  aerobic respiration* ATP:AD~
 2 AAC1   YMR056C         Glucose   0.1         1.54 aerobic respiration* ATP:AD~
 3 AAC1   YMR056C         Glucose   0.15        1.16 aerobic respiration* ATP:AD~
 4 AAC1   YMR056C         Glucose   0.2         1.04 aerobic respiration* ATP:AD~
 5 AAC1   YMR056C         Glucose   0.25        0.84 aerobic respiration* ATP:AD~
 6 AAC1   YMR056C         Glucose   0.3         0.01 aerobic respiration* ATP:AD~
 7 AAC1   YMR056C         Ammonia   0.05        0.8  aerobic respiration* ATP:AD~
```

```
 8 AAC1   YMR056C         Ammonia   0.1        1.47 aerobic respiration* ATP:AD~
 9 AAC1   YMR056C         Ammonia   0.15       0.97 aerobic respiration* ATP:AD~
10 AAC1   YMR056C         Ammonia   0.2        0.76 aerobic respiration* ATP:AD~
# i 198,420 more rows
```

```
# arrange by expression (default: increasing)
arrange(ydat, expression)
```

```
# A tibble: 198,430 x 7
   symbol systematic_name nutrient   rate expression bp                    mf
   <chr>  <chr>           <chr>     <dbl>      <dbl> <chr>                 <chr>
 1 SUL1   YBR294W         Phosphate  0.05      -6.5  sulfate transport     sulf~
 2 SUL1   YBR294W         Phosphate  0.1       -6.34 sulfate transport     sulf~
 3 ADH2   YMR303C         Phosphate  0.1       -6.15 fermentation*         alco~
 4 ADH2   YMR303C         Phosphate  0.3       -6.04 fermentation*         alco~
 5 ADH2   YMR303C         Phosphate  0.25      -5.89 fermentation*         alco~
 6 SUL1   YBR294W         Uracil     0.05      -5.55 sulfate transport     sulf~
 7 SFC1   YJR095W         Phosphate  0.2       -5.52 fumarate transport*   succ~
 8 JEN1   YKL217W         Phosphate  0.3       -5.44 lactate transport     lact~
 9 MHT1   YLL062C         Phosphate  0.05      -5.36 sulfur amino acid me~ homo~
10 SFC1   YJR095W         Phosphate  0.25      -5.35 fumarate transport*   succ~
# i 198,420 more rows
```

```
# arrange by decreasing expression
arrange(ydat, desc(expression))
```

```
# A tibble: 198,430 x 7
   symbol systematic_name nutrient   rate expression bp                    mf
   <chr>  <chr>           <chr>     <dbl>      <dbl> <chr>                 <chr>
 1 GAP1   YKR039W         Ammonia    0.05       6.64 amino acid transport* L-pr~
 2 DAL5   YJR152W         Ammonia    0.05       6.64 allantoate transport  alla~
 3 GAP1   YKR039W         Ammonia    0.1        6.64 amino acid transport* L-pr~
 4 DAL5   YJR152W         Ammonia    0.1        6.64 allantoate transport  alla~
 5 DAL5   YJR152W         Ammonia    0.15       6.64 allantoate transport  alla~
 6 DAL5   YJR152W         Ammonia    0.2        6.64 allantoate transport  alla~
 7 DAL5   YJR152W         Ammonia    0.25       6.64 allantoate transport  alla~
 8 DAL5   YJR152W         Ammonia    0.3        6.64 allantoate transport  alla~
 9 GIT1   YCR098C         Phosphate  0.05       6.64 glycerophosphodieste~ glyc~
10 PHM6   YDR281C         Phosphate  0.05       6.64 biological process u~ mole~
# i 198,420 more rows
```

Exercise 2

1. First, re-run the command you used above to filter the data for genes involved in the "leucine biosynthesis" biological process *and* where the limiting nutrient is Leucine.
2. Wrap this entire filtered result with a call to `arrange()` where you'll arrange the result of #1 by the gene symbol.
3. Wrap this entire result in a `View()` statement so you can see the entire result.

3.3.5 summarize()

The `summarize()` function summarizes multiple values to a single value. On its own the `summarize()` function doesn't seem to be all that useful. The dplyr package provides a few convenience functions called `n()` and `n_distinct()` that tell you the number of observations or the number of distinct values of a particular variable.

Notice that summarize takes a data frame and returns a data frame. In this case it's a 1x1 data frame with a single row and a single column. The name of the column, by default is whatever the expression was used to summarize the data. This usually isn't pretty, and if we wanted to work with this resulting data frame later on, we'd want to name that returned value something easier to deal with.

```
# Get the mean expression for all genes
summarize(ydat, mean(expression))
```

```
# A tibble: 1 x 1
  `mean(expression)`
              <dbl>
1            0.00337
```

```
# Use a more friendly name, e.g., meanexp, or whatever you want to call it.
summarize(ydat, meanexp=mean(expression))
```

```
# A tibble: 1 x 1
  meanexp
    <dbl>
1 0.00337
```

```r
# Measure the correlation between rate and expression
summarize(ydat, r=cor(rate, expression))
```

```
# A tibble: 1 x 1
        r
    <dbl>
1 -0.0220
```

```r
# Get the number of observations
summarize(ydat, n())
```

```
# A tibble: 1 x 1
   `n()`
   <int>
1 198430
```

```r
# The number of distinct gene symbols in the data
summarize(ydat, n_distinct(symbol))
```

```
# A tibble: 1 x 1
  `n_distinct(symbol)`
                 <int>
1                 4211
```

3.3.6 group_by()

We saw that `summarize()` isn't that useful on its own. Neither is `group_by()` All this does is takes an existing data frame and coverts it into a grouped data frame where operations are performed by group.

```r
ydat
```

```
# A tibble: 198,430 x 7
   symbol systematic_name nutrient  rate expression bp                     mf
   <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                  <chr>
 1 SFB2   YNL049C         Glucose   0.05      -0.24 ER to Golgi transport  mole~
 2 <NA>   YNL095C         Glucose   0.05       0.28 biological process un~ mole~
```

```
 3 QRI7   YDL104C         Glucose   0.05      -0.02 proteolysis and pepti~ meta~
 4 CFT2   YLR115W         Glucose   0.05      -0.33 mRNA polyadenylylatio~ RNA ~
 5 SSO2   YMR183C         Glucose   0.05       0.05 vesicle fusion*        t-SN~
 6 PSP2   YML017W         Glucose   0.05      -0.69 biological process un~ mole~
 7 RIB2   YOL066C         Glucose   0.05      -0.55 riboflavin biosynthes~ pseu~
 8 VMA13  YPR036W         Glucose   0.05      -0.75 vacuolar acidification hydr~
 9 EDC3   YEL015W         Glucose   0.05      -0.24 deadenylylation-indep~ mole~
10 VPS5   YOR069W         Glucose   0.05      -0.16 protein retention in ~ prot~
# i 198,420 more rows
```

```
group_by(ydat, nutrient)
```

```
# A tibble: 198,430 x 7
# Groups:   nutrient [6]
   symbol systematic_name nutrient  rate expression bp                     mf
   <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                  <chr>
 1 SFB2   YNL049C         Glucose   0.05      -0.24 ER to Golgi transport  mole~
 2 <NA>   YNL095C         Glucose   0.05       0.28 biological process un~ mole~
 3 QRI7   YDL104C         Glucose   0.05      -0.02 proteolysis and pepti~ meta~
 4 CFT2   YLR115W         Glucose   0.05      -0.33 mRNA polyadenylylatio~ RNA ~
 5 SSO2   YMR183C         Glucose   0.05       0.05 vesicle fusion*        t-SN~
 6 PSP2   YML017W         Glucose   0.05      -0.69 biological process un~ mole~
 7 RIB2   YOL066C         Glucose   0.05      -0.55 riboflavin biosynthes~ pseu~
 8 VMA13  YPR036W         Glucose   0.05      -0.75 vacuolar acidification hydr~
 9 EDC3   YEL015W         Glucose   0.05      -0.24 deadenylylation-indep~ mole~
10 VPS5   YOR069W         Glucose   0.05      -0.16 protein retention in ~ prot~
# i 198,420 more rows
```

```
group_by(ydat, nutrient, rate)
```

```
# A tibble: 198,430 x 7
# Groups:   nutrient, rate [36]
   symbol systematic_name nutrient  rate expression bp                     mf
   <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                  <chr>
 1 SFB2   YNL049C         Glucose   0.05      -0.24 ER to Golgi transport  mole~
 2 <NA>   YNL095C         Glucose   0.05       0.28 biological process un~ mole~
 3 QRI7   YDL104C         Glucose   0.05      -0.02 proteolysis and pepti~ meta~
 4 CFT2   YLR115W         Glucose   0.05      -0.33 mRNA polyadenylylatio~ RNA ~
 5 SSO2   YMR183C         Glucose   0.05       0.05 vesicle fusion*        t-SN~
 6 PSP2   YML017W         Glucose   0.05      -0.69 biological process un~ mole~
```

```
 7 RIB2   YOL066C         Glucose   0.05      -0.55 riboflavin biosynthes~ pseu~
 8 VMA13  YPR036W         Glucose   0.05      -0.75 vacuolar acidification hydr~
 9 EDC3   YEL015W         Glucose   0.05      -0.24 deadenylylation-indep~ mole~
10 VPS5   YOR069W         Glucose   0.05      -0.16 protein retention in ~ prot~
# i 198,420 more rows
```

The real power comes in where `group_by()` and `summarize()` are used together. First, write the `group_by()` statement. Then wrap the result of that with a call to `summarize()`.

```
# Get the mean expression for each gene
# group_by(ydat, symbol)
summarize(group_by(ydat, symbol), meanexp=mean(expression))
```

```
# A tibble: 4,211 x 2
   symbol  meanexp
   <chr>     <dbl>
 1 AAC1     0.529
 2 AAC3    -0.216
 3 AAD10    0.438
 4 AAD14   -0.0717
 5 AAD16    0.242
 6 AAD4    -0.792
 7 AAD6     0.290
 8 AAH1     0.0461
 9 AAP1    -0.00361
10 AAP1'   -0.421
# i 4,201 more rows
```

```
# Get the correlation between rate and expression for each nutrient
# group_by(ydat, nutrient)
summarize(group_by(ydat, nutrient), r=cor(rate, expression))
```

```
# A tibble: 6 x 2
  nutrient        r
  <chr>       <dbl>
1 Ammonia   -0.0175
2 Glucose   -0.0112
3 Leucine   -0.0384
4 Phosphate -0.0194
5 Sulfate   -0.0166
6 Uracil    -0.0353
```

3.4 The pipe: |>

3.4.1 How |> works

This is where things get awesome. The dplyr package imports functionality from the magrittr package that lets you *pipe* the output of one function to the input of another, so you can avoid nesting functions. It looks like this: `|>`. You don't have to load the magrittr package to use it since dplyr imports its functionality when you load the dplyr package.

Here's the simplest way to use it. Remember the `tail()` function. It expects a data frame as input, and the next argument is the number of lines to print. These two commands are identical:

```
tail(ydat, 5)
```

```
# A tibble: 5 x 7
  symbol systematic_name nutrient  rate expression bp                      mf
  <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                   <chr>
1 KRE1   YNL322C         Uracil     0.3       0.28 cell wall organization~ stru~
2 MTL1   YGR023W         Uracil     0.3       0.27 cell wall organization~ mole~
3 KRE9   YJL174W         Uracil     0.3       0.43 cell wall organization~ mole~
4 UTH1   YKR042W         Uracil     0.3       0.19 mitochondrion organiza~ mole~
5 <NA>   YOL111C         Uracil     0.3       0.04 biological process unk~ mole~
```

```
ydat |> tail(5)
```

```
# A tibble: 5 x 7
  symbol systematic_name nutrient  rate expression bp                      mf
  <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                   <chr>
1 KRE1   YNL322C         Uracil     0.3       0.28 cell wall organization~ stru~
2 MTL1   YGR023W         Uracil     0.3       0.27 cell wall organization~ mole~
3 KRE9   YJL174W         Uracil     0.3       0.43 cell wall organization~ mole~
4 UTH1   YKR042W         Uracil     0.3       0.19 mitochondrion organiza~ mole~
5 <NA>   YOL111C         Uracil     0.3       0.04 biological process unk~ mole~
```

Let's use one of the dplyr verbs.

```
filter(ydat, nutrient=="Leucine")
```

```
# A tibble: 33,178 x 7
   symbol systematic_name nutrient  rate expression bp                     mf
   <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                  <chr>
 1 SFB2   YNL049C         Leucine   0.05       0.18 ER to Golgi transport  mole~
 2 <NA>   YNL095C         Leucine   0.05       0.16 biological process un~ mole~
 3 QRI7   YDL104C         Leucine   0.05      -0.3  proteolysis and pepti~ meta~
 4 CFT2   YLR115W         Leucine   0.05      -0.27 mRNA polyadenylylatio~ RNA ~
 5 SSO2   YMR183C         Leucine   0.05      -0.59 vesicle fusion*        t-SN~
 6 PSP2   YML017W         Leucine   0.05      -0.17 biological process un~ mole~
 7 RIB2   YOL066C         Leucine   0.05      -0.02 riboflavin biosynthes~ pseu~
 8 VMA13  YPR036W         Leucine   0.05      -0.11 vacuolar acidification hydr~
 9 EDC3   YEL015W         Leucine   0.05       0.12 deadenylylation-indep~ mole~
10 VPS5   YOR069W         Leucine   0.05      -0.2  protein retention in ~ prot~
# i 33,168 more rows
```

```
ydat |> filter(nutrient=="Leucine")
```

```
# A tibble: 33,178 x 7
   symbol systematic_name nutrient  rate expression bp                     mf
   <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                  <chr>
 1 SFB2   YNL049C         Leucine   0.05       0.18 ER to Golgi transport  mole~
 2 <NA>   YNL095C         Leucine   0.05       0.16 biological process un~ mole~
 3 QRI7   YDL104C         Leucine   0.05      -0.3  proteolysis and pepti~ meta~
 4 CFT2   YLR115W         Leucine   0.05      -0.27 mRNA polyadenylylatio~ RNA ~
 5 SSO2   YMR183C         Leucine   0.05      -0.59 vesicle fusion*        t-SN~
 6 PSP2   YML017W         Leucine   0.05      -0.17 biological process un~ mole~
 7 RIB2   YOL066C         Leucine   0.05      -0.02 riboflavin biosynthes~ pseu~
 8 VMA13  YPR036W         Leucine   0.05      -0.11 vacuolar acidification hydr~
 9 EDC3   YEL015W         Leucine   0.05       0.12 deadenylylation-indep~ mole~
10 VPS5   YOR069W         Leucine   0.05      -0.2  protein retention in ~ prot~
# i 33,168 more rows
```

3.4.2 Nesting versus |>

So what?

Now, think about this for a minute. What if we wanted to get the correlation between the growth rate and expression separately for each limiting nutrient only for genes in the leucine biosynthesis pathway, and return a sorted list of those correlation coeffients rounded to two digits? Mentally we would do something like this:

0. Take the `ydat` dataset
1. *then* `filter()` it for genes in the leucine biosynthesis pathway
2. *then* `group_by()` the limiting nutrient
3. *then* `summarize()` to get the correlation (`cor()`) between rate and expression
4. *then* `mutate()` to round the result of the above calculation to two significant digits
5. *then* `arrange()` by the rounded correlation coefficient above

But in code, it gets ugly. First, take the `ydat` dataset

```
ydat
```

then `filter()` it for genes in the leucine biosynthesis pathway

```
filter(ydat, bp=="leucine biosynthesis")
```

then `group_by()` the limiting nutrient

```
group_by(filter(ydat, bp=="leucine biosynthesis"), nutrient)
```

then `summarize()` to get the correlation (`cor()`) between rate and expression

```
summarize(group_by(filter(ydat, bp == "leucine biosynthesis"), nutrient), r = cor(rate,
    expression))
```

then `mutate()` to round the result of the above calculation to two significant digits

```
mutate(summarize(group_by(filter(ydat, bp == "leucine biosynthesis"), nutrient),
    r = cor(rate, expression)), r = round(r, 2))
```

then `arrange()` by the rounded correlation coefficient above

```
arrange(
  mutate(
    summarize(
      group_by(
        filter(ydat, bp=="leucine biosynthesis"),
      nutrient),
    r=cor(rate, expression)),
  r=round(r, 2)),
r)
```

```
# A tibble: 6 x 2
```

```
  nutrient      r
  <chr>     <dbl>
1 Leucine   -0.58
2 Glucose   -0.04
3 Ammonia    0.16
4 Sulfate    0.33
5 Phosphate  0.44
6 Uracil     0.58
```

Now compare that with the mental process of what you're actually trying to accomplish. The way you would do this without pipes is completely inside-out and backwards from the way you express in words and in thought what you want to do. The pipe operator |> allows you to pass the output data frame from one function to the input data frame to another function.

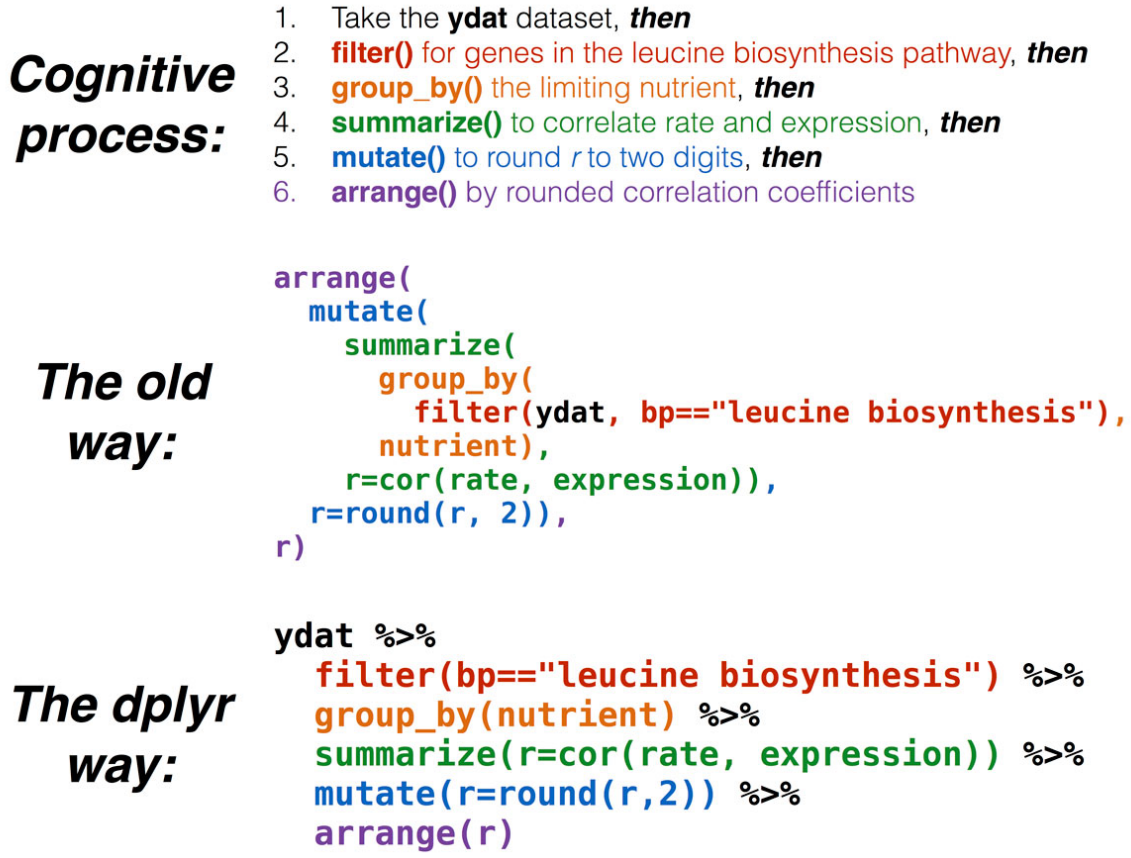


Figure 3.1: Nesting functions versus piping

This is how we would do that in code. It's as simple as replacing the word "then" in words to the symbol |> in code. (There's a keyboard shortcut that I'll use frequently to insert the

|> sequence – you can see what it is by clicking the *Tools* menu in RStudio, then selecting *Keyboard Shortcut Help*. On Mac, it's CMD-SHIFT-M.)

```
ydat |>
  filter(bp=="leucine biosynthesis") |>
  group_by(nutrient) |>
  summarize(r=cor(rate, expression)) |>
  mutate(r=round(r,2)) |>
  arrange(r)
```

```
# A tibble: 6 x 2
  nutrient      r
  <chr>     <dbl>
1 Leucine   -0.58
2 Glucose   -0.04
3 Ammonia    0.16
4 Sulfate    0.33
5 Phosphate  0.44
6 Uracil     0.58
```

3.5 Exercises

Here's a warm-up round. Try the following.

> Exercise 3
>
> Show the limiting nutrient and expression values for the gene ADH2 when the growth rate is restricted to 0.05. *Hint:* 2 pipes: `filter` and `select`.

```
# A tibble: 6 x 2
  nutrient  expression
  <chr>          <dbl>
1 Glucose         6.28
2 Ammonia         0.55
3 Phosphate      -4.6
4 Sulfate        -1.18
5 Leucine         4.15
6 Uracil          0.63
```

Exercise 4

What are the four most highly expressed genes when the growth rate is restricted to 0.05 by restricting glucose? Show only the symbol, expression value, and GO terms. *Hint:* 4 pipes: `filter`, `arrange`, `head`, and `select`.

```
# A tibble: 4 x 4
  symbol expression bp                  mf
  <chr>       <dbl> <chr>               <chr>
1 ADH2         6.28 fermentation*       alcohol dehydrogenase activity
2 HSP26        5.86 response to stress* unfolded protein binding
3 MLS1         5.64 glyoxylate cycle    malate synthase activity
4 HXT5         5.56 hexose transport    glucose transporter activity*
```

Exercise 5

When the growth rate is restricted to 0.05, what is the average expression level across all genes in the "response to stress" biological process, separately for each limiting nutrient? What about genes in the "protein biosynthesis" biological process? *Hint:* 3 pipes: `filter`, `group_by`, `summarize`.

```
# A tibble: 6 x 2
  nutrient  meanexp
  <chr>       <dbl>
1 Ammonia     0.943
2 Glucose     0.743
3 Leucine     0.811
4 Phosphate   0.981
5 Sulfate     0.743
6 Uracil      0.731

# A tibble: 6 x 2
  nutrient  meanexp
  <chr>       <dbl>
1 Ammonia    -1.61
2 Glucose    -0.691
3 Leucine    -0.574
4 Phosphate  -0.750
5 Sulfate    -0.913
6 Uracil     -0.880
```

That was easy, right? How about some tougher ones.

Exercise 6

First, some review. How do we see the number of distinct values of a variable? Use `n_distinct()` within a `summarize()` call.

```
ydat |> summarize(n_distinct(mf))
```

```
# A tibble: 1 x 1
  `n_distinct(mf)`
             <int>
1             1086
```

Exercise 7

Which 10 biological process annotations have the most genes associated with them? What about molecular functions? *Hint:* 4 pipes: `group_by`, `summarize` with `n_distinct`, `arrange`, `head`.

```
# A tibble: 10 x 2
   bp                                                          n
   <chr>                                                   <int>
 1 biological process unknown                                269
 2 protein biosynthesis                                      182
 3 protein amino acid phosphorylation*                        78
 4 protein biosynthesis*                                      73
 5 cell wall organization and biogenesis*                     64
 6 regulation of transcription from RNA polymerase II promoter*    49
 7 nuclear mRNA splicing, via spliceosome                     47
 8 DNA repair*                                                44
 9 ER to Golgi transport*                                     42
10 aerobic respiration*                                       42
```

```
# A tibble: 10 x 2
   mf                                      n
   <chr>                               <int>
 1 molecular function unknown            886
 2 structural constituent of ribosome    185
 3 protein binding                       107
 4 RNA binding                            63
 5 protein binding*                       53
 6 DNA binding*                           44
 7 structural molecule activity           43
```

```
 8 GTPase activity                            40
 9 structural constituent of cytoskeleton     39
10 transcription factor activity              38
```

Exercise 8

How many distinct genes are there where we know what process the gene is involved in but we don't know what it does? *Hint:* 3 pipes; `filter` where `bp!="biological process unknown" & mf=="molecular function unknown"`, and after `select`ing columns of interest, pipe the output to `distinct()`. The answer should be **737**, and here are a few:

```
# A tibble: 737 x 3
   symbol bp                                                                mf
   <chr>  <chr>                                                             <chr>
 1 SFB2   ER to Golgi transport                                             molec~
 2 EDC3   deadenylylation-independent decapping                             molec~
 3 PER1   response to unfolded protein*                                     molec~
 4 PEX25  peroxisome organization and biogenesis*                           molec~
 5 BNI5   cytokinesis*                                                      molec~
 6 CSN12  adaptation to pheromone during conjugation with cellular fusion   molec~
 7 SEC39  secretory pathway                                                 molec~
 8 ABC1   ubiquinone biosynthesis                                           molec~
 9 PRP46  nuclear mRNA splicing, via spliceosome                            molec~
10 MAM3   mitochondrion organization and biogenesis*                       molec~
# i 727 more rows
```

Exercise 9

When the growth rate is restricted to 0.05 by limiting Glucose, which biological processes are the most upregulated? Show a sorted list with the most upregulated BPs on top, displaying the biological process and the average expression of all genes in that process rounded to two digits. *Hint:* 5 pipes: `filter`, `group_by`, `summarize`, `mutate`, `arrange`.

```
# A tibble: 881 x 2
   bp                                          meanexp
   <chr>                                         <dbl>
 1 fermentation*                                  6.28
 2 glyoxylate cycle                               5.28
 3 oxygen and reactive oxygen species metabolism  5.04
 4 fumarate transport*                            5.03
 5 acetyl-CoA biosynthesis*                       4.32
 6 gluconeogenesis                                3.64
```

```
 7 fatty acid beta-oxidation                    3.57
 8 lactate transport                            3.48
 9 carnitine metabolism                         3.3
10 alcohol metabolism*                          3.25
# i 871 more rows
```

Exercise 10

Group the data by limiting nutrient (primarily) then by biological process. Get the average expression for all genes annotated with each process, separately for each limiting nutrient, where the growth rate is restricted to 0.05. Arrange the result to show the most upregulated processes on top. The initial result will look like the result below. Pipe this output to a `View()` statement. What's going on? Why didn't the `arrange()` work? *Hint:* 5 pipes: `filter`, `group_by`, `summarize`, `arrange`, `View`.

```
# A tibble: 5,257 x 3
# Groups:   nutrient [6]
   nutrient  bp                                                meanexp
   <chr>     <chr>                                               <dbl>
 1 Ammonia   allantoate transport                                6.64
 2 Ammonia   amino acid transport*                              6.64
 3 Phosphate glycerophosphodiester transport                    6.64
 4 Glucose   fermentation*                                      6.28
 5 Ammonia   allantoin transport                                5.56
 6 Glucose   glyoxylate cycle                                   5.28
 7 Ammonia   proline catabolism*                                5.14
 8 Ammonia   urea transport                                     5.14
 9 Glucose   oxygen and reactive oxygen species metabolism      5.04
10 Glucose   fumarate transport*                                5.03
# i 5,247 more rows
```

Exercise 11

Let's try to further process that result to get only the top three most upregulated biolgocal processes for each limiting nutrient. Google search "dplyr first result within group." You'll need a `filter(row_number()......)` in there somewhere. *Hint:* 5 pipes: `filter`, `group_by`, `summarize`, `arrange`, `filter(row_number()....`. *Note:* dplyr's pipe syntax used to be `%.%` before it changed to `|>`. So when looking around, you might still see some people use the old syntax. Now if you try to use the old syntax, you'll get a deprecation warning.

```
# A tibble: 18 x 3
```

```
# Groups:   nutrient [6]
   nutrient  bp                                                meanexp
   <chr>     <chr>                                             <dbl>
 1 Ammonia   allantoate transport                               6.64
 2 Ammonia   amino acid transport*                             6.64
 3 Phosphate glycerophosphodiester transport                   6.64
 4 Glucose   fermentation*                                     6.28
 5 Ammonia   allantoin transport                               5.56
 6 Glucose   glyoxylate cycle                                  5.28
 7 Glucose   oxygen and reactive oxygen species metabolism     5.04
 8 Uracil    fumarate transport*                               4.32
 9 Phosphate vacuole fusion, non-autophagic                    4.20
10 Leucine   fermentation*                                     4.15
11 Phosphate regulation of cell redox homeostasis*             4.03
12 Leucine   fumarate transport*                               3.72
13 Leucine   glyoxylate cycle                                  3.65
14 Sulfate   protein ubiquitination                            3.4
15 Sulfate   fumarate transport*                               3.27
16 Uracil    pyridoxine metabolism                             3.11
17 Uracil    asparagine catabolism*                            3.06
18 Sulfate   sulfur amino acid metabolism*                     2.69
```

Exercise 12

There's a slight problem with the examples above. We're getting the average expression of all the biological processes separately by each nutrient. But some of these biological processes only have a single gene in them! If we tried to do the same thing to get the correlation between rate and expression, the calculation would work, but we'd get a warning about a standard deviation being zero. The correlation coefficient value that results is `NA`, i.e., missing. While we're summarizing the correlation between rate and expression, let's also show the number of distinct genes within each grouping.

```
ydat |>
  group_by(nutrient, bp) |>
  summarize(r=cor(rate, expression), ngenes=n_distinct(symbol))
```

```
Warning: There was 1 warning in `summarize()`.
i In argument: `r = cor(rate, expression)`.
i In group 110: `nutrient = "Ammonia"` and `bp = "allantoate transport"`.
Caused by warning in `cor()`:
! the standard deviation is zero
```

```
# A tibble: 5,286 x 4
# Groups:   nutrient [6]
   nutrient bp                                              r ngenes
   <chr>    <chr>                                       <dbl>  <int>
 1 Ammonia  'de novo' IMP biosynthesis*                 0.312      8
 2 Ammonia  'de novo' pyrimidine base biosynthesis     -0.0482     3
 3 Ammonia  'de novo' pyrimidine base biosynthesis*     0.167      4
 4 Ammonia  35S primary transcript processing           0.508     13
 5 Ammonia  35S primary transcript processing*          0.424     30
 6 Ammonia  AMP biosynthesis*                           0.464      1
 7 Ammonia  ATP synthesis coupled proton transport      0.112     15
 8 Ammonia  ATP synthesis coupled proton transport*    -0.541      2
 9 Ammonia  C-terminal protein amino acid methylation   0.813      1
10 Ammonia  D-ribose metabolism                        -0.837      1
# i 5,276 more rows
```

Take the above code and continue to process the result to show only results where the process has at least 5 genes. Add a column corresponding to the absolute value of the correlation coefficient, and show for each nutrient the singular process with the highest correlation between rate and expression, regardless of direction. *Hint:* 4 more pipes: `filter`, `mutate`, `arrange`, and `filter` again with `row_number()==1`. Ignore the warning.

```
# A tibble: 6 x 5
# Groups:   nutrient [6]
  nutrient  bp                                              r ngenes  absr
  <chr>     <chr>                                       <dbl>  <int> <dbl>
1 Glucose   telomerase-independent telomere maintenance -0.95      7  0.95
2 Ammonia   telomerase-independent telomere maintenance -0.91      7  0.91
3 Leucine   telomerase-independent telomere maintenance -0.9       7  0.9
4 Phosphate telomerase-independent telomere maintenance -0.9       7  0.9
5 Uracil    telomerase-independent telomere maintenance -0.81      7  0.81
6 Sulfate   translational elongation*                     0.79      5  0.79
```

4 Tidy Data and Advanced Data Manipulation

Recommended reading prior to class: Sections 1-3 of Wickham, H. "Tidy Data." *Journal of Statistical Software* 59:10 (2014).

Data needed:

- Heart rate data: heartrate2dose.csv
- *Tidy* yeast data: brauer2007_tidy.csv
- *Original* (untidy) yeast data: brauer2007_messy.csv
- Yeast systematic names to GO terms: brauer2007_sysname2go.csv

4.1 Tidy data

So far we've dealt exclusively with tidy data – data that's easy to work with, manipulate, and visualize. That's because our dataset has two key properties:

1. Each *column* is a *variable*.
2. Each *row* is an *observation*.

You can read a lot more about tidy data in this paper. Let's load some untidy data and see if we can see the difference. This is some made-up data for five different patients (Jon, Ann, Bill, Kate, and Joe) given three different drugs (A, B, and C), at two doses (10 and 20), and measuring their heart rate. Download the heartrate2dose.csv file. Load **readr** and **dplyr**, and import and display the data.

```
library(readr)
library(dplyr)
hr <- read_csv("data/heartrate2dose.csv")
hr
```

```
# A tibble: 5 x 7
  name   a_10  a_20  b_10  b_20  c_10  c_20
  <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 jon      60    55    65    60    70    70
2 ann      65    60    70    65    75    75
```

```
3 bill     70    65    75    70    80    80
4 kate     75    70    80    75    85    85
5 joe      80    75    85    80    90    90
```

Notice how with the yeast data each variable (symbol, nutrient, rate, expression, etc.) were each in their own column. In this heart rate data, we have four variables: name, drug, dose, and heart rate. *Name* is in a column, but *drug* is in the header row. Furthermore the drug and *dose* are tied together in the same column, and the *heart rate* is scattered around the entire table. If we wanted to do things like `filter` the dataset where `drug=="a"` or `dose==20` or `heartrate>=80` we couldn't do it because these variables aren't in columns.

4.2 The tidyr package

The **tidyr** package helps with this. There are several functions in the tidyr package but the ones we're going to use are `separate()` and `gather()`. The `gather()` function takes multiple columns, and gathers them into key-value pairs: it makes "wide" data longer. The `separate()` function separates one column into multiple columns. So, what we need to do is *gather* all the drug/dose data into a column with their corresponding heart rate, and then *separate* that column into two separate columns for the drug and dose.

Before we get started, load the **tidyr** package, and look at the help pages for `?gather` and `?separate`. Notice how each of these functions takes a data frame as input and returns a data frame as output. Thus, we can pipe from one function to the next.

```
library(tidyr)
```

4.2.1 `gather()`

The help for `?gather` tells us that we first pass in a data frame (or omit the first argument, and pipe in the data with `|>`). The next two arguments are the names of the key and value columns to create, and all the relevant arguments that come after that are the columns we want to *gather* together. Here's one way to do it.

```
hr |> gather(key=drugdose, value=hr, a_10, a_20, b_10, b_20, c_10, c_20)
```

```
# A tibble: 30 x 3
   name  drugdose    hr
   <chr> <chr>    <dbl>
 1 jon   a_10        60
```

```
 2 ann   a_10        65
 3 bill  a_10        70
 4 kate  a_10        75
 5 joe   a_10        80
 6 jon   a_20        55
 7 ann   a_20        60
 8 bill  a_20        65
 9 kate  a_20        70
10 joe   a_20        75
# i 20 more rows
```

But that gets cumbersome to type all those names. What if we had 100 drugs and 3 doses of each? There are two other ways of specifying which columns to gather. The help for `?gather` tells you how to do this:

> ... Specification of columns to gather. Use bare variable names. Select all variables between x and z with x:z, exclude y with -y. For more options, see the `select` documentation.

So, we could accomplish the same thing by doing this:

```
hr |> gather(key=drugdose, value=hr, a_10:c_20)
```

```
# A tibble: 30 x 3
   name  drugdose    hr
   <chr> <chr>    <dbl>
 1 jon   a_10        60
 2 ann   a_10        65
 3 bill  a_10        70
 4 kate  a_10        75
 5 joe   a_10        80
 6 jon   a_20        55
 7 ann   a_20        60
 8 bill  a_20        65
 9 kate  a_20        70
10 joe   a_20        75
# i 20 more rows
```

But what if we didn't know the drug names or doses, but we *did* know that the only other column in there that we *don't* want to gather is `name`?

```
hr |> gather(key=drugdose, value=hr, -name)
```

```
# A tibble: 30 x 3
   name  drugdose    hr
   <chr> <chr>    <dbl>
 1 jon   a_10        60
 2 ann   a_10        65
 3 bill  a_10        70
 4 kate  a_10        75
 5 joe   a_10        80
 6 jon   a_20        55
 7 ann   a_20        60
 8 bill  a_20        65
 9 kate  a_20        70
10 joe   a_20        75
# i 20 more rows
```

4.2.2 `separate()`

Finally, look at the help for `?separate`. We can pipe in data and omit the first argument. The second argument is the column to separate; the `into` argument is a *character vector* of the new column names, and the `sep` argument is a character used to separate columns, or a number indicating the position to split at.

> ***Side note, and 60-second lesson on vectors***: We can create arbitrary-length *vectors*, which are simply variables that contain an arbitrary number of values. To create a numeric vector, try this: `c(5, 42, 22908)`. That creates a three element vector. Try `c("cat", "dog")`.

```
hr |>
  gather(key=drugdose, value=hr, -name) |>
  separate(drugdose, into=c("drug", "dose"), sep="_")
```

```
# A tibble: 30 x 4
   name  drug  dose     hr
   <chr> <chr> <chr> <dbl>
 1 jon   a     10       60
 2 ann   a     10       65
 3 bill  a     10       70
 4 kate  a     10       75
 5 joe   a     10       80
 6 jon   a     20       55
 7 ann   a     20       60
```

```
 8 bill  a     20       65
 9 kate  a     20       70
10 joe   a     20       75
# i 20 more rows
```

4.2.3 |> it all together

Let's put it all together with `gather |> separate |> filter |> group_by |> summarize`.

If we create a new data frame that's a tidy version of hr, we can do those kinds of manipulations we talked about before:

```
# Create a new data.frame
hrtidy <- hr |>
  gather(key=drugdose, value=hr, -name) |>
  separate(drugdose, into=c("drug", "dose"), sep="_")

# Optionally, view it
# View(hrtidy)

# filter
hrtidy |> filter(drug=="a")
```

```
# A tibble: 10 x 4
   name  drug  dose     hr
   <chr> <chr> <chr> <dbl>
 1 jon   a     10       60
 2 ann   a     10       65
 3 bill  a     10       70
 4 kate  a     10       75
 5 joe   a     10       80
 6 jon   a     20       55
 7 ann   a     20       60
 8 bill  a     20       65
 9 kate  a     20       70
10 joe   a     20       75
```

```
hrtidy |> filter(dose==20)
```

```
# A tibble: 15 x 4
```

```
   name  drug  dose     hr
   <chr> <chr> <chr> <dbl>
 1 jon   a     20       55
 2 ann   a     20       60
 3 bill  a     20       65
 4 kate  a     20       70
 5 joe   a     20       75
 6 jon   b     20       60
 7 ann   b     20       65
 8 bill  b     20       70
 9 kate  b     20       75
10 joe   b     20       80
11 jon   c     20       70
12 ann   c     20       75
13 bill  c     20       80
14 kate  c     20       85
15 joe   c     20       90
```

```
hrtidy |> filter(hr>=80)
```

```
# A tibble: 10 x 4
   name  drug  dose     hr
   <chr> <chr> <chr> <dbl>
 1 joe   a     10       80
 2 kate  b     10       80
 3 joe   b     10       85
 4 joe   b     20       80
 5 bill  c     10       80
 6 kate  c     10       85
 7 joe   c     10       90
 8 bill  c     20       80
 9 kate  c     20       85
10 joe   c     20       90
```

```
# analyze
hrtidy |>
  filter(name!="joe") |>
  group_by(drug, dose) |>
  summarize(meanhr=mean(hr))
```

```
# A tibble: 6 x 3
# Groups:   drug [3]
  drug  dose  meanhr
  <chr> <chr>  <dbl>
1 a     10      67.5
2 a     20      62.5
3 b     10      72.5
4 b     20      67.5
5 c     10      77.5
6 c     20      77.5
```

4.3 Tidy the yeast data

Now, let's take a look at the yeast data again. The data we've been working with up to this point was already cleaned up to a good degree. All of our variables (symbol, nutrient, rate, expression, GO terms, etc.) were each in their own column. Make sure you have the necessary libraries loaded, and read in the tidy data once more into an object called `ydat`.

```
# Load libraries
library(readr)
library(dplyr)
library(tidyr)

# Import data
ydat <- read_csv("data/brauer2007_tidy.csv")

# Optionally, View
# View(ydat)

# Or just display to the screen
ydat
```

```
# A tibble: 198,430 x 7
   symbol systematic_name nutrient  rate expression bp                     mf
   <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                  <chr>
 1 SFB2   YNL049C         Glucose   0.05      -0.24 ER to Golgi transport  mole~
 2 <NA>   YNL095C         Glucose   0.05       0.28 biological process un~ mole~
 3 QRI7   YDL104C         Glucose   0.05      -0.02 proteolysis and pepti~ meta~
 4 CFT2   YLR115W         Glucose   0.05      -0.33 mRNA polyadenylylatio~ RNA ~
 5 SSO2   YMR183C         Glucose   0.05       0.05 vesicle fusion*        t-SN~
 6 PSP2   YML017W         Glucose   0.05      -0.69 biological process un~ mole~
```

```
 7 RIB2   YOL066C         Glucose   0.05      -0.55 riboflavin biosynthes~ pseu~
 8 VMA13  YPR036W         Glucose   0.05      -0.75 vacuolar acidification hydr~
 9 EDC3   YEL015W         Glucose   0.05      -0.24 deadenylylation-indep~ mole~
10 VPS5   YOR069W         Glucose   0.05      -0.16 protein retention in ~ prot~
# i 198,420 more rows
```

But let's take a look to see what this data originally looked like.

```
yorig <- read_csv("data/brauer2007_messy.csv")
# View(yorig)
yorig
```

```
# A tibble: 5,536 x 40
   GID       YORF  NAME  GWEIGHT G0.05  G0.1 G0.15  G0.2 G0.25  G0.3 N0.05  N0.1
   <chr>     <chr> <chr>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1 GENE1331X A_06~ SFB2~       1 -0.24 -0.13 -0.21 -0.15 -0.05 -0.05  0.2   0.24
 2 GENE4924X A_06~ NA::~       1  0.28  0.13 -0.4  -0.48 -0.11  0.17  0.31  0
 3 GENE4690X A_06~ QRI7~       1 -0.02 -0.27 -0.27 -0.02  0.24  0.25  0.23  0.06
 4 GENE1177X A_06~ CFT2~       1 -0.33 -0.41 -0.24 -0.03 -0.03  0     0.2  -0.25
 5 GENE511X  A_06~ SSO2~       1  0.05  0.02  0.4   0.34 -0.13 -0.14 -0.35 -0.09
 6 GENE2133X A_06~ PSP2~       1 -0.69 -0.03  0.23  0.2   0    -0.27  0.17 -0.4
 7 GENE1002X A_06~ RIB2~       1 -0.55 -0.3  -0.12 -0.03 -0.16 -0.11  0.04  0
 8 GENE5478X A_06~ VMA1~       1 -0.75 -0.12 -0.07  0.02 -0.32 -0.41  0.11 -0.16
 9 GENE2065X A_06~ EDC3~       1 -0.24 -0.22  0.14  0.06  0    -0.13  0.3   0.07
10 GENE2440X A_06~ VPS5~       1 -0.16 -0.38  0.05  0.14 -0.04 -0.01  0.39  0.2
# i 5,526 more rows
# i 28 more variables: N0.15 <dbl>, N0.2 <dbl>, N0.25 <dbl>, N0.3 <dbl>,
#   P0.05 <dbl>, P0.1 <dbl>, P0.15 <dbl>, P0.2 <dbl>, P0.25 <dbl>, P0.3 <dbl>,
#   S0.05 <dbl>, S0.1 <dbl>, S0.15 <dbl>, S0.2 <dbl>, S0.25 <dbl>, S0.3 <dbl>,
#   L0.05 <dbl>, L0.1 <dbl>, L0.15 <dbl>, L0.2 <dbl>, L0.25 <dbl>, L0.3 <dbl>,
#   U0.05 <dbl>, U0.1 <dbl>, U0.15 <dbl>, U0.2 <dbl>, U0.25 <dbl>, U0.3 <dbl>
```

There are several issues here.

1. **Multiple variables are stored in one column.** The `NAME` column contains lots of information, split up by `::`'s.
2. **Nutrient and rate variables are stuck in column headers.** That is, the column names contain the values of two variables: nutrient (G, N, P, S, L, U) and growth rate (0.05-0.3). Remember, with tidy data, **each *column* is a *variable* and each *row* is an *observation*.** Here, we have not one observation per row, but 36 (6 nutrients $\times$ 6 rates)! There's no way we could filter this data by a certain nutrient, or try to calculate statistics between rate and expression.

3. **Expression values are scattered throughout the table.** Related to the problem above, and just like our heart rate example, `expression` isn't a single-column variable as in the cleaned tidy data, but it's scattered around these 36 columns.
4. **Other important information is in a separate table.** We're missing all the gene ontology information we had in the tidy data (no information about biological process (`bp`) or molecular function (`mf`)).

Let's tackle these issues one at a time, all on a `|>` pipeline.

4.3.1 `separate()` the `NAME`

Let's `separate()` the `NAME` column `into` multiple different variables. The first row looks like this:

```
SFB2::YNL049C::1082129
```

That is, it looks like we've got the gene symbol, the systematic name, and some other number (that isn't discussed in the paper). Let's `separate()`!

```
yorig |>
  separate(NAME, into=c("symbol", "systematic_name", "somenumber"), sep="::")
```

```
# A tibble: 5,536 x 42
   GID   YORF  symbol systematic_name somenumber GWEIGHT G0.05  G0.1 G0.15  G0.2
   <chr> <chr> <chr>  <chr>           <chr>        <dbl> <dbl> <dbl> <dbl> <dbl>
 1 GENE~ A_06~ SFB2   YNL049C         1082129          1 -0.24 -0.13 -0.21 -0.15
 2 GENE~ A_06~ NA     YNL095C         1086222          1  0.28  0.13 -0.4  -0.48
 3 GENE~ A_06~ QRI7   YDL104C         1085955          1 -0.02 -0.27 -0.27 -0.02
 4 GENE~ A_06~ CFT2   YLR115W         1081958          1 -0.33 -0.41 -0.24 -0.03
 5 GENE~ A_06~ SSO2   YMR183C         1081214          1  0.05  0.02  0.4   0.34
 6 GENE~ A_06~ PSP2   YML017W         1083036          1 -0.69 -0.03  0.23  0.2
 7 GENE~ A_06~ RIB2   YOL066C         1081766          1 -0.55 -0.3  -0.12 -0.03
 8 GENE~ A_06~ VMA13  YPR036W         1086860          1 -0.75 -0.12 -0.07  0.02
 9 GENE~ A_06~ EDC3   YEL015W         1082963          1 -0.24 -0.22  0.14  0.06
10 GENE~ A_06~ VPS5   YOR069W         1083389          1 -0.16 -0.38  0.05  0.14
# i 5,526 more rows
# i 32 more variables: G0.25 <dbl>, G0.3 <dbl>, N0.05 <dbl>, N0.1 <dbl>,
#   N0.15 <dbl>, N0.2 <dbl>, N0.25 <dbl>, N0.3 <dbl>, P0.05 <dbl>, P0.1 <dbl>,
#   P0.15 <dbl>, P0.2 <dbl>, P0.25 <dbl>, P0.3 <dbl>, S0.05 <dbl>, S0.1 <dbl>,
#   S0.15 <dbl>, S0.2 <dbl>, S0.25 <dbl>, S0.3 <dbl>, L0.05 <dbl>, L0.1 <dbl>,
#   L0.15 <dbl>, L0.2 <dbl>, L0.25 <dbl>, L0.3 <dbl>, U0.05 <dbl>, U0.1 <dbl>,
#   U0.15 <dbl>, U0.2 <dbl>, U0.25 <dbl>, U0.3 <dbl>
```

Now, let's `select()` out the stuff we don't want.

```
yorig |>
  separate(NAME, into=c("symbol", "systematic_name", "somenumber"), sep="::") |>
  select(-GID, -YORF, -somenumber, -GWEIGHT)
```

```
# A tibble: 5,536 x 38
   symbol systematic_name G0.05  G0.1 G0.15  G0.2 G0.25  G0.3 N0.05  N0.1 N0.15
   <chr>  <chr>           <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1 SFB2   YNL049C         -0.24 -0.13 -0.21 -0.15 -0.05 -0.05  0.2   0.24 -0.2
 2 NA     YNL095C          0.28  0.13 -0.4  -0.48 -0.11  0.17  0.31  0    -0.63
 3 QRI7   YDL104C         -0.02 -0.27 -0.27 -0.02  0.24  0.25  0.23  0.06 -0.66
 4 CFT2   YLR115W         -0.33 -0.41 -0.24 -0.03 -0.03  0     0.2  -0.25 -0.49
 5 SSO2   YMR183C          0.05  0.02  0.4   0.34 -0.13 -0.14 -0.35 -0.09 -0.08
 6 PSP2   YML017W         -0.69 -0.03  0.23  0.2   0    -0.27  0.17 -0.4  -0.54
 7 RIB2   YOL066C         -0.55 -0.3  -0.12 -0.03 -0.16 -0.11  0.04  0    -0.63
 8 VMA13  YPR036W         -0.75 -0.12 -0.07  0.02 -0.32 -0.41  0.11 -0.16 -0.26
 9 EDC3   YEL015W         -0.24 -0.22  0.14  0.06  0    -0.13  0.3   0.07 -0.3
10 VPS5   YOR069W         -0.16 -0.38  0.05  0.14 -0.04 -0.01  0.39  0.2   0.27
# i 5,526 more rows
# i 27 more variables: N0.2 <dbl>, N0.25 <dbl>, N0.3 <dbl>, P0.05 <dbl>,
#   P0.1 <dbl>, P0.15 <dbl>, P0.2 <dbl>, P0.25 <dbl>, P0.3 <dbl>, S0.05 <dbl>,
#   S0.1 <dbl>, S0.15 <dbl>, S0.2 <dbl>, S0.25 <dbl>, S0.3 <dbl>, L0.05 <dbl>,
#   L0.1 <dbl>, L0.15 <dbl>, L0.2 <dbl>, L0.25 <dbl>, L0.3 <dbl>, U0.05 <dbl>,
#   U0.1 <dbl>, U0.15 <dbl>, U0.2 <dbl>, U0.25 <dbl>, U0.3 <dbl>
```

4.3.2 `gather()` the data

Let's gather the data from wide to long format so we get nutrient/rate (key) and expression (value) in their own columns.

```
yorig |>
  separate(NAME, into=c("symbol", "systematic_name", "somenumber"), sep="::") |>
  select(-GID, -YORF, -somenumber, -GWEIGHT) |>
  gather(key=nutrientrate, value=expression, G0.05:U0.3)
```

```
# A tibble: 199,296 x 4
   symbol systematic_name nutrientrate expression
   <chr>  <chr>           <chr>             <dbl>
 1 SFB2   YNL049C         G0.05             -0.24
```

```
 2 NA     YNL095C         G0.05              0.28
 3 QRI7   YDL104C         G0.05             -0.02
 4 CFT2   YLR115W         G0.05             -0.33
 5 SSO2   YMR183C         G0.05              0.05
 6 PSP2   YML017W         G0.05             -0.69
 7 RIB2   YOL066C         G0.05             -0.55
 8 VMA13  YPR036W         G0.05             -0.75
 9 EDC3   YEL015W         G0.05             -0.24
10 VPS5   YOR069W         G0.05             -0.16
# i 199,286 more rows
```

And while we're at it, let's `separate()` that newly created key column. Take a look at the help for `?separate` again. The `sep` argument could be a delimiter or a number position to split at. Let's split after the first character. While we're at it, let's hold onto this intermediate data frame before we add gene ontology information. Call it `ynogo`.

```
ynogo <- yorig |>
  separate(NAME, into=c("symbol", "systematic_name", "somenumber"), sep="::") |>
  select(-GID, -YORF, -somenumber, -GWEIGHT) |>
  gather(key=nutrientrate, value=expression, G0.05:U0.3) |>
  separate(nutrientrate, into=c("nutrient", "rate"), sep=1)
```

4.3.3 `inner_join()` to GO

It's rare that a data analysis involves only a single table of data. You normally have many tables that contribute to an analysis, and you need flexible tools to combine them. The **dplyr** package has several tools that let you work with multiple tables at once. Do a Google image search for "SQL Joins", and look at RStudio's Data Wrangling Cheat Sheet to learn more.

First, let's import the dataset that links the systematic name to gene ontology information. It's the brauer2007_sysname2go.csv file. Let's call the imported data frame `sn2go`.

```
# Import the data
sn2go <- read_csv("data/brauer2007_sysname2go.csv")

# Take a look
# View(sn2go)
head(sn2go)
```

```
# A tibble: 6 x 3
  systematic_name bp                                mf
```

```
  <chr>           <chr>                         <chr>
1 YNL049C         ER to Golgi transport         molecular function unknown
2 YNL095C         biological process unknown    molecular function unknown
3 YDL104C         proteolysis and peptidolysis metalloendopeptidase activity
4 YLR115W         mRNA polyadenylylation*       RNA binding
5 YMR183C         vesicle fusion*               t-SNARE activity
6 YML017W         biological process unknown    molecular function unknown
```

Now, look up some help for `?inner_join`. Inner join will return a table with all rows from the first table where there are matching rows in the second table, and returns all columns from both tables. Let's give this a try.

```
yjoined <- inner_join(ynogo, sn2go, by="systematic_name")
# View(yjoined)
yjoined
```

```
# A tibble: 199,296 x 7
   symbol systematic_name nutrient rate  expression bp                    mf
   <chr>  <chr>           <chr>    <chr>      <dbl> <chr>                 <chr>
 1 SFB2   YNL049C         G        0.05       -0.24 ER to Golgi transport  mole~
 2 NA     YNL095C         G        0.05        0.28 biological process un~ mole~
 3 QRI7   YDL104C         G        0.05       -0.02 proteolysis and pepti~ meta~
 4 CFT2   YLR115W         G        0.05       -0.33 mRNA polyadenylylatio~ RNA ~
 5 SSO2   YMR183C         G        0.05        0.05 vesicle fusion*        t-SN~
 6 PSP2   YML017W         G        0.05       -0.69 biological process un~ mole~
 7 RIB2   YOL066C         G        0.05       -0.55 riboflavin biosynthes~ pseu~
 8 VMA13  YPR036W         G        0.05       -0.75 vacuolar acidification hydr~
 9 EDC3   YEL015W         G        0.05       -0.24 deadenylylation-indep~ mole~
10 VPS5   YOR069W         G        0.05       -0.16 protein retention in ~ prot~
# i 199,286 more rows
```

```
# The glimpse function makes it possible to see a little bit of everything in your data.
glimpse(yjoined)
```

```
Rows: 199,296
Columns: 7
$ symbol          <chr> "SFB2", "NA", "QRI7", "CFT2", "SSO2", "PSP2", "RIB2", ~
$ systematic_name <chr> "YNL049C", "YNL095C", "YDL104C", "YLR115W", "YMR183C",~
$ nutrient        <chr> "G", "G", "G", "G", "G", "G", "G", "G", "G", "G", "G",~
$ rate            <chr> "0.05", "0.05", "0.05", "0.05", "0.05", "0.05", "0.05"~
```

```
$ expression      <dbl> -0.24, 0.28, -0.02, -0.33, 0.05, -0.69, -0.55, -0.75, ~
$ bp              <chr> "ER to Golgi transport", "biological process unknown",~
$ mf              <chr> "molecular function unknown", "molecular function unkn~
```

There are many different kinds of two-table verbs/joins in dplyr. In this example, every systematic name in `ynogo` had a corresponding entry in `sn2go`, but if this weren't the case, those un-annotated genes would have been removed entirely by the `inner_join`. A `left_join` would have returned all the rows in `ynogo`, but would have filled in `bp` and `mf` with missing values (`NA`) when there was no corresponding entry. See also: `right_join`, `semi_join`, and `anti_join`.

4.3.4 Finishing touches

We're almost there but we have an obvious discrepancy in the number of rows between `yjoined` and `ydat`.

```
nrow(yjoined)
```

```
[1] 199296
```

```
nrow(ydat)
```

```
[1] 198430
```

Before we can figure out what rows are different, we need to make sure all of the columns are the same class and do something more miscellaneous cleanup.

In particular:

1. Convert rate to a numeric column
2. Make sure `NA` values are coded properly
3. Create (and merge) values to convert "G" to "Glucose", "L" to "Leucine", etc.
4. Rename and reorder columns

The code below implements those operations on `yjoined`.

```
nutrientlookup <-
  tibble(nutrient = c("G", "L", "N", "P", "S", "U"), nutrientname = c("Glucose", "Leucine"

yjoined <-
```

```
  yjoined |>
  mutate(rate = as.numeric(rate)) |>
  mutate(symbol = ifelse(symbol == "NA", NA, symbol)) |>
  left_join(nutrientlookup) |>
  select(-nutrient) |>
  select(symbol:systematic_name, nutrient = nutrientname, rate:mf)
```

Now we can determine what rows are different between `yjoined` and `ydat` using `anti_join`, which will return all of the rows that *do not* match.

```
anti_join(yjoined, ydat)
```

```
# A tibble: 866 x 7
   symbol systematic_name nutrient  rate expression bp                    mf
   <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                 <chr>
 1 <NA>   YLL030C         Glucose   0.05         NA <NA>                  <NA>
 2 <NA>   YOR050C         Glucose   0.05         NA <NA>                  <NA>
 3 <NA>   YPR039W         Glucose   0.05         NA <NA>                  <NA>
 4 <NA>   YOL013W-B       Glucose   0.05         NA <NA>                  <NA>
 5 HXT12  YIL170W         Glucose   0.05         NA biological process un~ mole~
 6 <NA>   YPR013C         Glucose   0.05         NA biological process un~ mole~
 7 <NA>   YOR314W         Glucose   0.05         NA <NA>                  <NA>
 8 <NA>   YJL064W         Glucose   0.05         NA <NA>                  <NA>
 9 <NA>   YPR136C         Glucose   0.05         NA <NA>                  <NA>
10 <NA>   YDR015C         Glucose   0.05         NA <NA>                  <NA>
# i 856 more rows
```

Hmmmm … so `yjoined` has some rows that have missing (`NA`) expression values. Let's try removing those and then comparing the data frame contents one more time.

```
yjoined <-
  yjoined |>
  filter(!is.na(expression))

nrow(yjoined)
```

```
[1] 198430
```

```
nrow(ydat)
```

```
[1] 198430
```

```
all.equal(ydat, yjoined)
```

```
[1] "Attributes: < Names: 1 string mismatch >"
[2] "Attributes: < Length mismatch: comparison on first 2 components >"
[3] "Attributes: < Component \"class\": Lengths (4, 3) differ (string compare on first 3) >"
[4] "Attributes: < Component \"class\": 3 string mismatches >"
[5] "Attributes: < Component 2: target is externalptr, current is numeric >"
```

Looks like that did it!

5 Data Visualization with ggplot2

This section will cover fundamental concepts for creating effective data visualization and will introduce tools and techniques for visualizing large, high-dimensional data using R. We will review fundamental concepts for visually displaying quantitative information, such as using series of small multiples, avoiding "chart-junk," and maximizing the data-ink ratio. We will cover the grammar of graphics (geoms, aesthetics, stats, and faceting), and using the ggplot2 package to create plots layer-by-layer.

This chapter assumes a basic familiarity with R (Chapter 1), data frames (Chapter 2), and manipulating data with dplyr and `|>` (Chapter 3).

5.1 Review

5.1.1 Gapminder data

We're going to work with a different dataset for this section. It's a cleaned-up excerpt from the Gapminder data. Download the **gapminder.csv** data by clicking here or using the link above.

Let's read in the data to an object called `gm` and take a look with `View`. Remember, we need to load both the dplyr and readr packages for efficiently reading in and displaying this data.

```
# Load packages
library(readr)
library(dplyr)

# Download the data locally and read the file
gm <- read_csv(file="data/gapminder.csv")

# Show the first few lines of the data
gm
```

```
# A tibble: 1,704 x 6
   country     continent  year lifeExp      pop gdpPercap
   <chr>       <chr>     <dbl>   <dbl>    <dbl>     <dbl>
```

```
 1 Afghanistan Asia       1952    28.8  8425333      779.
 2 Afghanistan Asia       1957    30.3  9240934      821.
 3 Afghanistan Asia       1962    32.0 10267083      853.
 4 Afghanistan Asia       1967    34.0 11537966      836.
 5 Afghanistan Asia       1972    36.1 13079460      740.
 6 Afghanistan Asia       1977    38.4 14880372      786.
 7 Afghanistan Asia       1982    39.9 12881816      978.
 8 Afghanistan Asia       1987    40.8 13867957      852.
 9 Afghanistan Asia       1992    41.7 16317921      649.
10 Afghanistan Asia       1997    41.8 22227415      635.
# i 1,694 more rows
```

```
# Optionally bring up data in a viewer window.
# View(gm)
```

This particular excerpt has 1704 observations on six variables:

- `country` a categorical variable 142 levels
- `continent`, a categorical variable with 5 levels
- `year`: going from 1952 to 2007 in increments of 5 years
- `pop`: population
- `gdpPercap`: GDP per capita
- `lifeExp`: life expectancy

5.1.2 dplyr review

The dplyr package gives you a handful of useful **verbs** for managing data. On their own they don't do anything that base R can't do. Here are some of the *single-table* verbs we'll be working with in this chapter (single-table meaning that they only work on a single table – contrast that to *two-table* verbs used for joining data together). They all take a `data.frame` or `tbl` as their input for the first argument, and they all return a `data.frame` or `tbl` as output.

1. `filter()`: filters *rows* of the data where some condition is true
2. `select()`: selects out particular *columns* of interest
3. `mutate()`: adds new columns or changes values of existing columns
4. `arrange()`: arranges a data frame by the value of a column
5. `summarize()`: summarizes multiple values to a single value, most useful when combined with…
6. `group_by()`: groups a data frame by one or more variable. Most data operations are useful done on groups defined by variables in the the dataset. The `group_by` function takes an existing data frame and converts it into a grouped data frame where `summarize()` operations are performed *by group*.

Additionally, the |> operator allows you to "chain" operations together. Rather than nesting functions inside out, the |> operator allows you to write operations left-to-right, top-to-bottom. Let's say we wanted to get the average life expectancy and GDP (not GDP per capita) for Asian countries for each year.

Cognitive process:

1. Take the **gm** data, **then**
2. **Mutate** it to add "gdp" variable, **then**
3. **Filter** where continent=="Asia", **then**
4. **Group by** year, **then**
5. **Summarize** to get mean life exp & GDP

The old way:

```
summarize(
  group_by(
    filter(
      mutate(gm, gdp=gdpPercap*pop),
    continent=="Asia"),
  year),
mean(lifeExp), mean(gdp))
```

The dplyr way:

```
gm %>%
  mutate(gdp=gdpPercap*pop) %>%
  filter(continent=="Asia") %>%
  group_by(year) %>%
  summarize(mean(lifeExp), mean(gdp))
```

The |> would allow us to do this:

```
gm |>
  mutate(gdp=gdpPercap*pop) |>
  filter(continent=="Asia") |>
  group_by(year) |>
  summarize(mean(lifeExp), mean(gdp))
```

```
# A tibble: 12 x 3
    year `mean(lifeExp)`   `mean(gdp)`
   <dbl>           <dbl>         <dbl>
 1  1952            46.3  34095762661.
```

```
 2  1957            49.3  47267432088.
 3  1962            51.6  60136869012.
 4  1967            54.7  84648519224.
 5  1972            57.3 124385747313.
 6  1977            59.6 159802590186.
 7  1982            62.6 194429049919.
 8  1987            64.9 241784763369.
 9  1992            66.5 307100497486.
10  1997            68.0 387597655323.
11  2002            69.2 458042336179.
12  2007            70.7 627513635079.
```

Instead of this:

```
summarize(
  group_by(
    filter(
      mutate(gm, gdp=gdpPercap*pop),
    continent=="Asia"),
  year),
mean(lifeExp), mean(gdp))
```

5.2 About ggplot2

ggplot2 is a widely used R package that extends R's visualization capabilities. It takes the hassle out of things like creating legends, mapping other variables to scales like color, or faceting plots into small multiples. We'll learn about what all these things mean shortly.

Where does the "gg" in ggplot2 come from? The **ggplot2** package provides an R implementation of Leland Wilkinson's *Grammar of Graphics* (1999). The *Grammar of Graphics* allows you to think beyond the garden variety plot types (e.g. scatterplot, barplot) and the consider the components that make up a plot or graphic, such as how data are represented on the plot (as lines, points, etc.), how variables are mapped to coordinates or plotting shape or color, what transformation or statistical summary is required, and so on.

Specifically, **ggplot2** allows you to build a plot layer-by-layer by specifying:

- a **geom**, which specifies how the data are represented on the plot (points, lines, bars, etc.),
- **aesthetics** that map variables in the data to axes on the plot or to plotting size, shape, color, etc.,
- a **stat**, a statistical transformation or summary of the data applied prior to plotting,

- **facets**, which we've already seen above, that allow the data to be divided into chunks on the basis of other categorical or continuous variables and the same plot drawn for each chunk.

First, a note about `qplot()`. The `qplot()` function is a quick and dirty way of making ggplot2 plots. You might see it if you look for help with ggplot2, and it's even covered extensively in the ggplot2 book. And if you're used to making plots with built-in base graphics, the `qplot()` function will probably feel more familiar. But the sooner you abandon the `qplot()` syntax the sooner you'll start to really understand ggplot2's approach to building up plots layer by layer. So we're not going to use it at all in this class.

5.3 Plotting bivariate data: continuous Y by continuous X

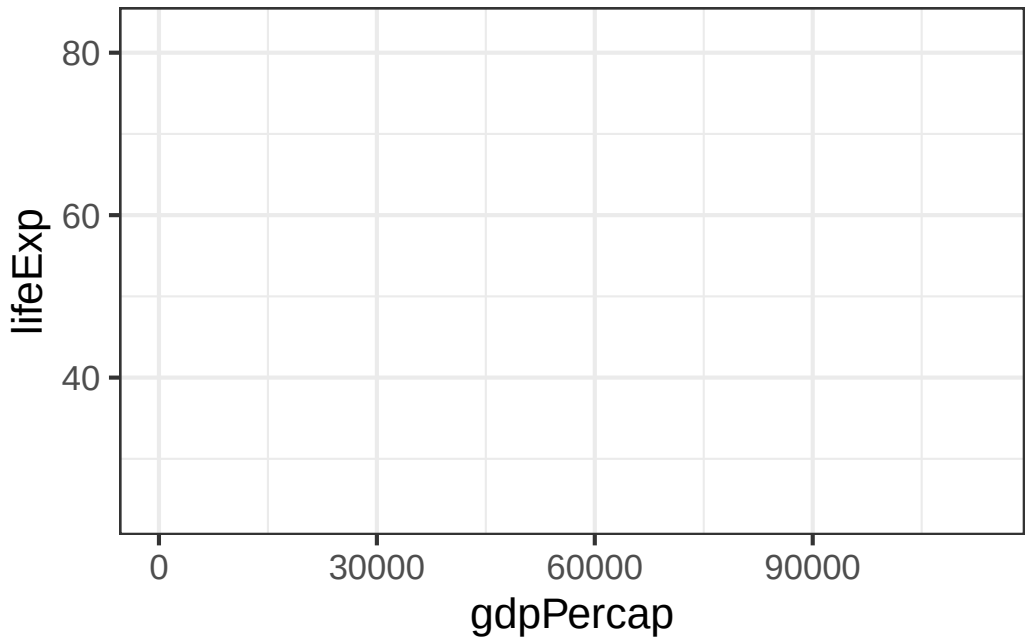
The `ggplot` function has two required arguments: the *data* used for creating the plot, and an *aesthetic* mapping to describe how variables in said data are mapped to things we can see on the plot.

First let's load the package:

```
library(ggplot2)
```

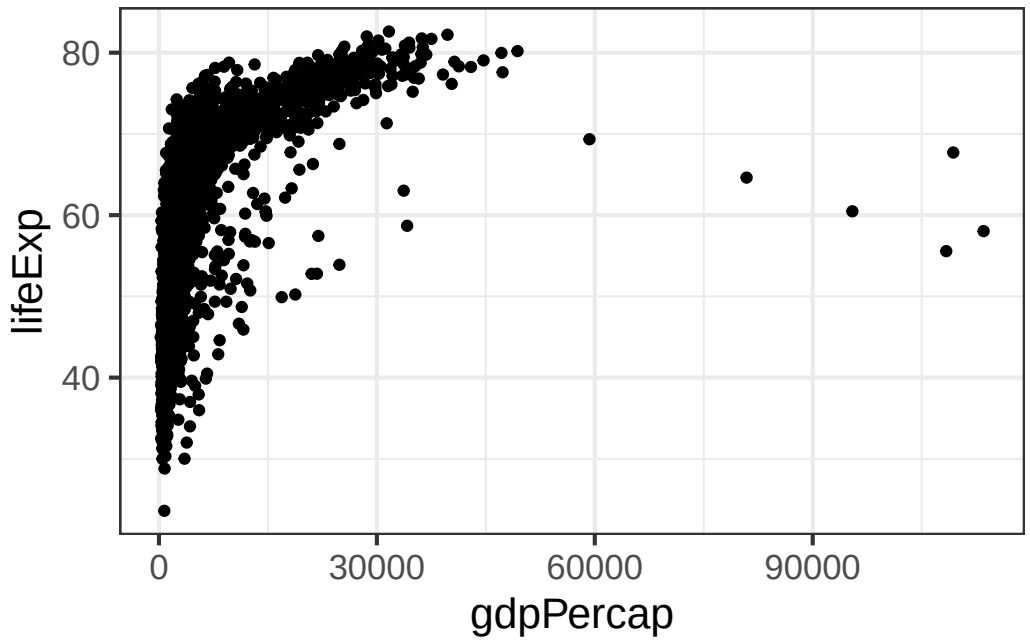
Now, let's lay out the plot. If we want to plot a continuous Y variable by a continuous X variable we're probably most interested in a scatter plot. Here, we're telling ggplot that we want to use the `gm` dataset, and the aesthetic mapping will map `gdpPercap` onto the x-axis and `lifeExp` onto the y-axis. Remember that the variable names are case sensitive!

```
ggplot(gm, aes(x = gdpPercap, y = lifeExp))
```



When we do that we get a blank canvas with no data showing (you might get an error if you're using an old version of ggplot2). That's because all we've done is laid out a two-dimensional plot specifying what goes on the x and y axes, but we haven't told it what kind of geometric object to plot. The obvious choice here is a point. Check out docs.ggplot2.org to see what kind of geoms are available.

```
ggplot(gm, aes(x = gdpPercap, y = lifeExp)) + geom_point()
```



Here, we've built our plot in layers. First, we create a canvas for plotting layers to come using the `ggplot` function, specifying which **data** to use (here, the **gm** data frame), and an **aesthetic mapping** of `gdpPercap` to the x-axis and `lifeExp` to the y-axis. We next add a layer to the plot, specifying a **geom**, or a way of visually representing the aesthetic mapping.

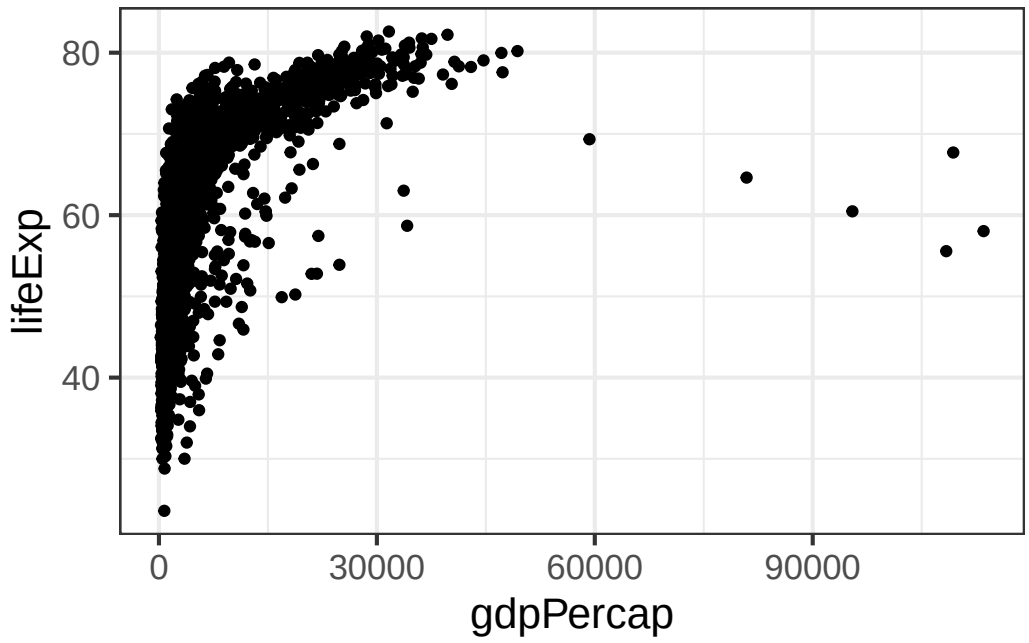
Now, the typical workflow for building up a ggplot2 plot is to first construct the figure and save that to a variable (for example, `p`), and as you're experimenting, you can continue to re-define the `p` object as you develop "keeper commands".

First, let's construct the graphic. Notice that we don't have to specify `x=` and `y=` if we specify the arguments in the correct order (x is first, y is second).

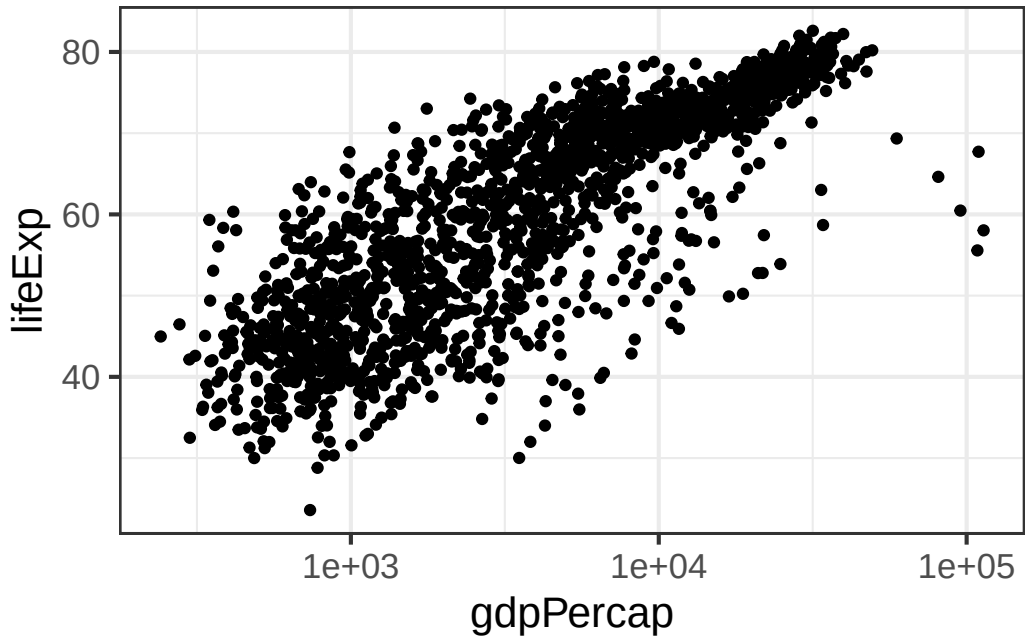
```
p <- ggplot(gm, aes(gdpPercap, lifeExp))
```

The `p` object now contains the canvas, but nothing else. Try displaying it by just running `p`. Let's experiment with adding points and a different scale to the x-axis.

```
# Experiment with adding poings
p + geom_point()
```




```
# Experiment with a different scale
p + geom_point() + scale_x_log10()
```

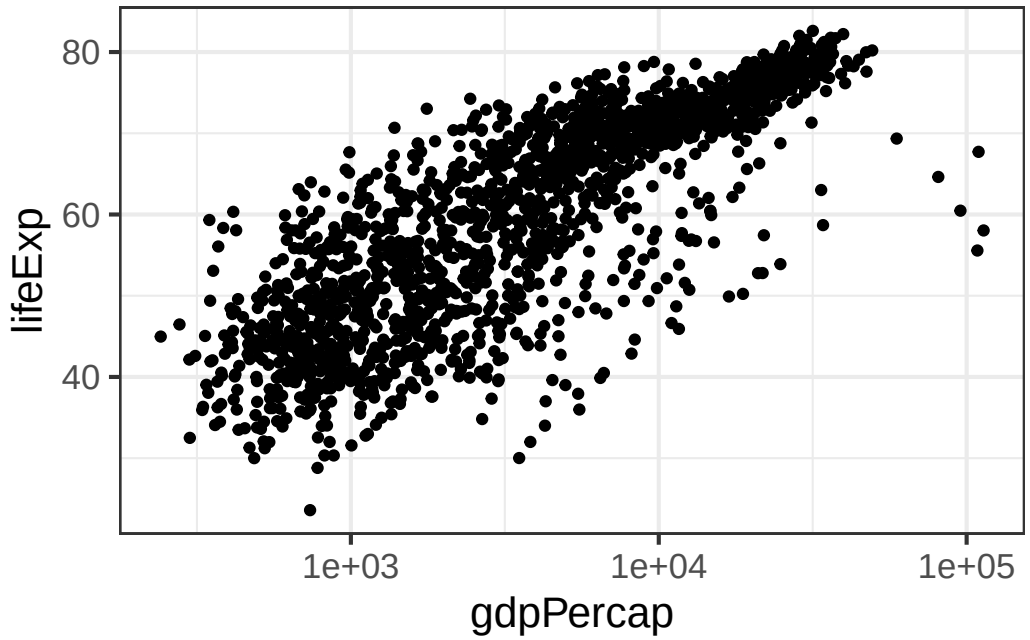


I like the look of using a log scale for the x-axis. Let's make that stick.

```
p <- p + scale_x_log10()
```

Now, if we re-ran `p` still nothing would show up because the `p` object just contains a blank canvas. Now, re-plot again with a layer of points:

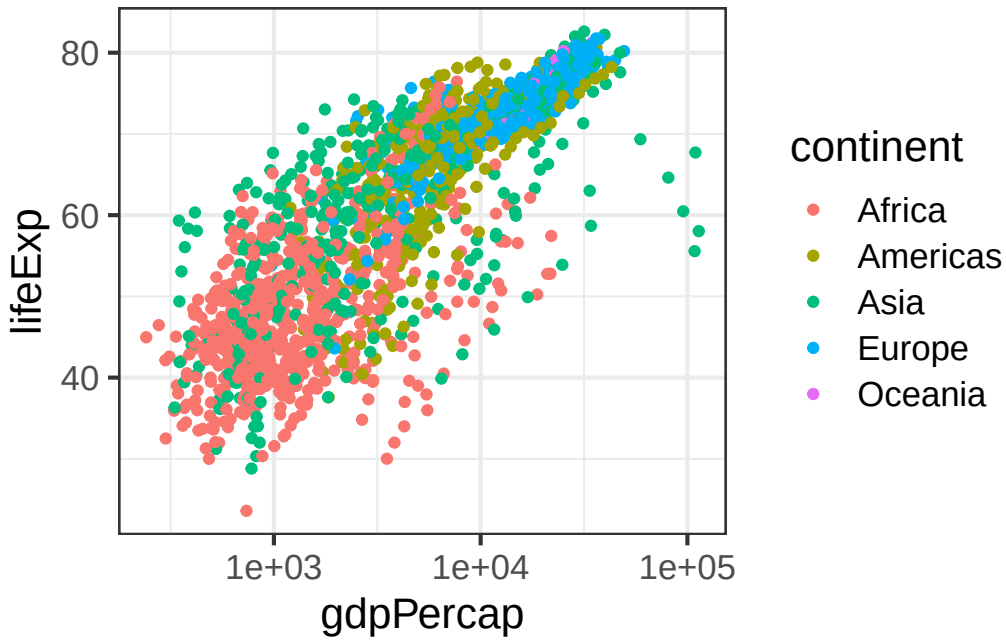
```
p + geom_point()
```



Now notice what I've saved to `p` at this point: only the basic plot layout and the log10 mapping on the x-axis. I didn't save any layers yet because I want to fiddle around with the points for a bit first.

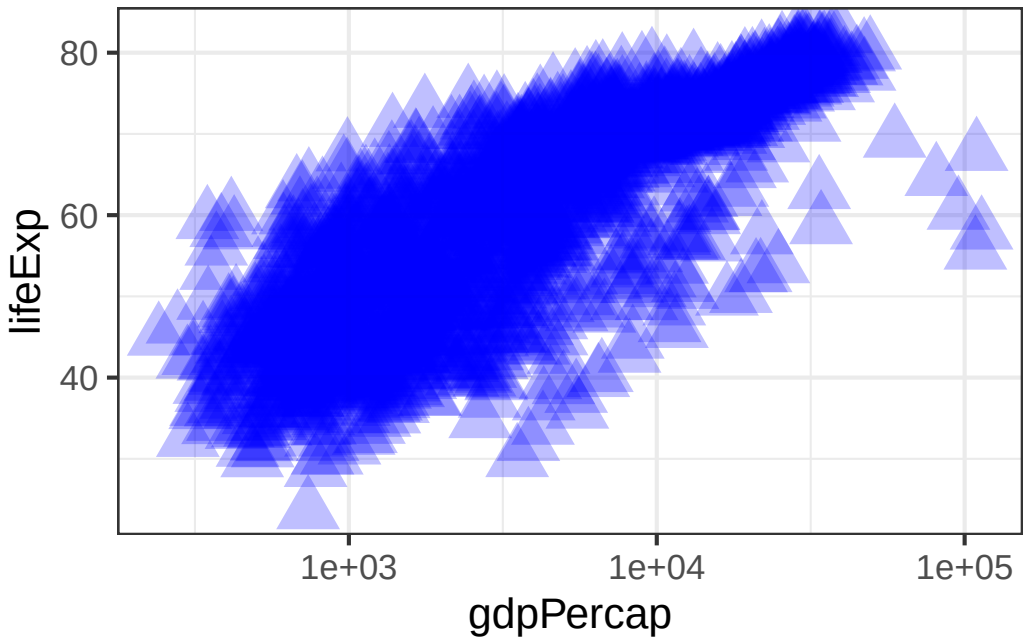
Above we implied the aesthetic mappings for the x- and y- axis should be `gdpPercap` and `lifeExp`, but we can also add aesthetic mappings to the geoms themselves. For instance, what if we wanted to color the points by the value of another variable in the dataset, say, continent?

```
p + geom_point(aes(color=continent))
```



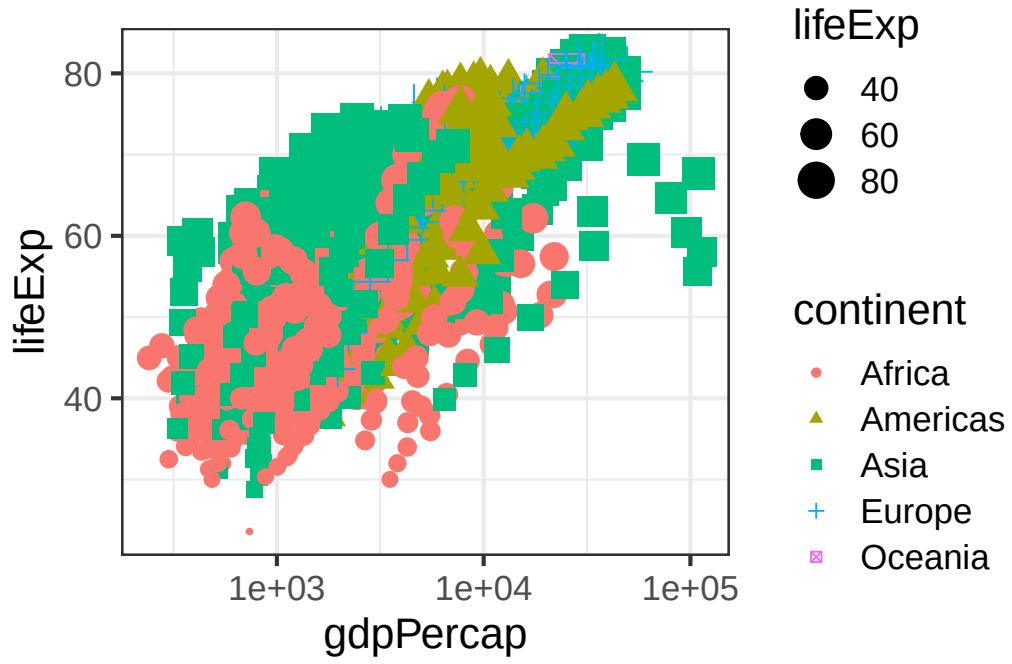
Notice the difference here. If I wanted the colors to be some static value, I wouldn't wrap that in a call to `aes()`. I would just specify it outright. Same thing with other features of the points. For example, lets make all the points huge (`size=8`) blue (`color="blue"`) semitransparent (`alpha=(1/4)`) triangles (`pch=17`):

```
p + geom_point(color="blue", pch=17, size=8, alpha=1/4)
```



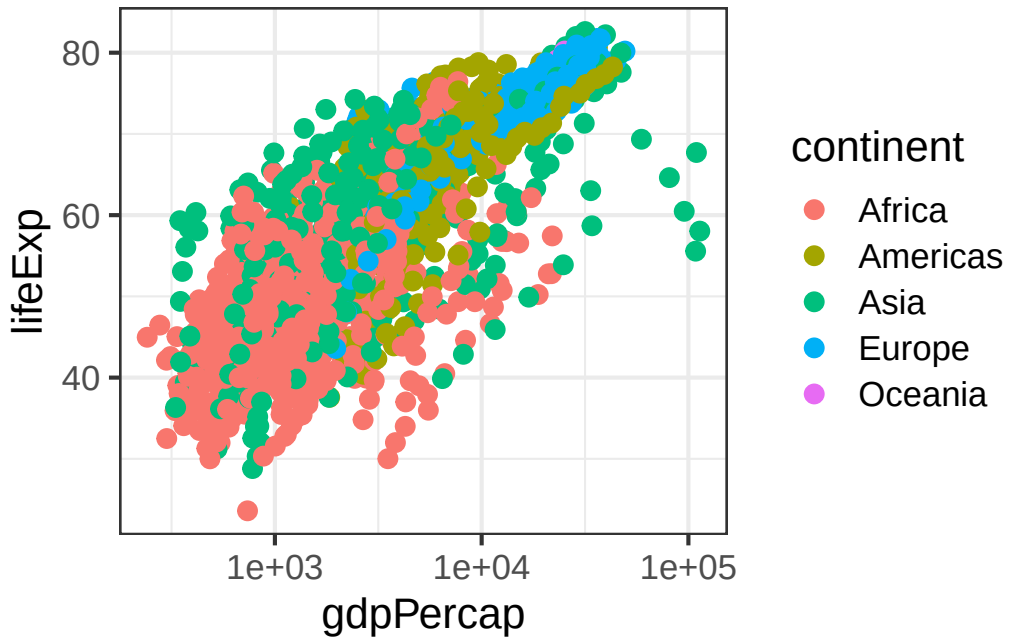
Now, this time, let's map the aesthetics of the point character to certain features of the data. For instance, let's give the points different colors and character shapes according to the continent, and map the size of the point onto the life Expectancy:

```
p + geom_point(aes(col=continent, shape=continent, size=lifeExp))
```



Now, this isn't a great plot because there are several aesthetic mappings that are redundant. Life expectancy is mapped to both the y-axis and the size of the points – the size mapping is superfluous. Similarly, continent is mapped to both the color and the point character (the shape is superfluous). Let's get rid of that, but let's make the points a little bigger outsize of an aesthetic mapping.

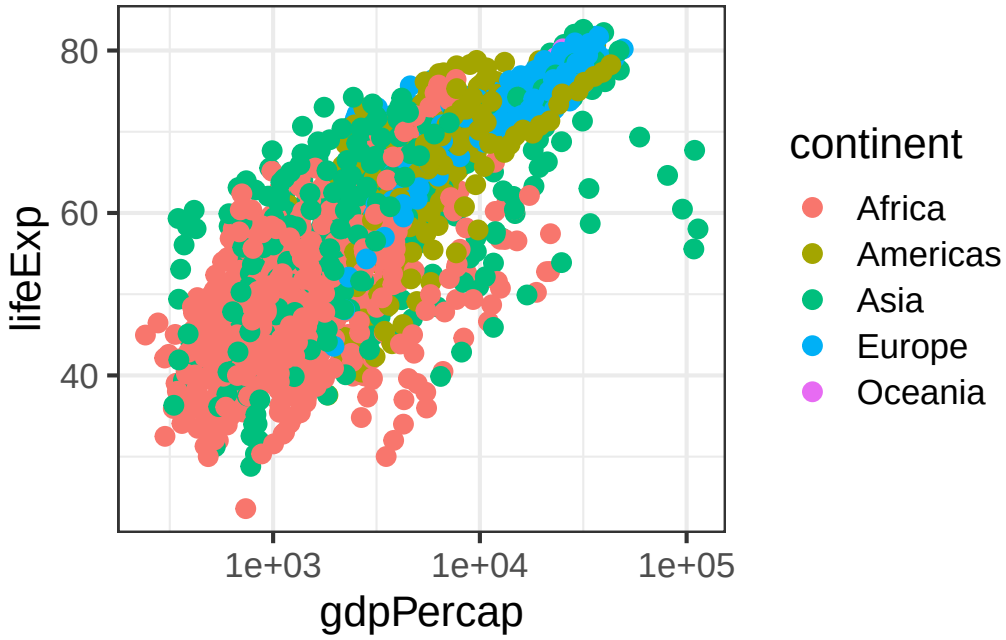
```
p + geom_point(aes(col=continent), size=3)
```



Exercise 1

Re-create this same plot from scratch without saving anything to a variable. That is, start from the `ggplot` call.

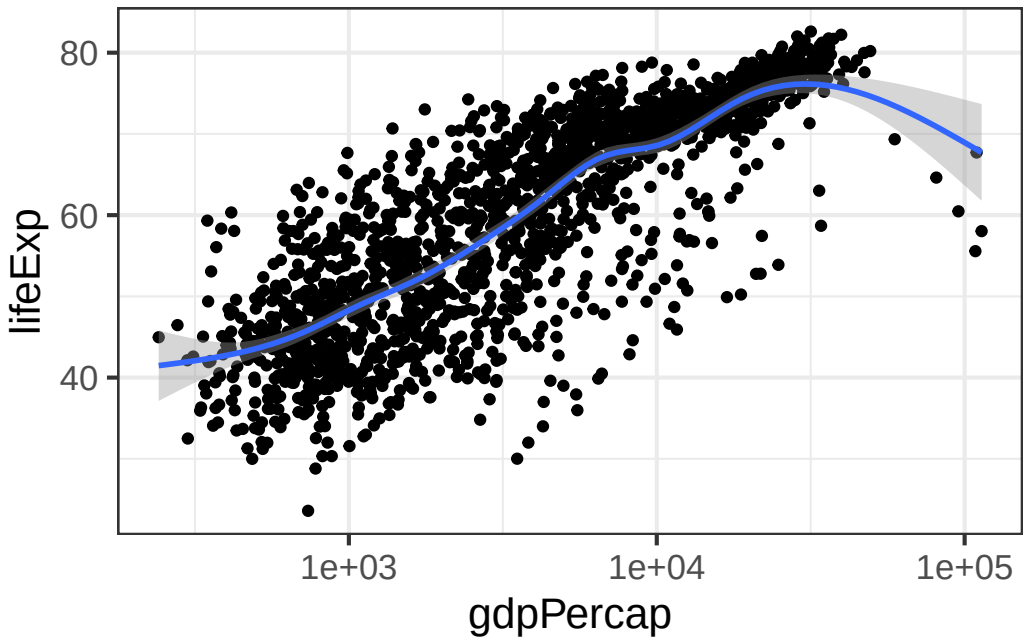
- Start with the `ggplot()` function.
- Use the gm data.
- Map `gdpPercap` to the x-axis and `lifeExp` to the y-axis.
- Add points to the plot
  - Make the points size 3
  - Map continent onto the aesthetics of the point
- Use a log10 scale for the x-axis.



5.3.1 Adding layers

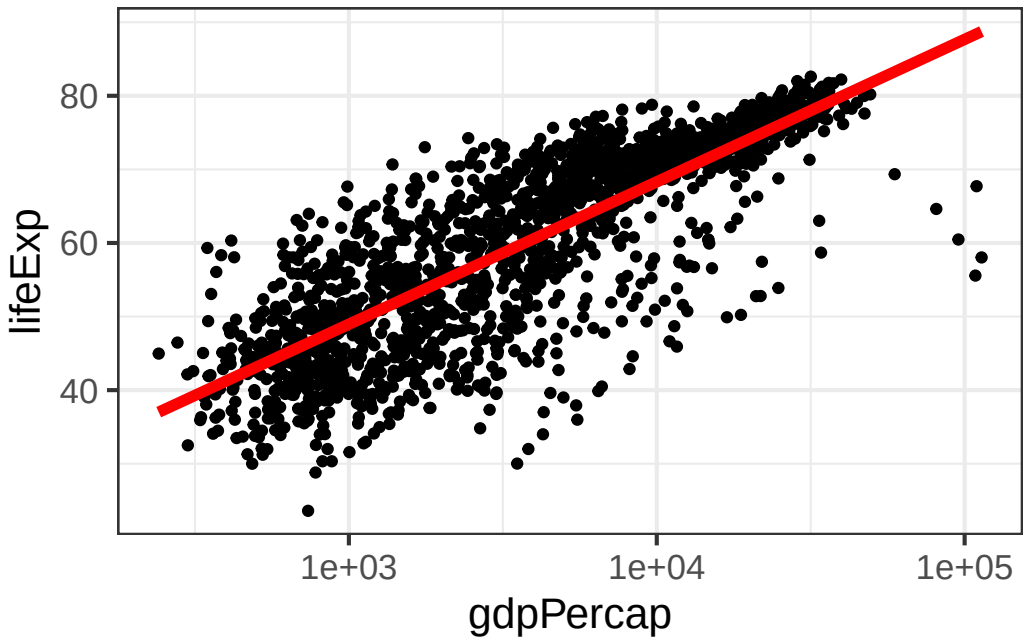
Let's add a fitted curve to the points. Recreate the plot in the `p` object if you need to.

```
p <- ggplot(gm, aes(gdpPercap, lifeExp)) + scale_x_log10()
p + geom_point() + geom_smooth()
```



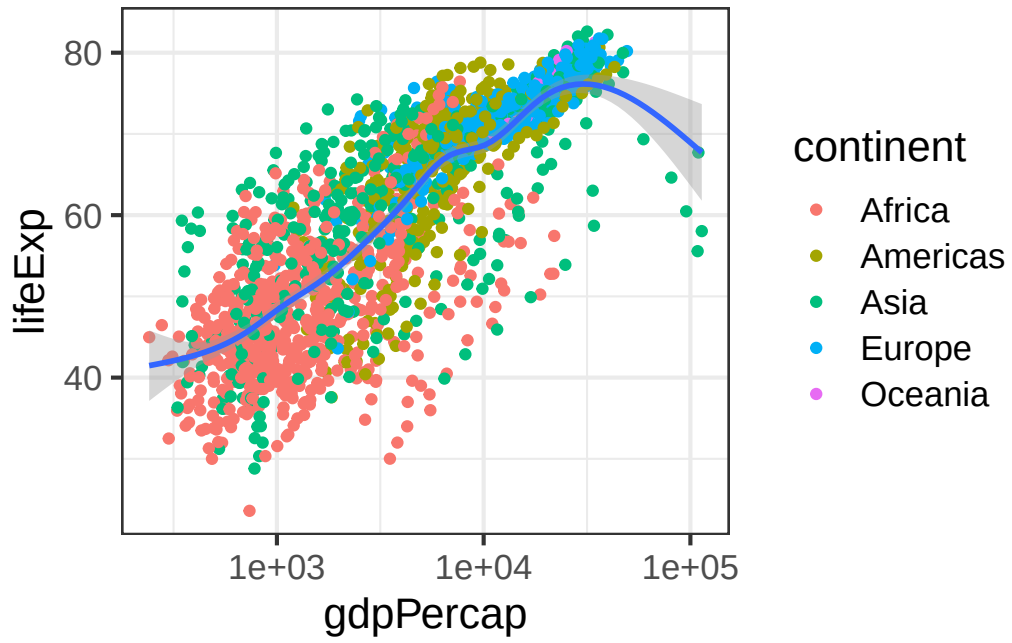
By default `geom_smooth()` will try to lowess for data with n<1000 or generalized additive models for data with n>1000. We can change that behavior by tweaking the parameters to use a thick red line, use a linear model instead of a GAM, and to turn off the standard error stripes.

```
p + geom_point() + geom_smooth(lwd=2, se=FALSE, method="lm", col="red")
```



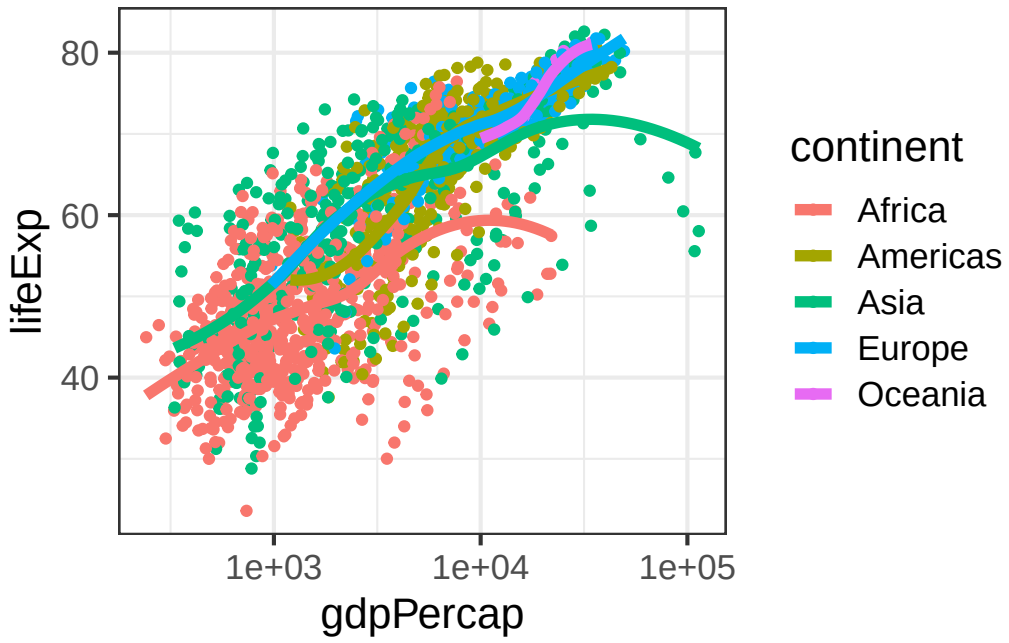
But let's add back in our aesthetic mapping to the continents. Notice what happens here. We're mapping continent as an aesthetic mapping *to the color of the points only* – so `geom_smooth()` still works only on the entire data.

```
p + geom_point(aes(color = continent)) + geom_smooth()
```



But notice what happens here: we make the call to `aes()` outside of the `geom_point()` call, and the continent variable gets mapped as an aesthetic to any further geoms. So here, we get separate smoothing lines for each continent. Let's do it again but remove the standard error stripes and make the lines a bit thicker.

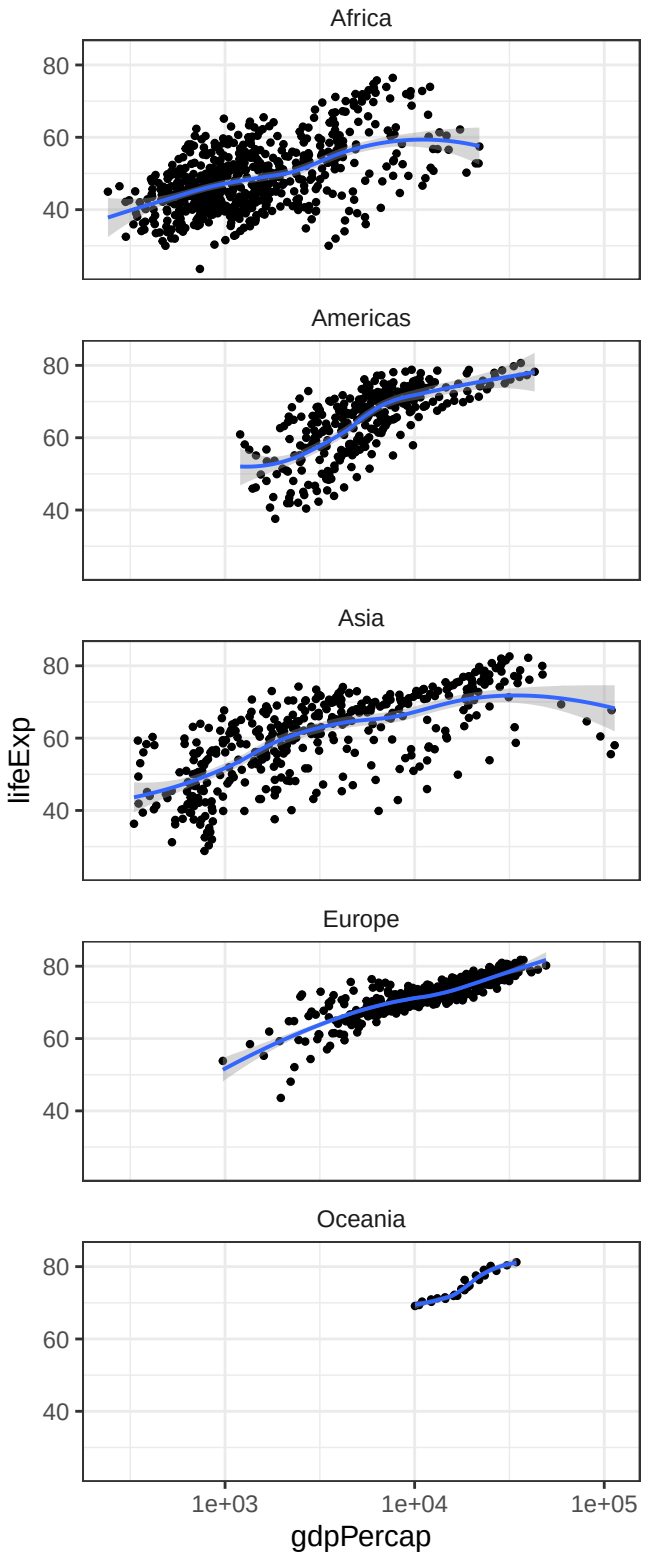
```
p + aes(color = continent) + geom_point() + geom_smooth()
p + aes(color = continent) + geom_point() + geom_smooth(se=F, lwd=2)
```



5.3.2 Faceting

Facets display subsets of the data in different panels. There are a couple ways to do this, but `facet_wrap()` tries to sensibly wrap a series of facets into a 2-dimensional grid of small multiples. Just give it a formula specifying which variables to facet by. We can continue adding more layers, such as smoothing. If you have a look at the help for `?facet_wrap()` you'll see that we can control how the wrapping is laid out.

```
p + geom_point() + facet_wrap(~continent)
p + geom_point() + geom_smooth() + facet_wrap(~continent, ncol=1)
```


Africa
Americas
Asia
Europe
Oceania
lifeExp
80
60
40
1e+03
1e+04
1e+05
gdpPercap

5.3.3 Saving plots

There are a few ways to save ggplots. The quickest way, that works in an interactive session, is to use the `ggsave()` function. You give it a file name and by default it saves the last plot that was printed to the screen.

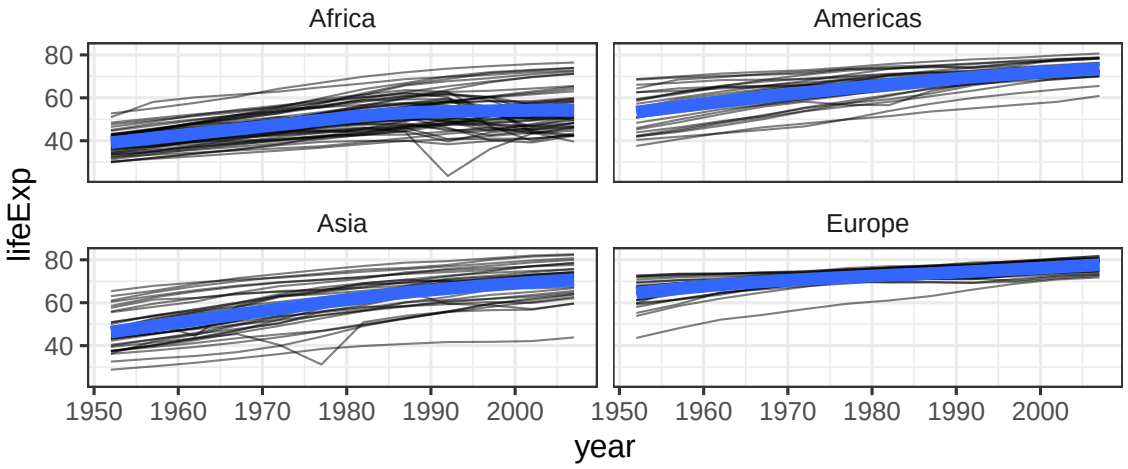
```
p + geom_point()
ggsave(file="myplot.png")
```

But if you're running this through a script, the best way to do it is to pass `ggsave()` the object containing the plot that is meant to be saved. We can also adjust things like the width, height, and resolution. `ggsave()` also recognizes the name of the file extension and saves the appropriate kind of file. Let's save a PDF.

```
pfinal <- p + geom_point() + geom_smooth() + facet_wrap(~continent, ncol=1)
ggsave(pfinal, file="myplot.pdf", width=5, height=15)
```

Exercise 2

1. Make a scatter plot of `lifeExp` on the y-axis against `year` on the x.
2. Make a series of small multiples faceting on continent.
3. Add a fitted curve, smooth or lm, with and without facets.
4. **Bonus**: using `geom_line()` and and aesthetic mapping `country` to `group=`, make a "spaghetti plot", showing *semitransparent* lines connected for each country, faceted by continent. Add a smoothed loess curve with a thick (`lwd=3`) line with no standard error stripe. Reduce the opacity (`alpha=`) of the individual black lines. *Don't* show Oceania countries (that is, `filter()` the data where `continent!="Oceania"` before you plot it).



5.4 Plotting bivariate data: continuous Y by categorical X

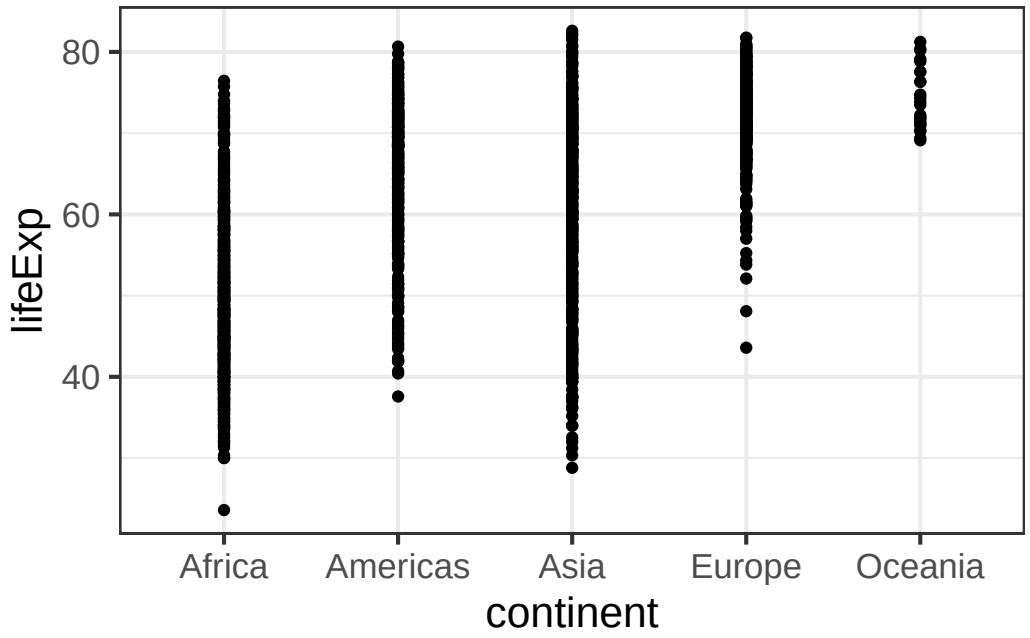
With the last example we examined the relationship between a continuous Y variable against a continuous X variable. A scatter plot was the obvious kind of data visualization. But what if we wanted to visualize a continuous Y variable against a categorical X variable? We sort of saw what that looked like in the last exercise. `year` is a continuous variable, but in this dataset, it's broken up into 5-year segments, so you could almost think of each year as a categorical variable. But a better example would be life expectancy against continent or country.

First, let's set up the basic plot:

```
p <- ggplot(gm, aes(continent, lifeExp))
```

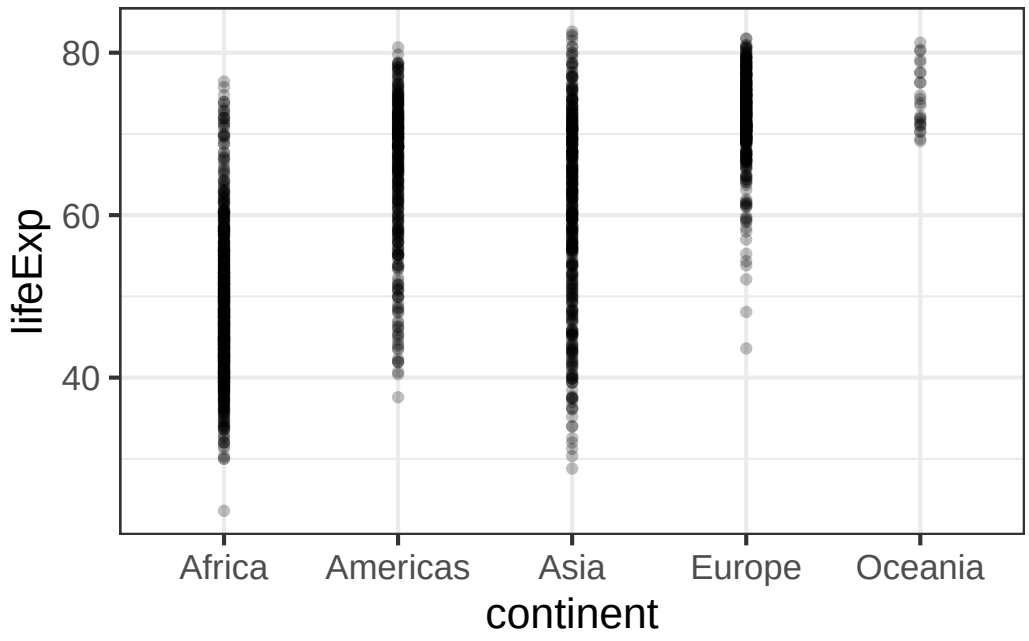
Then add points:

```
p + geom_point()
```



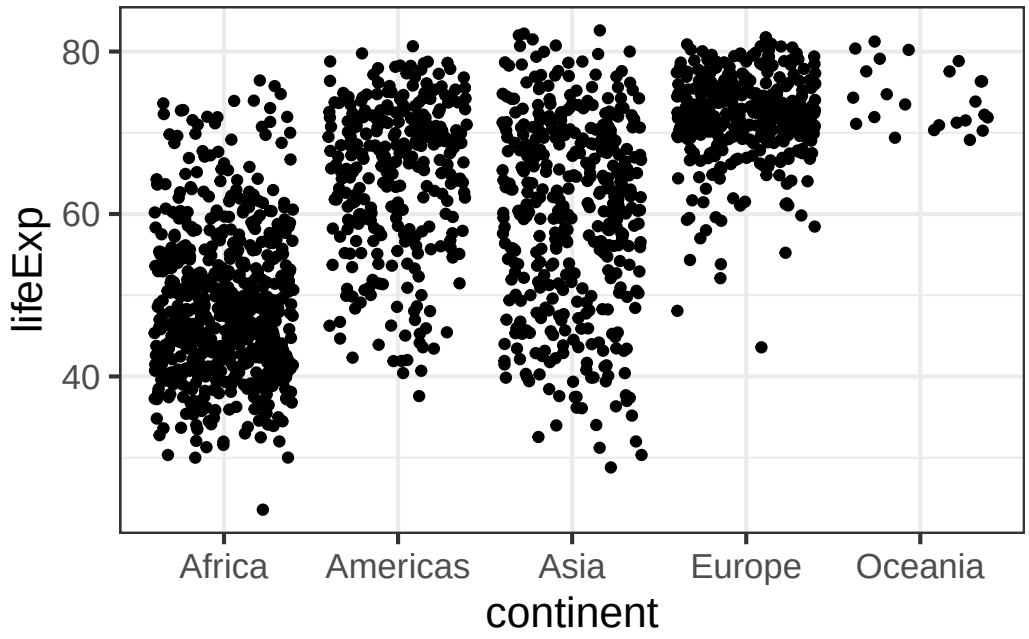
That's not terribly useful. There's a big overplotting problem. We can try to solve with transparency:

```
p + geom_point(alpha=1/4)
```



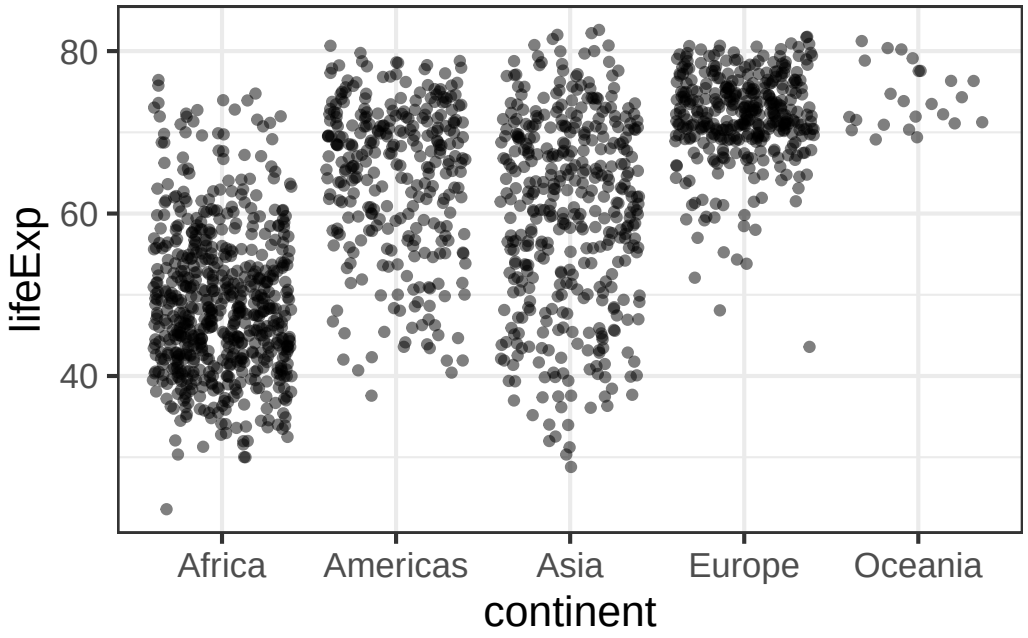
But that really only gets us so far. What if we spread things out by adding a little bit of horizontal noise (aka "jitter") to the data.

```
p + geom_jitter()
```



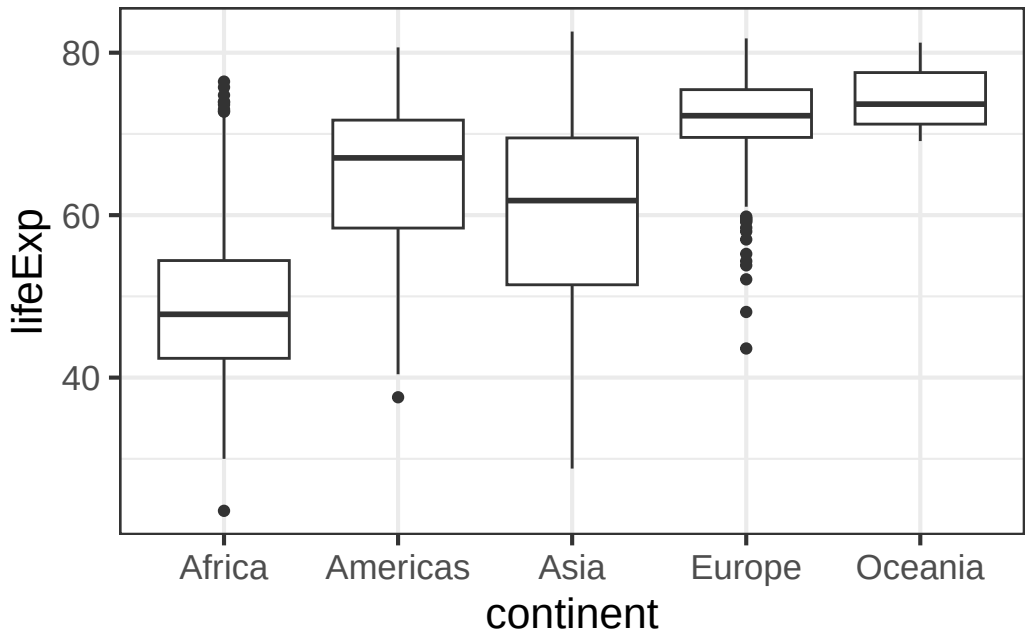
Note that the little bit of horizontal noise that's added to the jitter is random. If you run that command over and over again, each time it will look slightly different. The idea is to visualize the density at each vertical position, and spreading out the points horizontally allows you to do that. If there were still lots of over-plotting you might think about adding some transparency by setting the `alpha=` value for the jitter.

```
p + geom_jitter(alpha=1/2)
```



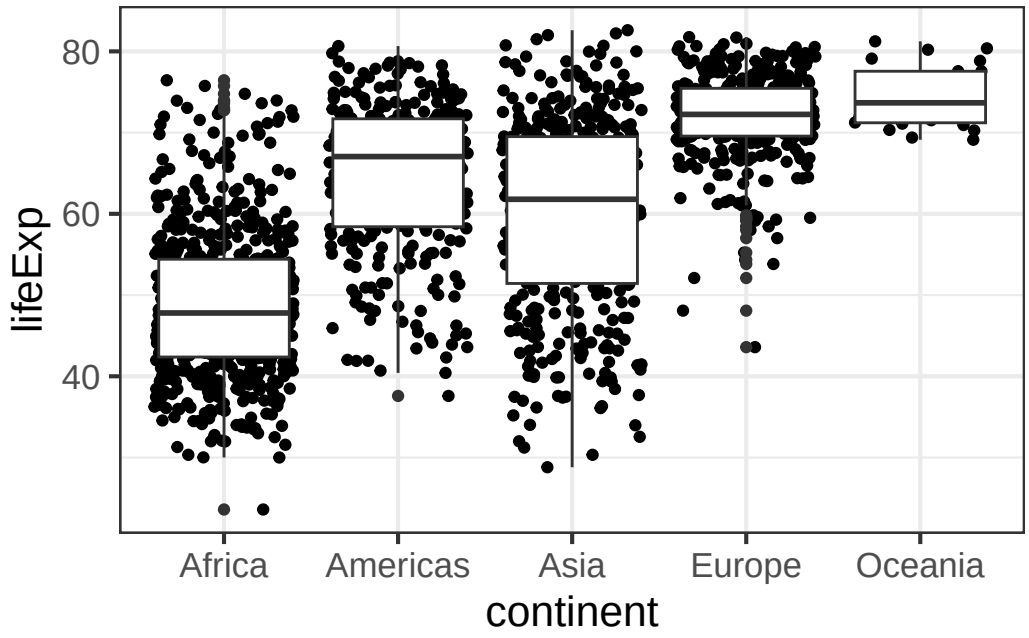
Probably a more common visualization is to show a box plot:

```
p + geom_boxplot()
```



But why not show the summary and the raw data?

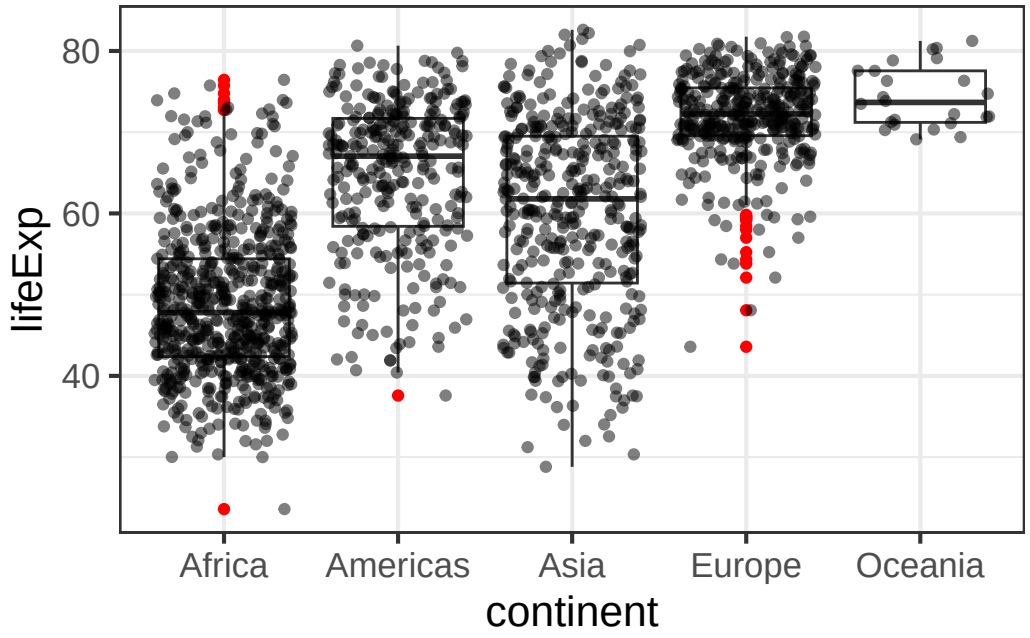
```
p + geom_jitter() + geom_boxplot()
```



Notice how in that example we first added the jitter layer then added the boxplot layer. But

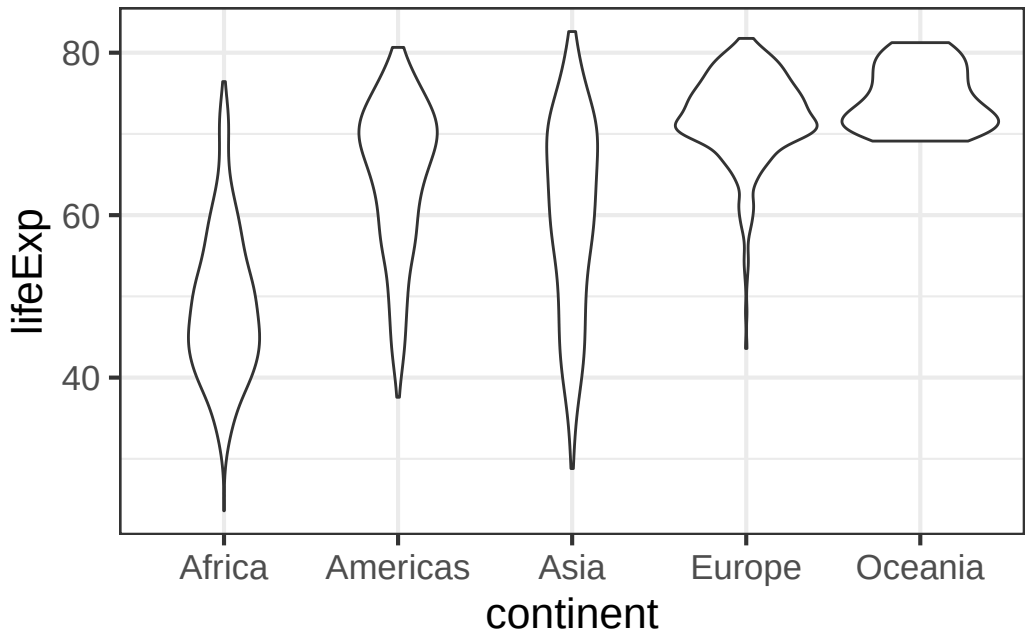
the boxplot is now superimposed over the jitter layer. Let's make the jitter layer go on top. Also, go back to just the boxplots. Notice that the outliers are represented as points. But there's no distinction between the outlier point from the boxplot geom and all the other points from the jitter geom. Let's change that. Notice the British spelling.

```
p + geom_boxplot(outlier.colour = "red") + geom_jitter(alpha=1/2)
```

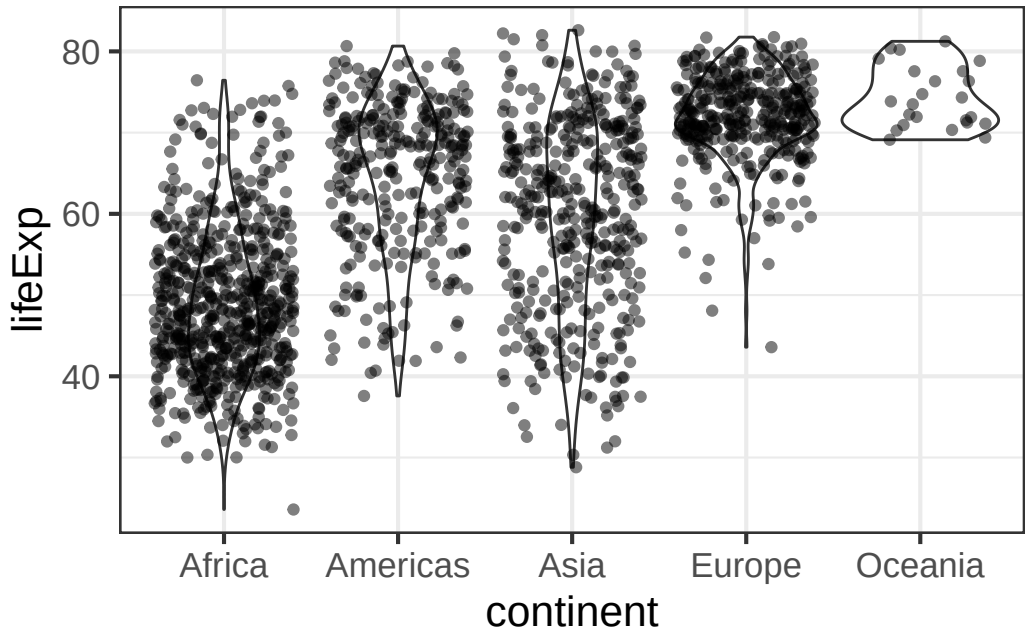


There's another geom that's useful here, called a voilin plot.

```
p + geom_violin()
```

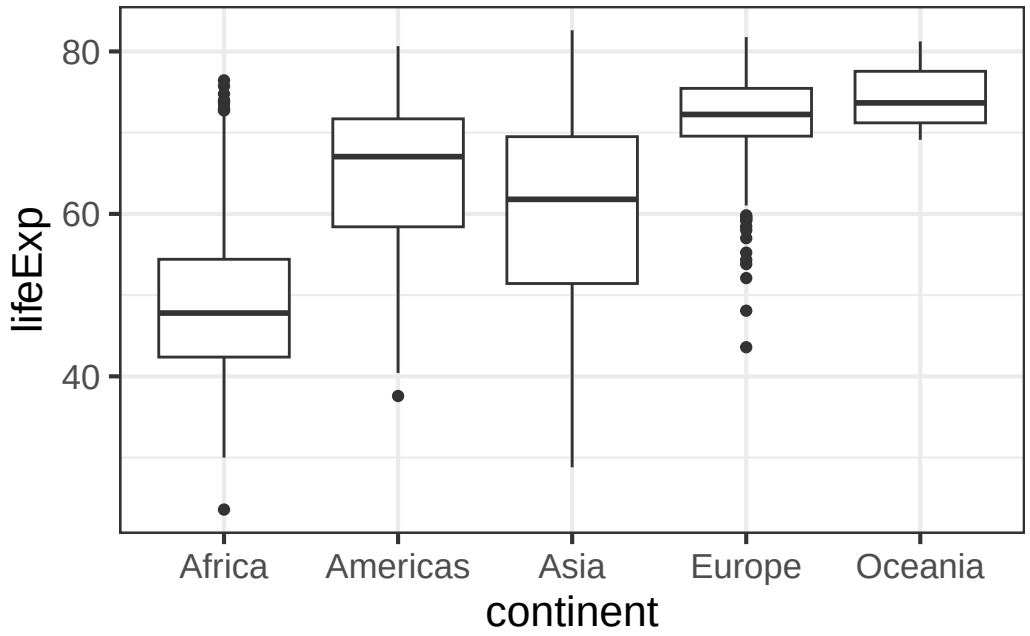



```
p + geom_violin() + geom_jitter(alpha=1/2)
```



Let's go back to our boxplot for a moment.

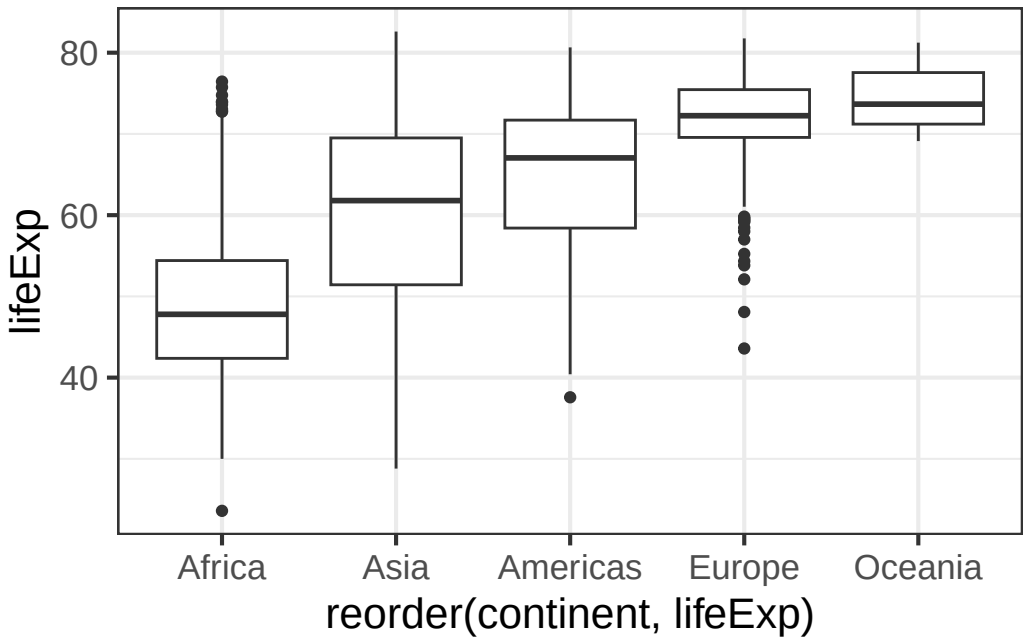
```
p + geom_boxplot()
```



This plot would be a lot more effective if the continents were shown in some sort of order other than alphabetical. To do that, we'll have to go back to our basic build of the plot again and use the `reorder` function in our original aesthetic mapping. Here, reorder is taking the first variable, which is some categorical variable, and ordering it by the level of the mean of the second variable, which is a continuous variable. It looks like this

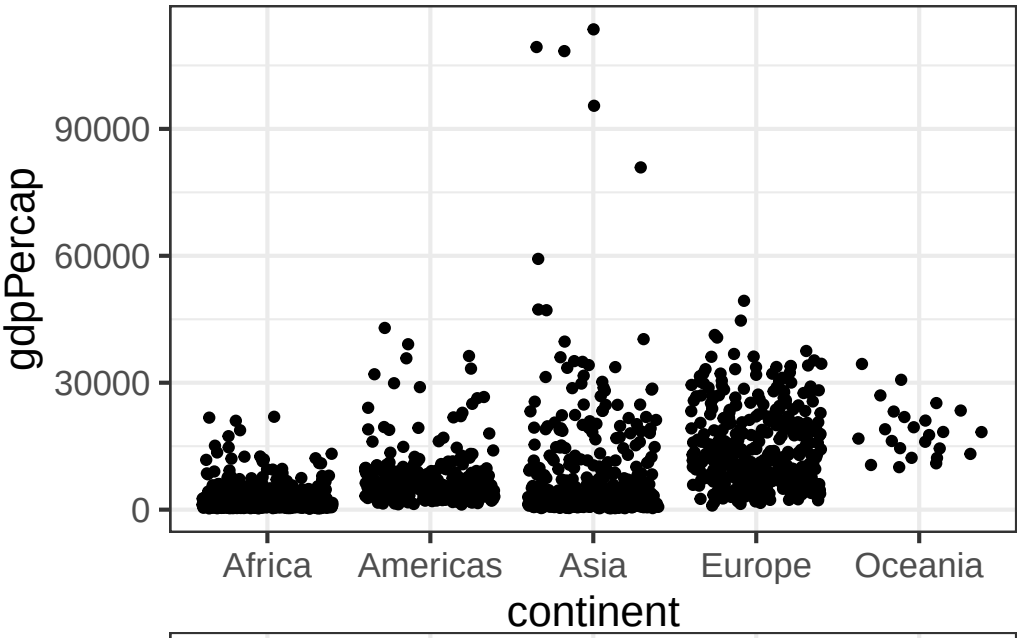
```
p <- ggplot(gm, aes(x=reorder(continent, lifeExp), y=lifeExp))
```

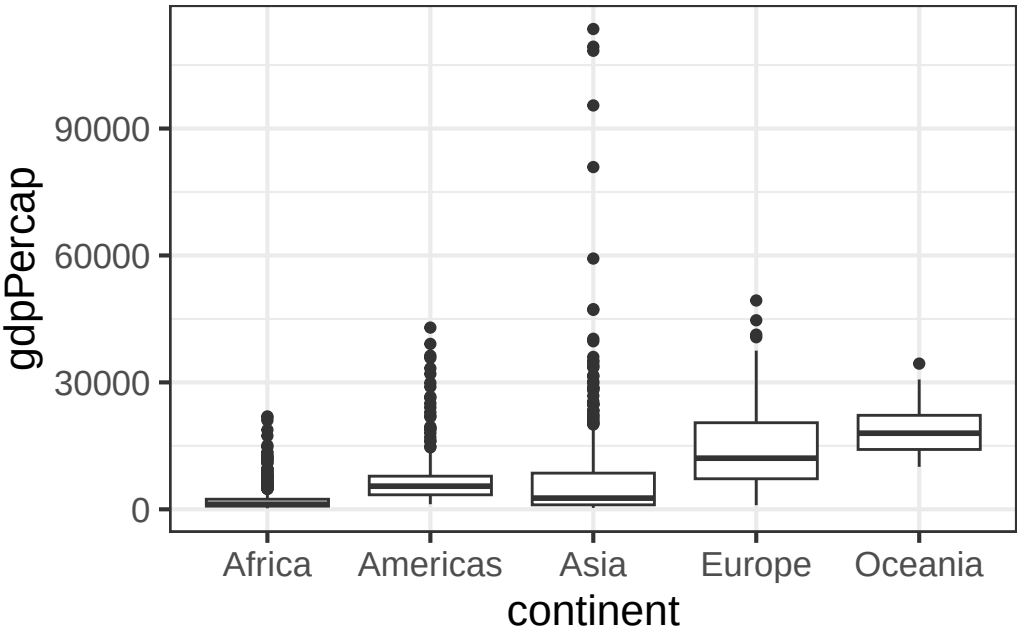
```
p + geom_boxplot()
```

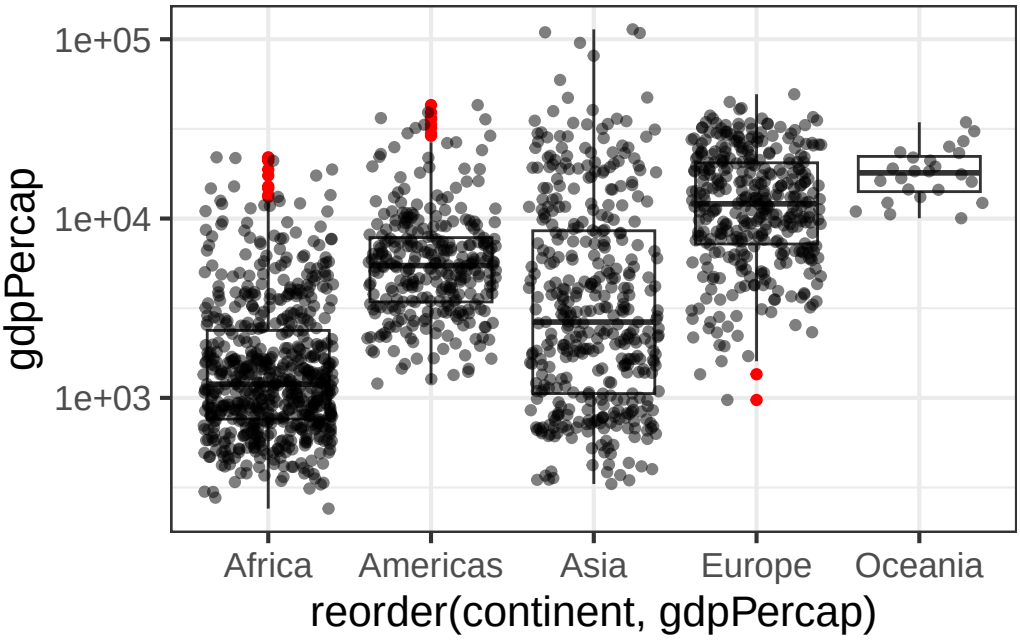


Exercise 3

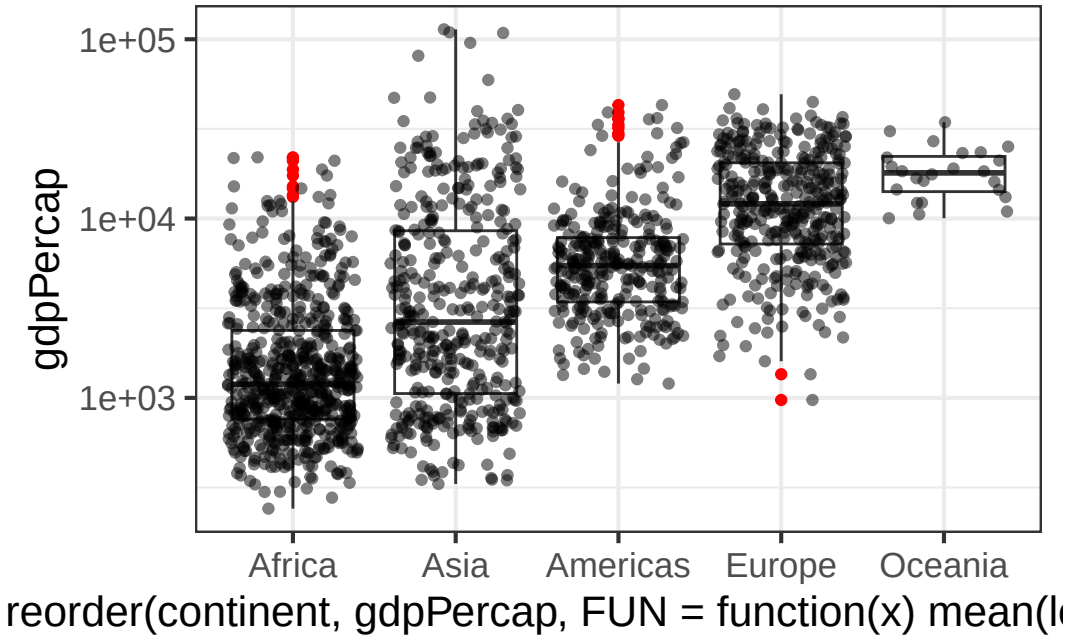
1. Make a jittered strip plot of GDP per capita against continent.
2. Make a box plot of GDP per capita against continent.
3. Using a log10 y-axis scale, overlay semitransparent jittered points on top of box plots, where outlying points are colored.
4. **BONUS**: Try to reorder the continents on the x-axis by GDP per capita. Why isn't this working as expected? See `?reorder` for clues.


gdpPercap
90000
60000
30000
0
Africa
Americas
Asia
Europe
Oceania
continent


gdpPercap
90000
60000
30000
0
Africa
Americas
Asia
Europe
Oceania
continent




```
# A tibble: 5 x 2
  continent `mean(gdpPercap)`
  <chr>                   <dbl>
1 Africa                  2194.
2 Americas                7136.
3 Asia                    7902.
4 Europe                 14469.
5 Oceania                18622.

# A tibble: 5 x 2
  continent `mean(log10(gdpPercap))`
  <chr>                          <dbl>
1 Africa                          3.15
2 Americas                        3.74
3 Asia                            3.51
4 Europe                          4.06
5 Oceania                         4.25
```

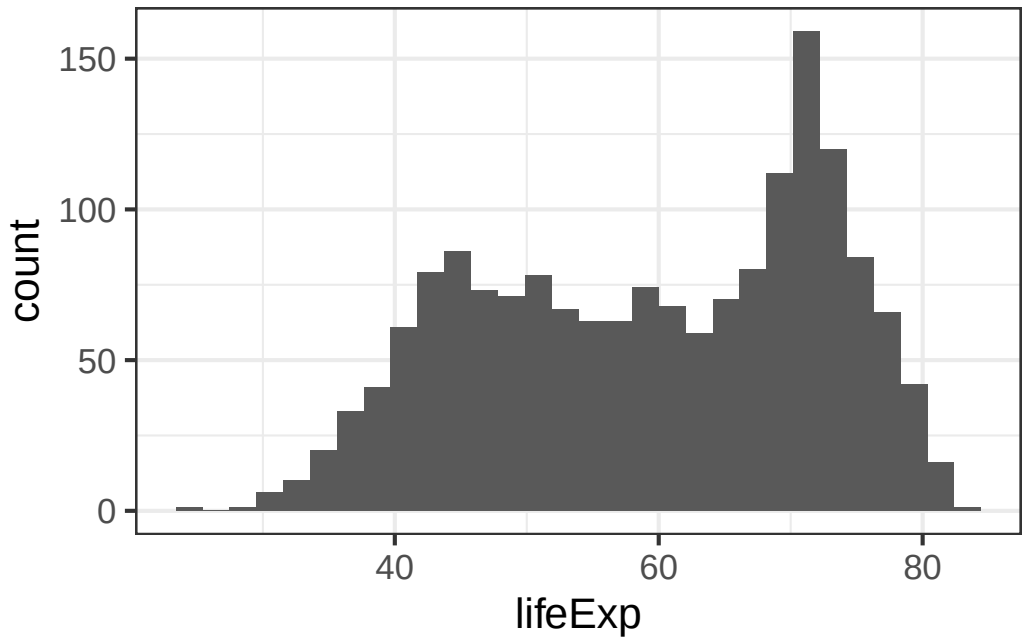


5.5 Plotting univariate continuous data

What if we just wanted to visualize distribution of a single continuous variable? A histogram is the usual go-to visualization. Here we only have one aesthetic mapping instead of two.

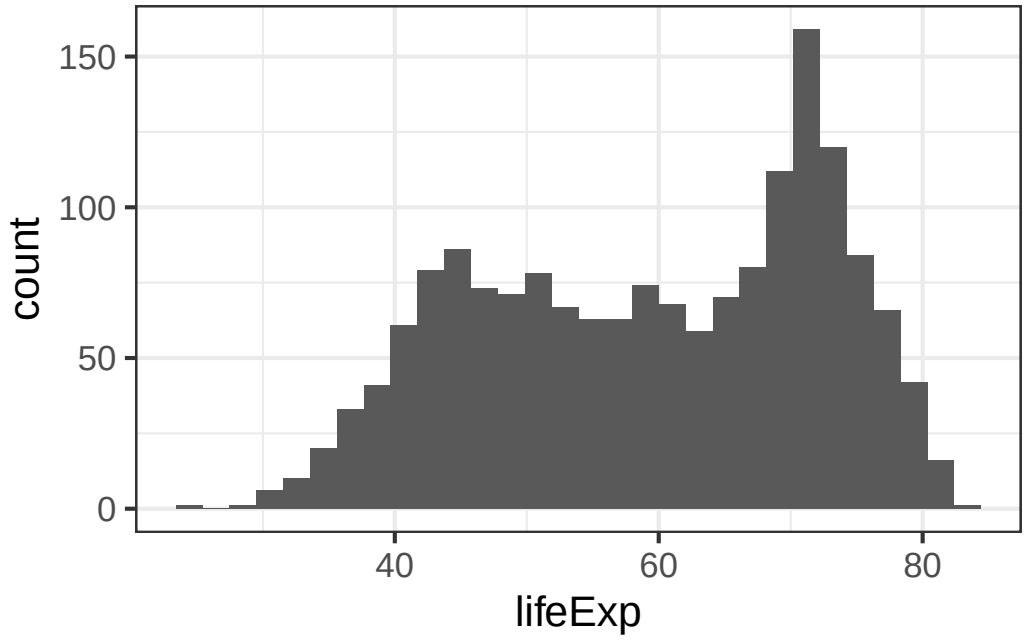
```
p <- ggplot(gm, aes(lifeExp))
```

```
p + geom_histogram()
```

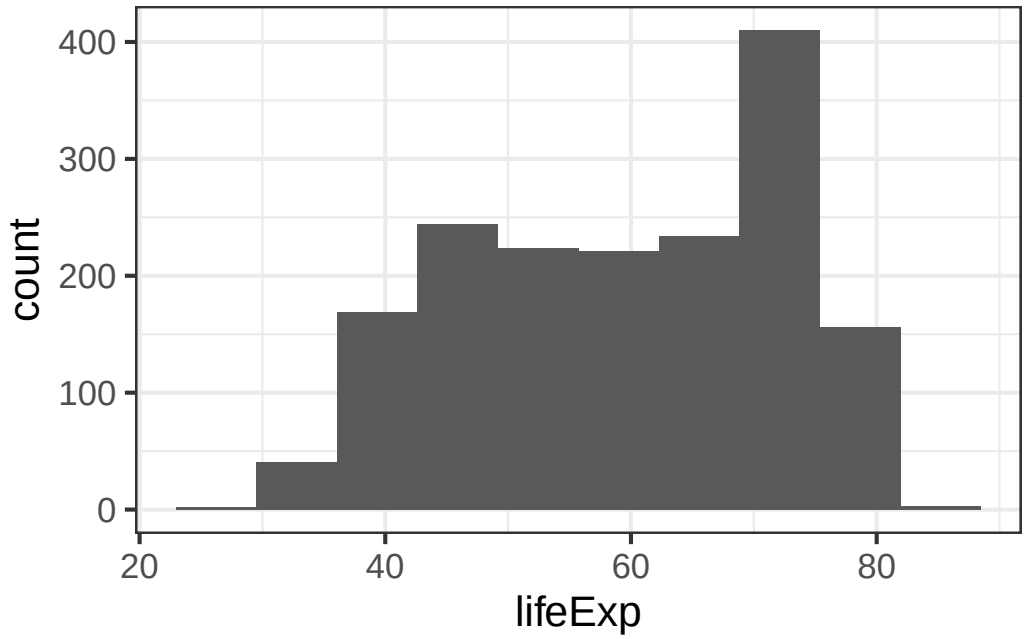


When we do this ggplot lets us know that we're automatically selecting the width of the bins, and we might want to think about this a little further.

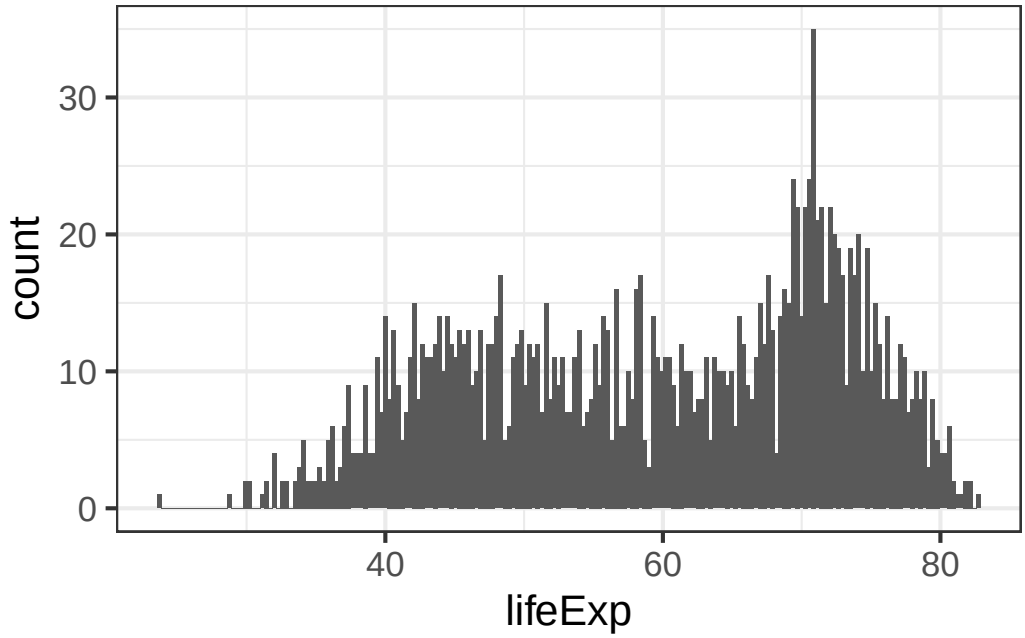
```
p + geom_histogram(bins=30)
```



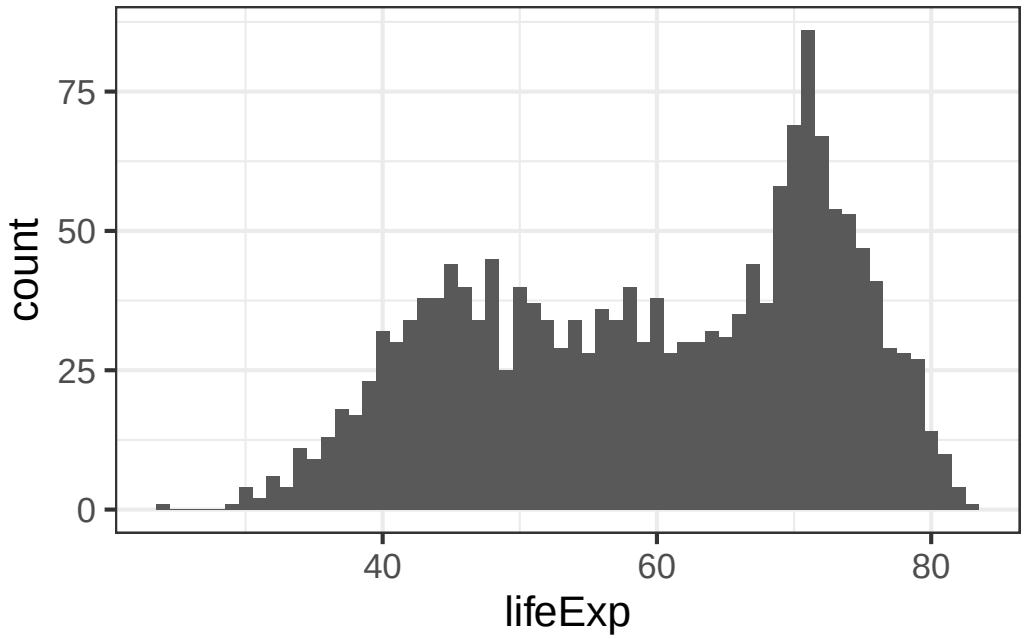
```
p + geom_histogram(bins=10)
```




```
p + geom_histogram(bins=200)
```

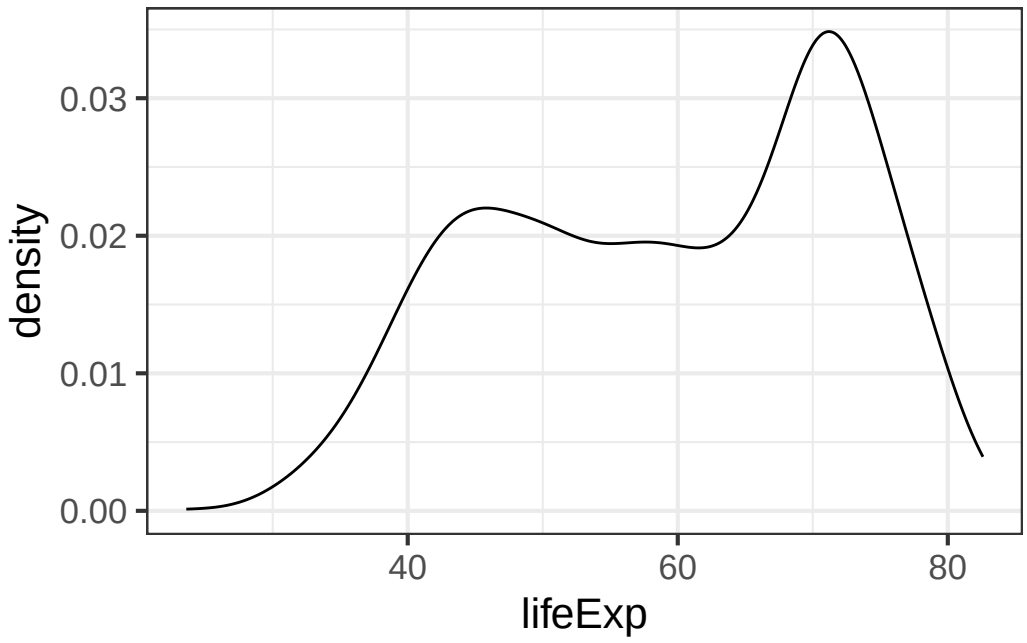


```
p + geom_histogram(bins=60)
```



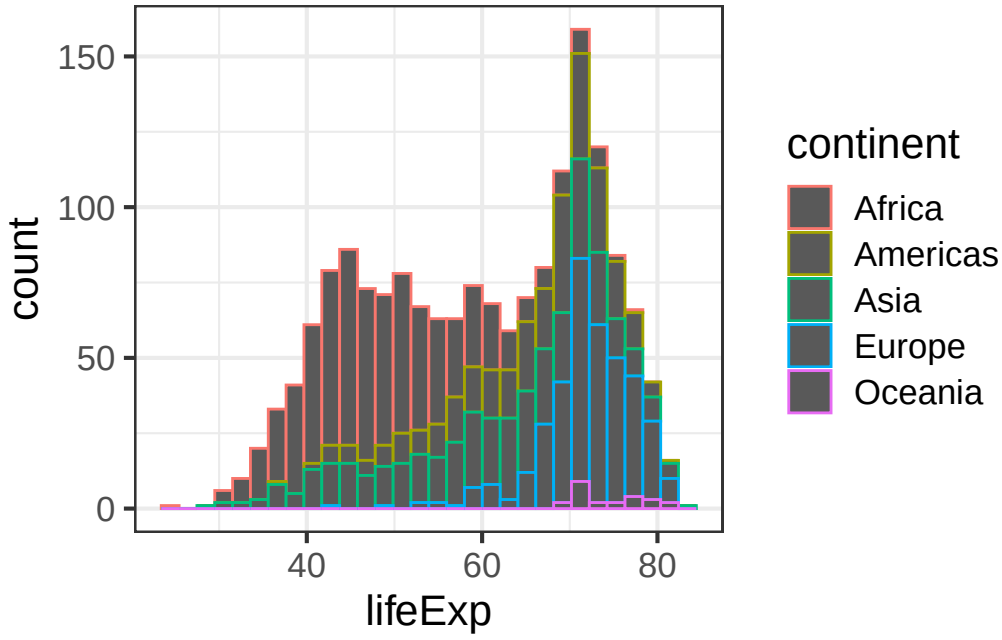
Alternative we could plot a smoothed density curve instead of a histogram:

```
p + geom_density()
```



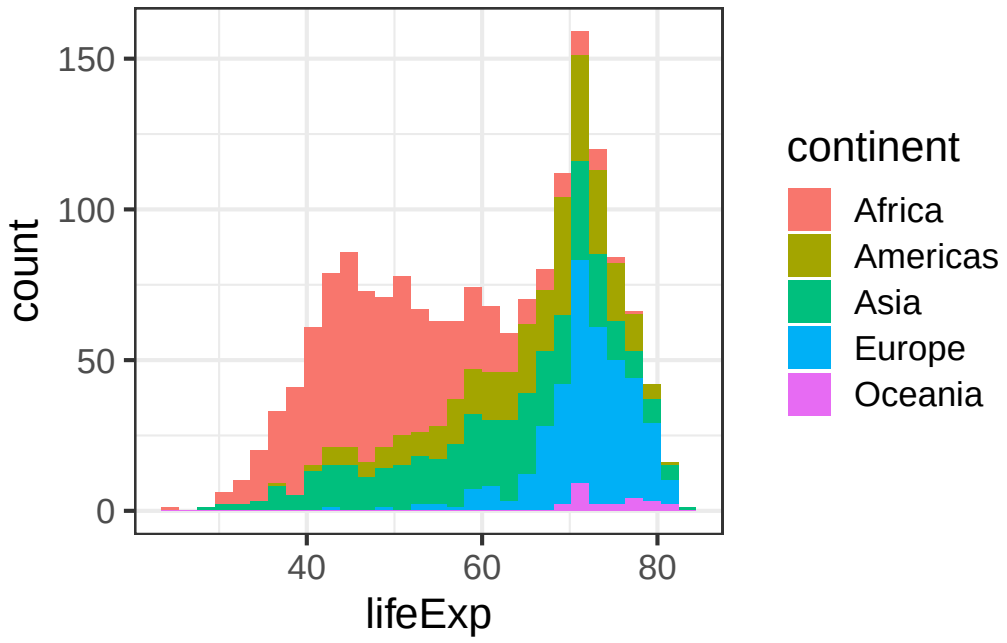
Back to histograms. What if we wanted to color this by continent?

```
p + geom_histogram(aes(color=continent))
```



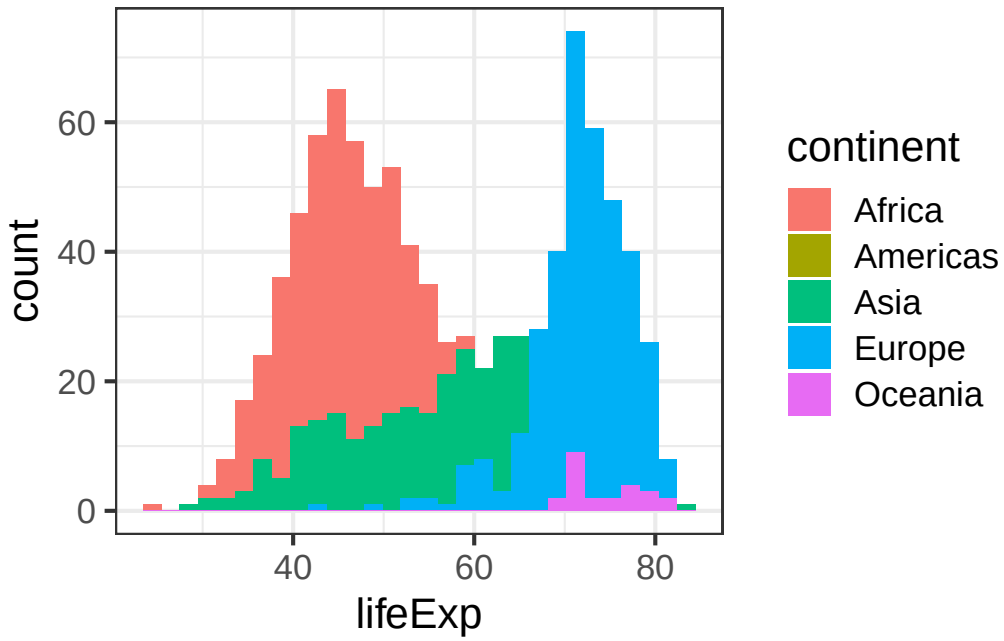
That's not what we had in mind. That's just the outline of the bars. We want to change the *fill* color of the bars.

```
p + geom_histogram(aes(fill=continent))
```



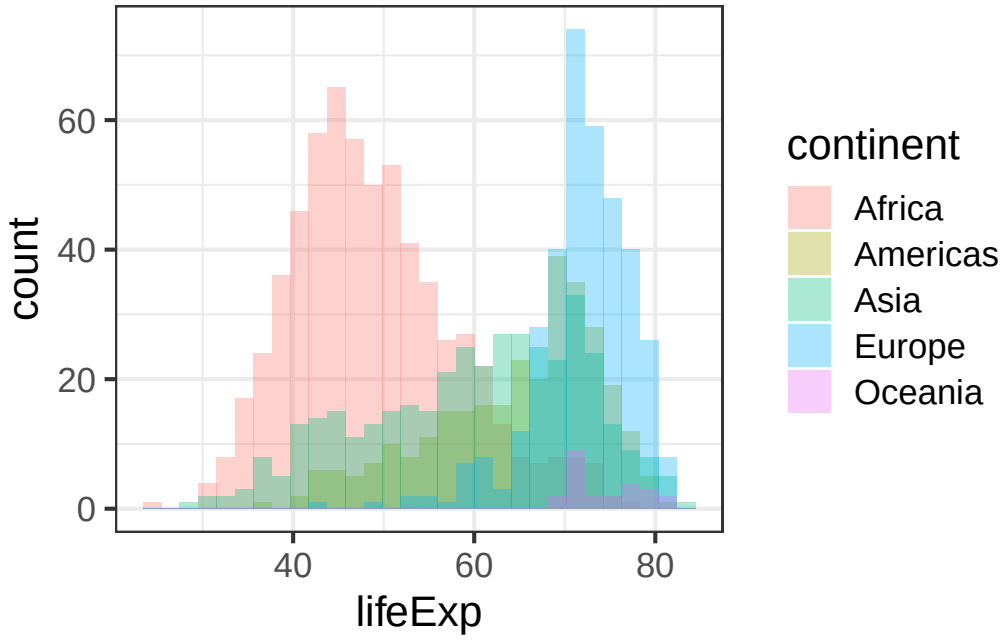
Well, that's not exactly what we want either. If you look at the help for `?geom_histogram` you'll see that by default it stacks overlapping points. This isn't really an effective visualization. Let's change the position argument.

```
p + geom_histogram(aes(fill=continent), position="identity")
```



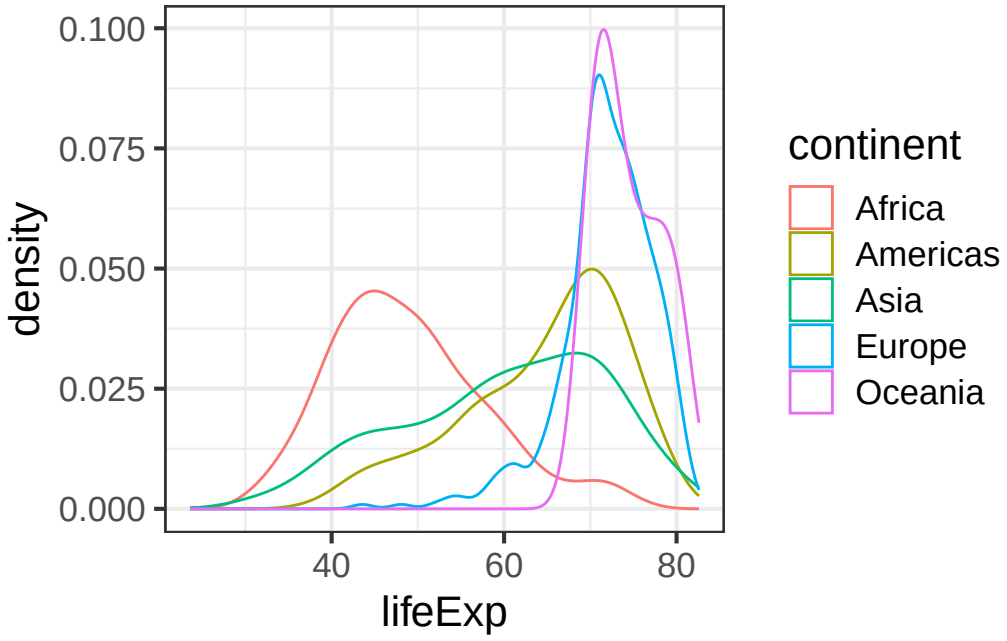
But the problem there is that the histograms are blocking each other. What if we tried transparency?

```
p + geom_histogram(aes(fill=continent), position="identity", alpha=1/3)
```



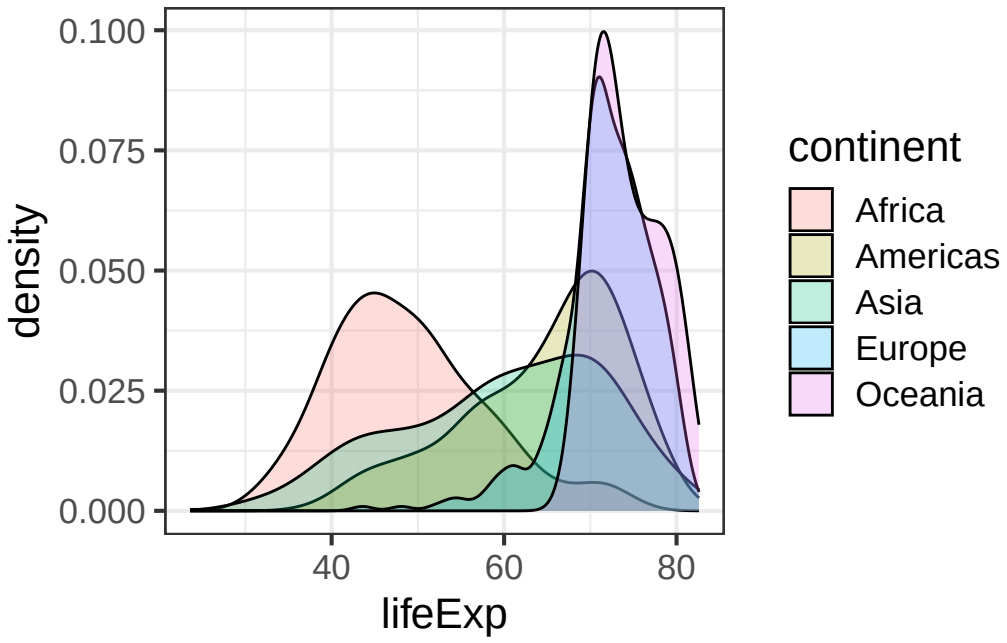
That's somewhat helpful, and might work for two distributions, but it gets cumbersome with 5. Let's go back and try this with density plots, first changing the color of the line:

```
p + geom_density(aes(color=continent))
```



Then by changing the color of the fill and setting the transparency to 25%:

```
p + geom_density(aes(fill=continent), alpha=1/4)
```



Exercise 4

1. Plot a histogram of GDP Per Capita.
2. Do the same but use a log10 x-axis.
3. Still on the log10 x-axis scale, try a density plot mapping continent to the fill of each density distribution, and reduce the opacity.
4. Still on the log10 x-axis scale, make a histogram faceted by continent *and* filled by continent. Facet with a single column (see `?facet_wrap` for help).
5. Save this figure to a 6x10 PDF file.

5.6 Publication-ready plots & themes

Let's make a plot we made earlier (life expectancy versus the log of GDP per capita with points colored by continent with lowess smooth curves overlaid without the standard error ribbon):

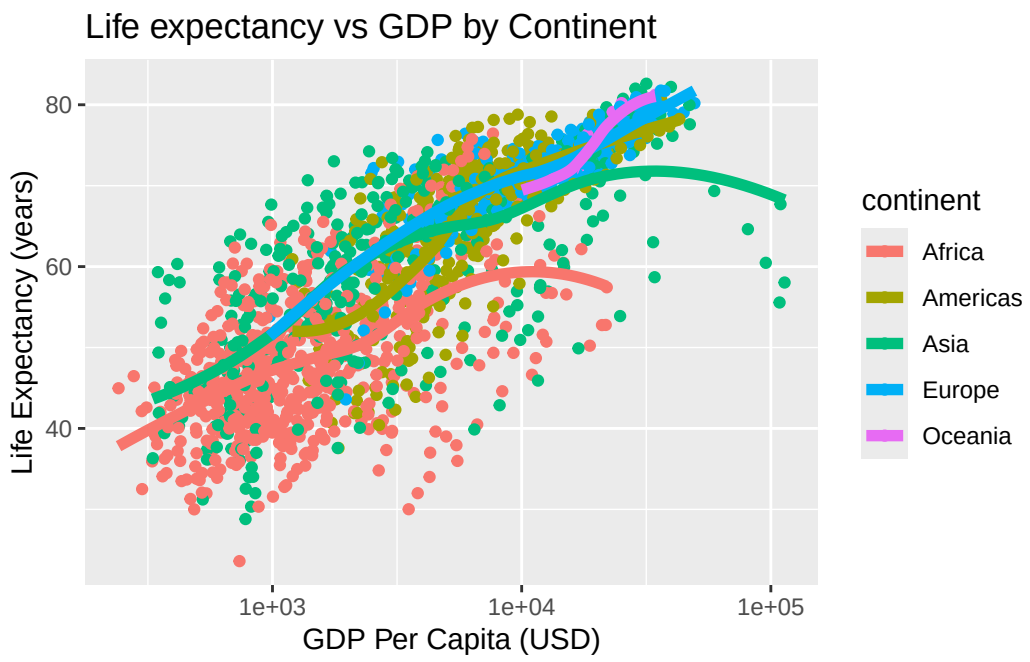
```
p <- ggplot(gm, aes(gdpPercap, lifeExp))
p <- p + scale_x_log10()
p <- p + aes(col=continent) + geom_point() + geom_smooth(lwd=2, se=FALSE)
```

Give the plot a title and axis labels:

```
p <- p + ggtitle("Life expectancy vs GDP by Continent")
p <- p + xlab("GDP Per Capita (USD)") + ylab("Life Expectancy (years)")
```

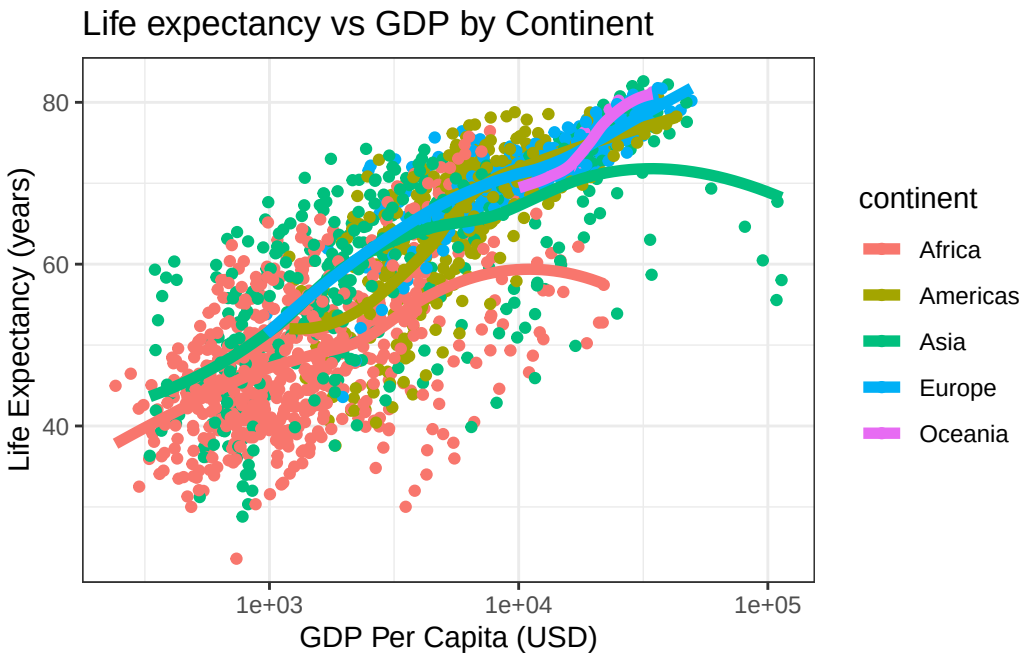
By default, the "gray" theme is the usual background (I've changed this course website to use the black and white background for all images).

```
p + theme_gray()
```



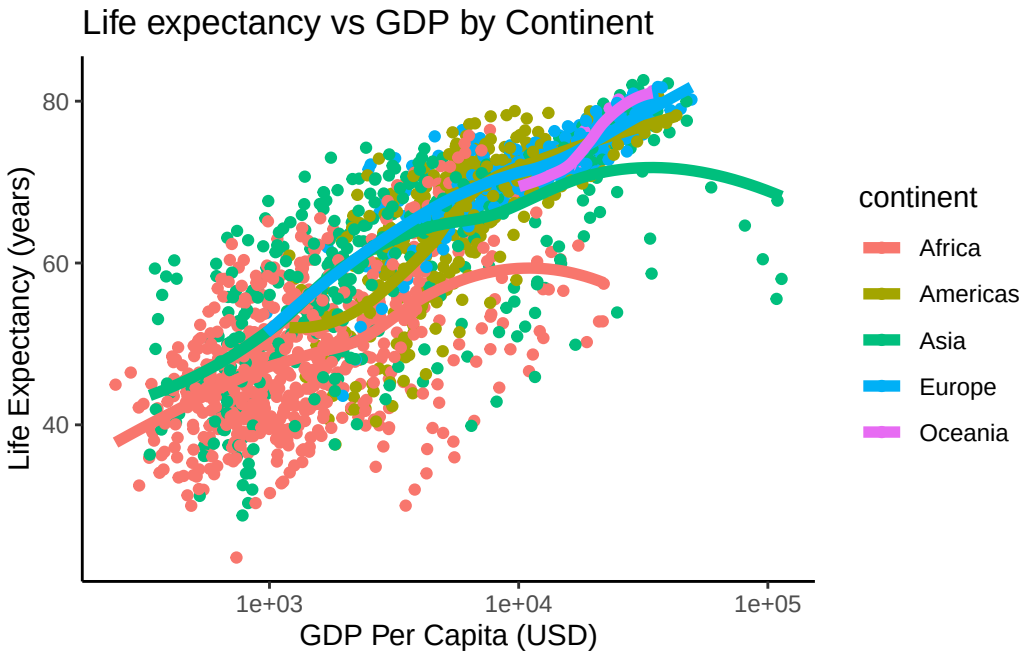
We could also get a black and white background:

```
p + theme_bw()
```



Or go a step further and remove the gridlines:

```
p + theme_classic()
```



Finally, there's another package that gives us lots of different themes. Install it if you don't have it already. Install all its dependencies along with it.

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

```
library(ggthemes)
p <- ggplot(gm, aes(gdpPercap, lifeExp))
p <- p + scale_x_log10()
p <- p + aes(col=continent) + geom_point() + geom_smooth(lwd=2, se=FALSE)
p + theme_excel()
p + theme_excel() + scale_colour_excel()
p + theme_gdocs() + scale_colour_gdocs()
p + theme_stata() + scale_colour_stata()
p + theme_wsj() + scale_colour_wsj()
p + theme_economist()
p + theme_fivethirtyeight()
p + theme_tufte()
```

6 Refresher: Tidy Exploratory Data Analysis

6.1 Chapter overview

This is a refresher chapter designed to be read after completing all the chapters that came before it.

The data and analyses here were inspired by the Tidy Tuesday project – a weekly social data project in R from the R for Data Science online learning community @R4DScommunity.

We're going to use two different data sets. One containing data on movie budgets and profits that was featured in a FiveThirtyEight article on horror movies and profits, and another with data on college majors and income from the American Community Survey.

Packages needed for this analysis are loaded below. If you don't have one of these packages installed, simply install it once using `install.packages("PackageName")`. A quick note on the **tidyverse** package (https://www.tidyverse.org/): the tidyverse is a collection of other packages that are often used together. When you install or load tidyverse, you also install and load all the packages that we've used previously: dplyr, tidyr, ggplot2, as well as several others. Because we'll be using so many different packages from the tidyverse collection, it's more efficient load this "meta-package" rather than loading each individual package separately.

```
library(tidyverse)
library(ggrepel)
library(scales)
library(lubridate)
```

I'll demonstrate some functionality from these other packages. They're handy to have installed, but are not strictly required.

```
library(plotly)
library(DT)
```

6.2 Horror Movies & Profit

6.2.1 About the data

The raw data can be downloaded here: **movies.csv**.

This data was featured in the FiveThirtyEight article, "Scary Movies Are The Best Investment In Hollywood".

> "Horror movies get nowhere near as much draw at the box office as the big-time summer blockbusters or action/adventure movies – the horror genre accounts for only 3.7 percent of the total box-office haul this year – but there's a huge incentive for studios to continue pushing them out.
>
> The return-on-investment potential for horror movies is absurd. For example, "Paranormal Activity" was made for $450,000 and pulled in $194 million – 431 times the original budget. That's an extreme, I-invested-in-Microsoft-when-Bill-Gates-was-working-in-a-garage case, but it's not rare. And that's what makes horror such a compelling genre to produce."

– Quote from Walt Hickey for fivethirtyeight article.

Data dictionary (data from the-numbers.com):

Header	Description
`release_date`	month-day-year
`movie`	Movie title
`production_budget`	Money spent to create the film
`domestic_gross`	Gross revenue from USA
`worldwide_gross`	Gross worldwide revenue
`distributor`	The distribution company
`mpaa_rating`	Appropriate age rating by the US-based rating agency
`genre`	Film category

6.2.2 Import and clean

If you haven't already loaded the packages we need, go ahead and do that now.

```
library(tidyverse)
library(ggrepel)
library(scales)
library(lubridate)
```

Now, use the `read_csv()` function from **readr** (loaded when you load **tidyverse**), to read in the **movies.csv** dataset into a new object called `mov_raw`.

```
mov_raw <- read_csv("data/movies.csv")
mov_raw
```

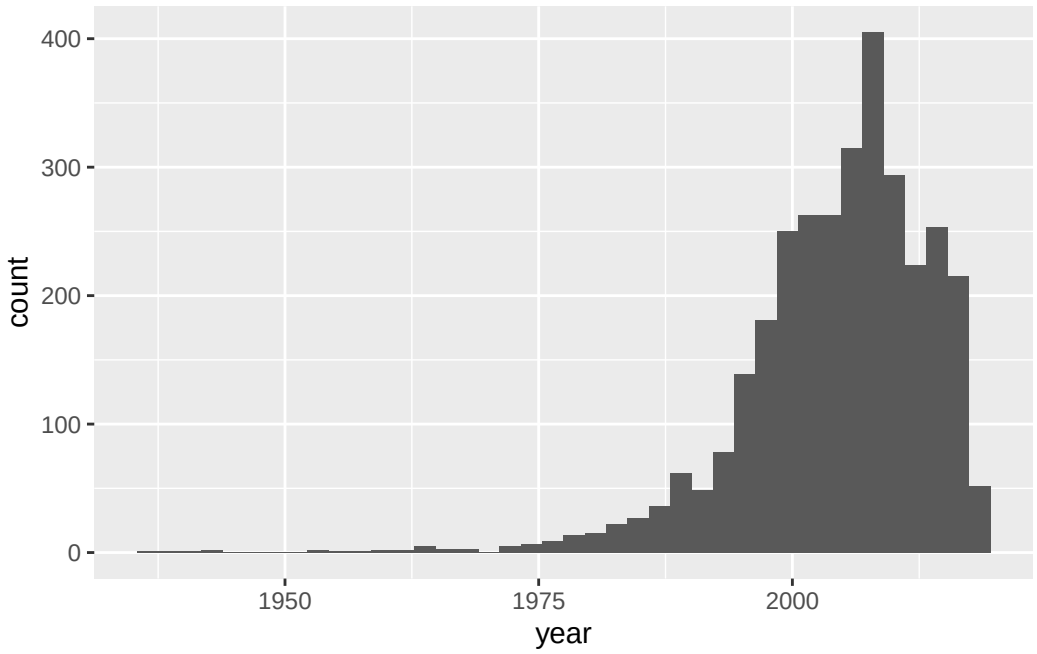
Let's clean up the data a bit. Remember, construct your pipeline one step at a time first. Once you're happy with the result, assign the results to a new object, `mov`.

- Get rid of the blank `X1` Variable.
- Change release date into an actual date.
- Calculate the return on investment as the `worldwide_gross/production_budget`.
- Calculate the percentage of total gross as domestic revenue.
- Get the year, month, and day out of the release date.
- Remove rows where the revenue is $0 (unreleased movies, or data integrity problems), and remove rows missing information about the distributor. Go ahead and remove any data where the rating is unavailable also.

```
mov <- mov_raw |>
  select(-...1) |>
  mutate(release_date = mdy(release_date)) |>
  mutate(roi = worldwide_gross / production_budget) |>
  mutate(pct_domestic = domestic_gross / worldwide_gross) |>
  mutate(year = year(release_date)) |>
  mutate(month = month(release_date, label = TRUE)) |>
  mutate(day = wday(release_date, label = TRUE)) |>
  arrange(desc(release_date)) |>
  filter(worldwide_gross > 0) |>
  filter(!is.na(distributor)) |>
  filter(!is.na(mpaa_rating))
mov
```

Let's take a look at the distribution of release date.

```
ggplot(mov, aes(year)) + geom_histogram(bins=40)
```



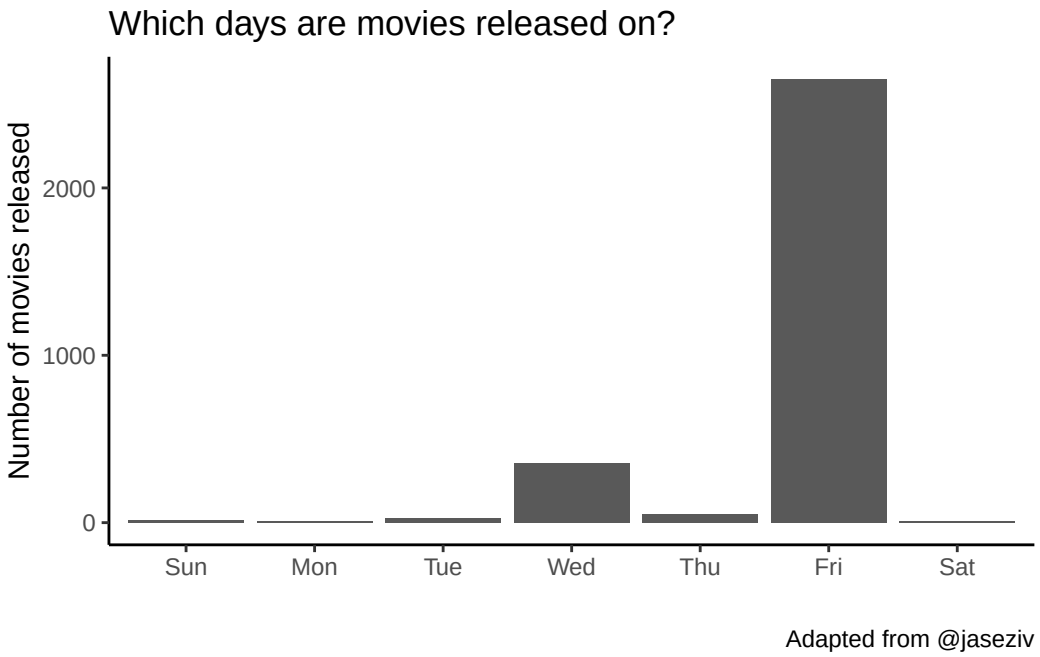
There doesn't appear to be much documented berfore 1975, so let's restrict (read: filter) the dataset to movies made since 1975. Also, we're going to be doing some analyses by year, and since the data for 2018 is still incomplete, let's remove all of 2018. Let's get anything produced in 1975 and after (`>=1975`) but before 2018 (`<2018`). Add the final filter statement to the assignment, and make the plot again.

```
mov <- mov_raw |>
  select(-...1) |>
  mutate(release_date = mdy(release_date)) |>
  mutate(roi = worldwide_gross / production_budget) |>
  mutate(pct_domestic = domestic_gross / worldwide_gross) |>
  mutate(year = year(release_date)) |>
  mutate(month = month(release_date, label = TRUE)) |>
  mutate(day = wday(release_date, label = TRUE)) |>
  arrange(desc(release_date)) |>
  filter(worldwide_gross > 0) |>
  filter(!is.na(distributor)) |>
  filter(!is.na(mpaa_rating)) |>
  filter(year>=1975 & year <2018)
mov
```

6.2.3 Exploratory Data Analysis

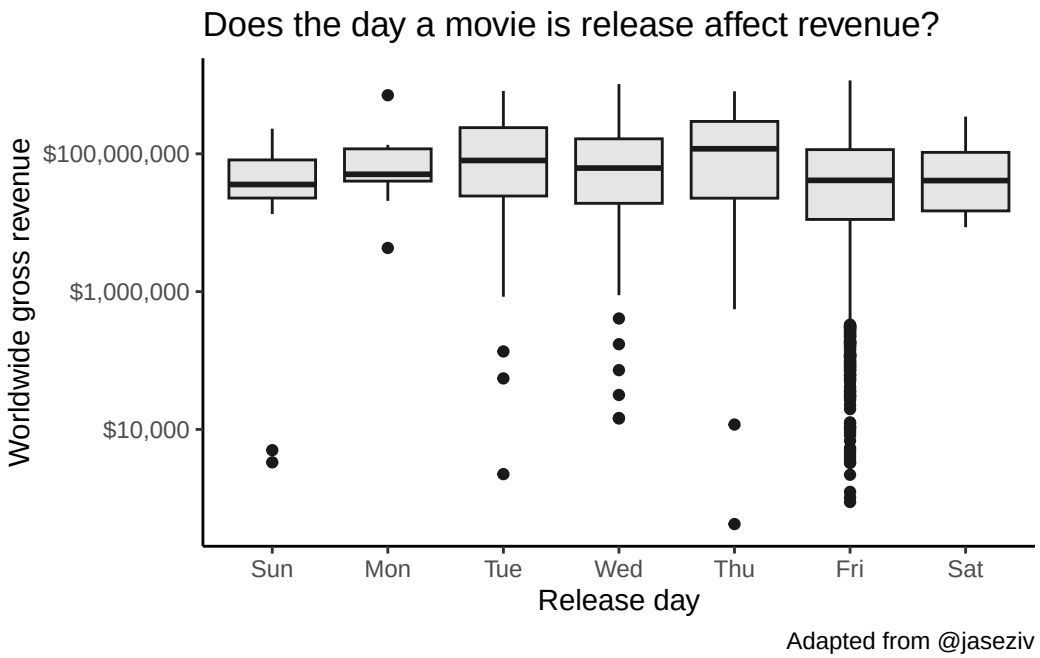
Which days are movies released on? The dplyr `count()` function counts the number of occurances of a particular variable. It's shorthand for a `group_by()` followed by `summarize(n=n())`. The `geom_col()` makes a bar chart where the height of the bar is the count of the number of cases, y, at each x position. Feel free to add labels if you want.

```
mov |>
  count(day, sort=TRUE) |>
  ggplot(aes(day, n)) +
  geom_col() +
  labs(x="", y="Number of movies released",
       title="Which days are movies released on?",
       caption="Adapted from @jaseziv") +
  theme_classic()
```



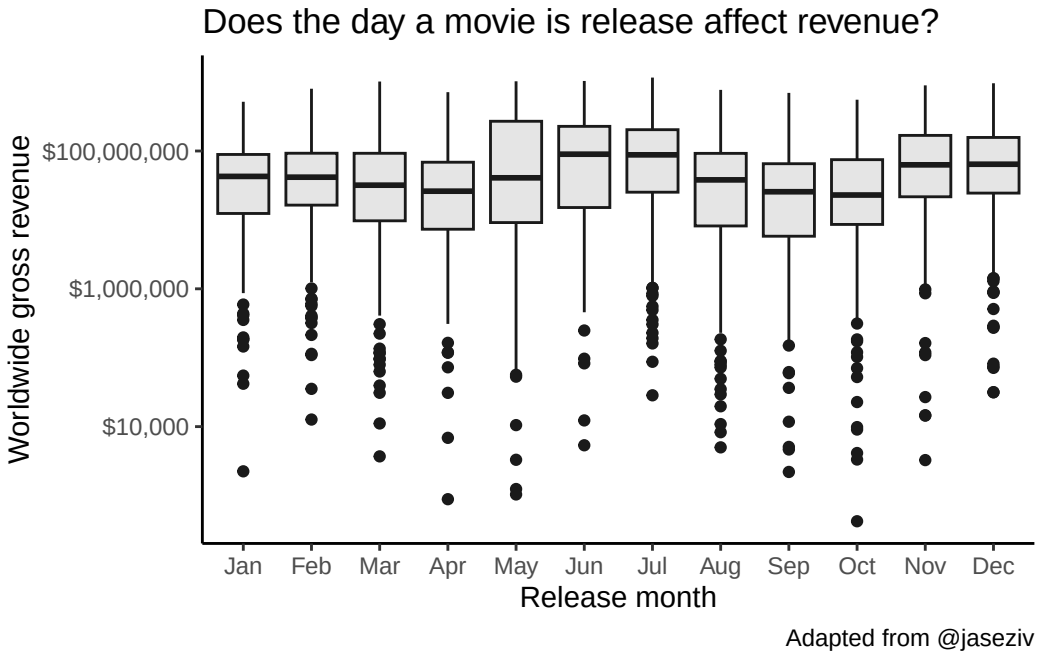
Exercise 1

Does the day a movie is release affect revenue? Make a boxplot showing the worldwide gross revenue for each day.



What about month? Just swap `day` for `month` in the code.

```
mov |>
  ggplot(aes(month, worldwide_gross)) +
  geom_boxplot(col="gray10", fill="gray90") +
  scale_y_log10(labels=dollar_format()) +
  labs(x="Release month",
       y="Worldwide gross revenue",
       title="Does the day a movie is release affect revenue?",
       caption="Adapted from @jaseziv") +
  theme_classic()
```



We could also get a quantitative look at the average revenue by day using a group-by summarize operation:

```
mov |>
  group_by(day) |>
  summarize(rev=mean(worldwide_gross))
```

```
# A tibble: 7 x 2
  day          rev
  <ord>      <dbl>
1 Sun    70256412.
2 Mon   141521289.
3 Tue   177233110.
4 Wed   130794183.
5 Thu   194466996.
6 Fri    90769834.
7 Sat    89889497.
```

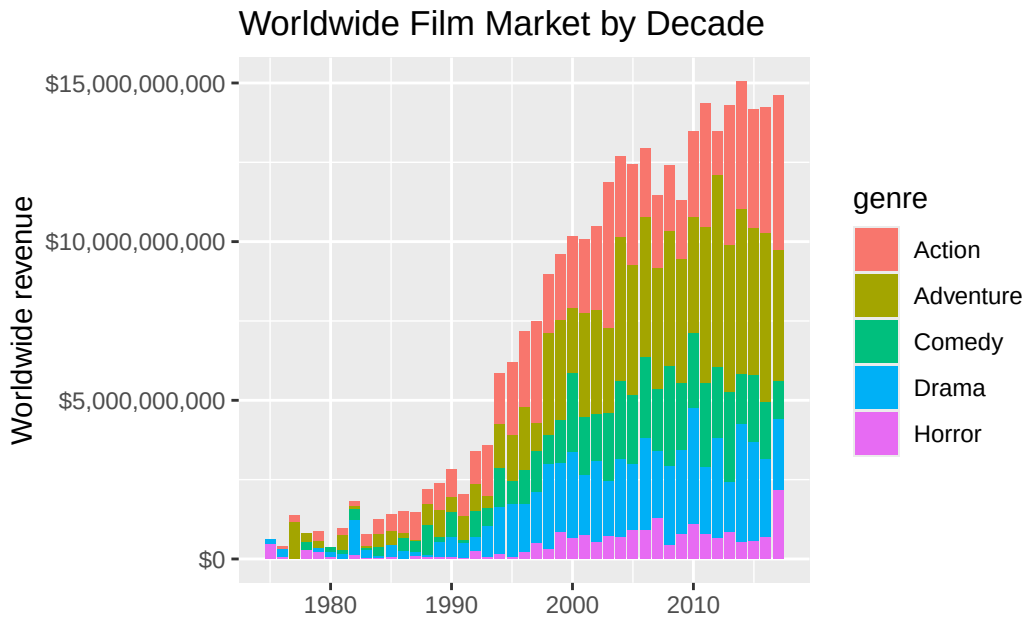
It looks like summer months and holiday months at the end of the year fare well. Let's look at a table and run a regression analysis.

```
mov |>
  group_by(month) |>
  summarize(rev=mean(worldwide_gross))
```

```
mov |>
  mutate(month=factor(month, ordered=FALSE)) |>
  lm(worldwide_gross~month, data=_) |>
  summary()
```

What does the worldwide movie market look like by decade? Let's first group by year and genre and compute the sum of the worldwide gross revenue. After we do that, let's plot a barplot showing year on the x-axis and the sum of the revenue on the y-axis, where we're passing the genre variable to the `fill` aesthetic of the bar.

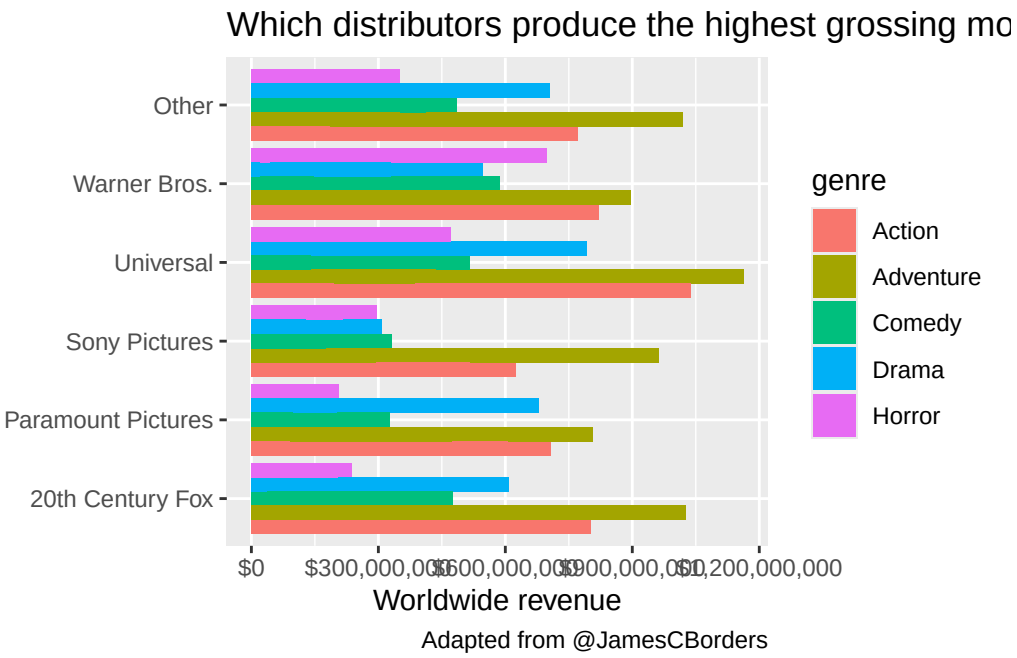
```
mov |>
  group_by(year, genre) |>
  summarize(revenue=sum(worldwide_gross)) |>
  ggplot(aes(year, revenue)) +
  geom_col(aes(fill=genre)) +
  scale_y_continuous(labels=dollar_format()) +
  labs(x="", y="Worldwide revenue", title="Worldwide Film Market by Decade")
```



Which distributors produce the highest grossing movies by genre? First let's lump all dis-

tributors together into 5 major distributors with the most movies, lumping all others into an "Other" category. The `fct_lump` function from the **forcats** package (loaded with **tidyverse**) will do this for you. Take a look at just that result first. Then let's plot a `geom_col()`, which plots the actual value of the thing we put on the y-axis (worldwide gross revenue in this case). Because `geom_col()` puts all the values on top of one another, the highest value will be the one displayed. Let's add `position="dodge"` so they're beside one another instead of stacked. We can continue to add additional things to make the plot pretty. I like the look of this better when we flip the coordinate system with `coord_flip()`.

```
mov |>
  mutate(distributor=fct_lump(distributor, 5)) |>
  ggplot(aes(distributor, worldwide_gross)) + geom_col(aes(fill=genre), position="dodge")
  scale_y_continuous(labels = dollar_format()) +
  labs(x="",
       y="Worldwide revenue",
       title="Which distributors produce the highest grossing movies by genre?",
       caption="Adapted from @JamesCBorders") +
  coord_flip()
```



It looks like Universal made the highest-grossing action and adventure movies, while Warner Bros made the highest grossing horror movies.

But what about return on investment?

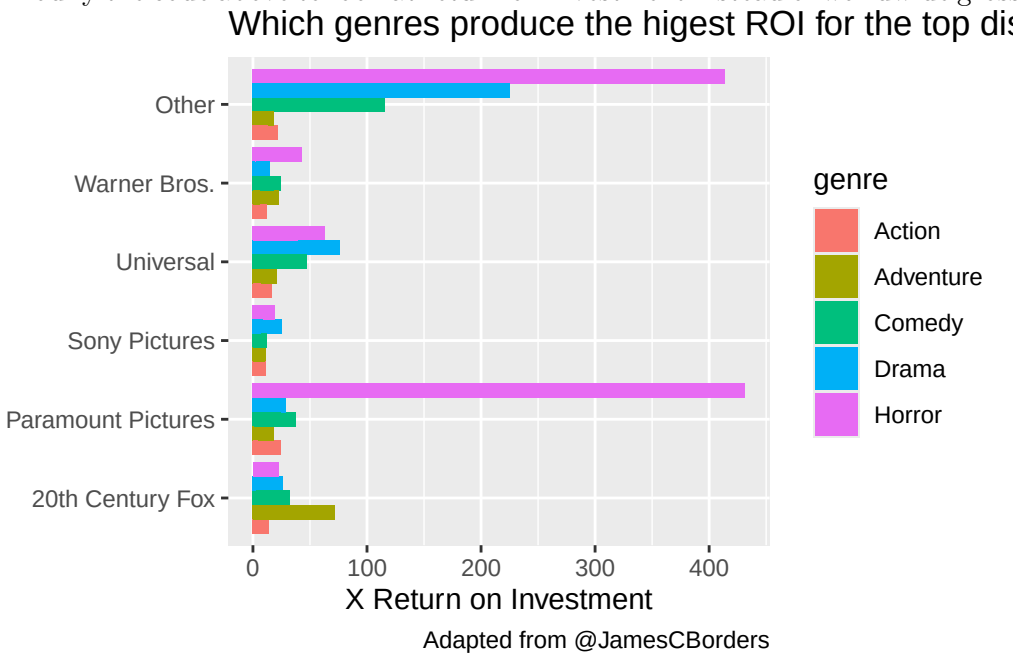
```
mov |>
  group_by(genre) |>
  summarize(roi=mean(roi))
```

```
# A tibble: 5 x 2
  genre       roi
  <chr>     <dbl>
1 Action     2.82
2 Adventure  3.60
3 Comedy     3.48
4 Drama      3.40
5 Horror    11.2
```

It looks like horror movies have overwhelmingly the highest return on investment. Let's look at this across the top distributors.

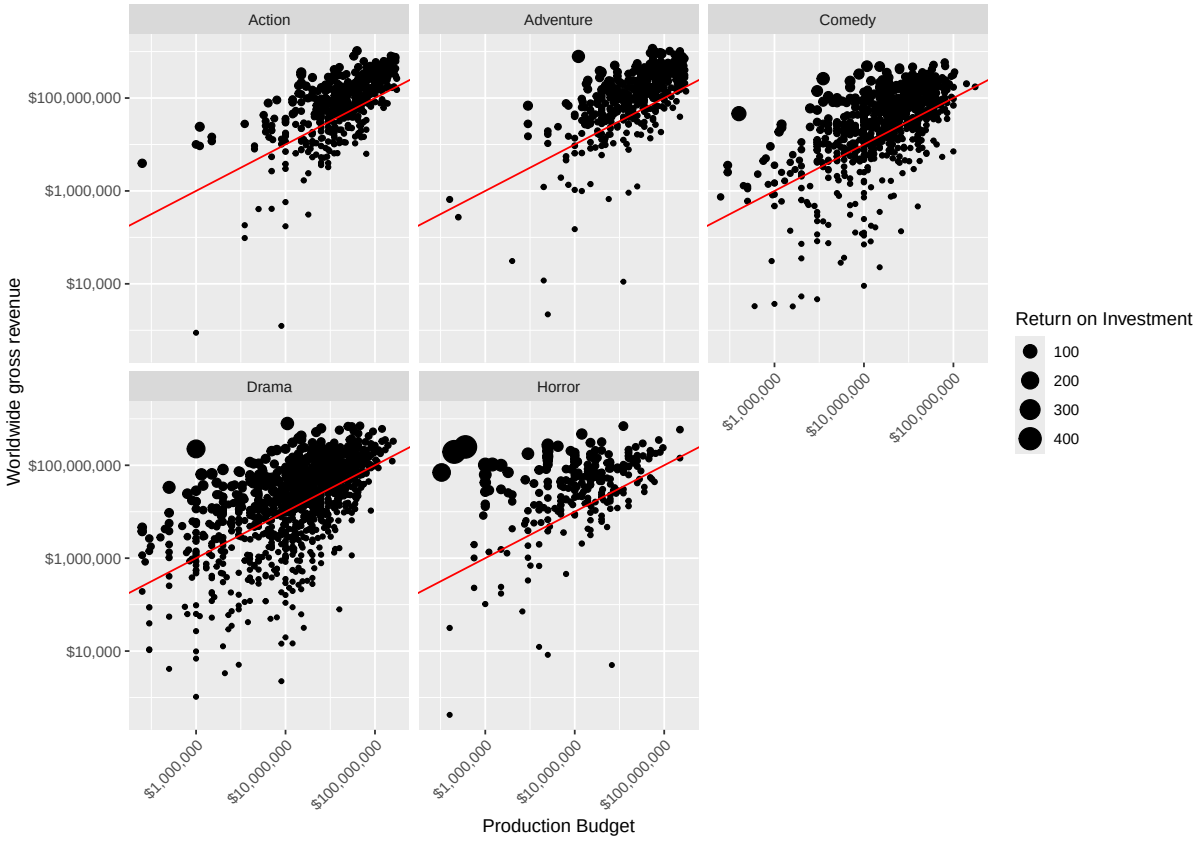
Exercise 2

Modify the code above to look at return on investment instead of worldwide gross revenue.



Let's make a scatter plot showing the worldwide gross revenue over the production budget. Let's make the size of the point relative to the ROI. Let's add a "breakeven" line that has a slope of 1 and a y-intercept of zero. Let's facet by genre.

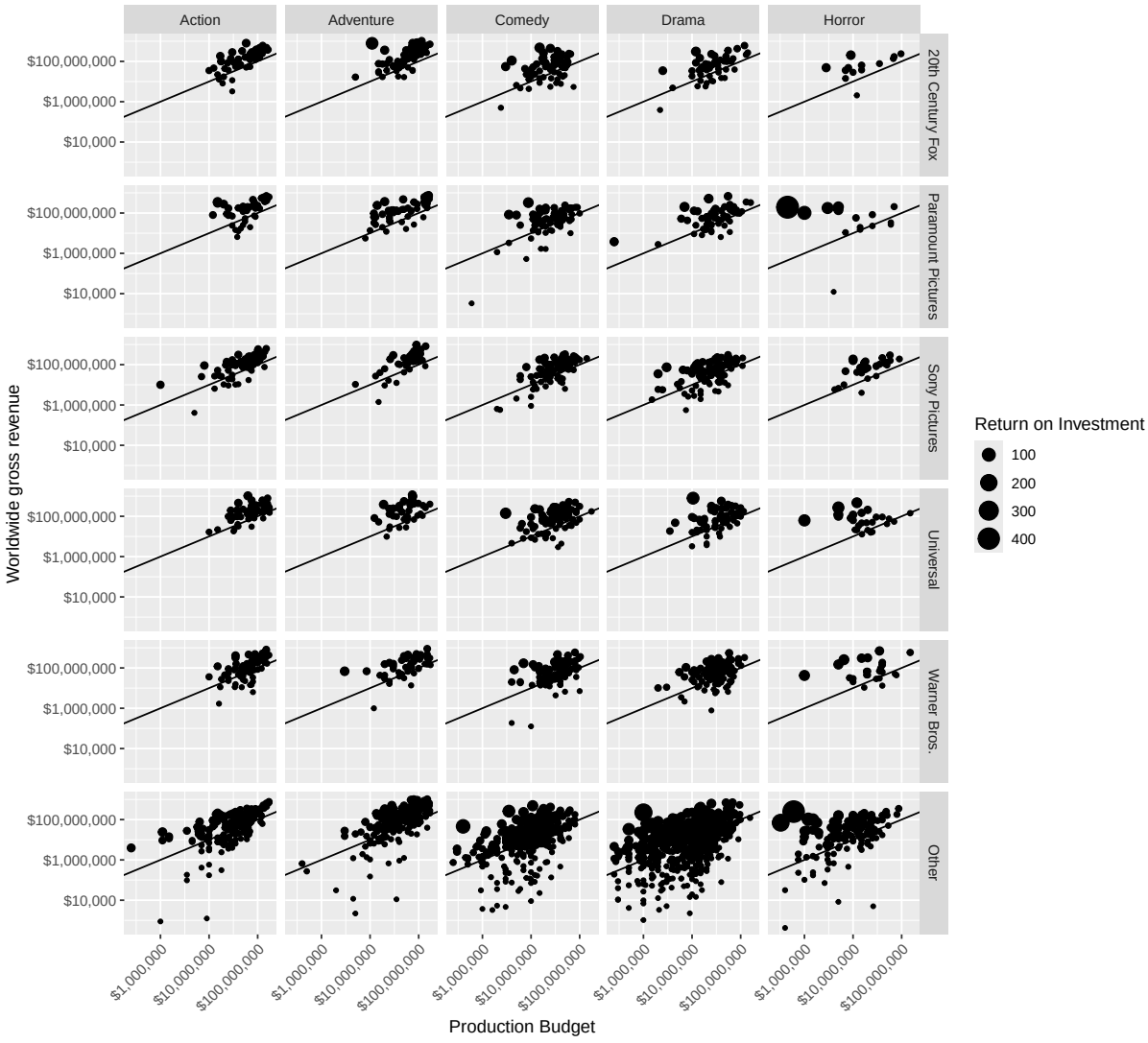
```
mov |>
  ggplot(aes(production_budget, worldwide_gross)) +
  geom_point(aes(size = roi)) +
  geom_abline(slope = 1, intercept = 0, col = "red") +
  facet_wrap( ~ genre) +
  scale_x_log10(labels = dollar_format()) +
  scale_y_log10(labels = dollar_format()) +
  theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
  labs(x = "Production Budget",
       y = "Worldwide gross revenue",
       size = "Return on Investment")
```



Generally most of the points lie above the "breakeven" line. This is good – if movies weren't profitable they wouldn't keep making them. Proportionally there seem to be many more larger points in the Horror genre, indicative of higher ROI.

Let's create a faceted grid showing distributor by genre. Paramount and Other distributors have the largest share of low-budget high-revenue horror films.

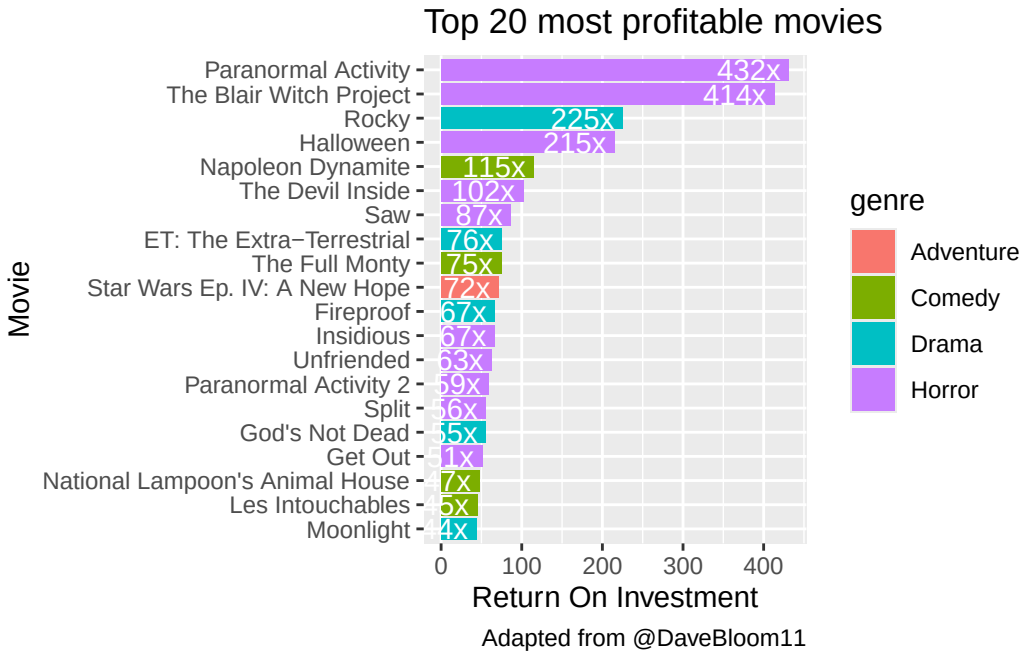
```r
mov |>
  mutate(distributor = fct_lump(distributor, 5)) |>
  ggplot(aes(production_budget, worldwide_gross)) +
  geom_point(aes(size = roi)) +
  geom_abline(slope = 1, intercept = 0) +
  facet_grid(distributor ~ genre) +
  scale_x_log10(labels = dollar_format()) +
  scale_y_log10(labels = dollar_format()) +
  theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
  labs(x = "Production Budget",
       y = "Worldwide gross revenue",
       size = "Return on Investment")
```



What were those super profitable movies? Looks like they're mostly horror movies. One thing that's helpful to do here is to make movies a factor variable, reordering its levels by the median ROI. Look at the help for `?fct_reorder` for this. I also like to `coord_flip()` this plot.

```
mov |>
  arrange(desc(roi)) |>
  head(20) |>
  mutate(movie=fct_reorder(movie, roi)) |>
  ggplot(aes(movie, roi)) +
  geom_col(aes(fill=genre)) +
  labs(x="Movie",
```

```
      y="Return On Investment",
      title="Top 20 most profitable movies",
      caption="Adapted from @DaveBloom11") +
  coord_flip() +
  geom_text(aes(label=paste0(round(roi), "x "), hjust=1), col="white")
```



It might be informative to run the same analysis for movies that had either exclusive US distribution, or no US distribution at all. We could simply filter for movies with 100% of the revenue coming from domestic gross revenue US only, or 0% from domestic (no US distribution). Just add a filter statement in the pipeline prior to plotting.

```
mov |>
  filter(pct_domestic==1) |>
  arrange(desc(roi)) |>
  head(20) |>
  mutate(movie=fct_reorder(movie, roi)) |>
  ggplot(aes(movie, roi)) +
  geom_col(aes(fill=genre)) +
  labs(x="Movie",
       y="Return On Investment",
       title="Top 20 most profitable movies with US-only distribution",
       caption="Adapted from @DaveBloom11") +
```

```
  coord_flip() +
  geom_text(aes(label=paste0(round(roi), "x "), hjust=1), col="white")
```

```
mov |>
  filter(pct_domestic==0) |>
  arrange(desc(roi)) |>
  head(20) |>
  mutate(movie=fct_reorder(movie, roi)) |>
  ggplot(aes(movie, roi)) +
  geom_col(aes(fill=genre)) +
  labs(x="Movie",
       y="Return On Investment",
       title="Top 20 most profitable movies with no US distribution",
       caption="Adapted from @DaveBloom11") +
  coord_flip()
```

What about movie ratings? R-rated movies have a lower average revenue but ROI isn't substantially less. The `n()` function is a helper function that just returns the number of rows for each group in a grouped data frame. We can see that while G-rated movies have the highest mean revenue, there were relatively few of them produced, and had a lower total revenue. There were more R-rated movies, but PG-13 movies really drove the total revenue worldwide.

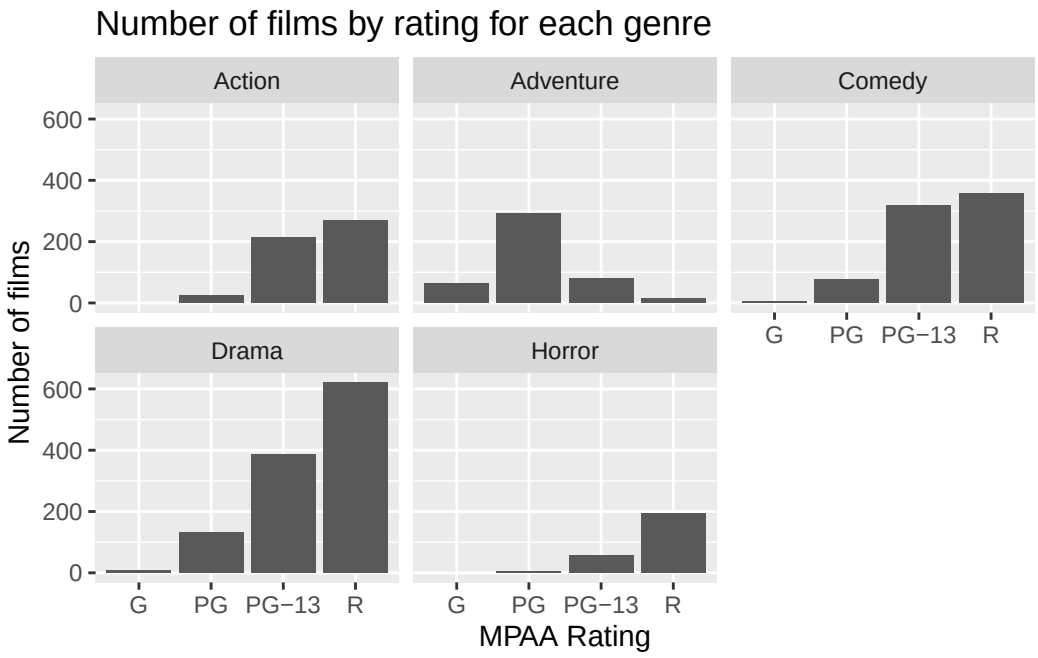
```
mov |>
  group_by(mpaa_rating) |>
  summarize(
    meanrev = mean(worldwide_gross),
    totrev = sum(worldwide_gross),
    roi = mean(roi),
    number = n()
  )
```

```
# A tibble: 4 x 5
  mpaa_rating    meanrev       totrev   roi number
  <chr>            <dbl>        <dbl> <dbl>  <int>
1 G           189913348   13863674404  4.42     73
2 PG          147227422.  78324988428  4.64    532
3 PG-13       113477939. 120173136920  3.06   1059
4 R            63627931.  92451383780  4.42   1453
```

Are there fewer R-rated movies being produced? Not really. Let's look at the overall number

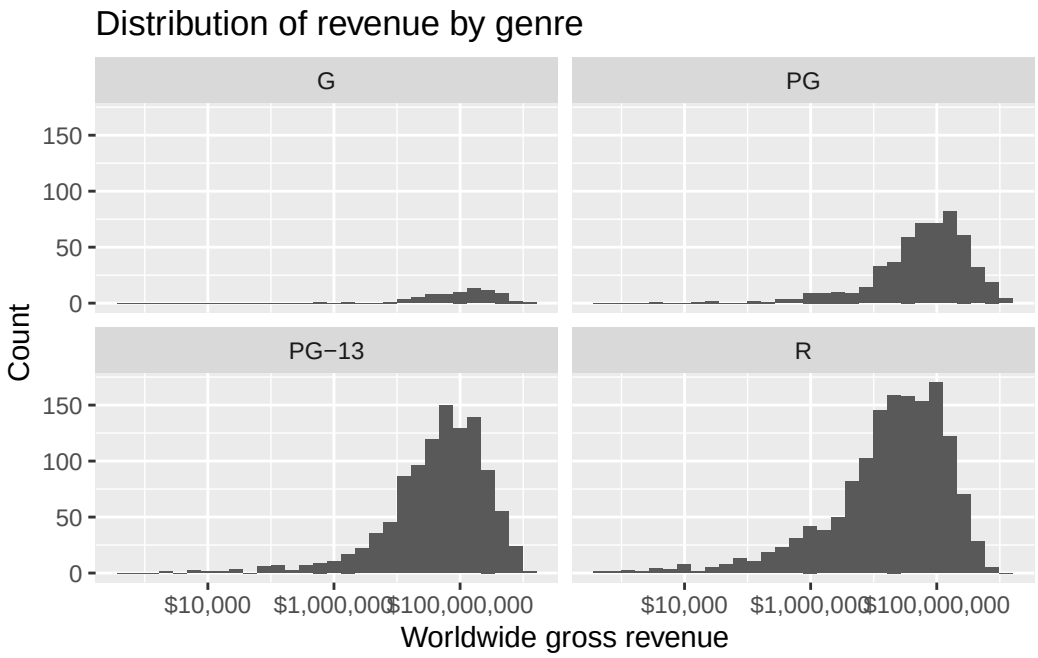
of movies with any particular rating faceted by genre.

```
mov |>
  count(mpaa_rating, genre) |>
  ggplot(aes(mpaa_rating, n)) +
  geom_col() +
  facet_wrap(~genre) +
  labs(x="MPAA Rating",
       y="Number of films",
       title="Number of films by rating for each genre")
```

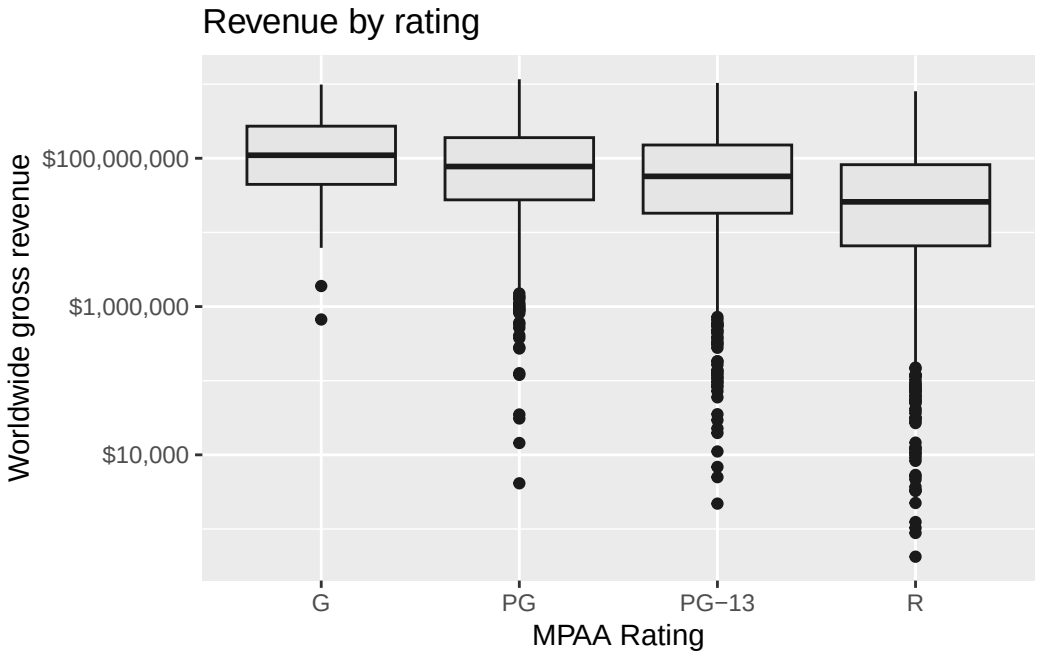


What about the distributions of ratings?

```
mov |>
  ggplot(aes(worldwide_gross)) +
  geom_histogram() +
  facet_wrap(~mpaa_rating) +
  scale_x_log10(labels=dollar_format()) +
  labs(x="Worldwide gross revenue",
       y="Count",
       title="Distribution of revenue by genre")
```

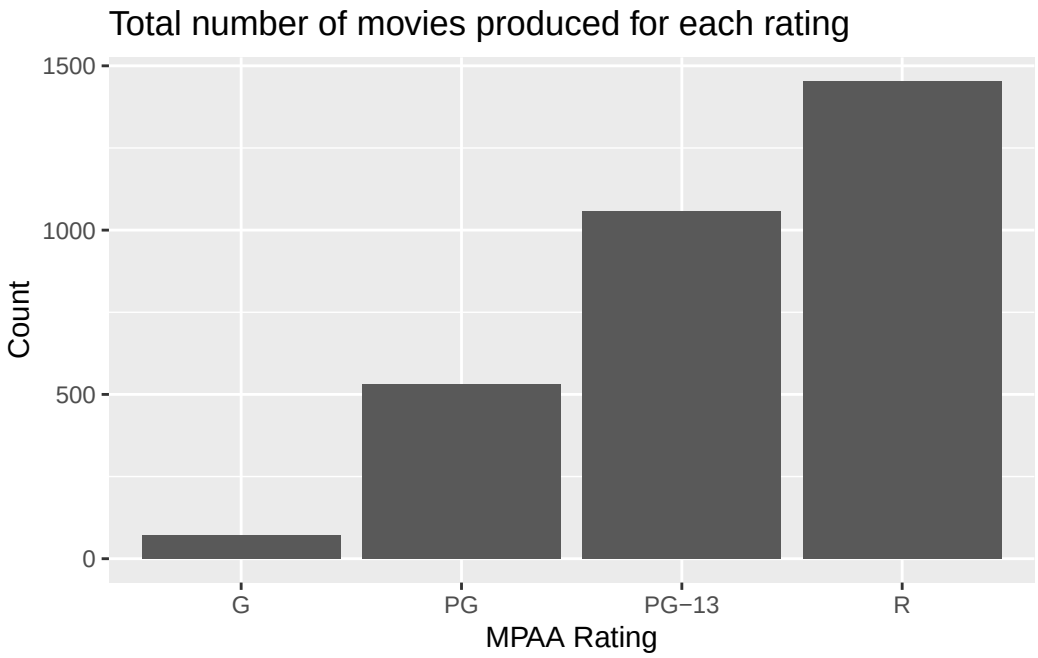



```
mov |>
  ggplot(aes(mpaa_rating, worldwide_gross)) +
  geom_boxplot(col="gray10", fill="gray90") +
  scale_y_log10(labels=dollar_format()) +
  labs(x="MPAA Rating", y="Worldwide gross revenue", title="Revenue by rating")
```

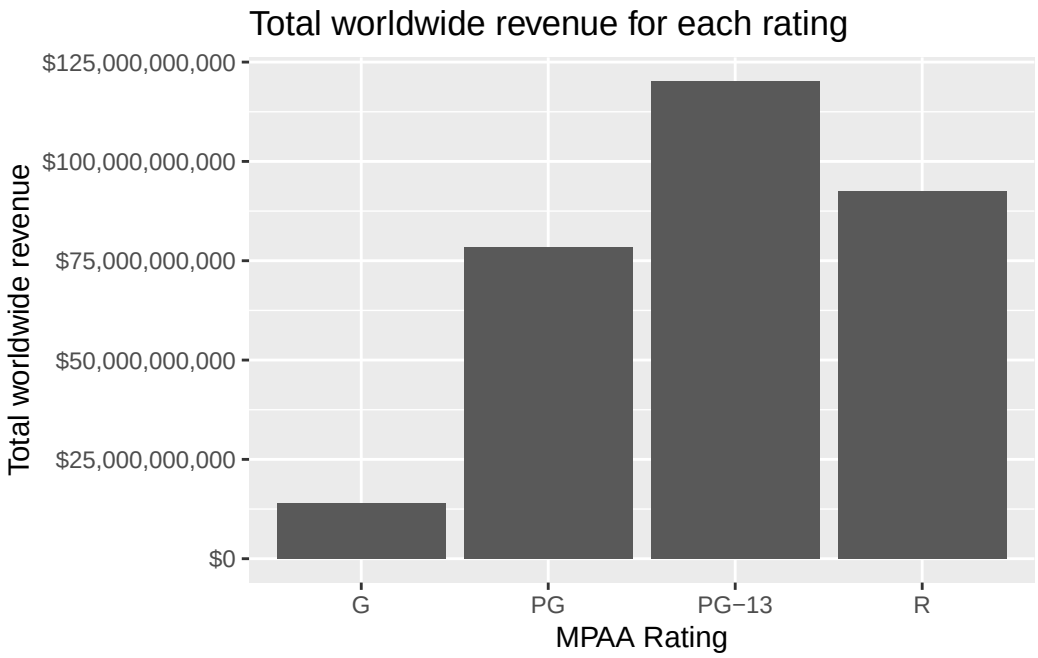


But, dont be fooled. Yes, on average G-rated movies look to perform better. But there aren't that many of them being produced, and they aren't bringing in the lions share of revenue.

```
mov |>
  count(mpaa_rating) |>
  ggplot(aes(mpaa_rating, n)) +
  geom_col() +
  labs(x="MPAA Rating",
       y="Count",
       title="Total number of movies produced for each rating")
```




```
mov |>
  group_by(mpaa_rating) |>
  summarize(total_revenue=sum(worldwide_gross)) |>
  ggplot(aes(mpaa_rating, total_revenue)) +
  geom_col() +
  scale_y_continuous(label=dollar_format()) +
  labs(x="MPAA Rating",
       y="Total worldwide revenue",
       title="Total worldwide revenue for each rating")
```



6.2.4 Join to IMDB reviews

Look back at the dplyr reference on joins. An inner join lets you take two tables, match by a common column (or columns), and return rows with an entry in both, returning all columns in each table. I've downloaded all the data underlying IMDB (imdb.com/interfaces), and created a reduced dataset having ratings for all the movies in IMDB. Let's join the movie data we have here with IMDB ratings. Download the data here: **movies_imdb.csv**. Once you've downloaded it, read it in with `read_csv()`:

```
imdb <- read_csv("data/movies_imdb.csv")
imdb
```

There are **177,519** movies in this dataset. There are **3,117** movies in the data we've already been using. Let's see how many we have that intersect in both:

```
movimdb <- inner_join(mov, imdb, by="movie")
movimdb
```

It turns out there are only **2,591** rows in the joined dataset. That's because there were some rows in `mov` that weren't in `imdb`, and vice versa. Some of these are truly cases where there isn't an entry in one. Others are cases where it's `Star Wars Ep. I: The Phantom Menace` in one dataset but `Star Wars: Episode I - The Phantom Menace` in another, or `Mr. & Mrs.`

`Smith` versus `Mr. and Mrs. Smith`. Others might be ascii versus unicode text incompatibility, e.g. the hyphen "-" versus the endash, "–".

Now that you have the datasets joined, try a few more exercises!

Exercise 3

Separately for each MPAA rating, display the mean IMDB rating and mean number of votes cast.

```
# A tibble: 4 x 3
  mpaa_rating meanimdb meanvotes
  <chr>          <dbl>     <dbl>
1 G               6.54   132015.
2 PG              6.31    81841.
3 PG-13           6.25   102740.
4 R               6.58   107575.
```

Exercise 4

Do the same but for each movie genre.

```
# A tibble: 5 x 3
  genre     meanimdb meanvotes
  <chr>        <dbl>     <dbl>
1 Action        6.28   154681.
2 Adventure     6.27   130027.
3 Comedy        6.08    71288.
4 Drama         6.88    91101.
5 Horror        5.90    89890.
```

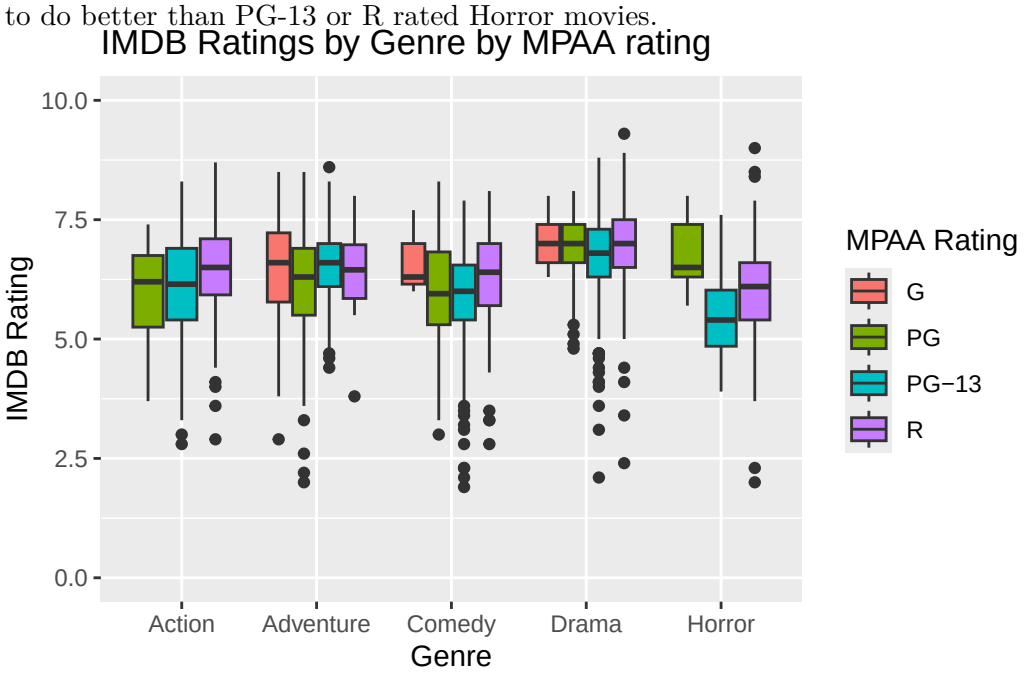
Exercise 5

Do the same but for each distributor, after lumping distributors in a mutate statement to the top 4 distributors, as we've done before.

```
# A tibble: 5 x 3
  distributor        meanimdb meanvotes
  <fct>                 <dbl>     <dbl>
1 Paramount Pictures     6.44   130546.
2 Sony Pictures          6.25   111913.
3 Universal              6.44   130028.
4 Warner Bros.           6.37   133997.
5 Other                  6.46    86070.
```

Exercise 6

Create a boxplot visually summarizing what you saw in #1 and #2 above. That is, show the distribution of IMDB ratings for each genre, but map the fill aesthetic for the boxplot onto the MPAA rating. Here we can see that Dramas tend to get a higher IMDB rating overall. Across most categories R rated movies fare better. We also see from this that there are no Action or Horror movies rated G (understandably!). In fact, after this I actually wanted to see what the "Horror" movies were having a PG rating that seemed to do better than PG-13 or R rated Horror movies.

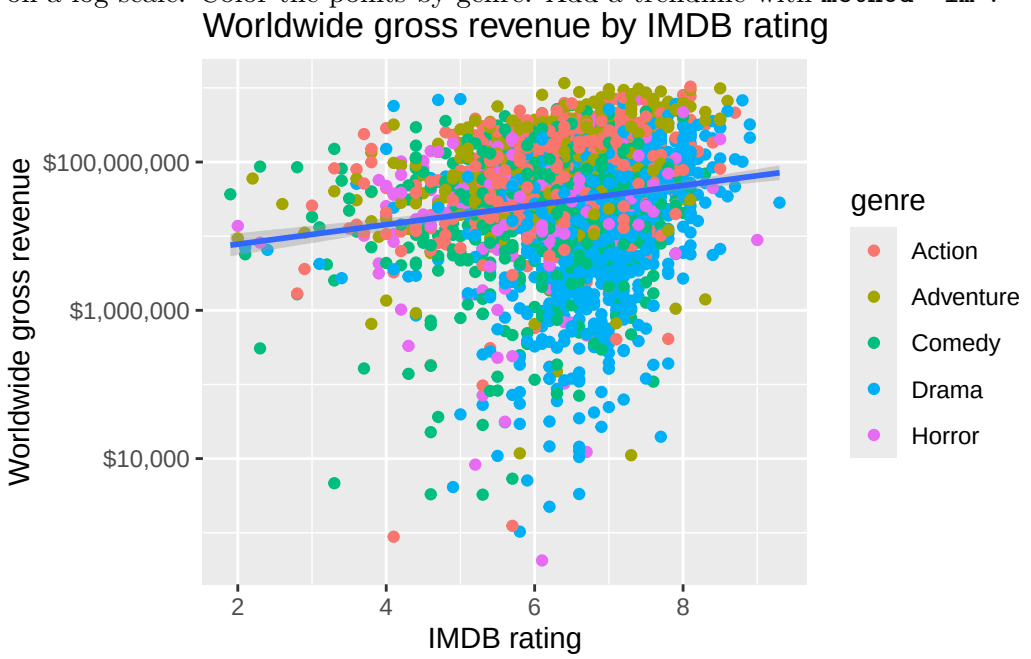



```
movimdb |>
  filter(mpaa_rating=="PG", genre=="Horror") |>
  select(release_date, movie, worldwide_gross, imdb, votes)
```

```
# A tibble: 5 x 5
  release_date movie                     worldwide_gross  imdb  votes
  <date>       <chr>                               <dbl> <dbl>  <dbl>
1 2015-10-16   Goosebumps                      158905324   6.3  67744
2 1983-06-24   Twilight Zone: The Movie         29500000   6.5  29313
3 1982-06-04   Poltergeist                     121706019   7.4 124178
4 1978-06-16   Jaws 2                          208900376   5.7  61131
5 1975-06-20   Jaws                            470700000   8   492525
```

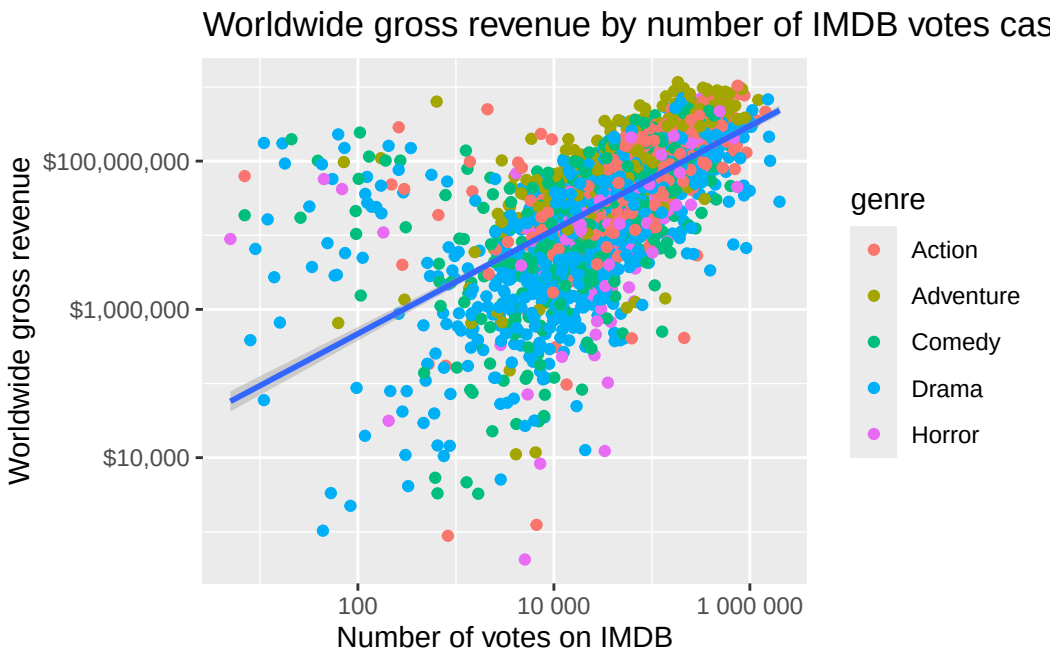
Exercise 7

Create a scatter plot of worldwide gross revenue by IMDB rating, with the gross revenue on a log scale. Color the points by genre. Add a trendline with `method="lm"`.



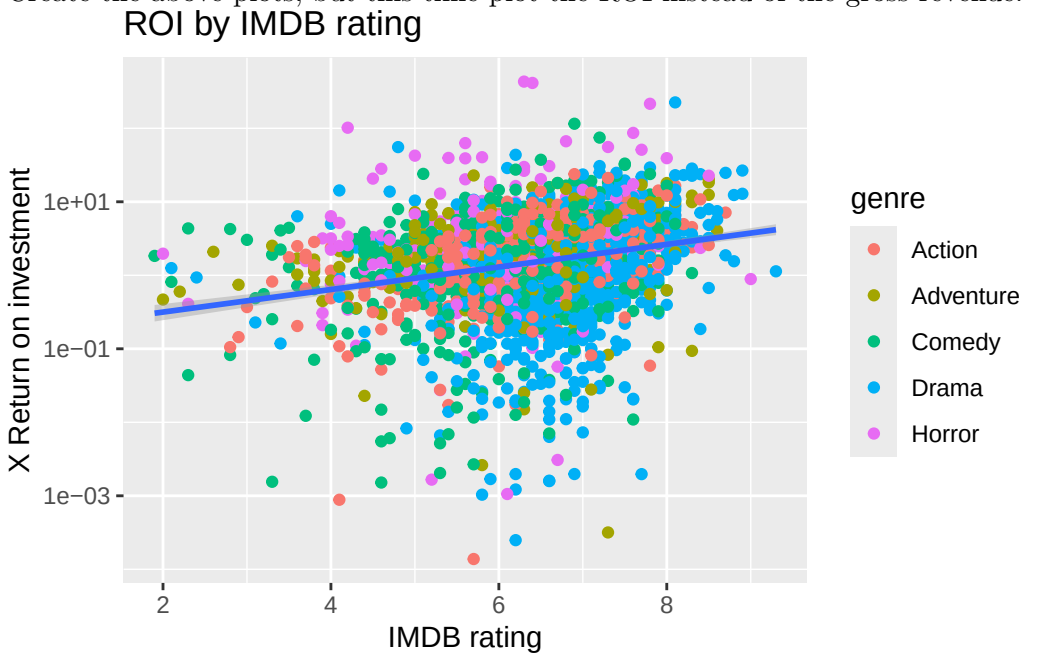
Exercise 8

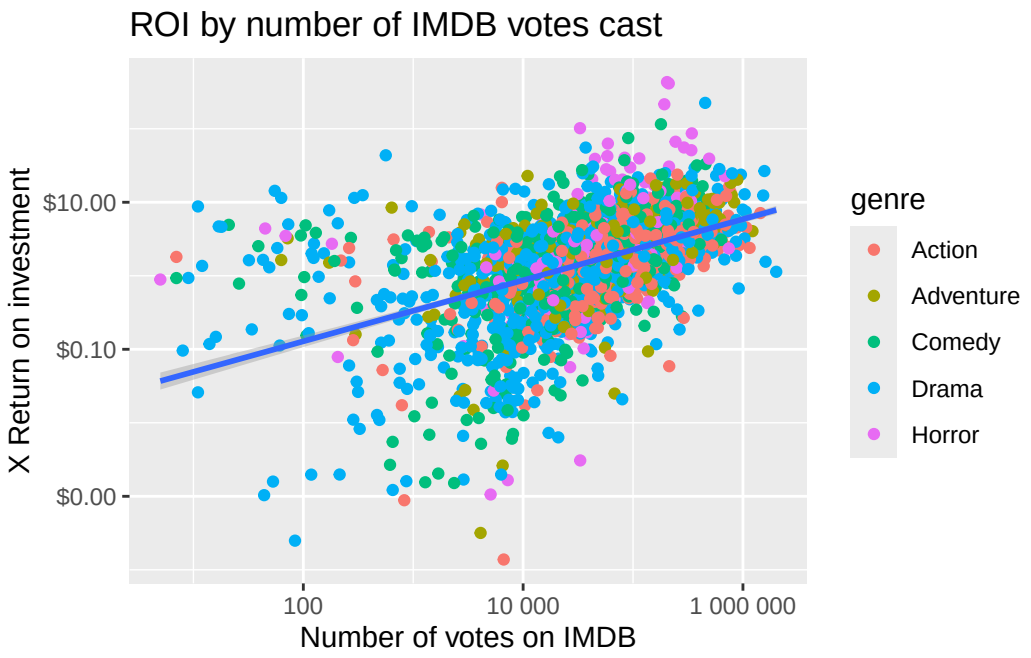
Create the same plot, this time putting the number of votes on the x-axis, and make both the x and y-axes log scale.



Exercise 9

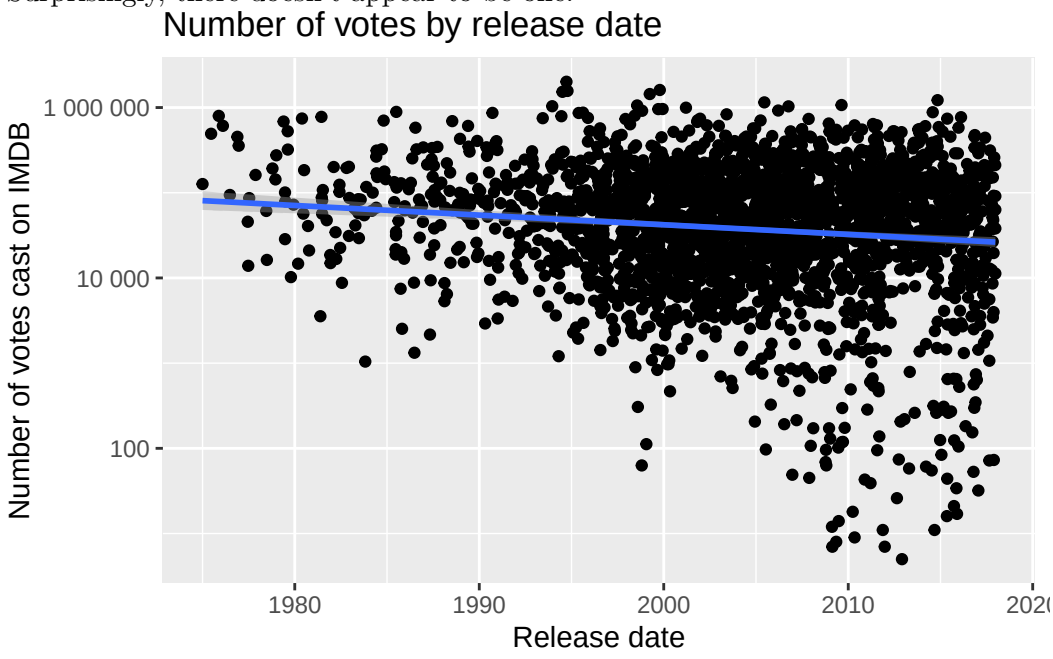
Create the above plots, but this time plot the ROI instead of the gross revenue.

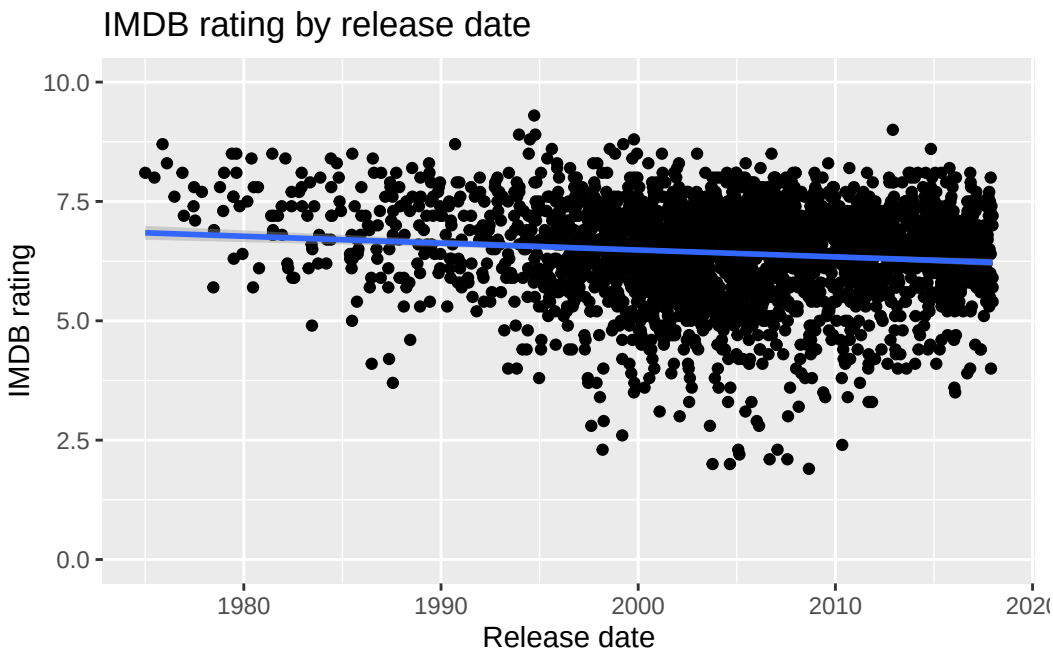




Exercise 10

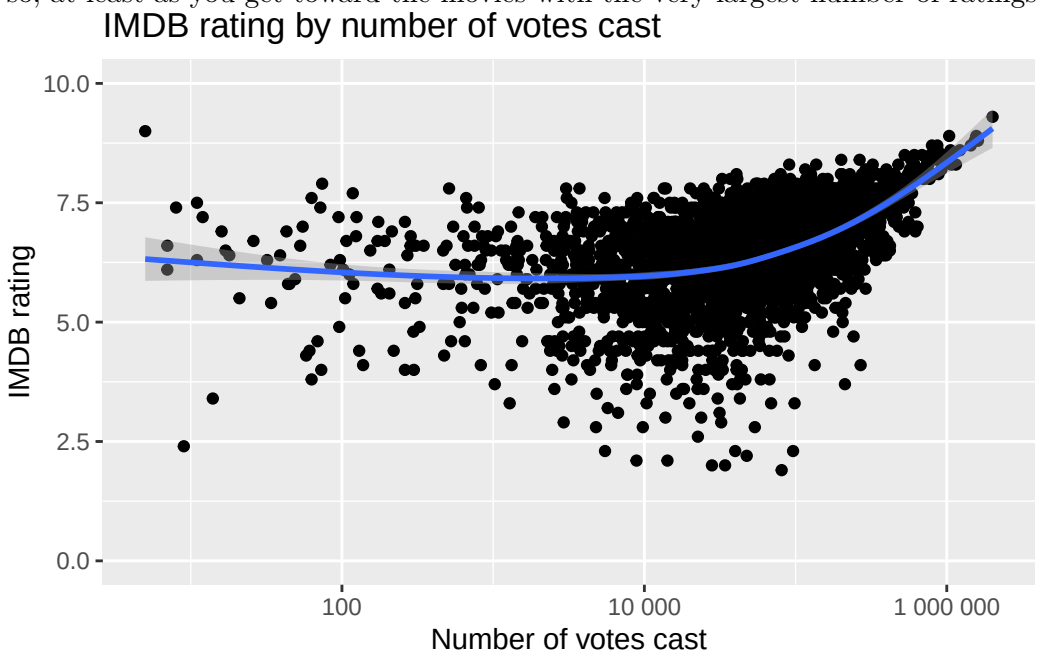
Is there a relationship between the release date and the IMDB ratings or votes cast? Surprisingly, there doesn't appear to be one.





Exercise 11

Is there a relationship between the IMDB rating and the number of votes cast? It appears so, at least as you get toward the movies with the very largest number of ratings.



Exercise 12

Looking at that above plot, I'm interested in (a) what are those movies with the largest number of votes? and (b) what are those movies with at least 50,000 votes that have the worst scores?

```
movimdb |>
  arrange(desc(votes)) |>
  head(10) |>
  select(release_date, movie, roi, imdb, votes)
```

```
# A tibble: 10 x 5
   release_date movie                      roi  imdb   votes
   <date>       <chr>                    <dbl> <dbl>   <dbl>
 1 1994-09-23   The Shawshank Redemption  1.13   9.3 2009031
 2 1999-10-15   Fight Club                1.55   8.8 1607508
 3 1994-10-14   Pulp Fiction             26.6    8.9 1568242
 4 1994-07-06   Forrest Gump             12.4    8.8 1529711
 5 1999-03-31   The Matrix                7.13   8.7 1441344
 6 2014-11-05   Interstellar              4.05   8.6 1221035
 7 2005-06-15   Batman Begins             2.39   8.3 1149747
 8 2009-08-21   Inglourious Basterds      4.53   8.3 1070753
 9 1998-07-24   Saving Private Ryan       7.46   8.6 1058789
10 1993-12-15   Schindler's List         12.9    8.9 1036894
```

No surprises there. These are some of the most universally loved films ever made. Interesting that the return on investment varies wildly (1.13x for the highest rated movie of all time, up to 26x for *Pulp Fiction*, which had to pay for an all-star cast).

```
movimdb |>
  filter(votes>50000) |>
  arrange(imdb) |>
  head(10) |>
  select(release_date, movie, roi, imdb, votes)
```

```
# A tibble: 10 x 5
   release_date movie                roi  imdb  votes
   <date>       <chr>              <dbl> <dbl>  <dbl>
 1 2008-08-29   Disaster Movie      1.84   1.9  80918
 2 2007-01-26   Epic Movie          4.34   2.3  96271
 3 2006-02-17   Date Movie          4.26   2.8  53781
 4 2011-11-11   Jack and Jill       1.91   3.3  68909
```

```
 5 2004-07-23   Catwoman                  0.821   3.3  98513
 6 1997-06-20   Batman & Robin            1.91    3.7 212085
 7 1997-06-13   Speed 2: Cruise Control   1.37    3.8  67296
 8 1994-12-23   Street Fighter            2.84    3.8  58912
 9 2015-02-13   Fifty Shades of Grey     14.3     4.1 269355
10 2010-07-01   The Last Airbender        2.13    4.1 133813
```

Interesting that several of these having such terrible reviews still have fairly high return on investment (>14x for *Fifty Shades of Grey*!).

6.3 College Majors & Income

6.3.1 About the data

This is the data behind the FiveThirtyEight article, "The Economic Guide To Picking A College Major".

- All data is from American Community Survey 2010-2012 Public Use Microdata Series.
- Original data and more: http://www.census.gov/programs-surveys/acs/data/pums.html.
- Documentation: http://www.census.gov/programs-surveys/acs/technical-documentation/pums.html

Data Dictionary:

Header	Description
`Rank`	Rank by median earnings
`Major_code`	Major code, FO1DP in ACS PUMS
`Major`	Major description
`Major_category`	Category of major from Carnevale et al
`Total`	Total number of people with major
`Sample_size`	Sample size (unweighted) of full-time, year-round ONLY (used for earnings)
`Men`	Male graduates
`Women`	Female graduates
`ShareWomen`	Women as share of total
`Employed`	Number employed (ESR == 1 or 2)
`Full_time`	Employed 35 hours or more
`Part_time`	Employed less than 35 hours
`Full_time_year_round`	Employed at least 50 weeks (WKW == 1) and at least 35 hours (WKHP >= 35)
`Unemployed`	Number unemployed (ESR == 3)

Header	Description
`Unemployment_rate`	Unemployed / (Unemployed + Employed)
`Median`	Median earnings of full-time, year-round workers
`P25th`	25th percentile of earnigns
`P75th`	75th percentile of earnings
`College_jobs`	Number with job requiring a college degree
`Non_college_jobs`	Number with job not requiring a college degree
`Low_wage_jobs`	Number in low-wage service jobs

6.3.2 Import and clean

If you haven't already loaded the packages we need, go ahead and do that now.

```
library(tidyverse)
library(ggrepel)
library(scales)
library(lubridate)
```

Now, use the `read_csv()` function from **readr** (loaded when you load **tidyverse**), to read in the **grads.csv** dataset into a new object called `grads_raw`.

Read in the raw data.

```
grads_raw <- read_csv("data/grads.csv")
grads_raw
```

Now clean it up a little bit. Remember, construct your pipeline one step at a time first. Once you're happy with the result, assign the results to a new object, `grads`.

- Make sure the data is arranged descending by Median income. It should be already, but don't make any assumptions.
- Make the Major sentence case so it's not ALL CAPS. This uses the `str_to_title()` function from the **stringr** package, loaded with **tidyverse**.
- Make it a factor variable with levels ordered according to median income.
- Do the same for `Major_category` – make it a factor variable with levels ordered according to median income.
- Add a new variable, `pct_college`, that's the proportion of graduates employed in a job requiring a college degree. We'll do some analysis with this later on to look at under-employment.
- There's one entry ("Military technologies") that has no data about employment. This new variable is therefore missing. Let's remove this entry.

- There's an entry with an unknown number of total majors, men, or women ("Food Science"). Let's remove it by removing anything with a missing Total number.

```
grads <- grads_raw |>
  arrange(desc(Median)) |>
  mutate(Major = str_to_title(Major)) |>
  mutate(Major = fct_reorder(Major, Median)) |>
  mutate(Major_category = fct_reorder(Major_category, Median)) |>
  mutate(pct_college=College_jobs/(College_jobs+Non_college_jobs)) |>
  filter(!is.na(pct_college)) |>
  filter(!is.na(Total))
grads
```

6.3.3 Exploratory Data Analysis

Let's start with an exercise.

Exercise 13

Remake table 1 from the FiveThirtyEight article.

- Use the `select()` function to get only the columns you care about.
- Use `head(10)` or `tail(10)` to show the first or last few rows.

```
                                    Major    Major_category Total Median
1                     Petroleum Engineering       Engineering  2339 110000
2             Mining And Mineral Engineering       Engineering   756  75000
3                  Metallurgical Engineering       Engineering   856  73000
4  Naval Architecture And Marine Engineering       Engineering  1258  70000
5                      Chemical Engineering       Engineering 32260  65000
6                       Nuclear Engineering       Engineering  2573  65000
7                         Actuarial Science          Business  3777  62000
8                Astronomy And Astrophysics Physical Sciences  1792  62000
9                    Mechanical Engineering       Engineering 91227  60000
10                   Electrical Engineering       Engineering 81527  60000

                                         Major            Major_category Total Median
1  Communication Disorders Sciences And Services                    Health 38279  28000
2                      Early Childhood Education                 Education 37589  28000
3                        Other Foreign Languages Humanities & Liberal Arts 11204  27500
4                         Drama And Theater Arts                      Arts 43249  27000
5                       Composition And Rhetoric Humanities & Liberal Arts 18953  27000
```

```
6                                  Zoology    Biology & Life Science  8409  26000
7                    Educational Psychology  Psychology & Social Work  2854  25000
8                      Clinical Psychology  Psychology & Social Work  2838  25000
9                    Counseling Psychology  Psychology & Social Work  4626  23400
10                        Library Science                 Education  1098  22000
```

If you have the **DT** package installed, you can make an interactive table just like the one in the FiveThirtyEight article.

```
library(DT)
grads |>
  select(Major, Major_category, Total, Median) |>
  datatable()
```

Show 10 entries Search:

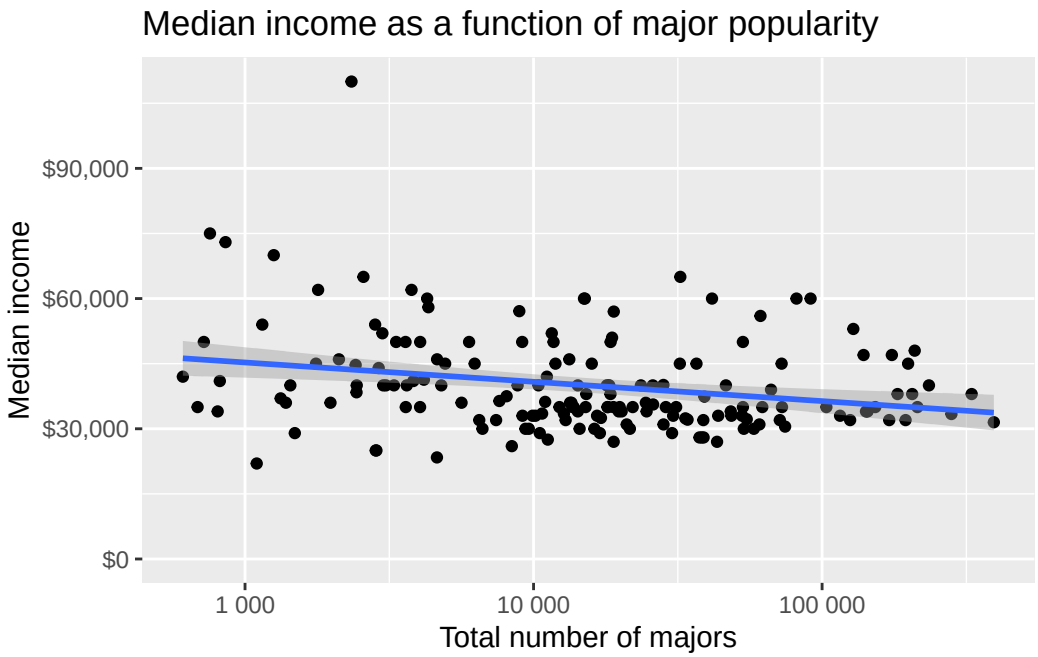
	Major	Major_category	Total	Median
1	Petroleum Engineering	Engineering	2339	110000
2	Mining And Mineral Engineering	Engineering	756	75000
3	Metallurgical Engineering	Engineering	856	73000
4	Naval Architecture And Marine Engineering	Engineering	1258	70000
5	Chemical Engineering	Engineering	32260	65000
6	Nuclear Engineering	Engineering	2573	65000
7	Actuarial Science	Business	3777	62000
8	Astronomy And Astrophysics	Physical Sciences	1792	62000
9	Mechanical Engineering	Engineering	91227	60000
10	Electrical Engineering	Engineering	81527	60000

Showing 1 to 10 of 171 entries

Previous 1 2 3 4 5 … 18 Next

Let's continue with more exploratory data analysis (EDA). Let's plot median income by the total number of majors. Is there a correlation between the number of people majoring in a topic and that major's median income? The `expand_limits` lets you put $0 on the Y-axis. You might try making the x-axis scale logarithmic.

```
ggplot(grads, aes(Total, Median)) +
  geom_point() +
  geom_smooth(method="lm") +
  expand_limits(y=0) +
  scale_x_log10(label=scales::number_format()) +
  scale_y_continuous(label=dollar_format()) +
  labs(x="Total number of majors",
       y="Median income",
       title="Median income as a function of major popularity")
```

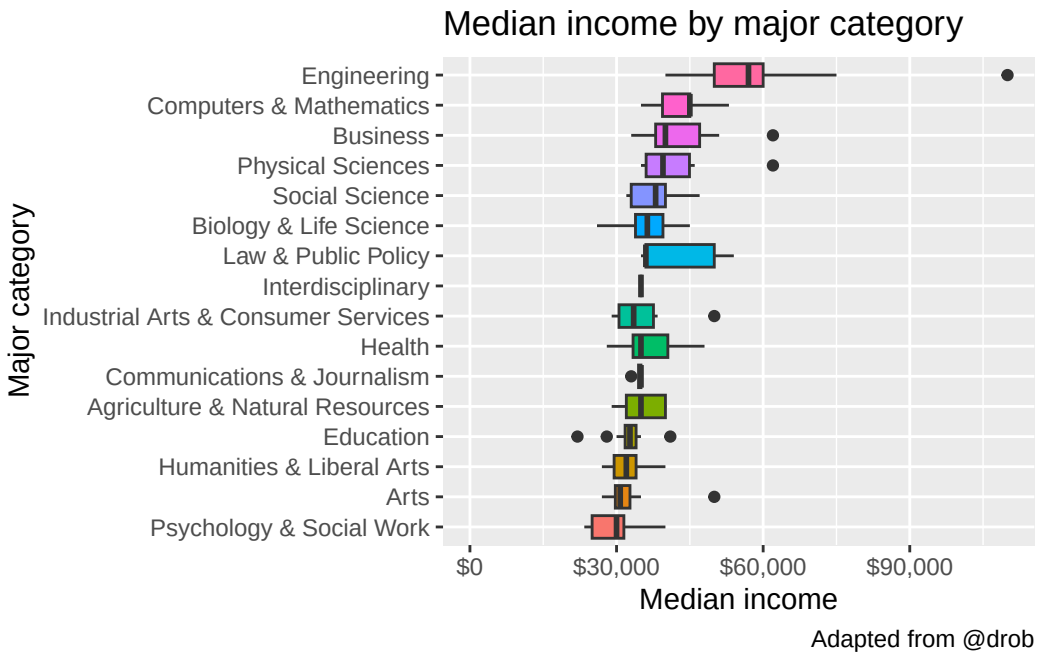


You could run a regression analysis to see if there's a trend.

```
lm(Median~(Total), data=grads) |> summary()
```

What categories of majors make more money than others? Let's make a boxplot of median income by major category. Let's expand the limits to include 0 on the y-axis, and flip the coordinate system.

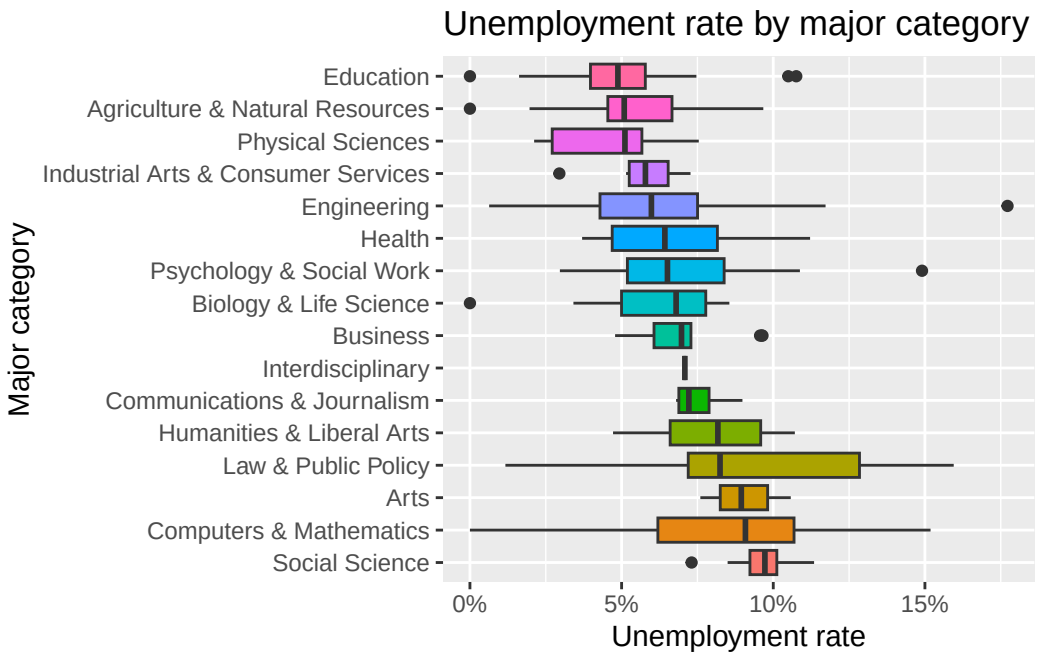
```
grads |>
  ggplot(aes(Major_category, Median)) +
  geom_boxplot(aes(fill = Major_category)) +
  expand_limits(y = 0) +
  coord_flip() +
  scale_y_continuous(labels = dollar_format()) +
  theme(legend.position = "none") +
  labs(x="Major category",
       y="Median income",
       title="Median income by major category",
       caption="Adapted from @drob")
```



What about unemployment rates? Let's to the same thing here but before ggplot'ing, let's mutate the major category to relevel it descending by the unemployment rate. Therefore the highest unemployment rate will be the first level of the factor. Let's expand limits again, and flip the coordinate system.

```
grads |>
  mutate(Major_category=fct_reorder(Major_category, -Unemployment_rate)) |>
  ggplot(aes(Major_category, Unemployment_rate, fill = Major_category)) +
  geom_boxplot() +
  expand_limits(y = 0) +
```

```
  coord_flip() +
  scale_y_continuous(labels = percent_format()) +
  theme(legend.position = "none") +
  labs(x="Major category",
       y="Unemployment rate",
       title="Unemployment rate by major category")
```



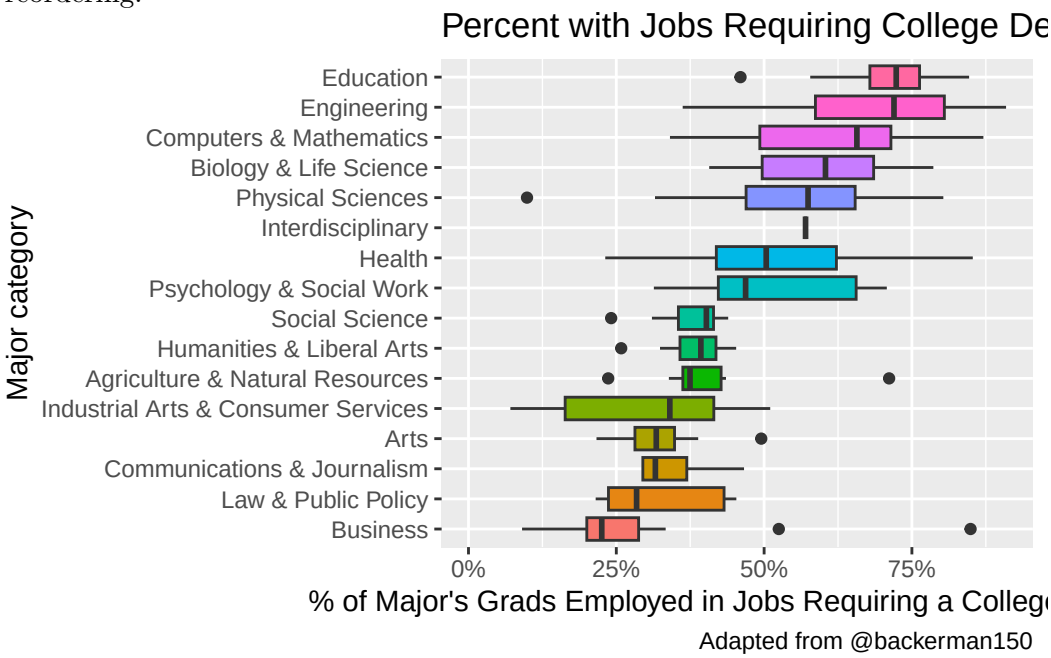
Most of these make sense except for the high median and large variability of "Computers & Mathematics" category. Especially considering how these had the second highest median salary. Let's see what these were. Perhaps it was the larger number of Computer and Information Systems, and Communication Technologies majors under this category that were dragging up the Unemployment rate.

```
grads |>
  filter(Major_category=="Computers & Mathematics") |>
  select(Major, Median, Sample_size, Unemployment_rate)
```

Exercise 14

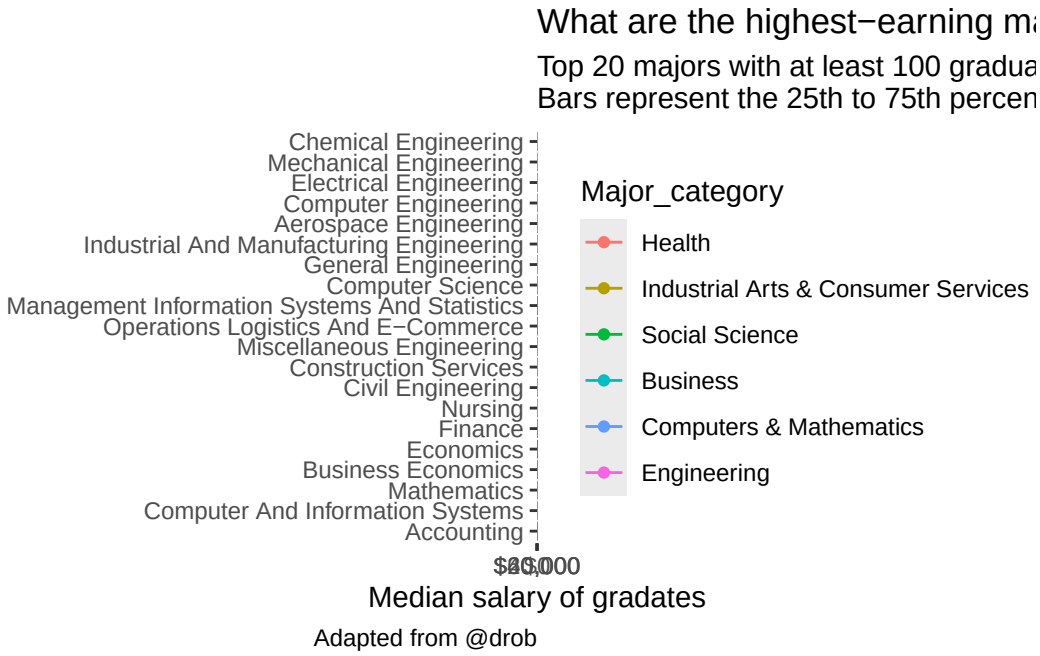
What about "underemployment?" Which majors have more students finding jobs requiring college degrees? This time make a boxplot of each major category's percentage of majors having jobs requiring a college degree (`pct_college`). Do the same factor

reordering.



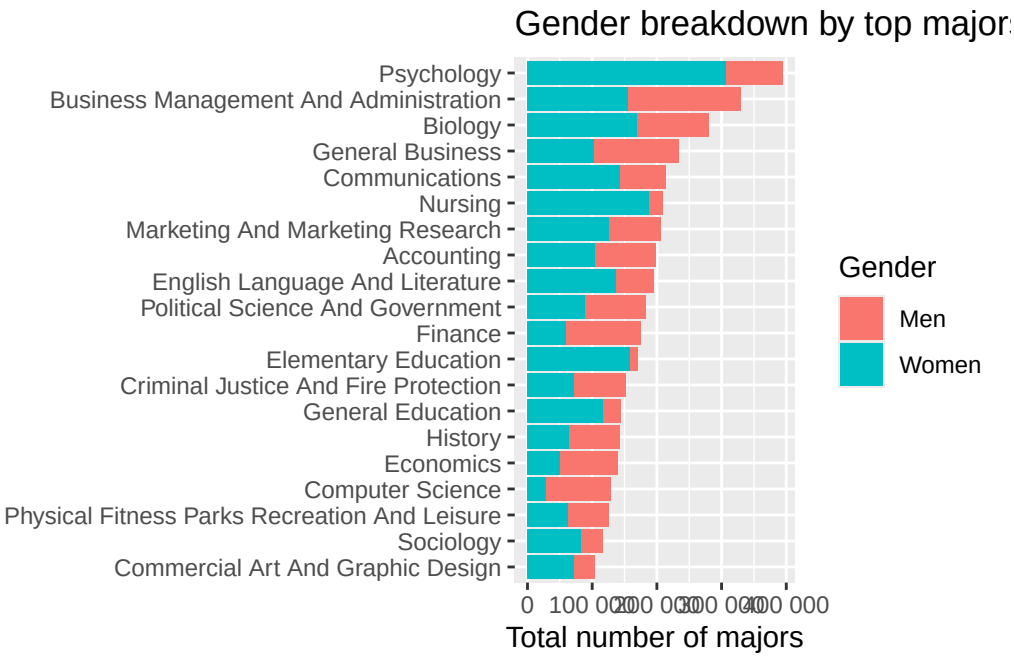
What are the highest earning majors? First, filter to majors having at least 100 samples to use for income data. Try changing `head(20)` to `tail(20)` to get the lowest earners.

```
grads |>
  filter(Sample_size >= 100) |>
  head(20) |>
  ggplot(aes(Major, Median, color = Major_category)) +
  geom_point() +
  geom_errorbar(aes(ymin = P25th, ymax = P75th)) +
  expand_limits(y = 0) +
  scale_y_continuous(labels = dollar_format()) +
  coord_flip() +
  labs(title = "What are the highest-earning majors?",
       subtitle = "Top 20 majors with at least 100 graduates surveyed.\nBars represent the
       x = "",
       y = "Median salary of gradates",
       caption="Adapted from @drob")
```



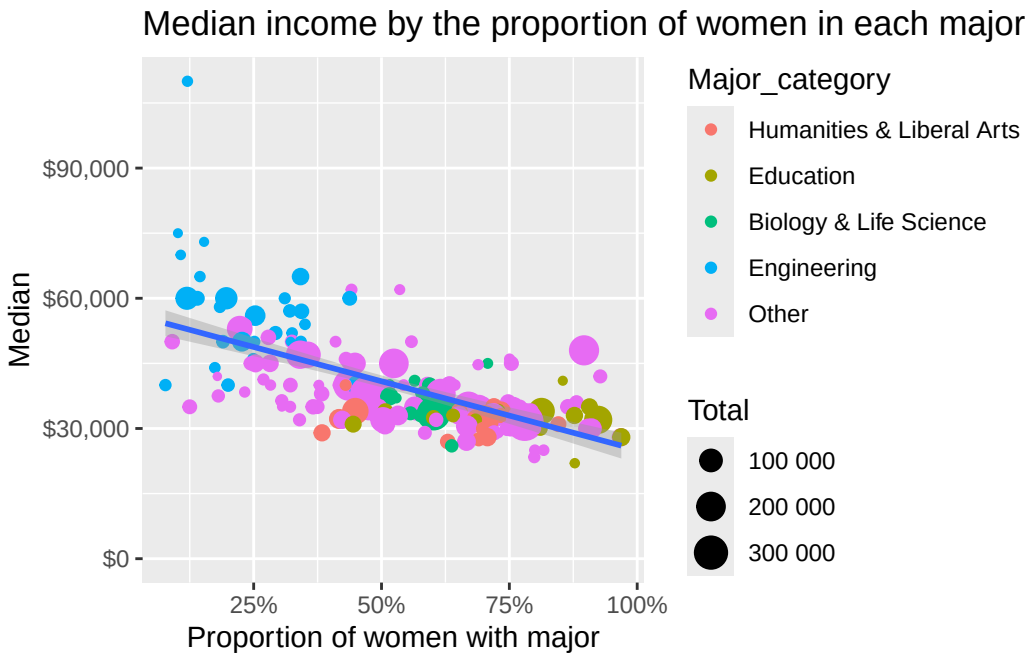
How do the top majors break down by gender? This plot first gets the top 20 most popular majors by total overall students. It reorders the "Major" variable by the total number of people taking it. It then gathers the "Men" and "Women" variable into a column with the number of men or women, with a key column called "Gender" indicating whether you're looking at men or women. It plots the total number in that major, and color-codes by gender.

```
grads |>
  arrange(desc(Total)) |>
  head(20) |>
  mutate(Major = fct_reorder(Major, Total)) |>
  gather(Gender, Number, Men, Women) |>
  ggplot(aes(Major, Number, fill = Gender)) +
  geom_col() +
  coord_flip() +
  scale_y_continuous(labels=number_format()) +
  labs(x="", y="Total number of majors", title="Gender breakdown by top majors")
```



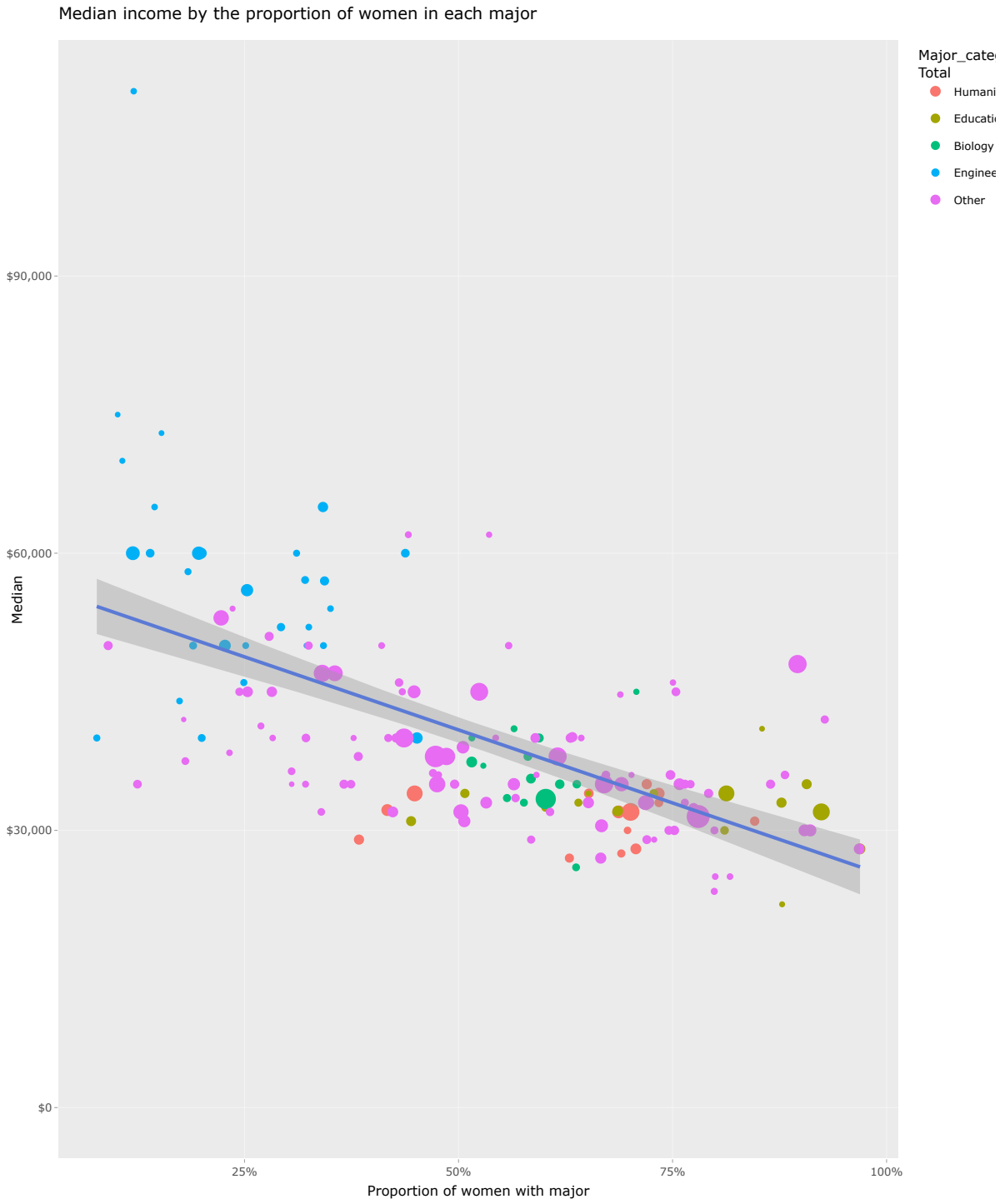
What do earnings look like by gender? Let's plot median salary by the Share of women in that major, making the size of the point proportional to the number of students enrolled in that major. Let's also lump all the major categories together if they're not one of the top four. I'm also passing the `label=` aesthetic mapping. You'll see why in a few moments. For now, there is no geom that takes advantage of the label aesthetic.

```
p <- grads |>
  mutate(Major_category = fct_lump(Major_category, 4)) |>
  ggplot(aes(ShareWomen, Median, label=Major)) +
  geom_point(aes(size=Total, color=Major_category)) +
  geom_smooth(method="lm") +
  expand_limits(y=0) +
  scale_size_continuous(labels=number_format()) +
  scale_y_continuous(labels=dollar_format()) +
  scale_x_continuous(labels=percent_format()) +
  labs(x="Proportion of women with major",
       title="Median income by the proportion of women in each major")
p
```



If you have the **plotly** package installed, you can make an interactive graphic. Try hovering over the points, or using your mouse to click+drag a box around a segment of the plot to zoom in on.

```
library(plotly)
ggplotly(p)
```



Let's run a regression analysis to see if the proportion of women in the major is correlated with

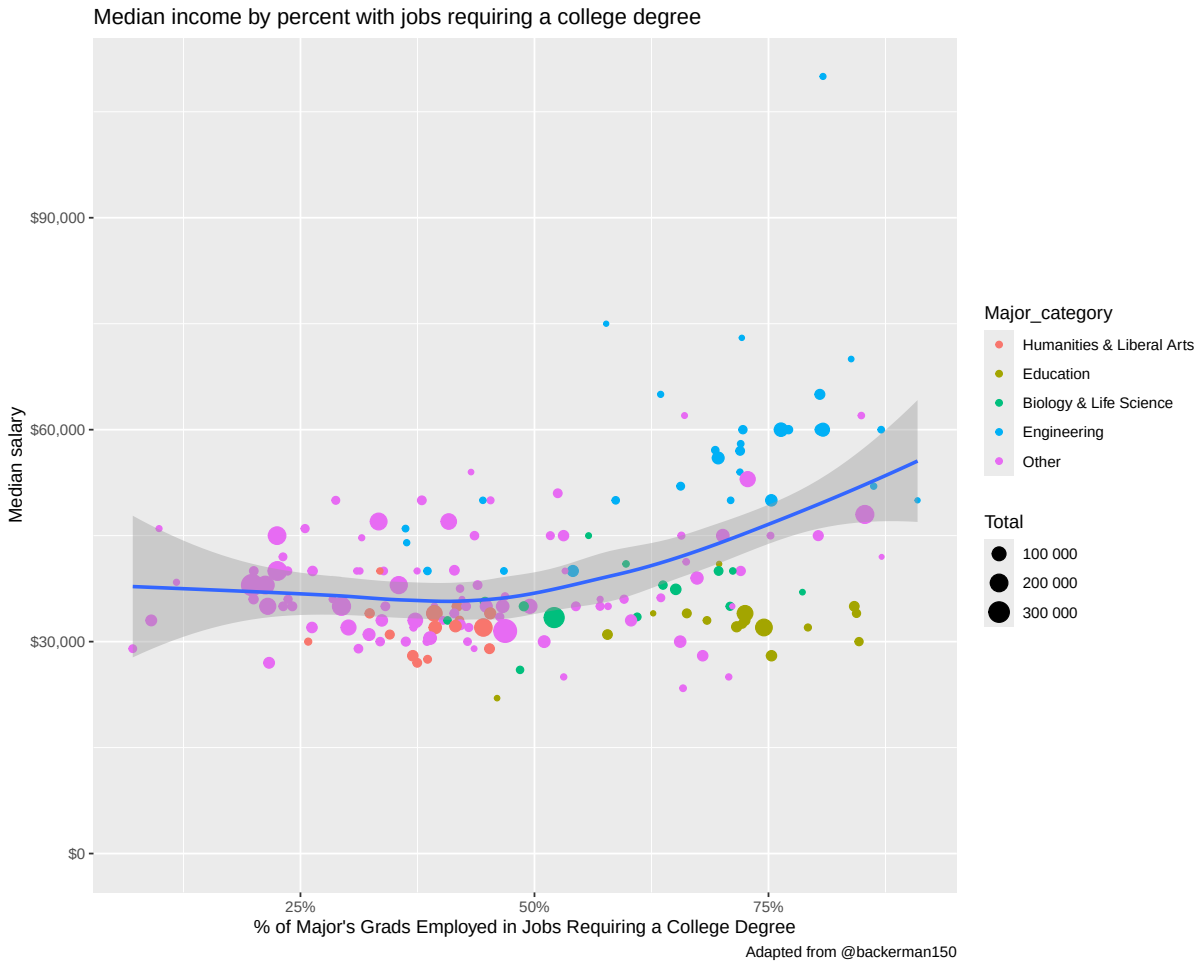
salary. It looks like every percentage point increase in the proportion of women in a particular major is correlated with a $23,650 decrease in salary.

```
lm(Median ~ ShareWomen, data = grads, weights = Sample_size) |>
  summary()
```

```
Call:
lm(formula = Median ~ ShareWomen, data = grads, weights = Sample_size)

Weighted Residuals:
    Min      1Q  Median      3Q     Max
-260544  -61278  -13324   33834  865216

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)    52079       1441  36.147   <2e-16
ShareWomen    -23660       2410  -9.816   <2e-16

Residual standard error: 123300 on 169 degrees of freedom
Multiple R-squared:  0.3631,    Adjusted R-squared:  0.3594
F-statistic: 96.36 on 1 and 169 DF,  p-value: < 2.2e-16
```

Let's run a similar analysis looking at the median income as a function of the percentage of majors getting a job requiring a college degree.

```
grads |>
  mutate(Major_category = fct_lump(Major_category, 4)) |>
  ggplot(aes(pct_college, Median)) +
  geom_point(aes(size=Total, col=Major_category)) +
  geom_smooth() +
  scale_x_continuous(label=percent_format()) +
  scale_y_continuous(label=dollar_format()) +
  scale_size_continuous(label=number_format()) +
  expand_limits(y=0) +
  labs(x="% of Major's Grads Employed in Jobs Requiring a College Degree",
       y="Median salary",
       title="Median income by percent with jobs requiring a college degree",
       caption="Adapted from @backerman150")
```



Here's Table 2 in the FiveThirtyEight piece. It uses the `mutate_at` function to run an arbitrary function on any number of variables defined in the `vars()` function. See the help for `?mutate_at` to learn more.

```
library(DT)
grads |>
  select(Major, Total, Median, P25th, P75th, Part_time, Non_college_jobs, Low_wage_jobs) |
  mutate_at(vars(Part_time, Non_college_jobs, Low_wage_jobs), funs(percent(./Total))) |>
  mutate_at(vars(Median, P25th, P75th), funs(dollar)) |>
  datatable()
```

Show 10 entries Search:

	Major	Total	Median	P25th	P75th	Part_time	Non_college_jobs	Low_wage_jobs
1	Petroleum Engineering	2339	$110,000	$95,000	$125,000	11.5434%	15.5622%	8.2514%
2	Mining And Mineral Engineering	756	$75,000	$55,000	$90,000	22.4868%	33.9947%	6.6138%
3	Metallurgical Engineering	856	$73,000	$50,000	$105,000	15.5374%	20.5607%	0.0000%
4	Naval Architecture And Marine Engineering	1258	$70,000	$43,000	$80,000	11.9237%	8.1081%	0.0000%
5	Chemical Engineering	32260	$65,000	$50,000	$75,000	16.0570%	13.7632%	3.0130%
6	Nuclear Engineering	2573	$65,000	$50,000	$102,000	10.2604%	25.5344%	9.4831%
7	Actuarial Science	3777	$62,000	$53,000	$72,000	7.8369%	8.3135%	6.8573%
8	Astronomy And Astrophysics	1792	$62,000	$31,500	$109,000	30.8594%	27.9018%	12.2768%
9	Mechanical Engineering	91227	$60,000	$48,000	$70,000	14.3609%	17.9596%	3.5658%
10	Electrical Engineering	81527	$60,000	$45,000	$72,000	15.5715%	13.3379%	3.8883%

Showing 1 to 10 of 171 entries Previous 1 2 3 4 5 … 18 Next

7 Reproducible Reporting with RMarkdown

Contemporary life science is plagued by reproducibility issues. This workshop covers some of the barriers to reproducible research and how to start to address some of those problems during the data management and analysis phases of the research life cycle. In this workshop we will cover using R and dynamic document generation with RMarkdown and RStudio to weave together reporting text with executable R code to automatically generate reports in the form of PDF, Word, or HTML documents.

Spend a few minutes to learn a little bit about *Markdown*. All you really need to know is that Markdown is a lightweight markup language that lets you create styled text (like **bold**, *italics*, links, etc.) using a very lightweight plain-text syntax: `(like **bold**, _italics_, [links](https://blog.stephenturner.us/), etc.)`. The resulting text file can be *rendered* into many downstream formats, like PDF (for printing) or HTML (websites).

1. *(30 seconds)* Read the summary paragraph on the Wikipedia page.
2. *(1 minute)* Bookmark and refer to this markdown reference: http://commonmark.org/help/.
3. *(5-10 minutes)* Run through this 10-minute in-browser markdown tutorial: http://commonmark.org/help/tutorial/.
4. *(5-10 minutes)* Go to http://dillinger.io/, an in-browser Markdown editor, and play around. Write a simple markdown document, and export it to HTML and/or PDF.
5. *(10 minutes)* See RStudio's excellent documentation on Rmarkdown at http://rmarkdown.rstudio.com/. Click "Getting Started" and watch the 1 minute video on the Introduction page. Continue reading through each section here on the navigation bar to the left (*Introduction* through *Cheatsheets*, and optionally download and print out the cheat sheet). Finally, browse through the RMarkdown Gallery.

7.1 Who cares about reproducible research?

Science is plagued by reproducibility problems. Especially genomics!

- Scientists in the United States spend $28 billion each year on basic biomedical research that cannot be repeated successfully.[1]

[1] Freedman, et al. "The economics of reproducibility in preclinical research." *PLoS Biol* 13.6 (2015): e1002165.

- A reproducibility study in psychology found that only 39 of 100 studies could be reproduced.[2]
- The Journal *Nature* on the issue of reproducibility:[3]
  - "*Nature* and the Nature research journals will introduce editorial measures to address the problem by improving the consistency and quality of reporting in life-sciences articles… **we will give more space to methods sections. We will examine statistics more closely and encourage authors to be transparent, for example by including their raw data.**"
  - *Nature* also released a checklist, unfortunately with *wimpy* computational check *(see #18)*.
- On microarray reproducibility:[4]
  - 18 Nat. Genet. microarray experiments
  - Less than 50% reproducible
  - Problems:
    - Missing data (38%)
    - Missing software/hardware details (50%)
    - Missing method/processing details (66%)
- NGS: run-of-the-mill variant calling (align, process, call variants):[5]
  - 299 articles published in 2011 citing the 1000 Genomes project pilot publication
  - Only 19 were NGS studies with similar design
  - Only 10 used tools recommended by 1000G.
  - Only 4 used full 1000G workflow (realignment & quality score recalibration).

Consider this figure:

[2] http://www.nature.com/news/first-results-from-psychology-s-largest-reproducibility-test-1.17433
[3] http://www.nature.com/news/reproducibility-1.17552
[4] Ioannidis, John PA, et al. "Repeatability of published microarray gene expression analyses." *Nature genetics* 41.2 (2009): 149-155.
[5] Nekrutenko, Anton, and James Taylor. "Next-generation sequencing data interpretation: enhancing reproducibility and accessibility." *Nature Reviews Genetics* 13.9 (2012): 667-672.

How do we reproduce it? What do we need?

- The data.
  - Data points themselves.
  - Other metadata.
- The code.
  - Should be readable.
  - Comments in the code / well-documented so a normal person can figure out how it runs.
  - How were the trend lines drawn?
  - What version of software / packages were used?

This kind of information is rarely available in scientific publications, but it's now extraordinarily easy to put this kind of information on the web.

Could I replicate Figure 1 from your last publication? If not, what would *you and your co-authors* need to provide or do so I could replicate Figure 1 from your last publication?

As scientists we should aim for *robust* and *reproducible* research

- "**Robust research** is about doing small things that stack the deck in your favor to prevent mistakes."
  —*Vince Buffalo, author of Bioinformatics Data Skills (2015).*
- **Reproducible research** can be repeated by other researchers with the same results.

7.1.1 Reproducibility is hard!

1. Genomics data is too large and high dimensional to easily inspect or visualize. Workflows involve multiple steps and it's hard to inspect every step.
2. Unlike in the wet lab, we don't always know what to expect of our genomics data analysis.
3. It can be hard to distinguish *good* from *bad* results.
4. Scientific code is usually only run once to generate results for a publication, and is more likely to contain silent bugs. (code that may produces unknowingly incorrect output rather than stopping with an error message).

7.1.2 What's in it for *you?*

Yeah, it takes a lot of effort to be robust and reproducible. However, *it will make your life (and science) easier!*

- Most likely, you will have to re-run your analysis more than once.

- In the future, you or a collaborator may have to re-visit part of the project.
- Your most likely collaborator is your future self, and your past self doesn't answer emails.
- You can make modularized parts of the project into re-useable tools for the future.

- Reproducibility makes you easier to work and collaborate with.

7.1.3 Some recommendations for reproducible research

1. **Write code for humans, write data for computers.**
   - Code should be broken down into small chunks that may be re-used.

   - Make names/variables consistent, distinctive and meaningful.

- Adopt a style be consistent.[6]
- Write concise and clear comments.

2. **Make incremental changes.** Work in small steps with frequent feedback. Use version control. See http://swcarpentry.github.io/git-novice/ for resources on version control.
3. **Make assertions and be loud, in code and in your methods.** Add tests in your code to make sure it's doing what you expect. See http://software-carpentry.org/v4/test/ for resources on testing code.
4. **Use existing libraries (packages) whenever possible.** Don't reinvent the wheel. Use functions that have already been developed and tested by others.
5. **Prevent catastrophe and help reproducibility by making your data *read-only*.** Rather than modifying your original data directly, always use a workflow that reads in data, processes/modifies, then writes out intermediate and final files as necessary.
6. **Encapsulate the full project into one directory that is supported with version control.** See: Noble, William Stafford. "A quick guide to organizing computational biology projects." *PLoS Comput Biol* 5.7 (2009): e1000424.
7. **Release your code and data.** Simple. Without your code and data, your research is not reproducible.

    - GitHub (https://github.com/) is a great place for storing, distributing, collaborating, and version-controlling code.
    - RPubs (http://rpubs.com/) allows you to share dynamic documents you write in RStudio online.
    - Figshare (http://figshare.com/) and Zenodo (https://zenodo.org/) allow you to upload any kind of research output, publishable or not, free and unlimited. Instantly get permanently available, citable DOI for your research output.
    - *"Data/code is available upon request"* or *"Data/code is available at the lab's website"* are completely unacceptable in the 21st century.

8. **Write code that uses relative paths.**

    - Don't use hard-coded absolute paths (i.e. `/Users/stephen/Data/seq-data.csv` or `C:\Stephen\Documents\Data\Project1\data.txt`).
    - Put the data in the project directory and reference it *relative* to where the code is, e.g., `data/gapminder.csv`, etc.

9. **Always set your seed.** If you're doing anything that involves random/monte-carlo approaches, always use `set.seed()`.
10. **Document everything and use code as documentation.**

    - Document why you do something, not mechanics.
    - Document your methods and workflows.
    - Document the origin of all data in your project directory.

[6] http://adv-r.had.co.nz/Style.html

- Document **when** and **how** you downloaded the data.
- Record **data** version info.
- Record **software** version info with `session_info()`.
- Use dynamic documentation to make your life easier.

7.2 RMarkdown

RMarkdown is a variant of Markdown that lets you embed R code chunks that execute when you compile the document. What, what? Markdown? Compile? What's all this about?

7.2.1 Markdown

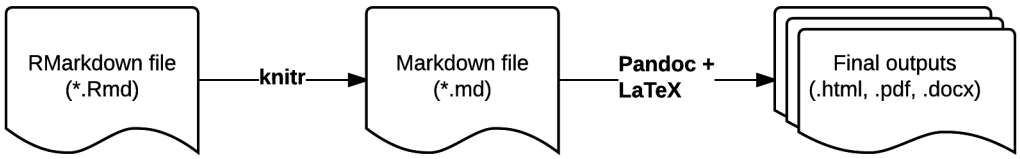
Ever heard of HTML? It's what drives the internet. HTML is a *markup language* - that's what the *ML* stands for. The terminology evolved from "marking up" paper manuscripts by editors, where the editor would instruct an author or typesetter how to render the resulting text. Markup languages let you annotate **text** that you want to display with instructions about how to display it.

I emphasize **text** because this is fundamentally different than word processing. When you use MS Word, for example, you're creating a special proprietary binary file (the .docx) file that shows you how a document looks. By contrast, writing in a markup language like HTML or Markdown, you're writing plain old text, using a text editor. The toolchain used to render the markup text into what you see on a display or in a PDF has always been and will always bee free and open.

You can learn Markdown in about 5 minutes. Let's open up a web-based Markdown editor like http://dillinger.io/ or use a desktop Markdown editor like MarkdownPad (Windows) or MacDown (Mac).

7.2.2 RMarkdown workflow

RMarkdown is an enhanced version of Markdown that lets you embed R code into the document. When the document is compiled/rendered, the R code is executed by R, the output is then automatically rendered as Markdown with the rest of the document. The Markdown is then further processed to final output formats like HTML, PDF, DOCX, etc.



7.3 Authoring RMarkdown documents

> ***Note***: Before going any further, open up the options (Tools, Global Options), click the RMarkdown section, and **uncheck** the box, *"Show output inline for all R Markdown documents."*

7.3.1 From scratch

First, open RStudio. Create a new project. Quit RStudio, then launch RStudio using the project file (.Rproj) you just created.

Next, download the gapminder data from the data page. Put this file in your R project directory. Maybe put it in a subdirectory called "data." Importantly, now your code and data will live in the same place.

Let's create a bare-bones RMarkdown document that compiles to HTML. In RStudio, select **File**, **New File**, **R Markdown...**. Don't worry about the title and author fields. When the new document launches, select everything then delete it. Let's author an RMarkdown file from scratch. Save it as `fromscratch.Rmd`.

````
# Introduction

This is my first RMarkdown document!

# Let's embed some R code

Let's load the **Gapminder** data:


```{r}
library(dplyr)
library(readr)
gm <- read_csv('data/gapminder.csv')
````
````

```
head(gm)
```


```
The mean life expectancy is `r mean(gm$lifeExp)` years.

The years surveyed in this data include: `r unique(gm$year)`.

# Session Information


```{r}
sessionInfo()
```
```

Hit the **Knit HTML** button in the editor window. You should see the rendered document pop up.

So let's break that down to see exactly what happened there. Recall the RMarkdown Workflow shown above. You start with an RMarkdown document (Rmd). When you hit the Knit HTML button, The **knitr** R package parses through your source document and executes all the R code chunks defined by the R code chunk blocks. The source code itself and the results are then turned back into regular markdown, inserted into an intermediate markdown file (.md), and finally rendered into HTML by Pandoc.

Try this. Instead of using the button, load the knitr package and just knit the document to markdown format. Run this in the console.

```
library(knitr)
knit("fromscratch.Rmd")
```

Now, open up that regular markdown file and take a look.

```
# Introduction

This is my first RMarkdown document!

# Let's embed some R code

Let's load the **Gapminder** data:


```r
```
```

```
library(dplyr)
library(readr)
gm <- read_csv("data/gapminder.csv")
head(gm)
```

```
##       country continent year lifeExp      pop gdpPercap
## 1 Afghanistan      Asia 1952  28.801  8425333  779.4453
## 2 Afghanistan      Asia 1957  30.332  9240934  820.8530
## 3 Afghanistan      Asia 1962  31.997 10267083  853.1007
## 4 Afghanistan      Asia 1967  34.020 11537966  836.1971
## 5 Afghanistan      Asia 1972  36.088 13079460  739.9811
## 6 Afghanistan      Asia 1977  38.438 14880372  786.1134
```

```
The mean life expectancy is 59.4744394 years.

The years surveyed in this data include: 1952, 1957, 1962, 1967, 1972, 1977, 1982, 1987, 199
```

7.3.2 From a template with YAML metadata

Go ahead and start a new R Markdown document. Fill in some title and author information.

This is going to put a YAML header in the file that looks something like this:

```
---
title: "Gapminder Analysis"
author: "Stephen Turner"
date: "January 1, 2017"
output: html_document
---
```

The stuff between the three ---s is metadata. You can read more about what kind of metadata can be included in the RMarkdown documentation. Try clicking the little wrench icon and setting some options, like including a table of contents and figure captions. Notice how the metadata front matter changes.

```
---
title: "Gapminder analysis"
author: "Stephen Turner"
```

```
date: "January 1, 2017"
output:
  html_document:
    fig_caption: yes
    toc: yes
---
```

Now, delete everything in that document below the metadata header and paste in what we had written before (above). Save this document under a different name (`rmdwithmeta.Rmd` for example). You'll now see that your HTML document takes the metadata and makes a nicely formatted title.

Let's add a plot in there. Open up a new R chunk with this:

````
```{r, fig.cap='Life Exp vs GDP'}
library(ggplot2)
ggplot(gm, aes(gdpPercap, lifeExp)) + geom_point()
```
````

Using RStudio you can fiddle around with different ways to make the graphic and keep the one you want. Maybe it looks like this:

````
```{r, fig.cap='Life Exp vs GDP'}
library(ggplot2)
ggplot(gm, aes(gdpPercap, lifeExp)) +
  geom_point() +
  scale_x_log10() +
  aes(col=continent)
```
````

7.3.3 Chunk options

You can modify the behavior of an R chunk with options. Options are passed in after a comma on the fence, as shown below.

````
```{r optionalChunkName, echo=TRUE, results='hide'}
# R code here
```
````

Some commonly used options include:

- `echo`: (`TRUE` by default) whether to include R source code in the output file.
- `results` takes several possible values:
  - `markup` (the default) takes the result of the R evaluation and turns it into markdown that is rendered as usual.
  - `hide` will hide results.
  - `hold` will hold all the output pieces and push them to the end of a chunk. Useful if you're running commands that result in lots of little pieces of output in the same chunk.
  - `asis` writes the raw results from R directly into the document. Only really useful for tables.
- `include`: (`TRUE` by default) if this is set to `FALSE` the R code is still evaluated, but neither the code nor the results are returned in the output document.
- `fig.width`, `fig.height`: used to control the size of graphics in the output.

Try modifying your first R chunk to use different values for `echo`, `results`, and `include`.

````
```{r}
gm <- read.csv('data/gapminder.csv')
head(gm)
tail(gm)
```
````

See the full list of options here: http://yihui.name/knitr/options/. There are lots!

A special note about **caching**: The `cache=` option is automatically set to `FALSE`. That is, every time you render the Rmd, all the R code is run again from scratch. If you use `cache=TRUE`, for this chunk, knitr will save the results of the evaluation into a directory that you specify. When you re-render the document, knitr will first check if there are previously cached results under the cache directory before really evaluating the chunk; if cached results exist and this code chunk has not been changed since last run (use MD5 sum to verify), the cached results will be (lazy-) loaded, otherwise new cache will be built; if a cached chunk depends on other chunks (see the `dependson` option) and any one of these chunks has changed, this chunk must be forcibly updated (old cache will be purged). **See the documentation for caching.**

7.3.4 Tables

The knitr package that runs the RMarkdown document in the background also has a function called `kable` that helps with printing tables nicely. It's only useful when you set `echo=FALSE` and `results='asis'`. Try this.

```
```{r}
head(gm)
```
```

Versus this:

```
```{r, results='asis'}
library(knitr)
kable(head(gm))
```
```

7.3.5 Changing output formats

Now try this. If you were successfully able to get a LaTeX distribution installed, you can render this document as a PDF instead of HTML. Try changing the line in the metadata from `html_document` to `pdf_document`. Notice how the *Knit HTML* button in RStudio now changes to *Knit PDF*. Try it. If you didn't get a LaTeX engine installed this won't work. Go back to the setup instructions after class to give this a try.

7.4 Distributing Analyses: Rpubs

RPubs.com is a free service from RStudio that allows you to seamlessly publish the results of your R analyses online. Sign up for an account at RPubs.com, then sign in on your browser.

Make sure your RMarkdown metadata is set to render to HTML rather than PDF. Render the document. Now notice the little **Publish** button in the HTML viewer pane. Click this. Sign in when asked, and give your document a name (usually the same name as the title of your Rmd).

Here are a few examples of documents I've published:

- http://rpubs.com/turnersd/daily_show_guests: Analysis of every guest who's ever been on *The Daily Show with Jon Stewart*.
- http://rpubs.com/turnersd/twoaxes: How to plot two different tracks of data with one axis on the left and one axis on the right.
- http://rpubs.com/turnersd/anscombe: Analysis of *Anscombe's Quartet* data.

Note how RPubs doesn't share your code! RPubs is a great way to share your analysis but doesn't let you share the source code. This is a huge barrier to reproducibility. There are plenty of ways to do this. One way is to go to gist.github.com and upload your code as a text file, then link back to the gist in your republished RPubs document.

Part II

Electives

8 Essential statistics

This chapter provides hands-on instruction and exercises covering basic statistical analysis in R. This will cover descriptive statistics, *t*-tests, linear models, chi-square, clustering, dimensionality reduction, and resampling strategies. We will also cover methods for "tidying" model results for downstream visualization and summarization.

Handouts: Download and print out these handouts and bring them to class:

- Cheat sheet
- Exercises handout

8.1 Our data: NHANES

8.1.1 About NHANES

The data we're going to work with comes from the National Health and Nutrition Examination Survey (NHANES) program at the CDC. You can read a lot more about NHANES on the CDC's website or Wikipedia. NHANES is a research program designed to assess the health and nutritional status of adults and children in the United States. The survey is one of the only to combine both survey questions and physical examinations. It began in the 1960s and since 1999 examines a nationally representative sample of about 5,000 people each year. The NHANES interview includes demographic, socioeconomic, dietary, and health-related questions. The physical exam includes medical, dental, and physiological measurements, as well as several standard laboratory tests. NHANES is used to determine the prevalence of major diseases and risk factors for those diseases. NHANES data are also the basis for national standards for measurements like height, weight, and blood pressure. Data from this survey is used in epidemiology studies and health sciences research, which help develop public health policy, direct and design health programs and services, and expand the health knowledge for the Nation.

We are using a small slice of this data. We're only using a handful of variables from the 2011-2012 survey years on about 5,000 individuals. The CDC uses a sampling strategy to purposefully oversample certain subpopulations like racial minorities. Naive analysis of the original NHANES data can lead to mistaken conclusions because the percentages of people from each racial group in the data are different from general population. The 5,000 individuals here

are resampled from the larger NHANES study population to undo these oversampling effects, so you can treat this as if it were a simple random sample from the American population.

You can download the data here: **nhanes.csv**. There's also a data dictionary here: **nhanes_dd.csv** that lists and describes each variable in our NHANES dataset. This table is copied below.

Variable	Definition
id	A unique sample identifier
Gender	Gender (sex) of study participant coded as male or female
Age	Age in years at screening of study participant. Note: Subjects 80 years or older were recorded as 80.
Race	Reported race of study participant, including non-Hispanic Asian category: Mexican, Hispanic, White, Black, Asian, or Other. Not availale for 2009-10.
Education	Educational level of study participant Reported for participants aged 20 years or older. One of 8thGrade, 9-11thGrade, HighSchool, SomeCollege, or CollegeGrad.
MaritalStatus	Marital status of study participant. Reported for participants aged 20 years or older. One of Married, Widowed, Divorced, Separated, NeverMarried, or LivePartner (living with partner).
RelationshipStatus	Simplification of MaritalStatus, coded as Committed if MaritalStatus is Married or LivePartner, and Single otherwise.
Insured	Indicates whether the individual is covered by health insurance.
Income	Numerical version of HHIncome derived from the middle income in each category
Poverty	A ratio of family income to poverty guidelines. Smaller numbers indicate more poverty
HomeRooms	How many rooms are in home of study participant (counting kitchen but not bathroom). 13 rooms = 13 or more rooms.
HomeOwn	One of Home, Rent, or Other indicating whether the home of study participant or someone in their family is owned, rented or occupied by some other arrangement.
Work	Indicates whether the individual is current working or not.
Weight	Weight in kg
Height	Standing height in cm. Reported for participants aged 2 years or older.
BMI	Body mass index (weight/height2 in kg/m2). Reported for participants aged 2 years or older.
Pulse	60 second pulse rate
BPSys	Combined systolic blood pressure reading, following the procedure outlined for BPXSAR.
BPDia	Combined diastolic blood pressure reading, following the procedure outlined for BPXDAR.

Variable	Definition
Testosterone	Testerone total (ng/dL). Reported for participants aged 6 years or older. Not available for 2009-2010.
HDLChol	Direct HDL cholesterol in mmol/L. Reported for participants aged 6 years or older.
TotChol	Total HDL cholesterol in mmol/L. Reported for participants aged 6 years or older.
Diabetes	Study participant told by a doctor or health professional that they have diabetes. Reported for participants aged 1 year or older as Yes or No.
DiabetesAge	Age of study participant when first told they had diabetes. Reported for participants aged 1 year or older.
nPregnancies	How many times participant has been pregnant. Reported for female participants aged 20 years or older.
nBabies	How many of participants deliveries resulted in live births. Reported for female participants aged 20 years or older.
SleepHrsNight	Self-reported number of hours study participant usually gets at night on weekdays or workdays. Reported for participants aged 16 years and older.
PhysActive	Participant does moderate or vigorous-intensity sports, fitness or recreational activities (Yes or No). Reported for participants 12 years or older.
PhysActiveDays	Number of days in a typical week that participant does moderate or vigorous-intensity activity. Reported for participants 12 years or older.
AlcoholDay	Average number of drinks consumed on days that participant drank alcoholic beverages. Reported for participants aged 18 years or older.
AlcoholYear	Estimated number of days over the past year that participant drank alcoholic beverages. Reported for participants aged 18 years or older.
SmokingStatus	Smoking status: Current Former or Never.

8.1.2 Import & inspect

First, let's load the dplyr and readr libraries.

```
library(readr)
library(dplyr)
```

If you see a warning that looks like this: `Error in library(dplyr) : there is no package called 'dplyr'` (or similar with readr), then you don't have the package installed correctly. See the (Appendix A)

Now, let's actually load the data. When we load data we assign it to a variable just like any other, and we can choose a name for that data. Since we're going to be referring to this data

a lot, let's give it a short easy name to type. I'm going to call it `nh`. Once we've loaded it we can type the name of the object itself (`nh`) to see it printed to the screen.

```
nh <- read_csv(file="data/nhanes.csv")
nh
```

```
# A tibble: 5,000 x 32
      id Gender   Age Race    Education MaritalStatus RelationshipStatus Insured
   <dbl> <chr>  <dbl> <chr>   <chr>     <chr>         <chr>               <chr>
 1 62163 male      14 Asian   <NA>      <NA>          <NA>                Yes
 2 62172 female    43 Black   High Sch~ NeverMarried  Single              Yes
 3 62174 male      80 White   College ~ Married       Committed           Yes
 4 62174 male      80 White   College ~ Married       Committed           Yes
 5 62175 male       5 White   <NA>      <NA>          <NA>                Yes
 6 62176 female    34 White   College ~ Married       Committed           Yes
 7 62178 male      80 White   High Sch~ Widowed       Single              Yes
 8 62180 male      35 White   College ~ Married       Committed           Yes
 9 62186 female    17 Black   <NA>      <NA>          <NA>                Yes
10 62190 female    15 Mexican <NA>      <NA>          <NA>                Yes
# i 4,990 more rows
# i 24 more variables: Income <dbl>, Poverty <dbl>, HomeRooms <dbl>,
#   HomeOwn <chr>, Work <chr>, Weight <dbl>, Height <dbl>, BMI <dbl>,
#   Pulse <dbl>, BPSys <dbl>, BPDia <dbl>, Testosterone <dbl>, HDLChol <dbl>,
#   TotChol <dbl>, Diabetes <chr>, DiabetesAge <dbl>, nPregnancies <dbl>,
#   nBabies <dbl>, SleepHrsNight <dbl>, PhysActive <chr>, PhysActiveDays <dbl>,
#   AlcoholDay <dbl>, AlcoholYear <dbl>, SmokingStatus <chr>
```

Take a look at that output. The nice thing about loading dplyr and reading data with readr functions is that data are displayed in a much more friendly way. This dataset has 5,000 rows and 32 columns. When you import/convert data this way and try to display the object in the console, instead of trying to display all 5,000 rows, you'll only see about 10 by default. Also, if you have so many columns that the data would wrap off the edge of your screen, those columns will not be displayed, but you'll see at the bottom of the output which, if any, columns were hidden from view.

> ***A note on characters versus factors:*** One thing that you immediately notice is that all the categorical variables are read in as *character* data types. This data type is used for storing strings of text, for example, IDs, names, descriptive text, etc. There's another related data type called ***factors***. Factor variables are used to represent categorical variables with two or more *levels*, e.g., "male" or "female" for Gender, or "Single" versus "Committed" for RelationshipStatus. For the most part, statistical analysis treats these two data types the same. It's often easier to

leave categorical variables as characters. However, in some cases you may get a warning message alerting you that a character variable was converted into a factor variable during analysis. Generally, these warnings are nothing to worry about. You can, if you like, convert individual variables to factor variables, or simply use dplyr's `mutate_if` to convert all character vectors to factor variables:

```
nh <- nh |> mutate_if(is.character, as.factor)
nh
```

Now just take a look at just a few columns that are now factors. Remember, you can look at individual variables with the `mydataframe$specificVariable` syntax.

```
nh$RelationshipStatus
nh$Race
levels(nh$Race)
```

If you want to see the whole dataset, there are two ways to do this. First, you can click on the name of the data.frame in the **Environment** panel in RStudio. Or you could use the `View()` function (*with a capital V*).

```
View(nh)
```

Recall several built-in functions that are useful for working with data frames.

- Content:
  - `head()`: shows the first few rows
  - `tail()`: shows the last few rows
- Size:
  - `dim()`: returns a 2-element vector with the number of rows in the first element, and the number of columns as the second element (the dimensions of the object)
  - `nrow()`: returns the number of rows
  - `ncol()`: returns the number of columns
- Summary:
  - `colnames()` (or just `names()`): returns the column names
  - `glimpse()` (from **dplyr**): Returns a glimpse of your data, telling you the structure of the dataset and information about the class, length and content of each column

```
head(nh)
tail(nh)
dim(nh)
names(nh)
```

```
glimpse(nh)
```

8.2 Descriptive statistics

We can access individual variables within a data frame using the `$` operator, e.g., `mydataframe$specificVariable`. Let's print out all the **`Race`** values in the data. Let's then see what are the **`unique`** values of each. Then let's calculate the **`mean`**, **`median`**, and **`range`** of the **`Age`** variable.

```
# Display all Race values
nh$Race

# Get the unique values of Race
unique(nh$Race)
length(unique(nh$Race))
# Do the same thing the dplyr way
nh$Race |> unique()
nh$Race |> unique() |> length()

# Age mean, median, range
mean(nh$Age)
median(nh$Age)
range(nh$Age)
```

You could also do the last few operations using dplyr, but remember, this returns a single-row, single-column tibble, *not* a single scalar value like the above. This is only really useful in the context of grouping and summarizing.

```
# Compute the mean age
nh |>
  summarize(mean(Age))

# Now grouped by other variables
nh |>
  group_by(Gender, Race) |>
  summarize(mean(Age))
```

The `summary()` function (note, this is different from **dplyr**'s `summarize()`) works differently depending on which kind of object you pass to it. If you run `summary()` on a data frame, you get some very basic summary statistics on each variable in the data.

```r
summary(nh)
```

8.2.1 Missing data

Let's try taking the mean of a different variable, either the dplyr way or the simpler `$` way.

```r
# the dplyr way: returns a single-row single-column tibble/dataframe
nh |> summarize(mean(Income))

# returns a single value
mean(nh$Income)
```

What happened there? `NA` indicates *missing data*. Take a look at the Income variable.

```r
# Look at just the Income variable
nh$Income

# Or view the dataset
# View(nh)
```

Notice that there are lots of missing values for Income. Trying to get the mean a bunch of observations with some missing data returns a missing value by default. This is almost universally the case with all summary statistics – a single `NA` will cause the summary to return `NA`. Now look at the help for `?mean`. Notice the `na.rm` argument. This is a logical (i.e., `TRUE` or `FALSE`) value indicating whether or not missing values should be removed prior to computing the mean. By default, it's set to `FALSE`. Now try it again.

```r
mean(nh$Income, na.rm=TRUE)
```

```
[1] 57078
```

The `is.na()` function tells you if a value is missing. Get the `sum()` of that vector, which adds up all the `TRUE`s to tell you how many of the values are missing.

```r
is.na(nh$Income)
sum(is.na(nh$Income))
```

Now, let's talk about exploratory data analysis (EDA).

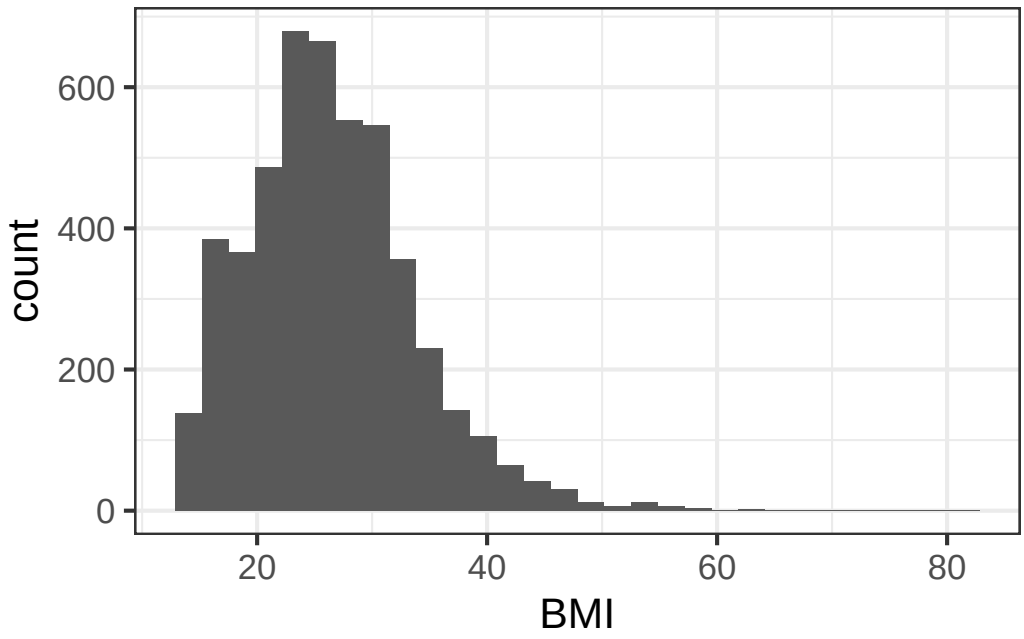
8.2.2 EDA

It's always worth examining your data visually before you start any statistical analysis or hypothesis testing. We could spend an entire day on **exploratory data analysis**. The data visualization section (Chapter 5) covers this in much broader detail. Here we'll just mention a few of the big ones: **histograms** and **scatterplots**.

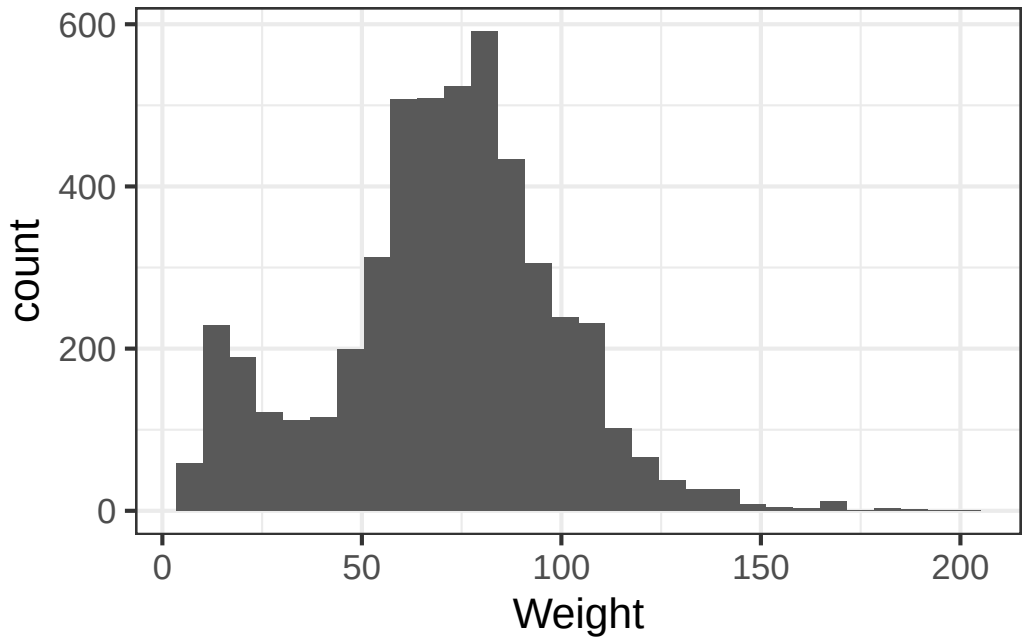
8.2.2.1 Histograms

We can learn a lot from the data just looking at the value distributions of particular variables. Let's make some histograms with ggplot2. Looking at BMI shows a few extreme outliers. Looking at weight initially shows us that the units are probably in kg. Replotting that in lbs with more bins shows a clear bimodal distribution. Are there kids in this data? The age distribution shows us the answer is *yes*.

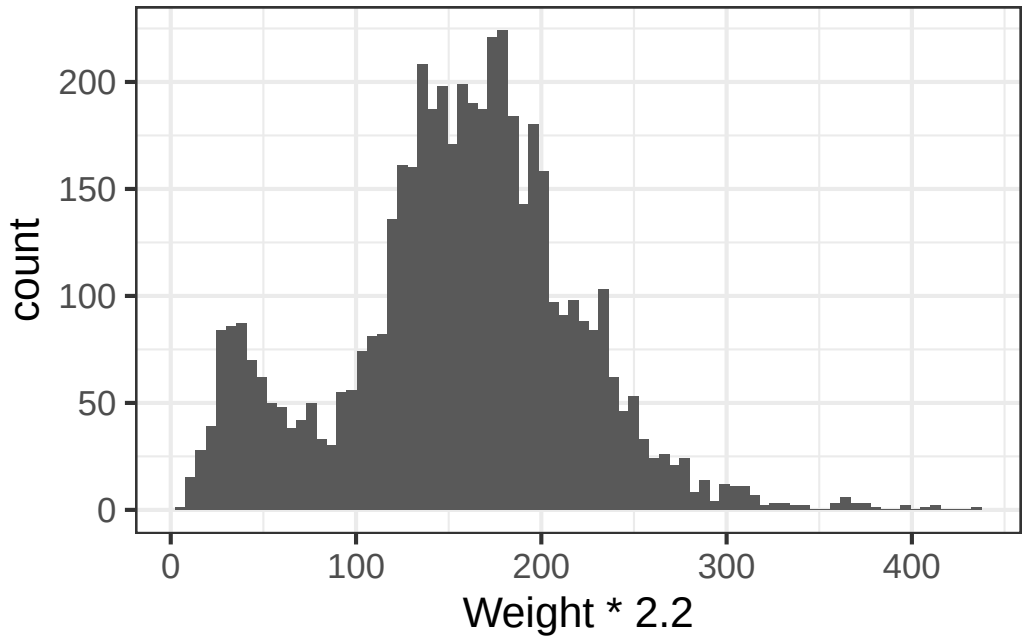
```
library(ggplot2)
ggplot(nh, aes(BMI)) + geom_histogram(bins=30)
```



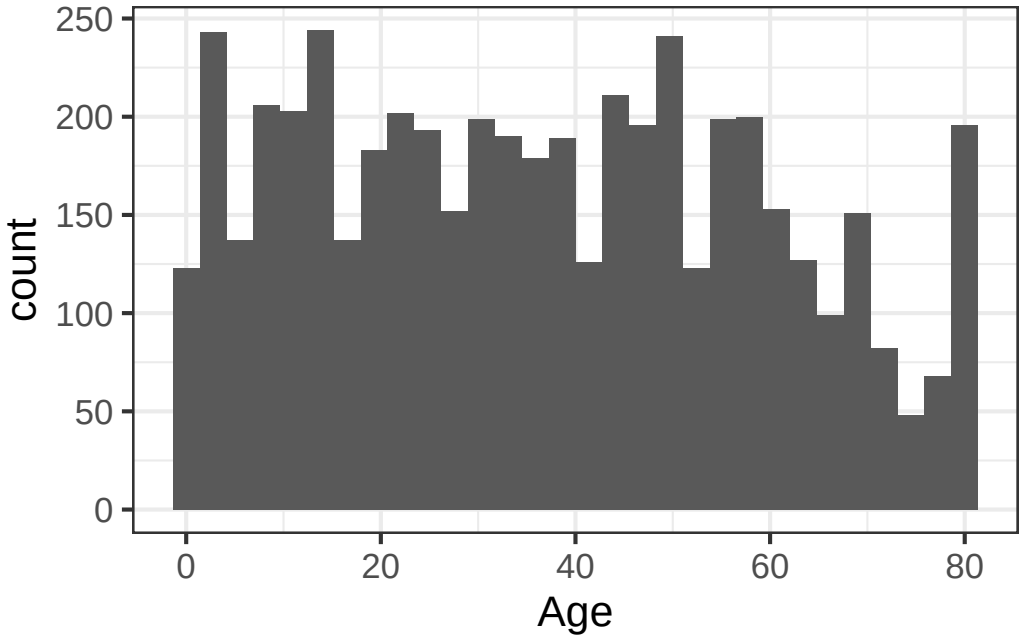

```
ggplot(nh, aes(Weight)) + geom_histogram(bins=30)
```




```
# In pounds, more bins
ggplot(nh, aes(Weight*2.2)) + geom_histogram(bins=80)
```



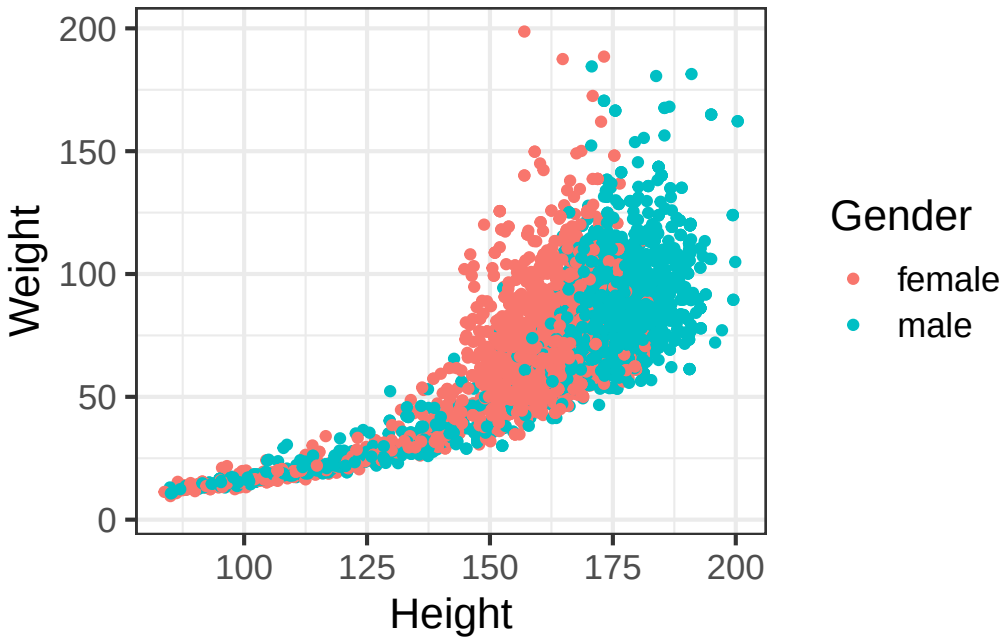
```
ggplot(nh, aes(Age)) + geom_histogram(bins=30)
```



8.2.2.2 Scatterplots

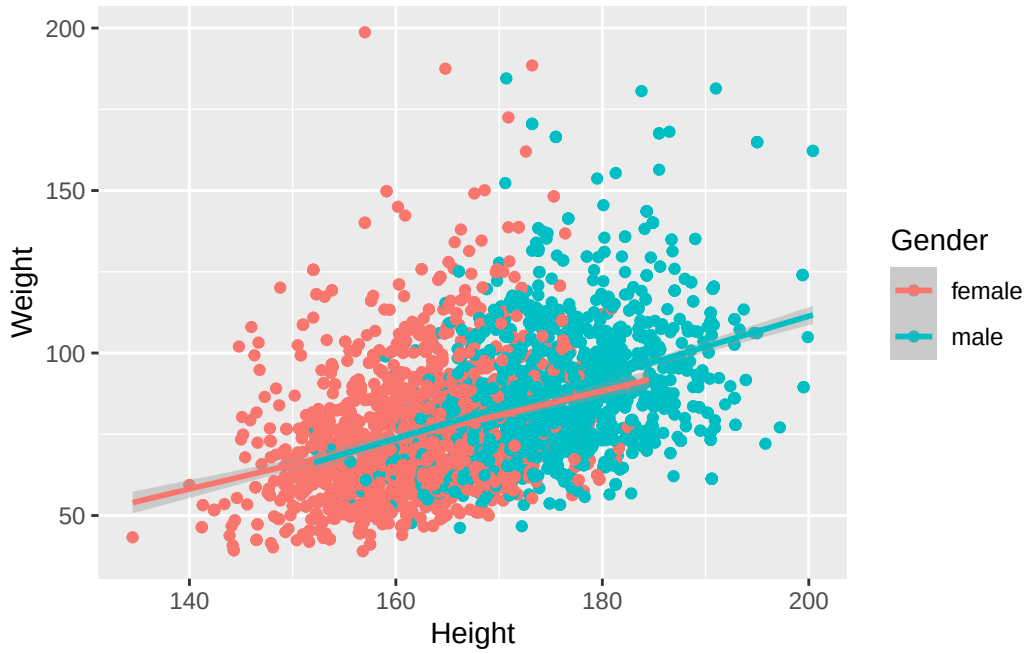
Let's look at how a few different variables relate to each other. E.g., height and weight:

```
ggplot(nh, aes(Height, Weight, col=Gender)) + geom_point()
```



Let's filter out all the kids, draw trend lines using a linear model:

```
nh |>
  filter(Age>=18) |>
  ggplot(aes(Height, Weight, col=Gender)) +
    geom_point() +
    geom_smooth(method="lm")
```



Check out the data visualization section (Chapter 5) for much more on this topic.

Exercise 1

What's the mean 60-second pulse rate for all participants in the data?

```
[1] 73.6
```

Exercise 2

What's the range of values for diastolic blood pressure in all participants? (Hint: see help for `min()`, `max()`, and `range()` functions, e.g., enter `?range` without the parentheses to get help).

```
[1]   0 116
```

Exercise 3

What are the median, lower, and upper quartiles for the age of all participants? (Hint: see help for `median`, or better yet, `quantile`).

```
 0%  25%  50%  75% 100%
  0   17   36   54   80
```

Exercise 4

What's the variance and standard deviation for income among all participants?

```
[1] 1.12e+09
```

```
[1] 33490
```

8.3 Continuous variables

8.3.1 T-tests

First let's create a new dataset from **nh** called **nha** that only has adults. To prevent us from making any mistakes downstream, let's remove the **nh** object.

```
nha <- filter(nh, Age>=18)
rm(nh)
# View(nha)
```

Let's do a few two-sample t-tests to test for *differences in means between two groups*. The function for a t-test is **t.test()**. See the help for **?t.test**. We'll be using the *forumla* method. The usage is **t.test(response~group, data=myDataFrame)**.

1. Are there differences in age for males versus females in this dataset?
2. Does BMI differ between diabetics and non-diabetics?
3. Do single or married/cohabitating people drink more alcohol? Is this relationship significant?

```
t.test(Age~Gender, data=nha)
```

```

    Welch Two Sample t-test

data:  Age by Gender
t = 2, df = 3697, p-value = 0.06
alternative hypothesis: true difference in means between group female and group male is not e
95 percent confidence interval:
 -0.0278  2.2219
sample estimates:
mean in group female   mean in group male
                47.1                 46.0
```

```r
t.test(BMI~Diabetes, data=nha)
```

```
    Welch Two Sample t-test

data:  BMI by Diabetes
t = -11, df = 407, p-value <2e-16
alternative hypothesis: true difference in means between group No and group Yes is not equal
95 percent confidence interval:
 -5.56 -3.92
sample estimates:
 mean in group No mean in group Yes
             28.1              32.8
```

```r
t.test(AlcoholYear~RelationshipStatus, data=nha)
```

```
    Welch Two Sample t-test

data:  AlcoholYear by RelationshipStatus
t = 5, df = 2675, p-value = 6e-08
alternative hypothesis: true difference in means between group Committed and group Single is
95 percent confidence interval:
 13.1 27.8
sample estimates:
mean in group Committed    mean in group Single
                   83.9                    63.5
```

See the heading, *Welch Two Sample t-test*, and notice that the degrees of freedom might not be what we expected based on our sample size. Now look at the help for `?t.test` again, and look at the `var.equal` argument, which is by default set to `FALSE`. One of the assumptions of the t-test is homoscedasticity, or homogeneity of variance. This assumes that the variance in the outcome (e.g., BMI) is identical across both levels of the predictor (diabetic vs non-diabetic). Since this is rarely the case, the t-test defaults to using the Welch correction, which is a more reliable version of the t-test when the homoscedasticity assumption is violated.

> ***A note on one-tailed versus two-tailed tests:*** A two-tailed test is almost always more appropriate. The hypothesis you're testing here is spelled out in the results ("alternative hypothesis: true difference in means is not equal to 0"). If the p-value is very low, you can reject the null hypothesis that there's no difference in

means. Because you typically don't know *a priori* whether the difference in means will be positive or negative (e.g., we don't know *a priori* whether Single people would be expected to drink more or less than those in a committed relationship), we want to do the two-tailed test. However, if we *only* wanted to test a very specific directionality of effect, we could use a one-tailed test and specify which direction we expect. This is more powerful if we "get it right", but much less powerful for the opposite effect. Notice how the p-value changes depending on how we specify the hypothesis. Again, the **two-tailed test is almost always more appropriate**.

```
# Two tailed
t.test(AlcoholYear~RelationshipStatus, data=nha)

# Difference in means is >0 (committed drink more)
t.test(AlcoholYear~RelationshipStatus, data=nha, alternative="greater")

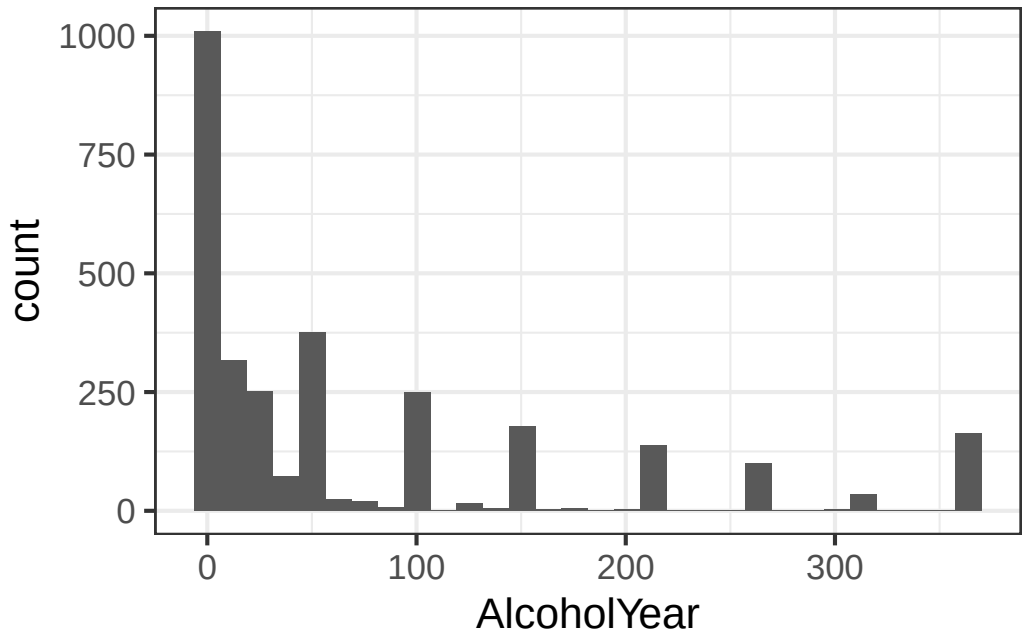
# Difference in means is <0 (committed drink less)
t.test(AlcoholYear~RelationshipStatus, data=nha, alternative="less")
```

A note on paired versus unpaired t-tests: The t-test we performed here was an unpaired test. Here the males and females are different people. The diabetics and nondiabetics are different samples. The single and committed individuals are completely independent, separate observations. In this case, an *unpaired* test is appropriate. An alternative design might be when data is derived from samples who have been measured at two different time points or locations, e.g., before versus after treatment, left versus right hand, etc. In this case, a ***paired t-test*** would be more appropriate. A paired test takes into consideration the intra and inter-subject variability, and is more powerful than the unpaired test. See the help for `?t.test` for more information on how to do this.

8.3.2 Wilcoxon test

Another assumption of the t-test is that data is normally distributed. Looking at the histogram for AlcoholYear shows that this data clearly isn't.

```
ggplot(nha, aes(AlcoholYear)) + geom_histogram()
```



The Wilcoxon rank-sum test (a.k.a. Mann-Whitney *U* test) is a nonparametric test of differences in mean that does not require normally distributed data. When data is perfectly normal, the t-test is uniformly more powerful. But when this assumption is violated, the t-test is unreliable. This test is called in a similar way as the t-test.

```
wilcox.test(AlcoholYear~RelationshipStatus, data=nha)
```

```
    Wilcoxon rank sum test with continuity correction

data:  AlcoholYear by RelationshipStatus
W = 1e+06, p-value = 2e-04
alternative hypothesis: true location shift is not equal to 0
```

The results are still significant, but much less than the p-value reported for the (incorrect) t-test above. Also note in the help for `?wilcox.test` that there's a `paired` option here too.

8.3.3 Linear models

Analysis of variance and linear modeling are complex topics that deserve an entire semester dedicated to theory, design, and interpretation. A very good resource is *An Introduction to Statistical Learning: with Applications in R* by Gareth James,

> Daniela Witten, Trevor Hastie and Robert Tibshirani. The PDF of the book and all the R code used throughout are available **free** on the author's website. What follows is a necessary over-simplification with more focus on implementation, and less on theory and design.

Where t-tests and their nonparametric substitutes are used for assessing the differences in means between two groups, ANOVA is used to assess the significance of differences in means between multiple groups. In fact, a t-test is just a specific case of ANOVA when you only have two groups. And both t-tests and ANOVA are just specific cases of linear regression, where you're trying to fit a model describing how a continuous outcome (e.g., BMI) changes with some predictor variable (e.g., diabetic status, race, age, etc.). The distinction is largely semantic – with a linear model you're asking, "do levels of a categorical variable affect the response?" where with ANOVA or t-tests you're asking, "does the mean response differ between levels of a categorical variable?"

Let's examine the relationship between BMI and relationship status (`RelationshipStatus` was derived from `MaritalStatus`, coded as *Committed* if MaritalStatus is Married or LivePartner, and *Single* otherwise). Let's first do this with a t-test, and for now, let's assume that the variances between groups *are* equal.

```
t.test(BMI~RelationshipStatus, data=nha, var.equal=TRUE)
```

```
    Two Sample t-test

data:  BMI by RelationshipStatus
t = -2, df = 3552, p-value = 0.1
alternative hypothesis: true difference in means between group Committed and group Single is
95 percent confidence interval:
 -0.7782  0.0955
sample estimates:
mean in group Committed    mean in group Single
                   28.5                    28.9
```

It looks like single people have a very slightly higher BMI than those in a committed relationship, but the magnitude of the difference is trivial, and the difference is not significant. Now, let's do the same test in a linear modeling framework. First, let's create the fitted model and store it in an object called `fit`.

```
fit <- lm(BMI~RelationshipStatus, data=nha)
```

You can display the object itself, but that isn't too interesting. You can get the more familiar ANOVA table by calling the `anova()` function on the `fit` object. More generally, the

`summary()` function on a linear model object will tell you much more. (Note this is different from dplyr's **summarize** function).

```
fit
```

```
Call:
lm(formula = BMI ~ RelationshipStatus, data = nha)

Coefficients:
             (Intercept)  RelationshipStatusSingle
                  28.513                     0.341
```

```
anova(fit)
```

```
Analysis of Variance Table

Response: BMI
                     Df Sum Sq Mean Sq F value Pr(>F)
RelationshipStatus    1     98    98.3    2.35   0.13
Residuals          3552 148819    41.9
```

```
summary(fit)
```

```
Call:
lm(formula = BMI ~ RelationshipStatus, data = nha)

Residuals:
   Min     1Q Median     3Q    Max
-12.81  -4.61  -0.95   3.29  52.09

Coefficients:
                          Estimate Std. Error t value Pr(>|t|)
(Intercept)                 28.513      0.139  205.44   <2e-16
RelationshipStatusSingle     0.341      0.223    1.53     0.13

Residual standard error: 6.47 on 3552 degrees of freedom
  (153 observations deleted due to missingness)
Multiple R-squared:  0.00066,   Adjusted R-squared:  0.000379
F-statistic: 2.35 on 1 and 3552 DF,  p-value: 0.126
```

Go back and re-run the t-test assuming equal variances as we did before. Now notice a few things:

```
t.test(BMI~RelationshipStatus, data=nha, var.equal=TRUE)
```

1. The p-values from all three tests (t-test, ANOVA, and linear regression) are all identical (p=0.1256). This is because they're all identical: a t-test is a specific case of ANOVA, which is a specific case of linear regression. There may be some rounding error, but we'll talk about extracting the exact values from a model object later on.
2. The test statistics are all related. The t statistic from the t-test is **1.532**, which is the same as the t-statistic from the linear regression. If you square that, you get **2.347**, the F statistic from the ANOVA.
3. The `t.test()` output shows you the means for the two groups, Committed and Single. Just displaying the `fit` object itself or running `summary(fit)` shows you the coefficients for a linear model. Here, the model assumes the "baseline" RelationshipStatus level is *Committed*, and that the *intercept* in a regression model (e.g., β_0 in the model $Y = \beta_0 + \beta_1 X$) is the mean of the baseline group. Being *Single* results in an increase in BMI of 0.3413. This is the β_1 coefficient in the model. You can easily change the ordering of the levels. See the help for `?factor`, and check out the new **forcats** package, which provides tools **for** manipulating **cat**egorical variables.

```
# P-value computed on a t-statistic with 3552 degrees of freedom
# (multiply times 2 because t-test is assuming two-tailed)
2*(1-pt(1.532, df=3552))
```

```
[1] 0.126
```

```
# P-value computed on an F-test with 1 and 3552 degrees of freedom
1-pf(2.347, df1=1, df2=3552)
```

```
[1] 0.126
```

> ***A note on dummy coding:*** If you have a k-level factor, R creates $k-1$ dummy variables, or indicator variables, by default, using the alphabetically first level as baseline. For example, the levels of RelationshipStatus are "Committed" and "Single". R creates a dummy variable called "RelationshipStatusSingle" that's **0** if you're committed, and **1** if you're Single. The linear model is saying for every unit increase in RelationshipStatusSingle, i.e., going from committed to single, results in a 0.314-unit increase in BMI. You can change the ordering of the factors to change the interpretation of the model (e.g., treating Single as baseline and going from Single to Committed). We'll do this in the next section.

8.3.4 ANOVA

Recap: t-tests are for assessing the differences in means between *two* groups. A t-test is a specific case of ANOVA, which is a specific case of a linear model. Let's run ANOVA, but this time looking for differences in means between more than two groups.

Let's look at the relationship between smoking status (Never, Former, or Current), and BMI.

```
fit <- lm(BMI~SmokingStatus, data=nha)
anova(fit)
```

```
Analysis of Variance Table

Response: BMI
                Df Sum Sq Mean Sq F value  Pr(>F)
SmokingStatus    2   1411     706      17 4.5e-08
Residuals     3553 147551      42
```

```
summary(fit)
```

```
Call:
lm(formula = BMI ~ SmokingStatus, data = nha)

Residuals:
   Min     1Q Median     3Q    Max
-12.56  -4.56  -1.06   3.32  51.74

Coefficients:
                   Estimate Std. Error t value Pr(>|t|)
(Intercept)          27.391      0.245  111.97  < 2e-16
SmokingStatusFormer   1.774      0.329    5.39  7.6e-08
SmokingStatusNever    1.464      0.284    5.16  2.6e-07

Residual standard error: 6.44 on 3553 degrees of freedom
  (151 observations deleted due to missingness)
Multiple R-squared:  0.00947,   Adjusted R-squared:  0.00891
F-statistic:   17 on 2 and 3553 DF,  p-value: 4.54e-08
```

The F-test on the ANOVA table tells us that there *is* a significant difference in means between current, former, and never smokers ($p=4.54 \times 10^{-8}$). However, the linear model output might

not have been what we wanted. Because the default handling of categorical variables is to treat the alphabetical first level as the baseline, "Current" smokers are treated as baseline, and this mean becomes the intercept, and the coefficients on "Former" and "Never" describe how those groups' means differ from current smokers.

Back to dummy coding / indicator variables: SmokingStatus is "Current", "Former", and "Never." By default, R will create *two* indicator variables here that in tandem will explain this variable.

Original SmokingStatus	Indicator: SmokingStatusFormer	Indicator: SmokingStatusNever
Current	0	0
Former	1	0
Never	0	1

What if we wanted "Never" smokers to be the baseline, followed by Former, then Current? Have a look at `?factor` to relevel the factor levels.

```
# Look at nha$SmokingStatus
nha$SmokingStatus

# What happens if we relevel it? Let's see what that looks like.
relevel(nha$SmokingStatus, ref="Never")

# If we're happy with that, let's change the value of nha$SmokingStatus in place
nha$SmokingStatus <- relevel(nha$SmokingStatus, ref="Never")

# Or we could do this the dplyr way
nha <- nha |>
  mutate(SmokingStatus=relevel(SmokingStatus, ref="Never"))
```

```
# Re-fit the model
fit <- lm(BMI~SmokingStatus, data=nha)

# Optionally, show the ANOVA table
# anova(fit)

# Print the full model statistics
summary(fit)
```

```
Call:
lm(formula = BMI ~ SmokingStatus, data = nha)

Residuals:
   Min     1Q Median     3Q    Max
-12.56  -4.56  -1.06   3.32  51.74

Coefficients:
                     Estimate Std. Error t value Pr(>|t|)
(Intercept)            28.856      0.144  200.60  < 2e-16
SmokingStatusCurrent   -1.464      0.284   -5.16  2.6e-07
SmokingStatusFormer     0.309      0.263    1.17     0.24

Residual standard error: 6.44 on 3553 degrees of freedom
  (151 observations deleted due to missingness)
Multiple R-squared:  0.00947,   Adjusted R-squared:  0.00891
F-statistic:   17 on 2 and 3553 DF,  p-value: 4.54e-08
```

Notice that the p-value on the ANOVA/regression didn't change, but the coefficients did. *Never* smokers are now treated as baseline. The intercept coefficient (28.856) is now the mean for *Never* smokers. The `SmokingStatusFormer` coefficient of .309 shows the apparent increase in BMI that former smokers have when compared to never smokers, but that difference is not significant (p=.24). The `SmokingStatusCurrent` coefficient of -1.464 shows that current smokers actually have a lower BMI than never smokers, and that this decrease is highly significant.

Finally, you can do the typical post-hoc ANOVA procedures on the fit object. For example, the `TukeyHSD()` function will run *Tukey's test* (also known as *Tukey's range test*, the *Tukey method*, *Tukey's honest significance test*, *Tukey's HSD test* (honest significant difference), or the *Tukey-Kramer method*). Tukey's test computes all pairwise mean difference calculation, comparing each group to each other group, identifying any difference between two groups that's greater than the standard error, while controlling the type I error for all multiple comparisons. First run `aov()` (**not** `anova()`) on the fitted linear model object, then run `TukeyHSD()` on the resulting analysis of variance fit.

```
TukeyHSD(aov(fit))
```

```
  Tukey multiple comparisons of means
    95% family-wise confidence level

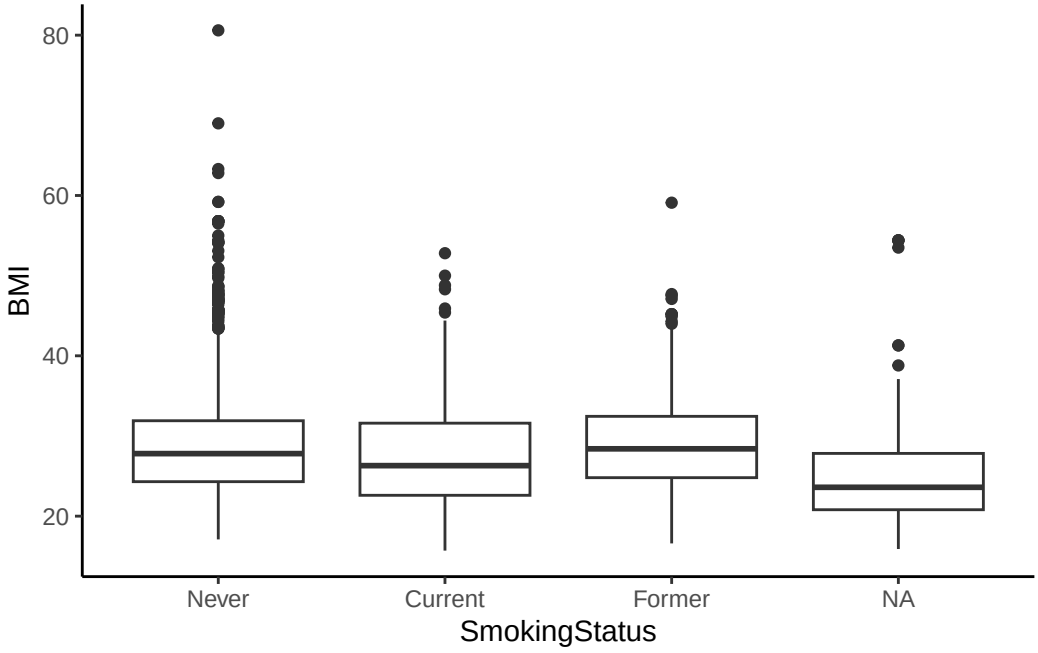
Fit: aov(formula = fit)
```

```
$SmokingStatus
                 diff    lwr    upr p adj
Current-Never  -1.464 -2.130 -0.799 0.000
Former-Never    0.309 -0.308  0.926 0.469
Former-Current  1.774  1.002  2.546 0.000
```

This shows that there isn't much of a difference between former and never smokers, but that both of these differ significantly from current smokers, who have significantly lower BMI.

Finally, let's visualize the differences in means between these groups. The **NA** category, which is omitted from the ANOVA, contains all the observations who have missing or non-recorded Smoking Status.

```
ggplot(nha, aes(SmokingStatus, BMI)) + geom_boxplot() + theme_classic()
```



8.3.5 Linear regression

Linear models are mathematical representations of the process that (*we think*) gave rise to our data. The model seeks to explain the relationship between a variable of interest, our Y, *outcome*, *response*, or *dependent* variable, and one or more X, *predictor*, or *independent* variables. Previously we talked about t-tests or ANOVA in the context of a simple linear regression model with only a single predictor variable, X:

$$Y = \beta_0 + \beta_1 X$$

But you can have multiple predictors in a linear model that are all additive, accounting for the effects of the others:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \epsilon$$

- Y is the response
- X_1 and X_2 are the predictors
- β_0 is the intercept, and β_1, β_2 etc are *coefficients* that describe what 1-unit changes in X_1 and X_2 do to the outcome variable Y.
- ϵ is random error. Our model will not perfectly predict Y. It will be off by some random amount. We assume this amount is a random draw from a Normal distribution with mean 0 and standard deviation σ.

Building a linear model means we propose a linear model and then estimate the coefficients and the variance of the error term. Above, this means estimating $\beta_0, \beta_1, \beta_2$ and σ. This is what we do in R.

Let's look at the relationship between height and weight.

```
fit <- lm(Weight~Height, data=nha)
summary(fit)
```

```
Call:
lm(formula = Weight ~ Height, data = nha)

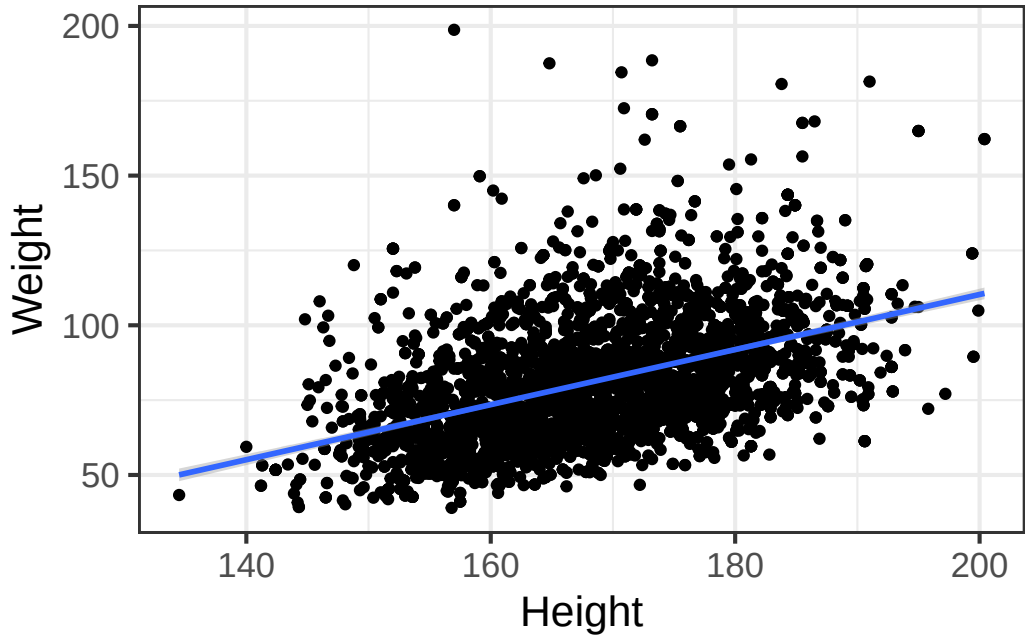
Residuals:
   Min     1Q Median     3Q    Max
-40.34 -13.11  -2.66   9.31 127.97

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   -73.71       5.08   -14.5   <2e-16
Height          0.92       0.03    30.6   <2e-16

Residual standard error: 18.6 on 3674 degrees of freedom
  (31 observations deleted due to missingness)
Multiple R-squared:  0.203, Adjusted R-squared:  0.203
F-statistic:  938 on 1 and 3674 DF,  p-value: <2e-16
```

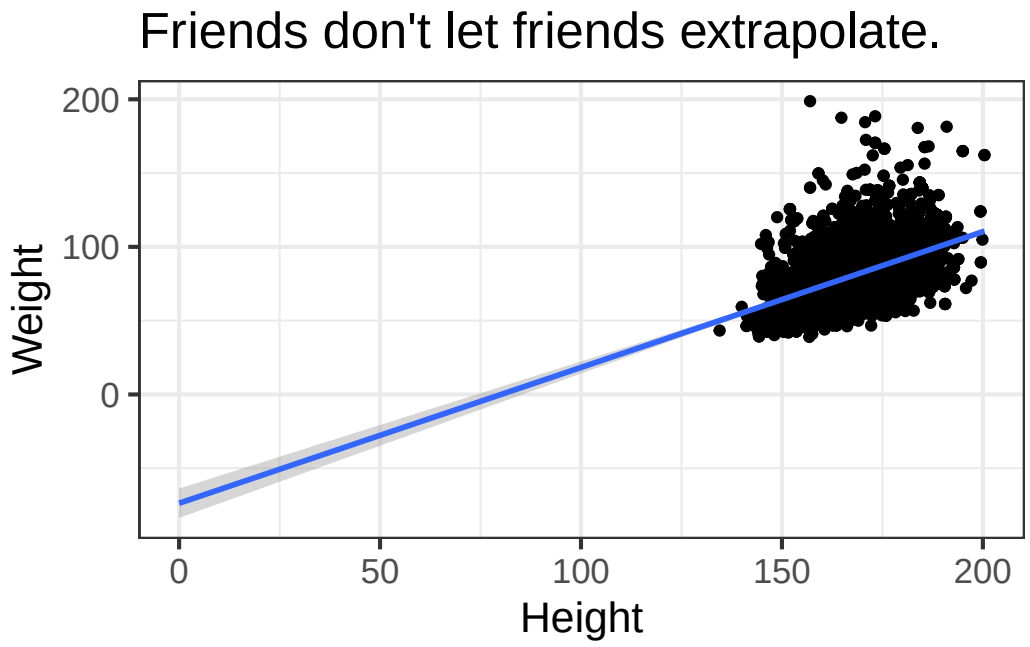
The relationship is highly significant ($P<2.2 \times 10^{-16}$). The intercept term is not very useful most of the time. Here it shows us what the value of Weight would be when Height=0, which could never happen. The Height coefficient is meaningful – each one unit increase in height results in a 0.92 increase in the corresponding unit of weight. Let's visualize that relationship:

```
ggplot(nha, aes(x=Height, y=Weight)) + geom_point() + geom_smooth(method="lm")
```



By default, this is only going to show the prediction over the range of the data. This is important! You never want to try to extrapolate response variables outside of the range of your predictor(s). For example, the linear model tells us that weight is -73.7kg when height is zero. We can extend the predicted model / regression line past the lowest value of the data down to height=0. The bands on the confidence interval tell us that the model is apparently confident within the regions defined by the gray boundary. But this is silly – we would never see a height of zero, and predicting past the range of the available training data is never a good idea.

```
ggplot(nha, aes(x=Height, y=Weight)) +
  geom_point() +
  geom_smooth(method="lm", fullrange=TRUE) +
  xlim(0, NA) +
  ggtitle("Friends don't let friends extrapolate.")
```



8.3.6 Multiple regression

Finally, let's do a multiple linear regression analysis, where we attempt to model the effect of multiple predictor variables at once on some outcome. First, let's look at the effect of physical activity on testosterone levels. Let's do this with a t-test and linear regression, showing that you get the same results.

```
t.test(Testosterone~PhysActive, data=nha, var.equal=TRUE)
```

```
    Two Sample t-test

data:  Testosterone by PhysActive
t = -2, df = 3436, p-value = 0.02
alternative hypothesis: true difference in means between group No and group Yes is not equal
95 percent confidence interval:
 -34.81  -3.72
sample estimates:
 mean in group No mean in group Yes
              208               227
```

```
summary(lm(Testosterone~PhysActive, data=nha))
```

```
Call:
lm(formula = Testosterone ~ PhysActive, data = nha)

Residuals:
   Min     1Q Median     3Q    Max
  -224   -196   -116    167   1588

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)     207.56       5.87   35.34   <2e-16
PhysActiveYes    19.27       7.93    2.43    0.015

Residual standard error: 231 on 3436 degrees of freedom
  (269 observations deleted due to missingness)
Multiple R-squared:  0.00172,   Adjusted R-squared:  0.00142
F-statistic:  5.9 on 1 and 3436 DF,  p-value: 0.0152
```

In both cases, the p-value is significant (p=0.01516), and the result suggest that increased physical activity is associated with increased testosterone levels. Does increasing your physical activity increase your testosterone levels? Or is it the other way – will increased testosterone encourage more physical activity? Or is it none of the above – is the apparent relationship between physical activity and testosterone levels only apparent because both are correlated with yet a third, unaccounted for variable? Let's throw Age into the model as well.

```
summary(lm(Testosterone~PhysActive+Age, data=nha))
```

```
Call:
lm(formula = Testosterone ~ PhysActive + Age, data = nha)

Residuals:
   Min     1Q Median     3Q    Max
  -239   -197   -112    167   1598

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)   247.883     13.085   18.94  < 2e-16
PhysActiveYes  13.674      8.081    1.69  0.09073
Age            -0.800      0.232   -3.45  0.00057

Residual standard error: 231 on 3435 degrees of freedom
```

```
  (269 observations deleted due to missingness)
Multiple R-squared:  0.00516,   Adjusted R-squared:  0.00458
F-statistic:  8.9 on 2 and 3435 DF,  p-value: 0.000139
```

This shows us that after accounting for age that the testosterone / physical activity link is no longer significant. Every 1-year increase in age results in a highly significant decrease in testosterone, and since increasing age is also likely associated with decreased physical activity, perhaps age is the confounder that makes this relationship apparent.

Adding other predictors can also swing things the other way. We know that men have much higher testosterone levels than females. Sex is probably the single best predictor of testosterone levels in our dataset. By not accounting for this effect, our unaccounted-for variation remains very high. By accounting for Gender, we now reduce the residual error in the model, and the physical activity effect once again becomes significant. Also notice that our model fits much better (higher R-squared), and is much more significant overall.

```
summary(lm(Testosterone ~ PhysActive+Age+Gender, data=nha))
```

```
Call:
lm(formula = Testosterone ~ PhysActive + Age + Gender, data = nha)

Residuals:
   Min     1Q Median     3Q    Max
-397.9  -31.0   -4.4   20.5 1400.9

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)    46.693      7.573    6.17  7.8e-10
PhysActiveYes   9.275      4.462    2.08    0.038
Age            -0.590      0.128   -4.60  4.3e-06
Gendermale    385.199      4.351   88.53  < 2e-16

Residual standard error: 128 on 3434 degrees of freedom
  (269 observations deleted due to missingness)
Multiple R-squared:  0.697, Adjusted R-squared:  0.697
F-statistic: 2.63e+03 on 3 and 3434 DF,  p-value: <2e-16
```

We've only looked at the `summary()` and `anova()` functions for extracting information from an `lm` class object. There are several other accessor functions that can be used on a linear model object. Check out the help page for each one of these to learn more.

- `coefficients()`
- `predict.lm()`
- `fitted.values()`
- `residuals()`

Exercise 5

Is the average BMI different in single people versus those in a committed relationship? Perform a t-test.

Exercise 6

The `Work` variable is coded "Looking" (n=159), "NotWorking" (n=1317), and "Working" (n=2230).

- Fit a linear model of `Income` against `Work`. Assign this to an object called `fit`. What does the `fit` object tell you when you display it directly?
- Run an `anova()` to get the ANOVA table. Is the model significant?
- Run a Tukey test to get the pairwise contrasts. (Hint: `TukeyHSD()` on `aov()` on the fit). What do you conclude?
- Instead of thinking of this as ANOVA, think of it as a linear model. After you've thought about it, get some `summary()` statistics on the fit. Do these results jive with the ANOVA model?

Exercise 7

Examine the relationship between HDL cholesterol levels (`HDLChol`) and whether someone has diabetes or not (`Diabetes`).

- Is there a difference in means between diabetics and nondiabetics? Perform a t-test *without* a Welch correction (that is, assuming equal variances – see `?t.test` for help).
- Do the same analysis in a linear modeling framework.
- Does the relationship hold when adjusting for `Weight`?
- What about when adjusting for `Weight`, `Age`, `Gender`, `PhysActive` (whether someone participates in moderate or vigorous-intensity sports, fitness or recreational activities, coded as yes/no). What is the effect of each of these explanatory variables?

8.4 Discrete variables

Until now we've only discussed analyzing *continuous* outcomes / dependent variables. We've tested for differences in means between two groups with t-tests, differences among means between n groups with ANOVA, and more general relationships using linear regression. In all of these cases, the dependent variable, i.e., the outcome, or Y variable, was *continuous*, and usually normally distributed. What if our outcome variable is *discrete*, e.g., "Yes/No", "Mutant/WT", "Case/Control", etc.? Here we use a different set of procedures for assessing significant associations.

8.4.1 Contingency tables

The `xtabs()` function is useful for creating contingency tables from categorical variables. Let's create a gender by diabetes status contingency table, and assign it to an object called **`xt`**. After making the assignment, type the name of the object to view it.

```
xt <- xtabs(~Gender+Diabetes, data=nha)
xt
```

```
        Diabetes
Gender     No  Yes
  female 1692  164
  male   1653  198
```

There are two useful functions, `addmargins()` and `prop.table()` that add more information or manipulate how the data is displayed. By default, `prop.table()` will divide the number of observations in each cell by the total. But you may want to specify *which margin* you want to get proportions over. Let's do this for the first (row) margin.

```
# Add marginal totals
addmargins(xt)
```

```
        Diabetes
Gender     No  Yes  Sum
  female 1692  164 1856
  male   1653  198 1851
  Sum    3345  362 3707
```

```r
# Get the proportional table
prop.table(xt)
```

```
        Diabetes
Gender       No    Yes
  female 0.4564 0.0442
  male   0.4459 0.0534
```

```r
# That wasn't really what we wanted.
# Do this over the first (row) margin only.
prop.table(xt, margin=1)
```

```
        Diabetes
Gender       No    Yes
  female 0.9116 0.0884
  male   0.8930 0.1070
```

Looks like men have slightly higher rates of diabetes than women. But is this significant?

The chi-square test is used to assess the independence of these two factors. That is, if the null hypothesis that gender and diabetes are independent is true, the we would expect a proportionally equal number of diabetics across each sex. Males seem to be at slightly higher risk than females, but the difference is just short of statistically significant.

```r
chisq.test(xt)
```

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

data:  xt
X-squared = 3, df = 1, p-value = 0.06
```

An alternative to the chi-square test is Fisher's exact test. Rather than relying on a critical value from a theoretical chi-square distribution, Fisher's exact test calculates the *exact* probability of observing the contingency table as is. It's especially useful when there are very small n's in one or more of the contingency table cells. Both the chi-square and Fisher's exact test give us p-values of approximately 0.06.

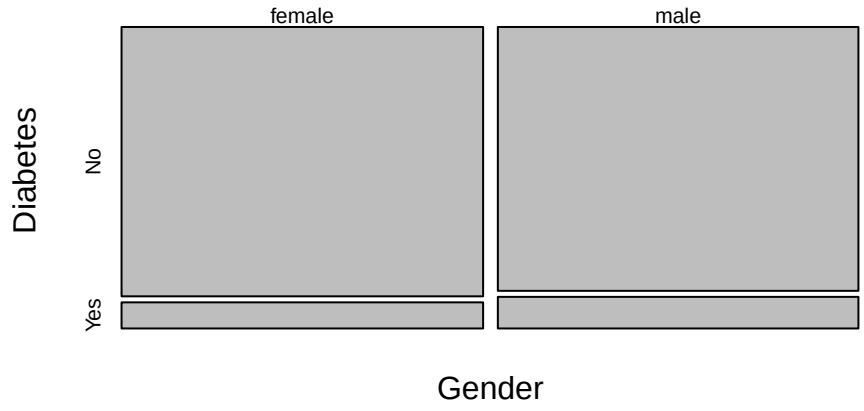
```r
fisher.test(xt)
```

```
    Fisher's Exact Test for Count Data

data:  xt
p-value = 0.06
alternative hypothesis: true odds ratio is not equal to 1
95 percent confidence interval:
 0.988 1.547
sample estimates:
odds ratio
      1.24
```

There's a useful plot for visualizing contingency table data called a *mosaic* plot. Call the `mosaicplot()` function on the contingency table object. Note this is a built-in plot, *not* a ggplot2-style plot.

```
mosaicplot(xt, main=NA)
```



Let's create a different contingency table, this time looking at the relationship between race and whether the person had health insurance. Display the table with marginal totals.

```
xt <- xtabs(~Race+Insured, data=nha)
addmargins(xt)
```

```
          Insured
Race         No  Yes  Sum
  Asian      46  169  215
  Black      86  330  416
  Hispanic   89  151  240
```

```
  Mexican   147  141  288
  Other      33   65   98
  White     307 2141 2448
  Sum       708 2997 3705
```

Let's do the same thing as above, this time showing the proportion of people in each race category having health insurance.

```
prop.table(xt, margin=1)
```

```
          Insured
Race          No   Yes
  Asian    0.214 0.786
  Black    0.207 0.793
  Hispanic 0.371 0.629
  Mexican  0.510 0.490
  Other    0.337 0.663
  White    0.125 0.875
```

Now, let's run a chi-square test for independence.

```
chisq.test(xt)
```

```

    Pearson's Chi-squared test

data:  xt
X-squared = 323, df = 5, p-value <2e-16
```

The result is *highly* significant. In fact, so significant, that the display rounds off the p-value to something like $< 2.2 \times 10^{-16}$. If you look at the help for `?chisq.test` you'll see that displaying the test only shows you summary information, but other components can be accessed. For example, we can easily get the actual p-value, or the expected counts under the null hypothesis of independence.

```
chisq.test(xt)$p.value
```

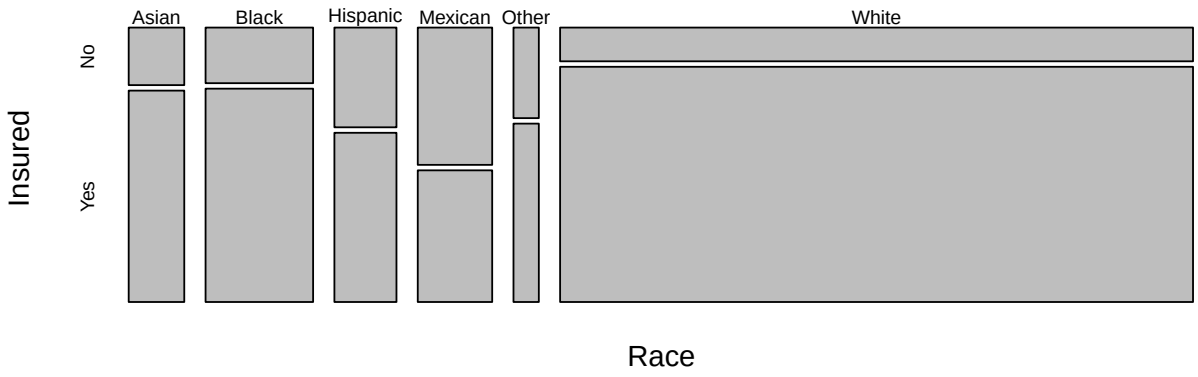
```
[1] 9.75e-68
```

```
chisq.test(xt)$expected
```

```
          Insured
Race          No    Yes
  Asian     41.1  173.9
  Black     79.5  336.5
  Hispanic  45.9  194.1
  Mexican   55.0  233.0
  Other     18.7   79.3
  White    467.8 1980.2
```

We can also make a mosaic plot similar to above:

```
mosaicplot(xt, main=NA)
```



8.4.2 Logistic regression

(See slides)

What if we wanted to model the discrete outcome, e.g., whether someone is insured, against several other variables, similar to how we did with multiple linear regression? We can't use linear regression because the outcome isn't continuous – it's binary, either *Yes* or *No*. For this we'll use *logistic regression* to model the *log odds* of binary response. That is, instead of modeling the outcome variable, Y, directly against the inputs, we'll model the *log odds* of the outcome variable.

If p is the probability that the individual is insured, then $\frac{p}{1-p}$ is the *odds* that person is insured. Then it follows that the linear model is expressed as:

$$log(\frac{p}{1-p}) = \beta_0 + \beta_1 x_1 + \cdots + \beta_k x_k$$

Where β_0 is the intercept, β_1 is the increase in the odds of the outcome for every unit increase in x_1, and so on.

Logistic regression is a type of *generalized linear model* (GLM). We fit GLM models in R using the `glm()` function. It works like the `lm()` function except we specify which GLM to fit using the `family` argument. Logistic regression requires `family=binomial`.

The typical use looks like this:

```
mod <- glm(y ~ x, data=yourdata, family='binomial')
summary(mod)
```

Before we fit a logistic regression model let's *relevel* the Race variable so that "White" is the baseline. We saw above that people who identify as "White" have the highest rates of being insured. When we run the logistic regression, we'll get a separate coefficient (effect) for each level of the factor variable(s) in the model, telling you the increased odds that that level has, *as compared to the baseline group*.

```
#Look at Race. The default ordering is alphabetical
nha$Race

# Let's relevel that where the group with the highest rate of insurance is "baseline"
relevel(nha$Race, ref="White")

# If we're happy with that result, permanently change it
nha$Race <- relevel(nha$Race, ref="White")

# Or do it the dplyr way
nha <- nha |>
  mutate(Race=relevel(Race, ref="White"))
```

Now, let's fit a logistic regression model assessing how the odds of being insured change with different levels of race.

```
fit <- glm(Insured~Race, data=nha, family="binomial")
summary(fit)
```

```
Call:
glm(formula = Insured ~ Race, family = "binomial", data = nha)
```

```
Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept)     1.942      0.061   31.82  < 2e-16
RaceAsian      -0.641      0.177   -3.62    3e-04
RaceBlack      -0.597      0.136   -4.41  1.1e-05
RaceHispanic   -1.413      0.147   -9.62  < 2e-16
RaceMexican    -1.984      0.133  -14.95  < 2e-16
RaceOther      -1.264      0.222   -5.69  1.3e-08

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 3614.6  on 3704  degrees of freedom
Residual deviance: 3336.6  on 3699  degrees of freedom
  (2 observations deleted due to missingness)
AIC: 3349

Number of Fisher Scoring iterations: 4
```

The `Estimate` column shows the log of the odds ratio – how the log odds of having health insurance changes at each level of race compared to White. The P-value for each coefficient is on the far right. This shows that *every* other race has *significantly less* rates of health insurance coverage. But, as in our multiple linear regression analysis above, are there other important variables that we're leaving out that could alter our conclusions? Lets add a few more variables into the model to see if something else can explain the apparent Race-Insured association. Let's add a few things likely to be involved (Age and Income), and something that's probably irrelevant (hours slept at night).

```
fit <- glm(Insured ~ Age+Income+SleepHrsNight+Race, data=nha, family="binomial")
summary(fit)
```

```
Call:
glm(formula = Insured ~ Age + Income + SleepHrsNight + Race,
    family = "binomial", data = nha)

Coefficients:
              Estimate Std. Error z value Pr(>|z|)
(Intercept)  -3.50e-01   2.92e-01   -1.20    0.230
Age           3.37e-02   2.95e-03   11.43  < 2e-16
Income        1.53e-05   1.54e-06    9.98  < 2e-16
SleepHrsNight -1.76e-02   3.52e-02   -0.50    0.616
```

```
RaceAsian     -4.55e-01   2.03e-01   -2.24    0.025
RaceBlack     -2.39e-01   1.54e-01   -1.55    0.120
RaceHispanic  -1.01e+00   1.64e-01   -6.18  6.6e-10
RaceMexican   -1.40e+00   1.48e-01   -9.47  < 2e-16
RaceOther     -9.89e-01   2.42e-01   -4.08  4.5e-05

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 3284.3  on 3395  degrees of freedom
Residual deviance: 2815.0  on 3387  degrees of freedom
  (311 observations deleted due to missingness)
AIC: 2833

Number of Fisher Scoring iterations: 5
```

A few things become apparent:

1. Age and income are both highly associated with whether someone is insured. Both of these variables are highly significant ($P < 2.2 \times 10^{-16}$), and the coefficient (the `Estimate` column) is positive, meaning that for each unit increase in one of these variables, the odds of being insured increases by the corresponding amount.
2. Hours slept per night is not meaningful at all.
3. After accounting for age and income, several of the race-specific differences are no longer statistically significant, but others remain so.
4. The absolute value of the test statistic (column called `z value`) can roughly be taken as an estimate of the "importance" of that variable to the overall model. So, age and income are the most important influences in this model; self-identifying as Hispanic or Mexican are also very highly important, hours slept per night isn't important at all, and the other race categories fall somewhere in between.

There is *much* more to go into with logistic regression. This chapter only scratches the surface. Missing from this chapter are things like regression diagnostics, model comparison approaches, penalization, interpretation of model coefficients, fitting interaction effects, and much more. Alan Agresti's *Categorical Data Analysis* has long been considered the definitive text on this topic. I also recommend Agresti's *Introduction to Categorical Data Analysis* (a.k.a. "Agresti lite") for a gentler introduction.

Exercise 8

What's the relationship between diabetes and participating in rigorous physical activity or sports?

- Create a contingency table with Diabetes status in rows and physical activity status in columns.

- Display that table with margins.
- Show the proportions of diabetics and nondiabetics, separately, who are physically active or not.
- Is this relationship significant?
- Create a mosaic plot to visualize the relationship.

Exercise 9

Model the same association in a logistic regression framework to assess the risk of diabetes using physical activity as a predictor.

- Fit a model with just physical activity as a predictor, and display a model summary.
- Add gender to the model, and show a summary.
- Continue adding weight and age to the model. What happens to the gender association?
- Continue and add income to the model. What happens to the original association with physical activity?

8.5 Power & sample size

This is a necessarily short introduction to the concept of power and sample size calculations. Statistical power, also sometimes called sensitivity, is defined as the probability that your test correctly rejects the null hypothesis when the alternative hypothesis is true. That is, if there really is an effect (difference in means, association between categorical variables, etc.), how likely are you to be able to *detect* that effect at a given statistical significance level, given certain assumptions. Generally there are a few moving pieces, and if you know all but one of them, you can calculate what that last one is.

1. Power: How likely are you to detect the effect? (Usually like to see 80% or greater).
2. N: What is the sample size you have (or require)?
3. Effect size: How big is the difference in means, odds ratio, etc?

If we know we want 80% power to detect a certain magnitude of difference between groups, we can calculate our required sample size. Or, if we know we can only collect 5 samples, we can calculate how likely we are to detect a particular effect. Or, we can work to solve the last one - if we want 80% power and we have 5 samples, what's the smallest effect we can hope to detect?

All of these questions require certain assumptions about the data and the testing procedure. Which kind of test is being performed? What's the true effect size (often unknown, or estimated from preliminary data), what's the standard deviation of samples that will be collected

(often unknown, or estimated from preliminary data), what's the level of statistical significance needed (traditionally p<0.05, but must consider multiple testing corrections).

8.5.1 T-test power/N

The `power.t.test()` empirically estimates power or sample size of a t-test for differences in means. If we have 20 samples in each of two groups (e.g., control versus treatment), and the standard deviation for whatever we're measuring is **2.3**, and we're expecting a true difference in means between the groups of **2**, what's the power to detect this effect?

```
power.t.test(n=20, delta=2, sd=2.3)
```

```

     Two-sample t test power calculation

              n = 20
          delta = 2
             sd = 2.3
      sig.level = 0.05
          power = 0.764
    alternative = two.sided

NOTE: n is number in *each* group
```

What's the sample size we'd need to detect a difference of 0.8 given a standard deviation of 1.5, assuming we want 80% power?

```
power.t.test(power=.80, delta=.8, sd=1.5)
```

```

     Two-sample t test power calculation

              n = 56.2
          delta = 0.8
             sd = 1.5
      sig.level = 0.05
          power = 0.8
    alternative = two.sided

NOTE: n is number in *each* group
```

8.5.2 Proportions power/N

What about a two-sample proportion test (e.g., chi-square test)? If we have two groups (control and treatment), and we're measuring some outcome (e.g., infected yes/no), and we know that the proportion of infected controls is 80% but 20% in treated, what's the power to detect this effect in 5 samples per group?

```
power.prop.test(n=5, p1=0.8, p2=0.2)
```

```
     Two-sample comparison of proportions power calculation

              n = 5
             p1 = 0.8
             p2 = 0.2
      sig.level = 0.05
          power = 0.469
    alternative = two.sided

NOTE: n is number in *each* group
```

How many samples would we need for 90% power?

```
power.prop.test(power=0.9, p1=0.8, p2=0.2)
```

```
     Two-sample comparison of proportions power calculation

              n = 12.4
             p1 = 0.8
             p2 = 0.2
      sig.level = 0.05
          power = 0.9
    alternative = two.sided

NOTE: n is number in *each* group
```

Also check out the **pwr** package which has power calculation functions for other statistical tests.

Function	Power calculations for
`pwr.2p.test()`	Two proportions (equal n)
`pwr.2p2n.test()`	Two proportions (unequal n)
`pwr.anova.test()`	Balanced one way ANOVA
`pwr.chisq.test()`	Chi-square test
`pwr.f2.test()`	General linear model
`pwr.p.test()`	Proportion (one sample)
`pwr.r.test()`	Correlation
`pwr.t.test()`	T-tests (one sample, 2 sample, paired)
`pwr.t2n.test()`	T-test (two samples with unequal n)

Exercise 10

You're doing a gene expression experiment. What's your power to detect a 2-fold change in a gene with a standard deviation of 0.7, given 3 samples? (Note - fold change is usually given on the log_2 scale, so a 2-fold change would be a `delta` of 1. That is, if the fold change is 2x, then $log_2(2) = 1$, and you should use 1 in the calculation, not 2).

```
[1] 0.271
```

Exercise 11

How many samples would you need to have 80% power to detect this effect?

```
[1] 8.76
```

Exercise 12

You're doing a population genome-wide association study (GWAS) looking at the effect of a SNP on disease X. Disease X has a baseline prevalence of 5% in the population, but you suspect the SNP might increase the risk of disease X by 10% (this is typical for SNP effects on common, complex diseases). Calculate the number of samples do you need to have 80% power to detect this effect, given that you want a genome-wide statistical significance of $p < 5 \times 10^{-8}$ to account for multiple testing.[1] (Hint, you can expressed 5×10^{-8} in R using `5e-8` instead of `.00000005`).

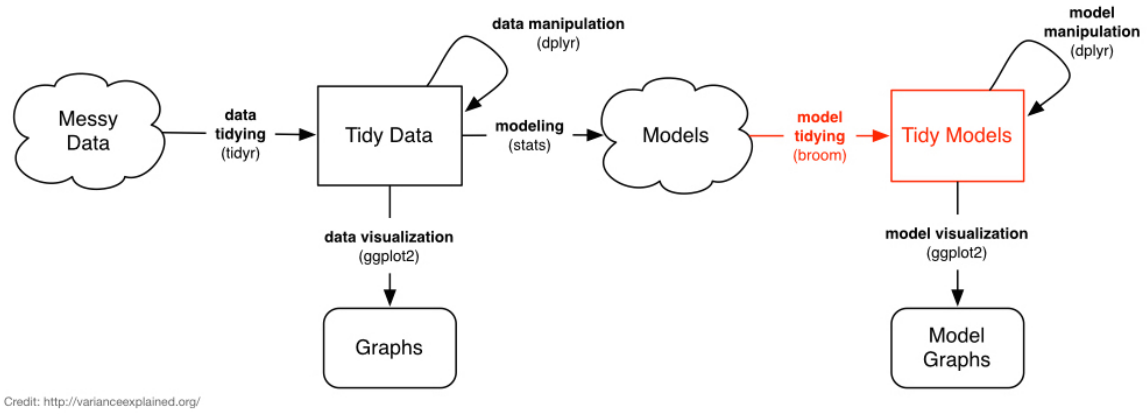
```
[1] 157589
```

[1] https://www.quora.com/Why-is-P-value-5x10-8-chosen-as-a-threshold-to-reach-genome-wide-significance

8.6 Tidying models

We spent a lot of time in previous chapters on *tidy* ***data***, where each column is a variable and each row is an observation. Tidy data is easy to filter observations based on values in a column (e.g., we could get just adult males with `filter(nha, Gender=="male" & Age>=18)`, and easy to select particular variables/features of interest by their column name.

Even when we start with tidy *data*, we don't end up with tidy *models*. The output from tests like `t.test` or `lm` are not data.frames, and it's difficult to get the information out of the model object that we want. The **broom** package bridges this gap.



Depending on the type of model object you're using, broom provides three methods that do different kinds of tidying:

1. `tidy`: constructs a data frame that summarizes the model's statistical findings like coefficients and p-values.
2. `augment`: add columns to the original data that was modeled, like predictions and residuals.
3. `glance`: construct a concise *one-row* summary of the model with information like R^2 that are computed once for the entire model.

Let's go back to our linear model example.

```
# Try modeling Testosterone against Physical Activity, Age, and Gender.
fit <- lm(Testosterone~PhysActive+Age+Gender, data=nha)

# See what that model looks like:
summary(fit)
```

```
Call:
lm(formula = Testosterone ~ PhysActive + Age + Gender, data = nha)

Residuals:
   Min     1Q Median     3Q    Max
-397.9  -31.0   -4.4   20.5 1400.9

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)    46.693      7.573    6.17  7.8e-10
PhysActiveYes   9.275      4.462    2.08    0.038
Age            -0.590      0.128   -4.60  4.3e-06
Gendermale    385.199      4.351   88.53  < 2e-16

Residual standard error: 128 on 3434 degrees of freedom
  (269 observations deleted due to missingness)
Multiple R-squared:  0.697, Adjusted R-squared:  0.697
F-statistic: 2.63e+03 on 3 and 3434 DF,  p-value: <2e-16
```

What if we wanted to pull out the coefficient for Age, or the P-value for PhysActive? It gets pretty gross. We first have to `coef(summary(lmfit))` to get a matrix of coefficients, the terms are still stored in row names, and the column names are inconsistent with other packages (e.g. `Pr(>|t|)` compared to `p.value`). Yuck!

```
coef(summary(fit))["Age", "Estimate"]
```

```
[1] -0.59
```

```
coef(summary(fit))["PhysActiveYes", "Pr(>|t|)"]
```

```
[1] 0.0377
```

Instead, you can use the `tidy` function, from the broom package, on the fit:

```
# Install the package if you don't have it
# install.packages("broom")

library(broom)
tidy(fit)
```

```
# A tibble: 4 x 5
  term          estimate std.error statistic  p.value
  <chr>            <dbl>     <dbl>     <dbl>    <dbl>
1 (Intercept)     46.7       7.57       6.17 7.83e-10
2 PhysActiveYes    9.27      4.46       2.08 3.77e- 2
3 Age             -0.590     0.128     -4.60 4.28e- 6
4 Gendermale     385.        4.35      88.5  0
```

This gives you a data.frame with all your model results. The row names have been moved into a column called `term`, and the column names are simple and consistent (and can be accessed using `$`). These can be manipulated with dplyr just like any other data frame.

```
tidy(fit) |>
  filter(term!="(Intercept)") |>
  select(term, p.value) |>
  arrange(p.value)
```

```
# A tibble: 3 x 2
  term             p.value
  <chr>              <dbl>
1 Gendermale    0
2 Age           0.00000428
3 PhysActiveYes 0.0377
```

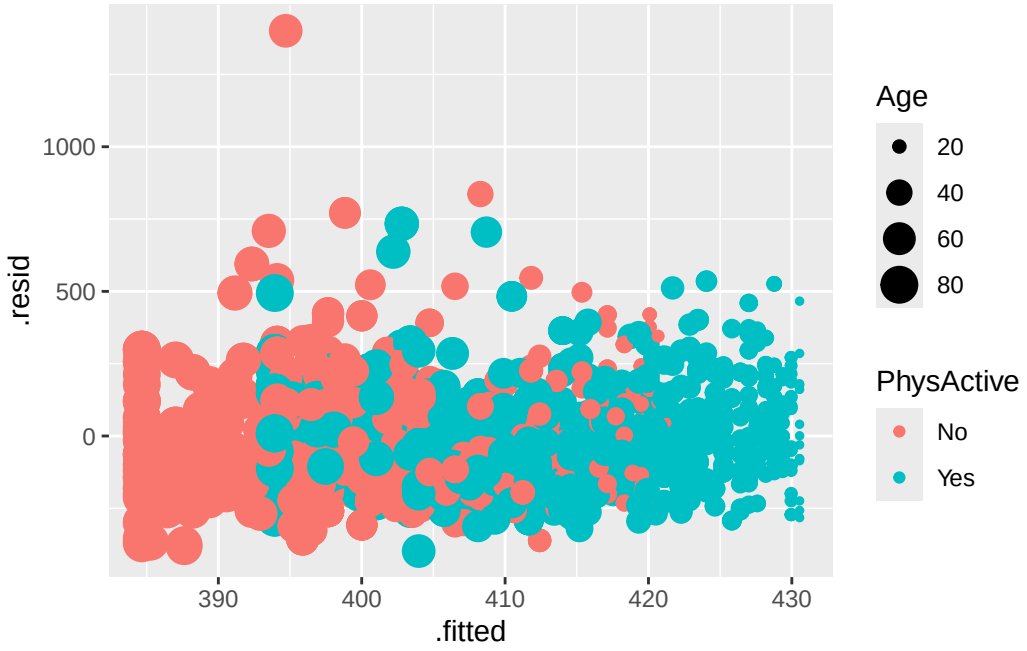
Instead of viewing the coefficients, you might be interested in the fitted values and residuals for each of the original points in the regression. For this, use `augment`, which augments the original data with information from the model. New columns begins with a `.` (to avoid overwriting any of the original columns).

```
# Augment the original data
# IF you get a warning about deprecated... purrr..., ignore. It's a bug that'll be fixed s
augment(fit) |> head()
```

```
# A tibble: 6 x 11
  .rownames Testosterone PhysActive   Age Gender .fitted .resid     .hat .sigma
  <chr>            <dbl> <fct>      <dbl> <fct>    <dbl>  <dbl>    <dbl>  <dbl>
1 1                 47.5 No            43 female    21.3  26.2  0.000989   128.
2 2                643.  No            80 male     385.  258.   0.00185    127.
3 3                643.  No            80 male     385.  258.   0.00185    127.
4 4                 21.1 Yes           34 female    35.9 -14.8  0.000928   128.
```

```
5 5                563.  No            80 male     385.  178.    0.00185    128.
6 6                402.  No            35 male     411.   -9.45 0.00117    128.
# i 2 more variables: .cooksd <dbl>, .std.resid <dbl>
```

```
# Plot residuals vs fitted values for males,
# colored by Physical Activity, size scaled by age
augment(fit) |>
  filter(Gender=="male") |>
  ggplot(aes(.fitted, .resid, col=PhysActive, size=Age)) + geom_point()
```



Finally, several summary statistics are computed for the entire regression, such as R^2 and the F-statistic. These can be accessed with `glance`:

```
glance(fit)
```

```
# A tibble: 1 x 12
  r.squared adj.r.squared sigma statistic p.value    df  logLik    AIC    BIC
      <dbl>         <dbl> <dbl>     <dbl>   <dbl> <dbl>   <dbl>  <dbl>  <dbl>
1     0.697         0.697  128.     2632.       0     3 -21545. 43100. 43130.
# i 3 more variables: deviance <dbl>, df.residual <int>, nobs <int>
```

The **broom** functions work on a pipe, so you can `|>` your model directly to any of the functions like `tidy()`. Let's tidy up our t-test:

```
t.test(AlcoholYear~RelationshipStatus, data=nha)
```

```
    Welch Two Sample t-test

data:  AlcoholYear by RelationshipStatus
t = 5.4315, df = 2674.8, p-value = 6.09e-08
alternative hypothesis: true difference in means between group Committed and group Single is
95 percent confidence interval:
 13.05949 27.81603
sample estimates:
mean in group Committed    mean in group Single
               83.93416                63.49640
```

```
t.test(AlcoholYear~RelationshipStatus, data=nha) |> tidy()
```

```
# A tibble: 1 x 10
  estimate estimate1 estimate2 statistic    p.value parameter conf.low conf.high
     <dbl>     <dbl>     <dbl>     <dbl>      <dbl>     <dbl>    <dbl>     <dbl>
1     20.4      83.9      63.5      5.43    6.09e-8     2675.     13.1      27.8
# i 2 more variables: method <chr>, alternative <chr>
```

…and our Mann-Whitney U test / Wilcoxon rank-sum test:

```
wilcox.test(AlcoholYear~RelationshipStatus, data=nha)
```

```
    Wilcoxon rank sum test with continuity correction

data:  AlcoholYear by RelationshipStatus
W = 1067954, p-value = 0.0001659
alternative hypothesis: true location shift is not equal to 0
```

```
wilcox.test(AlcoholYear~RelationshipStatus, data=nha) |> tidy()
```

```
# A tibble: 1 x 4
  statistic  p.value method                                                alternative
      <dbl>    <dbl> <chr>                                                 <chr>
1  1067954. 0.000166 Wilcoxon rank sum test with continuity correct~ two.sided
```

...and our Fisher's exact test on the cross-tabulated data:

```
xtabs(~Gender+Diabetes, data=nha) |> fisher.test()
```

```
    Fisher's Exact Test for Count Data

data:  xtabs(~Gender + Diabetes, data = nha)
p-value = 0.05992
alternative hypothesis: true odds ratio is not equal to 1
95 percent confidence interval:
 0.9883143 1.5466373
sample estimates:
odds ratio
  1.235728
```

```
xtabs(~Gender+Diabetes, data=nha) |> fisher.test() |> tidy()
```

```
# A tibble: 1 x 6
  estimate p.value conf.low conf.high method                          alternative
     <dbl>   <dbl>    <dbl>     <dbl> <chr>                           <chr>
1     1.24  0.0599    0.988      1.55 Fisher's Exact Test for Count~ two.sided
```

...and finally, a logistic regression model:

```
# fit the model and summarize it the usual way
glmfit <- glm(Insured~Race, data=nha, family=binomial)
summary(glmfit)
```

```
Call:
glm(formula = Insured ~ Race, family = binomial, data = nha)

Coefficients:
```

```
             Estimate Std. Error z value Pr(>|z|)
(Intercept)   1.94218    0.06103  31.825  < 2e-16
RaceAsian    -0.64092    0.17715  -3.618 0.000297
RaceBlack    -0.59744    0.13558  -4.406 1.05e-05
RaceHispanic -1.41354    0.14691  -9.622  < 2e-16
RaceMexican  -1.98385    0.13274 -14.946  < 2e-16
RaceOther    -1.26430    0.22229  -5.688 1.29e-08

(Dispersion parameter for binomial family taken to be 1)

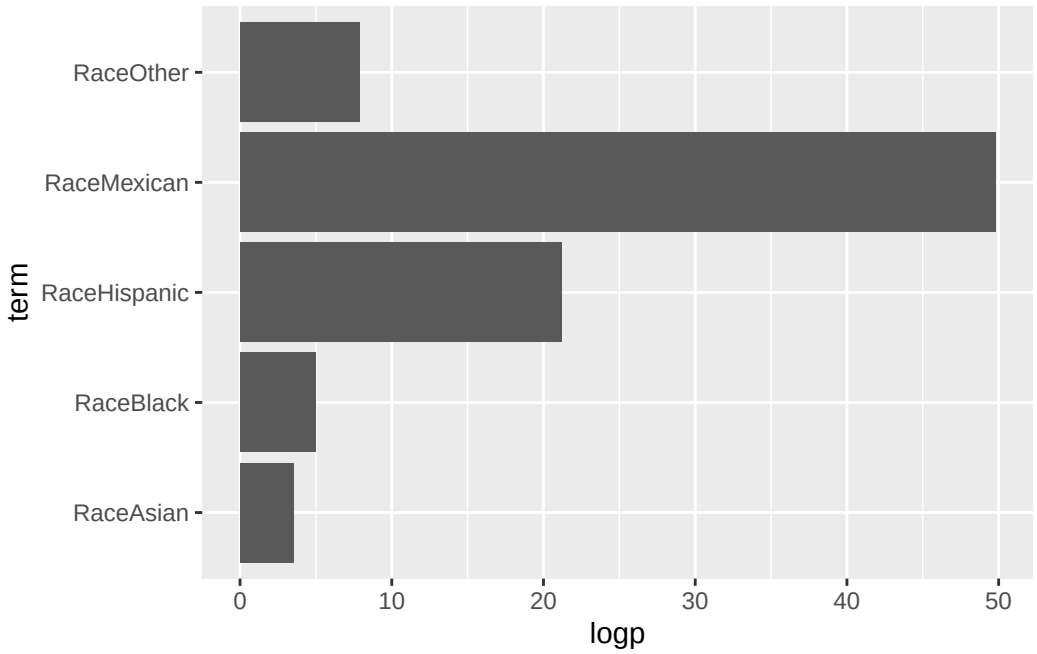
    Null deviance: 3614.6  on 3704  degrees of freedom
Residual deviance: 3336.6  on 3699  degrees of freedom
  (2 observations deleted due to missingness)
AIC: 3348.6

Number of Fisher Scoring iterations: 4
```

```
# tidy it up!
tidy(glmfit)
```

```
# A tibble: 6 x 5
  term         estimate std.error statistic   p.value
  <chr>           <dbl>     <dbl>     <dbl>     <dbl>
1 (Intercept)     1.94     0.0610     31.8  2.96e-222
2 RaceAsian      -0.641    0.177      -3.62 2.97e-  4
3 RaceBlack      -0.597    0.136      -4.41 1.05e-  5
4 RaceHispanic   -1.41     0.147      -9.62 6.47e- 22
5 RaceMexican    -1.98     0.133     -14.9  1.66e- 50
6 RaceOther      -1.26     0.222      -5.69 1.29e-  8
```

```
# do whatever you want now
tidy(glmfit) |>
  filter(term!="(Intercept)") |>
  mutate(logp=-1*log10(p.value)) |>
  ggplot(aes(term, logp)) + geom_bar(stat="identity") + coord_flip()
```



Check out some of the other **broom** vignettes on CRAN, and also check out the **biobroom** package on bioconductor for turning bioconductor objects and analytical results into tidy data frames.

8.7 Additional topics & recommended reading

8.7.1 1. Batch effects

Batch effects are sources of technical variation introduced during an experiment, such as processing with different reagents, handling by a different technician, sequencing on a different flow cell, or processing samples in groups on different days. If these *batch effects* are strongly confounded with the study variable of interest, they can call into question the validity of your results, and in some cases, render collected data completely useless. The papers below discuss batch effects and how they can be mitigated.

1. **Chapter 5** of Scherer, Andreas. *Batch effects and noise in microarray experiments: sources and solutions.* Vol. 868. John Wiley & Sons, 2009.
    - Chapter 5 only: http://onlinelibrary.wiley.com/doi/10.1002/9780470685983.ch5/pdf.
    - Entire book: https://faculty.mu.edu.sa/public/uploads/1382673974.784197804707 41382.pdf.

2. Leek, Jeffrey T., et al. "Tackling the widespread and critical impact of batch effects in high-throughput data." *Nature Reviews Genetics* 11.10 (2010): 733-739. Available at https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3880143/.

8.7.2 2. What's my *n*?

"What's my n" isn't always a straightforward question to answer, especially when it comes to cell culture expriments. The post and article below go into some of these details.

1. Statistics for Experimental Biologists: "What is 'n' in cell culture experiments?" Available at http://labstats.net/articles/cell_culture_n.html.
2. Vaux, David L., Fiona Fidler, and Geoff Cumming. "Replicates and repeats—what is the difference and is it significant?." *EMBO reports* 13.4 (2012): 291-296. Available at https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3321166/.

8.7.3 3. Technical versus biological replicates

Technical replicates involve taking multiple measurements on the same sample. Biological replicates are different samples each with separate measurements/assays. While technical replicates can help calibrate the precision of an instrument or assay, biological replicates are necessary for statistical analysis to make inferences about a condition or treatment. Read the paper and note below for more information on technical vs biological replication.

1. Blainey, Paul, Martin Krzywinski, and Naomi Altman. "Points of significance: replication." *Nature methods* 11.9 (2014): 879-880. Available at http://rdcu.be/yguA.
2. Illumina Technical Note: "The Power of Replicates." Available at https://www.illumina.com/Documents/products/technotes/technote_power_replicates.pdf.

9 Survival Analysis

This chapter will provide hands-on instruction and exercises covering survival analysis using R. Some of the data to be used here will come from The Cancer Genome Atlas (TCGA), where we may also cover programmatic access to TCGA through Bioconductor if time allows.

Handouts: Download and print out these handouts and bring them to class:

- Cheat sheet
- Background handout
- Exercises handout

9.1 Background

In the chapter on essential statistics (Chapter 8) we covered basic categorical data analysis – comparing proportions (risks, rates, etc) between different groups using a chi-square or fisher exact test, or logistic regression. For example, we looked at how the diabetes rate differed between males and females. In this kind of analysis you implicitly assume that the rates are constant over the period of the study, or as defined by the different groups you defined.

But, in longitudinal studies where you track samples or subjects from one time point (e.g., entry into a study, diagnosis, start of a treatment) until you observe some outcome *event* (e.g., death, onset of disease, relapse), it doesn't make sense to assume the rates are constant. For example: the risk of death after heart surgery is highest immediately post-op, decreases as the patient recovers, then rises slowly again as the patient ages. Or, recurrence rate of different cancers varies highly over time, and depends on tumor genetics, treatment, and other environmental factors.

9.1.1 Definitions

Survival analysis lets you analyze the rates of occurrence of events over time, without assuming the rates are constant. Generally, survival analysis lets you model the *time until an*

event occurs,[1] or compare the time-to-event between different groups, or how time-to-event correlates with quantitative variables.

The **hazard** is the instantaneous event (death) rate at a particular time point t. Survival analysis doesn't assume the hazard is constant over time. The *cumulative hazard* is the total hazard experienced up to time t.

The **survival function**, is the probability an individual survives (or, the probability that the event of interest does not occur) up to and including time t. It's the probability that the event (e.g., death) hasn't occured yet. It looks like this, where T is the time of death, and $Pr(T > t)$ is the probability that the time of death is greater than some time t. S is a probability, so $0 \leq S(t) \leq 1$, since survival times are always positive ($T \geq 0$).

$$S(t) = Pr(T > t)$$

The **Kaplan-Meier** curve illustrates the survival function. It's a step function illustrating the cumulative survival probability over time. The curve is horizontal over periods where no event occurs, then drops vertically corresponding to a change in the survival function at each time an event occurs.

Censoring is a type of missing data problem unique to survival analysis. This happens when you track the sample/subject through the end of the study and the event never occurs. This could also happen due to the sample/subject dropping out of the study for reasons other than death, or some other loss to followup. The sample is *censored* in that you only know that the individual survived up to the loss to followup, but you don't know anything about survival after that.[2]

Proportional hazards assumption: The main goal of survival analysis is to compare the survival functions in different groups, e.g., leukemia patients as compared to cancer-free controls. If you followed both groups until everyone died, both survival curves would end at 0%, but one group might have survived on average a lot longer than the other group. Survival analysis does this by comparing the *hazard* at different times over the observation period. Survival analysis doesn't assume that the hazard is constant, but *does* assume that the *ratio* of hazards between groups is constant over time.[3] This class does *not* cover methods to deal with non-proportional hazards, or interactions of covariates with the time to event.

[1] In the medical world, we typically think of *survival analysis* literally – tracking time until death. But, it's more general than that – survival analysis models time until an *event* occurs (*any* event). This might be death of a biological organism. But it could also be the time until a hardware failure in a mechanical system, time until recovery, time someone remains unemployed after losing a job, time until a ripe tomato is eaten by a grazing deer, time until someone falls asleep in a workshop, etc. *Survival analysis* also goes by *reliability theory* in engineering, *duration analysis* in economics, and *event history analysis* in sociology.

[2] This describes the most common type of censoring – *right censoring*. *Left censoring* less commonly occurs when the "start" is unknown, such as when an initial diagnosis or exposure time is unknown.

[3] And, following the definitions above, assumes that the *cumulative hazard* ratio between two groups remains constant over time.

Proportional hazards regression a.k.a. **Cox regression** is the most common approach to assess the effect of different variables on survival.

9.1.2 Cox PH Model

Kaplan-Meier curves are good for visualizing differences in survival between two categorical groups,[4] but they don't work well for assessing the effect of *quantitative* variables like age, gene expression, leukocyte count, etc. Cox PH regression can assess the effect of both categorical and continuous variables, and can model the effect of multiple variables at once.[5]

Cox PH regression models the natural log of the hazard at time t, denoted $h(t)$, as a function of the baseline hazard ($h_0(t)$) (the hazard for an individual where all exposure variables are 0) and multiple exposure variables x_1, x_1, ..., x_p. The form of the Cox PH model is:

$$log(h(t)) = log(h_0(t)) + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_p x_p$$

If you exponentiate both sides of the equation, and limit the right hand side to just a single categorical exposure variable (x_1) with two groups ($x_1 = 1$ for exposed and $x_1 = 0$ for unexposed), the equation becomes:

$$h_1(t) = h_0(t) \times e^{\beta_1 x_1}$$

Rearranging that equation lets you estimate the **hazard ratio**, comparing the exposed to the unexposed individuals at time t:

$$HR(t) = \frac{h_1(t)}{h_0(t)} = e^{\beta_1}$$

This model shows that **the hazard ratio is e^{β_1},** and remains constant over time t (hence the name *proportional hazards regression*). The β values are the regression coefficients that are estimated from the model, and represent the $log(Hazard\,Ratio)$ for each unit increase in the corresponding predictor variable. The interpretation of the hazards ratio depends on the measurement scale of the predictor variable, but in simple terms, a positive coefficient indicates worse survival and a negative coefficient indicates better survival for the variable in question.

[4] And there's a chi-square-like statistical test for these differences called the log-rank test that compare the survival functions categorical groups.

[5] See the multiple regression section of the essential statistics section (Chapter 8).

9.2 Survival analysis in R

The core survival analysis functions are in the **survival** package. The survival package is one of the few "core" packages that comes bundled with your basic R installation, so you probably didn't need to `install.packages()` it. But, you'll need to load it like any other library when you want to use it. We'll also be using the **dplyr** package, so let's load that too. Finally, we'll also want to load the **survminer** package, which provides much nicer Kaplan-Meier plots out-of-the-box than what you get out of base graphics.

```
library(dplyr)
library(survival)
library(survminer)
```

The core functions we'll use out of the survival package include:

- `Surv()`: Creates a survival object.
- `survfit()`: Fits a survival curve using either a formula, of from a previously fitted Cox model.
- `coxph()`: Fits a Cox proportional hazards regression model.

Other optional functions you might use include:

- `cox.zph()`: Tests the proportional hazards assumption of a Cox regression model.
- `survdiff()`: Tests for differences in survival between two groups using a log-rank / Mantel-Haenszel test.[6]

`Surv()` creates the response variable, and typical usage takes the time to event,[7] and whether or not the event occured (i.e., death vs censored). `survfit()` creates a survival curve that you could then display or plot. `coxph()` implements the regression analysis, and models specified the same way as in regular linear models, but using the `coxph()` function.

9.2.1 Getting started

We're going to be using the built-in lung cancer dataset[8] that ships with the survival package. You can get some more information about the dataset by running `?lung`. The help tells us there are 10 variables in this data:

[6] Cox regression and the logrank test from `survdiff` are going to give you similar results most of the time. The log-rank test is asking if survival curves differ significantly between two groups. Cox regression is asking which of many categorical or continuous variables significantly affect survival.

[7] `Surv()` can also take start and stop times, to account for left censoring. See the help for `?Surv`.

[8] Loprinzi et al. Prospective evaluation of prognostic variables from patient-completed questionnaires. North Central Cancer Treatment Group. *Journal of Clinical Oncology*. 12(3):601-7, 1994.

```
library(survival)
?lung
```

1. `inst`: Institution code
2. **`time`: Survival time in days**
3. **`status`: censoring status 1=censored, 2=dead**
4. `age`: Age in years
5. **`sex`: Male=1 Female=2**
6. `ph.ecog`: ECOG performance score (0=good 5=dead)
7. `ph.karno`: Karnofsky performance score as rated by physician
8. `pat.karno`: Karnofsky performance score as rated by patient
9. `meal.cal`: Calories consumed at meals
10. `wt.loss`: Weight loss in last six months

You can access the data just by running `lung`, as if you had read in a dataset and called it `lung`. You can operate on it just like any other data frame.

```
head(lung)
class(lung)
dim(lung)
View(lung)
```

Notice that lung is a plain `data.frame` object. You could see what it looks like as a tibble (prints nicely, tells you the type of variable each column is). You could then reassign `lung` to the `as_tibble()`-ified version.

```
as_tibble(lung)
lung <- as_tibble(lung)
lung
```

9.2.2 Survival Curves

Check out the help for `?Surv`. This is the main function we'll use to create the survival object. You can play fast and loose with how you specify the arguments to `Surv`. The help tells you that when there are two unnamed arguments, they will match `time` and `event` in that order. This is the common shorthand you'll often see for right-censored data. The alternative lets you specify interval data, where you give it the start and end times (`time` and `time2`). If you keep reading you'll see how `Surv` tries to guess how you're coding the status variable. It will try to guess whether you're using 0/1 or 1/2 to represent censored vs "dead", respectively.[9]

[9]Where "dead" really refers to the occurance of the event (any event), not necessarily death.

Try creating a survival object called `s`, then display it. If you go back and `head(lung)` the data, you can see how these are related. It's a special type of vector that tells you both how long the subject was tracked for, and whether or not the event occured or the sample was censored (shown by the +).

```
s <- Surv(lung$time, lung$status)
class(s)
```

```
[1] "Surv"
```

```
s
```

```
  [1]  306   455  1010+  210   883  1022+  310   361   218   166   170   654
 [13]  728    71   567   144   613   707    61    88   301    81   624   371
 [25]  394   520   574   118   390    12   473    26   533   107    53   122
 [37]  814   965+   93   731   460   153   433   145   583    95   303   519
 [49]  643   765   735   189    53   246   689    65     5   132   687   345
 [61]  444   223   175    60   163    65   208   821+  428   230   840+  305
 [73]   11   132   226   426   705   363    11   176   791    95   196+  167
 [85]  806+  284   641   147   740+  163   655   239    88   245   588+   30
 [97]  179   310   477   166   559+  450   364   107   177   156   529+   11
[109]  429   351    15   181   283   201   524    13   212   524   288   363
[121]  442   199   550    54   558   207    92    60   551+  543+  293   202
[133]  353   511+  267   511+  371   387   457   337   201   404+  222    62
[145]  458+  356+  353   163    31   340   229   444+  315+  182   156   329
[157]  364+  291   179   376+  384+  268   292+  142   413+  266+  194   320
[169]  181   285   301+  348   197   382+  303+  296+  180   186   145   269+
[181]  300+  284+  350   272+  292+  332+  285   259+  110   286   270    81
[193]  131   225+  269   225+  243+  279+  276+  135
 [ reached getOption("max.print") -- omitted 28 entries ]
```

```
head(lung)
```

```
# A tibble: 6 x 10
   inst  time status   age   sex ph.ecog ph.karno pat.karno meal.cal wt.loss
  <dbl> <dbl>  <dbl> <dbl> <dbl>   <dbl>    <dbl>     <dbl>    <dbl>   <dbl>
1     3   306      2    74     1       1       90       100     1175      NA
2     3   455      2    68     1       0       90        90     1225      15
3     3  1010      1    56     1       0       90        90       NA      15
```

```
4     5   210       2    57      1        1        90         60      1150       11
5     1   883       2    60      1        0       100         90        NA        0
6    12  1022       1    74      1        1        50         80       513        0
```

Now, let's **fit a survival curve** with the `survfit()` function. See the help for `?survfit`. Here we'll create a simple survival curve that doesn't consider any different groupings, so we'll specify just an intercept (e.g., `~1`) in the formula that `survfit` expects. We can do what we just did by "modeling" the survival object `s` we just created against an intercept only, but from here out, we'll just do this in one step by nesting the `Surv()` call within the `survfit()` call, and similar to how we specify data for linear models with `lm()`, we'll use the `data=` argument to specify which data we're using. Similarly, we can assign that to another object called `sfit` (or whatever we wanted to call it).

```
survfit(s~1)
```

```
Call: survfit(formula = s ~ 1)

      n events median 0.95LCL 0.95UCL
[1,] 228    165    310     285     363
```

```
survfit(Surv(time, status)~1, data=lung)
```

```
Call: survfit(formula = Surv(time, status) ~ 1, data = lung)

      n events median 0.95LCL 0.95UCL
[1,] 228    165    310     285     363
```

```
sfit <- survfit(Surv(time, status)~1, data=lung)
sfit
```

```
Call: survfit(formula = Surv(time, status) ~ 1, data = lung)

      n events median 0.95LCL 0.95UCL
[1,] 228    165    310     285     363
```

Now, that object itself isn't very interesting. It's more interesting to run `summary` on what it creates. This will show a life table.

```
summary(sfit)
```

```
Call: survfit(formula = Surv(time, status) ~ 1, data = lung)

 time n.risk n.event survival std.err lower 95% CI upper 95% CI
    5    228       1   0.9956 0.00438       0.9871        1.000
   11    227       3   0.9825 0.00869       0.9656        1.000
   12    224       1   0.9781 0.00970       0.9592        0.997
   13    223       2   0.9693 0.01142       0.9472        0.992
   15    221       1   0.9649 0.01219       0.9413        0.989
   26    220       1   0.9605 0.01290       0.9356        0.986
   30    219       1   0.9561 0.01356       0.9299        0.983
   31    218       1   0.9518 0.01419       0.9243        0.980
   53    217       2   0.9430 0.01536       0.9134        0.974
   54    215       1   0.9386 0.01590       0.9079        0.970
   59    214       1   0.9342 0.01642       0.9026        0.967
   60    213       2   0.9254 0.01740       0.8920        0.960
   61    211       1   0.9211 0.01786       0.8867        0.957
   62    210       1   0.9167 0.01830       0.8815        0.953
   65    209       2   0.9079 0.01915       0.8711        0.946
   71    207       1   0.9035 0.01955       0.8660        0.943
   79    206       1   0.8991 0.01995       0.8609        0.939
   81    205       2   0.8904 0.02069       0.8507        0.932
   88    203       2   0.8816 0.02140       0.8406        0.925
   92    201       1   0.8772 0.02174       0.8356        0.921
   93    199       1   0.8728 0.02207       0.8306        0.917
   95    198       2   0.8640 0.02271       0.8206        0.910
  105    196       1   0.8596 0.02302       0.8156        0.906
  107    194       2   0.8507 0.02362       0.8056        0.898
  110    192       1   0.8463 0.02391       0.8007        0.894
  116    191       1   0.8418 0.02419       0.7957        0.891
  118    190       1   0.8374 0.02446       0.7908        0.887
  122    189       1   0.8330 0.02473       0.7859        0.883
 [ reached getOption("max.print") -- omitted 111 rows ]
```

These tables show a row for each time point where either the event occured or a sample was censored. It shows the number at risk (number still remaining), and the cumulative survival at that instant.

What's more interesting though is if we model something besides just an intercept. Let's fit survival curves separately by sex.

```r
sfit <- survfit(Surv(time, status)~sex, data=lung)
sfit
```

```
Call: survfit(formula = Surv(time, status) ~ sex, data = lung)

       n events median 0.95LCL 0.95UCL
sex=1 138    112    270     212     310
sex=2  90     53    426     348     550
```

```r
summary(sfit)
```

```
Call: survfit(formula = Surv(time, status) ~ sex, data = lung)

                sex=1
 time n.risk n.event survival std.err lower 95% CI upper 95% CI
   11    138       3   0.9783  0.0124       0.9542        1.000
   12    135       1   0.9710  0.0143       0.9434        0.999
   13    134       2   0.9565  0.0174       0.9231        0.991
   15    132       1   0.9493  0.0187       0.9134        0.987
   26    131       1   0.9420  0.0199       0.9038        0.982
   30    130       1   0.9348  0.0210       0.8945        0.977
   31    129       1   0.9275  0.0221       0.8853        0.972
   53    128       2   0.9130  0.0240       0.8672        0.961
   54    126       1   0.9058  0.0249       0.8583        0.956
   59    125       1   0.8986  0.0257       0.8496        0.950
   60    124       1   0.8913  0.0265       0.8409        0.945
   65    123       2   0.8768  0.0280       0.8237        0.933
   71    121       1   0.8696  0.0287       0.8152        0.928
   81    120       1   0.8623  0.0293       0.8067        0.922
   88    119       2   0.8478  0.0306       0.7900        0.910
   92    117       1   0.8406  0.0312       0.7817        0.904
   93    116       1   0.8333  0.0317       0.7734        0.898
   95    115       1   0.8261  0.0323       0.7652        0.892
  105    114       1   0.8188  0.0328       0.7570        0.886
  107    113       1   0.8116  0.0333       0.7489        0.880
  110    112       1   0.8043  0.0338       0.7408        0.873
  116    111       1   0.7971  0.0342       0.7328        0.867
  118    110       1   0.7899  0.0347       0.7247        0.861
  131    109       1   0.7826  0.0351       0.7167        0.855
  132    108       2   0.7681  0.0359       0.7008        0.842
```

```
 135    106       1   0.7609  0.0363       0.6929        0.835
 142    105       1   0.7536  0.0367       0.6851        0.829
 144    104       1   0.7464  0.0370       0.6772        0.823
[ reached getOption("max.print") -- omitted 71 rows ]

              sex=2
time n.risk n.event survival std.err lower 95% CI upper 95% CI
   5     90       1   0.9889  0.0110       0.9675        1.000
  60     89       1   0.9778  0.0155       0.9478        1.000
  61     88       1   0.9667  0.0189       0.9303        1.000
  62     87       1   0.9556  0.0217       0.9139        0.999
  79     86       1   0.9444  0.0241       0.8983        0.993
  81     85       1   0.9333  0.0263       0.8832        0.986
  95     83       1   0.9221  0.0283       0.8683        0.979
 107     81       1   0.9107  0.0301       0.8535        0.972
 122     80       1   0.8993  0.0318       0.8390        0.964
 145     79       2   0.8766  0.0349       0.8108        0.948
 153     77       1   0.8652  0.0362       0.7970        0.939
 166     76       1   0.8538  0.0375       0.7834        0.931
 167     75       1   0.8424  0.0387       0.7699        0.922
 182     71       1   0.8305  0.0399       0.7559        0.913
 186     70       1   0.8187  0.0411       0.7420        0.903
 194     68       1   0.8066  0.0422       0.7280        0.894
 199     67       1   0.7946  0.0432       0.7142        0.884
 201     66       2   0.7705  0.0452       0.6869        0.864
 208     62       1   0.7581  0.0461       0.6729        0.854
 226     59       1   0.7452  0.0471       0.6584        0.843
 239     57       1   0.7322  0.0480       0.6438        0.833
 245     54       1   0.7186  0.0490       0.6287        0.821
 268     51       1   0.7045  0.0501       0.6129        0.810
 285     47       1   0.6895  0.0512       0.5962        0.798
 293     45       1   0.6742  0.0523       0.5791        0.785
 305     43       1   0.6585  0.0534       0.5618        0.772
 310     42       1   0.6428  0.0544       0.5447        0.759
 340     39       1   0.6264  0.0554       0.5267        0.745
[ reached getOption("max.print") -- omitted 23 rows ]
```

Now, check out the help for `?summary.survfit`. You can give the `summary()` function an option for what times you want to show in the results. Look at the range of followup times in the lung dataset with `range()`. You can create a sequence of numbers going from one number to another number by increments of yet another number with the `seq()` function.

```
# ?summary.survfit
range(lung$time)
```

```
[1]    5 1022
```

```
seq(0, 1100, 100)
```

```
 [1]    0  100  200  300  400  500  600  700  800  900 1000 1100
```

And we can use that sequence vector with a summary call on sfit to get life tables at those intervals separately for both males (1) and females (2). From these tables we can start to see that males tend to have worse survival than females.

```
summary(sfit, times=seq(0, 1000, 100))
```

```
Call: survfit(formula = Surv(time, status) ~ sex, data = lung)

                sex=1
 time n.risk n.event survival std.err lower 95% CI upper 95% CI
    0    138       0   1.0000  0.0000       1.0000        1.000
  100    114      24   0.8261  0.0323       0.7652        0.892
  200     78      30   0.6073  0.0417       0.5309        0.695
  300     49      20   0.4411  0.0439       0.3629        0.536
  400     31      15   0.2977  0.0425       0.2250        0.394
  500     20       7   0.2232  0.0402       0.1569        0.318
  600     13       7   0.1451  0.0353       0.0900        0.234
  700      8       5   0.0893  0.0293       0.0470        0.170
  800      6       2   0.0670  0.0259       0.0314        0.143
  900      2       2   0.0357  0.0216       0.0109        0.117
 1000      2       0   0.0357  0.0216       0.0109        0.117

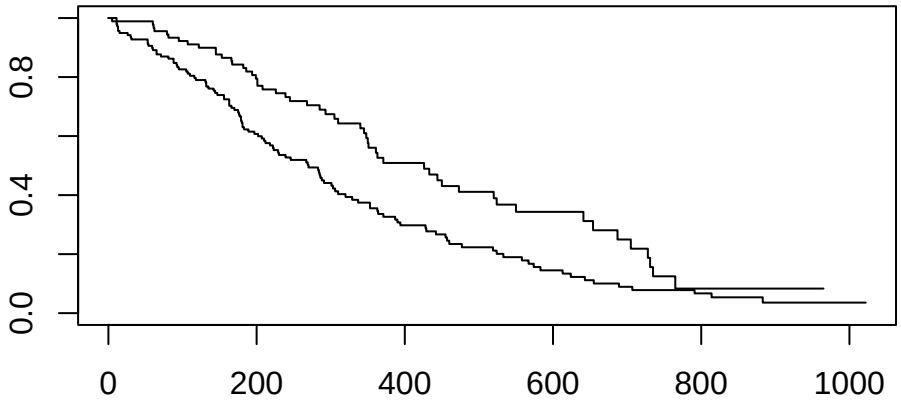
                sex=2
 time n.risk n.event survival std.err lower 95% CI upper 95% CI
    0     90       0   1.0000  0.0000       1.0000        1.000
  100     82       7   0.9221  0.0283       0.8683        0.979
  200     66      11   0.7946  0.0432       0.7142        0.884
  300     43       9   0.6742  0.0523       0.5791        0.785
  400     26      10   0.5089  0.0603       0.4035        0.642
  500     21       5   0.4110  0.0626       0.3050        0.554
```

```
600     11       3    0.3433  0.0634        0.2390         0.493
700      8       3    0.2496  0.0652        0.1496         0.417
800      2       5    0.0832  0.0499        0.0257         0.270
900      1       0    0.0832  0.0499        0.0257         0.270
```

9.2.3 Kaplan-Meier Plots

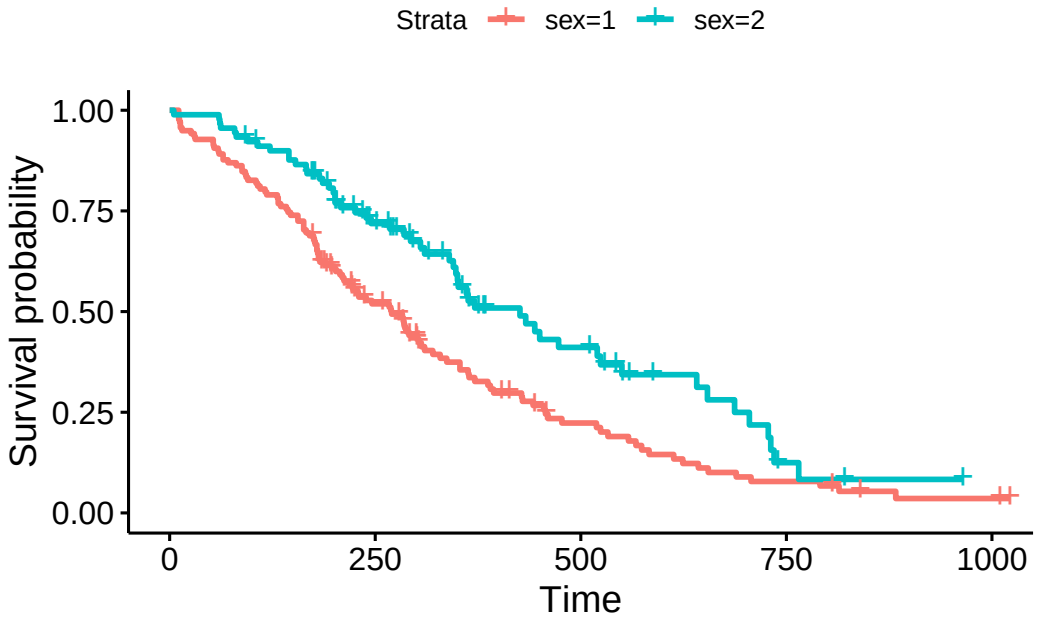
Now that we've fit a survival curve to the data it's pretty easy to visualize it with a **Kaplan-Meier** plot. Create the survival object if you don't have it yet, and instead of using `summary()`, use `plot()` instead.

```
sfit <- survfit(Surv(time, status)~sex, data=lung)
plot(sfit)
```



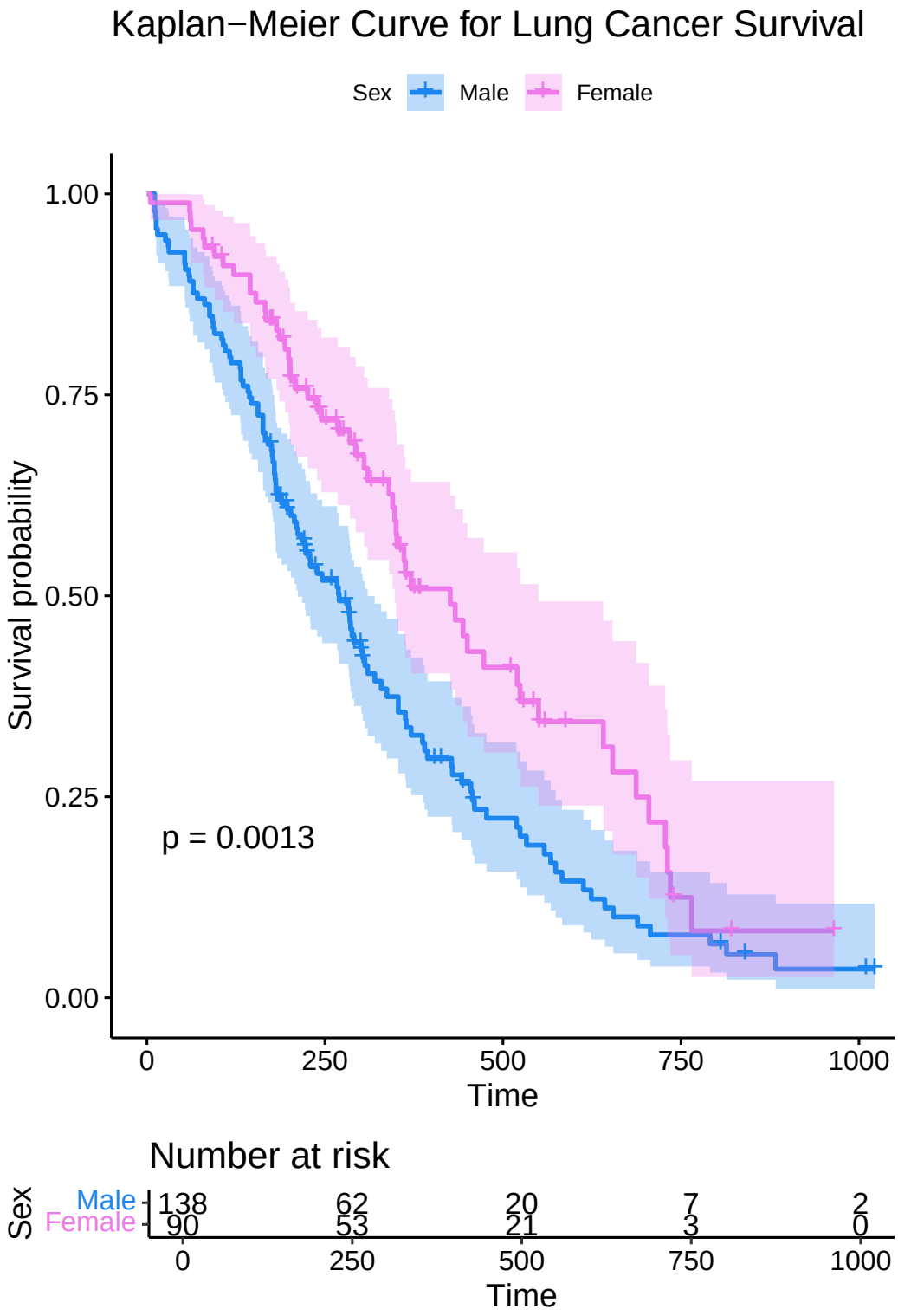
There are lots of ways to modify the plot produced by base R's `plot()` function. You can see more options with the help for `?plot.survfit`. We're not going to go into any more detail here, because there's another package called **survminer** that provides a function called **`ggsurvplot()`** that makes it much easier to produce publication-ready survival plots, and if you're familiar with ggplot2 syntax it's pretty easy to modify. So, let's load the package and try it out.

```
library(survminer)
ggsurvplot(sfit)
```



This plot is substantially more informative by default, just because it automatically color codes the different groups, adds axis labels, and creates and automatic legend. But there's a lot more you can do pretty easily here. Let's add confidence intervals, show the p-value for the log-rank test, show a risk table below the plot, and change the colors and the group labels.

```
ggsurvplot(sfit, conf.int=TRUE, pval=TRUE, risk.table=TRUE,
           legend.labs=c("Male", "Female"), legend.title="Sex",
           palette=c("dodgerblue2", "orchid2"),
           title="Kaplan-Meier Curve for Lung Cancer Survival",
           risk.table.height=.15)
```


Kaplan−Meier Curve for Lung Cancer Survival
Sex
Male
Female
1.00
0.75
0.50
0.25
0.00
Survival probability
p = 0.0013
0
250
500
750
1000
Time
Number at risk
Sex
Male
Female
138
62
20
7
2
90
53
21
3
0
0
250
500
750
1000
Time

Exercise 1

Take a look at the built in `colon` dataset. If you type `?colon` it'll ask you if you wanted help on the colon dataset from the survival package, or the colon operator. Click "Chemotherapy for Stage B/C colon cancer", or be specific with `?survival::colon`. This dataset has survival and recurrence information on 929 people from a clinical trial on colon cancer chemotherapy. There are two rows per person, indidicated by the event type (`etype`) variable – `etype==1` indicates that row corresponds to recurrence; `etype==2` indicates death.
First, let's turn the colon data into a tibble, then filter the data to only include the survival data, not the recurrence data. Let's call this new object `colondeath`. The `filter()` function is in the **dplyr** library, which you can get by running `library(dplyr)`. If you don't have dplyr you can use the base `subset()` function instead.

```
library(dplyr)
colon <- as_tibble(colon)
colondeath <- filter(colon, etype==2)

# Or, using base subset()
# colondeath <- subset(colon, etype==2)

head(colondeath)
```

```
# A tibble: 6 x 16
     id study rx        sex   age obstruct perfor adhere nodes status differ
  <dbl> <dbl> <fct>   <dbl> <dbl>    <dbl>  <dbl>  <dbl> <dbl>  <dbl>  <dbl>
1     1     1 Lev+5FU     1    43        0      0      0     5      1      2
2     2     1 Lev+5FU     1    63        0      0      0     1      0      2
3     3     1 Obs         0    71        0      0      1     7      1      2
4     4     1 Lev+5FU     0    66        1      0      0     6      1      2
5     5     1 Obs         1    69        0      0      0    22      1      2
6     6     1 Lev+5FU     0    57        0      0      0     9      1      2
# i 5 more variables: extent <dbl>, surg <dbl>, node4 <dbl>, time <dbl>,
#   etype <dbl>
```

Exercise 2

Look at the help for `?colon` again. How are `sex` and `status` coded? How is this different from the lung data?

Exercise 3

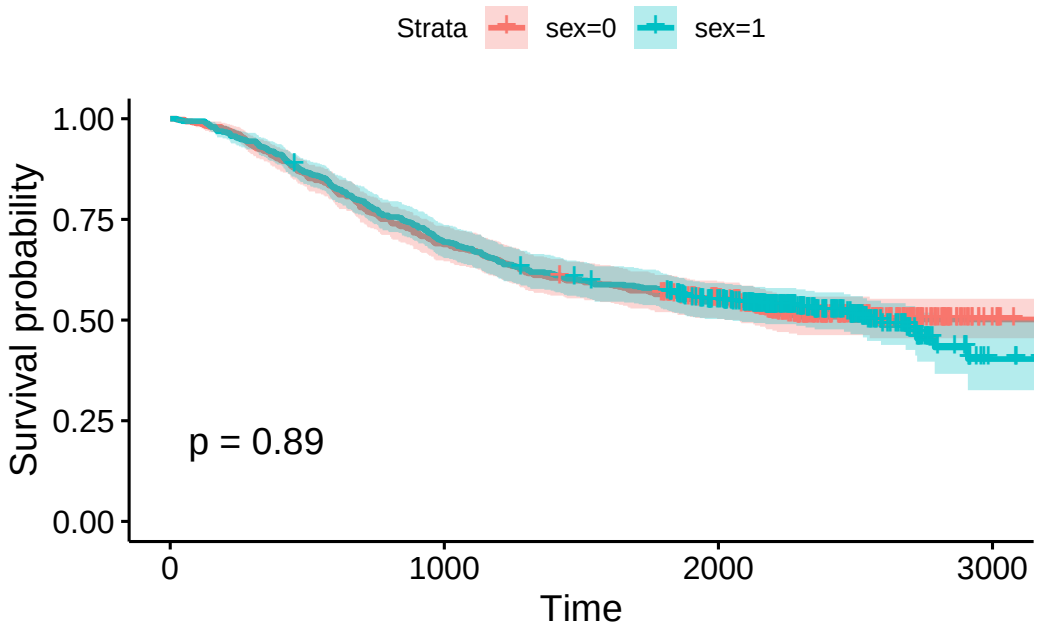
Using `survfit(Surv(..., ...,)~..., data=colondeath)`, create a survival curve separately for males versus females. Call the resulting object `sfit`. Run a `summary()` on this object, showing time points 0, 500, 1000, 1500, and 2000. Do males or females appear to fair better over this time period?

```
                sex=0
 time n.risk n.event survival std.err lower 95% CI upper 95% CI
    0    445       0    1.000  0.0000        1.000        1.000
  500    381      64    0.856  0.0166        0.824        0.889
 1000    306      75    0.688  0.0220        0.646        0.732
 1500    265      40    0.598  0.0232        0.554        0.645
 2000    218      22    0.547  0.0236        0.503        0.596

                sex=1
 time n.risk n.event survival std.err lower 95% CI upper 95% CI
    0    484       0    1.000  0.0000        1.000        1.000
  500    418      65    0.866  0.0155        0.836        0.897
 1000    335      83    0.694  0.0210        0.654        0.736
 1500    287      46    0.598  0.0223        0.556        0.644
 2000    238      25    0.545  0.0227        0.503        0.592
```

Exercise 4

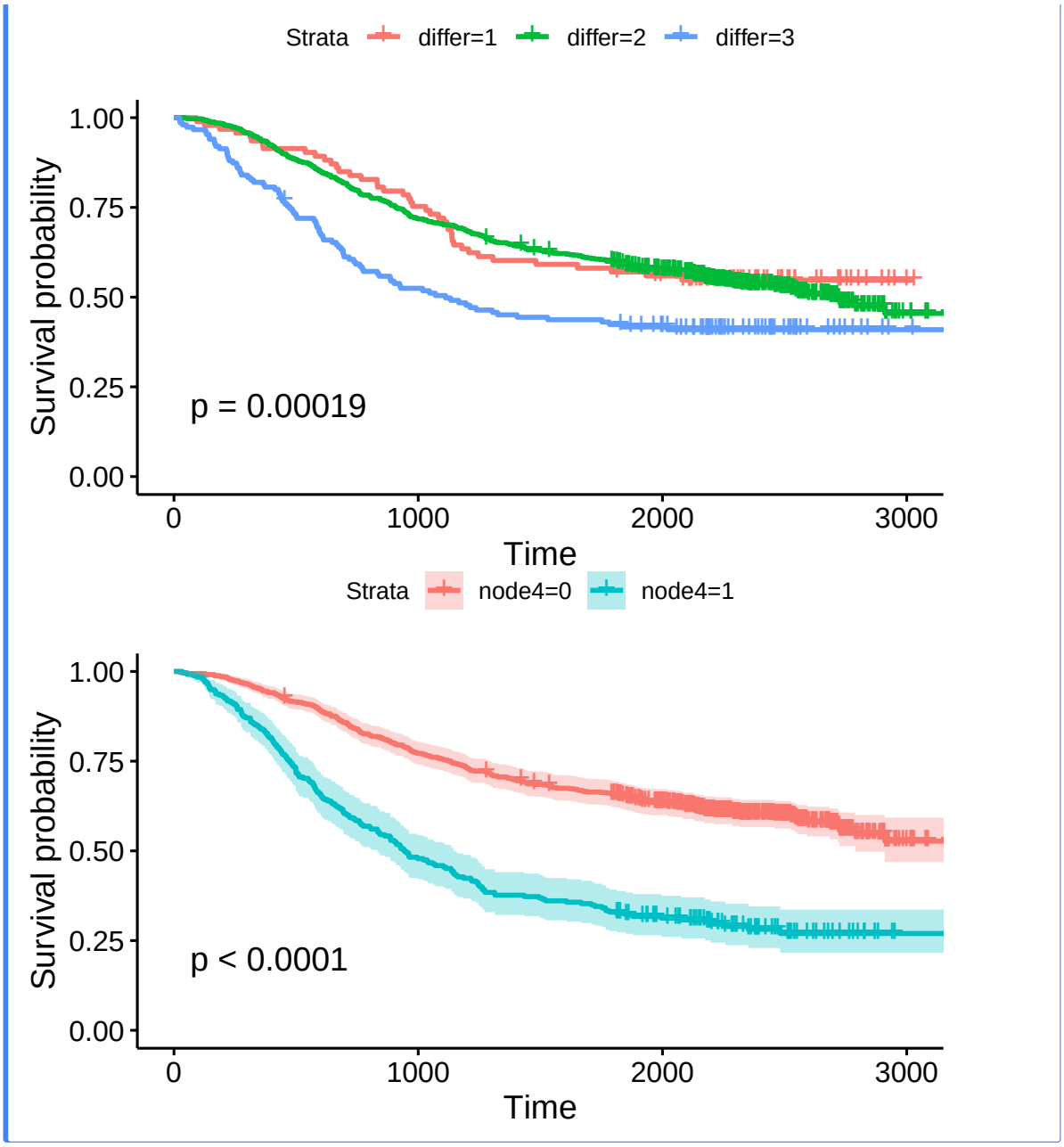
Using the survminer package, plot a Kaplan-Meier curve for this analysis with confidence intervals and showing the p-value. See `?ggsurvplot` for help. Is there a significant difference between males and females?



Exercise 5

Create Kaplan-Meier plot stratifying by:

1. The extent of differentiation (well, moderate, poor), showing the p-value.
2. Whether or not there was detectable cancer in >=4 lymph nodes, showing the p-value and confidence bands.

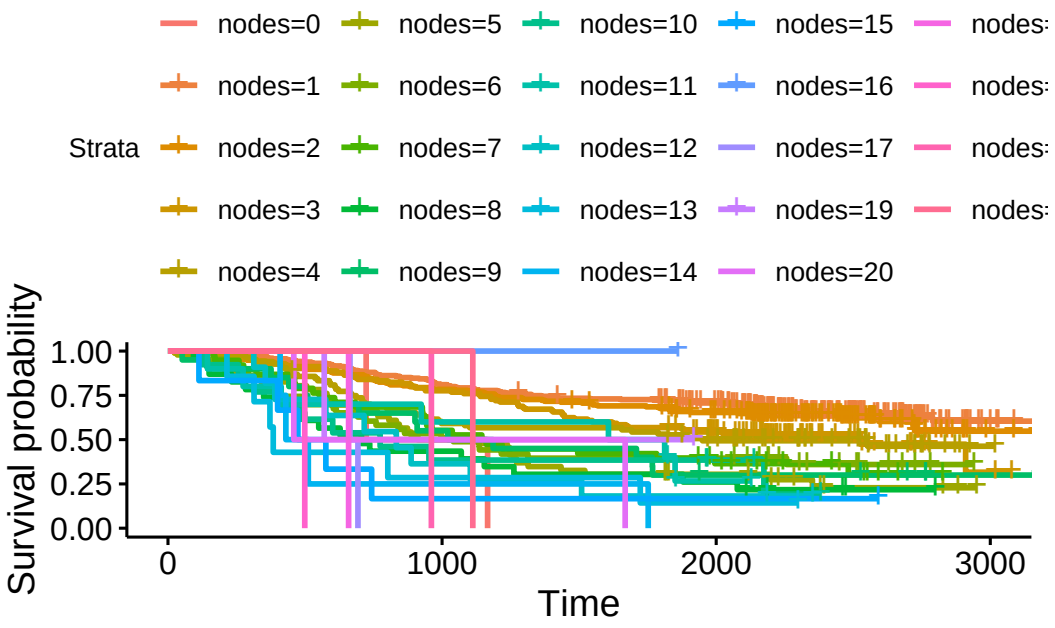


9.2.4 Cox Regression

Kaplan-Meier curves are good for visualizing differences in survival between two categorical groups, and the log-rank test you get when you ask for `pval=TRUE` is useful for asking if there are differences in survival between different groups. But this doesn't generalize well

for assessing the effect of *quantitative* variables. Just try creating a K-M plot for the `nodes` variable, which has values that range from 0-33. What a mess! Don't do this.

```
ggsurvplot(survfit(Surv(time, status)~nodes, data=colondeath))
```



At some point using a categorical grouping for K-M plots breaks down, and further, you might want to assess how *multiple* variables work together to influence survival. For example, you might want to simultaneously examine the effect of race and socioeconomic status, so as to adjust for factors like income, access to care, etc., before concluding that ethnicity influences some outcome.

Cox PH regression can assess the effect of both categorical and continuous variables, and can model the effect of multiple variables at once. The `coxph()` function uses the same syntax as `lm()`, `glm()`, etc. The response variable you create with `Surv()` goes on the left hand side of the formula, specified with a `~`. Explanatory variables go on the right side.

Let's go back to the lung cancer data and run a Cox regression on sex.

```
fit <- coxph(Surv(time, status)~sex, data=lung)
fit
```

```
Call:
coxph(formula = Surv(time, status) ~ sex, data = lung)
```

```
    coef exp(coef) se(coef)  z     p
sex -0.5       0.6      0.2 -3 0.001

Likelihood ratio test=11  on 1 df, p=0.001
n= 228, number of events= 165
```

The `exp(coef)` column contains e^{β_1} (see background section above for more info). This is the **hazard ratio** – the multiplicative effect of that variable on the hazard rate (for each unit increase in that variable). So, for a categorical variable like sex, going from male (baseline) to female results in approximately ~40% reduction in hazard. You could also flip the sign on the `coef` column, and take `exp(0.531)`, which you can interpret as being male resulting in a 1.7-fold increase in hazard, or that males die ad approximately 1.7x the rate per unit time as females (females die at 0.588x the rate per unit time as males).

Just remember:

- HR=1: No effect
- HR>1: Increase in hazard
- HR<1: Reduction in hazard (protective)

You'll also notice there's a p-value on the `sex` term, and a p-value on the overall model. That 0.00111 p-value is really close to the p=0.00131 p-value we saw on the Kaplan-Meier plot. That's because the KM plot is showing the log-rank test p-value. You can get this out of the Cox model with a call to `summary(fit)`. You can directly calculate the log-rank test p-value using `survdiff()`.

```
summary(fit)
```

```
Call:
coxph(formula = Surv(time, status) ~ sex, data = lung)

  n= 228, number of events= 165

      coef exp(coef) se(coef)     z Pr(>|z|)
sex -0.531     0.588    0.167 -3.18   0.0015

    exp(coef) exp(-coef) lower .95 upper .95
sex     0.588        1.7     0.424     0.816

Concordance= 0.579  (se = 0.021 )
Likelihood ratio test= 10.6  on 1 df,   p=0.001
Wald test            = 10.1  on 1 df,   p=0.001
Score (logrank) test = 10.3  on 1 df,   p=0.001
```

```
survdiff(Surv(time, status)~sex, data=lung)
```

```
Call:
survdiff(formula = Surv(time, status) ~ sex, data = lung)

        N Observed Expected (O-E)^2/E (O-E)^2/V
sex=1 138      112     91.6      4.55      10.3
sex=2  90       53     73.4      5.68      10.3

 Chisq= 10.3  on 1 degrees of freedom, p= 0.001
```

Let's create another model where we analyze all the variables in the dataset! This shows us how all the variables, when considered together, act to influence survival. Some are very strong predictors (sex, ECOG score). Interestingly, the Karnofsky performance score as rated by the physician was marginally significant, while the same score as rated by the patient was not.

```
fit <- coxph(Surv(time, status)~sex+age+ph.ecog+ph.karno+pat.karno+meal.cal+wt.loss, data=
fit
```

```
Call:
coxph(formula = Surv(time, status) ~ sex + age + ph.ecog + ph.karno +
    pat.karno + meal.cal + wt.loss, data = lung)

              coef exp(coef) se(coef)    z     p
sex       -6e-01     6e-01    2e-01 -2.7 0.006
age        1e-02     1e+00    1e-02  0.9 0.359
ph.ecog    7e-01     2e+00    2e-01  3.3 0.001
ph.karno   2e-02     1e+00    1e-02  2.0 0.046
pat.karno -1e-02     1e+00    8e-03 -1.5 0.123
meal.cal   3e-05     1e+00    3e-04  0.1 0.898
wt.loss   -1e-02     1e+00    8e-03 -1.8 0.065

Likelihood ratio test=28  on 7 df, p=2e-04
n= 168, number of events= 121
   (60 observations deleted due to missingness)
```

Exercise 6

Let's go back to the `colon` cancer dataset. Remember, you created a `colondeath` object in the first exercise that only includes survival (`etype==2`), not recurrence data points.

See `?colon` for more information about this dataset.
Take a look at `levels(colondeath$rx)`. This tells you that the `rx` variable is the type of treatment the patient was on, which is either nothing (coded `Obs`, short for Observation), Levamisole (coded `Lev`), or Levamisole + 5-fluorouracil (coded `Lev+5FU`). This is a factor variable coded with these levels, in that order. This means that `Obs` is treated as the baseline group, and other groups are dummy-coded to represent the respective group.

Table 9.1: With k levels of a categorical factor variable, you get *k-1* dummy variables created, each 0/1, indicating that the sample is a particular non-reference category. Having value 0 for all dummy variables indicates that the sample is baseline.

rx	Lev	Lev+5FU
Obs	0	0
Lev	1	0
Lev+5FU	0	1

Exercise 7

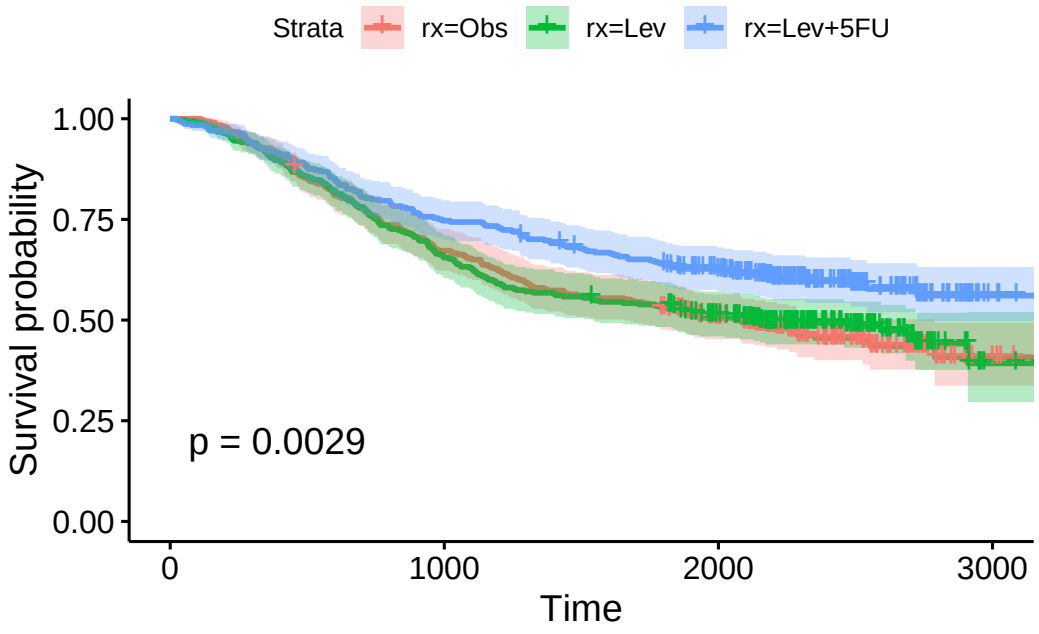
Run a Cox proportional hazards regression model against this `rx` variable. How do you interpret the result? Which treatment seems to be significantly different from the control (`Obs`ervation)?

```
           coef exp(coef) se(coef)    z     p
rxLev     -0.03      0.97     0.11 -0.2 0.809
rxLev+5FU -0.37      0.69     0.12 -3.1 0.002

Likelihood ratio test=12  on 2 df, p=0.002
n= 929, number of events= 452
```

Exercise 8

Show the results using a Kaplan-Meier plot, with confidence intervals and the p-value.



Exercise 9

Fit another Cox regression model accounting for age, sex, and the number of nodes with detectable cancer. Notice the test statistic on the likelihood ratio test becomes much larger, and the overall model becomes more significant. What do you think accounted for this increase in our ability to model survival?

```
             coef exp(coef) se(coef)    z      p
rxLev      -0.080     0.923    0.112 -0.7    0.5
rxLev+5FU -0.403     0.669    0.121 -3.3  8e-04
age         0.005     1.005    0.004  1.3    0.2
sex        -0.028     0.972    0.096 -0.3    0.8
nodes       0.093     1.097    0.009 10.5 <2e-16

Likelihood ratio test=88  on 5 df, p=<2e-16
n= 911, number of events= 441
   (18 observations deleted due to missingness)
```

9.2.5 Categorizing for KM plots

Let's go back to the lung data and look at a Cox model for age. Looks like age is very slightly significant when modeled as a continuous variable.

```
coxph(Surv(time, status)~age, data=lung)
```

```
Call:
coxph(formula = Surv(time, status) ~ age, data = lung)

     coef exp(coef) se(coef) z    p
age 0.019     1.019    0.009 2 0.04

Likelihood ratio test=4  on 1 df, p=0.04
n= 228, number of events= 165
```

Now that your regression analysis shows you that age is marginally significant, let's make a Kaplan-Meier plot. But, as we saw before, we can't just do this, because we'll get a separate curve for every unique value of age!

```
ggsurvplot(survfit(Surv(time, status)~age, data=lung))
```

One thing you might see here is an attempt to categorize a continuous variable into different groups – tertiles, upper quartile vs lower quartile, a median split, etc – so you can make the KM plot. But, how you make that cut is meaningful! Check out the help for `?cut`. `cut()` takes a continuous variable and some breakpoints and creats a categorical variable from that. Let's get the average age in the dataset, and plot a histogram showing the distribution of age.

```
mean(lung$age)
hist(lung$age)
ggplot(lung, aes(age)) + geom_histogram(bins=20)
```

Now, let's try creating a categorical variable on `lung$age` with cut pounts at 0, 62 (the mean), and +Infinity (no upper limit). We could continue adding a `labels=` option here to label the groupings we create, for instance, as "young" and "old". Finally, we could assign the result of this to a new object in the lung dataset.

```
cut(lung$age, breaks=c(0, 62, Inf))
```

```
  [1] (62,Inf] (62,Inf] (0,62]   (0,62]   (0,62]   (62,Inf] (62,Inf] (62,Inf]
  [9] (0,62]   (0,62]   (0,62]   (62,Inf] (62,Inf] (0,62]   (0,62]   (62,Inf]
 [17] (62,Inf] (62,Inf] (0,62]   (0,62]   (62,Inf] (0,62]   (0,62]   (0,62]
 [25] (62,Inf] (62,Inf] (0,62]   (62,Inf] (0,62]   (62,Inf] (62,Inf] (62,Inf]
 [33] (0,62]   (0,62]   (0,62]   (0,62]   (62,Inf] (62,Inf] (62,Inf] (62,Inf]
```

```
 [41] (62,Inf] (62,Inf] (0,62]   (0,62]   (62,Inf] (62,Inf] (62,Inf] (62,Inf]
 [49] (62,Inf] (0,62]   (62,Inf] (62,Inf] (62,Inf] (0,62]   (0,62]   (0,62]
 [57] (62,Inf] (0,62]   (0,62]   (62,Inf] (62,Inf] (0,62]   (62,Inf] (62,Inf]
 [65] (62,Inf] (62,Inf] (62,Inf] (62,Inf] (62,Inf] (62,Inf] (62,Inf] (0,62]
 [73] (62,Inf] (0,62]   (0,62]   (62,Inf] (0,62]   (0,62]   (62,Inf] (62,Inf]
 [81] (0,62]   (0,62]   (0,62]   (0,62]   (0,62]   (62,Inf] (0,62]   (0,62]
 [89] (0,62]   (62,Inf] (62,Inf] (62,Inf] (62,Inf] (0,62]   (62,Inf] (62,Inf]
 [97] (62,Inf] (62,Inf] (62,Inf] (62,Inf] (0,62]   (62,Inf] (0,62]   (62,Inf]
[105] (0,62]   (62,Inf] (0,62]   (62,Inf] (0,62]   (62,Inf] (62,Inf] (0,62]
[113] (62,Inf] (62,Inf] (0,62]   (62,Inf] (0,62]   (62,Inf] (62,Inf] (62,Inf]
[121] (62,Inf] (0,62]   (62,Inf] (62,Inf] (62,Inf] (62,Inf] (0,62]   (62,Inf]
[129] (62,Inf] (0,62]   (0,62]   (0,62]   (0,62]   (0,62]   (62,Inf] (62,Inf]
[137] (0,62]   (0,62]   (0,62]   (0,62]   (62,Inf] (62,Inf] (62,Inf] (62,Inf]
[145] (0,62]   (0,62]   (62,Inf] (0,62]   (62,Inf] (0,62]   (62,Inf] (0,62]
[153] (0,62]   (0,62]   (0,62]   (62,Inf] (62,Inf] (0,62]   (62,Inf] (0,62]
[161] (0,62]   (0,62]   (62,Inf] (62,Inf] (62,Inf] (0,62]   (0,62]   (0,62]
[169] (0,62]   (62,Inf] (0,62]   (0,62]   (0,62]   (0,62]   (0,62]   (0,62]
[177] (0,62]   (0,62]   (0,62]   (62,Inf] (0,62]   (0,62]   (62,Inf] (62,Inf]
[185] (0,62]   (0,62]   (62,Inf] (0,62]   (62,Inf] (0,62]   (62,Inf] (0,62]
[193] (0,62]   (62,Inf] (62,Inf] (62,Inf] (62,Inf] (62,Inf] (0,62]   (0,62]
 [ reached getOption("max.print") -- omitted 28 entries ]
Levels: (0,62] (62,Inf]
```

```
cut(lung$age, breaks=c(0, 62, Inf), labels=c("young", "old"))
```

```
  [1] old   old   young young young old   old   old   young young young old
 [13] old   young young old   old   old   young young old   young young young
 [25] old   old   young old   young old   old   old   young young young young
 [37] old   old   old   old   old   old   young young old   old   old   old
 [49] old   young old   old   old   young young young old   young young old
 [61] old   young old   old   old   old   old   old   old   old   old   young
 [73] old   young young old   young young old   old   young young young young
 [85] young old   young young young old   old   old   old   young old   old
 [97] old   old   old   old   young old   young old   young old   young old
[109] young old   old   young old   old   young old   young old   old   old
[121] old   young old   old   old   old   young old   old   young young young
[133] young young old   old   young young young young old   old   old   old
[145] young young old   young old   young old   young young young young old
[157] old   young old   young young young old   old   old   young young young
[169] young old   young young young young young young young young young old
[181] young young old   old   young young old   young old   young old   young
```

```
[193] young old   old   old   old   old   young young
 [ reached getOption("max.print") -- omitted 28 entries ]
Levels: young old
```

```r
# the base r way:
lung$agecat <- cut(lung$age, breaks=c(0, 62, Inf), labels=c("young", "old"))

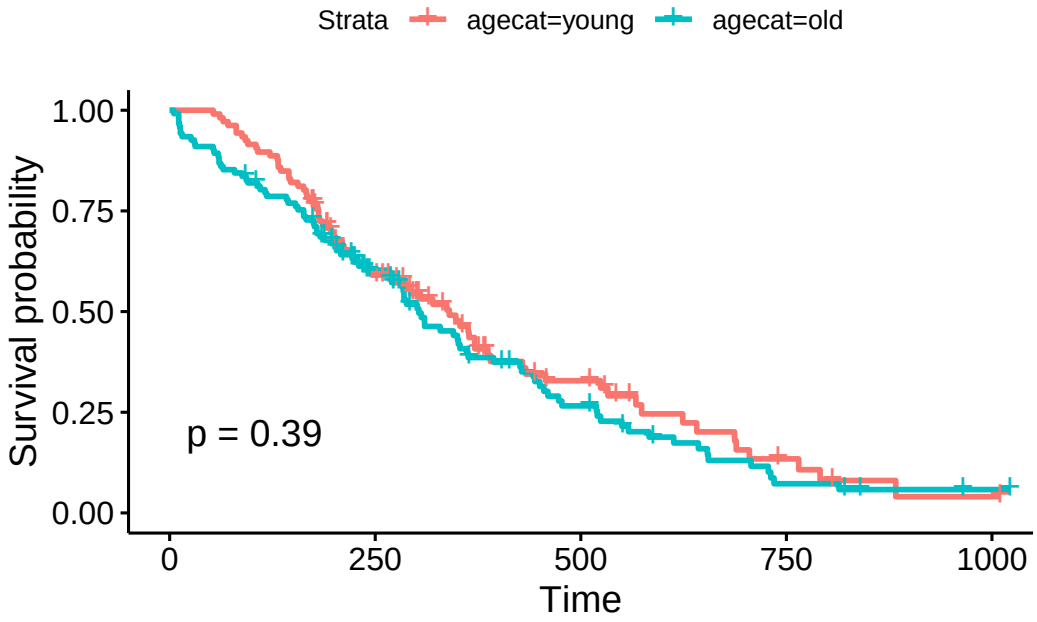
# or the dplyr way:
lung <- lung %>%
  mutate(agecat=cut(age, breaks=c(0, 62, Inf), labels=c("young", "old")))

head(lung)
```

```
# A tibble: 6 x 11
   inst  time status   age   sex ph.ecog ph.karno pat.karno meal.cal wt.loss
  <dbl> <dbl>  <dbl> <dbl> <dbl>   <dbl>    <dbl>     <dbl>    <dbl>   <dbl>
1     3   306      2    74     1       1       90       100     1175      NA
2     3   455      2    68     1       0       90        90     1225      15
3     3  1010      1    56     1       0       90        90       NA      15
4     5   210      2    57     1       1       90        60     1150      11
5     1   883      2    60     1       0      100        90       NA       0
6    12  1022      1    74     1       1       50        80      513       0
# i 1 more variable: agecat <fct>
```

Now, what happens when we make a KM plot with this new categorization? It looks like there's some differences in the curves between "old" and "young" patients, with older patients having slightly worse survival odds. But at p=.39, the difference in survival between those younger than 62 and older than 62 are not significant.

```r
ggsurvplot(survfit(Surv(time, status)~agecat, data=lung), pval=TRUE)
```

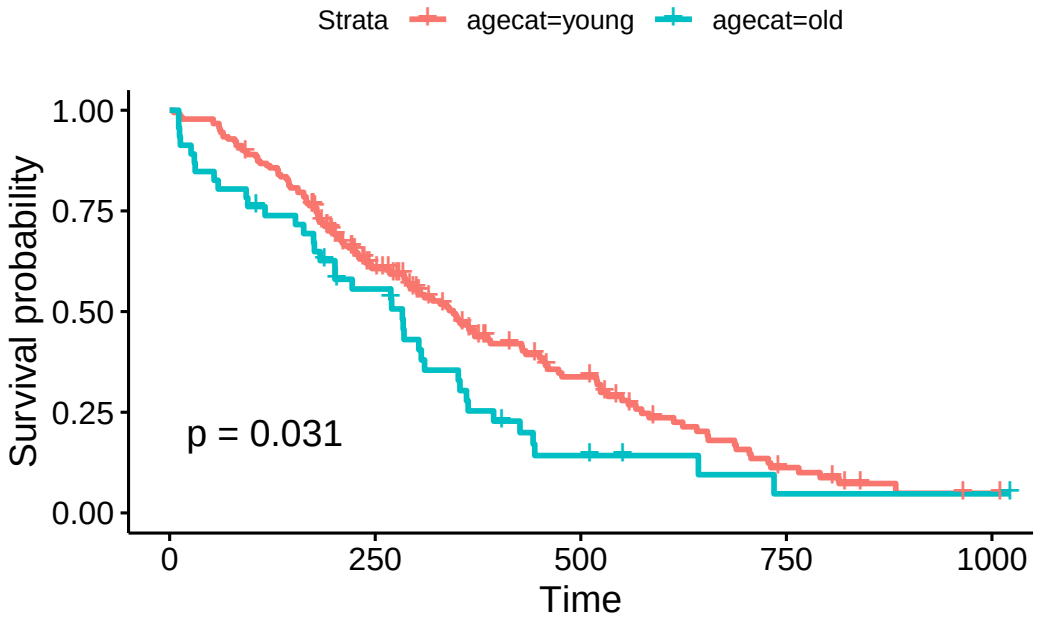


But, what if we chose a different cut point, say, 70 years old, which is roughly the cutoff for the upper quartile of the age distribution (see `?quantile`). The result is now marginally significant!

```
# the base r way:
lung$agecat <- cut(lung$age, breaks=c(0, 70, Inf), labels=c("young", "old"))

# or the dplyr way:
lung <- lung %>%
  mutate(agecat=cut(age, breaks=c(0, 70, Inf), labels=c("young", "old")))

# plot!
ggsurvplot(survfit(Surv(time, status)~agecat, data=lung), pval=TRUE)
```



Remember, the Cox regression analyzes the continuous variable over the whole range of its distribution, where the log-rank test on the Kaplan-Meier plot can change depending on how you categorize your continuous variable. They're answering a similar question in a different way: the regression model is asking, *"what is the effect of age on survival?"*, while the log-rank test and the KM plot is asking, *"are there differences in survival between those less than 70 and those greater than 70 years old?"*.

(New in survminer 0.2.4: the survminer package can now determine the optimal cutpoint for one or multiple continuous variables at once, using the `surv_cutpoint()` *and* `surv_categorize()` *functions. Refer to this blog post for more information.)*

9.3 TCGA

The Cancer Genome Atlas (TCGA) is a collaboration between the National Cancer Institute (NCI) and the National Human Genome Research Institute (NHGRI) that collected lots of clinical and genomic data across 33 cancer types. The entire TCGA dataset is over 2 petabytes worth of gene expression, CNV profiling, SNP genotyping, DNA methylation, miRNA profiling, exome sequencing, and other types of data. You can learn more about TCGA at cancergenome.nih.gov. The data is now housed at the Genomic Data Commons Portal. There are lots of ways to access TCGA data without actually downloading and parsing through the data from GDC. We'll cover more of these below. But first, let's look at an R package that provides convenient, direct access to TCGA data.

9.3.1 RTCGA

The RTCGA package (bioconductor.org/packages/RTCGA) and all the associated data packages provide convenient access to clinical and genomic data in TCGA. Each of the data packages is a separate package, and must be installed (once) individually.

```
# Load the bioconductor installer.
# Try http:// if https:// doesn't work.
source("https://bioconductor.org/biocLite.R")

# Install the main RTCGA package
biocLite("RTCGA")

# Install the clinical and mRNA gene expression data packages
biocLite("RTCGA.clinical")
biocLite("RTCGA.mRNA")
```

Let's load the RTCGA package, and use the `infoTCGA()` function to get some information about the kind of data available for each cancer type.

```
library(RTCGA)
infoTCGA()
```

9.3.1.1 Survival Analysis with RTCGA Clinical Data

Next, let's load the `RTCGA.clinical` package and get a little help about what's available there.

```
library(RTCGA.clinical)
?clinical
```

This tells us all the clinical datasets available for each cancer type. If we just focus on breast cancer, look at how big the data is! There are 1098 rows by 3703 columns in this data alone. Let's look at some of the variable names. ***Be careful with `View()` here*** – with so many columns, depending on which version of RStudio you have that may or may not have fixed this issue, Viewing a large dataset like this may lock up your RStudio.

```
dim(BRCA.clinical)
names(BRCA.clinical)
# View(BRCA.clinical)
```

We're going to use the `survivalTCGA()` function from the RTCGA package to pull out survival information from the clinical data. It does this by looking at vital status (dead or alive) and creating a `times` variable that's either the days to death or the days followed up before being censored. Look at the help for `?survivalTCGA` for more info. You give it a list of clinical datasets to pull from, and a character vector of variables to extract. Let's look at breast cancer, ovarian cancer, and glioblastoma multiforme. Let's just extract the cancer type (`admin.disease_code`).

```
# Create the clinical data
clin <- survivalTCGA(BRCA.clinical, OV.clinical, GBM.clinical,
                     extract.cols="admin.disease_code")
# Show the first few lines
head(clin)
```

```
           times bcr_patient_barcode patient.vital_status admin.disease_code
379.31.0    3767        TCGA-3C-AAAU                    0               brca
379.31.0.1  3801        TCGA-3C-AALI                    0               brca
379.31.0.2  1228        TCGA-3C-AALJ                    0               brca
379.31.0.3  1217        TCGA-3C-AALK                    0               brca
379.31.0.4   158        TCGA-4H-AAAK                    0               brca
379.31.0.5  1477        TCGA-5L-AAT0                    0               brca
```

```
# How many samples of each type?
table(clin$admin.disease_code)
```

```
brca  gbm   ov
1098  595  576
```

```
# Tabulate by outcome
xtabs(~admin.disease_code+patient.vital_status, data=clin) %>% addmargins()
```

```
                 patient.vital_status
admin.disease_code    0    1  Sum
              brca  994  104 1098
              gbm   149  446  595
              ov    279  297  576
              Sum  1422  847 2269
```

Now let's run a Cox PH model against the disease code. By default it's going to treat breast cancer as the baseline, because alphabetically it's first. But you can reorder this if you want with `factor()`.

```
coxph(Surv(times, patient.vital_status)~admin.disease_code, data=clin)
```

```
Call:
coxph(formula = Surv(times, patient.vital_status) ~ admin.disease_code,
    data = clin)

                      coef exp(coef) se(coef)  z      p
admin.disease_codegbm  2.9      17.9      0.1 26 <2e-16
admin.disease_codeov   1.5       4.7      0.1 13 <2e-16

Likelihood ratio test=904  on 2 df, p=<2e-16
n= 2269, number of events= 847
```

This tells us that compared to the baseline `brca` group, GBM patients have a ~18x increase in hazards, and ovarian cancer patients have ~5x worse survival. Let's create a survival curve, visualize it with a Kaplan-Meier plot, and show a table for the first 5 years survival rates.

```
sfit <- survfit(Surv(times, patient.vital_status)~admin.disease_code, data=clin)
summary(sfit, times=seq(0,365*5,365))
```

```
Call: survfit(formula = Surv(times, patient.vital_status) ~ admin.disease_code,
    data = clin)

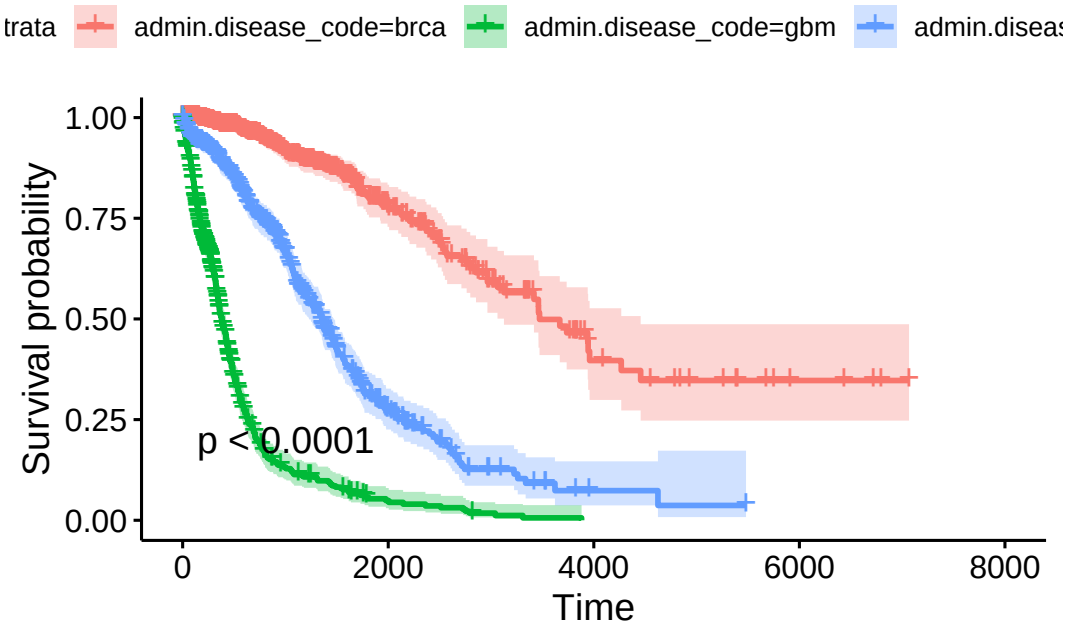
                admin.disease_code=brca
 time n.risk n.event survival std.err lower 95% CI upper 95% CI
    0   1096       0    1.000 0.00000        1.000        1.000
  365    588      13    0.981 0.00516        0.971        0.992
  730    413      11    0.958 0.00851        0.942        0.975
 1095    304      20    0.905 0.01413        0.878        0.933
 1460    207       9    0.873 0.01719        0.840        0.908
 1825    136      14    0.799 0.02474        0.752        0.849

                admin.disease_code=gbm
 time n.risk n.event survival std.err lower 95% CI upper 95% CI
    0    595       2   0.9966 0.00237       0.9920       1.0000
  365    224     257   0.5110 0.02229       0.4692       0.5567
  730     75     127   0.1998 0.01955       0.1649       0.2420
```

```
1095      39       31    0.1135 0.01617        0.0858        0.1500
1460      27        9    0.0854 0.01463        0.0610        0.1195
1825      12        9    0.0534 0.01259        0.0336        0.0847

                admin.disease_code=ov
time n.risk n.event survival std.err lower 95% CI upper 95% CI
   0    576       0    1.000  0.0000        1.000         1.000
 365    411      59    0.888  0.0139        0.861         0.915
 730    314      55    0.761  0.0198        0.724         0.801
1095    210      59    0.602  0.0243        0.556         0.651
1460    133      49    0.451  0.0261        0.402         0.505
1825     78      39    0.310  0.0260        0.263         0.365
```

```
ggsurvplot(sfit, conf.int=TRUE, pval=TRUE)
```



9.3.1.2 Gene Expression Data

Let's load the gene expression data.

```
library(RTCGA.mRNA)
?mRNA
```

Take a look at the size of the `BRCA.mRNA` dataset, show a few rows and columns.

```
dim(BRCA.mRNA)
BRCA.mRNA[1:5, 1:5]
```

Extra credit assignment: See if you can figure out how to join the gene expression data to the clinical data for any particular cancer type.

```
# Take the mRNA data
BRCA.mRNA %>%
  # then make it a tibble (nice printing while debugging)
  as_tibble() %>%
  # then get just a few genes
  select(bcr_patient_barcode, PAX8, GATA3, ESR1) %>%
  # then trim the barcode (see head(clin), and ?substr)
  mutate(bcr_patient_barcode = substr(bcr_patient_barcode, 1, 12)) %>%
  # then join back to clinical data
  inner_join(clin, by="bcr_patient_barcode")
```

Similar to how `survivalTCGA()` was a nice helper function to pull out survival information from multiple different clinical datasets, `expressionsTCGA()` can pull out specific gene expression measurements across different cancer types. See the help for `?expressionsTCGA`. Let's pull out data for PAX8, GATA-3, and the estrogen receptor genes from breast, ovarian, and endometrial cancer, and plot the expression of each with a box plot.

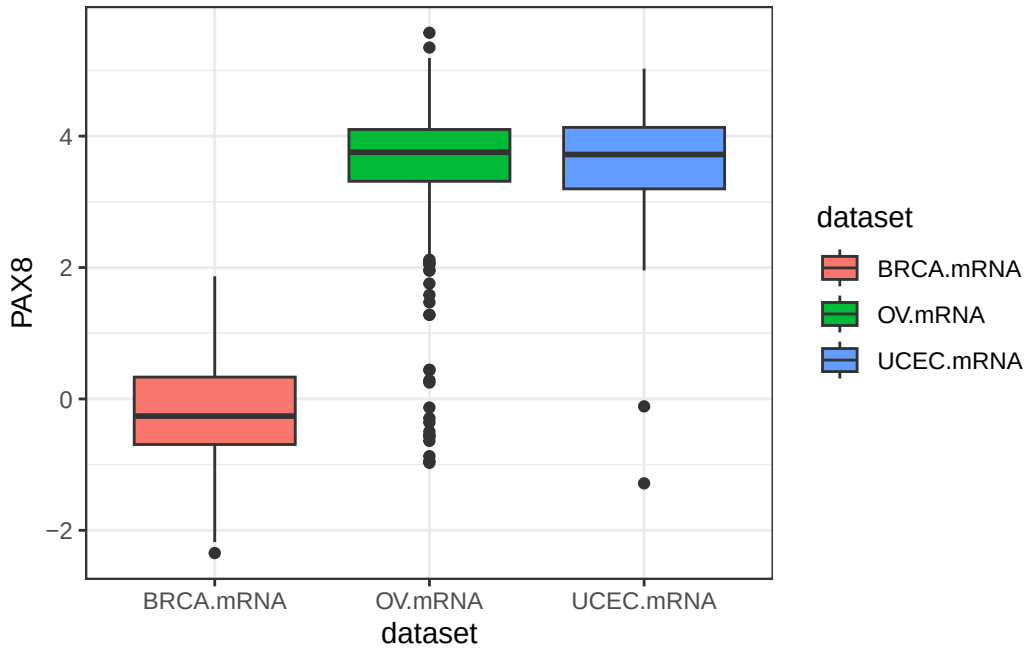
```
library(ggplot2)
expr <- expressionsTCGA(BRCA.mRNA, OV.mRNA, UCEC.mRNA,
                        extract.cols = c("PAX8", "GATA3", "ESR1"))
head(expr)
```

```
# A tibble: 6 x 5
  bcr_patient_barcode          dataset     PAX8 GATA3   ESR1
  <chr>                        <chr>      <dbl> <dbl>  <dbl>
1 TCGA-A1-A0SD-01A-11R-A115-07 BRCA.mRNA -0.542  2.87  3.08
2 TCGA-A1-A0SE-01A-11R-A084-07 BRCA.mRNA -0.595  2.17  2.39
3 TCGA-A1-A0SH-01A-11R-A084-07 BRCA.mRNA  0.500  1.32  0.791
4 TCGA-A1-A0SJ-01A-11R-A084-07 BRCA.mRNA -0.588  1.84  2.50
5 TCGA-A1-A0SK-01A-12R-A084-07 BRCA.mRNA -0.965 -6.03 -4.86
6 TCGA-A1-A0SM-01A-11R-A084-07 BRCA.mRNA  0.573  1.80  2.80
```

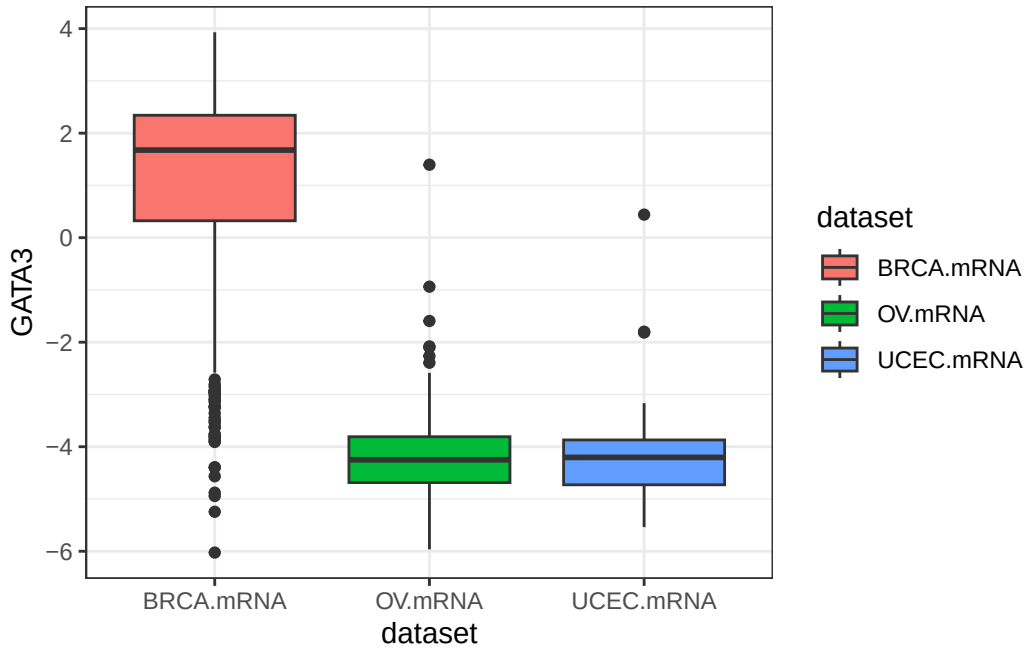
```
table(expr$dataset)
```

```
BRCA.mRNA    OV.mRNA UCEC.mRNA
      590        561         54
```

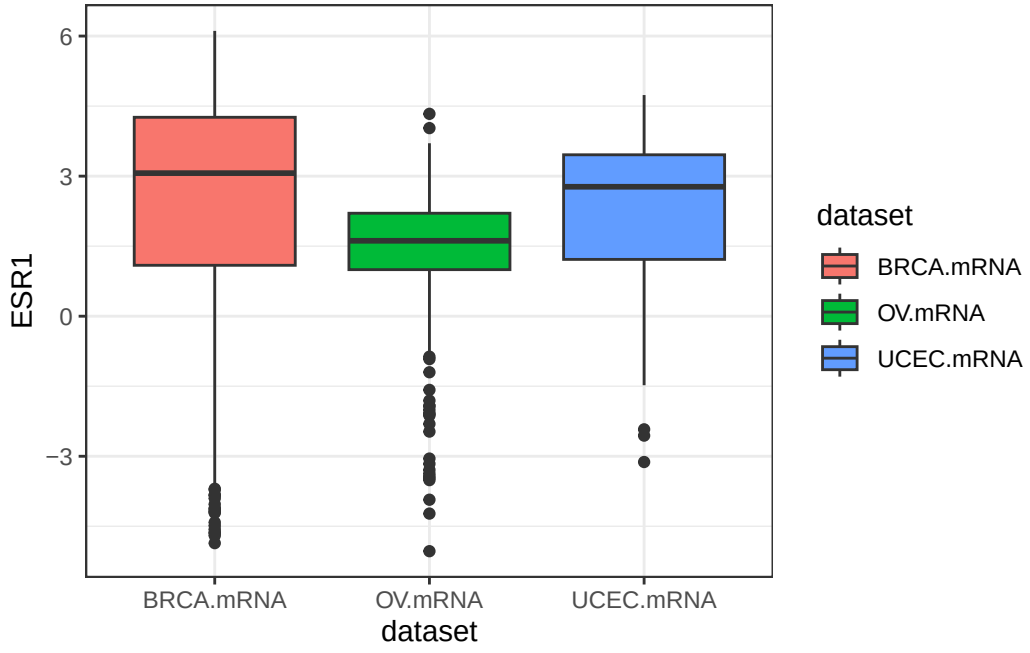
```
ggplot(expr, aes(dataset, PAX8, fill=dataset)) + geom_boxplot()
```



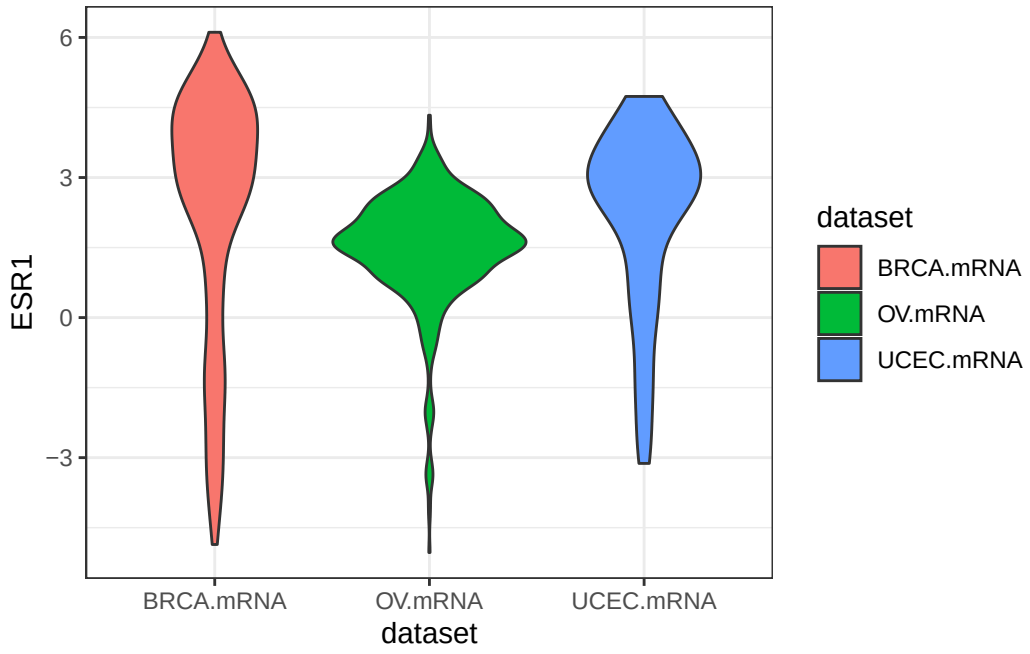

```
ggplot(expr, aes(dataset, GATA3, fill=dataset)) + geom_boxplot()
```




```
ggplot(expr, aes(dataset, ESR1, fill=dataset)) + geom_boxplot()
```

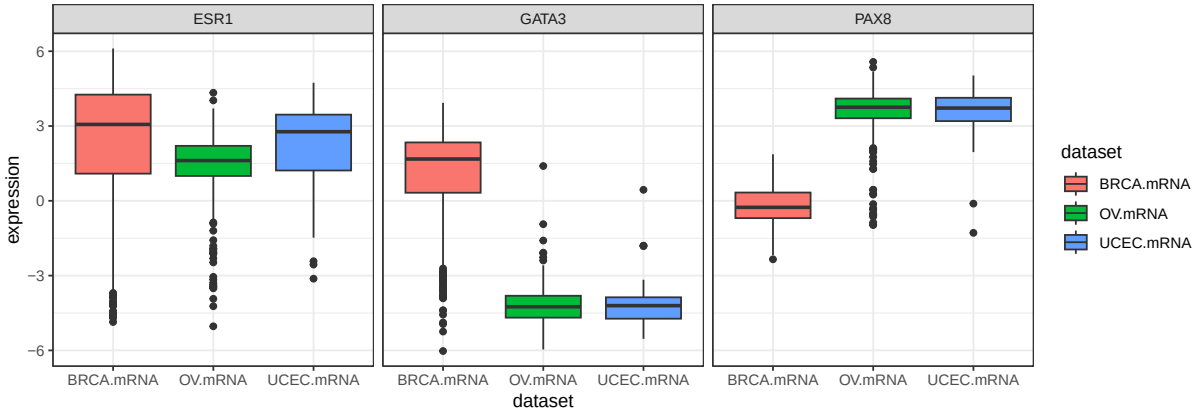



```
ggplot(expr, aes(dataset, ESR1, fill=dataset)) + geom_violin()
```



We could also use tidyr to do this all in one go.

```
library(tidyr)
expr %>%
  as_tibble() %>%
  gather(gene, expression, PAX8, GATA3, ESR1) %>%
  ggplot(aes(dataset, expression, fill=dataset)) +
    geom_boxplot() +
    facet_wrap(~gene)
```



Exercise 10

The "KIPAN" cohort (in `KIPAN.clinical`) is the pan-kidney cohort, consisting of KICH (chromaphobe renal cell carcinoma), KIRC (renal clear cell carcinoma), and KIPR (papillary cell carcinoma). The `KIPAN.clinical` has `KICH.clinical`, `KIRC.clinical`, and `KIPR.clinical` all combined.
Using `survivalTCGA()`, create a new object called `clinkid` using the `KIPAN.clinical` cohort. For the columns to extract, get both the disease code and the patient's gender (`extract.cols=c("admin.disease_code", "patient.gender")`). The first few rows will look like this.

```
           times bcr_patient_barcode patient.vital_status admin.disease_code
226.62.0    1158        TCGA-KL-8323                    1               kich
226.62.0.1  4311        TCGA-KL-8324                    0               kich
226.62.0.2   725        TCGA-KL-8325                    1               kich
226.62.0.3  3322        TCGA-KL-8326                    0               kich
226.62.0.4  3553        TCGA-KL-8327                    0               kich
226.62.0.5  3127        TCGA-KL-8328                    0               kich
           patient.gender
226.62.0           female
226.62.0.1         female
226.62.0.2         female
226.62.0.3           male
226.62.0.4         female
226.62.0.5           male
```

Exercise 11

The `xtabs()` command will produce tables of counts for categorical variables. Here's an example for how to use `xtabs()` for the built-in colon cancer dataset, which will tell you the number of samples split by sex and by treatment.

```
xtabs(~rx+sex, data=colon)
```

```
         sex
rx          0   1
  Obs     298 332
  Lev     266 354
  Lev+5FU 326 282
```

Use the same command to examine how many samples you have for each kidney sample type, separately by sex.

```
                  patient.gender
admin.disease_code female male
              kich     51   61
              kirc    191  346
              kirp     76  212
```

Exercise 12

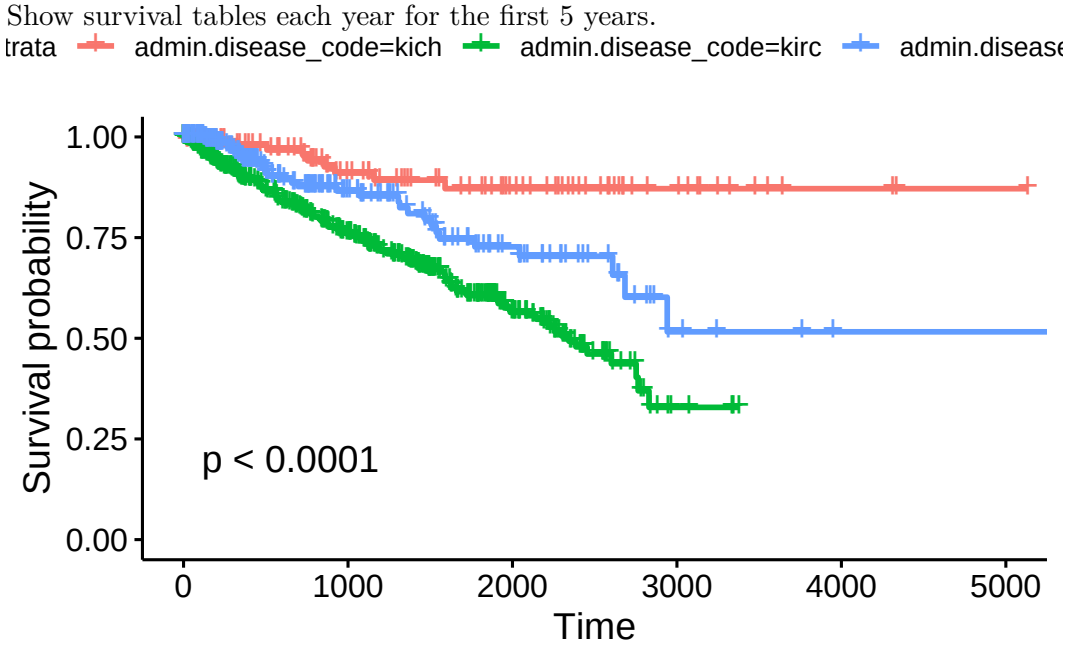
Run a Cox PH regression on the cancer type and gender. What's the effect of gender? Is it significant? How does survival differ by each type? Which has the worst prognosis?

```
                        coef exp(coef) se(coef)    z     p
admin.disease_codekirc  1.59      4.92     0.34  4.6 4e-06
admin.disease_codekirp  1.00      2.71     0.38  2.6 0.009
patient.gendermale     -0.06      0.94     0.15 -0.4 0.672

Likelihood ratio test=39  on 3 df, p=1e-08
n= 937, number of events= 203
```

Exercise 13

Create survival curves for each different subtype. a. Produce a Kaplan-Meier plot. b. Show survival tables each year for the first 5 years.




```
Call: survfit(formula = Surv(times, patient.vital_status) ~ admin.disease_code,
```

```
   data = clinkid)

                admin.disease_code=kich
time n.risk n.event survival std.err lower 95% CI upper 95% CI
   0    111       0    1.000  0.0000        1.000        1.000
 365     86       2    0.980  0.0144        0.952        1.000
 730     72       2    0.954  0.0226        0.911        0.999
1095     54       3    0.910  0.0329        0.848        0.977
1460     44       1    0.893  0.0366        0.824        0.967
1825     38       1    0.871  0.0415        0.794        0.957

                admin.disease_code=kirc
time n.risk n.event survival std.err lower 95% CI upper 95% CI
   0    536       0    1.000  0.0000        1.000        1.000
 365    385      49    0.895  0.0142        0.868        0.924
 730    313      32    0.816  0.0186        0.781        0.853
1095    250      26    0.744  0.0217        0.703        0.788
1460    181      20    0.678  0.0243        0.633        0.728
1825    112      16    0.606  0.0277        0.554        0.663

                admin.disease_code=kirp
time n.risk n.event survival std.err lower 95% CI upper 95% CI
   0    288       0    1.000  0.0000        1.000        1.000
 365    145      10    0.941  0.0182        0.906        0.977
 730    100       8    0.877  0.0278        0.824        0.933
1095     67       2    0.853  0.0316        0.793        0.917
1460     54       3    0.810  0.0388        0.737        0.889
1825     36       5    0.727  0.0495        0.636        0.831
```

9.3.2 Other TCGA Resources

RTCGA isn't the only resource providing easy access to TCGA data. In fact, it isn't even the only R/Bioconductor package. Take a look at some of the other resources shown below.

- **TCGAbiolinks**: another R package that allows direct query and analysis from the NCI GDC.
  - R package: bioconductor.org/packages/TCGAbiolinks
  - Paper: *Nucleic Acids Research* 2015 DOI: 10.1093/nar/gkv1507.
- **cBioPortal**: cbioportal.org
  - Nice graphical user interface

  - Quick/easy summary info on patients, demographics, mutations, copy number alterations, etc.
  - Query individual genes, find coexpressed genes
  - Survival analysis against different subtypes, expression, CNAs, etc.

- **OncoLnc**: oncolnc.org
  - Focus on survival analysis and RNA-seq data.
  - Simple query interface across all cancers for any mRNA, miRNA, or lncRNA gene (try SERPINA1)
  - Precomputed Cox PH regression for every gene, for every cancer
  - Kaplan-Meier plots produced on demand

- TANRIC: focus on noncoding RNA
- MEXPRESS: focus on methylation and gene expression

10 Predictive Analytics: Predicting and Forecasting Influenza

This chapter will provide hands-on instruction for using machine learning algorithms to predict a disease outcome. We will cover data cleaning, feature extraction, imputation, and using a variety of models to try to predict disease outcome. We will use resampling strategies to assess the performance of predictive modeling procedures such as Random Forest, stochastic gradient boosting, elastic net regularized regression (LASSO), and k-nearest neighbors. We will also demonstrate demonstrate how to *forecast* future trends given historical infectious disease surveillance data using methodology that accounts for seasonality and nonlinearity.

Handout: Predictive Modeling Handout.

10.1 Predictive Modeling

Here we're going to use some epidemiological data collected during an influenza A (H7N9) outbreak in China in 2013. Of 134 cases with data, 31 died, 46 recovered, but 57 cases do not have a recorded outcome. We'll develop models capable of predicting death or recovery from the unlabeled cases. Along the way, we will:

- Do some exploratory data analysis and data visualization to get an overall sense of the data we have.
- Extract and recode *features* from the raw data that are more amenable to data mining / machine learning algorithms.
- Impute missing data points from some of the predictor variables.
- Use a framework that enables consistent access to hundreds of classification and regression algorithms, and that facilitates automated parameter tuning using bootstrapping-based resampling for model assessment.
- We will develop models using several different approaches (Random Forest, stochastic gradient boosting, elastic net regularized logistic regression, k-nearest neighbor) by training and testing the models on the data where the outcome is known
- We will compare the performance of each of the models and apply the best to predict the outcome for cases where we didn't know the outcome.

10.1.1 H7N9 Outbreak Data

The data we're using here is from the 2013 outbreak of influenza A H7N9 in China, analyzed by Kucharski et al., published in 2014.

> **Publication**: A. Kucharski, H. Mills, A. Pinsent, C. Fraser, M. Van Kerkhove, C. A. Donnelly, and S. Riley. 2014. Distinguishing between reservoir exposure and human-to-human transmission for emerging pathogens using case onset data. *PLOS Currents Outbreaks* (2014) Mar 7 Edition 1.
>
> **Data**: Kucharski A, Mills HL, Pinsent A, Fraser C, Van Kerkhove M, Donnelly CA, Riley S (2014) Data from: Distinguishing between reservoir exposure and human-to-human transmission for emerging pathogens using case onset data. *Dryad Digital Repository*. https://doi.org/10.5061/dryad.2g43n.

The data is made available in the outbreaks package, which is a collection of several simulated and real outbreak datasets, and has been very slightly modified for use here. The analysis we'll do here is inspired by and modified in part from a similar analysis by Shirin Glander.

There are two datasets available in data.zip:

1. **h7n9.csv**: The original dataset. Contains the following variables, with lots of missing data throughout.
    - `case_id`: the sample identifier
    - `date_onset`: date of onset of syptoms
    - `date_hospitalization`: date the patient was hospitalized, if available
    - `date_outcome`: date the outcome (recovery, death) was observed, if available
    - `outcome`: "Death" or "Recover," if available
    - `gender`: male (m) or female (f)
    - `age`: age of the individual, if known
    - `province`: either Shanghai, Jiangsu, Zhejiang, or Other (lumps together less common provinces)
2. **h7n9_analysisready.csv**: The "analysis-ready" dataset. This data has been cleaned up, with some "feature extraction" / variable recoding done to make the data more suitable to data mining / machine learning methods used here. We still have the outcome variable, either *Death*, *Recover* or unknown (NA).
    - `case_id`: *(same as above)*
    - `outcome`: *(same as above)*
    - `age`: *(same as above, imputed if unknown)*
    - `male`: Instead of sex (m/f), this is a 0/1 indicator, where 1=male, 0=female.
    - `hospital`: Indicator variable whether or not the patient was hospitalized
    - `days_to_hospital`: The number of days between onset and hospitalization
    - `days_to_outcome`: The number of days between onset and outcome (if available)

- **early_outcome**: Whether or not the outcome was recorded prior to the median date of the outcome in the dataset
- **Jiangsu**: Indicator variable: 1 = the patient was from the Jiangsu province.
- **Shanghai**: Indicator variable: 1 = the patient was from the Shanghai province.
- **Zhejiang**: Indicator variable: 1 = the patient was from the Zhejiang province.
- **Other**: Indicator variable: 1 = the patient was from some other less common province.

10.1.2 Importing H7N9 data

First, let's load the packages we'll need initially.

```
library(dplyr)
library(readr)
library(tidyr)
library(ggplot2)
```

Now let's read in the data and take a look. Notice that it correctly read in the dates as date-formatted variables. Later on, when we run functions such as `median()` on a date variable, it knows how to handle that properly. You'll also notice that there are missing values throughout.

```
# Read in data
flu <- read_csv("data/h7n9.csv")

# View in RStudio (capital V)
# View(flu)

# Take a look
flu
```

```
# A tibble: 134 x 8
   case_id date_onset date_hospitalization date_outcome outcome gender   age
   <chr>   <date>     <date>               <date>       <chr>   <chr>  <dbl>
 1 case_1  2013-02-19 NA                   2013-03-04   Death   m         58
 2 case_2  2013-02-27 2013-03-03           2013-03-10   Death   m          7
 3 case_3  2013-03-09 2013-03-19           2013-04-09   Death   f         11
 4 case_4  2013-03-19 2013-03-27           NA           <NA>    f         18
 5 case_5  2013-03-19 2013-03-30           2013-05-15   Recover f         20
 6 case_6  2013-03-21 2013-03-28           2013-04-26   Death   f          9
 7 case_7  2013-03-20 2013-03-29           2013-04-09   Death   m         54
```

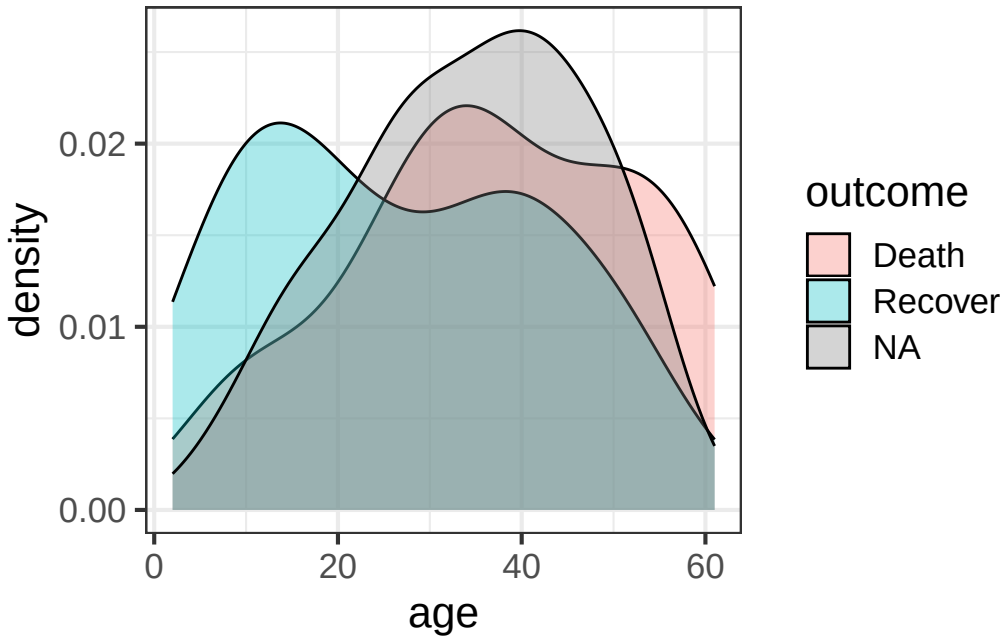
```
 8 case_8  2013-03-07 2013-03-18           2013-03-27   Death   m          14
 9 case_9  2013-03-25 2013-03-25           NA           <NA>    m          39
10 case_10 2013-03-28 2013-04-01           2013-04-03   Death   m          20
# i 124 more rows
# i 1 more variable: province <chr>
```

10.1.3 Exploratory data analysis

Let's use ggplot2 to take a look at the data. Refer back to the visualization section (Chapter 5) if you need a refresher.

The **outcome** variable is the thing we're most interested in here – it's the thing we want to eventually predict for the unknown cases. Let's look at the distribution of that outcome variable (Death, Recover or unknown (NA)), by **age**. We'll create a density distribution looking at age, with the fill of the distribution colored by outcome status.

```
ggplot(flu, aes(age)) + geom_density(aes(fill=outcome), alpha=1/3)
```

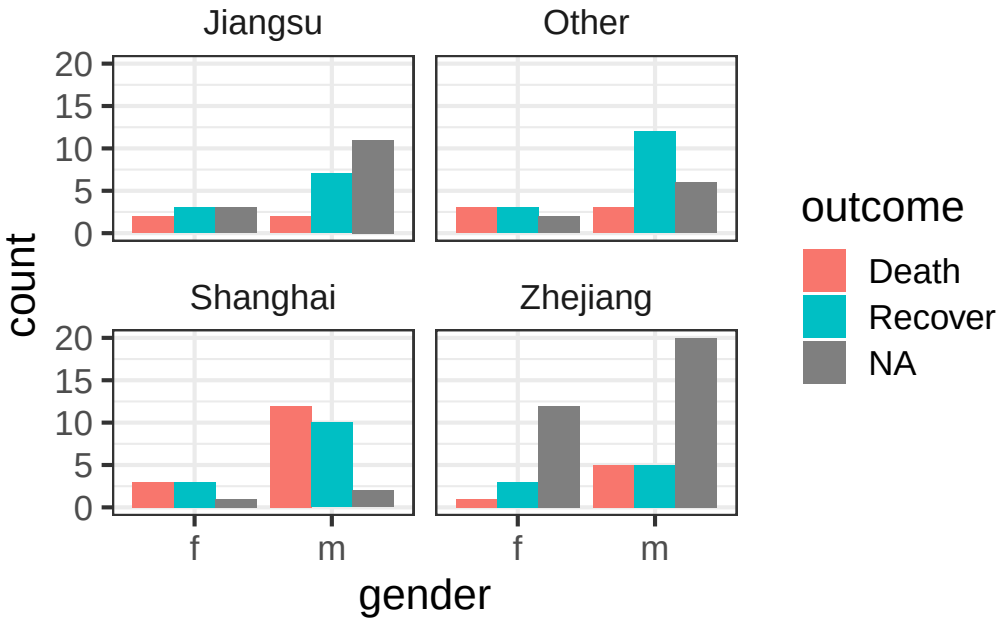


Let's look at the counts of the number of deaths, recoveries, and unknowns by sex, then separately by province.

```
ggplot(flu, aes(gender)) +
  geom_bar(aes(fill=outcome), position="dodge")
```

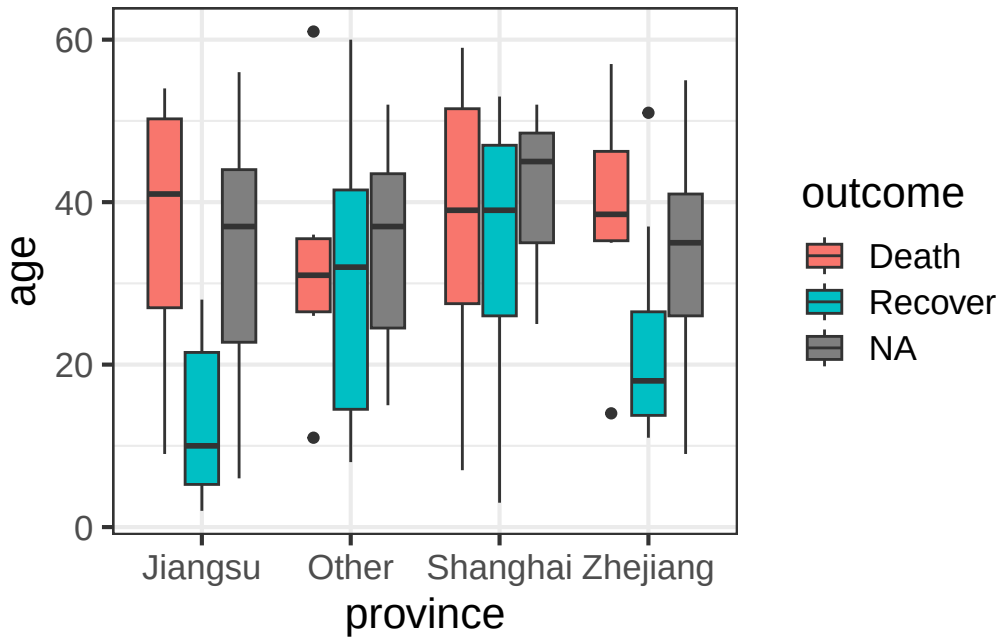
We can simply add a `facet_wrap` to split by province.

```
ggplot(flu, aes(gender)) +
  geom_bar(aes(fill=outcome), position="dodge") +
  facet_wrap(~province)
```



Let's draw a boxplot showing the age distribution by province, by outcome. This shows that there's a higher rate of death in older individuals but this is only observed in Jiangsu and Zhejiang provinces.

```
# First just by age
ggplot(flu, aes(province, age)) + geom_boxplot()
# Then by age and outcome
ggplot(flu, aes(province, age)) + geom_boxplot(aes(fill=outcome))
```



Let's try something a little bit more advanced. First, take a look at the data again.

```
flu
```

Notice how we have three different date variables: date of onset, hospitalization, and outcome. I'd like to draw a plot showing the date on the x-axis with a line connecting the three points from onset, to hospitalization, to outcome (if known) for each patient. I'll put age on the y-axis so the individuals are separated, and I'll do this faceted by province.

First we need to use the `gather` function from the **tidyr** package to gather up all the `date_?` variables into a single column we'll call `key`, with the actual values being put into a new column called `date`.

```
# Gather the date columns
flugather <- flu %>%
  gather(key, date, starts_with("date_"))

# Look at the data as is
# flugather

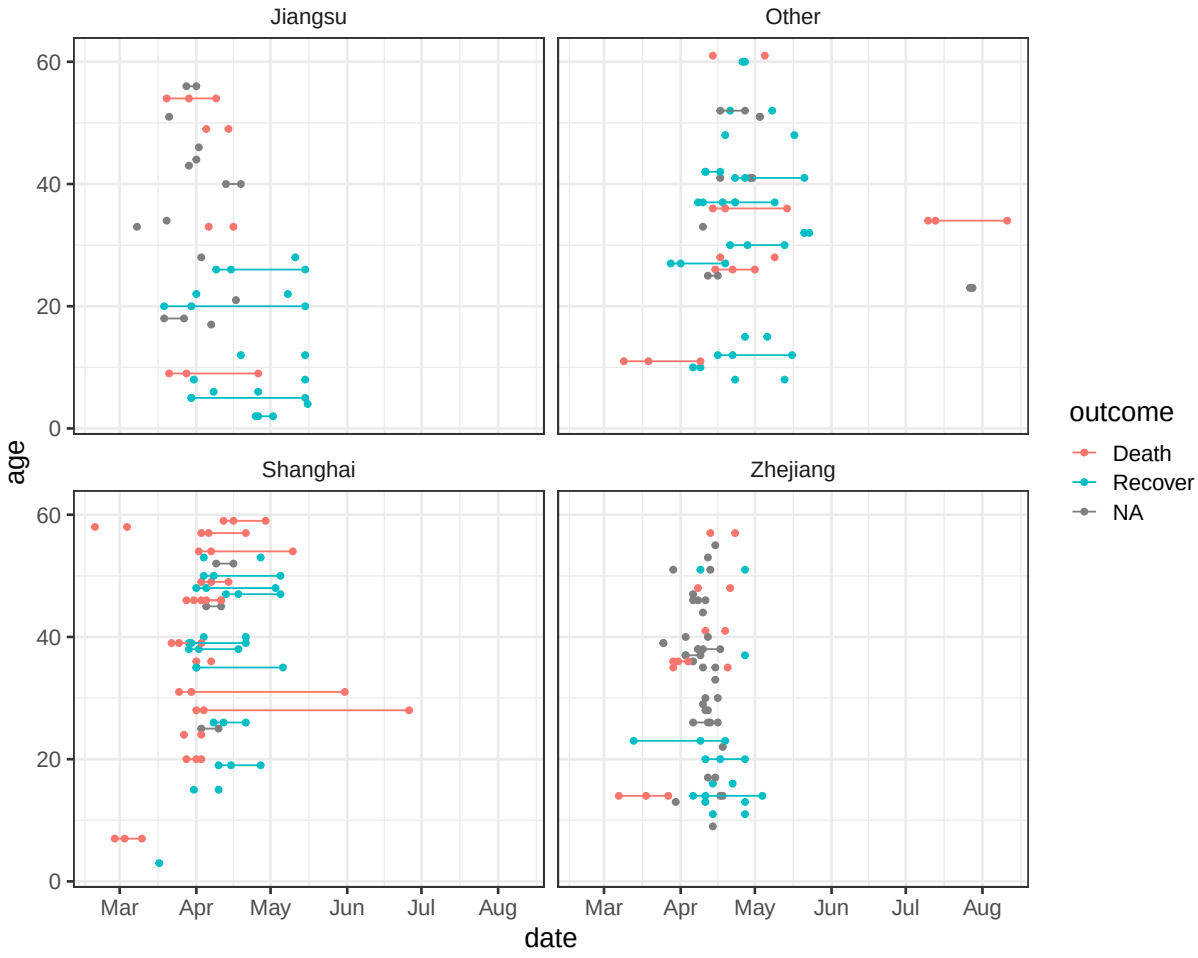
# Better: Show the data arranged by case_id so you see the three entries
flugather %>% arrange(case_id)
```

```
# A tibble: 402 x 7
```

```
   case_id  outcome gender   age province key                  date
   <chr>    <chr>   <chr>  <dbl> <chr>    <chr>                <date>
 1 case_1   Death   m         58 Shanghai date_onset          2013-02-19
 2 case_1   Death   m         58 Shanghai date_hospitalization NA
 3 case_1   Death   m         58 Shanghai date_outcome        2013-03-04
 4 case_10  Death   m         20 Shanghai date_onset          2013-03-28
 5 case_10  Death   m         20 Shanghai date_hospitalization 2013-04-01
 6 case_10  Death   m         20 Shanghai date_outcome        2013-04-03
 7 case_100 <NA>    m         30 Zhejiang date_onset          2013-04-16
 8 case_100 <NA>    m         30 Zhejiang date_hospitalization NA
 9 case_100 <NA>    m         30 Zhejiang date_outcome        NA
10 case_101 <NA>    f         51 Zhejiang date_onset          2013-04-13
# i 392 more rows
```

Now that we have this, let's visualize the number of days that passed between onset, hospitalization and outcome, for each case. We see that there are lots of unconnected points, especially in Jiangsu and Zhejiang provinces, where one of these dates isn't known.

```
ggplot(flugather, aes(date, y=age, color=outcome)) +
  geom_point() +
  geom_path(aes(group=case_id)) +
  facet_wrap(~province)
```



10.1.4 Feature Extraction

The variables in our data are useful for summary statistics, visualization, EDA, etc. But we need to do some *feature extraction* or variable recoding to get the most out of machine learning models.

- Age: we'll keep this one as is.
- Gender: instead of m/f, let's convert this into a binary indicator variable where 0=female, 1=male.
- Province: along the same lines, let's create binary classifiers that indicate Shanghai, Zhejiang, Jiangsu, or other provinces.
- Hospitalization: let's create a binary classifier where 0=not hospitalized, 1=hospitalized.
- Dates: Let's also take the *dates* of onset, hospitalization, and outcome, and transform these into *days* between onset and hospitalization, and days from onset to outcome. The

algorithms aren't going to look at one column then another to do this math – we have to extract this feature ourselves.
- Early outcome: let's create another binary 0/1 indicating whether someone had an early outcome (earlier than the median outcome date observed).

Let's build up this pipeline one step at a time. If you want to skip ahead, you can simply read in the already extracted/recoded/imputed dataset at data/h7n9_analysisready.csv.

First, let's make a backup of the original data in case we mess something up.

```
flu_orig <- flu
```

10.1.4.1 Create gender / hospitalization indicators

Now let's start recoding, one step at a time. First of all, when we mutate to add a new variable, we can put in a logical comparison to tell us whether a statement is TRUE or FALSE. For example, let's look at the gender variable.

```
flu$gender
```

We can ask if gender is male ("m") like this:

```
flu$gender=="m"
```

So we can do that with a mutate statement on a pipeline. Once we do that, we can remove the old gender variable. E.g.:

```
flu %>%
  mutate(male = gender=="m") %>%
  select(-gender)
```

Similarly, let's get an indicator whether someone was hospitalized or not. If hospitalization is missing, this would return TRUE. If you want to ask whether they are *not* missing, you would use `!` to negate the logical question, i.e., `!is.na(flu$date_hospitalization)`.

```
flu$date_hospitalization
is.na(flu$date_hospitalization)
!is.na(flu$date_hospitalization)
```

So now, let's add that to our pipeline from above.

```
flu %>%
  mutate(male = gender=="m") %>%
  select(-gender) %>%
  mutate(hospital = !is.na(date_hospitalization))
```

10.1.4.2 Convert dates to "days to ____"

Let's continue to add days from onset to hospitalization and days to outcome by subtracting one date from the other, and converting the value to numeric. We'll also create an early outcome binary variable indicating whether the date of the outcome was less than the median, after removing missing variables. We'll finally remove all the variables that start with "date." Finally, we'll use the `mutate_if` function, which takes a predicate and an action function. We'll ask – *if* the variable is logical (TRUE/FALSE), turn it into an integer (1/0).

```
# What's the median outcome date?
median(flu$date_outcome, na.rm=TRUE)

# Run the whole pipeline
flu %>%
  mutate(male = gender=="m") %>%
  select(-gender) %>%
  mutate(hospital = !is.na(date_hospitalization)) %>%
  mutate(days_to_hospital = as.numeric(date_hospitalization - date_onset)) %>%
  mutate(days_to_outcome  = as.numeric(date_outcome - date_onset)) %>%
  mutate(early_outcome = date_outcome < median(date_outcome, na.rm=TRUE)) %>%
  select(-starts_with("date")) %>%
  mutate_if(is.logical, as.integer)
```

Once you're satisfied your pipeline works, reassign the pipeline back to the `flu` object itself (remember, we created the backup above in case we messed something up here).

```
# Make the assignment
flu <- flu %>%
  mutate(male = gender=="m") %>%
  select(-gender) %>%
  mutate(hospital = !is.na(date_hospitalization)) %>%
  mutate(days_to_hospital = as.numeric(date_hospitalization - date_onset)) %>%
  mutate(days_to_outcome  = as.numeric(date_outcome - date_onset)) %>%
  mutate(early_outcome = date_outcome < median(date_outcome, na.rm=TRUE)) %>%
  select(-starts_with("date")) %>%
```

```
  mutate_if(is.logical, as.integer)

# Take a look
flu
```

```
# A tibble: 134 x 9
   case_id outcome   age province  male hospital days_to_hospital
   <chr>   <chr>   <dbl> <chr>    <int>    <int>            <dbl>
 1 case_1  Death      58 Shanghai     1        0               NA
 2 case_2  Death       7 Shanghai     1        1                4
 3 case_3  Death      11 Other        0        1               10
 4 case_4  <NA>       18 Jiangsu      0        1                8
 5 case_5  Recover    20 Jiangsu      0        1               11
 6 case_6  Death       9 Jiangsu      0        1                7
 7 case_7  Death      54 Jiangsu      1        1                9
 8 case_8  Death      14 Zhejiang     1        1               11
 9 case_9  <NA>       39 Zhejiang     1        1                0
10 case_10 Death      20 Shanghai     1        1                4
# i 124 more rows
# i 2 more variables: days_to_outcome <dbl>, early_outcome <int>
```

10.1.4.3 Create indicators for province

Now, there's one more thing we want to do. Instead of a single "province" variable that has multiple levels, we want to do the dummy coding ourselves. When we ran regression models R handled this internally without our intervention. But we need to be explicit here. Here's one way to do it.

First, there's a built-in function called `model.matrix` that creates dummy codes. You have to give it a formula like you do in linear models, but here, I give it a `~0+variable` syntax so that it doesn't try to create an intercept. That is, instead of *k-1* dummy variables, it'll create *k*. Try it.

```
model.matrix(~0+province, data=flu)
```

There's another built-in function called `cbind` that binds columns together. This can be dangerous to use if you're not certain that rows are in the same order (there, it's better to use an inner join). But here, we're certain they're in the same order. Try binding the results of that to the original data.

```
cbind(flu, model.matrix(~0+province, data=flu))
```

Finally, turn it into a tibble and select out the original province variable. Once you've run the pipeline, go back and make the assignment back to the `flu` object itself.

```
flu <- cbind(flu, model.matrix(~0+province, data=flu)) %>%
  as_tibble() %>%
  select(-province)
flu
```

```
# A tibble: 134 x 12
   case_id outcome   age  male hospital days_to_hospital days_to_outcome
   <chr>   <chr>   <dbl> <int>    <int>            <dbl>           <dbl>
 1 case_1  Death      58     1        0               NA              13
 2 case_2  Death       7     1        1                4              11
 3 case_3  Death      11     0        1               10              31
 4 case_4  <NA>       18     0        1                8              NA
 5 case_5  Recover    20     0        1               11              57
 6 case_6  Death       9     0        1                7              36
 7 case_7  Death      54     1        1                9              20
 8 case_8  Death      14     1        1               11              20
 9 case_9  <NA>       39     1        1                0              NA
10 case_10 Death      20     1        1                4               6
# i 124 more rows
# i 5 more variables: early_outcome <int>, provinceJiangsu <dbl>,
#   provinceOther <dbl>, provinceShanghai <dbl>, provinceZhejiang <dbl>
```

Optional: Notice how the new variables are provinceJiangsu, provinceOther, provinceShanghai, provinceZhejiang. If we want to strip off the "province" we can do that. There's a built-in command called `gsub` that can help here. Take a look at the help for `?gsub`.

```
# Take a look at the names of the flu dataset.
names(flu)

# Remove "province"
gsub("province", "", names(flu))

# Now make the assignment back to names(flu)
names(flu) <- gsub("province", "", names(flu))

# Take a look
flu
```

10.1.5 Imputation

We have a lot of missing data points throughout. Most of the data mining algorithms we're going to use later can't handle missing data, so observations with any missing data are excluded from the model completely. If we have a large dataset and only a few missing values, it's probably better to exclude them and proceed. But since we've already got a pretty low number of observations, we need to try to impute missing values to maximize our use of the data we have.

There are lots of different imputation approaches. An overly simplistic method is simply a mean or median imputation – you simply plug in the mean value for that column for the missing sample's value. This leaves the mean unchanged (good) but artificially decreases the variance (not good). We're going to use the **mice** package for imputation (Multivariate Imputation by Chained Equations). This package gives you functions that can impute continuous, binary, and ordered/unordered categorical data, imputing each incomplete variable with a separate model. It tries to account for relations in the data and uncertainty about those relationships. The methods are described in the paper.

> Buuren, S., & Groothuis-Oudshoorn, K. (2011). mice: Multivariate imputation by chained equations in R. *Journal of statistical software*, 45(3).

Let's load the mice package, and take a look at our data again.

```
library(mice)
flu
```

```
# A tibble: 134 x 12
   case_id outcome   age  male hospital days_to_hospital days_to_outcome
   <chr>   <chr>   <dbl> <int>    <int>            <dbl>           <dbl>
 1 case_1  Death      58     1        0               NA              13
 2 case_2  Death       7     1        1                4              11
 3 case_3  Death      11     0        1               10              31
 4 case_4  <NA>       18     0        1                8              NA
 5 case_5  Recover    20     0        1               11              57
 6 case_6  Death       9     0        1                7              36
 7 case_7  Death      54     1        1                9              20
 8 case_8  Death      14     1        1               11              20
 9 case_9  <NA>       39     1        1                0              NA
10 case_10 Death      20     1        1                4               6
# i 124 more rows
# i 5 more variables: early_outcome <int>, Jiangsu <dbl>, Other <dbl>,
#   Shanghai <dbl>, Zhejiang <dbl>
```

Eventually we want to predict the outcome, so we don't want to factor that into the imputation. We also don't want to factor in the case ID, because that's just an individual's identifier. So let's create a new dataset selecting out those two variables so we can try to impute everything else.

```
flu %>%
  select(-1, -2)
```

The `mice()` function itself returns a special kind of object called a multiply imputed data set, and from this we can run mice's `complete()` on the thing returned by `mice()` to complete the dataset that was passed to it. Here's what we'll do. We'll take the flu data, select out the first two columns, create the imputation, then complete the original data, assigning that to a new dataset called `fluimp`. First let's set the random number seed generator to some number (use the same as I do if you want identical results).

```
set.seed(42)
fluimp <- flu %>%
  select(-1, -2) %>%
  mice(print=FALSE) %>%
  complete()
fluimp
```

Now, we need to put the data back together again. We do this by selecting the original two columns from the original flu data, and then using `cbind()` like above to mash the two datasets together side by side. Finally, we'll turn it back into a tibble. Once you've run the pipeline and you like the result, assign it back to `fluimp`.

```
# Run the pipeline successfully first before you reassign!
fluimp <- flu %>%
  select(1,2) %>%
  cbind(fluimp) %>%
  as_tibble()
fluimp
```

```
# A tibble: 134 x 12
   case_id outcome   age  male hospital days_to_hospital days_to_outcome
   <chr>   <chr>   <dbl> <int>    <int>            <dbl>           <dbl>
 1 case_1  Death      58     1        0                7              13
 2 case_2  Death       7     1        1                4              11
 3 case_3  Death      11     0        1               10              31
 4 case_4  <NA>       18     0        1                8              38
 5 case_5  Recover    20     0        1               11              57
```

```
 6 case_6  Death       9     0        1                7              36
 7 case_7  Death      54     1        1                9              20
 8 case_8  Death      14     1        1               11              20
 9 case_9  <NA>       39     1        1                0              18
10 case_10 Death      20     1        1                4               6
# i 124 more rows
# i 5 more variables: early_outcome <int>, Jiangsu <dbl>, Other <dbl>,
#   Shanghai <dbl>, Zhejiang <dbl>
```

At this point we're almost ready to do some predictive modeling! If you didn't make it this far and you just want to read in the analysis ready dataset, you can do that too.

```
fluimp <- read_csv("data/h7n9_analysisready.csv")
```

10.1.6 The caret package

We're going to use the **caret** package for building and testing predictive models using a variety of different data mining / ML algorithms. The package was published in JSS in 2008. Max Kuhn's slides from the 2013 useR! conference are also a great resource, as is the caret package vignette, and the detailed e-book documentation.

> Kuhn, M. (2008). Building Predictive Models in R Using the caret Package. *Journal of Statistical Software*, 28(5), 1 - 26. doi: http://dx.doi.org/10.18637/jss.v028.i05

The **caret** package (short for **C**lassification **A**nd **RE**gression **T**raining) is a set of functions that streamline the process for creating and testing a wide variety of predictive models with different resampling approaches, as well as estimating variable importance from developed models. There are many different modeling functions in R spread across many different packages, and they all have different syntax for model training and/or prediction. The **caret** package provides a uniform interface the functions themselves, as well as a way to standardize common tasks (such parameter tuning and variable importance).

The `train` function from caret is used to:

- evaluate, using resampling, the effect of model tuning parameters on performance
- choose the "optimal" model across these parameters
- estimate model performance from a training set

10.1.6.1 Models available in caret

First you have to choose a specific type of model or algorithm. Currently there are **239** different algorithms implemented in caret. Caret provides the interface to the method, but you still need the external package installed. For example, we'll be fitting a Random Forest model, and for that we'll need the randomForest package installed. You can see all the methods that you can deploy by looking at the help for train.

```
library(caret)
?train
```

From here, click on the link to see the available models or models by tag. From here you can search for particular models by name. We're going to fit models using Random Forest, stochastic gradient boosting, k-Nearest Neighbors, Lasso and Elastic-Net Regularized Generalized Linear Models. These require the packages randomForest, gbm, kknn, and glmnet, respectively.

Each of the models may have one or more *tuning parameters* – some value or option you can set to tweak how the algorithm develops. In k-nearest neighbors, we can try different values of k. With random forest, we can set the m_{try} option – the algorithm will select m_{try} number of predictors to attempt a split for classification. Caret attempts to do this using a procedure like this:

```
Define sets of model parameter values to evaluate
for each parameter set do
    for each resampling iteration do
        Hold–out specific samples
        [Optional] Pre–process the data
        Fit the model on the remainder
        Predict the hold–out samples
    end
    Calculate the average performance across hold–out predictions
end
Determine the optimal parameter set
Fit the final model to all the training data using the optimal parameter set
```

Figure 10.1: *The caret model training algorithm. Image from the caret paper.*

That is, it sweeps through each possible parameter you can set for the particular type of model you choose, and uses some kind of resampling scheme with your training data, fitting the model on a subset and testing on the held-out samples.

10.1.6.2 Resampling

The default resampling scheme caret uses is the bootstrap. Bootstrapping takes a random sample *with replacement* from your data that's the same size of the original data. Samples might be selected more than once, and some aren't selected at all. On average, each sample has a ~63.2% chance of showing up at least once in a bootstrap sample. Some samples won't show up at all, and these *held out* samples are the ones that are used for testing the performance of the trained model. You repeat this process many times (e.g., 25, 100, etc) to get an average performance estimate on unseen data. Here's what it looks like in practice.

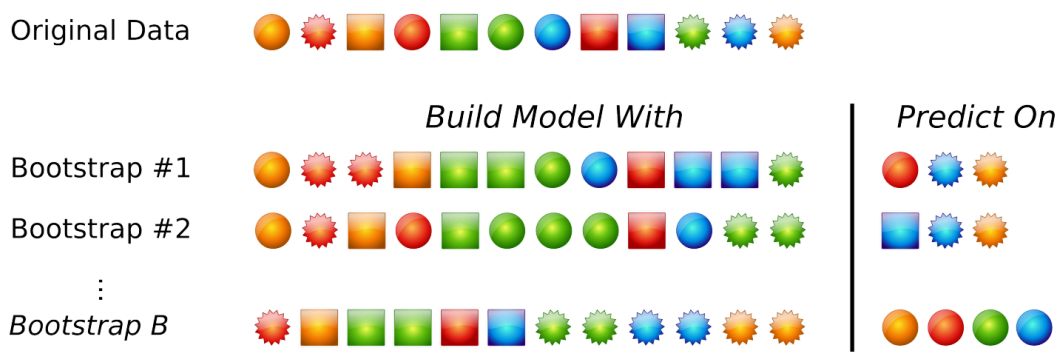


Figure 10.2: *Bootstrapping schematic. Image from Max Kuhn's 2013 useR! talk.*

Many alternatives exist. Another popular approach is cross-validation. Here, a subset of your data (e.g., 4/5ths, or 80%) is used for training, and the remaining 1/5th or 20% is used for performance assessment. You slide the cross-validation interval over and use the next 4/5ths for training and 1/5th for testing. You do this again for all 5ths of the data. You can optionally repeat this process many times (*repeated cross-validation*) to get an average cross validation prediction accuracy for a given model and set of tuning parameters.

The `trainControl` option in the `train` function controls this, and you can learn more about this under the Basic Parameter Tuning section of the caret documentation.

10.1.7 Model training

Let's try it out! If you didn't make it through the data preprocessing steps and you just want to read in the analysis ready dataset, you can do this:

```
fluimp <- read_csv("data/h7n9_analysisready.csv")
```

10.1.7.1 Splitting data into known and unknown outcomes

Before we continue, let's split the dataset into samples where we know the outcome, and those where we don't. The unknown samples will be the ones where `is.na(outcome)` is TRUE. So you can use a filter statement.

```
# Run the pipeline successfully first before you reassign!
# These are samples with unknown data we'll use later to predict
unknown <- fluimp %>%
  filter(is.na(outcome))
unknown
```

The known samples are the cases where `!is.na(outcome)` is TRUE, that is, cases where the outcome is not (`!`) missing. One thing we want to do here while we're at it is remove the case ID. This is just an arbitrary numerically incrementing counter and we *don't* want to use this in building a model!

```
# Run the pipeline successfully first before you reassign!
# Samples with known outcomes used for model training.
known <- fluimp %>%
  filter(!is.na(outcome)) %>%
  select(-case_id)
known
```

10.1.7.2 A note on reproducibility and `set.seed()`

When we train a model using resampling, that sampling is going to happen *pseudo*-randomly. Try running this function which generates five numbers from a random uniform distribution between 0 and 1.

```
runif(5)
```

If you run that function over and over again, you'll get different results. But, we can set the random number seed generator with any value we choose, and we'll get the same result. Try setting the seed, drawing the random numbers, then re-setting the same seed, and re-running the `runif` function again. You should get identical results.

```
set.seed(22908)
runif(5)
```

Eventually I'm going to compare different models to each other, so I want to set the random number seed generator to the same value for each model so the same random bootstrap samples are identical across models.

10.1.7.3 Random Forest

Let's fit a random forest model. See the help for `?train` and click on the link therein to see what abbreviations correspond to which model. First set the random number seed generator to some number, e.g., 8382, that we'll use for all other models we make. The model forumula here takes the know data, and the `responseVar~.` syntax says "predict `responseVar` using every other variable in the data." Finally, notice how when we call `train()` from the **caret** package using "rf" as the type of model, it automatically loads the **randomForest** package that you installed. If you didn't have it installed, it would probably ask you to install it first.

```
# Set the random number seed generator
set.seed(8382)

# Fit a random forest model for outcome against everything in the model (~.)
modrf <- train(outcome~., data=known, method="rf")

# Take a look at the output
modrf
```

```
Random Forest

77 samples
10 predictors
 2 classes: 'Death', 'Recover'

No pre-processing
Resampling: Bootstrapped (25 reps)
Summary of sample sizes: 77, 77, 77, 77, 77, 77, ...
Resampling results across tuning parameters:

  mtry  Accuracy  Kappa
   2    0.688     0.328
   6    0.684     0.322
  10    0.693     0.345

Accuracy was used to select the optimal model using the largest value.
The final value used for the model was mtry = 10.
```

Take a look at what that tells us. It tells us it's fitting a Random Forest model using 77 samples, predicting a categorical outcome class (Death or Recover) based on 10 predictors. It's not doing any pre-processing like centering or scaling, and it's doing bootstrap resampling of 77 samples with replacement, repeated 25 times each. Random Forest has a single tuning parameter, m_{try} – the algorithm will select m_{try} number of predictors to attempt a split for classification when building a classification tree. The caret package does 25 bootstrap resamples for different values of m_{try} (you can also control this too if you want), and computes accuracy and kappa measures of performance on the held-out samples.

Accuracy is the number of true assignments to the correct class divided by the total number of samples. Kappa takes into account the expected accuracy while considering chance agreement, and is useful for extremely imbalanced class distributions. For continuous outcomes, you can measure things like RMSE or correlation coefficients.

> ***A bit about random forests.*** Random forests are an ensemble learning approach based on classification trees. The CART (classification and regression tree) method searches through all available predictors to try to find a value of a single variable that splits the data into two groups by minimizing the impurity of the outcome between the two groups. The process is repeated over and over again until a hierarchical (tree) structure is created. But trees don't have great performance (prediction accuracy) compared to other models. Small changes in the data can drastically affect the structure of the tree.
>
> Tree algorithms are improved by ensemble approaches - instead of growing a single tree, grow many trees and aggregate (majority vote or averaging) the predictions made by the ensemble. The random forest algorithm is essentially:
>
> 1. From the training data of n samples, draw a bootstrap sample of size n.
> 2. For each bootstrap sample, grow a classification tree, but with a small modification compared to the traditional algorithm: instead of selecting from all possible predictor variables to form a split, choose the best split among a randomly selected subset of m_{try} predictors. Here, m_{try} is the only tuning parameter. The trees are grown to their maximum size and not "pruned" back.
> 3. Repeat the steps agove until a large number of trees is grown.
> 4. Estimate the performance of the ensemble of trees using the "out-of-bag" samples - i.e., those that were never selected during the bootstrap procedure in step #1.
> 5. Estimate the importance of each variable in the model by randomly permuting each predictor variable in testing on the out-of-bag samples. If a predictor is important, prediction accuracy will degrade. If the predictor isn't that helpful, performance doesn't deteriorate as much.
>
> Random forests are efficient compared to growing a single tree. For one, the RF algorithm only selects from m_{try} predictors at each step, rather than all available

predictors. Usually m_{try} is by default somewhere close to the square root of the total number of available predictors, so the search is very fast. Second, while the traditional CART tree algorithm has to go through extensive cross-validation based pruning to avoid overfitting, the RF algorithm doesn't do any pruning at all. In fact, building an RF model *can* be faster than building a single tree!

Caret also provides a function for assessing the importance of each variable. The `varImp` function knows what kind of model was fitted and knows how to estimate variable importance. For Random Forest, it's an estimate of how much worse the prediction gets after randomly shuffling the values of each predictor variable in turn. A variable that's important will result in a much worse prediction than a variable that's not as meaningful.

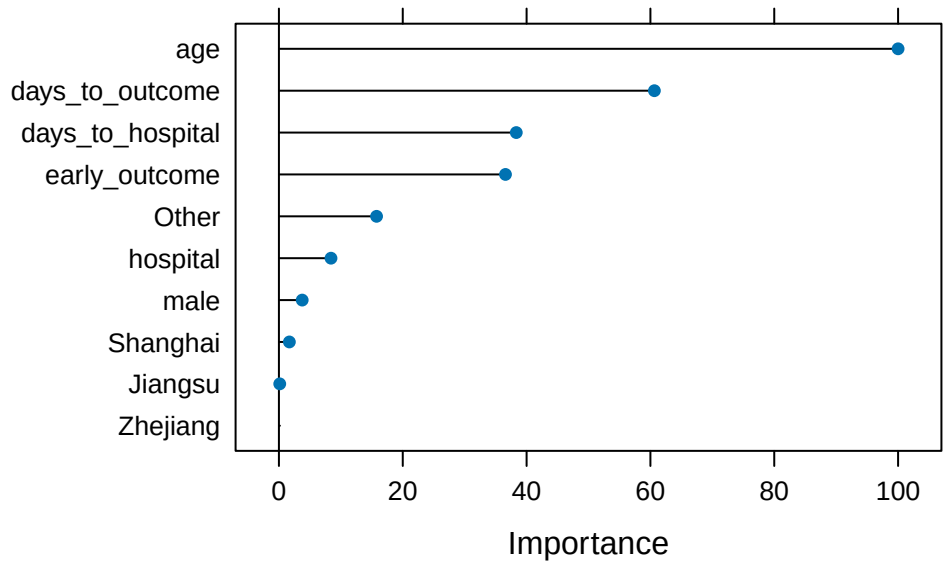
```
varImp(modrf, scale=TRUE)
```

```
rf variable importance
```

	Overall
age	100.000
days_to_outcome	60.642
days_to_hospital	38.333
early_outcome	36.591
Other	15.772
hospital	8.410
male	3.758
Shanghai	1.687
Jiangsu	0.133
Zhejiang	0.000

You can also pass that whole thing to `plot()`, or wrap the statement in `plot()`, to see a graphical representation.

```
varImp(modrf, scale=TRUE) %>% plot()
```



10.1.7.4 Stochastic Gradient Boosting

Let's try a different method, stochastic gradient boosting, which uses a different method for building an ensemble of classification trees (see this post for a discussion of bagging vs boosting). This requires the **gbm** package. Again, set the random seed generator.

```
set.seed(8382)
modgbm <- train(outcome~., data=known, method="gbm", verbose=FALSE)
modgbm
```

```
Stochastic Gradient Boosting

77 samples
10 predictors
 2 classes: 'Death', 'Recover'

No pre-processing
Resampling: Bootstrapped (25 reps)
Summary of sample sizes: 77, 77, 77, 77, 77, 77, ...
Resampling results across tuning parameters:

  interaction.depth  n.trees  Accuracy  Kappa
  1                   50      0.630     0.210
  1                  100      0.627     0.210
  1                  150      0.630     0.213
```

```
  2                   50      0.633     0.222
  2                  100      0.636     0.218
  2                  150      0.632     0.208
  3                   50      0.616     0.188
  3                  100      0.639     0.227
  3                  150      0.636     0.218

Tuning parameter 'shrinkage' was held constant at a value of 0.1

Tuning parameter 'n.minobsinnode' was held constant at a value of 10
Accuracy was used to select the optimal model using the largest value.
The final values used for the model were n.trees = 100, interaction.depth =
 3, shrinkage = 0.1 and n.minobsinnode = 10.
```

Notice how stochastic gradient boosting has two different tuning parameters - interaction depth and n trees. There were others (shrinkage, and `n.minobsinnode`) that were held constant. The caret package automates the bootstrap resampling based performance assessment across all combinations of depth and ntrees, and it tells you where you got the best performance. Notice that the performance here doesn't seem to be as good as random forest. We can also look at variable importance here too, and see similar rankings.

```
library(gbm) # needed because new version of caret doesn't load
varImp(modgbm, scale=TRUE)
varImp(modgbm, scale=TRUE) %>% plot()
```

10.1.7.5 Model comparison: Random Forest vs Gradient Boosting

Let's compare those two models. Because the random seed was set to the same number (8382), the bootstrap resamples were identical across each model. Let's directly compare the results for the best models from each method.

```
modsum <- resamples(list(gbm=modgbm, rf=modrf))
summary(modsum)
```

```
Call:
summary.resamples(object = modsum)

Models: gbm, rf
Number of resamples: 25
```

```
Accuracy
     Min. 1st Qu. Median  Mean 3rd Qu.  Max. NA's
gbm 0.483   0.577  0.625 0.639   0.692 0.812    0
rf  0.552   0.654  0.692 0.693   0.731 0.864    0

Kappa
       Min. 1st Qu. Median  Mean 3rd Qu.  Max. NA's
gbm -0.1600   0.103  0.250 0.227   0.319 0.591    0
rf  -0.0162   0.255  0.366 0.345   0.421 0.697    0
```

It appears that random forest is doing much better in terms of both accuracy and kappa. Let's train a few other types of models.

10.1.7.6 Elastic net regularized logistic regression

Elastic net regularization is a method that combines both the lasso and ridge methods of reguarizing a model. Regularization is a method for *penalizing* a model as it gains complexity with more predictors in an attempt to avoid overfitting. You'll need the **glmnet** package for this.

```
set.seed(8382)
modglmnet <- train(outcome~., data=known, method="glmnet")
modglmnet
```

```
glmnet

77 samples
10 predictors
 2 classes: 'Death', 'Recover'

No pre-processing
Resampling: Bootstrapped (25 reps)
Summary of sample sizes: 77, 77, 77, 77, 77, 77, ...
Resampling results across tuning parameters:

  alpha  lambda    Accuracy  Kappa
  0.10   0.000391  0.635     0.226
  0.10   0.003908  0.634     0.226
  0.10   0.039077  0.630     0.217
  0.55   0.000391  0.635     0.226
```

```
  0.55   0.003908  0.633     0.223
  0.55   0.039077  0.633     0.226
  1.00   0.000391  0.635     0.226
  1.00   0.003908  0.630     0.215
  1.00   0.039077  0.643     0.243

Accuracy was used to select the optimal model using the largest value.
The final values used for the model were alpha = 1 and lambda = 0.0391.
```

10.1.7.7 k-nearest neighbor

k-nearest neighbor attempts to assign samples to their closest labeled neighbors in high-dimensional space. You'll need the **kknn** package for this.

```
set.seed(8382)
modknn <- train(outcome~., data=known, method="kknn")
modknn
```

```
k-Nearest Neighbors

77 samples
10 predictors
 2 classes: 'Death', 'Recover'

No pre-processing
Resampling: Bootstrapped (25 reps)
Summary of sample sizes: 77, 77, 77, 77, 77, 77, ...
Resampling results across tuning parameters:

  kmax  Accuracy  Kappa
  5     0.635     0.218
  7     0.635     0.218
  9     0.633     0.214

Tuning parameter 'distance' was held constant at a value of 2
Tuning
 parameter 'kernel' was held constant at a value of optimal
Accuracy was used to select the optimal model using the largest value.
The final values used for the model were kmax = 7, distance = 2 and kernel
 = optimal.
```

10.1.7.8 Compare all the models

Now let's look at the performance characteristics for the best performing model across all four types of models we produced. It still looks like random forest is coming through as the winner.

```
modsum <- resamples(list(gbm=modgbm, rf=modrf, glmnet=modglmnet, knn=modknn))
summary(modsum)
```

```
Call:
summary.resamples(object = modsum)

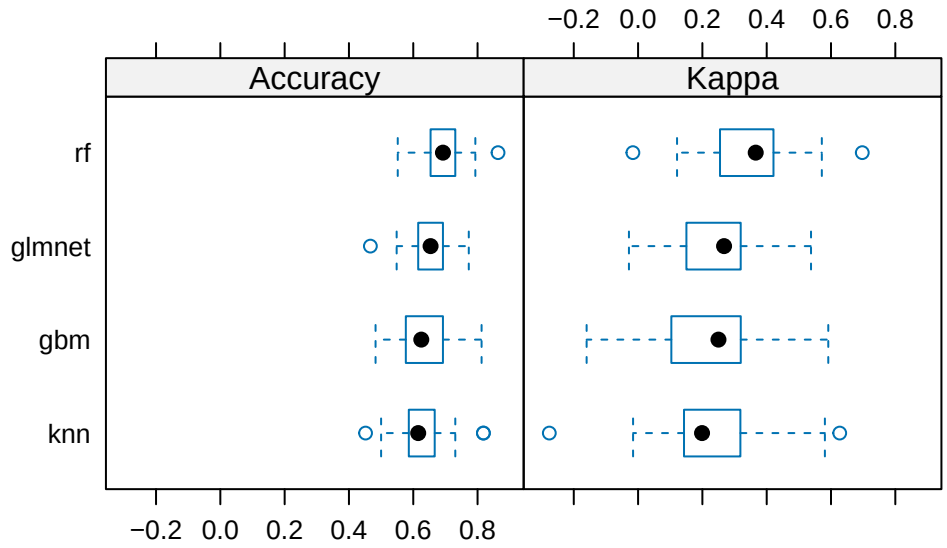
Models: gbm, rf, glmnet, knn
Number of resamples: 25

Accuracy
        Min. 1st Qu. Median  Mean 3rd Qu.  Max. NA's
gbm    0.483   0.577  0.625 0.639   0.692 0.812    0
rf     0.552   0.654  0.692 0.693   0.731 0.864    0
glmnet 0.467   0.615  0.654 0.643   0.692 0.773    0
knn    0.452   0.586  0.615 0.635   0.667 0.818    0

Kappa
          Min. 1st Qu. Median  Mean 3rd Qu.  Max. NA's
gbm    -0.1600   0.103  0.250 0.227   0.319 0.591    0
rf     -0.0162   0.255  0.366 0.345   0.421 0.697    0
glmnet -0.0284   0.150  0.267 0.243   0.319 0.538    0
knn    -0.2760   0.143  0.199 0.218   0.318 0.627    0
```

The `bwplot()` function can take this model summary object and visualize it.

```
bwplot(modsum)
```



10.1.8 Prediction on unknown samples

Once we have a model trained it's fairly simple to predict the class of the unknown samples. Take a look at the unknown data again:

```
unknown
```

Now, since Random Forest worked best, let's use that model to predict the outcome!

```
predict(modrf, newdata=unknown)
```

```
 [1] Recover Recover Death   Recover Death   Death   Recover Recover Death
[10] Recover Death   Recover Recover Recover Recover Death   Recover Death
[19] Death   Death   Recover Recover Recover Recover Recover Recover Recover
[28] Recover Death   Death   Recover Recover Recover Recover Recover Recover
[37] Recover Recover Recover Recover Recover Recover Recover Death   Recover
[46] Death   Recover Death   Recover Recover Recover Recover Recover Recover
[55] Recover Recover Recover
Levels: Death Recover
```

This gives you a vector of values that would be the outcome for the individuals in the `unknown` dataset. From here it's pretty simple to put them back in the data with a `mutate()`.

```
unknown %>%
  mutate(outcome=predict(modrf, newdata=unknown))
```

```
# A tibble: 57 x 12
   case_id outcome   age  male hospital days_to_hospital days_to_outcome
   <chr>   <fct>   <dbl> <dbl>    <dbl>            <dbl>           <dbl>
 1 case_4  Recover    18     0        1                8              46
 2 case_9  Recover    39     1        1                0              18
 3 case_15 Death      34     0        0               11              38
 4 case_16 Recover    51     1        0                3              20
 5 case_22 Death      56     1        1                4              17
 6 case_28 Death      51     1        0                6               6
 7 case_31 Recover    43     1        0                4              21
 8 case_32 Recover    46     1        0                3              20
 9 case_38 Death      28     1        0                2               7
10 case_39 Recover    38     1        1                0              18
# i 47 more rows
# i 5 more variables: early_outcome <dbl>, Jiangsu <dbl>, Other <dbl>,
#   Shanghai <dbl>, Zhejiang <dbl>
```

Alternatively, you could pass in `type="prob"` to get prediction probabilities instead of predicted classes.

```
predict(modrf, newdata=unknown, type="prob") %>% head()
```

```
  Death Recover
1 0.040   0.960
2 0.030   0.970
3 0.564   0.436
4 0.138   0.862
5 0.774   0.226
6 0.972   0.028
```

You could also imagine going further to get the prediction probabilities out of each type of model we made. You could add up the prediction probabilities for Death and Recovery for each individual across model types, and then compute a ratio. If across all the models that ratio is, for example, 2x in favor of death, you could predict death, or if it's 2x in favor of recovery, you predict recover, and if it's in between, you might call it "uncertain." This lets you not only reap the advantages of ensemble learning within a single algorithm, but also lets you use information across a variety of different algorithm types.

10.2 Forecasting

10.2.1 The Prophet Package

Forecasting is a common data science task that helps with things like capacity planning, goal setting, anomaly detection, and resource use projection. Forecasting can involve complex models, where overly simplistic models can be brittle and can be too inflexible to incorporate useful assumptions about the underlying data.

Recently, the data science team at Facebook released as open-source a tool they developed for forecasting, called **prophet**, as both an R and python package.

- Paper (preprint): https://peerj.com/preprints/3190/
- Project homepage: https://facebook.github.io/prophet/
- Documentation: https://facebook.github.io/prophet/docs/quick_start.html
- R package: https://cran.r-project.org/web/packages/prophet/index.html
- Python package: https://pypi.python.org/pypi/fbprophet/
- Source code: https://github.com/facebook/prophet

  Google and Twitter have released as open-source similar packages: Google's **CausalImpact** software (https://google.github.io/CausalImpact/) assists with inferring causal effects of a design intervention on a time series, and Twitter's **AnomalyDetection** package (https://github.com/twitter/AnomalyDetection) was designed to detect blips and anomalies in time series data given the presence of seasonality and underlying trends. See also Rob Hyndman's **forecast** package in R.

Prophet is optimized for forecasting problems that have the following characteristics:

- Hourly, daily, or weekly observations with at least a few months (preferably a year) of history
- Strong multiple "human-scale" seasonalities: day of week and time of year
- Important holidays that occur at irregular intervals that are known in advance (e.g. the Super Bowl)
- A reasonable number of missing observations or large outliers
- Historical trend changes, for instance due to product launches or logging changes
- Trends that are non-linear growth curves, where a trend hits a natural limit or saturates

These use cases are optimized for business forecasting problems encountered at Facebook, but many of the characteristics here apply well to other kinds of forecasting problems. Further, while the default settings can produce fairly high-quality forecasts, if the results aren't satisfactory, you aren't stuck with a completely automated model you can't change. The prophet package allows you to tweak forecasts using different parameters. The process is summarized in the figure below.

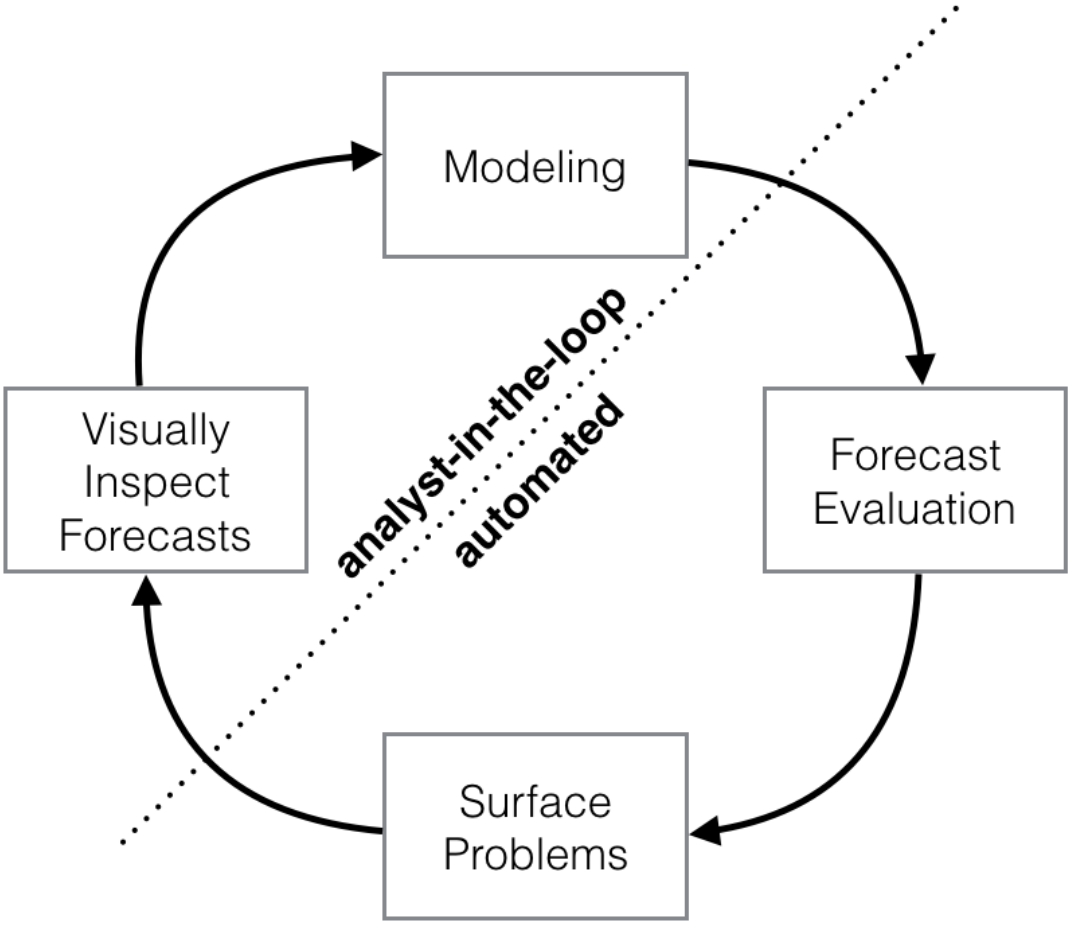


Figure 10.3: *Schematic view of the analyst-in-the-loop approach to forecasting at scale, which best makes use of human and automated tasks. Image from the Prophet preprint noted above.*

Prophet is a good replacement for the **forecast** package because:

1. **Prophet makes it easy.** The forecast package offers many different techniques, each with their own strengths, weaknesses, and tuning parameters. While the choice of parameter settings and model specification gives the expert user great flexibility, the downside is that choosing the wrong parameters as a non-expert can give you poor results. Prophet's defaults work pretty well.
2. **Prophet's forecasts are intuitively customizable.** You can choose smoothing parameters for seasonality that adjust how closely you fit historical cycles, and you can adjust how agressively to follow historical trend changes. You can manually specify the upper limit on growth curves, which allows for you to supplement the automatic forecast with your own prior information about how your forecast will grow (or decline). You can also specify irregular events or time points (e.g., election day, the Super Bowl, holiday travel times, etc) that can result in outlying data points.

The prophet procedure is essentially a regression model with some additional components:

1. A piecewise linear or logistic growth curve trend. Prophet automatically detects changes in trends by selecting changepoints from the data.
2. A yearly seasonal component modeled using Fourier series.
3. A weekly seasonal component using dummy variables.
4. A user-provided list of important holidays.

See the prophet preprint for more.

> Taylor SJ, Letham B. (2017) Forecasting at scale. *PeerJ Preprints* 5:e3190v2 https://doi.org/10.7287/peerj.preprints.3190v2

10.2.2 CDC ILI time series data

Here we're going to use historical flu tracking data from the CDC's U.S. Outpatient Influenza-like Illness Surveillance Network along with data from the National Center for Health Statistics (NCHS) Mortality Surveillance System. This contains ILI totals from CDC and flu + pneumonia death data from NCHS through the end of October 2017. It's the **ilinet.csv** file. Let's read it in, then take a look. Notice that `week_start` was automatically read in as a date data type. What you see as `2003-01-06` is actually represented internally as a date, not a character.

```
# Read in the ILI data.
ili <- read_csv("data/ilinet.csv")
ili
```

```
# A tibble: 818 x 6
   week_start ilitotal total_patients fludeaths pneumoniadeaths all_deaths
   <date>        <dbl>          <dbl>     <dbl>            <dbl>      <dbl>
 1 2003-01-06     3260         171193        NA               NA         NA
 2 2003-01-13     3729         234513        NA               NA         NA
 3 2003-01-20     4204         231550        NA               NA         NA
 4 2003-01-27     5696         235566        NA               NA         NA
 5 2003-02-03     7079         246969        NA               NA         NA
 6 2003-02-10     7782         245751        NA               NA         NA
 7 2003-02-17     7649         253656        NA               NA         NA
 8 2003-02-24     7228         241110        NA               NA         NA
 9 2003-03-03     5606         241683        NA               NA         NA
10 2003-03-10     4450         228549        NA               NA         NA
# i 808 more rows
```

We have information on ILI frequency since January 2003, but we don't have information on death data until 2009. From here, we have data up through the end of September 2018.

```
tail(ili)
```

```
# A tibble: 6 x 6
  week_start ilitotal total_patients fludeaths pneumoniadeaths all_deaths
  <date>        <dbl>          <dbl>     <dbl>            <dbl>      <dbl>
1 2018-08-20     6519         798422         9             2426      46033
2 2018-08-27     7257         762601         5             2321      45679
3 2018-09-03     8049         823571         4             2430      44689
4 2018-09-10     9457         821290         7             2329      44279
5 2018-09-17     9966         858050         7             2239      41875
6 2018-09-24    11057         832495         6             1896      35305
```

10.2.3 Forecasting with prophet

Let's load the prophet library then take a look at the help for `?prophet`.

```
library(prophet)
# ?prophet
```

The help tells you that prophet requires a data frame containing columns named `ds` of type date and `y`, containing the time series data. Many other options are available. Let's start with the data, select `week_start` calling it `ds`, and `ilitotal` calling it `y`.

```
ili %>%
  select(week_start, ilitotal)

ili %>%
  select(ds=week_start, y=ilitotal)
```

Once we do that, we can simply pipe this to `prophet()` to produce the prophet forecast model.

```
pmod <- ili %>%
  select(ds=week_start, y=ilitotal) %>%
  prophet()
```

Now, let's make a "future" dataset to use to predict. Looking at `?make_future_dataframe` will tell you that this function takes the prophet model and the number of days forward to project.

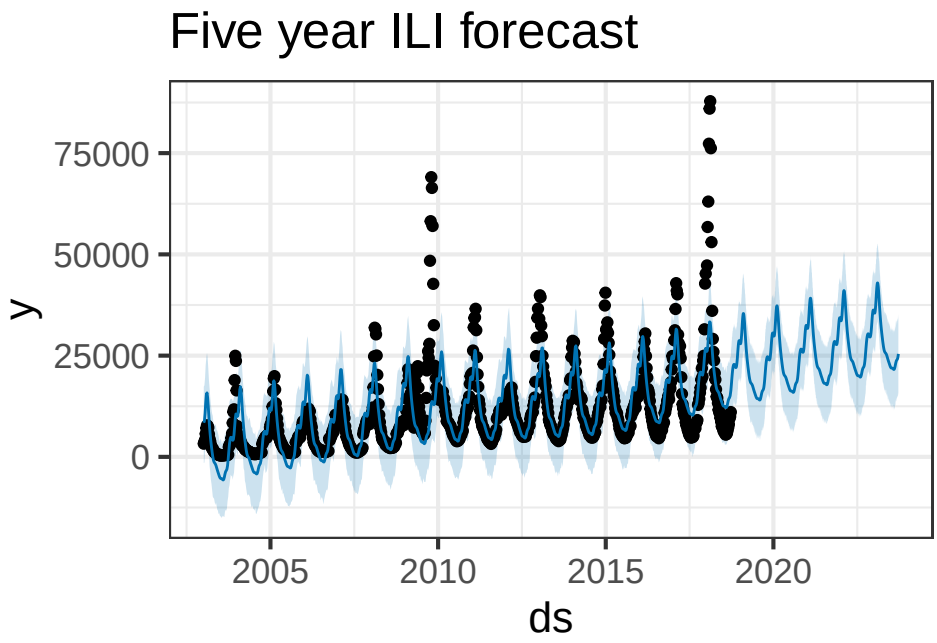
```
future <- make_future_dataframe(pmod, periods=365*5)
tail(future)
```

Now, let's forecast the future! Take a look - the `yhat`, `yhat_lower`, and `yhat_upper` columns are the predictions, lower, and upper confidence bounds. There are additional columns for seasonal and yearly trend effects.

```
forecast <- predict(pmod, future)
tail(forecast)
```

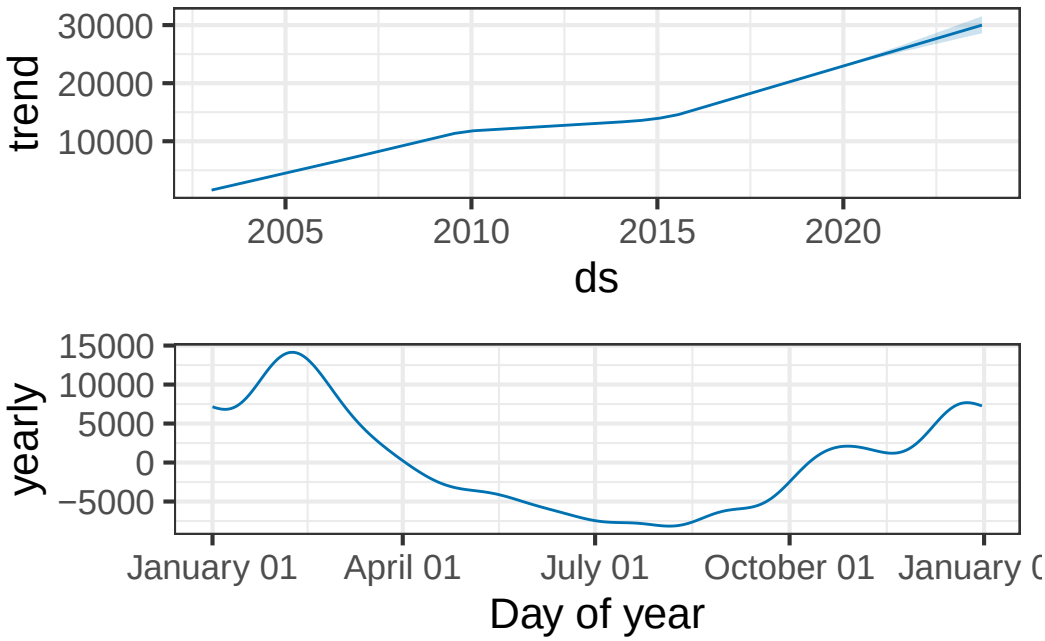
If we pass the prophet model and the forecast into the generic `plot()` function, it knows what kind of objects are being passed, and will visualize the data appropriately.

```
plot(pmod, forecast) + ggtitle("Five year ILI forecast")
```



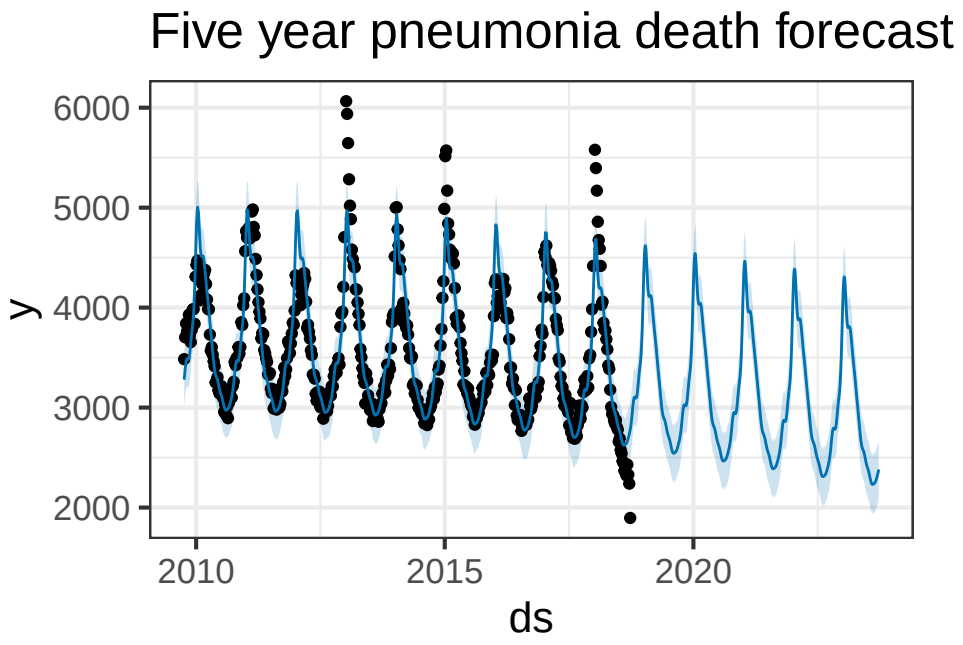
You can also use the `prophet_plot_components` function to see the forecast broken down into trend and yearly seasonality. We see an inflection point around 2010 where ILI reports seem to stop rising – if you go back to the previous plot you'll see it there too. Perhaps this is due to a change in surveillance or reporting strategy. You also see the yearly trend, which makes sense for flu outbreaks. You also noticed that when we originally fit the model, daily and weekly seasonality was disabled. This makes sense for broad time-scale things like influenza surveillance over decades, but you might enable it for more granular time-series data.

```
prophet_plot_components(pmod, forecast)
```



Try it with the flu death data. Look at both flu deaths and pneumonia deaths. First, limit the data frame to only include the latter portion where we have death surveillance data. Then use the same procedure.

```
pmod <- ili %>%
  filter(!is.na(pneumoniadeaths)) %>%
  select(ds=week_start, y=pneumoniadeaths) %>%
  prophet()
future <- make_future_dataframe(pmod, periods=365*5)
forecast <- predict(pmod, future)
plot(pmod, forecast) + ggtitle("Five year pneumonia death forecast")
```



See the prophet preprint for more.

Taylor SJ, Letham B. (2017) Forecasting at scale. *PeerJ Preprints* 5:e3190v2 https://doi.org/10.7287/peerj.preprints.3190v2

11 Text Mining and NLP

11.1 Chapter overview

Most of the data we've dealt with so far in this course has been rectangular, in the form of a data frame or tibble, and mostly numeric. But lots of data these days comes in the form of unstructured text. This workshop provides an overview of fundamental principles in text mining, and introduces the **tidytext** package that allows you to apply to text data the same "tidy" methods you're familiar with for wrangling and vizualizing data.[1]

This course is *not* an extensive deep dive into natural language processing (NLP). For that check out the CRAN task view on NLP for a long list of packages that will aid you in computational linguistics.

Before we get started, let's load the packages we'll need.

```
library(tidyverse)
library(tidytext)
library(gutenbergr)
library(topicmodels)
```

11.2 The Tidy Text Format

In the previous chapters linked above we discussed the three features of Tidy Data, as outlined in Hadley Wickham's Tidy Data paper:

- Each variable is a column
- Each observation is a row
- Each type of observational unit is a table

Tidy text format can be defined as **a table with one-token-per-row.** A **token** is any meaningful unit of text, such as a word, that we are interested in using for analysis. **Tokenization** is the process of splitting text into tokens. This is in contrast to storing text in

[1] ***Attribution*:** This workshop was inspired by and/or modified in part from *Text Mining with R* by Julia Silge and David Robinson.

strings or in a document-term matrix (discussed later). Here, the token stored in a single row is most often a single word. The **tidytext** package provides functionality to tokenize strings by words (or n-grams, or sentences) and convert to a one-term-per-row format. By keeping text in "tidy" tables, you can use the normal tools you're familiar with, including dplyr, tidyr, ggplot2, etc., for manipulation, analysis, and visualization. The tidytext package also includes functions to convert to and from other data structures for text processing, such as a *corpus*[2] or a *document-term matrix*.[3]

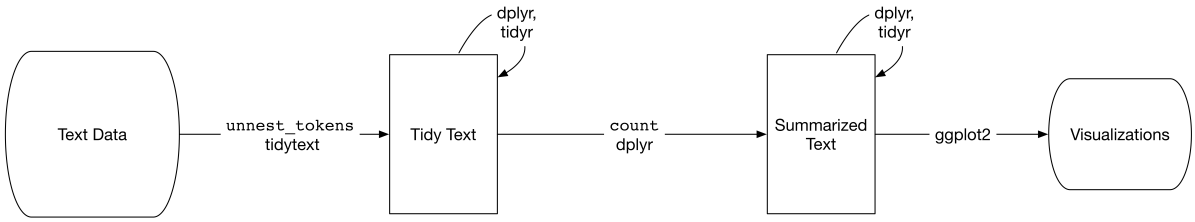


Figure 11.1: Workflow for text analysis using tidy principles.

11.2.1 The `unnest_tokens` function

We briefly mentioned before how to create vectors using the `c()` function. Let's create a simple character vector.

```
text <- c("a", "banana", "crouton")
```

Let's extend that to create another character vector, this time with sentences:

```
text <- c("It was the best of times,",
          "it was the worse of times,",
          "It was the spring of hope, it was the winter of despair.")
text
```

Before we can turn this into a tidy text dataset, we first have to put it in a data frame.

```
text_df <- tibble(line = 1:3, text = text)
text_df
```

```
# A tibble: 3 x 2
```

[2]Corpus objects contain strings annotated with additional metadata.

[3]This is a (sparse) matrix describing a collection (corpus) of documents with one row for each document and one column for each term. The value in the matrix is typically word count or tf-idf for the document in that row for the term in that column.

```
   line text
  <int> <chr>
1     1 It was the best of times,
2     2 it was the worse of times,
3     3 It was the spring of hope, it was the winter of despair.
```

This data isn't yet "tidy." We can't do the kinds of operations like filter out particular words or summarize operations, for instance, to count which occur most frequently, since each row is made up of multiple combined words. We need to convert this so that it has **one-token-per-document-per-row**. Here we only have a single document, but later we'll have multiple documents.

We need to (1) break the text into individual tokens (i.e. *tokenization*) and transform it to a tidy data structure. To do this, we use tidytext's `unnest_tokens()` function.

```
text_df |>
  unnest_tokens(output=word, input=text)
```

```
# A tibble: 24 x 2
    line word
   <int> <chr>
 1     1 it
 2     1 was
 3     1 the
 4     1 best
 5     1 of
 6     1 times
 7     2 it
 8     2 was
 9     2 the
10     2 worse
# i 14 more rows
```

The `unnest_tokens` function takes a data frame (or tibble), and two additional parameters, the `output` and `input` column names. If you specify them in the correct order, you don't have to specify `output=` or `input=`. You can pipe to `print(n=Inf)` to print them all.

```
text_df |>
  unnest_tokens(word, text) |>
  print(n=Inf)
```

First you give it the output column name that will be created as the text is unnested into it (`word`, in this example). This is a column name that you choose – you could call it anything, but `word` usually makes sense. Then you give it the input column that the text comes from in the data frame you're passing to it (`text`, in this case). Our `text_df` dataset has a column called `text` that contains the data of interest.

The `unnest_tokens` function splits each row so that there is one word per row of the new data frame; the default tokenization in `unnest_tokens()` is for single words, as shown here. Also notice:

- Other columns, such as the line number each word came from, are retained.
- Punctuation has been stripped.
- By default, `unnest_tokens()` converts the tokens to lowercase, which makes them easier to compare or combine with other datasets. (Use the `to_lower = FALSE` argument to turn off this behavior).

Now our data is in a tidy format, and we can easily use all the normal dplyr, tidyr, and ggplot2 tools.

11.2.2 Example: Jane Austen Novels

Let's load the **austen.csv** data.

```
jaorig <- read_csv("data/austen.csv")
jaorig
```

Click the `jaorig` dataset in the environment pane or use `View(jaorig)` to see what's being read in here. Before we can do anything else we'll need to tidy this up by unnesting the `text` column into words.

```
jatidy <- jaorig |>
  unnest_tokens(word, text)
jatidy
```

Let's use the dplyr `count` function to count how many occurances we have for each word in the entire corpus. The `sort=TRUE` option puts the most common results on top.

```
jatidy |>
  count(word, sort = TRUE)
```

Not surprisingly the most common words are some of the most commonly used words in the English language. These are known as stop words. They're words you'll want to filter out before doing any text mining. There are lists of stop words online, but the tidytext package

comes with a `stop_words` built-in dataset with some of the most common stop words across three different lexicons. See `?stop_words` for more information.

```
data(stop_words)
stop_words
```

As in a previous chapter where we did an `inner_join` to link information across two different tables by a common key, there's also an `anti_join()` which takes two tibbles, x and y, and returns all rows from x where there are not matching values in y, keeping just columns from x. Let's `anti_join` the data to the stop words. Because we chose "word" as the output variable to `unnest_tokens()`, and "word" is the variable in the `stop_words` dataset, we don't have to be specific about which columns we're joining.

```
jatidy |>
  anti_join(stop_words)
```

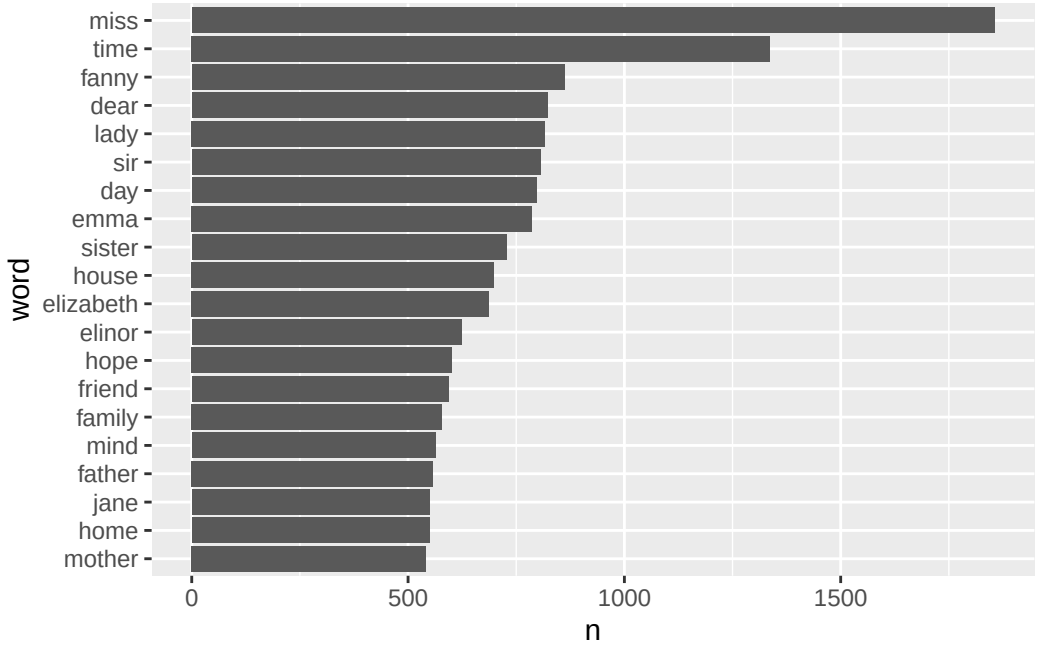
Now there are *far* fewer rows than initially present. Let's run that count again, now with the stop words removed.

```
jatidy |>
  anti_join(stop_words) |>
  count(word, sort = TRUE)
```

```
# A tibble: 13,914 x 2
   word       n
   <chr>  <int>
 1 miss    1855
 2 time    1337
 3 fanny    862
 4 dear     822
 5 lady     817
 6 sir      806
 7 day      797
 8 emma     787
 9 sister   727
10 house    699
# i 13,904 more rows
```

That's *much* more in line with what we want. We have this data in a tibble. Let's keep piping to other operations!

```
jatidy |>
  anti_join(stop_words) |>
  count(word, sort = TRUE) |>
  head(20) |>
  mutate(word = reorder(word, n)) |>
  ggplot(aes(word, n)) +
  geom_col() +
  coord_flip()
```



11.3 Sentiment Analysis

Let's start to do some high-level analysis of the text we have. Sentiment analysis[4], also called opinion mining, is the use of text mining to "systematically identify, extract, quantify, and study affective states and subjective information." It's a way to try to understand the emotional intent of words to infer whether a section of text is positive or negative, or perhaps characterized by some other more nuanced emotion like surprise or disgust.

If you make a simplifying assumption regarding the text you have as a combination of its individual words, you can treat the sentiment content of the whole text as the sum of the sentiment content of the individual words. It's a simplification, and it isn't the only way to approach sentiment analysis, but it's simple and easy to do with tidy principles.

[4] https://en.wikipedia.org/wiki/Sentiment_analysis

To get started you'll need a *sentiment lexicon* that attempt to evaluate the opinion or emotion in text. The tidytext package contains several sentiment lexicons in the `sentiments` dataset. All three of these lexicons are based on single words in the English language, assigning scores for positive/negative sentiment, or assigning emotions like joy, anger, sadness, etc.

- **`nrc`** from Saif Mohammad and Peter Turney[5] categorizes words in a binary fashion ("yes"/"no") into categories of positive, negative, anger, anticipation, disgust, fear, joy, sadness, surprise, and trust.
- **`bing`** from Bing Liu and collaborators[6] categorizes words in a binary fashion into positive and negative categories.
- **`AFINN`** from Finn Arup Nielsen[7] assigns words with a score that runs between -5 and 5, with negative scores indicating negative sentiment and positive scores indicating positive sentiment.

The built-in `sentiments` dataset available when you load the tidytext package contains all of this information. You could filter it to a single lexicon with the dplyr `filter()` function, or use tidytext's `get_sentiments()` to get specific sentiment lexicons containing only the data used for that lexicon.

```
# Look at the sentiments data
data(sentiments)
sentiments
sentiments |> filter(lexicon=="nrc")
sentiments |> filter(lexicon=="bing")
sentiments |> filter(lexicon=="AFINN")

# Use the built-in get_sentiments() function
get_sentiments("nrc")
get_sentiments("bing")
get_sentiments("afinn")
```

There are a few major caveats to be aware of.

1. The sentiment lexicons we're using here were constructed either via crowdsourcing or by the work of the authors, and validated using crowdsourcing, movie/restaurant reviews, or Twitter data. It's unknown how useful it is to apply these lexicons to text from a completely different time and place (e.g., 200-year old fiction novels). Further, there are other domain-specific lexicons available, e.g., for finance data, that are better used in that context.
2. May words in the English language are fairly neutral, and aren't included in any sentiment lexicon.

[5] http://saifmohammad.com/WebPages/NRC-Emotion-Lexicon.htm
[6] https://www.cs.uic.edu/~liub/FBS/sentiment-analysis.html
[7] http://www2.imm.dtu.dk/pubdb/views/publication_details.php?id=6010

3. Methods based on unigrams (single words) do not take into account qualifiers before a word, such as in "no good" or "not true". If you have sustained sections of sarcasm or negated text, this could be problematic.
4. The size of the chunk of text that we use to add up single-word sentiment scores matters. Sentiment across many paragraphs often has positive and negative sentiment averaging out to about zero, but sentence-sized or paragraph-sized text might be better.

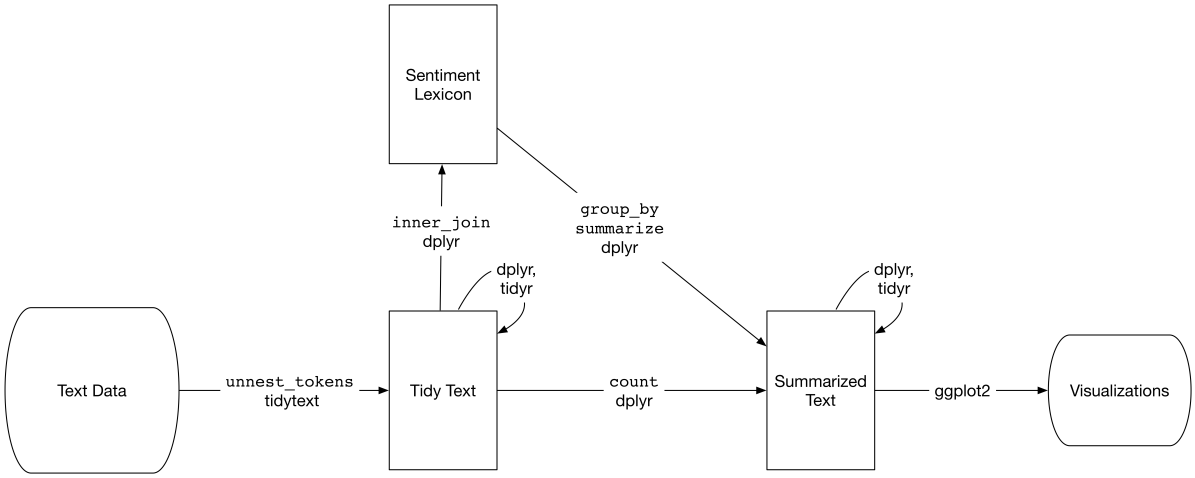


Figure 11.2: Workflow for sentiment analysis using tidy principles.

11.3.1 Sentiment analysis with tidy tools

Let's look at the most common joy words in *Emma*. To do this we will:

1. Start with the unnested Jane Austen text data.
2. Join it to the NRC sentiment lexicon.
3. Filter it to only include "joy" words.
4. Filter for only words in *Emma*.
5. Count the number of occurences of each word, sorting the output with the highest on top.

```
jatidy |>
  inner_join(get_sentiments("nrc")) |>
  filter(sentiment=="joy") |>
  filter(book=="Emma") |>
  count(word, sort=TRUE)
```

```
# A tibble: 301 x 2
```

```
   word          n
   <chr>     <int>
 1 good        359
 2 friend      166
 3 hope        143
 4 happy       125
 5 love        117
 6 deal         92
 7 found        92
 8 present      89
 9 kind         82
10 happiness    76
# i 291 more rows
```

Try running the same code but replacing "joy" with "anger" or "trust."

```
jatidy |>
  inner_join(get_sentiments("nrc")) |>
  filter(sentiment=="anger") |>
  filter(book=="Emma") |>
  count(word, sort=TRUE)
```

Let's look at how sentiment changes over time throughout each novel.

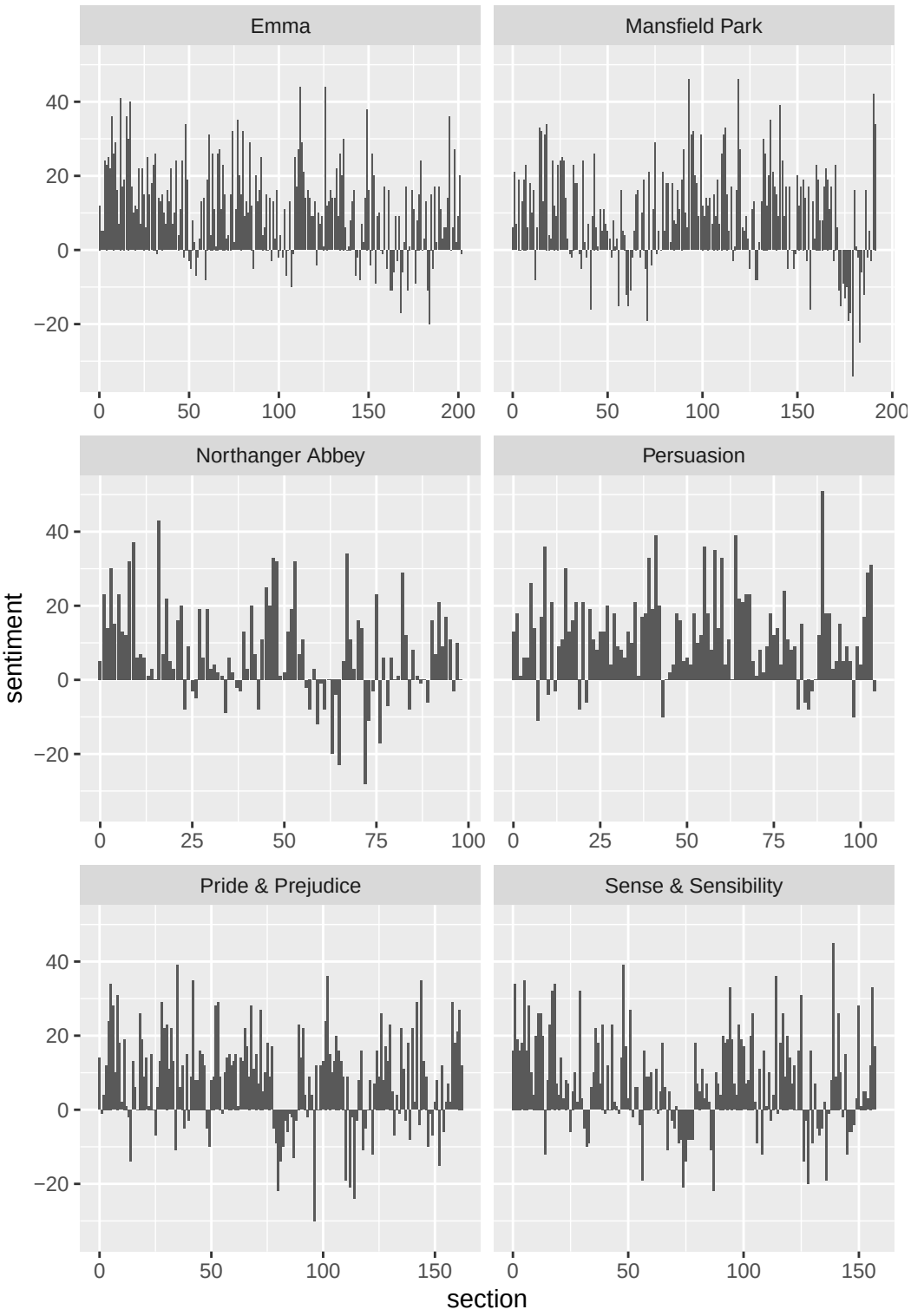
1. Start with the unnested Jane Austen text data.
2. Join it to the 'bing' sentiment lexicon (positive vs negative).
3. Create a new variable that counts up each 80-line section. First note that the %/% operator does integer division. It tells you the integer quotient without the remainder. This is a way for us to keep track of which 80-line section of text we are counting up negative and positive sentiment in.
4. Count the number of occurances of each sentiment (positive vs negative) in each section, for each book.
5. Spread the sentiment column into new columns, and fill in missing values with zeros.
6. Create your own summary sentiment score that's the total number of positive words minus the total number of negative words.

```
jatidy |>
  inner_join(get_sentiments("bing")) |>
  mutate(section=linenumber %/% 80) |>
  count(book, section, sentiment) |>
  spread(sentiment, n, fill=0) |>
  mutate(sentiment=positive-negative)
```

```
# A tibble: 920 x 5
   book  section negative positive sentiment
   <chr>   <dbl>    <dbl>    <dbl>     <dbl>
 1 Emma        0       31       43        12
 2 Emma        1       28       33         5
 3 Emma        2       30       35         5
 4 Emma        3       27       51        24
 5 Emma        4       23       46        23
 6 Emma        5       25       50        25
 7 Emma        6       25       47        22
 8 Emma        7       27       63        36
 9 Emma        8       21       47        26
10 Emma        9       11       40        29
# i 910 more rows
```

Now let's pipe that whole thing to ggplot2 to see how the sentiment changes over the course of each novel. Facet by book, and pass `scales="free_x"` so the x-axis is filled for each panel.

```
jatidy |>
  inner_join(get_sentiments("bing")) |>
  mutate(section=linenumber %/% 80) |>
  count(book, section, sentiment) |>
  spread(sentiment, n, fill=0) |>
  mutate(sentiment=positive-negative) |>
  ggplot(aes(section, sentiment)) +
    geom_col() +
    facet_wrap(~book, ncol = 2, scales = "free_x")
```


Emma
Mansfield Park
Northanger Abbey
Persuasion
Pride & Prejudice
Sense & Sensibility
sentiment
section
40
20
0
−20
0
50
100
150
200
25
75

Try comparing different sentiment lexicons. You might see different results! Different lexicons contain different ratios of positive to negative sentiment words, and thus will give you different results. You would probably want to try a few different lexicons using a known dataset to see what lexicon is most appropriate for your purpose. For more information on this topic, see section 2.3 of the Tidy Text Mining book.

11.3.2 Measuring contribution to sentiment

We could also analyze word counts that contribute to each sentiment. This first joins Jane Austen's tidy text data to the bing lexicon and counts how many times each word-sentiment linkage exists.

```
jatidy |>
  inner_join(get_sentiments("bing")) |>
  count(word, sentiment, sort=TRUE)
```

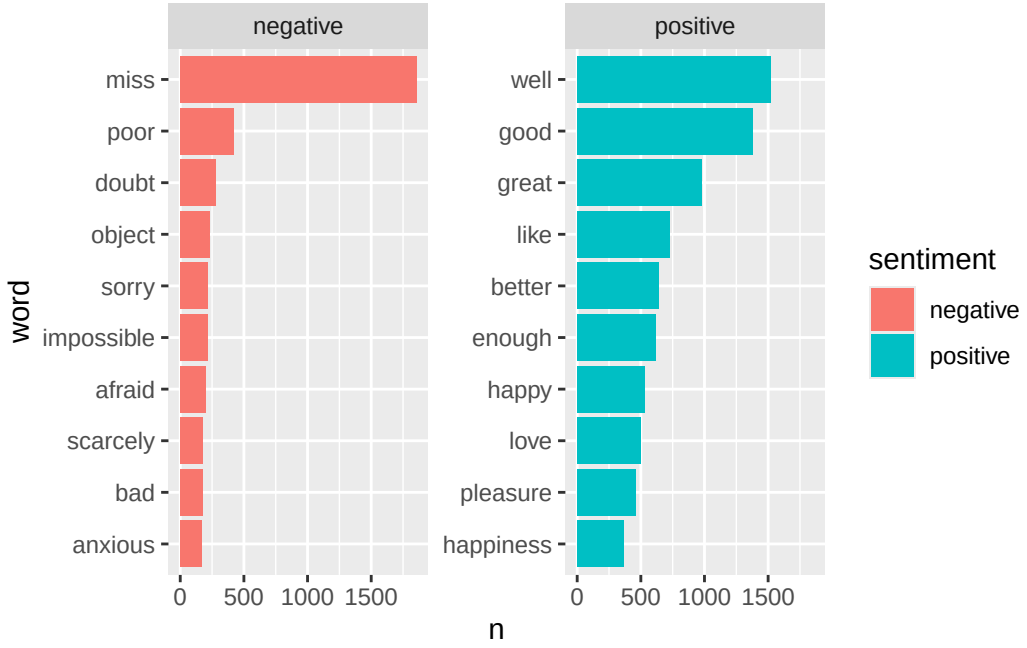
```
# A tibble: 2,585 x 3
   word      sentiment     n
   <chr>     <chr>     <int>
 1 miss      negative   1855
 2 well      positive   1523
 3 good      positive   1380
 4 great     positive    981
 5 like      positive    725
 6 better    positive    639
 7 enough    positive    613
 8 happy     positive    534
 9 love      positive    495
10 pleasure  positive    462
# i 2,575 more rows
```

Look at the help for `?top_n`. It's similar to arranging a dataset then using `head` to get the first few rows. But if we want the top n from each group, we need the `top_n` function. Let's continue the pipeline above.

1. First group by sentiment.
2. Next get the top 10 observations in each group. By default, it uses the last column here as a ranking metric.
3. The `top_n` function leaves the dataset grouped. In this case we want to ungroup the data.

4. Let's plot a bar plot showing the n for each word separately for positive and negative words.
5. We could mutate word to reorder it as a factor by n.

```
jatidy |>
  inner_join(get_sentiments("bing")) |>
  count(word, sentiment, sort=TRUE) |>
  group_by(sentiment) |>
  top_n(10) |>
  ungroup() |>
  mutate(word=reorder(word, n)) |>
  ggplot(aes(word, n)) +
  geom_col(aes(fill=sentiment)) +
  facet_wrap(~sentiment, scale="free_y") +
  coord_flip()
```



Notice that "miss" is probably erroneous here. It's used as a title for unmarried women in Jane Austen's works, and should probably be excluded from analysis. You could filter this, or you could create a custom stop words lexicon and add this to it. You could also unnest the corpus using bigrams instead of single words, then filter to look for bigrams that start with "miss," counting to show the most common ones.

```
jaorig |>
  unnest_tokens(bigram, text, token="ngrams", n=2) |>
  filter(str_detect(bigram, "^miss")) |>
  count(bigram, sort=TRUE)
```

```
# A tibble: 169 x 2
   bigram             n
   <chr>          <int>
 1 miss crawford    196
 2 miss woodhouse   143
 3 miss fairfax      98
 4 miss bates        92
 5 miss tilney       74
 6 miss bingley      67
 7 miss dashwood     55
 8 miss bennet       52
 9 miss morland      50
10 miss smith        48
# i 159 more rows
```

11.4 Word and Document Frequencies

11.4.1 TF, IDF, and TF-IDF

In text mining we're trying to get at "what is this text about?" We can start to get a sense of this by looking at the words that make up the text, and we can start to measure measure how important a word is by its **term frequency** (tf), how frequently a word occurs in a document. When we did this we saw some common words in the English language, so we took an approach to filter out our data first by a list of common stop words.

```
jatidy |>
  anti_join(stop_words) |>
  count(word, sort=TRUE)
```

Another way is to look at a term's **inverse document frequency** (idf), which decreases the weight for commonly used words and increases the weight for words that are not used very much in a collection of documents. It's defined as:

$$idf(\text{term}) = \ln\left(\frac{n_{\text{documents}}}{n_{\text{documents containing term}}}\right)$$

If you multiply the two values together, you get the **tf-idf**[8], which is the frequency of a term adjusted for how rarely it is used. The tf-idf measures how important a word is to a document in a collection (or corpus) of documents, for example, to one novel in a collection of novels or to one website in a collection of websites.

We want to use tf-idf to find the important words for the content of each document by decreasing the weight for common words and increasing the weight for words that are not used very much in a corpus of documents (in this case, the group of Jane Austen's novels). Calculating tf-idf attempts to find the words that are important (i.e., common) in a text, but not *too* common.

You could do this all manually, but there's a nice function in the tidytext package called `bind_tf_idf` that does this for you. It takes a tidy text dataset as input with one row per word, per document. One column (`word` here) contains the terms/tokens, one column contains the documents (`book` in this case), and the last necessary column contains the counts, how many times each document contains each term (`n` in this example).

Let's start by counting the number of occurances of each word in each book:

```
jatidy |>
  count(book, word, sort=TRUE)
```

Then we simply pipe that to the `bind_tf_idf` function, giving it the column names for the word, document, and count column (`word`, `book`, and `n` here):

```
jatidy |>
  count(word, book, sort=TRUE) |>
  bind_tf_idf(word, book, n)
```

You'll see that the idf (and the tf-idf) are zero for really common words. These are all words that appear in all six of Jane Austen's novels, so the idf is zero. This is how this approach decreases the weight for common words. The inverse document frequency will be a higher number for words that occur in fewer of the documents in the collection. Let's arrange descending by tf-idf (`tf_idf` with an underscore).

```
jatidy |>
  count(word, book, sort=TRUE) |>
  bind_tf_idf(word, book, n) |>
  arrange(desc(tf_idf))
```

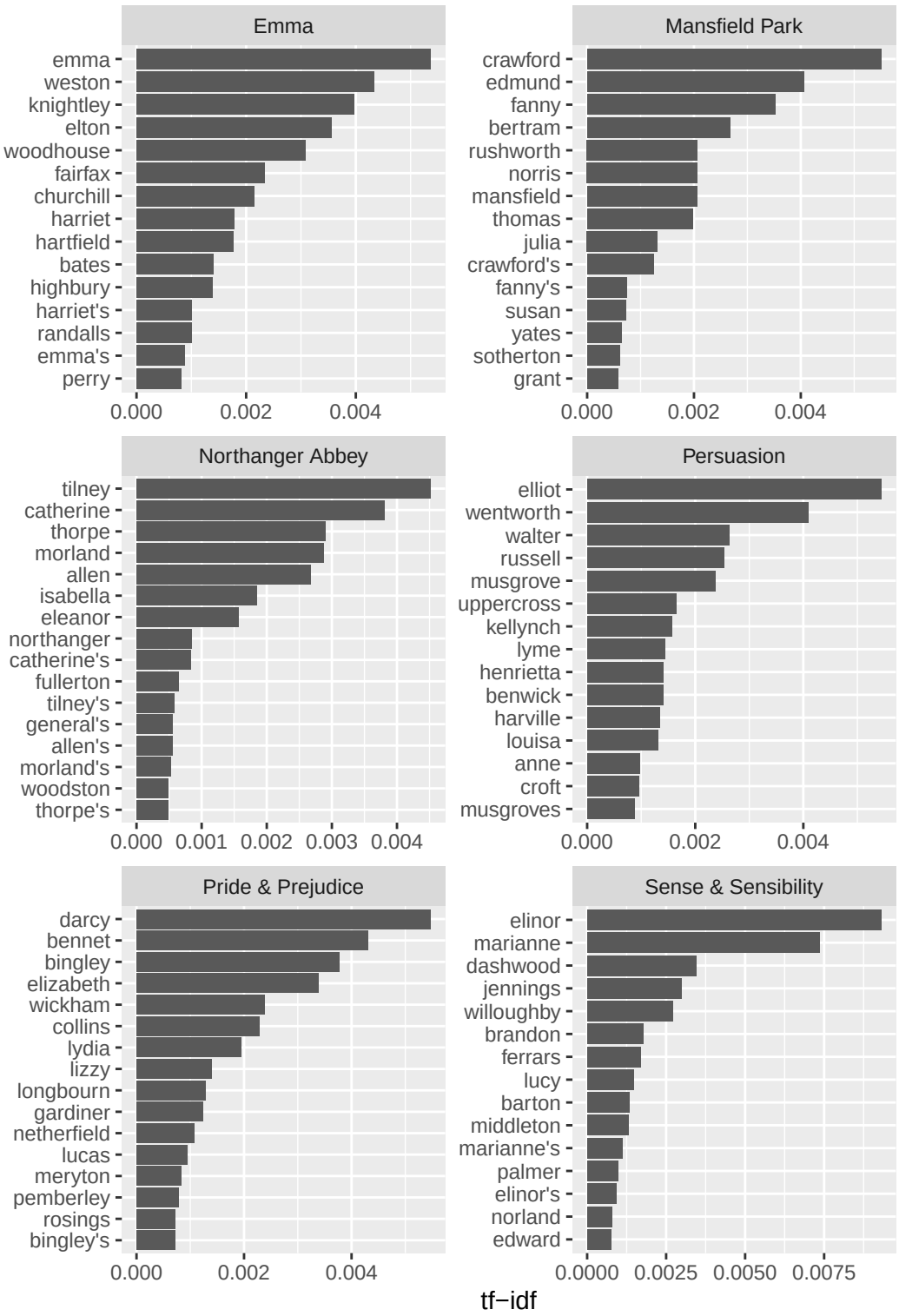
```
# A tibble: 40,379 x 6
   word      book                    n      tf   idf  tf_idf
```

[8] https://en.wikipedia.org/wiki/Tf%E2%80%93idf

```
   <chr>     <chr>               <int>   <dbl> <dbl>   <dbl>
 1 elinor    Sense & Sensibility   623 0.00519  1.79 0.00931
 2 marianne  Sense & Sensibility   492 0.00410  1.79 0.00735
 3 crawford  Mansfield Park        493 0.00307  1.79 0.00551
 4 darcy     Pride & Prejudice     373 0.00305  1.79 0.00547
 5 elliot    Persuasion            254 0.00304  1.79 0.00544
 6 emma      Emma                  786 0.00488  1.10 0.00536
 7 tilney    Northanger Abbey      196 0.00252  1.79 0.00452
 8 weston    Emma                  389 0.00242  1.79 0.00433
 9 bennet    Pride & Prejudice     294 0.00241  1.79 0.00431
10 wentworth Persuasion            191 0.00228  1.79 0.00409
# i 40,369 more rows
```

No surprise - we see all proper nouns, names that are important for each novel. None of them occur in all of novels, and they are important, characteristic words for each text within the entire corpus of Jane Austen's novels. Let's visualize this data!

```
jatidy |>
  count(word, book, sort=TRUE) |>
  bind_tf_idf(word, book, n) |>
  arrange(desc(tf_idf)) |>
  group_by(book) |>
  top_n(15) |>
  ungroup() |>
  mutate(word=reorder(word, tf_idf)) |>
  ggplot(aes(word, tf_idf)) +
  geom_col() +
  labs(x = NULL, y = "tf-idf") +
  facet_wrap(~book, ncol = 2, scales = "free") +
  coord_flip()
```


Emma
emma
weston
knightley
elton
woodhouse
fairfax
churchill
harriet
hartfield
bates
highbury
harriet's
randalls
emma's
perry
0.000
0.002
0.004
Mansfield Park
crawford
edmund
fanny
bertram
rushworth
norris
mansfield
thomas
julia
crawford's
fanny's
susan
yates
sotherton
grant
0.000
0.002
0.004
Northanger Abbey
tilney
catherine
thorpe
morland
allen
isabella
eleanor
northanger
catherine's
fullerton
tilney's
general's
allen's
morland's
woodston
thorpe's
0.000
0.001
0.002
0.003
0.004
Persuasion
elliot
wentworth
walter
russell
musgrove
uppercross
kellynch
lyme
henrietta
benwick
harville
louisa
anne
croft
musgroves
0.000
0.002
0.004
Pride & Prejudice
darcy
bennet
bingley
elizabeth
wickham
collins
lydia
lizzy
longbourn
gardiner
netherfield
lucas
meryton
pemberley
rosings
bingley's
0.000
0.002
0.004
Sense & Sensibility
elinor
marianne
dashwood
jennings
willoughby
brandon
ferrars
lucy
barton
middleton
marianne's
palmer
elinor's
norland
edward
0.0000
0.0025
0.0050
0.0075
tf−idf

11.4.2 Project Gutenberg

Project Gutenberg (https://www.gutenberg.org/) is a collection of freely available books that are in the public domain. You can get most books in all kinds of different formats (plain text, HTML, epub/kindle, etc). The **gutenbergr** package includes tools for downloading books (and stripping header/footer information), and a complete dataset of Project Gutenberg metadata that can be used to find words of interest. Includes:

- A function `gutenberg_download()` that downloads one or more works from Project Gutenberg by ID: e.g., `gutenberg_download(84)` downloads the text of Frankenstein.
- Metadata for all Project Gutenberg works as R datasets, so that they can be searched and filtered:
  - `gutenberg_metadata` contains information about each work, pairing Gutenberg ID with title, author, language, etc
  - `gutenberg_authors` contains information about each author, such as aliases and birth/death year
  - `gutenberg_subjects` contains pairings of works with Library of Congress subjects and topics

Let's use a different corpus of documents, to see what terms are important in a different set of works. Let's download some classic science texts from Project Gutenberg and see what terms are important in these works, as measured by tf-idf. We'll use three classic physics texts, and a classic Darwin text. Let's use:

- *Discourse on Floating Bodies* by Galileo Galilei: http://www.gutenberg.org/ebooks/37729
- *Treatise on Light* by Christiaan Huygens: http://www.gutenberg.org/ebooks/14725
- *Experiments with Alternate Currents of High Potential and High Frequency* by Nikola Tesla: http://www.gutenberg.org/ebooks/13476
- *On the Origin of Species By Means of Natural Selection* by Charles Darwin: http://www.gutenberg.org/ebooks/5001

These might all be physics classics, but they were written across a 300-year timespan, and some of them were first written in other languages and then translated to English.

```
library(gutenbergr)
sci <- gutenberg_download(c(37729, 14725, 13476, 1228), meta_fields = "author")
```

Now that we have the texts, let's use `unnest_tokens()` and `count()` to find out how many times each word was used in each text. Let's assign this to an object called `sciwords`. Let's go ahead and add the tf-idf also.

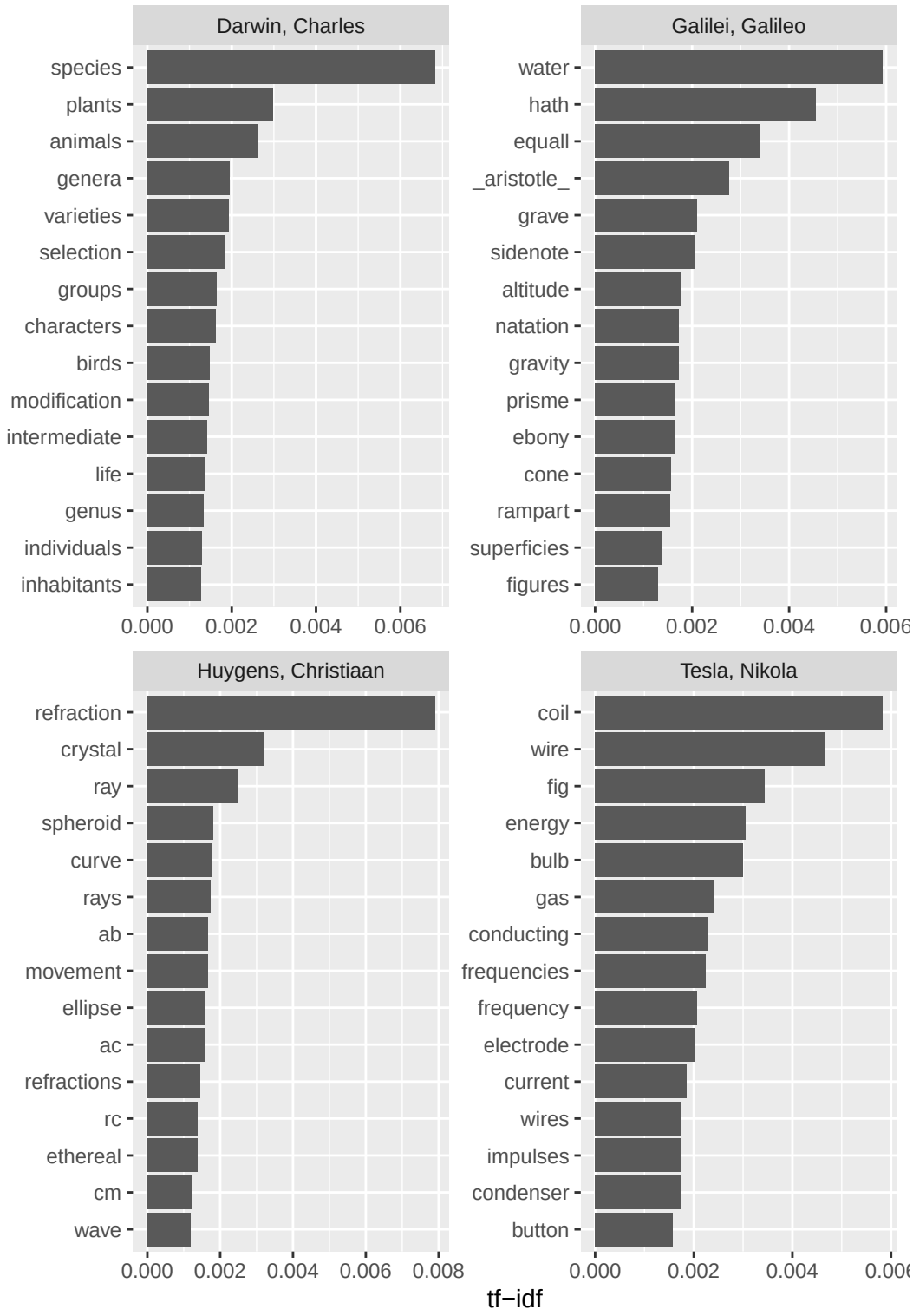
```
scitidy <- sci |>
  unnest_tokens(word, text)

sciwords <- scitidy |>
  count(word, author, sort = TRUE) |>
  bind_tf_idf(word, author, n)
sciwords
```

```
# A tibble: 16,992 x 6
   word  author                 n     tf   idf tf_idf
   <chr> <chr>              <int>  <dbl> <dbl>  <dbl>
 1 the   Darwin, Charles    10287 0.0656     0      0
 2 of    Darwin, Charles     7849 0.0501     0      0
 3 and   Darwin, Charles     4439 0.0283     0      0
 4 in    Darwin, Charles     4016 0.0256     0      0
 5 the   Galilei, Galileo    3760 0.0935     0      0
 6 to    Darwin, Charles     3605 0.0230     0      0
 7 the   Tesla, Nikola       3604 0.0913     0      0
 8 the   Huygens, Christiaan 3553 0.0928     0      0
 9 a     Darwin, Charles     2470 0.0158     0      0
10 that  Darwin, Charles     2083 0.0133     0      0
# i 16,982 more rows
```

Now let's do the same thing we did before with Jane Austen's novels:

```
sciwords |>
  group_by(author) |>
  top_n(15) |>
  ungroup() |>
  mutate(word=reorder(word, tf_idf)) |>
  ggplot(aes(word, tf_idf)) +
  geom_col() +
  labs(x = NULL, y = "tf-idf") +
  facet_wrap(~author, ncol = 2, scales = "free") +
  coord_flip()
```


Darwin, Charles
species
plants
animals
genera
varieties
selection
groups
characters
birds
modification
intermediate
life
genus
individuals
inhabitants
0.000
0.002
0.004
0.006
Galilei, Galileo
water
hath
equall
aristotle
grave
sidenote
altitude
natation
gravity
prisme
ebony
cone
rampart
superficies
figures
0.000
0.002
0.004
0.006
Huygens, Christiaan
refraction
crystal
ray
spheroid
curve
rays
ab
movement
ellipse
ac
refractions
rc
ethereal
cm
wave
0.000
0.002
0.004
0.006
0.008
Tesla, Nikola
coil
wire
fig
energy
bulb
gas
conducting
frequencies
frequency
electrode
current
wires
impulses
condenser
button
0.000
0.002
0.004
tf−idf

We see some weird things here. We see "fig" for Tesla, but I doubt he was writing about a fruit tree. We see things like ab, ac, rc, etc for Huygens – these are names of rays and angles, etc. We could create a custom stop words dictionary to remove these. Let's create a stop words data frame, then anti join that before plotting.

```
mystopwords <- tibble(word=c("ab", "ac", "rc", "cm", "cg", "cb", "ak", "bn", "fig"))
sciwords |>
  anti_join(mystopwords) |>
  group_by(author) |>
  top_n(15) |>
  ungroup() |>
  mutate(word=reorder(word, tf_idf)) |>
  ggplot(aes(word, tf_idf)) +
  geom_col() +
  labs(x = NULL, y = "tf-idf") +
  facet_wrap(~author, ncol = 2, scales = "free") +
  coord_flip()
```

11.5 Topic Modeling

Topic modeling[9] is a method for unsupervised classification of such documents, similar to clustering on numeric data, which finds natural groups of items even when we're not sure what we're looking for. It's a way to find abstract "topics" that occur in a collection of documents, and it's frequently used to find hidden semantic structures in a text body. Topic models can help us understand large collections of unstructured text bodies. In addition to text mining tasks like what we'll do here, topic models have been used to detect useful structures in data such as genetic information, images, and networks, and have also been used in bioinformatics.[10]

Latent Dirichlet Allocation[11] is one of the most common algorithms used in topic modeling. LDA treats each document as a mixture of topics, and each topic as a mixture of words:

1. *Each document is a mixture of topics.* Each document contains words from several topics in particular proportions. For example, in a two-topic model we could say "Document 1 is 90% topic A and 10% topic B, while Document 2 is 30% topic A and 70% topic B."
2. *Every topic is a mixture of words.* Imagine a two-topic model of American news, with one topic for "politics" and one for "entertainment." Common words in the politics topic might be "President", "Congress", and "government", while the entertainment topic may

[9] https://en.wikipedia.org/wiki/Topic_model

[10] Blei, David (April 2012). "Probabilistic Topic Models". *Communications of the ACM*. 55 (4): 77-84. doi:10.1145/2133806.2133826

[11] https://en.wikipedia.org/wiki/Latent_Dirichlet_allocation

be made up of words such as "movies", "television", and "actor". Words can be shared between topics; a word like "budget" might appear in both equally.

LDA attempts to estimate both of these at the same time: finding words associated with each topic, while simutaneously determining the mixture of topics that describes each document.

11.5.1 Document-term matrix

Before we can get started in topic modeling we need to take a look at another common format for storing text data that's not the tidy one-token-per-document-per-row format we've used so far (what we get from `unnest_tokens`). Another very common structure that's used by other text mining packages (such as **tm** or **quanteda**) is the **document-term matrix**[12] (DTM). This is a matrix where:

- Each row represents one document (such as a book or article).
- Each column represents one term.
- Each value contains the number of appearances of that term in that document.

Since most pairings of document and term do not occur (they have the value zero), DTMs are usually implemented as sparse matrices. These objects can be treated as though they were matrices (for example, accessing particular rows and columns), but are stored in a more efficient format. DTM objects can't be used directly with tidy tools, just as tidy data frames can't be used as input for other text mining packages. The tidytext package provides two verbs that convert between the two formats.

- `tidy()` turns a DTM into a tidy data frame. This verb comes from the broom package.
- `cast()` turns a tidy one-term-per-row data frame into a matrix. tidytext provides three variations of this verb, each converting to a different type of matrix:
  - `cast_sparse()`: converts to a sparse matrix from the Matrix package.
  - `cast_dtm()`: converts to a `DocumentTermMatrix` object from tm.
  - `cast_dfm()`: converts to a `dfm` object from quanteda.

First let's load the `AssociatedPress` data from the topicmodels package. Take a look. We can see that the `AssociatedPress` data is a DTM with 2,246 documents (AP articles from 1988) and 10,473 terms. The 99% sparsity indicates that the matrix is almost complete made of zeros, i.e., almost all the document-term pairs are zero – most terms are not used in most documents. If we want to use the typical tidy tools we've used above, we'll use the `tidy()` function to melt this matrix into a tidy one-token-per-document-per-row format. Notice that this only returns the 302,031 non-zero entries.

[12] https://en.wikipedia.org/wiki/Document-term_matrix

```
library(topicmodels)
data(AssociatedPress)
AssociatedPress
tidy(AssociatedPress)
```

First let's use the `LDA()` function from the topicmodels package, setting `k = 2`, to create a two-topic LDA model. This function returns an object containing the full details of the model fit, such as how words are associated with topics and how topics are associated with documents. Fitting the model is easy. For the rest of this section we'll be exploring and interpreting the model.

```
# set a seed so that the output of the model is predictable
ap_lda <- LDA(AssociatedPress, k = 2, control=list(seed=1234))
ap_lda
```

11.5.2 Word-topic probabilities

Displaying the model itself, `ap_lda` isn't that interesting. The tidytext package provides a `tidy` method for extracting the per-topic-per-word probabilities, called β ("beta"), from the model.

```
ap_topics <- tidy(ap_lda, matrix = "beta")
ap_topics
```

```
# A tibble: 20,946 x 3
   topic term           beta
   <int> <chr>         <dbl>
 1     1 aaron      1.69e-12
 2     2 aaron      3.90e- 5
 3     1 abandon    2.65e- 5
 4     2 abandon    3.99e- 5
 5     1 abandoned  1.39e- 4
 6     2 abandoned  5.88e- 5
 7     1 abandoning 2.45e-33
 8     2 abandoning 2.34e- 5
 9     1 abbott     2.13e- 6
10     2 abbott     2.97e- 5
# i 20,936 more rows
```

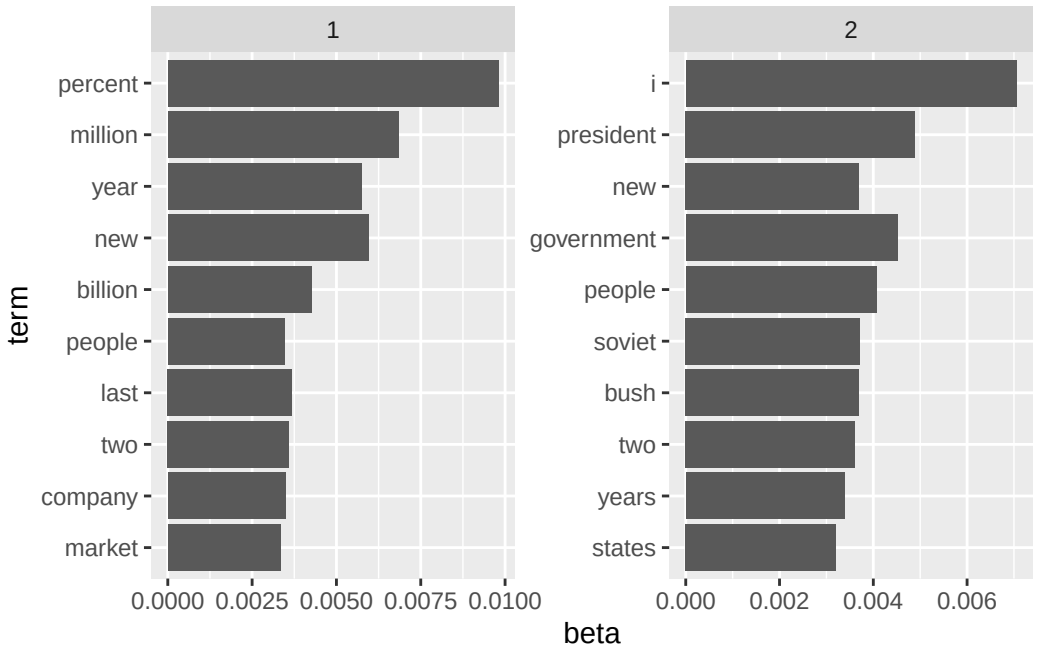
This returns a one-topic-per-term-per-row format. For each combination, the model computes the probability of that term being generated from that topic. For example, the term "aaron"

has a 1.686917×10^{-12} probability of being generated from topic 1, but a 3.8959408×10^{-5} probability of being generated from topic 2.

We could use dplyr's `top_n()` to find the 10 terms that are most common within each topic. Because this returns a tidy data frame, we could easily continue piping to ggplot2.

```
# What are the top words for each topic?
ap_topics |>
  group_by(topic) |>
  top_n(10) |>
  ungroup() |>
  arrange(topic, desc(beta))
```

```
# Continue piping to ggplot2
ap_topics |>
  group_by(topic) |>
  top_n(10) |>
  ungroup() |>
  arrange(topic, desc(beta)) |>
  mutate(term = reorder(term, beta)) |>
  ggplot(aes(term, beta)) +
  geom_col() +
  facet_wrap(~topic, scales = "free") +
  coord_flip()
```



This visualization lets us understand the two topics that were extracted from the articles. Common words in topic 1 include "percent", "million", "billion", and "company". Perhaps topic 1 represents business or financial news. Common in topic 2 include "president", "government", and "soviet", suggeting that this topic represents political news. Note that some words, such as "new" and "people", are common within both topics. This is an advantage (as opposed to "hard clustering" methods): topics used in natural language could have some overlap in terms of words.

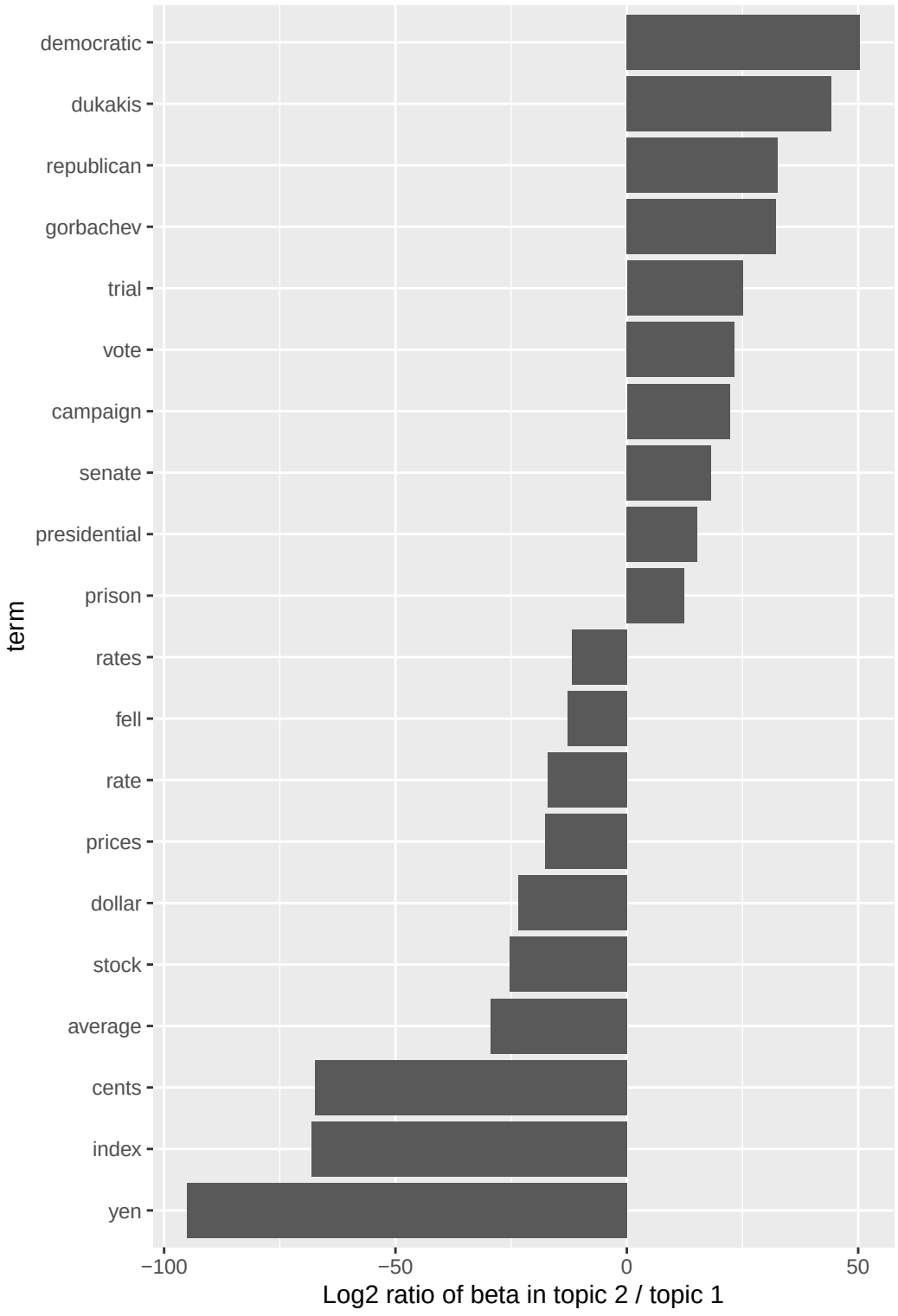
Let's look at the terms that had the *greatest difference* in β between topic 1 and topic 2. This can be estimated based on the log ratio of the two: $\log_2(\frac{\beta_2}{\beta_1})$ (a log ratio is useful because it makes the difference symmetrical: β_2 being twice as large leads to a log ratio of 1, while β_1 being twice as large results in -1). To constrain it to a set of especially relevant words, we can filter for relatively common words, such as those that have a β greater than 1/1000 in at least one topic.

First let's turn `1` and `2` into `topic1` and `topic2` so that after the `spread` we'll easily be able to work with those columns.

```
ap_topics |>
  mutate(topic = paste0("topic", topic)) |>
  spread(topic, beta) |>
  filter(topic1 > .001 | topic2 > .001) |>
  mutate(log_ratio = log2(topic2 / topic1))
```

We could continue piping to ggplot2. First, let's create a new variable we'll group on, which is the direction of imbalance. We'll also create a variable showing the absolute value of the log ratio, which is a directionless value indicating the magnitude of the effect. This lets us select the top 10 terms most associated with either topic1 or topic2.

```
ap_topics |>
  mutate(topic = paste0("topic", topic)) |>
  spread(topic, beta) |>
  filter(topic1 > .001 | topic2 > .001) |>
  mutate(log_ratio = log2(topic2 / topic1)) |>
  mutate(direction = (log_ratio>0)) |>
  mutate(absratio=abs(log_ratio)) |>
  group_by(direction) |>
  top_n(10) |>
  ungroup() |>
  mutate(term = reorder(term, log_ratio)) |>
  ggplot(aes(term, log_ratio)) +
  geom_col() +
  labs(y = "Log2 ratio of beta in topic 2 / topic 1") +
  coord_flip()
```


democratic
dukakis
republican
gorbachev
trial
vote
campaign
senate
presidential
prison
rates
fell
rate
prices
dollar
stock
average
cents
index
yen
term
−100
−50
0
50
Log2 ratio of beta in topic 2 / topic 1

We can see that the words more common in topic 2 include political parties such as "democratic" and "republican", as well as politician's names such as "dukakis" and "gorbachev". Topic 1 was more characterized by currencies like "yen" and "dollar", as well as financial terms such as "index", "prices" and "rates". This helps confirm that the two topics the algorithm identified were political and financial news.

11.5.3 Document-topic probabilities

Above we estimated the per-topic-per-word probabilities, β ("beta"). LDA also models each document as a mixture of topics. Let's look at the per-document-per-topic probabilities, γ ("gamma"), with the `matrix = "gamma"` argument to `tidy()`.

```
ap_documents <- tidy(ap_lda, matrix = "gamma")
ap_documents
```

```
# A tibble: 4,492 x 3
   document topic    gamma
      <int> <int>    <dbl>
 1        1     1 0.248
 2        2     1 0.362
 3        3     1 0.527
 4        4     1 0.357
 5        5     1 0.181
 6        6     1 0.000588
 7        7     1 0.773
 8        8     1 0.00445
 9        9     1 0.967
10       10     1 0.147
# i 4,482 more rows
```

These values represent the estimated proportion of words from that document that are generated from that topic. For example, the the model estimates only about 25% of the words in document 1 were generated from topic 1.

Most of these documents were drawn from a mix of the two topics, but document 6 was drawn almost entirely from topic 2, having a γ from topic 1 close to zero. To check this answer, we could `tidy()` the document-term matrix.

```
tidy(AssociatedPress) |>
  filter(document == 6) |>
  arrange(desc(count))
```

```
# A tibble: 287 x 3
   document term           count
      <int> <chr>          <dbl>
 1        6 noriega           16
 2        6 panama            12
 3        6 jackson            6
 4        6 powell             6
 5        6 administration     5
 6        6 economic           5
 7        6 general            5
 8        6 i                  5
 9        6 panamanian         5
10        6 american           4
# i 277 more rows
```

Based on the most common words, this looks like an article about the relationship between the American government and Panamanian dictator Manuel Noriega, which means the algorithm was right to place it in topic 2 (as political/national news).

11.6 Case Studies & Examples

11.6.1 The Great Library Heist

> From section 6.2 of Tidy Text Mining.

When examining a statistical method, it can be useful to try it on a very simple case where you know the "right answer". For example, we could collect a set of documents that definitely relate to four separate topics, then perform topic modeling to see whether the algorithm can correctly distinguish the four groups. This lets us double-check that the method is useful, and gain a sense of how and when it can go wrong. We'll try this with some data from classic literature.

Suppose a vandal has broken into your study and torn apart four of your books:

- *Great Expectations* by Charles Dickens
- *The War of the Worlds* by H.G. Wells
- *Twenty Thousand Leagues Under the Sea* by Jules Verne
- *Pride and Prejudice* by Jane Austen

This vandal has torn the books into individual chapters, and left them in one large pile. How can we restore these disorganized chapters to their original books? This is a challenging problem since the individual chapters are **unlabeled**: we don't know what words might distinguish

them into groups. We'll thus use topic modeling to discover how chapters cluster into distinct topics, each of them (presumably) representing one of the books.

We'll retrieve the text of these four books using the gutenbergr package:

```
library(gutenbergr)
titles <- c("Twenty Thousand Leagues under the Sea",
            "The War of the Worlds",
            "Pride and Prejudice",
            "Great Expectations")
books <- gutenberg_works(title %in% titles) |>
  gutenberg_download(meta_fields = "title")
```

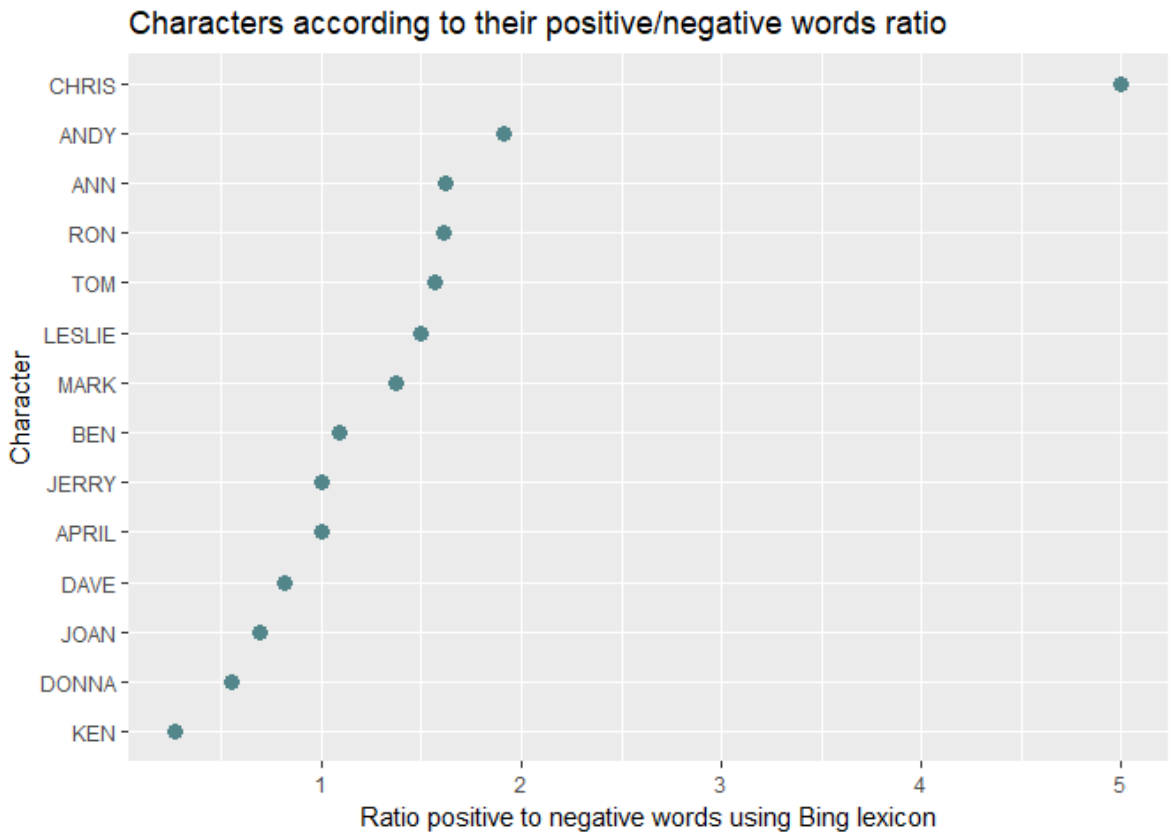
You'll want to start by dividing the books these into chapters, use tidytext's `unnest_tokens()` to separate them into words, then remove `stop_words`. You'll be treating every chapter as a separate "document", each with a name like `Great Expectations_1` or `Pride and Prejudice_11`. You'll cast this into a DTM then run LDA. You'll look at the word-topic probabilities to try to get a sense of which topic represents which book, and you'll use document-topic probabilities to assign chapters to their books. See section 6.2 of Tidy Text Mining for code and a walk-through.

11.6.2 Happy Galentine's Day!

Source: **https://suzan.rbind.io/2018/02/happy-galentines-day/**

This analysis does a tidy text mining analysis of several scripts from *Parks and Recreation*. In addition to the kinds of analyses we've performed here, it also illustrates some additional functionality for extracting text from PDF documents (the scripts were only available as PDFs).

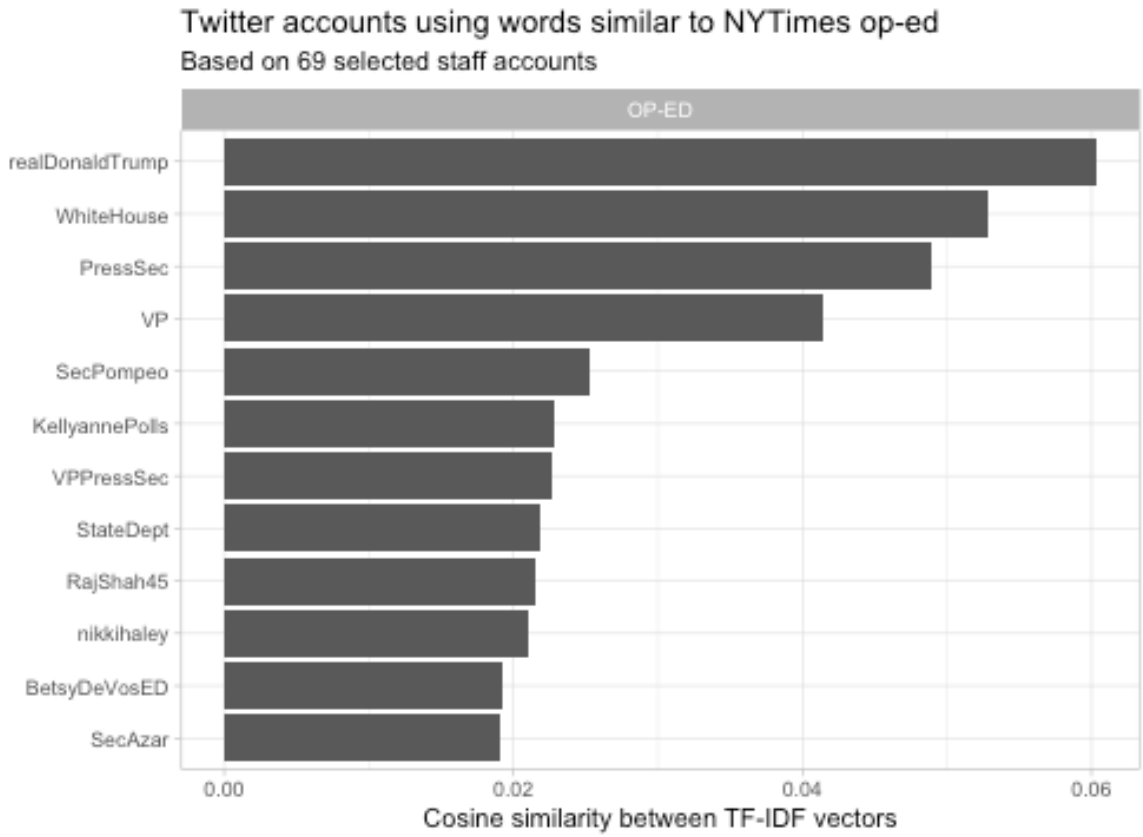

ANDY
rush
sugar
lying
rock
she
walrus
vegetables
shut
pick
meat
manager
macklin
license
horses
hates
gizmo
fruits
fighting
drummer
bert
babe
ANN
date
doctors
definitely
doing
wasn't
seven
hey
coming
totter
telling
surprisingly
snakes
sitting
saying
pretend
downs
kitchen
impossible
immerse
harris
dr
cult
basic
APRIL
derek
dad
pink
gay
dumb
boyfriend
woman
upset
teddy
jeremy
ben
BEN
eighteen
town
rhymes
ice
bird
curse
perd
mayor
sorry
totally
LESLIE
wamapoke
ben
forum
pikitis
festival
dave
gorp
community
our
pawnee
park
RON
hernia
canada
april
type
jaeger
child
chair
wendy
maze
line
duke
damn
TOM
chick
jerry
duke
hey
hot
sweetheart
skates
soap
smile
smell
sky
single
peace
paul
momma's
machete
hot
rain
coaster
business
bottles
plant
party
answer
annie
word
tf_idf



11.6.3 Who wrote the anti-Trump New York Times op-ed?

Source: **http://varianceexplained.org/r/op-ed-text-analysis/**

In September 2018 the New York Times published an anonymous op-ed, "I Am Part of the Resistance Inside the Trump Administration", written by a "senior official in the Trump administration". Lots of data scientists tried to use text-mining techniques to determine who wrote this op-ed. This analysis compares the text of the op-ed to the set of documents representing "senior officials." In addition to what we've covered here, this also covers scraping text from Twitter accounts, and methods for comparing TF-IDF vectors using cosine similarity, which was touched on in section 4.2 of Tidy Text Mining.



11.6.4 Seinfeld dialogues

Source: **https://pradeepadhokshaja.wordpress.com/2018/08/06/looking-at-seinfeld-dialogues-using-tidytext/**

Data: **https://www.kaggle.com/thec03u5/seinfeld-chronicles**

This analysis uses the tidytext package to analyze the full text of the entire *Seinfeld* series that ran 1989-1998.

11.6.5 Sentiment analysis in Shakespeare tragedies

Source: **https://peerchristensen.netlify.com/post/fair-is-foul-and-foul-is-fair-a-tidytext-entiment-analysis-of-shakespeare-s-tragedies/**

This analysis illustrates a tidytext approach to examine the use of sentiment words in the tragedies written by William Shakespeare.

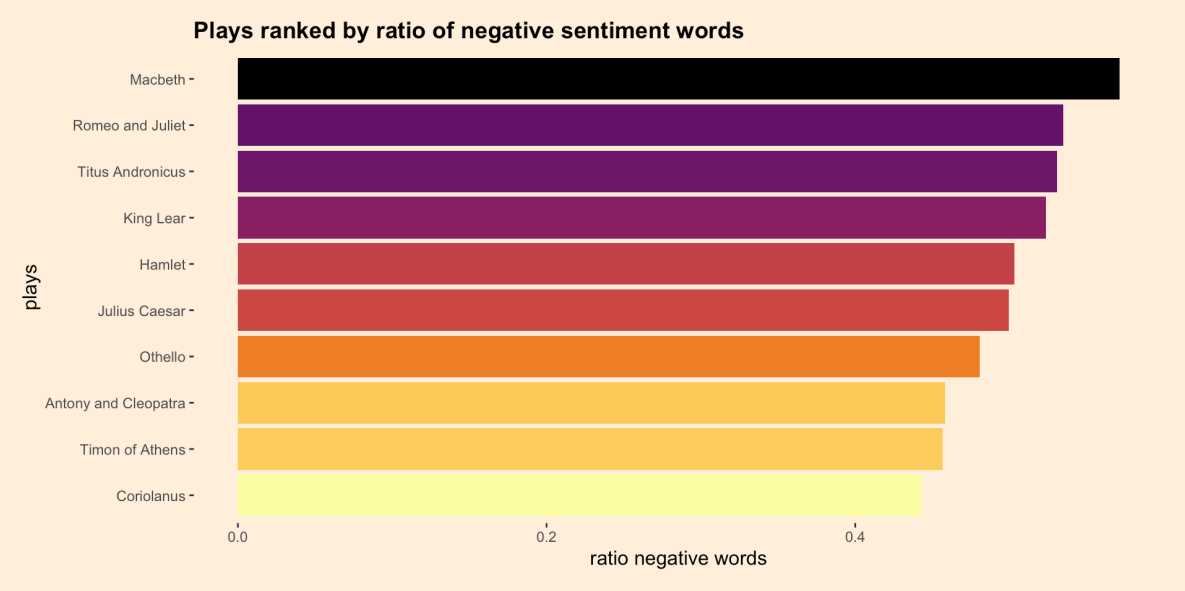


Figure 11.3: Plays ranked by ratio of negative sentiment words

Figure 11.4: Sentiment over time for Romeo & Juliet

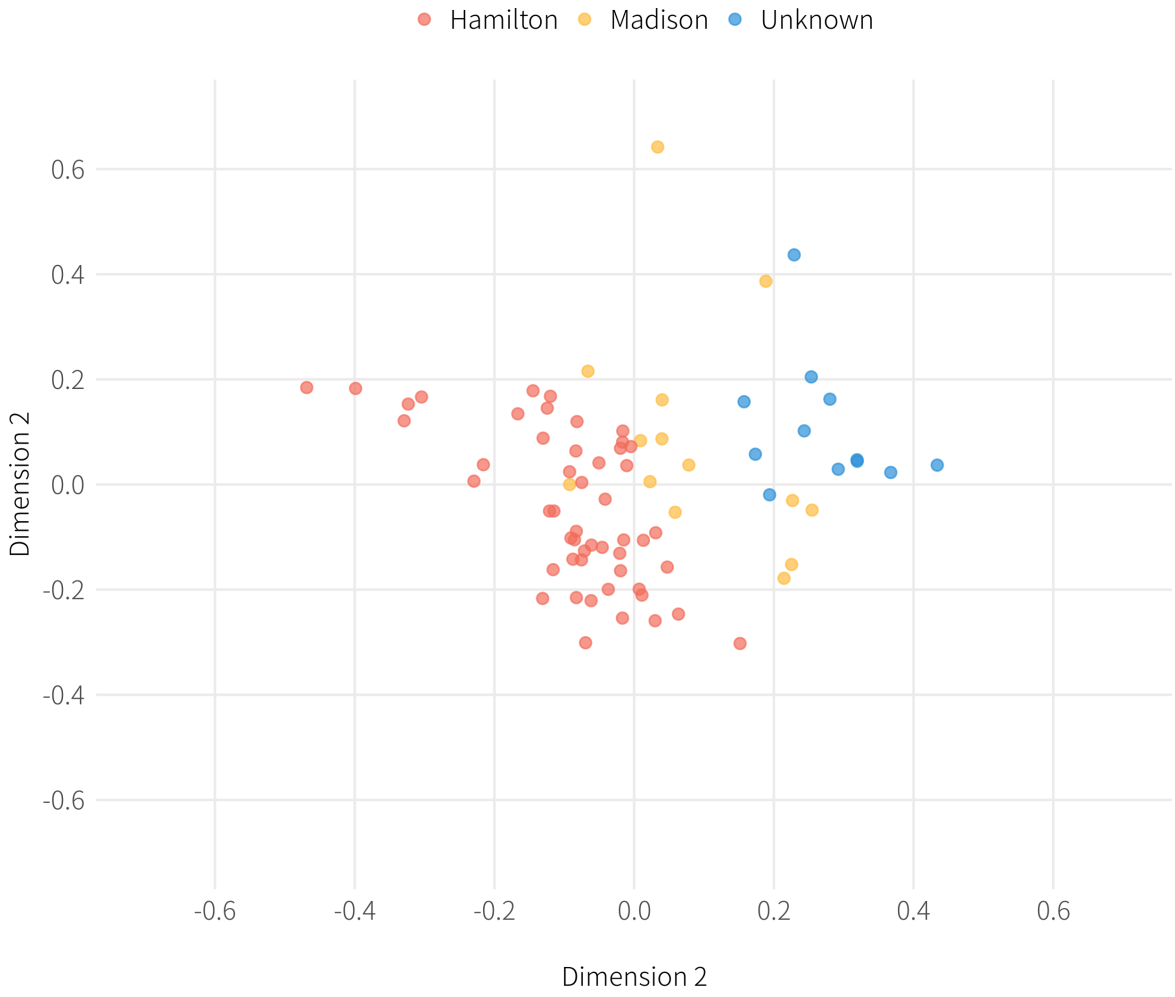
11.6.6 Authorship of the Federalist Papers

Source: **https://kanishka.xyz/2018/my-first-few-open-source-contributions-authorship-attribution-of-the-federalist-papers/**

The Federalist Papers were written as essays between 1787-1788 by Alexander Hamilton, John Jay and James Madison to promote the ratification of the constitution. They were all authored under the pseudonym 'Publius', which was a tribute to the founder of the Roman Republic, but were then confirmed to be written by the three authors where Hamilton wrote 51 essays, Jay wrote 5, Madison wrote 14, and Hamilton and Madison co-authored 3. The authorship of the remaining 12 has been in dispute. This post uses tidy text mining and some additional functionality to try to determine who authored the 12 in dispute.

Authorship Analysis of the Federalist Papers

Papers with disputed authors lie far apart from Hamilton
but much closer to Madison



12 Count-Based Differential Expression Analysis of RNA-seq Data

This is an introduction to RNAseq analysis involving reading in quantitated gene expression data from an RNA-seq experiment, exploring the data using base R functions and then analysis with the DESeq2 package.

Recommended reading:

1. Conesa et al. A survey of best practices for RNA-seq data analysis. *Genome Biology* 17:13 (2016).
2. Soneson et al. "Differential analyses for RNA-seq: transcript-level estimates improve gene-level inferences." *F1000Research* 4 (2015).
3. Abstract and introduction sections of Himes et al. "RNA-Seq transcriptome profiling identifies CRISPLD2 as a glucocorticoid responsive gene that modulates cytokine function in airway smooth muscle cells." *PLoS ONE* 9.6 (2014): e99625.
4. Review the *Introduction* (10.1), *Tibbles vs. data.frame* (10.3), and *Interacting with Older Code* (10.4) sections of the ***R for Data Science* book**. We will initially be using the `read_*` functions from the **readr** package. These functions load data into a *tibble* instead of R's traditional data.frame. Tibbles are data frames, but they tweak some older behaviors to make life a little easier. These sections explain the few key small differences between traditional data.frames and tibbles.

Data needed:

- Length-scaled count matrix (i.e., `countData`): airway_scaledcounts.csv
- Sample metadata (i.e., `colData`): airway_metadata.csv
- Gene Annotation data: annotables_grch38.csv

12.1 Background

12.1.1 The biology

The data for this chapter comes from:

> Himes *et al.* "RNA-Seq Transcriptome Profiling Identifies CRISPLD2 as a Glucocorticoid Responsive Gene that Modulates Cytokine Function in Airway Smooth Muscle Cells." *PLoS ONE*. 2014 Jun 13;9(6):e99625. PMID: 24926665.

Glucocorticoids are potent inhibitors of inflammatory processes, and are widely used to treat asthma because of their anti-inflammatory effects on airway smooth muscle (ASM) cells. But what's the molecular mechanism? This study used RNA-seq to profile gene expression changes in four different ASM cell lines treated with dexamethasone, a synthetic glucocorticoid molecule. They found a number of differentially expressed genes comparing dexamethasone-treated ASM cells to control cells, but focus much of the discussion on a gene called CRISPLD2. This gene encodes a secreted protein known to be involved in lung development, and SNPs in this gene in previous GWAS studies are associated with inhaled corticosteroid resistance and bronchodilator response in asthma patients. They confirmed the upregulated CRISPLD2 mRNA expression with qPCR and increased protein expression using Western blotting.

They did their analysis using Tophat and Cufflinks. We're taking a different approach and using an R package called DESeq2. Click here to read more on DESeq2 and other approaches.

12.1.2 Data pre-processing

Analyzing an RNAseq experiment begins with sequencing reads. There are many ways to begin analyzing this data, and you should check out the three papers below to get a sense of other analysis strategies. In the workflow we'll use here, sequencing reads were pseudoaligned to a reference transcriptome and the abundance of each transcript quantified using **kallisto** (software, paper). Transcript-level abundance estimates were then summarized to the gene level to produce length-scaled counts using **txImport** (software, paper), suitable for using in count-based analysis tools like DESeq. This is the starting point - a "count matrix," where each cell indicates the number of reads mapping to a particular gene (in rows) for each sample (in columns). This is one of several potential workflows, and relies on having a well-annotated reference transcriptome. However, there are many well-established alternative analysis paths, and the goal here is to provide a reference point to acquire fundamental skills that will be applicable to other bioinformatics tools and workflows.

1. Conesa, A. et al. "A survey of best practices for RNA-seq data analysis." *Genome Biology* 17:13 (2016).
2. Soneson, C., Love, M. I. & Robinson, M. D. "Differential analyses for RNA-seq: transcript-level estimates improve gene-level inferences." *F1000Res.* 4:1521 (2016).
3. Griffith, Malachi, et al. "Informatics for RNA sequencing: a web resource for analysis on the cloud." *PLoS Comput Biol* 11.8: e1004393 (2015).

This data was downloaded from GEO (GSE:GSE52778). You can read more about how the data was processed by going over the slides. If you'd like to see the code used for the upstream pre-processing with kallisto and txImport, see the code.

12.1.3 Data structure

We'll come back to this again later, but the data at our starting point looks like this (*note: this is a generic schematic - our genes are not actually* `geneA` *and* `geneB`*, and our samples aren't called* `ctrl_1`*,* `ctrl_2`*, etc.*):

countData

gene	ctrl_1	ctrl_2	exp_1	exp_1
geneA	10	11	56	45
geneB	0	0	128	54
geneC	42	41	59	41
geneD	103	122	1	23
geneE	10	23	14	56
geneF	0	1	2	0
…	…	…	…	…
…	…	…	…	…
…	…	…	…	…

colData

id	treatment	sex
ctrl_1	control	male
ctrl_2	control	female
exp_1	treatment	male
exp_2	treatment	female

Sample names:
ctrl_1, ctrl_2, exp_1, exp_2

countData is the count matrix
(number of reads mapping to each gene for each sample)

colData describes metadata about the *columns* of countData

First column of colData must match column names of countData (-1st)

That is, we have **two tables**:

1. The "count matrix" (called the `countData` in DESeq-speak) – where *genes* are in *rows* and *samples* are in *columns*, and the number in each cell is the number of reads that mapped to exons in that gene for that sample: **airway_scaledcounts.csv**.
2. The sample metadata (called the `colData` in DESeq-speak) – where *samples* are in *rows* and *metadata* about those samples are in *columns*: **airway_metadata.csv**. It's called the `colData` because this table supplies metadata/information about the columns of the `countData` matrix. Notice that the first column of `colData` must match the column names of `countData` (except the first, which is the gene ID column).[1]

[1]This only works when using the argument `tidy=TRUE` when creating the `DESeqDataSetFromMatrix()`.

12.2 Import data

First, let's load the **readr**, **dplyr**, and **ggplot2** packages. Then let's import our data with **readr**'s `read_csv()` function (*note*: not `read.csv()`). Let's read in the actual count data and the experimental metadata.

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

mycounts <- read_csv("data/airway_scaledcounts.csv")
metadata <-  read_csv("data/airway_metadata.csv")
```

Now, take a look at each.

```
mycounts
```

```
# A tibble: 38,694 x 9
   ensgene     SRR1039508 SRR1039509 SRR1039512 SRR1039513 SRR1039516 SRR1039517
   <chr>            <dbl>      <dbl>      <dbl>      <dbl>      <dbl>      <dbl>
 1 ENSG000000~        723        486        904        445       1170       1097
 2 ENSG000000~          0          0          0          0          0          0
 3 ENSG000000~        467        523        616        371        582        781
 4 ENSG000000~        347        258        364        237        318        447
 5 ENSG000000~         96         81         73         66        118         94
 6 ENSG000000~          0          0          1          0          2          0
 7 ENSG000000~       3413       3916       6000       4308       6424      10723
 8 ENSG000000~       2328       1714       2640       1381       2165       2262
 9 ENSG000000~        670        372        692        448        917        807
10 ENSG000000~        426        295        531        178        740        651
# i 38,684 more rows
# i 2 more variables: SRR1039520 <dbl>, SRR1039521 <dbl>
```

```
metadata
```

```
# A tibble: 8 x 4
  id         dex     celltype geo_id
  <chr>      <chr>   <chr>    <chr>
1 SRR1039508 control N61311   GSM1275862
2 SRR1039509 treated N61311   GSM1275863
```

```
3 SRR1039512 control N052611  GSM1275866
4 SRR1039513 treated N052611  GSM1275867
5 SRR1039516 control N080611  GSM1275870
6 SRR1039517 treated N080611  GSM1275871
7 SRR1039520 control N061011  GSM1275874
8 SRR1039521 treated N061011  GSM1275875
```

Notice something here. The sample IDs in the metadata sheet (SRR1039508, SRR1039509, etc.) exactly match the column names of the countdata, except for the first column, which contains the Ensembl gene ID. This is important, and we'll get more strict about it later on.

12.3 Poor man's DGE

Let's look for differential gene expression. ***Note: this analysis is for demonstration only. NEVER do differential expression analysis this way!***

Let's start with an exercise.

Exercise 1

If we look at our metadata, we see that the control samples are SRR1039508, SRR1039512, SRR1039516, and SRR1039520. This bit of code will take the `mycounts` data, `mutate()` it to add a column called `controlmean`, then `select()` only the gene name and this newly created column, and assigning the result to a new object called `meancounts`. (*Hint*: `mycounts |> mutate(...) |> select(...)`)

```
meancounts <- mycounts |>
  mutate(controlmean = (SRR1039508+SRR1039512+SRR1039516+SRR1039520)/4) |>
  select(ensgene, controlmean)
meancounts
```

```
# A tibble: 38,694 x 2
   ensgene         controlmean
   <chr>                 <dbl>
 1 ENSG00000000003      901.
 2 ENSG00000000005        0
 3 ENSG00000000419      520.
 4 ENSG00000000457      340.
 5 ENSG00000000460       97.2
 6 ENSG00000000938        0.75
 7 ENSG00000000971     5219
 8 ENSG00000001036     2327
```

```
 9 ENSG00000001084      756.
10 ENSG00000001167      528.
# i 38,684 more rows
```

Exercise 2

Build off of this code, `mutate()` it once more (prior to the `select()`) function, to add another column called `treatedmean` that takes the mean of the expression values of the treated samples. Then `select()` only the `ensgene`, `controlmean` and `treatedmean` columns, assigning it to a new object called meancounts.

```
# A tibble: 38,694 x 3
   ensgene         controlmean treatedmean
   <chr>                 <dbl>       <dbl>
 1 ENSG00000000003      901.         658
 2 ENSG00000000005        0            0
 3 ENSG00000000419      520.         546
 4 ENSG00000000457      340.         316.
 5 ENSG00000000460       97.2         78.8
 6 ENSG00000000938        0.75         0
 7 ENSG00000000971     5219         6688.
 8 ENSG00000001036     2327         1786.
 9 ENSG00000001084      756.         578
10 ENSG00000001167      528.         348.
# i 38,684 more rows
```

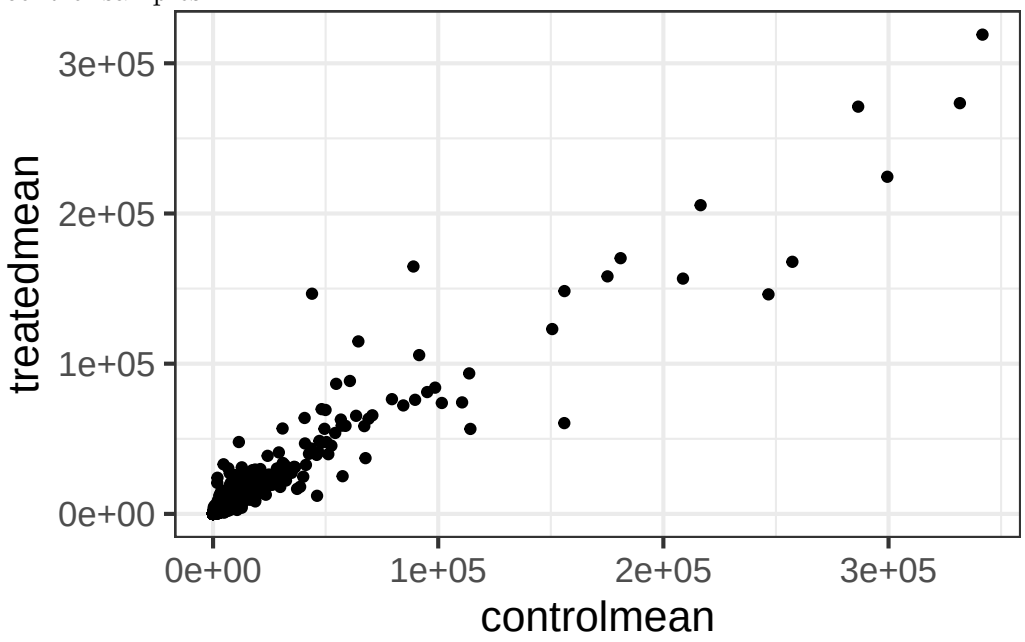
Exercise 3

Directly comparing the raw counts is going to be problematic if we just happened to sequence one group at a higher depth than another. Later on we'll do this analysis properly, normalizing by sequencing depth per sample using a better approach. But for now, `summarize()` the data to show the `sum` of the mean counts across all genes for each group. Your answer should look like this:

```
# A tibble: 1 x 2
  `sum(controlmean)` `sum(treatedmean)`
               <dbl>              <dbl>
1          23005324.          22196524.
```

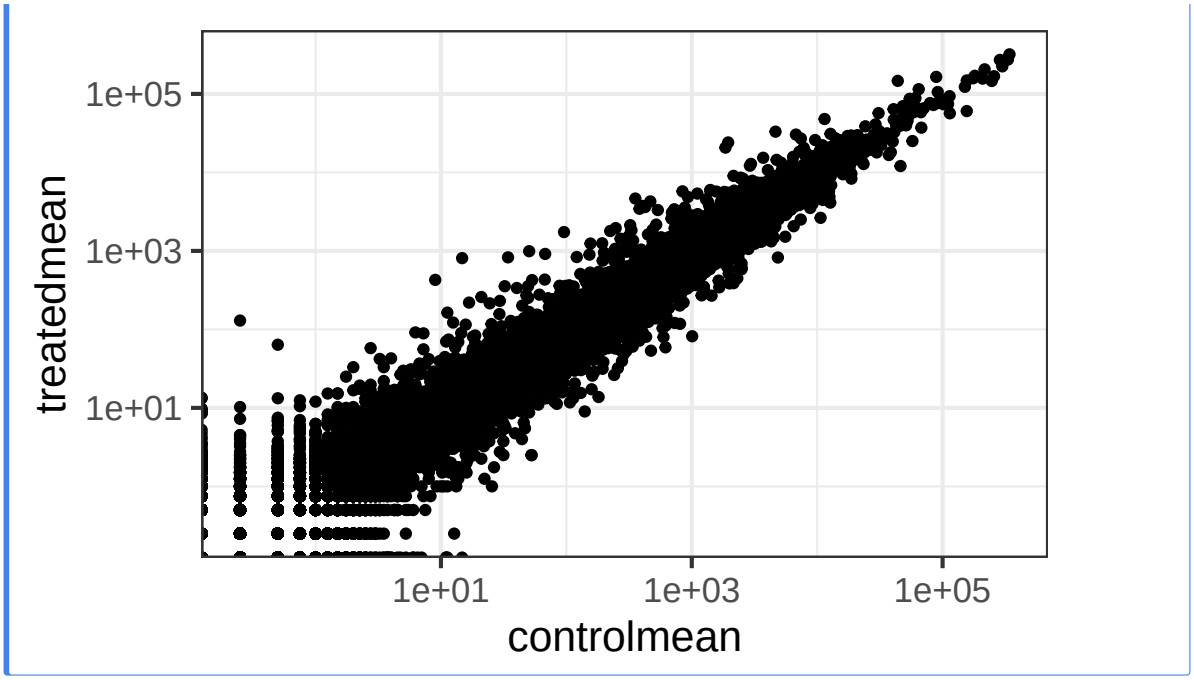
Exercise 4

Create a scatter plot showing the mean of the treated samples against the mean of the control samples.



Exercise 5

Wait a sec. There are 60,000-some rows in this data, but I'm only seeing a few dozen dots at most outside of the big clump around the origin. Try plotting both axes on a log scale (*hint*: `... + scale_..._log10()`)



We can find candidate differentially expressed genes by looking for genes with a large change between control and dex-treated samples. We usually look at the log_2 of the fold change, because this has better mathematical properties. On the absolute scale, upregulation goes from 1 to infinity, while downregulation is bounded by 0 and 1. On the log scale, upregulation goes from 0 to infinity, and downregulation goes from 0 to negative infinity. So, let's mutate our `meancounts` object to add a `log2foldchange` column. Optionally pipe this to `View()`.

```
meancounts |> mutate(log2fc=log2(treatedmean/controlmean))
```

```
# A tibble: 38,694 x 4
   ensgene         controlmean treatedmean    log2fc
   <chr>                 <dbl>       <dbl>     <dbl>
 1 ENSG00000000003      901.         658     -0.453
 2 ENSG00000000005        0            0    NaN
 3 ENSG00000000419      520.         546      0.0690
 4 ENSG00000000457      340.         316.    -0.102
 5 ENSG00000000460       97.2         78.8   -0.304
 6 ENSG00000000938        0.75         0   -Inf
 7 ENSG00000000971     5219         6688.     0.358
 8 ENSG00000001036     2327         1786.    -0.382
 9 ENSG00000001084      756.         578     -0.387
10 ENSG00000001167      528.         348.    -0.600
# i 38,684 more rows
```

There are a couple of "weird" results. Namely, the `NaN` ("not a number") and `-Inf` (negative infinity) results. The `NaN` is returned when you divide by zero and try to take the log. The `-Inf` is returned when you try to take the log of zero. It turns out that there are a lot of genes with zero expression. Let's filter our `meancounts` data, mutate it to add the $log_2(FoldChange)$, and when we're happy with what we see, let's *reassign* the result of that operation *back to the meancounts object*. (*Note: this is destructive. If you're coding interactively like we're doing now, before you do this it's good practice to see what the result of the operation is prior to making the reassignment.*)

```
# Try running the code first, prior to reassigning.
meancounts <- meancounts |>
  filter(controlmean>0 & treatedmean>0) |>
  mutate(log2fc=log2(treatedmean/controlmean))
meancounts
```

```
# A tibble: 21,817 x 4
   ensgene         controlmean treatedmean  log2fc
   <chr>                 <dbl>       <dbl>   <dbl>
 1 ENSG00000000003       901.        658   -0.453
 2 ENSG00000000419       520.        546    0.0690
 3 ENSG00000000457       340.        316.  -0.102
 4 ENSG00000000460        97.2        78.8 -0.304
 5 ENSG00000000971      5219        6688.   0.358
 6 ENSG00000001036      2327        1786.  -0.382
 7 ENSG00000001084       756.        578   -0.387
 8 ENSG00000001167       528.        348.  -0.600
 9 ENSG00000001460       227.        186.  -0.290
10 ENSG00000001461      3170.       2701.  -0.231
# i 21,807 more rows
```

A common threshold used for calling something differentially expressed is a $log_2(FoldChange)$ of greater than 2 or less than -2. Let's filter the dataset both ways to see how many genes are up or down-regulated.

```
meancounts |> filter(log2fc>2)
```

```
# A tibble: 250 x 4
   ensgene         controlmean treatedmean log2fc
   <chr>                 <dbl>       <dbl>  <dbl>
 1 ENSG00000004799      270.       1429.     2.40
 2 ENSG00000006788        2.75       19.8    2.84
```

```
 3 ENSG00000008438        0.5         2.75   2.46
 4 ENSG00000011677        0.5         2.25   2.17
 5 ENSG00000015413        0.5         3      2.58
 6 ENSG00000015592        0.5         2.25   2.17
 7 ENSG00000046653      323         2126.    2.72
 8 ENSG00000070190        0.5         3      2.58
 9 ENSG00000070388        3.5        17.5    2.32
10 ENSG00000074317        0.25        1.75   2.81
# i 240 more rows
```

```
meancounts |> filter(log2fc<(-2))
```

```
# A tibble: 367 x 4
   ensgene         controlmean treatedmean log2fc
   <chr>                 <dbl>       <dbl>  <dbl>
 1 ENSG00000015520       32           6     -2.42
 2 ENSG00000019186       26.5         1.75  -3.92
 3 ENSG00000025423      295          54.2   -2.44
 4 ENSG00000028277       88.2        22     -2.00
 5 ENSG00000029559        1.25        0.25  -2.32
 6 ENSG00000049246      405          93     -2.12
 7 ENSG00000049247        1.25        0.25  -2.32
 8 ENSG00000052344        2.25        0.25  -3.17
 9 ENSG00000054179        3           0.25  -3.58
10 ENSG00000064201       30           6.5   -2.21
# i 357 more rows
```

Exercise 6

Look up help on `?inner_join` or Google around for help for using **dplyr**'s `inner_join()` to join two tables by a common column/key. You downloaded annotables_grch38.csv from the data downloads page. Load this data with `read_csv()` into an object called `anno`. Pipe it to `View()` or click on the object in the Environment pane to view the entire dataset. This table links the unambiguous Ensembl gene ID to things like the gene symbol, full gene name, location, Entrez gene ID, etc.

```
anno <- read_csv("data/annotables_grch38.csv")
anno
```

```
# A tibble: 66,531 x 9
   ensgene         entrez symbol  chr    start    end strand biotype description
   <chr>            <dbl> <chr>   <chr>  <dbl>  <dbl>  <dbl> <chr>   <chr>
```

```
 1 ENSG00000000003   7105 TSPAN6  X     1.01e8 1.01e8     -1 protei~ tetraspani~
 2 ENSG00000000005  64102 TNMD    X     1.01e8 1.01e8      1 protei~ tenomoduli~
 3 ENSG00000000419   8813 DPM1    20    5.09e7 5.10e7     -1 protei~ dolichyl-p~
 4 ENSG00000000457  57147 SCYL3   1     1.70e8 1.70e8     -1 protei~ SCY1-like,~
 5 ENSG00000000460  55732 C1orf1~ 1     1.70e8 1.70e8      1 protei~ chromosome~
 6 ENSG00000000938   2268 FGR     1     2.76e7 2.76e7     -1 protei~ FGR proto-~
 7 ENSG00000000971   3075 CFH     1     1.97e8 1.97e8      1 protei~ complement~
 8 ENSG00000001036   2519 FUCA2   6     1.43e8 1.44e8     -1 protei~ fucosidase~
 9 ENSG00000001084   2729 GCLC    6     5.35e7 5.36e7     -1 protei~ glutamate-~
10 ENSG00000001167   4800 NFYA    6     4.11e7 4.11e7      1 protei~ nuclear tr~
# i 66,521 more rows
```

Exercise 7

Take our newly created `meancounts` object, and `arrange()` it `desc`ending by the absolute value (`abs()`) of the `log2fc` column. The first few rows should look like this:

```
# A tibble: 3 x 4
  ensgene         controlmean treatedmean log2fc
  <chr>                 <dbl>       <dbl>  <dbl>
1 ENSG00000179593        0.25       130.    9.02
2 ENSG00000277196        0.5         63.8   6.99
3 ENSG00000109906       14.8        809.    5.78
```

Exercise 8

Continue on that pipeline, and `inner_join()` it to the `anno` data by the `ensgene` column. Either assign it to a temporary object or pipe the whole thing to `View` to take a look. What do you notice? Would you trust these results? Why or why not?

```
# A tibble: 21,995 x 12
   ensgene      controlmean treatedmean log2fc entrez symbol chr    start    end
   <chr>              <dbl>       <dbl>  <dbl>  <dbl> <chr>  <chr>  <dbl>  <dbl>
 1 ENSG0000017~        0.25      130.     9.02 2.47e2 ALOX1~ 17    8.04e6 8.05e6
 2 ENSG0000027~        0.5        63.8    6.99 1.03e8 AC007~ KI27~ 1.38e5 1.62e5
 3 ENSG0000010~       14.8       809.     5.78 7.70e3 ZBTB16 11    1.14e8 1.14e8
 4 ENSG0000012~       12.8         0.25  -5.67 2.85e3 MCHR1  22    4.07e7 4.07e7
 5 ENSG0000017~        9         427.     5.57 1.02e4 ANGPT~ 1     1.12e7 1.12e7
 6 ENSG0000013~        0.25       10.2    5.36 4.32e3 MMP7   11    1.03e8 1.03e8
 7 ENSG0000024~        0.25        7.25   4.86 5.85e4 LY6G5B CHR_~ 3.17e7 3.17e7
 8 ENSG0000027~        0.5        13.2    4.73 4.40e5 GPR179 17    3.83e7 3.83e7
 9 ENSG0000011~       25.5         1     -4.67 8.45e2 CASQ2  1     1.16e8 1.16e8
```

```
10 ENSG0000012~       34.2       827.     4.59 7.97e4 STEAP4 7     8.83e7 8.83e7
# i 21,985 more rows
# i 3 more variables: strand <dbl>, biotype <chr>, description <chr>
```

12.4 DESeq2 analysis

12.4.1 DESeq2 package

Let's do this the right way. DESeq2 is an R package for analyzing count-based NGS data like RNA-seq. It is available from Bioconductor. Bioconductor is a project to provide tools for analysing high-throughput genomic data including RNA-seq, ChIP-seq and arrays. You can explore Bioconductor packages here.

Bioconductor packages usually have great documentation in the form of *vignettes*. For a great example, take a look at the DESeq2 vignette for analyzing count data. This 40+ page manual is packed full of examples on using DESeq2, importing data, fitting models, creating visualizations, references, etc.

Just like R packages from CRAN, you only need to install Bioconductor packages once (instructions here), then load them every time you start a new R session.

```
library(DESeq2)
citation("DESeq2")
```

Take a second and read through all the stuff that flies by the screen when you load the DESeq2 package. When you first installed DESeq2 it may have taken a while, because DESeq2 *depends* on a number of other R packages (S4Vectors, BiocGenerics, parallel, IRanges, etc.) Each of these, in turn, may depend on other packages. These are all loaded into your working environment when you load DESeq2. Also notice the lines that start with `The following objects are masked from 'package:...`. One example of this is the `rename()` function from the dplyr package. When the S4Vectors package was loaded, it loaded it's own function called `rename()`. Now, if you wanted to use dplyr's rename function, you'll have to call it explicitly using this kind of syntax: `dplyr::rename()`. See this Q&A thread for more.

12.4.2 Importing data

DESeq works on a particular type of object called a DESeqDataSet. The DESeqDataSet is a single object that contains input values, intermediate calculations like how things are normalized, and all results of a differential expression analysis. You can construct a DESeqDataSet from a count matrix, a metadata file, and a formula indicating the design of the experiment.

See the help for `?DESeqDataSetFromMatrix`. If you read through the DESeq2 vignette you'll read about the structure of the data that you need to construct a DESeqDataSet object.

`DESeqDataSetFromMatrix` requires the count matrix (`countData` argument) to be a matrix or numeric data frame. either the row names or the first column of the `countData` must be the identifier you'll use for each gene. The column names of `countData` are the sample IDs, and they must match the row names of `colData` (or the first column when `tidy=TRUE`). `colData` is an additional dataframe describing sample metadata. Both `colData` and `countData` must be regular `data.frame` objects – they can't have the special `tbl` class wrapper created when importing with `readr::read_*`.

countData

gene	ctrl_1	ctrl_2	exp_1	exp_1
geneA	10	11	56	45
geneB	0	0	128	54
geneC	42	41	59	41
geneD	103	122	1	23
geneE	10	23	14	56
geneF	0	1	2	0
…	…	…	…	…
…	…	…	…	…
…	…	…	…	…

colData

id	treatment	sex
ctrl_1	control	male
ctrl_2	control	female
exp_1	treatment	male
exp_2	treatment	female

Sample names:
ctrl_1, ctrl_2, exp_1, exp_2

countData is the count matrix
(number of reads mapping to each gene for each sample)

colData describes metadata about the *columns* of countData

First column of colData must match column names of countData (-1st)

Let's look at our mycounts and metadata again.

```
mycounts
```

```
# A tibble: 38,694 x 9
   ensgene     SRR1039508 SRR1039509 SRR1039512 SRR1039513 SRR1039516 SRR1039517
   <chr>            <dbl>      <dbl>      <dbl>      <dbl>      <dbl>      <dbl>
```

```
 1 ENSG000000~        723        486        904        445       1170       1097
 2 ENSG000000~          0          0          0          0          0          0
 3 ENSG000000~        467        523        616        371        582        781
 4 ENSG000000~        347        258        364        237        318        447
 5 ENSG000000~         96         81         73         66        118         94
 6 ENSG000000~          0          0          1          0          2          0
 7 ENSG000000~       3413       3916       6000       4308       6424      10723
 8 ENSG000000~       2328       1714       2640       1381       2165       2262
 9 ENSG000000~        670        372        692        448        917        807
10 ENSG000000~        426        295        531        178        740        651
# i 38,684 more rows
# i 2 more variables: SRR1039520 <dbl>, SRR1039521 <dbl>
```

```
metadata
```

```
# A tibble: 8 x 4
  id         dex     celltype geo_id
  <chr>      <chr>   <chr>    <chr>
1 SRR1039508 control N61311   GSM1275862
2 SRR1039509 treated N61311   GSM1275863
3 SRR1039512 control N052611  GSM1275866
4 SRR1039513 treated N052611  GSM1275867
5 SRR1039516 control N080611  GSM1275870
6 SRR1039517 treated N080611  GSM1275871
7 SRR1039520 control N061011  GSM1275874
8 SRR1039521 treated N061011  GSM1275875
```

```
class(mycounts)
```

```
[1] "spec_tbl_df" "tbl_df"      "tbl"         "data.frame"
```

```
class(metadata)
```

```
[1] "spec_tbl_df" "tbl_df"      "tbl"         "data.frame"
```

Remember, we read in our count data and our metadata using `read_csv()` which read them in as those "special" dplyr data frames / tbls. We'll need to convert them back to regular data frames for them to work well with DESeq2.

```
mycounts <- as.data.frame(mycounts)
metadata <- as.data.frame(metadata)
head(mycounts)
head(metadata)
class(mycounts)
class(metadata)
```

Let's check that the column names of our count data (except the first, which is `ensgene`) are the same as the IDs from our colData.

```
names(mycounts)[-1]
```

```
[1] "SRR1039508" "SRR1039509" "SRR1039512" "SRR1039513" "SRR1039516"
[6] "SRR1039517" "SRR1039520" "SRR1039521"
```

```
metadata$id
```

```
[1] "SRR1039508" "SRR1039509" "SRR1039512" "SRR1039513" "SRR1039516"
[6] "SRR1039517" "SRR1039520" "SRR1039521"
```

```
names(mycounts)[-1]==metadata$id
```

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

```
all(names(mycounts)[-1]==metadata$id)
```

```
[1] TRUE
```

Now we can move on to constructing the actual DESeqDataSet object. The last thing we'll need to specify is a *design* – a *formula* which expresses how the counts for each gene depend on the variables in colData. Take a look at `metadata` again. The thing we're interested in is the `dex` column, which tells us which samples are treated with dexamethasone versus which samples are untreated controls. We'll specify the design with a tilde, like this: `design=~dex`. (The tilde is the shifted key to the left of the number 1 key on my keyboard. It looks like a little squiggly line). So let's contruct the object and call it `dds`, short for our DESeqDataSet. If you get a warning about "some variables in design formula are characters, converting to factors" don't worry about it. Take a look at the `dds` object once you create it.

```
dds <- DESeqDataSetFromMatrix(countData=mycounts,
                              colData=metadata,
                              design=~dex,
                              tidy=TRUE)
dds
```

```
class: DESeqDataSet
dim: 38694 8
metadata(1): version
assays(1): counts
rownames(38694): ENSG00000000003 ENSG00000000005 ... ENSG00000283120
  ENSG00000283123
rowData names(0):
colnames(8): SRR1039508 SRR1039509 ... SRR1039520 SRR1039521
colData names(4): id dex celltype geo_id
```

12.4.3 DESeq pipeline

Next, let's run the DESeq pipeline on the dataset, and reassign the resulting object back to the same variable. Before we start, `dds` is a bare-bones DESeqDataSet. The `DESeq()` function takes a DESeqDataSet and returns a DESeqDataSet, but with lots of other information filled in (normalization, dispersion estimates, differential expression results, etc). Notice how if we try to access these objects before running the analysis, nothing exists.

```
sizeFactors(dds)
```

```
NULL
```

```
dispersions(dds)
```

```
NULL
```

```
results(dds)
```

```
Error in results(dds): couldn't find results. you should first run DESeq()
```

Here, we're running the DESeq pipeline on the `dds` object, and reassigning the whole thing back to `dds`, which will now be a DESeqDataSet populated with all those values. Get some help on `?DESeq` (notice, no "2" on the end). This function calls a number of other functions within the package to essentially run the entire pipeline (normalizing by library size by estimating the "size factors," estimating dispersion for the negative binomial model, and fitting models and getting statistics for each gene for the design specified when you imported the data).

```
dds <- DESeq(dds)
```

12.4.4 Getting results

Since we've got a fairly simple design (single factor, two groups, treated versus control), we can get results out of the object simply by calling the `results()` function on the DESeqDataSet that has been run through the pipeline. The help page for `?results` and the vignette both have extensive documentation about how to pull out the results for more complicated models (multi-factor experiments, specific contrasts, interaction terms, time courses, etc.).

Note two things:

1. We're passing the `tidy=TRUE` argument, which tells DESeq2 to output the results table with rownames as a first column called 'row.' If we didn't do this, the gene names would be stuck in the row.names, and we'd have a hard time filtering or otherwise using that column.
2. This returns a regular old data frame. Try displaying it to the screen by just typing `res`. You'll see that it doesn't print as nicly as the data we read in with `read_csv`. We can add this "special" attribute to the raw data returned which just tells R to print it nicely.

```
res <- results(dds, tidy=TRUE)
res <- as_tibble(res)
res
```

```
# A tibble: 38,694 x 7
   row             baseMean log2FoldChange  lfcSE   stat  pvalue   padj
   <chr>              <dbl>          <dbl>  <dbl>  <dbl>   <dbl>  <dbl>
 1 ENSG00000000003  747.           -0.351   0.168 -2.08   0.0371  0.163
 2 ENSG00000000005    0            NA      NA     NA     NA      NA
 3 ENSG00000000419  520.            0.206   0.101  2.04   0.0414  0.176
 4 ENSG00000000457  323.            0.0245  0.145  0.169  0.866   0.962
 5 ENSG00000000460   87.7          -0.147   0.257 -0.573  0.567   0.816
 6 ENSG00000000938    0.319        -1.73    3.49  -0.496  0.620  NA
 7 ENSG00000000971 5760.            0.459   0.234  1.96   0.0500  0.201
 8 ENSG00000001036 2025.           -0.228   0.125 -1.83   0.0679  0.247
```

```
 9 ENSG00000001084  652.           -0.253   0.203 -1.25   0.212   0.495
10 ENSG00000001167  412.           -0.534   0.229 -2.33   0.0197  0.105
# i 38,684 more rows
```

Either click on the `res` object in the environment pane or pass it to `View()` to bring it up in a data viewer. Why do you think so many of the adjusted p-values are missing (`NA`)? Try looking at the `baseMean` column, which tells you the average overall expression of this gene, and how that relates to whether or not the p-value was missing. Go to the DESeq2 vignette and read the section about "Independent filtering and multiple testing."

> The goal of independent filtering is to filter out those tests from the procedure that have no, or little chance of showing significant evidence, without even looking at the statistical result. Genes with very low counts are not likely to see significant differences typically due to high dispersion. This results in increased detection power at the same experiment-wide type I error [*i.e., better FDRs*].

Exercise 9

Using a `|>`, `arrange` the results by the adjusted p-value.

```
# A tibble: 38,694 x 7
   row             baseMean log2FoldChange  lfcSE  stat   pvalue     padj
   <chr>              <dbl>          <dbl>  <dbl> <dbl>    <dbl>    <dbl>
 1 ENSG00000152583     955.           4.37 0.237   18.4 8.74e-76 1.32e-71
 2 ENSG00000179094     743.           2.86 0.176   16.3 8.11e-60 6.14e-56
 3 ENSG00000116584    2278.          -1.03 0.0651 -15.9 6.93e-57 3.50e-53
 4 ENSG00000189221    2384.           3.34 0.212   15.7 9.14e-56 3.46e-52
 5 ENSG00000120129    3441.           2.97 0.204   14.6 5.26e-48 1.59e-44
 6 ENSG00000148175   13494.           1.43 0.100   14.2 7.25e-46 1.83e-42
 7 ENSG00000178695    2685.          -2.49 0.178  -14.0 2.11e-44 4.57e-41
 8 ENSG00000109906     440.           5.93 0.428   13.8 1.36e-43 2.58e-40
 9 ENSG00000134686    2934.           1.44 0.106   13.6 4.05e-42 6.82e-39
10 ENSG00000101347   14135.           3.85 0.285   13.5 1.25e-41 1.90e-38
# i 38,684 more rows
```

Exercise 10

Continue piping to `inner_join()`, joining the results to the `anno` object. See the help for `?inner_join`, specifically the `by=` argument. You'll have to do something like `... |> inner_join(anno, by=c("row"="ensgene"))`. Once you're happy with this result, reassign the result back to `res`. It'll look like this.

```
              row   baseMean log2FoldChange      lfcSE      stat       pvalue
1 ENSG00000152583   954.7709       4.368359 0.23712679  18.42204 8.744898e-76
```

```
2 ENSG00000179094   743.2527       2.863889 0.17556931  16.31201 8.107836e-60
3 ENSG00000116584  2277.9135      -1.034701 0.06509844 -15.89440 6.928546e-57
4 ENSG00000189221  2383.7537       3.341544 0.21240579  15.73189 9.144326e-56
5 ENSG00000120129  3440.7038       2.965211 0.20369513  14.55710 5.264243e-48
6 ENSG00000148175 13493.9204       1.427168 0.10038904  14.21638 7.251278e-46
          padj entrez  symbol chr     start       end strand        biotype
1 1.324415e-71   8404 SPARCL1   4  87473335  87531061     -1 protein_coding
2 6.139658e-56   5187    PER1  17   8140472   8156506     -1 protein_coding
3 3.497761e-53   9181 ARHGEF2   1 155946851 156007070     -1 protein_coding
4 3.462270e-52   4128    MAOA   X  43654907  43746824      1 protein_coding
5 1.594539e-44   1843   DUSP1   5 172768090 172771195     -1 protein_coding
6 1.830344e-42   2040    STOM   9 121338988 121370304     -1 protein_coding
                                                                        description
1                                SPARC-like 1 (hevin) [Source:HGNC Symbol;Acc:HGNC:11220]
2                              period circadian clock 1 [Source:HGNC Symbol;Acc:HGNC:8845]
3 Rho/Rac guanine nucleotide exchange factor (GEF) 2 [Source:HGNC Symbol;Acc:HGNC:682]
4                                  monoamine oxidase A [Source:HGNC Symbol;Acc:HGNC:6833]
5                       dual specificity phosphatase 1 [Source:HGNC Symbol;Acc:HGNC:3064]
6                                            stomatin [Source:HGNC Symbol;Acc:HGNC:3383]
```

Exercise 11

How many are significant with an adjusted p-value <0.05? (Pipe to `filter()`).

```
[1] 2186
```

Exercise 12

Finally, let's write out the significant results. See the help for `?write_csv`, which is part of the **readr** package (note: this is *not* the same as `write.csv` with a dot.). We can continue that pipe and write out the significant results to a file like so:

```
res |>
  filter(padj<0.05) |>
  write_csv("sigresults.csv")
```

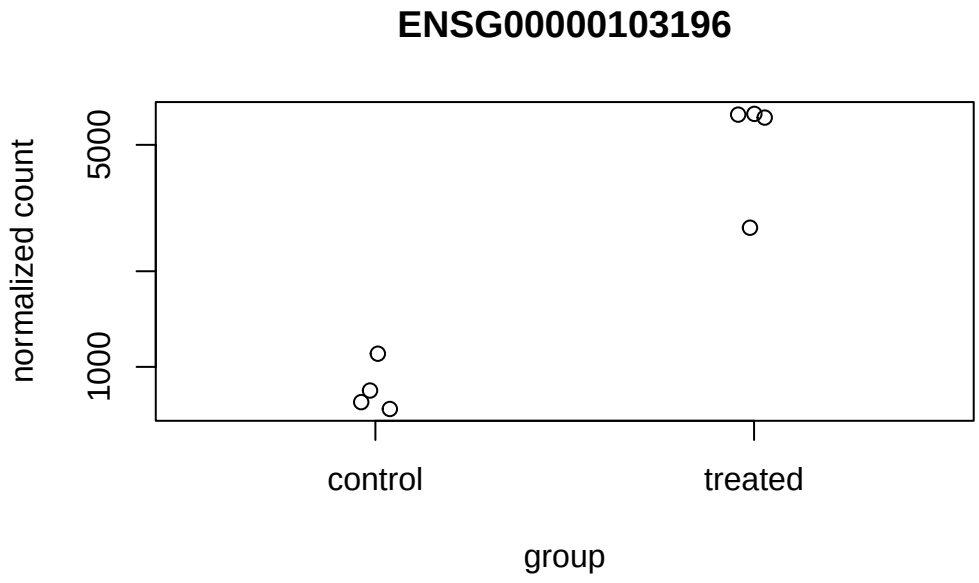
You can open this file in Excel or any text editor (try it now).

12.5 Data Visualization

12.5.1 Plotting counts

DESeq2 offers a function called `plotCounts()` that takes a DESeqDataSet that has been run through the pipeline, the name of a gene, and the name of the variable in the `colData` that you're interested in, and plots those values. See the help for `?plotCounts`. Let's first see what the gene ID is for the CRISPLD2 gene using `res |> filter(symbol=="CRISPLD2")`. Now, let's plot the counts, where our `intgroup`, or "interesting group" variable is the "dex" column.

```
plotCounts(dds, gene="ENSG00000103196", intgroup="dex")
```



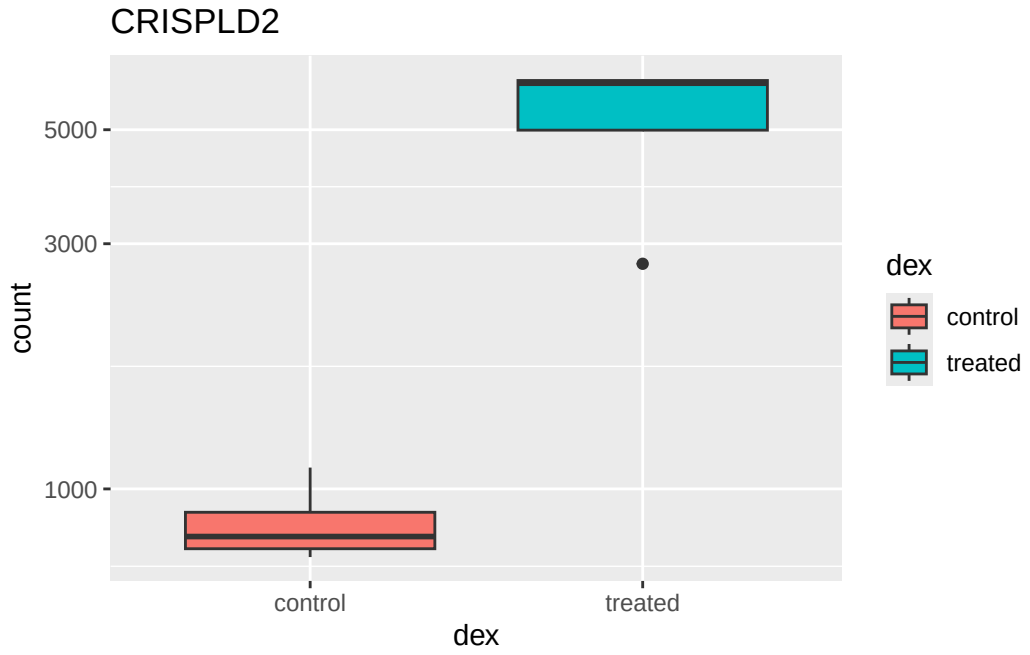
That's just okay. Keep looking at the help for `?plotCounts`. Notice that we could have actually returned the data instead of plotting. We could then pipe this to ggplot and make our own figure. Let's make a boxplot.

```
# Return the data
plotCounts(dds, gene="ENSG00000103196", intgroup="dex", returnData=TRUE)
```

```
               count     dex
SRR1039508  774.5002 control
SRR1039509 6258.7915 treated
SRR1039512 1100.2741 control
```

```
SRR1039513 6093.0324 treated
SRR1039516  736.9483 control
SRR1039517 2742.1908 treated
SRR1039520  842.5452 control
SRR1039521 6224.9923 treated
```

```
# Plot it
plotCounts(dds, gene="ENSG00000103196", intgroup="dex", returnData=TRUE) |>
  ggplot(aes(dex, count)) + geom_boxplot(aes(fill=dex)) + scale_y_log10() + ggtitle("CRISP
```



12.5.2 MA & Volcano plots

Let's make some commonly produced visualizations from this data. First, let's mutate our results object to add a column called `sig` that evaluates to `TRUE` if padj<0.05, and FALSE if not, and NA if padj is also NA.

```
# Create the new column
res <- res |> mutate(sig=padj<0.05)

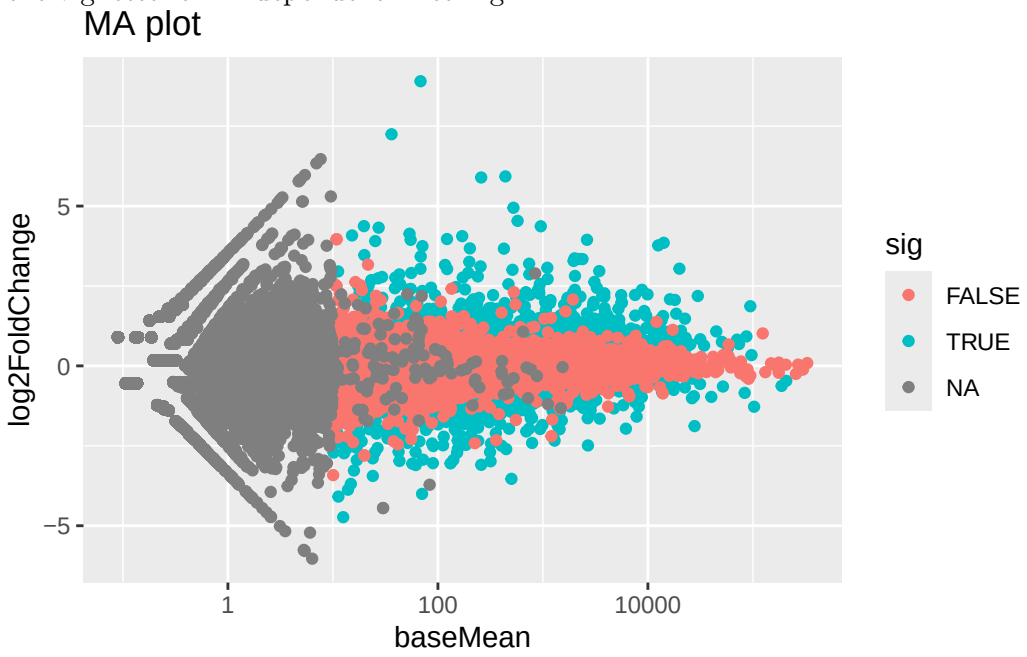
# How many of each?
res |>
  group_by(sig) |>
```

```
    summarize(n=n())
```

```
# A tibble: 3 x 2
  sig       n
  <lgl> <int>
1 FALSE 13106
2 TRUE   2186
3 NA    23665
```

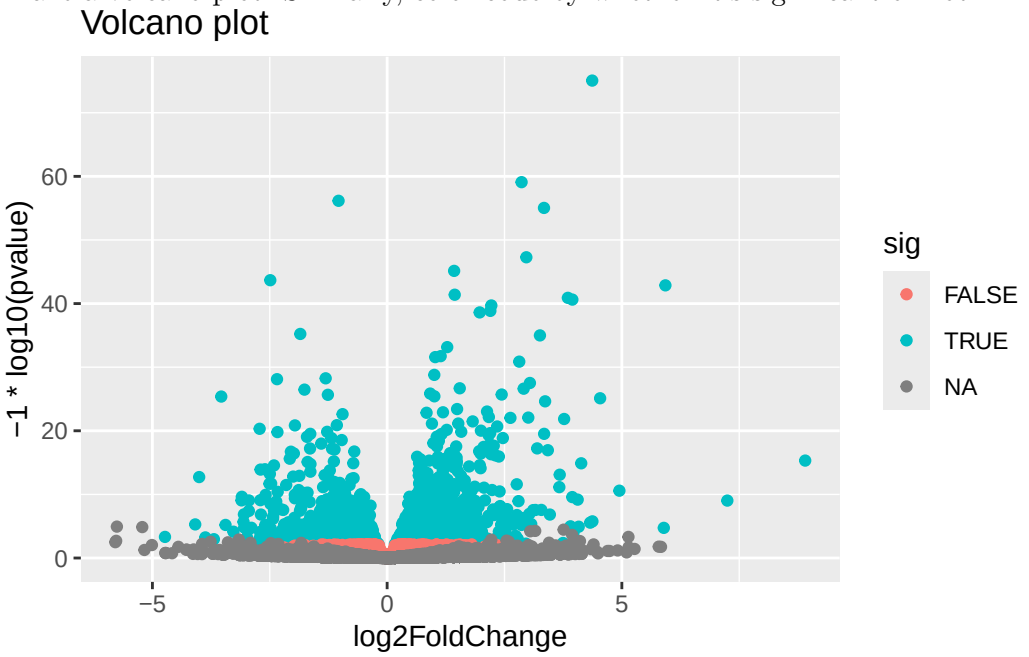
Exercise 13

Look up the Wikipedia articles on MA plots and volcano plots. An MA plot shows the average expression on the X-axis and the log fold change on the y-axis. A volcano plot shows the log fold change on the X-axis, and the $-log_{10}$ of the p-value on the Y-axis (the more significant the p-value, the larger the $-log_{10}$ of that value will be).
Make an MA plot. Use a log_{10}-scaled x-axis, color-code by whether the gene is significant, and give your plot a title. It should look like this. What's the deal with the gray points? Why are they missing? Go to the DESeq2 website on Bioconductor and look through the vignette for "Independent Filtering."



Exercise 14

Make a volcano plot. Similarly, color-code by whether it's significant or not.



12.5.3 Transformation

To test for differential expression we operate on raw counts. But for other downstream analyses like heatmaps, PCA, or clustering, we need to work with transformed versions of the data, because it's not clear how to best compute a distance metric on untransformed counts. The go-to choice might be a log transformation. But because many samples have a zero count (and $log(0) = -\infty$, you might try using pseudocounts, i. e. $y = log(n + 1)$ or more generally, $y = log(n + n_0)$, where n represents the count values and n_0 is some positive constant.

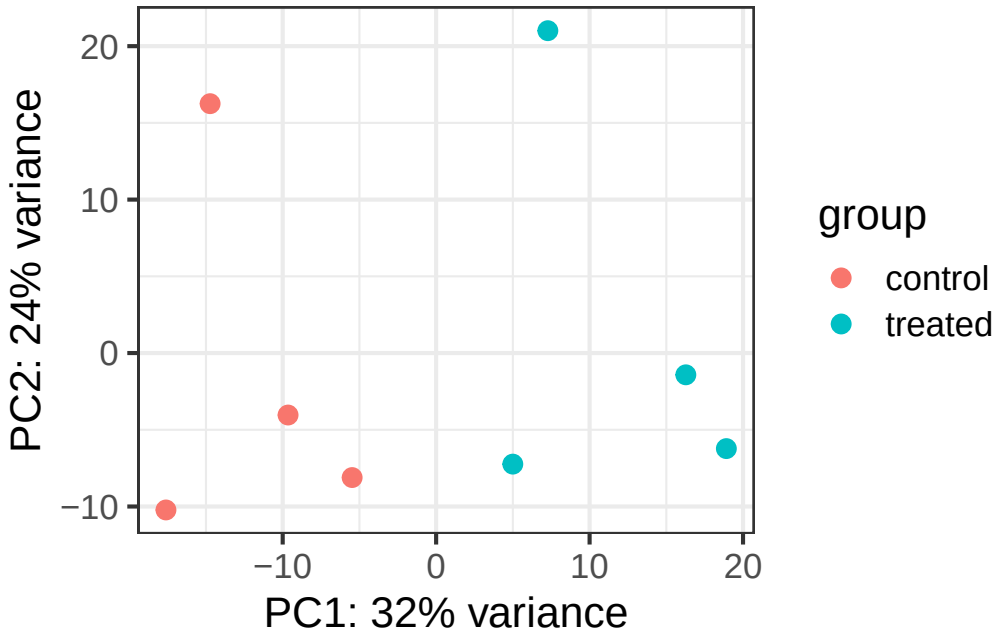
But there are other approaches that offer better theoretical justification and a rational way of choosing the parameter equivalent to n_0, and they produce transformed data on the log scale that's normalized to library size. One is called a *variance stabilizing transformation* (VST), and it also removes the dependence of the variance on the mean, particularly the high variance of the log counts when the mean is low.

```
vsdata <- vst(dds, blind=FALSE)
```

12.5.4 PCA

Let's do some exploratory plotting of the data using principal components analysis on the variance stabilized data from above. Let's use the DESeq2-provided `plotPCA` function. See the help for `?plotPCA` and notice that it also has a `returnData` option, just like `plotCounts`.

```
plotPCA(vsdata, intgroup="dex")
```

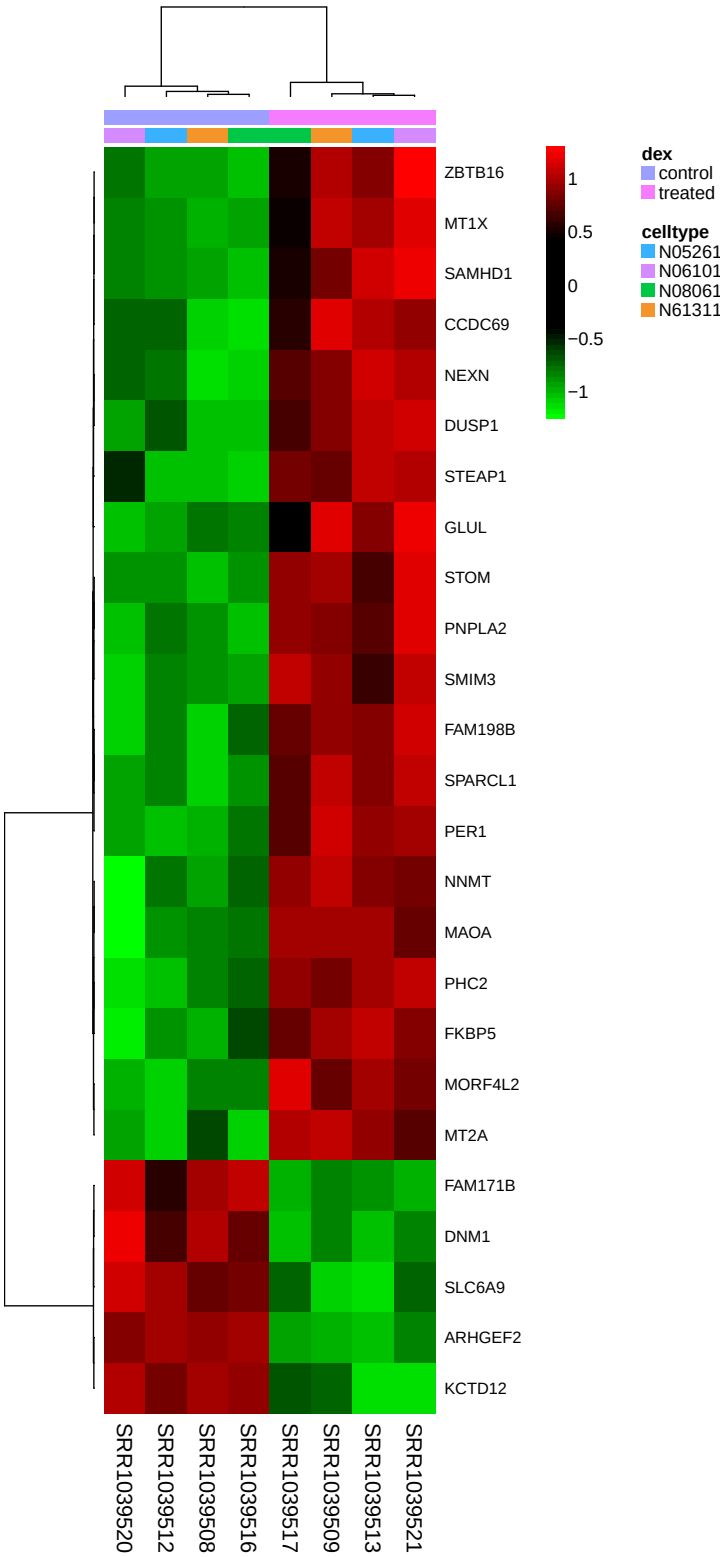


Principal Components Analysis (PCA) is a dimension reduction and visualization technique that is here used to project the multivariate data vector of each sample into a two-dimensional plot, such that the spatial arrangement of the points in the plot reflects the overall data (dis)similarity between the samples. In essence, principal component analysis distills all the global variation between samples down to a few variables called *principal components*. The majority of variation between the samples can be summarized by the first principal component, which is shown on the x-axis. The second principal component summarizes the residual variation that isn't explained by PC1. PC2 is shown on the y-axis. The percentage of the global variation explained by each principal component is given in the axis labels. In a two-condition scenario (e.g., mutant vs WT, or treated vs control), you might expect PC1 to separate the two experimental conditions, so for example, having all the controls on the left and all experimental samples on the right (or vice versa - the units and directionality isn't important). The secondary axis may separate other aspects of the design - cell line, time point, etc. Very often the experimental design is reflected in the PCA plot, and in this case, it is. But this kind of diagnostic can be useful for finding outliers, investigating batch effects, finding sample swaps,

and other technical problems with the data. This YouTube video from the Genetics Department at UNC gives a very accessible explanation of what PCA is all about in the context of a gene expression experiment, without the need for an advanced math background. Take a look.

12.5.5 Bonus: Heatmaps

Heatmaps are complicated, and are often poorly understood. It's a type of visualization used very often in high-throughput biology where data are clustered on rows and columns, and the actual data is displayed as tiles on a grid, where the values are mapped to some color spectrum. Our R useRs group MeetUp had a session on making heatmaps, which I summarized in this blog post. Take a look at the code from that meetup, and the documentation for the `aheatmap` function in the NMF package to see if you can re-create this image. Here, I'm clustering all samples using the top 25 most differentially regulated genes, labeling the rows with the gene symbol, and putting two annotation color bars across the top of the main heatmap panel showing treatment and cell line annotations from our metadata.


ZBTB16
MT1X
SAMHD1
CCDC69
NEXN
DUSP1
STEAP1
GLUL
STOM
PNPLA2
SMIM3
FAM198B
SPARCL1
PER1
NNMT
MAOA
PHC2
FKBP5
MORF4L2
MT2A
FAM171B
DNM1
SLC6A9
ARHGEF2
KCTD12
SRR1039520
SRR1039512
SRR1039508
SRR1039516
SRR1039517
SRR1039509
SRR1039513
SRR1039521
1
0.5
0
−0.5
−1
dex
control
treated
celltype
N05261
N06101
N08061
N61311

12.6 Record `sessionInfo()`

The `sessionInfo()` prints version information about R and any attached packages. It's a good practice to always run this command at the end of your R session and record it for the sake of reproducibility in the future.

```
sessionInfo()
```

```
R version 4.4.1 (2024-06-14)
Platform: aarch64-apple-darwin20
Running under: macOS Sonoma 14.3

Matrix products: default
BLAS:   /Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/lib/libRblas.0.dylib
LAPACK: /Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/lib/libRlapack.dylib;  L

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

time zone: America/New_York
tzcode source: internal

attached base packages:
[1] stats4    stats     graphics  grDevices utils     datasets  methods
[8] base

other attached packages:
 [1] DESeq2_1.44.0               SummarizedExperiment_1.34.0
 [3] Biobase_2.64.0              MatrixGenerics_1.16.0
 [5] matrixStats_1.3.0           GenomicRanges_1.56.1
 [7] GenomeInfoDb_1.40.1         IRanges_2.38.1
 [9] S4Vectors_0.42.1            BiocGenerics_0.50.0
[11] dplyr_1.1.4                 readr_2.1.5
[13] ggplot2_3.5.1

loaded via a namespace (and not attached):
 [1] gtable_0.3.5            xfun_0.46               lattice_0.22-6
 [4] tzdb_0.4.0              vctrs_0.6.5             tools_4.4.1
 [7] generics_0.1.3          parallel_4.4.1          tibble_3.2.1
[10] fansi_1.0.6             cluster_2.1.6           pkgconfig_2.0.3
[13] Matrix_1.7-0            RColorBrewer_1.1-3      rngtools_1.5.2
[16] lifecycle_1.0.4         GenomeInfoDbData_1.2.12 stringr_1.5.1
```

```
[19] compiler_4.4.1          farver_2.1.2            munsell_0.5.1
[22] tinytex_0.52            codetools_0.2-20        htmltools_0.5.8.1
[25] yaml_2.3.10             pillar_1.9.0            crayon_1.5.3
[28] BiocParallel_1.38.0     DelayedArray_0.30.1     iterators_1.0.14
[31] foreach_1.5.2           abind_1.4-5             tidyselect_1.2.1
[34] locfit_1.5-9.10         digest_0.6.36           stringi_1.8.4
[37] reshape2_1.4.4          labeling_0.4.3          fastmap_1.2.0
[40] grid_4.4.1              colorspace_2.1-1        cli_3.6.3
[43] SparseArray_1.4.8       magrittr_2.0.3          S4Arrays_1.4.1
[46] utf8_1.2.4              withr_3.0.1             scales_1.3.0
[49] UCSC.utils_1.0.0        bit64_4.0.5             registry_0.5-1
[52] rmarkdown_2.28          XVector_0.44.0          httr_1.4.7
[55] bit_4.0.5               hms_1.1.3               evaluate_0.24.0
[58] knitr_1.48              doParallel_1.0.17       NMF_0.28
[61] rlang_1.1.4             Rcpp_1.0.13             gridBase_0.4-7
[64] glue_1.7.0              BiocManager_1.30.25     rstudioapi_0.16.0
[67] vroom_1.6.5             jsonlite_1.8.8          plyr_1.8.9
[70] R6_2.5.1                zlibbioc_1.50.0
```

12.7 Pathway Analysis

Pathway analysis or *gene set analysis* means many different things, general approaches are nicely reviewed in: Khatri, et al. "Ten years of pathway analysis: current approaches and outstanding challenges." *PLoS Comput Biol* 8.2 (2012): e1002375.

There are many freely available tools for pathway or over-representation analysis. Bioconductor alone has over 70 packages categorized under *gene set enrichment* and over 100 packages categorized under *pathways*. I wrote this tutorial in 2015 showing how to use the GAGE (Generally Applicable Gene set Enrichment)[2] package to do KEGG pathway enrichment analysis on differential expression results.

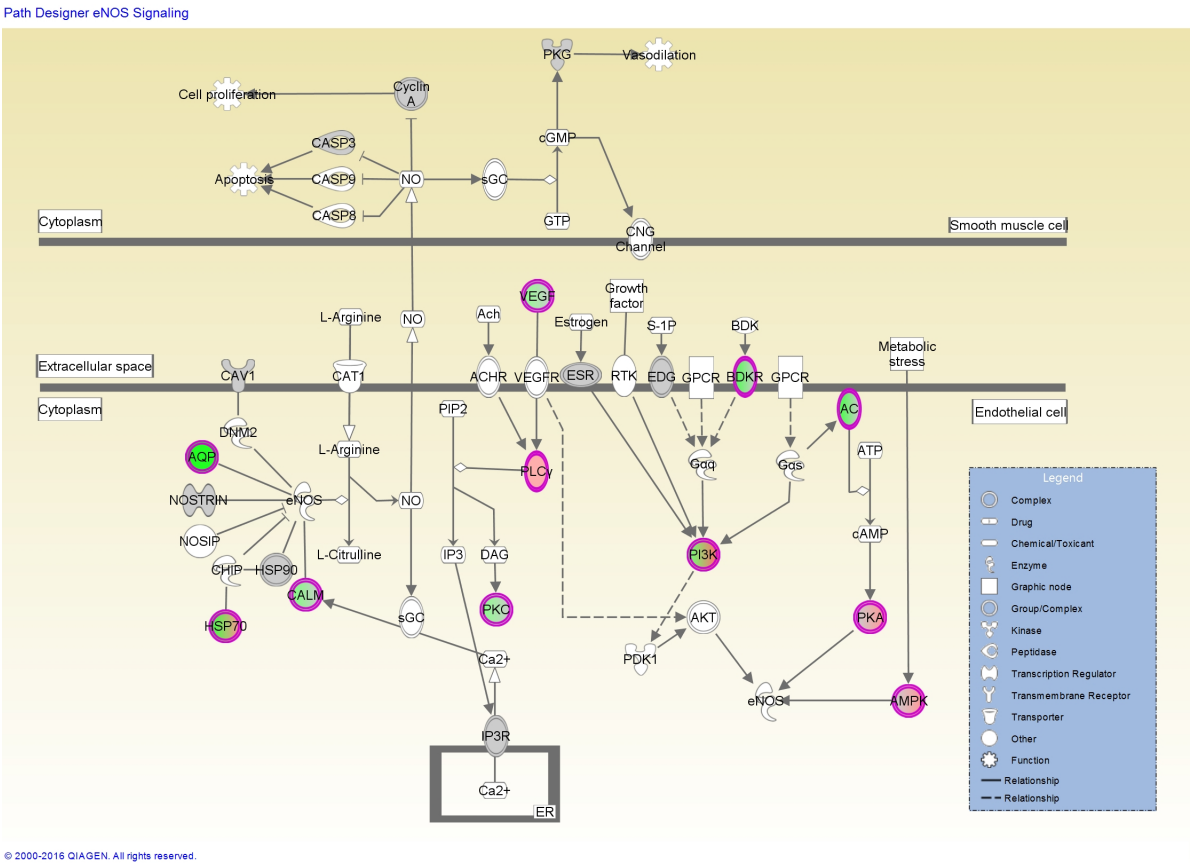
While there are many freely available tools to do this, and some are truly fantastic, many of them are poorly maintained or rarely updated. The DAVID tool that a lot of folks use wasn't updated at all between Jan 2010 and Oct 2016.

UVA has a site license to Ingenuity Pathway Analysis. Statistically, IPA isn't doing anything revolutionary methodologically, but the real value comes in with its (1) ease of use, and (2) highly curated knowledgebase. You can get access to IPA through the Health Sciences Library at this link, and there are also links to UVA support resources for using IPA.

[2] Luo, W. et al., 2009. GAGE: generally applicable gene set enrichment for pathway analysis. *BMC bioinformatics*, 10:161. Package: bioconductor.org/packages/gage.

This summary report is the first thing you would get out of IPA after running a core analysis on the results of this analysis. Open it up and take a look.

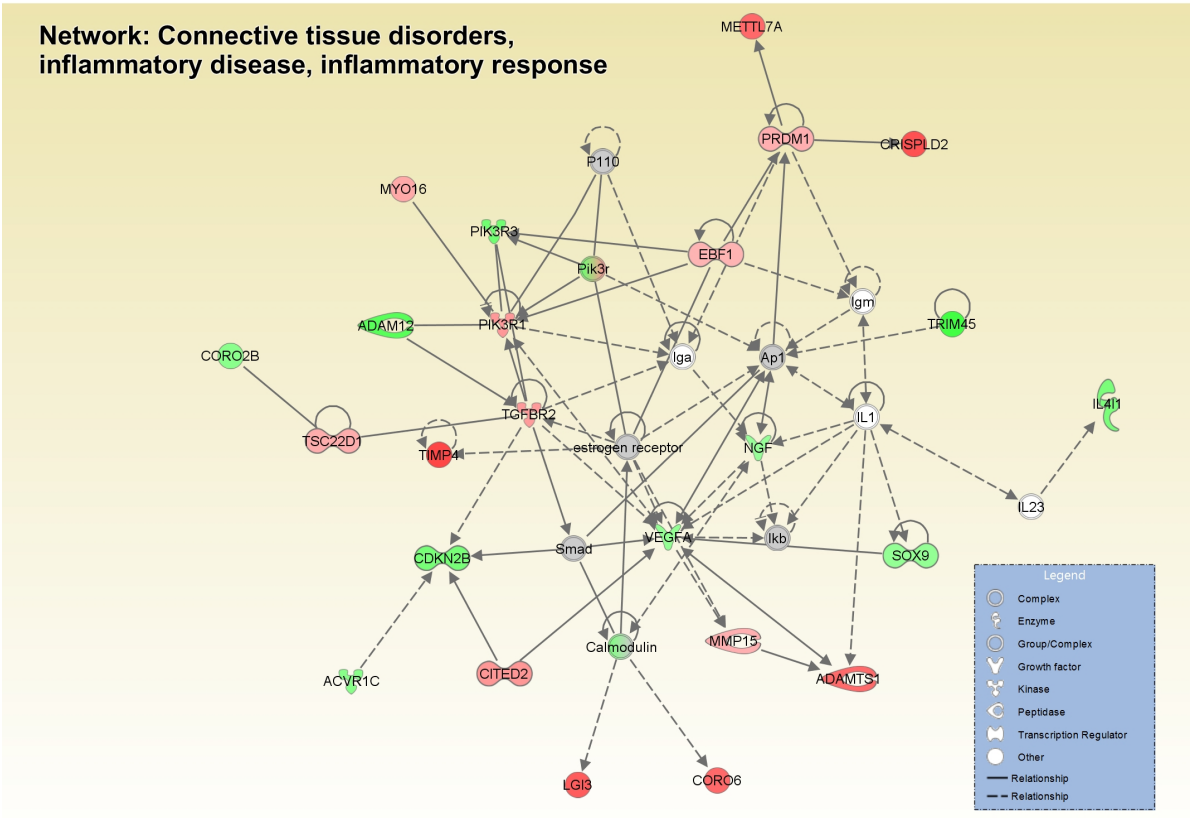
It shows, among other things, that the endothelial nitric-oxide synthase signaling pathway is highly over-represented among the most differentially expressed genes. For this, or any pathway you're interested in, IPA will give you a report like this one for eNOS with a very detailed description of the pathway, what kind of diseases it's involved in, which molecules are in the pathway, what drugs might perturb the pathway, and more. If you're logged into IPA, clicking any of the links will take you to IPA's knowledge base where you can learn more about the connection between that molecule, the pathway, and a disease, and further overlay any of your gene expression data on top of the pathway.



The report also shows us some upstream regulators, which serves as a great positive control that this stuff actually works, because it's inferring that dexamethasone might be an upstream regulator based on the target molecules that are dysregulated in our data.

You can also start to visualize networks in the context of biology and how your gene expression data looks in those molecules. Here's a network related to "Connective Tissue Disorders, Inflammatory Disease, Inflammatory Response" showing dysregulation of some of the genes in our data.

Path Designer Network 5



13 Visualizing and Annotating Phylogenetic Trees

This chapter demonstrates how to use **ggtree**, an extension of the ggplot2 package to visualize and annotate phylogenetic trees. Many of the examples here were modified from the ggtree vignettes.

This chapter does *not* cover methods and software for *generating* phylogenetic trees, nor does it it cover *interpreting* phylogenies. **Here's a quick primer on how to read a phylogeny** that you should definitely review prior to this chapter, but it is by no means extensive. Genome-wide sequencing allows for examination of the entire genome, and from this, many methods and software tools exist for comparative genomics using SNP- and gene-based phylogenetic analysis, either from unassembled sequencing reads, draft assemblies/contigs, or complete genome sequences. These methods are beyond the scope of this chapter.

13.1 The `ggtree` Package

ggtree is an R package that extends ggplot2 for visualizating and annotating phylogenetic trees with their covariates and other associated data. It is available from Bioconductor. Bioconductor is a project to provide tools for analyzing and annotating various kinds of genomic data. You can search and browse Bioconductor packages here.

1. **ggtree Bioconductor page**: bioconductor.org/packages/ggtree.
2. **ggtree homepage**: guangchuangyu.github.io/ggtree (contains more information about the package, more documentation, a gallery of beautiful images, and links to related resources).
3. **ggtree publication**: Yu, Guangchuang, et al. "ggtree: an r package for visualization and annotation of phylogenetic trees with their covariates and other associated data." *Methods in Ecology and Evolution* (2016) DOI:10.1111/2041-210X.12628.

Bioconductor packages usually have great documentation in the form of *vignettes*. Take a look at the landing page for ggtree – about halfway down the page under the "Documentation" heading there are multiple walkthrough tutorials directed to different applications and functionalities of ggtree, chock full of runnable examples and explanations.

```
library(ggtree)
```

A note on masked functions: If you already loaded a package like **dplyr**, take a second and look through some of the output that you see when you load **ggtree** after **dplyr**. When you first installed ggtree it may have taken a while, because ggtree *depends* on a number of other R packages. Each of these, in turn, may depend on other packages. These are all loaded into your working environment when you load ggtree. Also notice the lines that start with `The following objects are masked from 'package:...`. One example of this is the `collapse()` function from dplyr. When ggtree was loaded, it loaded it's own function called `collapse()`. Now, if you wanted to use dplyr's collapse function, you'll have to call it explicitly using this kind of syntax: `dplyr::collapse()`. See this Q&A thread for more.

13.2 Tree Import

From the ggtree landing page take a look at the Tree Data Import vignette. There are many different software packages for creating phylogenetic trees from different types of data, and there are many formats for storing the resulting phylogenetic trees they produce.

Most tree viewer software (including `R` packages) focus on **Newick** and **Nexus** file formats, and other evolution analysis software might also contain supporting evidence within the file that are ready for annotating a phylogenetic tree. ggtree supports several file formats, including:

- Newick
- Nexus
- Phylip
- Jplace
- New Hampshire eXtended format (NHX)

and software output from:

- BEAST
- EPA
- HYPHY
- PAML
- PHYLDOG
- pplacer
- r8s
- RAxML
- RevBayes

The `ggtree` package implement several parser functions, including:

- `read.tree` for reading Newick files.

- `read.phylip` for reading Phylip files.
- `read.jplace` for reading Jplace files.
- `read.nhx` for reading NHX files.
- `read.beast` for parsing output of BEAST
- `read.codeml` for parsing output of CODEML (`rst` and `mlc` files)
- `read.codeml_mlc` for parsing `mlc` file (output of `CODEML`)
- `read.hyphy` for parsing output of HYPHY
- `read.jplace` for parsing `jplace` file including output from EPA and pplacer
- `read.nhx` for parsing `NHX` file including output from PHYLODOG and RevBayes
- `read.paml_rst` for parsing `rst` file (output of `BASEML` and `CODEML`)
- `read.r8s` for parsing output of r8s
- `read.raxml` for parsing output of RAxML

13.3 Basic trees

Let's first import our tree data. We're going to work with a made-up phylogeny with 13 samples ("tips"). Download the **tree_newick.nwk** data by clicking here or using the link above. Let's load the libraries you'll need if you haven't already, and then import the tree using `read.tree()`. Displaying the object itself really isn't useful. The output just tells you a little bit about the tree itself.

```
library(ggtree)

tree <- read.tree("data/tree_newick.nwk")
tree
```

```
Phylogenetic tree with 13 tips and 12 internal nodes.

Tip labels:
  A, B, C, D, E, F, ...

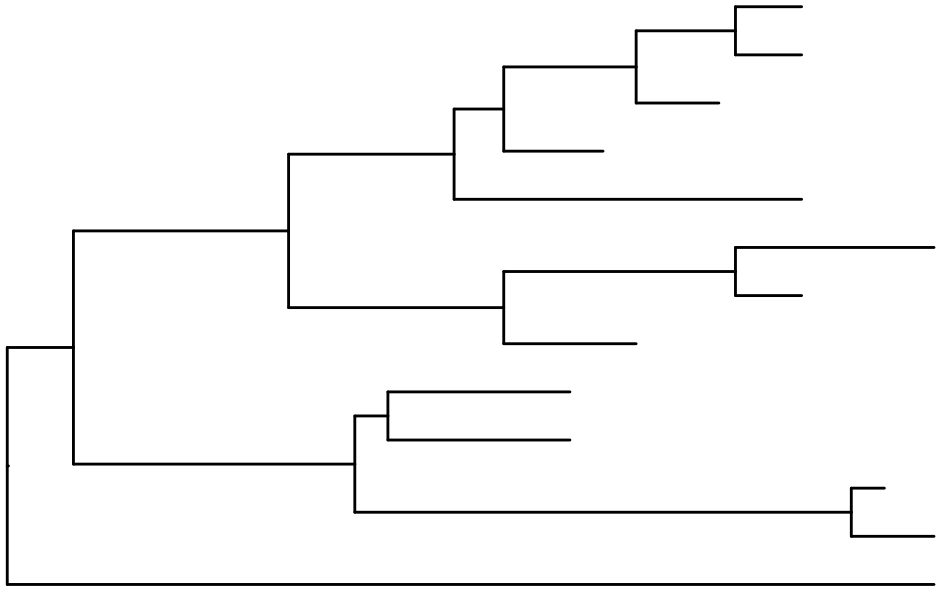
Rooted; includes branch lengths.
```

Just like with ggplot2 we created a basic canvas with `ggplot(...)` and added layers with `+geom_???()`, we can do the same here. The ggtree package gives us a `geom_tree()` function. Because ggtree is built on top of ggplot2, you get ggplot2's default gray theme with white lines. You can override this with a theme from the ggtree package.

Because you'll almost always want to add a tree geom and remove the default background and axes, the `ggtree()` function is essentially a shortcut for `ggplot(...) + geom_tree() + theme_tree()`.

```
# build a ggplot with a geom_tree
ggplot(tree) + geom_tree() + theme_tree()

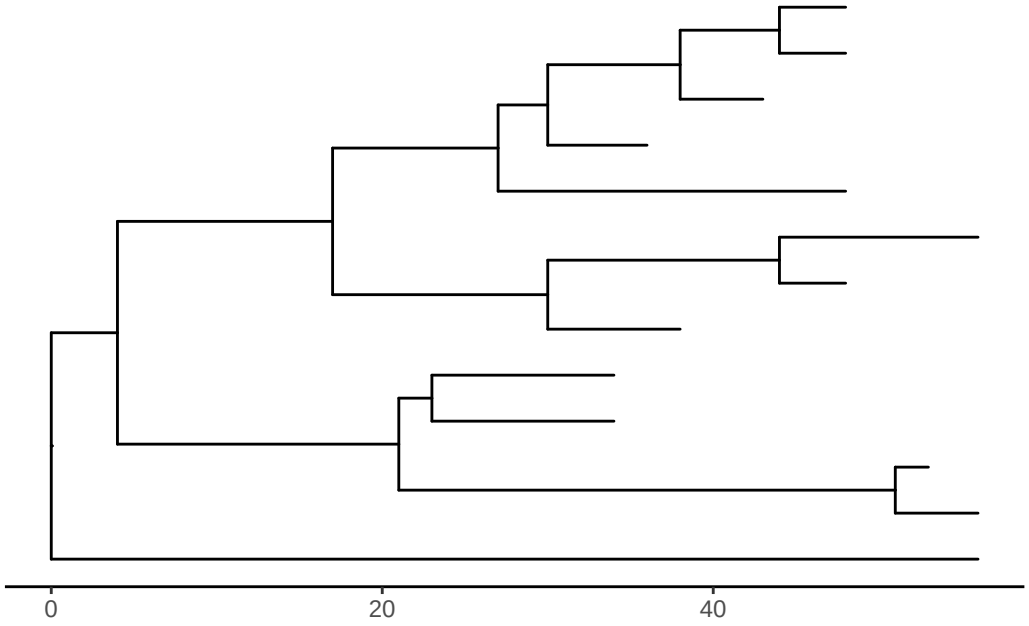
# This is convenient shorthand
ggtree(tree)
```



There's also the treescale geom, which adds a scale bar, or alternatively, you can change the default `ggtree()` theme to `theme_tree2()`, which adds a scale on the x-axis. The horizontal dimension in this plot shows the amount of genetic change, and the branches and represent evolutionary lineages changing over time. The longer the branch in the horizonal dimension, the larger the amount of change, and the scale tells you this. The units of branch length are usually nucleotide substitutions per site – that is, the number of changes or substitutions divided by the length of the sequence (alternatively, it could represent the percent change, i.e., the number of changes per 100 bases). See this article for more.

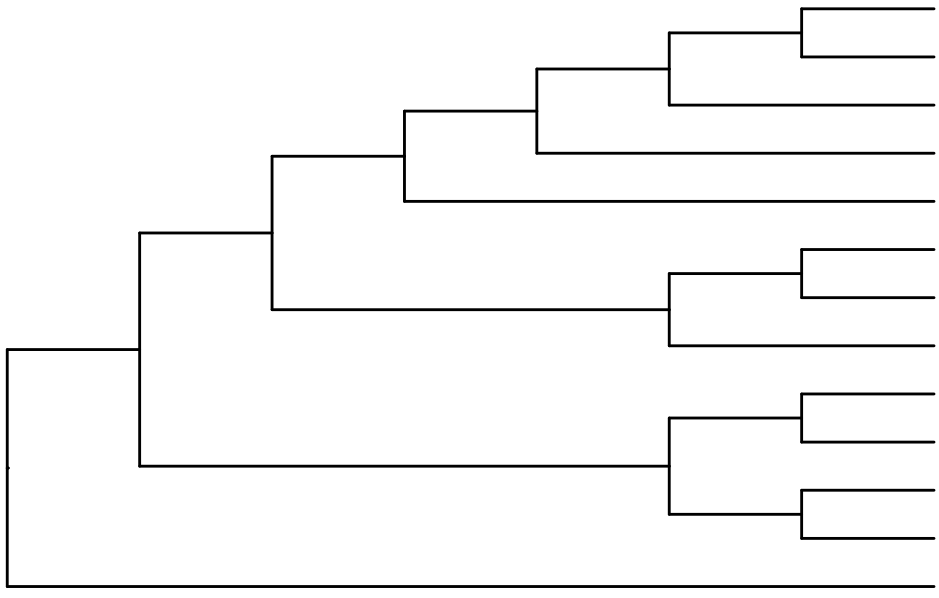
```
# add a scale
ggtree(tree) + geom_treescale()

# or add the entire scale to the x axis with theme_tree2()
ggtree(tree) + theme_tree2()
```



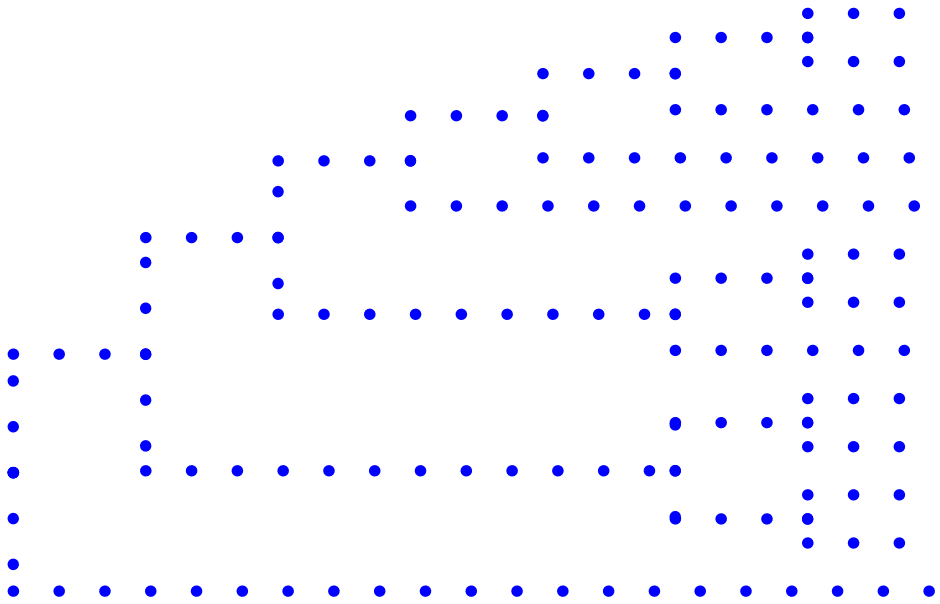
The default is to plot a phylogram, where the x-axis shows the genetic change / evolutionary distance. If you want to disable scaling and produce a cladogram instead, set the `branch.length="none"` option inside the `ggtree()` call. See `?ggtree` for more.

```
ggtree(tree, branch.length="none")
```



The `...` option in the help for `?ggtree` represents additional options that are further passed to `ggplot()`. You can use this to change aesthetics of the plot. Let's draw a cladogram (no branch scaling) using thick blue dotted lines (note that I'm not mapping these aesthetics to features of the data with `aes()` – we'll get to that later).

```
ggtree(tree, branch.length="none", color="blue", size=2, linetype=3)
```



Exercise 1

Look at the help again for `?ggtree`, specifically at the `layout=` option. By default, it produces a rectangular layout.

1. Create a slanted phylogenetic tree.
2. Create a circular phylogenetic tree.
3. Create a circular unscaled cladogram with thick red lines.

13.3.1 Other tree geoms

Let's add additional layers. As we did in the visualization section (Chapter 5), we can create a plot object, e.g., `p`, to store the basic layout of a ggplot, and add more layers to it as we desire. Let's add node and tip points. Let's finally label the tips.

```
# create the basic plot
p <- ggtree(tree)

# add node points
p + geom_nodepoint()

# add tip points
p + geom_tippoint()

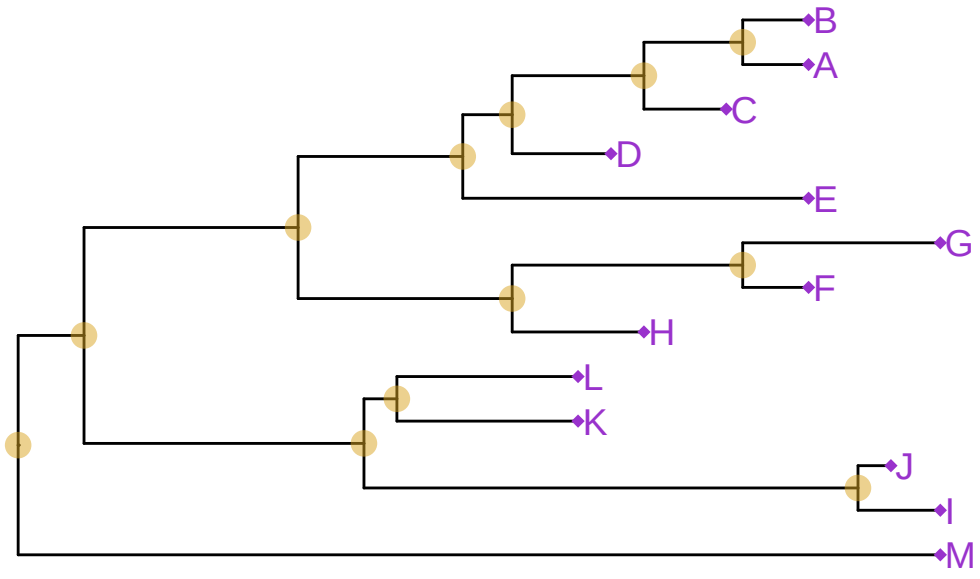
# Label the tips
p + geom_tiplab()
```

Exercise 2

Similar to how we change the aesthetics for the tree inside the `ggtree()` call, we can also change the aesthetics of the points themselves by passing graphical parameters inside the `geom_nodepoint()` or `geom_tippoint()` calls. Create a phylogeny with the following aesthetic characteristics:

- tips labeled in purple
- purple-colored diamond-shape tip points (hint: Google search "R point characters")
- large semitransparent yellow node points (hint: `alpha=`)
- Add a title with `+ ggtitle(...)`

Exercise Figure: Not the prettiest phylogenetic aesthetics, but it'll do



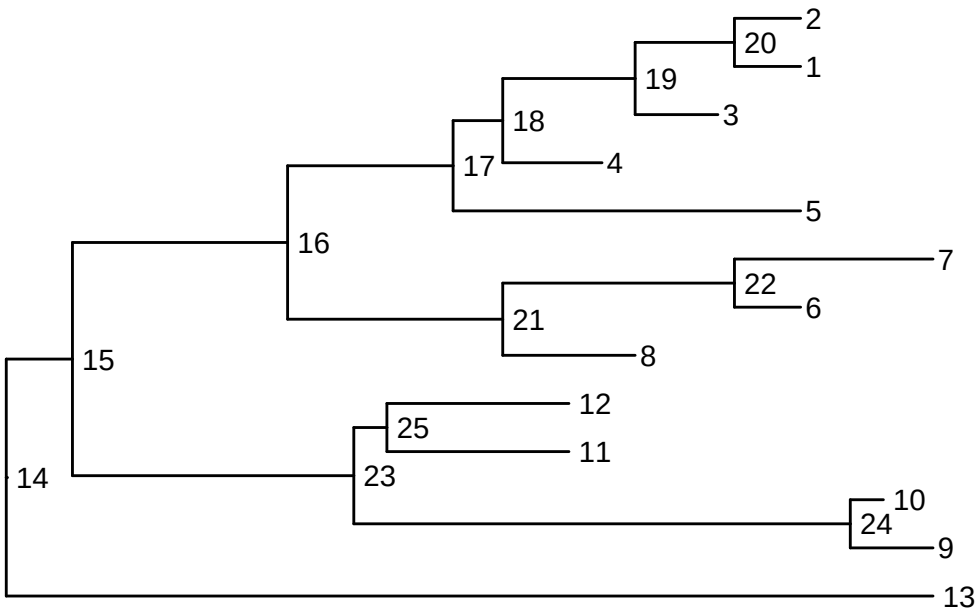
13.4 Tree annotation

The `geom_tiplab()` function adds some very rudimentary annotation. Let's take annotation a bit further. See the tree annotation and advanced tree annotation vignettes for more.

13.4.1 Internal node number

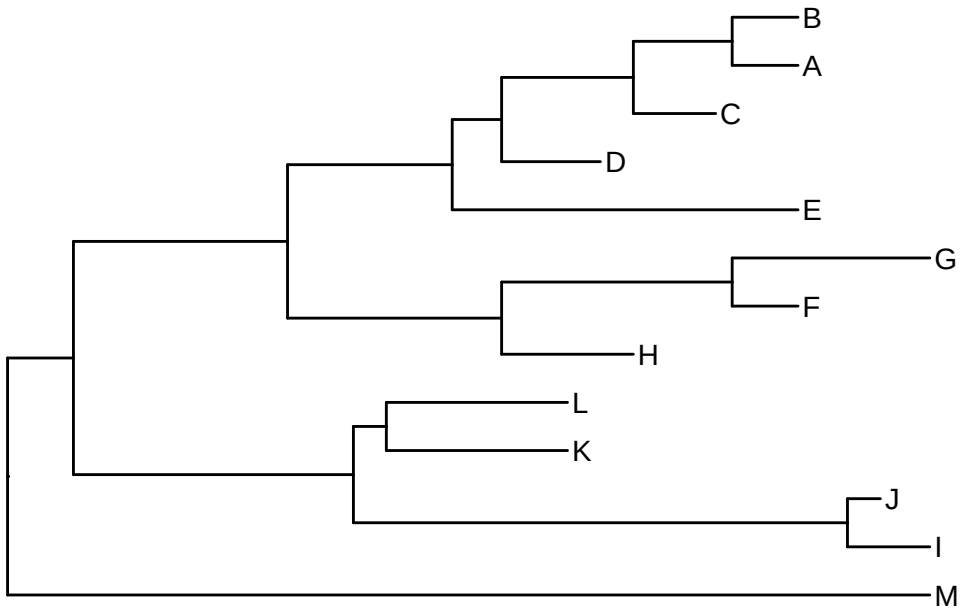
Before we can go further we need to understand how ggtree is handling the tree structure internally. Some of the functions in ggtree for annotating clades need a parameter specifying the internal node number. To get the internal node number, user can use `geom_text` to display it, where the label is an aesthetic mapping to the "node variable" stored inside the tree object (think of this like the continent variable inside the gapminder object). We also supply the `hjust` option so that the labels aren't sitting right on top of the nodes. Read more about this process in the ggtree manipulation vignette.

```
ggtree(tree) + geom_text(aes(label=node), hjust=-.3)
```



Another way to get the internal node number is using `MRCA()` function by providing a vector of taxa names (created using `c("taxon1", "taxon2")`).. The function will return node number of input taxa's most recent commond ancestor (MRCA). First, re-create the plot so you can choose which taxa you want to grab the MRCA from.

```
ggtree(tree) + geom_tiplab()
```



Let's grab the most recent common ancestor for taxa C+E, and taxa G+H. We can use `MRCA()` to get the internal node numbers. Go back to the node-labeled plot from before to confirm this.

```
MRCA(tree, tip=c("C", "E"))
MRCA(tree, tip=c("G", "H"))
```

13.4.2 Labeling clades

We can use `geom_cladelabel()` to add another geom layer to annotate a selected clade with a bar indicating the clade with a corresponding label. You select the clades using the internal node number for the node that connects all the taxa in that clade. See the tree annotation vignette for more.

Let's annotate the clade with the most recent common ancestor between taxa C and E (internal node 17). Let's make the annotation red. See `?geom_cladelabel` help for more.

```
ggtree(tree) +
  geom_cladelabel(node=17, label="Some random clade", color="red")
```

Let's add back in the tip labels. Notice how now the clade label is too close to the tip labels. Let's add an offset to adjust the position. You might have to fiddle with this number to get it looking right.

```
ggtree(tree) +
  geom_tiplab() +
  geom_cladelabel(node=17, label="Some random clade",
                  color="red2", offset=.8)
```

Now let's add another label for the clade connecting taxa G and H (internal node 21).

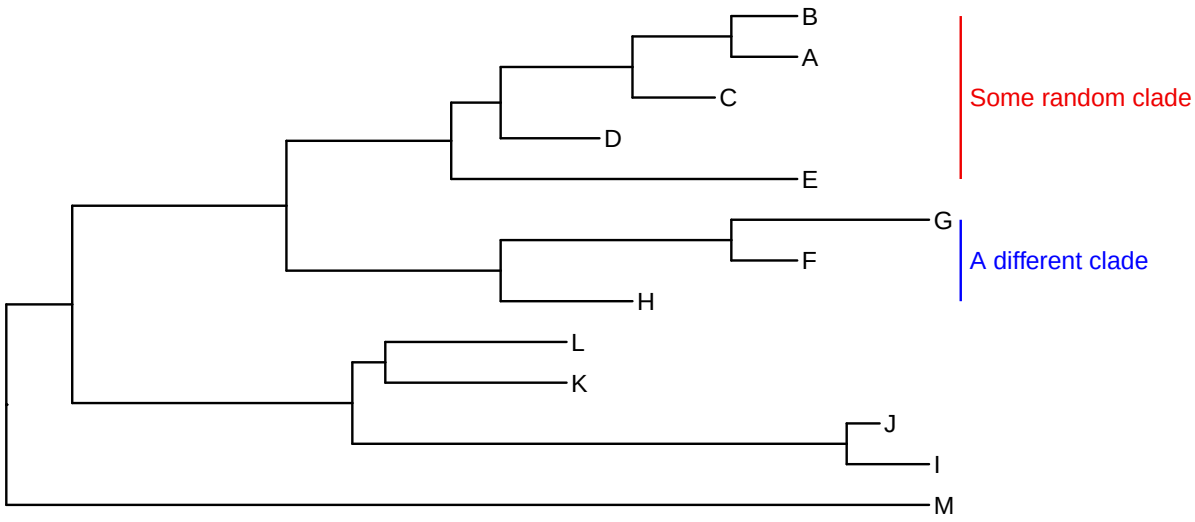
```
ggtree(tree) +
  geom_tiplab() +
  geom_cladelabel(node=17, label="Some random clade",
                  color="red2", offset=.8) +
  geom_cladelabel(node=21, label="A different clade",
                  color="blue", offset=.8)
```

Uh oh. Now we have two problems. First, the labels would look better if they were aligned. That's simple. Pass `align=TRUE` to `geom_cladelabel()` (see `?geom_cladelabel` help for more). But now, the labels are falling off the edge of the plot. That's because `geom_cladelabel()` is just adding it this layer onto the end of the existing canvas that was originally layed out in the ggtree call. This default layout tried to optimize by plotting the entire tree over the entire region of the plot. Here's how we'll fix this.

1. First create the generic layout of the plot with `ggtree(tree)`.
2. Add some tip labels.
3. Add each clade label.
4. Remember **`theme_tree2()`**? We used it way back to add a scale to the x-axis showing the genetic distance. This is the unit of the x-axis. We need to set the limits on the x-axis. Google around for something like "ggplot2 x axis limits" and you'll wind up on this StackOverflow page that tells you exactly how to solve it – just add on a `+ xlim(..., ...)` layer. Here let's extend out the axis a bit further to the right.
5. Finally, if we want, we can either *comment out* the `theme_tree2()` segment of the code, or we could just add another theme layer on top of the plot altogether, which will override the theme that was set before. `theme_tree()` doesn't have the scale.

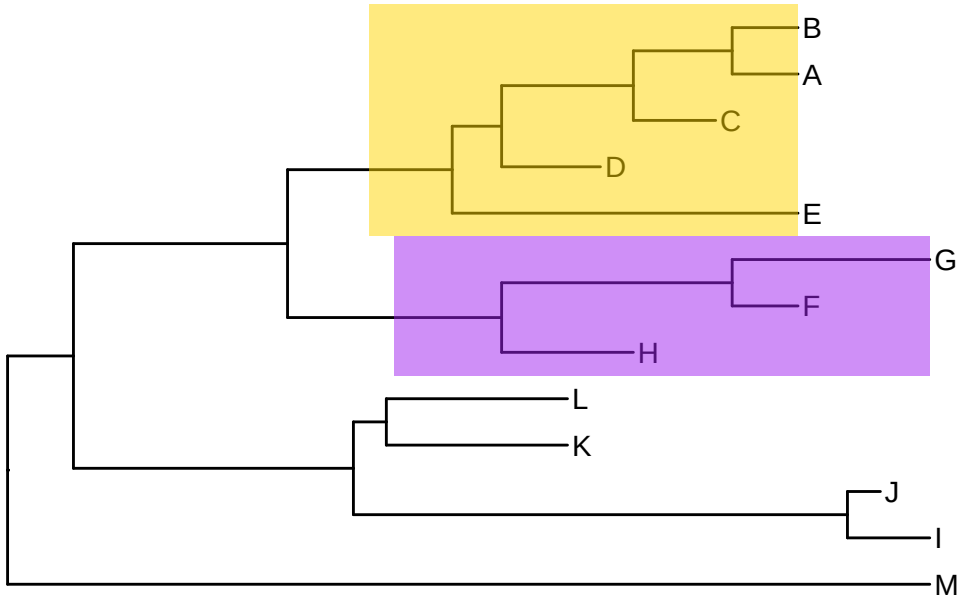
```
ggtree(tree) +
  geom_tiplab() +
  geom_cladelabel(node=17, label="Some random clade",
                  color="red2", offset=.8, align=TRUE) +
  geom_cladelabel(node=21, label="A different clade",
```

```
                color="blue", offset=.8, align=TRUE) +
  theme_tree2() +
  xlim(0, 70) +
  theme_tree()
```



Alternatively, we could highlight the entire clade with `geom_hilight()`. See the help for options to tweak.

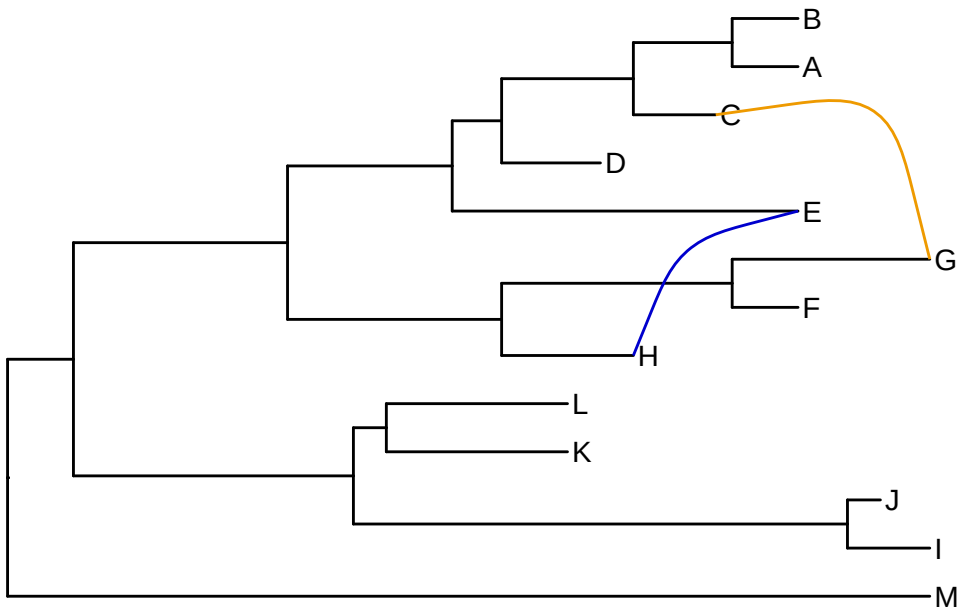
```
ggtree(tree) +
  geom_tiplab() +
  geom_hilight(node=17, fill="gold") +
  geom_hilight(node=21, fill="purple")
```



13.4.3 Connecting taxa

Some evolutionary events (e.g. reassortment, horizontal gene transfer) can be visualized with some simple annotations on a tree. The `geom_taxalink()` layer draws straight or curved lines between any of two nodes in the tree, allow it to show evolutionary events by connecting taxa. Take a look at the tree annotation vignette and `?geom_taxalink` for more.

```
ggtree(tree) +
  geom_tiplab() +
  geom_taxalink("E", "H", color="blue3") +
  geom_taxalink("C", "G", color="orange2", curvature=-.9)
```

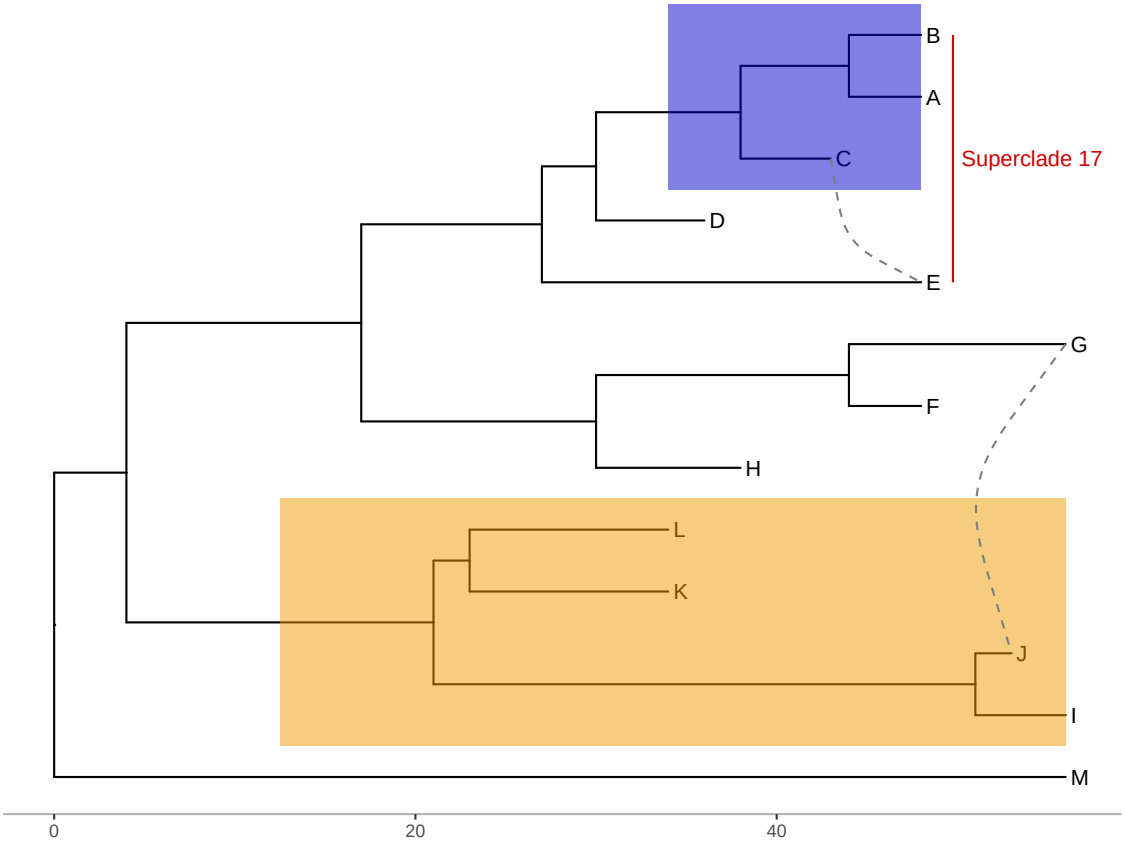


Exercise 3

Produce the figure below.

1. First, find what the MRCA is for taxa **B+C**, and taxa **L+J**. You can do this in one of two ways:
   a. Easiest: use `MRCA(tree, tip=c("taxon1", "taxon2"))` for B/C and L/J separately.
   b. Alternatively: use `ggtree(tree) + geom_text(aes(label=node), hjust=-.3)` to see what the node labels are on the plot. You might also add tip labels here too.
2. Draw the tree with `ggtree(tree)`.
3. Add tip labels.
4. Highlight these clades with separate colors.
5. Add a clade label to the larger superclade (node=17) that we saw before that includes A, B, C, D, and E. You'll probably need an offset to get this looking right.
6. Link taxa C to E, and G to J with a dashed gray line (hint: get the geom working first, then try changing the aesthetics. You'll need `linetype=2` somewhere in the `geom_taxalink()`).
7. Add a scale bar to the bottom by changing the theme.
8. Add a title.
9. Optionally, go back to the original `ggtree(tree, ...)` call and change the layout to `"circular"`.

Exercise title: Not sure what we're trying to show here...



13.5 Advanced tree annotation

Let's use a previously published dataset from this paper:

Liang et al. "Expansion of genotypic diversity and establishment of 2009 H1N1 pandemic-origin internal genes in pigs in China." *Journal of virology* (2014): 88(18):10864-74.

This data was reanalyzed in the ggtree paper.

The subset of the data used here contains 76 H3 hemagglutinin gene sequences of a lineage containing both swine and human influenza A viruses. The sequence data set was re-analyzed by using BEAST (available at http://beast.bio.ed.ac.uk/). BEAST (Bayesian Evolutionary Analysis Sampling Trees) can give you rooted, time-measured phylogenies inferred using molecular clock models.

For this you'll need the `flu_tree_beast.tree` output file from BEAST and the `flu_aasequence.fasta` FASTA file with the multiple sequence alignment. These are both available on the data downloads page. First let's read in the tree with `read.beast()` (instead of the `read.tree()` we used before). Let's add a scale bar with theme_tree2(). This gives you genetic distance. But, we have time measured here with molecular clock models. We've only estimated the *relative* time between branching events, so if we want to actually see *dates* on the x-axis, we need to supply the most recent sampling date to the `ggtree()` call. Do this by setting `mrsd="YYYY-MM-DD"` inside `ggtree()`.

Finally, let's add some tip labels. We'll want to right-align them, and by default the dotted line is a little too thick. Let's reduce the `linesize` a bit. Now, some of the labels might be falling off the margin. Set the `xlim` to limit the axis to show between 1990 and 2020. You could get MRCAs and node numbers and do all the annotations that we did before the same way here.

```
# Read the data
tree <- read.beast("data/flu_tree_beast.tree")

# supply a most recent sampling date so you get the dates
# and add a scale bar
ggtree(tree, mrsd="2013-01-01") +
  theme_tree2()

# Finally, add tip labels and adjust axis
ggtree(tree, mrsd="2013-01-01") +
  theme_tree2() +
  # geom_tiplab(align=TRUE, linesize=.5) +
  geom_tiplab(linesize=.5) +
  xlim(1990, 2020)
```

Finally, let's look at `?msaplot`. This puts the multiple sequence alignment and the tree side-by-side. The function takes a tree object (produced with `ggtree()`) and the path to the FASTA multiple sequence alignment. You can do it with the entire MSA, or you could restrict to just a window. Want something interesting-looking, but maybe not all that useful? Try changing the coordinate system of the plot itself by passing `+ coord_polar(theta="y")` to the end of the command!

```
msaplot(p=ggtree(tree), fasta="data/flu_aasequence.fasta", window=c(150, 175))
```

Take a look at the advanced tree annotation vignette for much, much more!

13.6 Bonus!

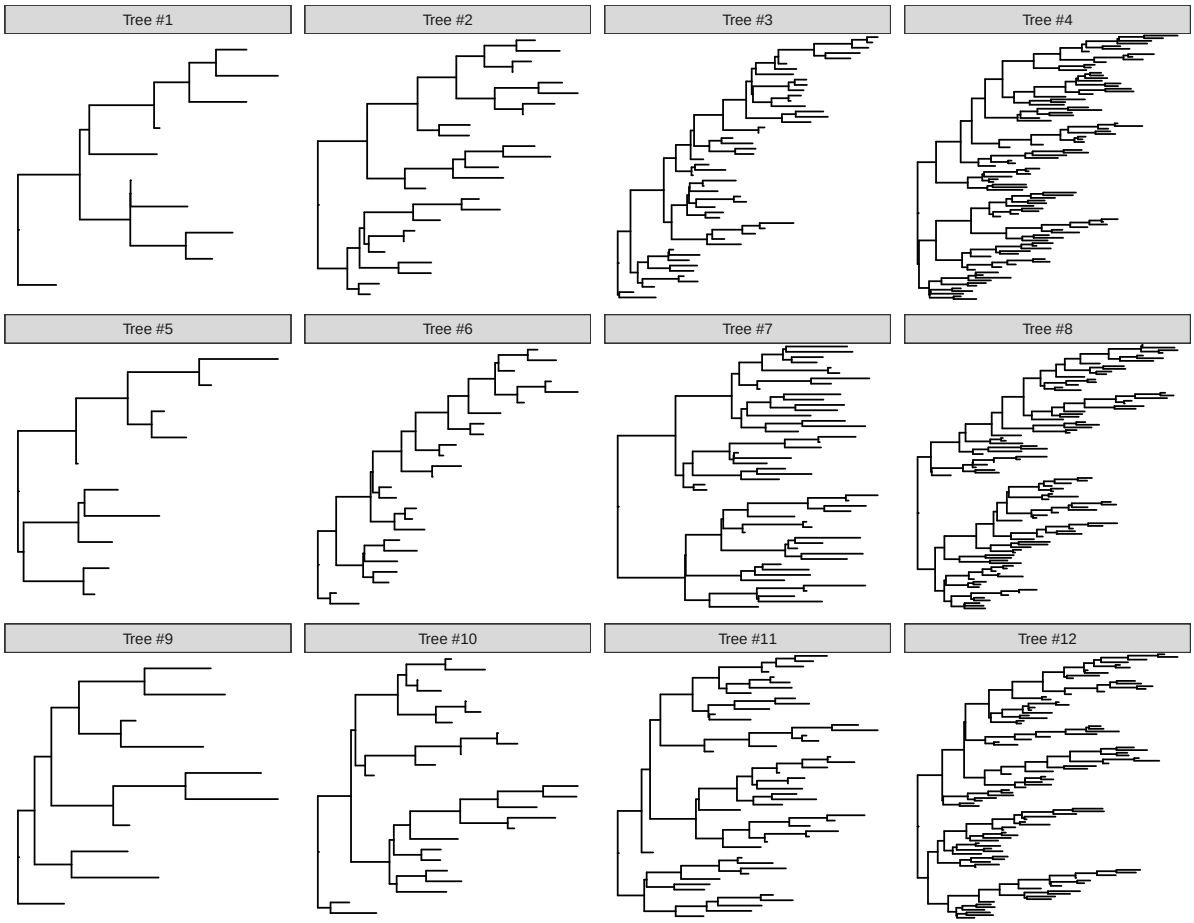
See the ggtree vignettes for more details on how these work.

13.6.1 Many trees

ggtree will let you plot many trees at once, and you can facet them the normal ggplot2 way. Let's generate 3 replicates each of 4 random trees with 10, 25, 50, and 100 tips, plotting them all.

```
set.seed(42)
trees <- lapply(rep(c(10, 25, 50, 100), 3), rtree)
class(trees) <- "multiPhylo"
ggtree(trees) + ggplot2::facet_wrap(~.id, scale="free", ncol=4) + ggplot2::ggtitle("Many t
```

Many trees. Such phylogenetics. Wow.



13.6.2 Plot tree with other data

For showing a phylogenetic tree alongside other panels with your own data, the `facet_plot()` function accepts a input data.frame and a `geom` function to draw the input data.

```
# Generate a random tree with 30 tips
tree <- rtree(30)

# Make the original plot
p <- ggtree(tree)

# generate some random values for each tip label in the data
d1 <- data.frame(id=tree$tip.label, val=rnorm(30, sd=3))

# Make a second plot with the original, naming the new plot "dot",
# using the data you just created, with a point geom.
p2 <- facet_plot(p, panel="dot", data=d1, geom=geom_point, aes(x=val), color='red3')

# Make some more data with another random value.
d2 <- data.frame(id=tree$tip.label, value = abs(rnorm(30, mean=100, sd=50)))

# Now add to that second plot, this time using the new d2 data above,
# This time showing a bar segment, size 3, colored blue.
p3 <- facet_plot(p2, panel='bar', data=d2, geom=geom_segment,
           aes(x=0, xend=value, y=y, yend=y), size=3, color='blue4')

# Show all three plots with a scale
p3 + theme_tree2()
```

13.6.3 Overlay organism silouhettes

phylopic.org hosts free silhouette images of animals, plants, and other life forms, all under Creative Commons or Public Domain. You can use ggtree to overlay a phylopic image on your plot at a node of your choosing. Let's show some gram-negative bacteria over the whole plot, and put a *Homo sapiens* and a dog on those clades we're working with.

```
read.tree("data/tree_newick.nwk") %>%
  ggtree() %>%
  phylopic("ba0a446e-18d7-4db9-9937-5adec24721b5",
           color="gold2", alpha = .25) %>%
  phylopic("c089caae-43ef-4e4e-bf26-973dd4cb65c5",
```

```
          color="purple3", alpha = .5, node=17) %>%
phylopic("6c9cb19d-1d8a-4215-88ba-d49cd4917a5e",
          color="purple3", alpha = .5, node=21)
```

References

Bryan, Jennifer. 2019. "STAT 545: Data Wrangling, Exploration, and Analysis with r." https://stat545.com/.

Love, Michael I., Wolfgang Huber, and Simon Anders. 2014. "Moderated Estimation of Fold Change and Dispersion for RNA-seq Data with DESeq2." *Genome Biology* 15 (12): 1–21.

Robinson, David. 2015. "Variance Explained." http://varianceexplained.org/.

Silge, Julia, and David Robinson. 2017. *Text Mining with R: A Tidy Approach*. 1st edition. Beijing ; Boston: O'Reilly Media.

Teal, Tracy K., Karen A. Cranston, Hilmar Lapp, Ethan White, Greg Wilson, Karthik Ram, and Aleksandra Pawlik. 2015. "Data Carpentry: Workshops to Increase Data Literacy for Researchers."

Wilson, Greg. 2014. "Software Carpentry: Lessons Learned." *F1000Research* 3.

Yu, Guangchuang. 2022. "Ggtree: An r Package for Visualization of Tree and Annotation Data." http://bioconductor.org/packages/ggtree/.

Yu, Guangchuang, David K. Smith, Huachen Zhu, Yi Guan, and Tommy Tsan-Yuk Lam. 2017. "Ggtree: An R Package for Visualization and Annotation of Phylogenetic Trees with Their Covariates and Other Associated Data." *Methods in Ecology and Evolution* 8 (1): 28–36.

A Setup

A.1 Software

1. **R**. If you don't have R installed, download and install it from CRAN.
2. **RStudio**. Download and install it from RStudio's website.
3. **R packages**. Install the following packages by running the following code in RStudio:

```
# Needed for most chapters
install.packages("tidyverse")

# Needed for certain chapter
install.packages(c("plotly",
                   "DT",
                   "knitr",
                   "rmarkdown",
                   "survminer",
                   "ModelMetrics",
                   "gower",
                   "randomForest",
                   "gbm",
                   "glmnet",
                   "mice",
                   "prophet",
                   "tidytext",
                   "gutenbergr",
                   "tm",
                   "topicmodels"))

# For the predictive modeling chapter
install.packages("caret", dependencies = c("Depends", "Suggests"))

# Bioconductor packages are installed differently
install.packages("BiocManager")
BiocManager::install(c("DESeq2",
                       "RTCGA",
```

```
                    "RTCGA.clinical",
                    "RTCGA.mRNA",
                    "ggtree",
                  "Biostrings"))
```

A.2 Data

1. **Option 1: Download all the data**. Download and extract **this zip file** (11.36 Mb) with all the data for the entire workshop. This may include additional datasets that we won't use here.
2. **Option 2: Download individual datasets as needed.**
    - Create a new folder somewhere on your computer that's easy to get to (e.g., your Desktop). Name it `bds`. Inside that folder, make a folder called `data`, all lowercase.
    - Download individual data files as needed, saving them to the new `bdsr/data` folder you just made. Click to download. If data displays in your browser, right-click and select *Save link as…* (or similar) to save to the desired location.

- data/airway_metadata.csv
- data/airway_scaledcounts.csv
- data/annotables_grch38.csv
- data/austen.csv
- data/brauer2007_messy.csv
- data/brauer2007_sysname2go.csv
- data/brauer2007_tidy.csv
- data/dmd.csv
- data/flu_genotype.csv
- data/gapminder.csv
- data/grads_dd.csv
- data/grads.csv
- data/h7n9_analysisready.csv
- data/h7n9.csv
- data/heartrate2dose.csv
- data/ilinet.csv
- data/movies_dd.csv
- data/movies_imdb.csv
- data/movies.csv
- data/nhanes_dd.csv
- data/nhanes.csv
- data/SRP026387_metadata.csv
- data/SRP026387_scaledcounts.csv

- data/stressEcho.csv

B Further Resources

B.1 R resources

B.1.1 Getting Help

- Google it!: Try Googling generalized versions of any error messages you get. That is, remove text that is specific to your problem (names of variables, paths, datasets, etc.). You'd be surprised how many other people have probably had the same problem and solved it.
- Stack Overflow: There are over 100,000 questions tagged with "R" on SO. Here are the most popular ones, ranked by vote. Always search before asking, and make a reproducible example if you want to get useful advice. This is a minimal example that allows others who are trying to help you to see the error themselves.
- Bioconductor Support Site: Like SO, but specifically for Bioconductor-related questions.
- Read package vignettes. For example, see the dplyr CRAN page, scroll about halfway down to see the introduction to dplyr vignette.

B.1.2 General R Resources

- TryR: An interactive, browser-based R tutor
- Swirl: An R package that teaches you R (*and statistics!*) from within R
- Jenny Bryan's Stat 545 "Data wrangling, exploration, and analysis with R" course material: An excellent resource for learning R, dplyr, and ggplot2
- DataCamp's free introduction to R
- More DataCamp courses (UVA's education benefits will cover these!).
- RStudio's printable cheat sheets
- Rseek: A custom Google search for R-related sites
- Bioconductor vignettes, workflows, and course/conference materials

B.1.3 dplyr resources

- The dplyr vignette
- A longer dplyr tutorial with video and code
- The dplyr tutorial from the HarvardX Biomedical Data Science MOOC

- A dplyr cheat sheet from RStudio

B.1.4 ggplot2 resources

- The official ggplot2 documentation
- The ggplot2 book, edition 1, by the developer, Hadley Wickham
- New version of the ggplot2 book, freely available on GitHub
- The ggplot2 Google Group (mailing list, support forum)
- LearnR: A blog with a good number of posts describing how to reproduce various kind of plots using ggplot2
- SO questions tagged with ggplot2
- A catalog of graphs made with ggplot2, complete with accompanying R code
- RStudio's ggplot2 cheat sheet

B.1.5 Markdown / RMarkdown resources

- Basic Markdown + RMarkdown reference
- In-browser markdown editors:
  - Minimal: bioconnector.github.io/markdown-editor
  - Better: stackedit.io, dillinger.io
- A good markdown reference
- A good 10-minute markdown tutorial
- RStudio's RMarkdown Cheat Sheet and RMarkdown Reference Sheet
- The RMarkdown documentation has an excellent getting started guide, a gallery of demos, and several articles illustrating advanced usage.
- The knitr website has lots of useful reference material about how knitr works, options, and more.

B.2 RNA-seq resources

- University of Oregon's RNA-seqlopedia: a comprehensive guide to RNA-seq starting with experimental design, going through library prep, sequencing, and data analysis.
- Conesa et al. A survey of best practices for RNA-seq data analysis. *Genome Biology* 17:13 (2016). If there's one review to read on RNA-seq and data analysis, it's this one.
- rnaseq.wiki & accompanying paper for hands-on RNA-seq data analysis examples using cloud computing.
- RNA-seq blog: Several blog posts per week on new methods and tools for RNA-seq analysis.

- YouTube playlist: 2015 UC Davis Workshop on RNA-seq methods & algorithms (Harold Pimentel).
- What the FPKM? A review of RNA-Seq expression units A blog post from Harold Pimentel describing the relationship between R/FPKM and TPM.
- RNA-seq analysis exercise using Galaxy: an example analysis you can run yourself using the Tophat+Cufflinks workflow.
- "RNA-Seq workflow: gene-level exploratory analysis and differential expression." This paper walks through an end-to-end gene-level RNA-Seq differential expression workflow using Bioconductor packages, starting from FASTQ files.
- The DESeq2 paper describes the modeling approach and shows some benchmarks against other normalization and differential expression strategies.
- The DESeq2 vignette is packed full of examples on using DESeq2, importing data, fitting models, creating visualizations, references, etc.
- Lior Pachter's paper "Models for transcript quantification from RNA-Seq" reviews different approaches for quantifying expression from RNA-seq data and how these affect downstream analysis.
- SEQAnswers RNA-seq and more general bioinformatics forums are a great place to search for answers.
- Biostars RNA-seq Q&A section.
- Blog post and printable PDF I created demonstrating how to do pathway analysis with RNA-seq data and R.