CADI 2017

Héctor Corrada Bravo

2017-06-07

Contents

Chapter 1

Preamble

Here is where the course notes will be included.

Chapter 2

Introduction and Overview

2.1 What is Data Science?

Data science encapsulates the interdisciplinary activities required to create data-centric artifacts and applications that address specific scientific, socio-political, business, or other questions.

Let's look at the constiuent parts of this statement:

2.1.1 Data

Measureable units of information gathered or captured from activity of people, places and things.

2.1.2 Specific Questions

Seeking to understand a phenomenon, natural, social or other, can we formulate specific questions for which an answer posed in terms of patterns observed, tested and or modeled in data is appropriate.

2.1.3 Interdisciplinary Activities

Formulating a question, assessing the appropriateness of the data and findings used to find an answer require understanding of the specific subject area. Deciding on the appropriateness of models and inferences made from models based on the data at hand requires understanding of statistical and computational methods.

2.1.4 Data-centric artifacts and applications

Answers to questions derived from data are usually shared and published in meaningful, succint but sufficient, reproducible artifacts (papers, books, movies, comics). Going a step further, interactive applications that let others explore data, models and inferences are great.

2.2 Why Data Science?

The granularity, size and accessibility data, comprising both physical, social, commercial and political spheres has exploded in the last decade or more.

> I keep saying that the sexy job in the next 10 years will be statisticians"
>
> Hal Varian, Chief Economist at Google (http://www.nytimes.com/2009/08/06/technology/06stats.html?_r=0)
>
> "The ability to take data—to be able to understand it, to process it, to extract value from it, to visualize it, to communicate it—that's going to be a hugely important skill in the next decades, not only at the professional level but even at the educational level for elementary school kids, for high school kids, for college kids."
>
> "Because now we really do have essentially free and ubiquitous data. So the complimentary scarce factor is the ability to understand that data and extract value from it."
>
> Hal Varian (http://www.mckinsey.com/insights/innovation/hal_varian_on_how_the_web_challenges_managers)

2.3 Data Science in Humanities and Social Sciences

Because of the large amount of data produced across many spheres of human social and creative activity, many questions in the humanities and social sciences may be addressed by establishing patterns in data. In the humanities, this can range from unproblematic quesitions of how to dissect a large creative corpora, say music, literature, based on raw characteristics of those works, text, sound and image. To more problematic questions, of analysis of intent, understanding, appreciation and valuation of these creative corpora.

In the social sciences, issues of fairness and transparency in the current era of big data are especially problematic. Is data collected representative of population for which inferences are drawn? Are methods employed learning latent unfair factors from ostensibly fair data? These are issues that the research community is now starting to address.

In all settings, issues of ethical collection of data, application of models, and deployment of data-centric artifacts are essential to grapple with. Issues of privacy are equally important.

2.4 Course organization

This course will cover basics of how to use the R data analysis environment for data science activities in humanities and social science. The general organization of the course will be the following:

2.4.1 Day 1: Introduction and preparation

- Get ourselved settled with R
- Formulating a question that can be addressed with data
- How to get data to address a specific question
- How to manipulate this data to get to what we need for our question of interest
- How to visualize this data

2.4.2 Day 2: Use cases

- Social media analysis: text and network structures
- Fitting and evaluating regression models in R
- Text analysis
- Machine Learning modeling in R
- Survey analysis
- Nested models in R

2.4.3 Day 3: Presentation and teaching

- How to present and publish analysis using R
- Using R in teaching

2.5 General Workflow

The data science activities we will cover are roughly organized into a general workflow that will help us navigate this material.

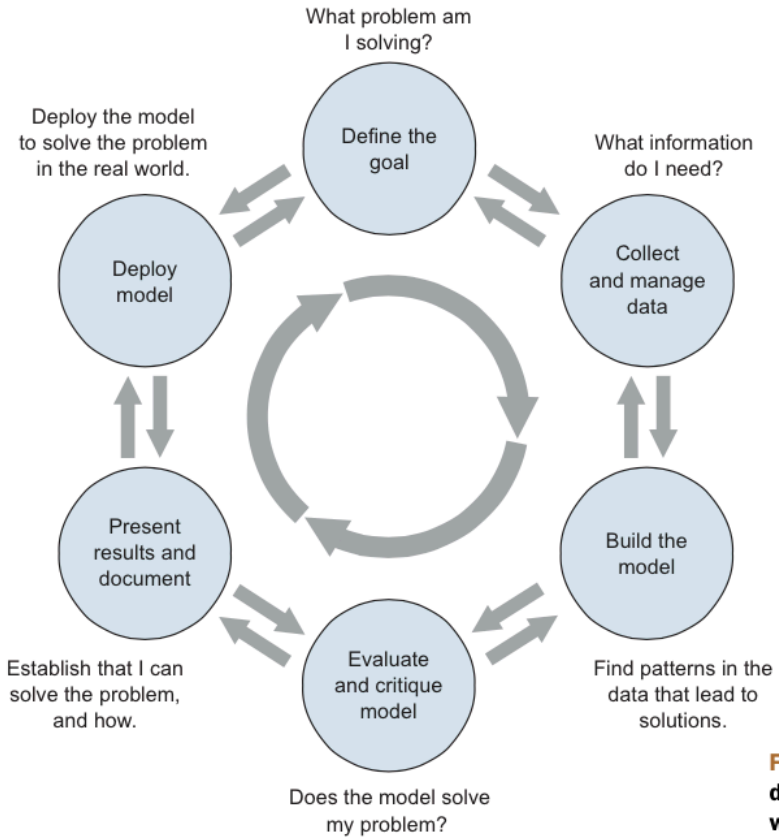


Figure 1.1 The lifecycle of a data science project: loops within loops

Figure 2.1:

2.5.1 Defining the goal

- What is the question/problem?
- Who wants to answer/solve it?
- What do they know/do now?
- How well can we expect to answer/solve it?
- How well do they want us to answer/solve it?

2.5.2 Data collection and Management

- What data is available?
- Is it good enough?
- Is it enough?
- What are sensible measurements to derive from this data? Units, transformations, rates, ratios, etc.

2.5.3 Modeling

- What kind of problem is it? E.g., classification, clustering, regression, etc.
- What kind of model should I use?
- Do I have enough data for it?
- Does it really answer the question?

2.5.4 Model evaluation

- Did it work? How well?
- Can I interpret the model?
- What have I learned?

2.5.5 Presentation

- Again, what are the measurements that tell the real story?
- How can I describe and visualize them effectively?

2.5.6 Deployment

- Where will it be hosted?
- Who will use it?
- Who will maintain it?

Chapter 3

An Illustrative Analysis

http://fivethirtyeight.com has a clever series of articles on the types of movies different actors make in their careers: https://fivethirtyeight.com/tag/hollywood-taxonomy/

I'd like to do a similar analysis. Let's do this in order:

1) Let's do this analysis for Diego Luna
2) Let's use a clustering algorithm to determine the different types of movies they make
3) Then, let's write an application that performs this analysis for any actor and test it with Gael García Bernal
4) Let's make the application interactive so that a user can change the actor and the number of movie clusters the method learns.

For now, we will go step by step through this analysis without showing how we perform this analysis using R. As the course progresses, we will learn how to carry out these steps.

3.1 Gathering data

3.1.1 Movie ratings

For this analysis we need to get the movies Diego Luna was in, along with their Rotten Tomatoes ratings. For that we scrape this webpage: https://www.rottentomatoes.com/celebrity/diego_luna.

Once we scrape the data from the Rotten Tomatoes website and clean it up, this is part of what we have so far:

RATING	TITLE	CREDIT	BOX OFFICE	YEAR
85	Rogue One: A Star Wars Story	Captain Cassian Andor	$532.2M	2016
89	Blood Father	Jonah	—	2016
82	The Book of Life	Manolo	—	2014
100	I Stay with You (Me quedo contigo)	Actor	—	2014
67	Elysium	Julio	$90.8M	2013
41	Casa de mi padre	Raul	$5.9M	2012
51	Contraband	Gonzalo	$66.5M	2012

This data includes, for each of the movies Diego Luna has acted in, the rotten tomatoes rating, the movie title, Diego Luna's role in the movie, the U.S. domestic gross and the year of release.

3.1.2 Movie budgets and revenue

For the movie budgets and revenue data we scrape this webpage: http://www.the-numbers.com/movie/budgets/all

This is part of what we have for that table after scraping and cleaning up:

release_date	movie	production_budget	domestic_gross	worldwide_gross
2009-12-18	Avatar	425	760.50762	2783.9190
2015-12-18	Star Wars Ep. VII: The Force Awakens	306	936.66223	2058.6622
2007-05-24	Pirates of the Caribbean: At World's End	300	309.42043	963.4204
2015-11-06	Spectre	300	200.07417	879.6209
2012-07-20	The Dark Knight Rises	275	448.13910	1084.4391
2013-07-02	The Lone Ranger	275	89.30212	260.0021
2012-03-09	John Carter	275	73.05868	282.7781
2010-11-24	Tangled	260	200.82194	586.5819
2017-06-21	Transformers: The Last Knight	260	0.00000	0.0000
2007-05-04	Spider-Man 3	258	336.53030	890.8753

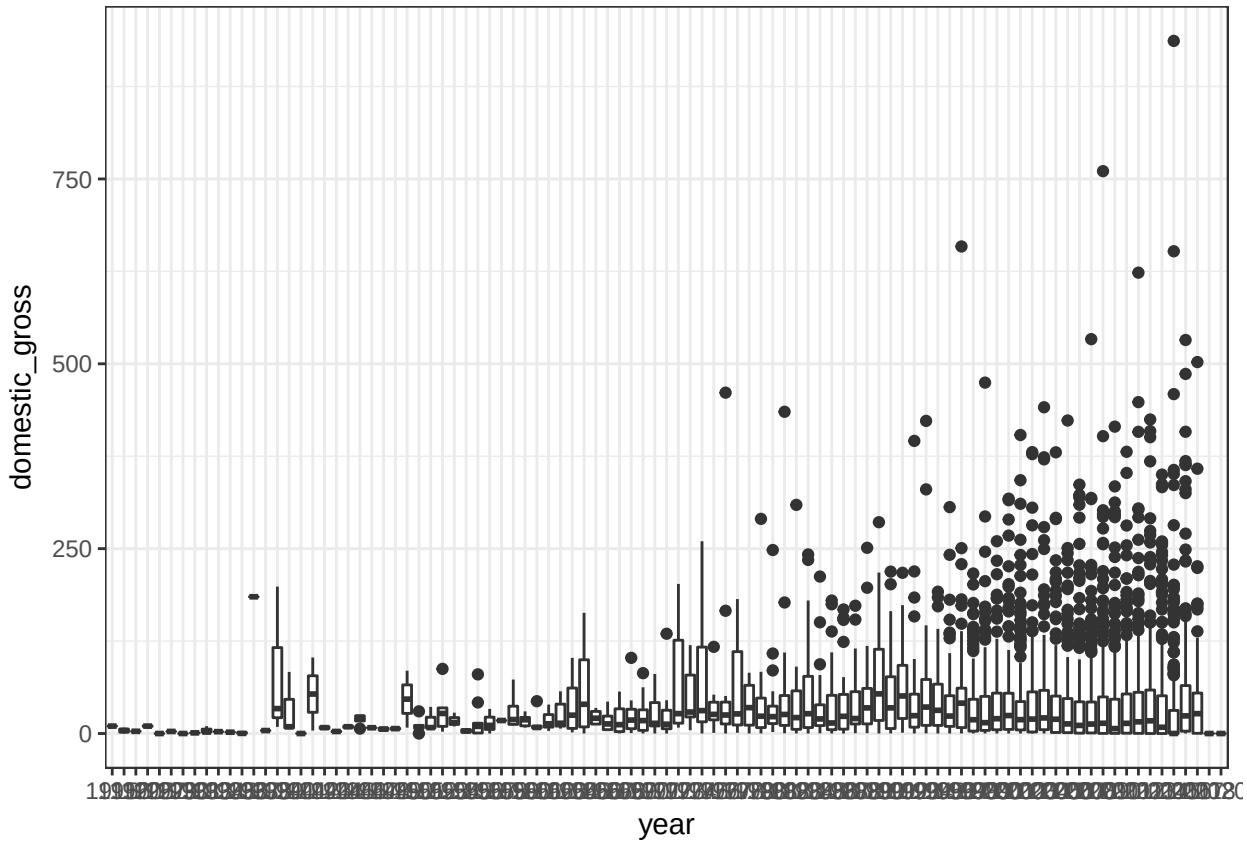
This data is for 5368 movies, including its release date, title, production budget, US domestic and worlwide gross earnings. The latter three are in millions of U.S. dollars.

One thing we might want to check is if the budget and gross entries in this table are inflation adjusted or not. To do this, we can make a plot of domestic gross, which we are using for the subsequent analyses.

```
## Loading required package: methods
```

```
##
## Attaching package: 'lubridate'
```

```
## The following object is masked from 'package:base':
##
##     date
```



Although we don't know for sure, since the source of our data does not state this specifically, it looks like the domestic gross measurement is inflation adjusted since average gross is stable across years.

3.2 Manipulating the data

Next, we combine the datasets we obtained to get closer to the data we need to make the plot we want.

We combine the two datasets using the movie title, so that the end result has the information in both tables for each movie.

RATING	TITLE	CREDIT	BOX OFFICE	YEAR	release_date	producti
85	Rogue One: A Star Wars Story	Captain Cassian Andor	$532.2M	2016	2016-12-16	
82	The Book of Life	Manolo	—	2014	2014-10-17	
67	Elysium	Julio	$90.8M	2013	2013-08-09	
51	Contraband	Gonzalo	$66.5M	2012	2012-01-13	
94	Milk	Jack Lira	$31.8M	2008	2008-11-26	
69	Criminal	Rodrigo	$0.8M	2004	2016-04-15	
61	The Terminal	Enrique Cruz	$77.1M	2004	2004-06-18	
79	Open Range	Button	$58.3M	2003	2003-08-15	
76	Frida	Alejandro Gomez	$25.7M	2002	2002-10-25	

3.3 Visualizing the data

Now that we have the data we need, we can make a plot:

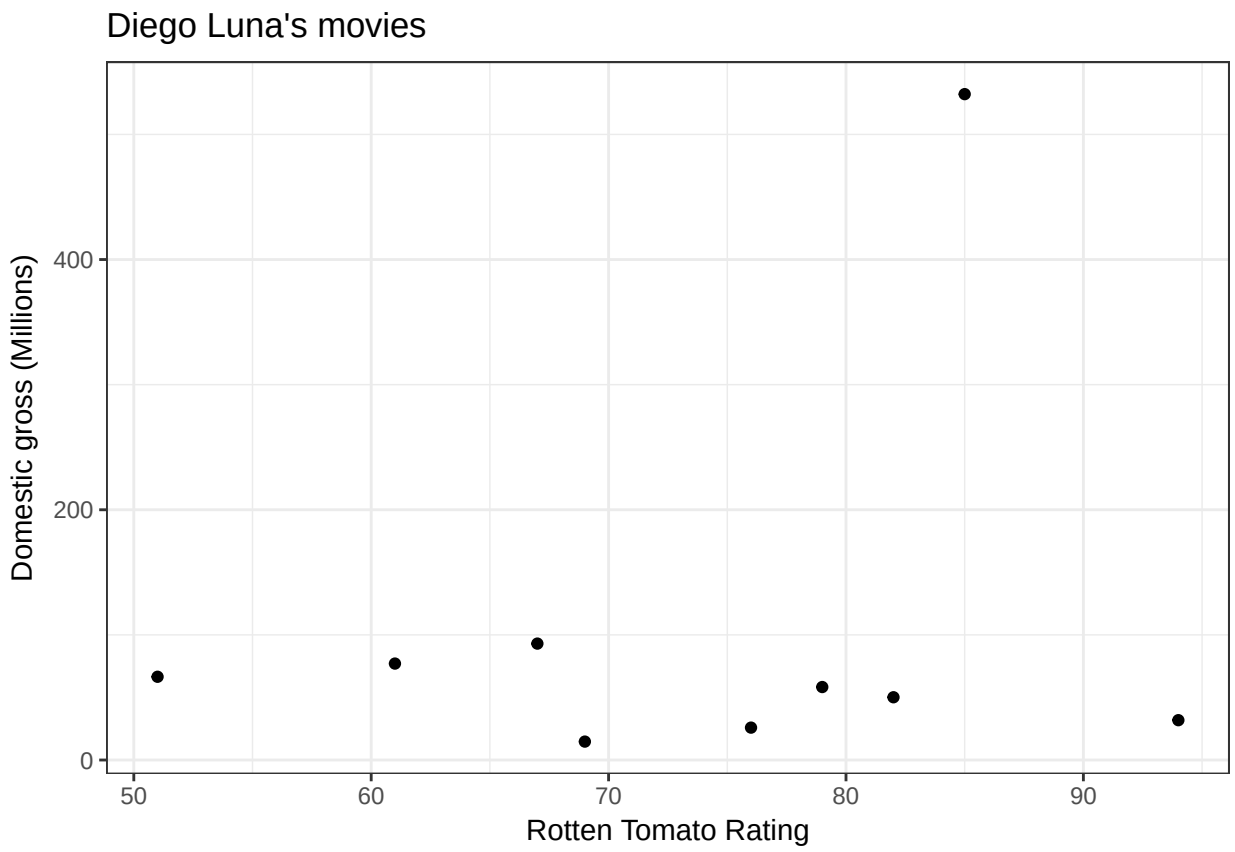


Figure 3.1: Ratings and U.S. Domestic Gross of Diego Luna's movies.

We see that there is one clear outlier in Diego Luna's movies, which probably is the one Star Wars movie he acted in. The remaining movies could potentially be grouped into two types of movies, those with higher rating and those with lower ratings.

3.4 Modeling data

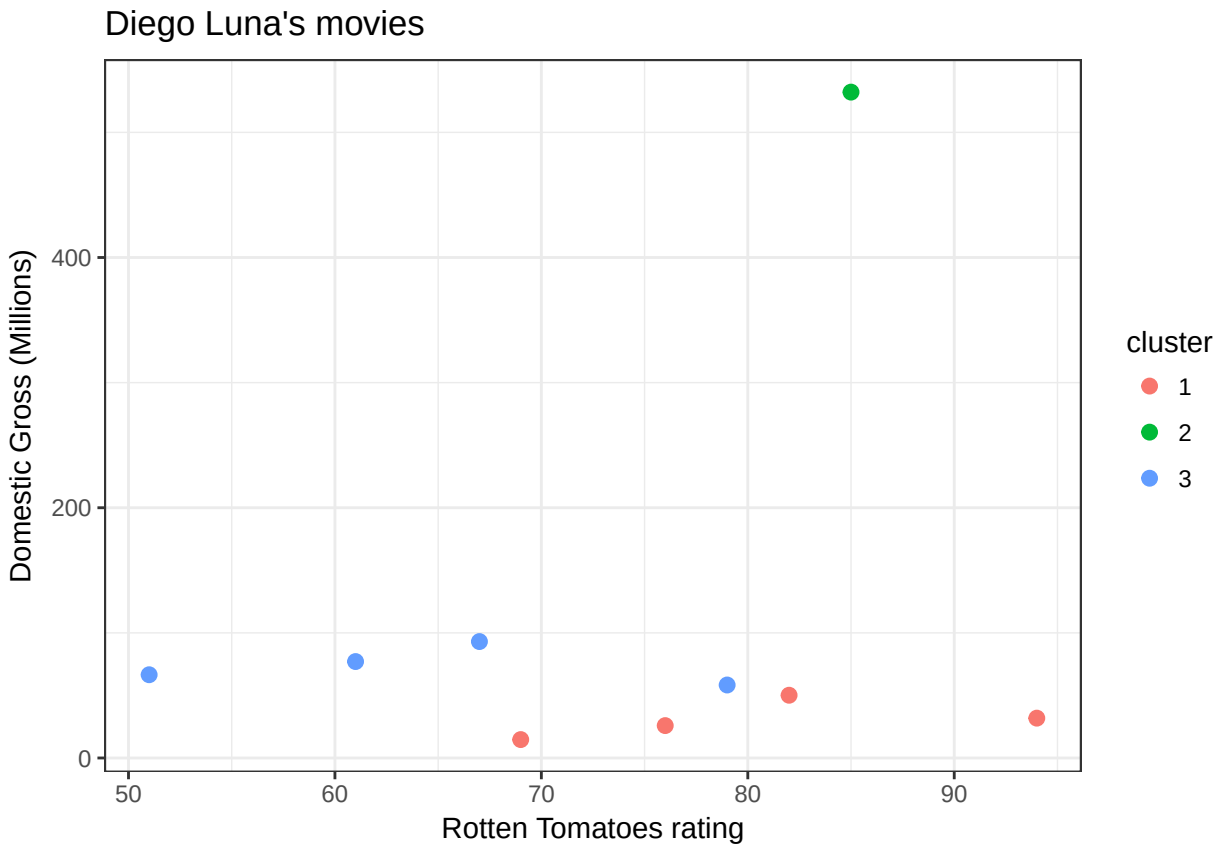
We can use a clustering algorithm to partition Diego Luna's movies. We can use the data we obtained so far and see if the k-means clustering algorithm partitions these movies into three sensible groups using the movie's rating and domestic gross.

Let's see how the movies are grouped:

TITLE	RATING	domestic_gross	cluster
The Book of Life	82	50.15154	1
Milk	94	31.84130	1
Criminal	69	14.70870	1
Frida	76	25.88500	1
Rogue One: A Star Wars Story	85	532.17732	2
Elysium	67	93.05012	3
Contraband	51	66.52800	3
The Terminal	61	77.07396	3
Open Range	79	58.33125	3

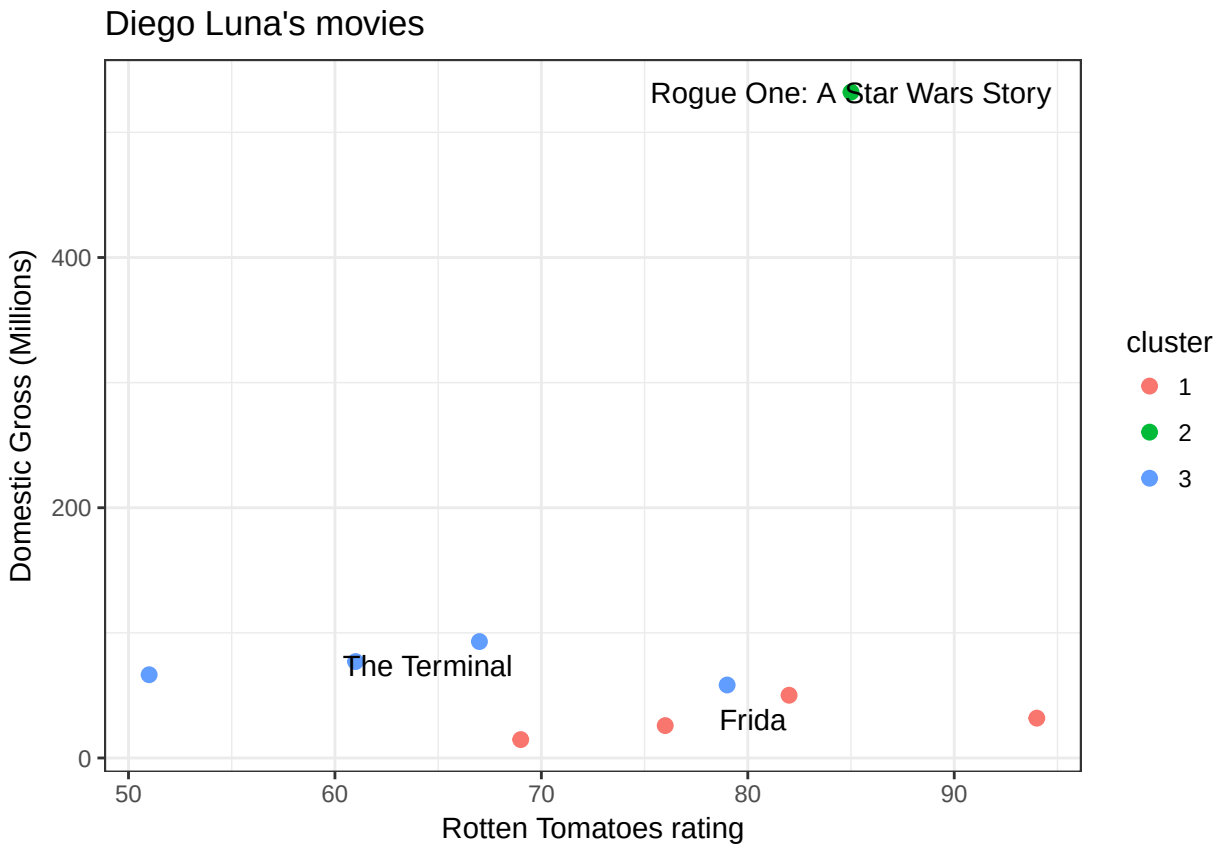
3.5 Visualizing model result

Let's remake the same plot as before, but use color to indicate each movie's cluster assignment given by the k-means algorithm.



The algorithm did make the Star Wars movie it's own group since it's so different that the other movies. The grouping of the remaining movies is not as clean.

To make the plot and clustering more interpretable, let's annotate the graph with some movie titles. In the k-means algorithm, each group of movies is represented by an average rating and an average domestic gross. What we can do is find the movie in each group that is closest to the average and use that movie title to annotate each group in the plot.



Roughly, movies are clustered into Star Wars and low vs. high rated movies. The latter seem to have some difference in domestic gross. For example, movies like "The Terminal" have lower rating but make slightly more money than movies like "Frida". We could use statistical modeling to see if that's the case, but will skip that for now. Do note also, that the clustering algorithm we used seems to be assigning one of the movies incorrectly, which warrants further investigation.

3.6 Abstracting the analysis

While not a tremendous success, we decide we want to carry on with this analysis. We would like to do this for other actors' movies. One of the big advantages of using R is that we can write a piece of code that takes an actor's name as input, and reproduces the steps of this analysis for that actor. We call these functions, we'll see them and use them a lot in this course.

For our analysis, this function must do the following:

1. Scrape movie ratings from Rotten Tomatoes
2. Clean up the scraped data
3. Join with the budget data we downloaded previously
4. Perform the clustering algorithm
5. Make the final plot

With this in mind, we can write functions for each of these steps, and then make one final function that puts all of these together.

For instance, let's write the scraping function. It will take an actor's name and output the scraped data.

Let's test it with Gael García Bernal:

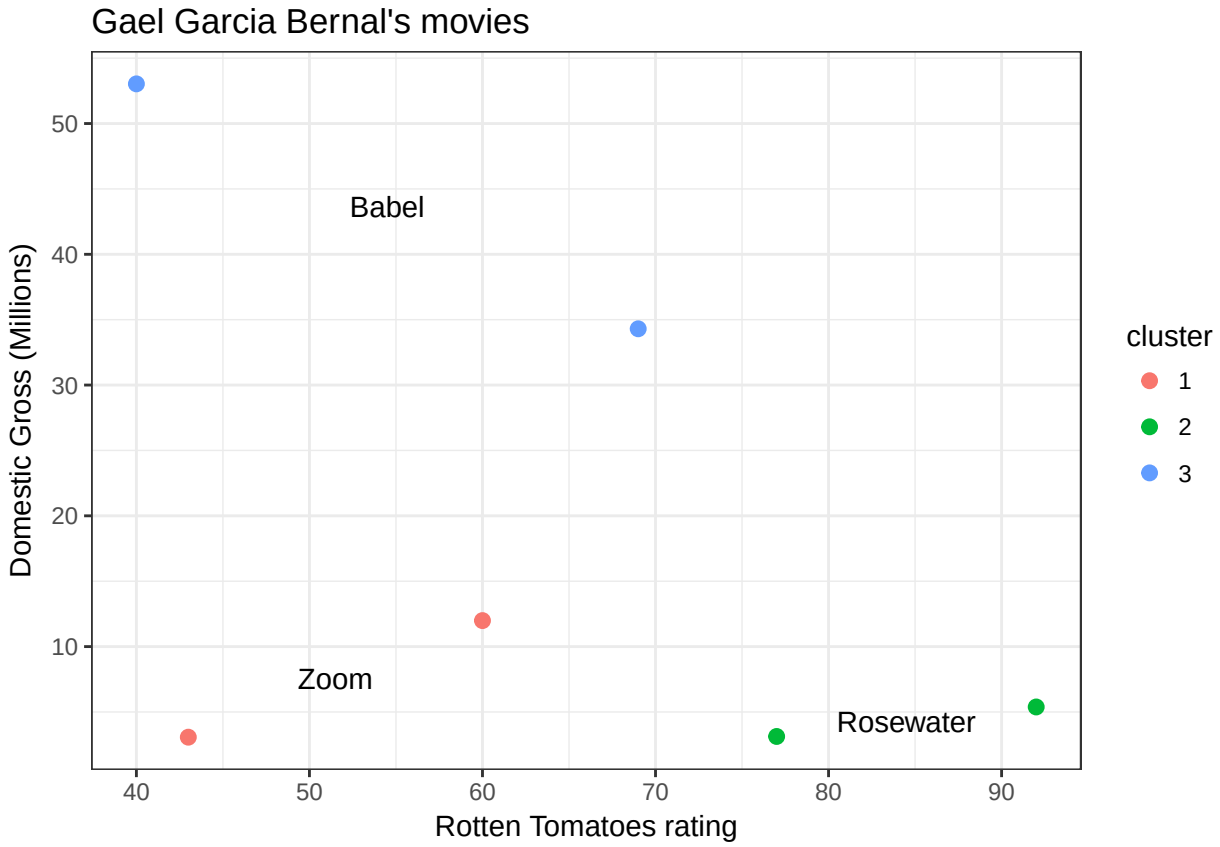
RATING	TITLE	CREDIT	BOX OFFICE	YEAR
30%	Salt and Fire	Dr. Fabio Cavani	—	2017
75%	You're Killing Me Susana (Me estás matando Susana)	Eligio	—	2017
94%	Neruda	Oscar Peluchonneau	$1M	2016

Good start. We can then write functions for each of the steps we did with Diego Luna before.

Then put all of these steps into one function that calls our new functions to put all of our analysis together:

We can test this with Gael García Bernal

```
analyze_actor("Gael Garcia Bernal")
```



3.7 Making analyses accessible

Now that we have written a function to analyze an actor's movies, we can make these analyses easier to produce by creating an interactive application that wraps our new function. The `shiny` R package makes creating this type of application easy.

3.8 Summary

In this analysis we saw examples of the common steps and operations in a data analysis:

1) Data ingestion: we scraped and cleaned data from publicly accessible sites
2) Data manipulation: we integrated data from multiple sources to prepare our analysis
3) Data visualization: we made plots to explore patterns in our data
4) Data modeling: we made a model to capture the grouping patterns in data automatically, using visualization to explore the results of this modeling
5) Publishing: we abstracted our analysis into an application that allows us and others to perform this analysis over more datasets and explore the result of modeling using a variety of parameters

Chapter 4

Setting up R

Here we setup R, RStudio and anything else we will use in the course.

4.1 Setting up R

R is a free, open source, environment for data analysis. It is available as a free binary download for Mac, Linux and Windows. For the more adventorous, it can also be compiled from source. To install R in your computer go to https://cran.r-project.org/index.html and download and install the appropriate binary file.

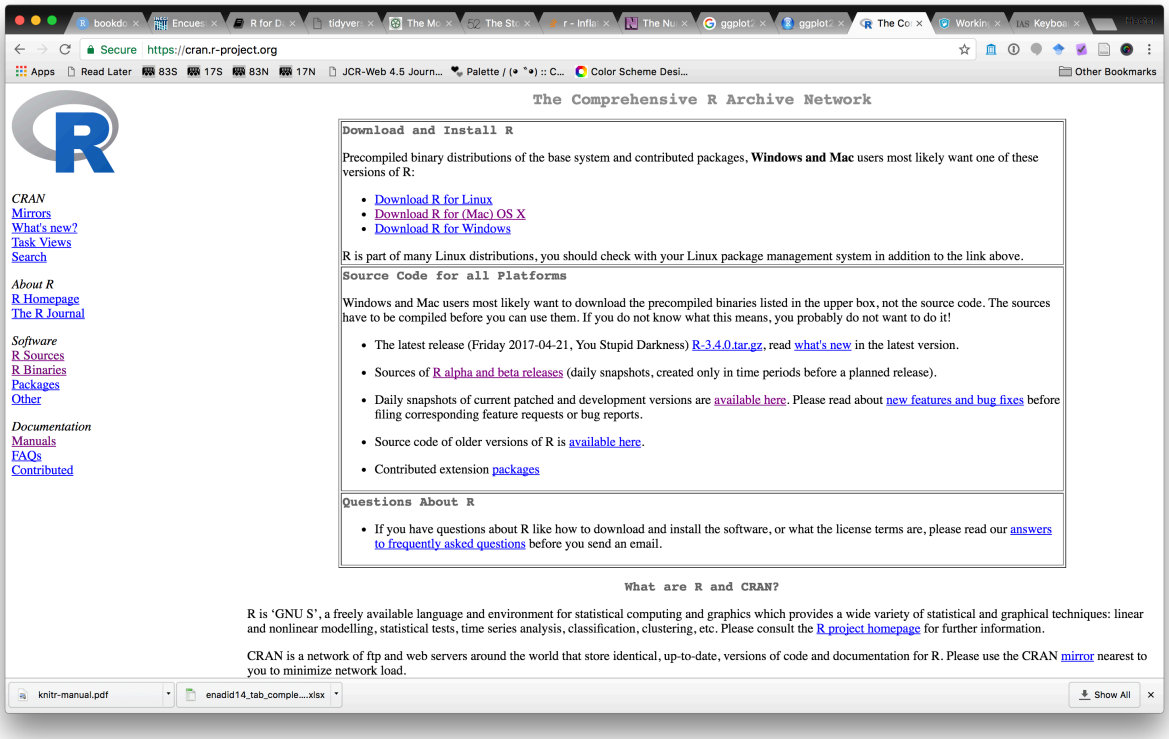


Figure 4.1:

This will install the base R system: the R programming language, a few packages for common data analyses and a development environment.

4.2 Setting up Rstudio

We will actually use Rstudio to interact with R. Rstudio is a very powerful application to make data analysis with R easier to do. To install go to https://www.rstudio.com/products/rstudio/download/ and download the appropriate version of Rstudio.

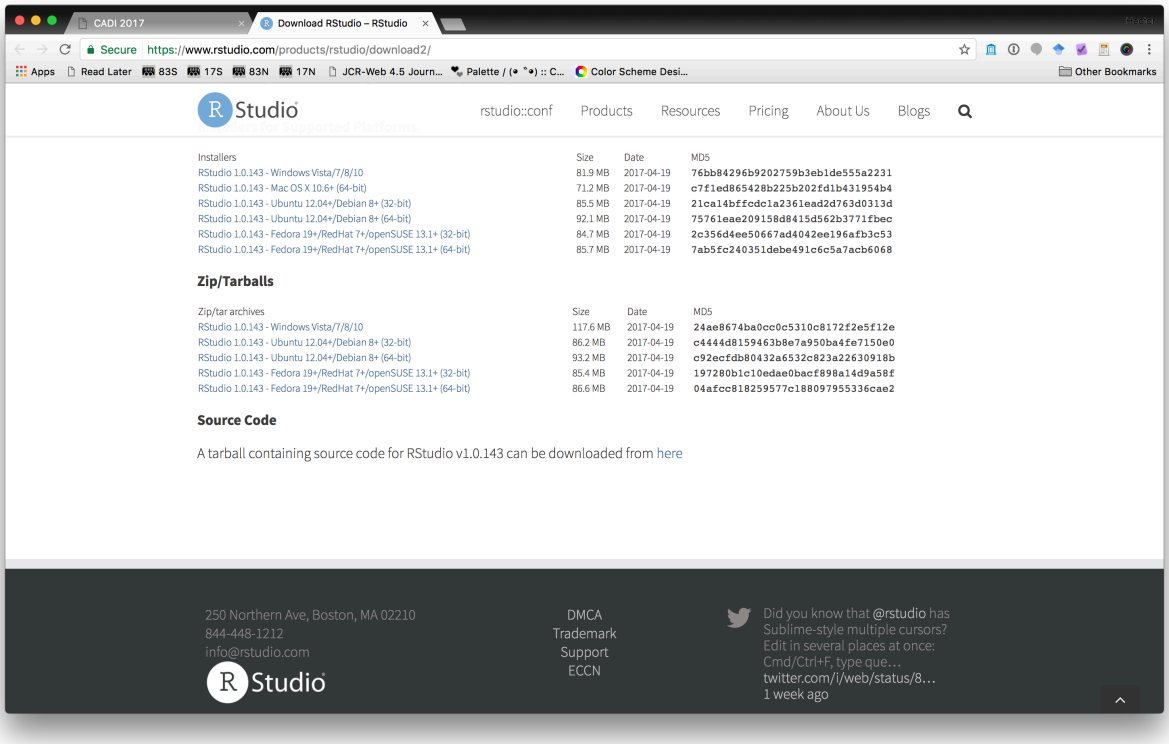


Figure 4.2:

4.3 A first look at Rstudio

Let's take a first look at Rstudio. The first thing you will notice is that Rstudio is divided into panes. Let's take a look first at the *Console*.

4.3.1 Interactive Console

The most immediate way to interact with R is through the interactive console. Here we can write R instructions to perform our data analyses. We want to start using data so the first instructions we will look at deal with loading data.

When you installed R, a few illustrative datasets were installed as well. Let's take a look at the list of datasets you now have access to. Write the following command in the console

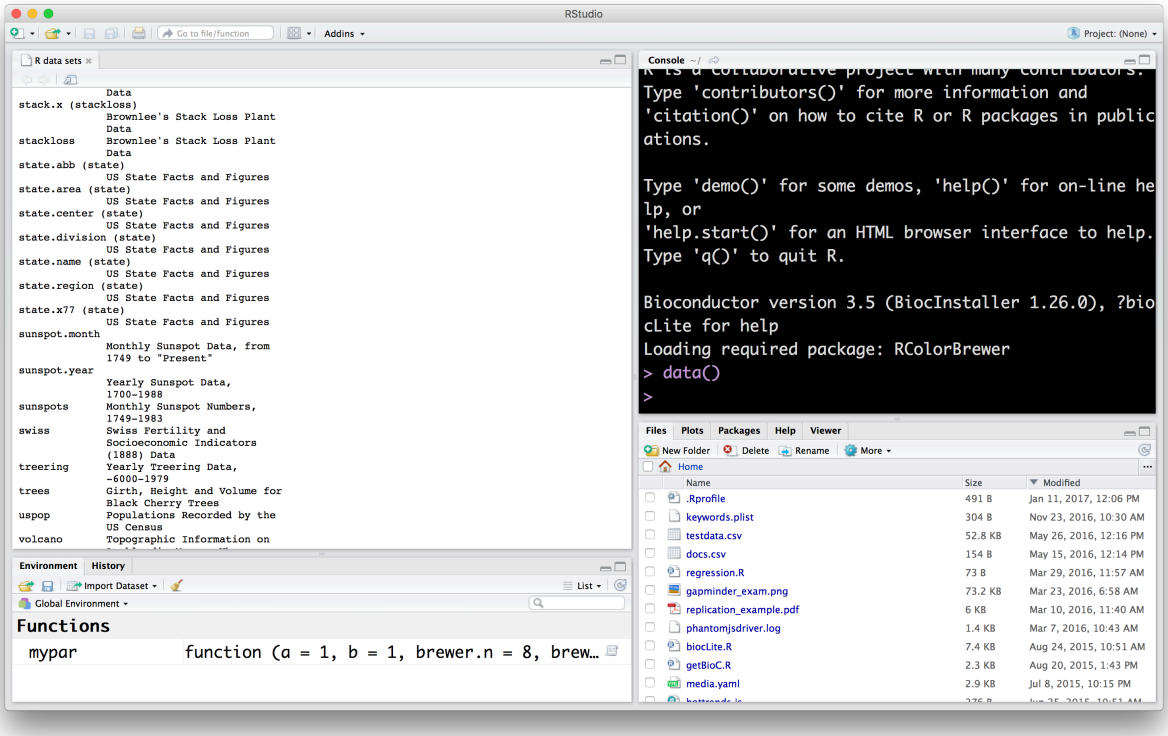


Figure 4.3:

This will list names and descriptions of datasets available in your R installation. Let's try to find out more information about these datasets. In R, the first attempt to get help with something is to use the `?` operation. So, to get help about the `swiss` dataset we can enter the following in the console

This will make the documentation for the `swiss` dataset open in another pane.

On your own: Find more information about a different dataset using the `?` operator.

4.3.2 Data Viewer

According to the documentation we just saw for `swiss`, this is a `data.frame` with 47 observations and 6 variables. The `data.frame` is the basic structure we will use to represent data throughout the course. We will see this again repeatedly, and use a couple of other names (e.g., `tibble`) to refer to this. Intuitively, you can think of the `data.frame` like a spreadsheet, with rows representing observations, and columns representing variables that describe those observations. Let's see what the `swiss` data looks like using the Rstudio data viewer.

The Data Viewer lets you reorder data by the values in a column. It also lets you filter rows of the data by values as well.

On your own: Use the Data Viewer to explore another of the datasets you saw listed before.

4.3.3 Names, values and functions

Let's make a very short pause to talk about something you may have noticed. In the console, we've now written a few instructions, e.g. `View(swiss)`. Let's take a closer look at how these instructions are put

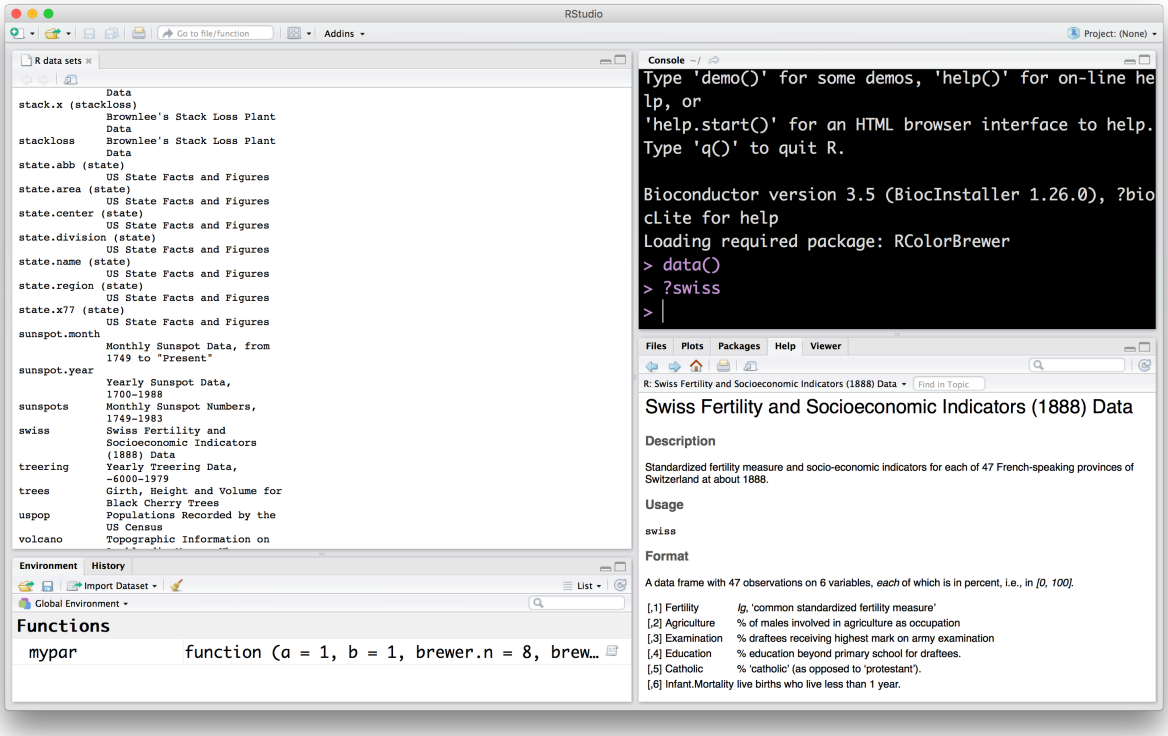


Figure 4.4:

together.

expressions: first of all, we call these instructions *expressions*, which are just text that R can evaluate into a value. `View(swiss)` is an expression.

values: so, what's a value? They are numbers, strings, data frames, etc. This is the data we will be working with. The number `2` is a value. So is the string `"Hector"`.

So, what value is produced when R evaluates the expression `View(swiss)`? Nothing, which we also treat as a value. That wasn't very interesting, but it does have a side effect: it shows the `swiss` dataset in the Data viewer.

How about a simpler expression: `swiss`, what value is produced when R evaluates the expression `swiss`? The data.frame containing that data. Try it out in the console.

names: so if `swiss` isn't a value, what is it? It is a *name*. We use these to refer to values. So, when we write the expression `swiss`, we tell R we want the *value* referenced by the name `swiss`, that is, the data itself!

functions: Besides numbers, strings, data frames, etc. another important type of value is the *function*. Functions are a series of instructions that take some input value and produce a different value. The name `View` refers to the function that takes a data frame as input, and displays it in the Data viewer. Functions are called using the parentheses we saw before: `View(swiss)`, the parentheses say that you are passing input `swiss` to the function `View`. We'll see later how we can write our own functions.

4.3.4 Plotting

Next, I want to show the *Plots* pane in Rstudio. Let's make a plot using the `swiss` dataset:

It's not pretty, but it was very easy to produce. There's a couple of things going on here…

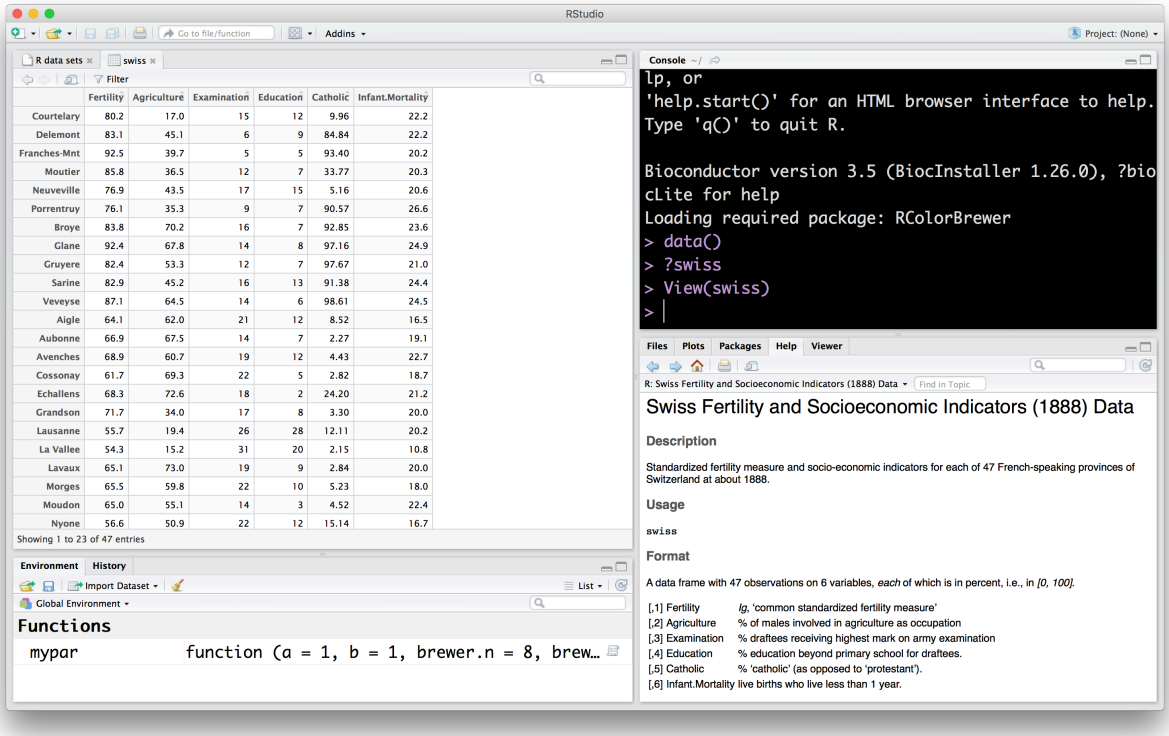


Figure 4.5:

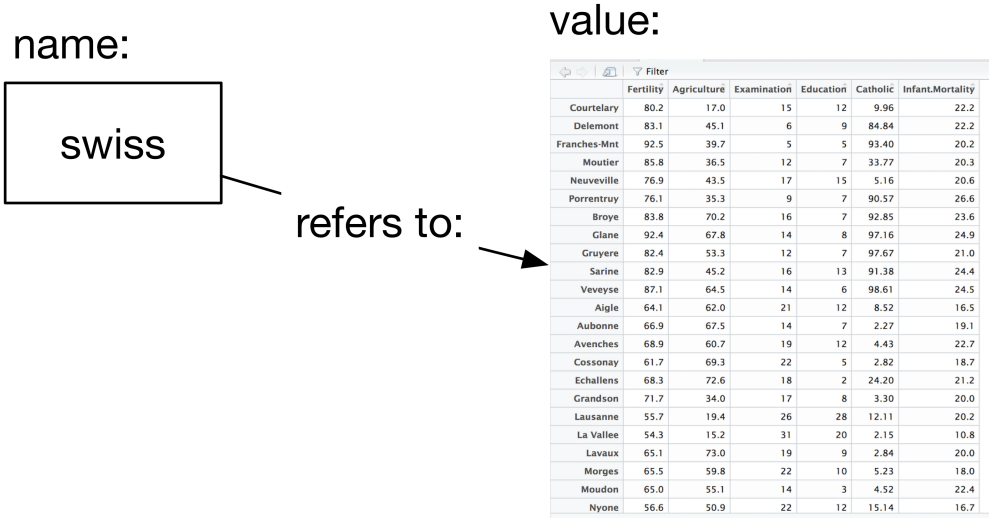


Figure 4.6:

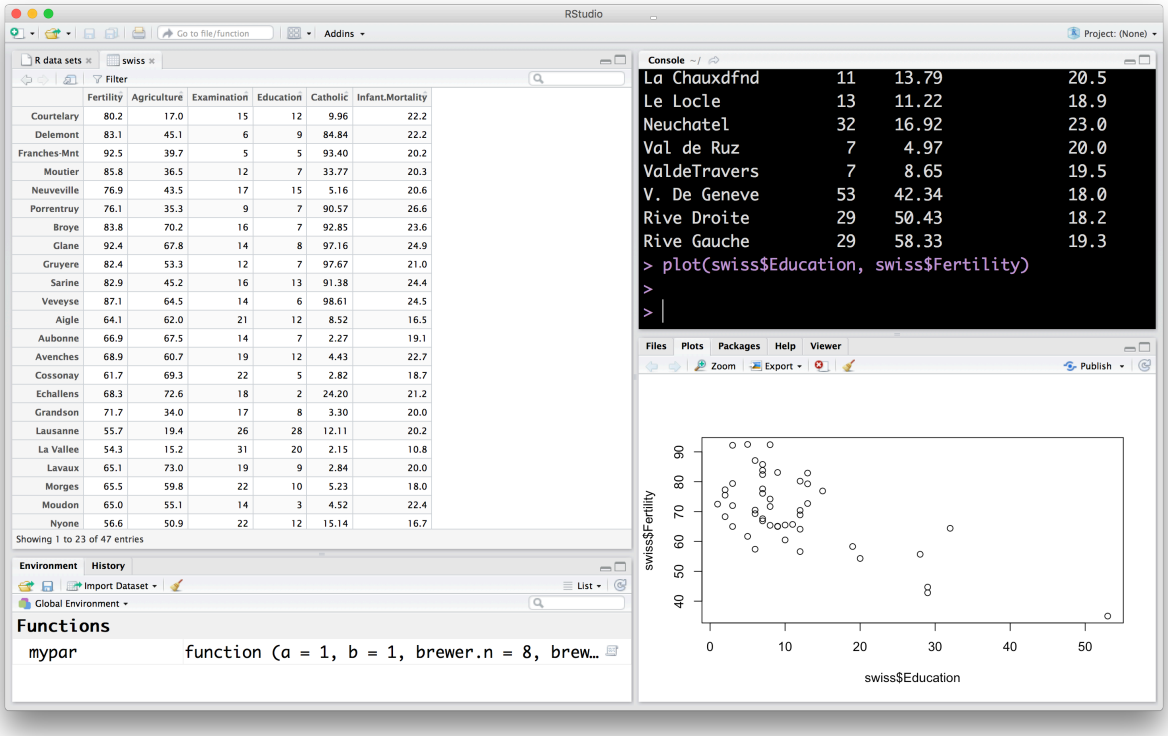


Figure 4.7:

- `plot` is a function, it takes two inputs, the data to put in the x and y axes, evaluates to nothing, but creates a plot of the data
- `swiss$Education` is how we refer to the `Education` column in the `swiss` data frame.

On your own: Make a plot using other variables in the `swiss` dataset.

4.3.5 Editor

So far, we've made some good progress: we know how to write expressions on the R console so that they are evaluated, we are starting to get a basic understanding of how these expressions are constructed, we can use the Data viewer to explore data frames, and made one plot that was displayed in the Plots pane. To finish this quick tour, I want to look at two more Rstudio panes: the file editor, and the File viewer.

As you have noticed, everytime we want to evaluate an expression on the console, we have to write it in. For example, if we want to change the plot we made above to include a different variable, we have to write the whole thing again. Also, what if I forgot what expression I used to make a specific plot? Even better, what if I wanted somebody else to make the plot I just made?

By far, one of the biggest advantages of using R over Excel or other similar programs, is that we can write expressions in scripts that are easy to share with others, making analyses easier to reproduce. Let's write a script that we can use to make the same plot we just made.

In the Rstudio menu select `File>New File>R Script`

This will open a tab in the File editor in which we can write expressions:

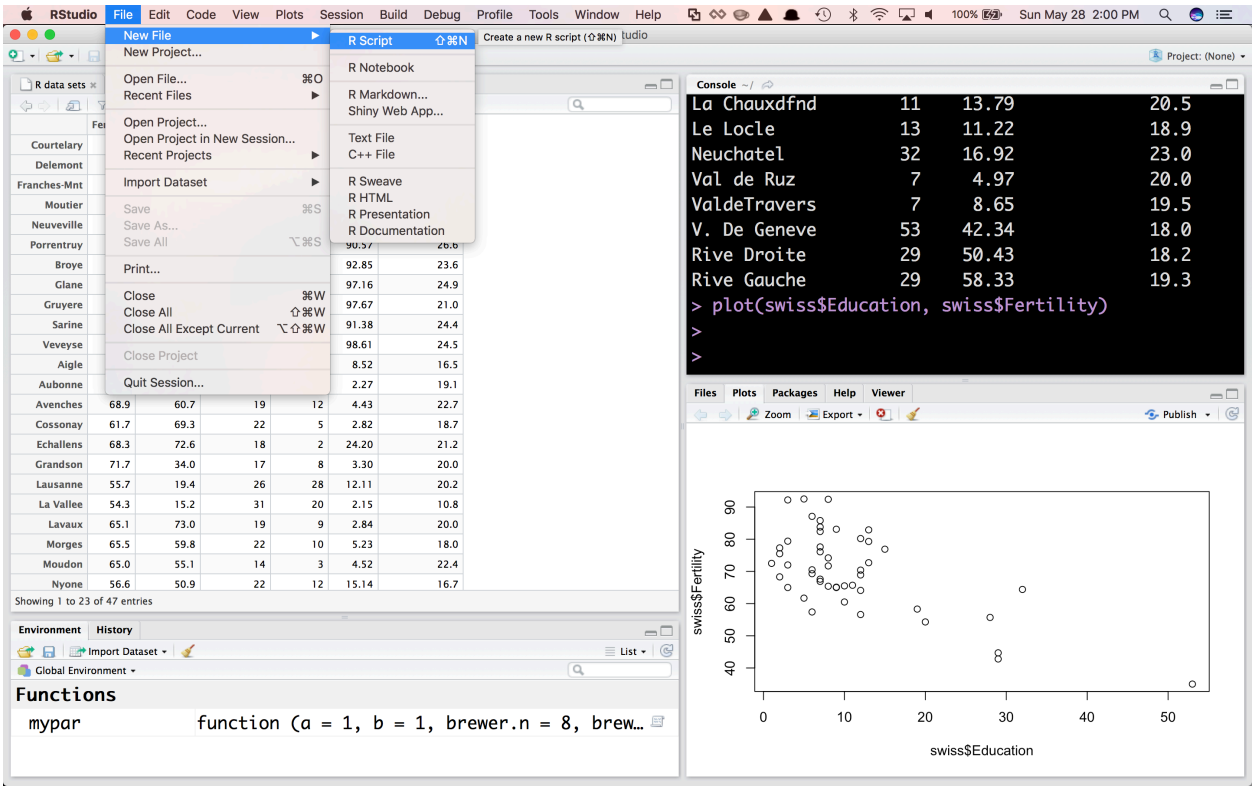


Figure 4.8:

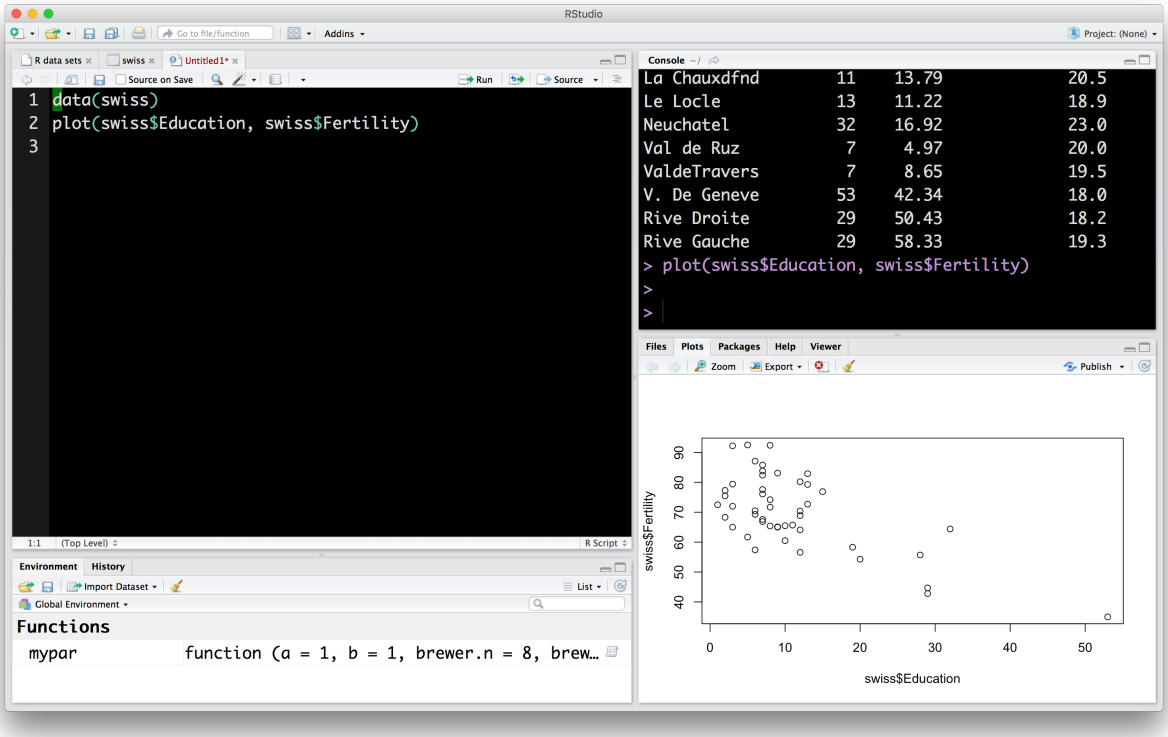


Figure 4.9:

We can then evaluate the expressions in the file one at a time, or all at the same time.

We can then save these expressions in a script. In the Rstudio menu select `File>Save` and save as a text file. The convention is to use the `.R` or `.r` file extension, e.g., `swiss_plot.r`.

On your own: Add expressions for additional plots to the script and save again. Run the new expressions.

4.3.6 Files viewer

Rstudio includes a Files viewer that you can use to find and load files. You can find the Files near the Plots viewer

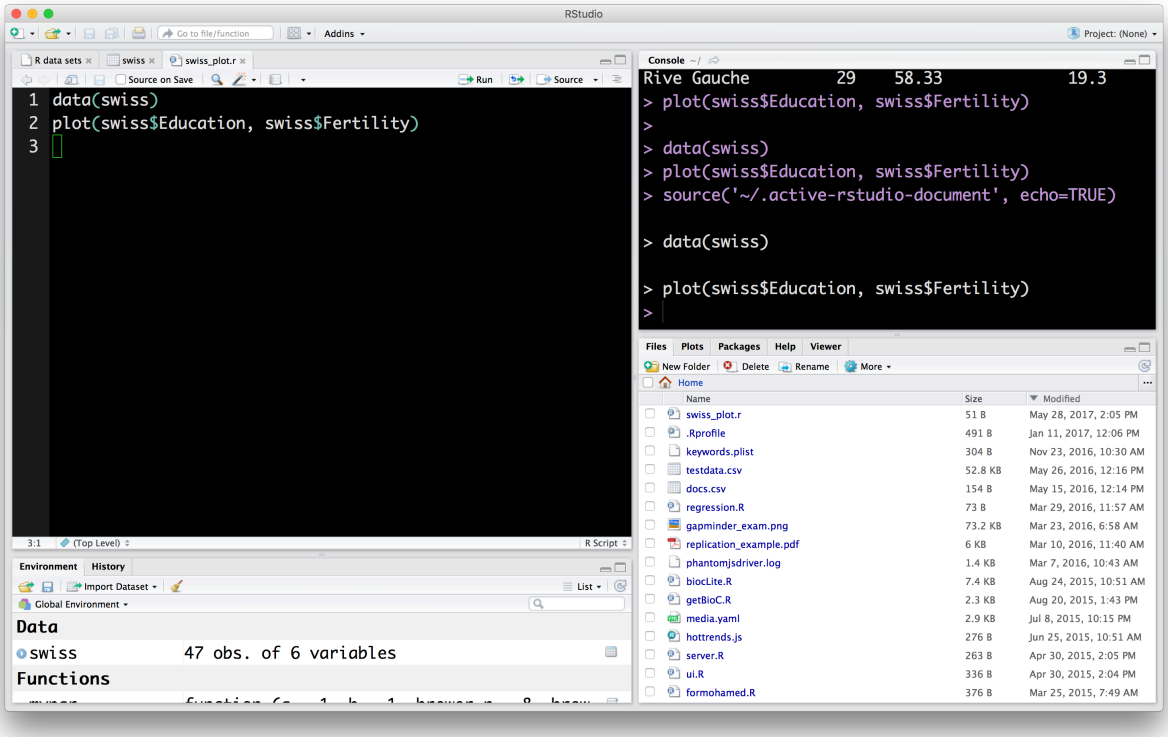


Figure 4.10:

4.4 R packages

Another of R's advantages for data analysis is that it has attracted a large number of extremely useful additions provided by users worldwide. These are housed in CRAN.

In this course we will make a lot of use of a set of packages bundled together into the `tidyverse` by Hadley Wickham and others. These packages make preparing, modeling and visualizing certain kinds data (which covers the vast majority of use cases) quite fun and pleasent. There is a webpage for the general tidyverse project: http://tidyverse.org, which includes pages for each of the packages included there.

Let's install the `tidyverse` into your R environment. There are two ways of installing packages. In the console, you can use the expression:

In Rstudio, you can use the *Packages* tab:

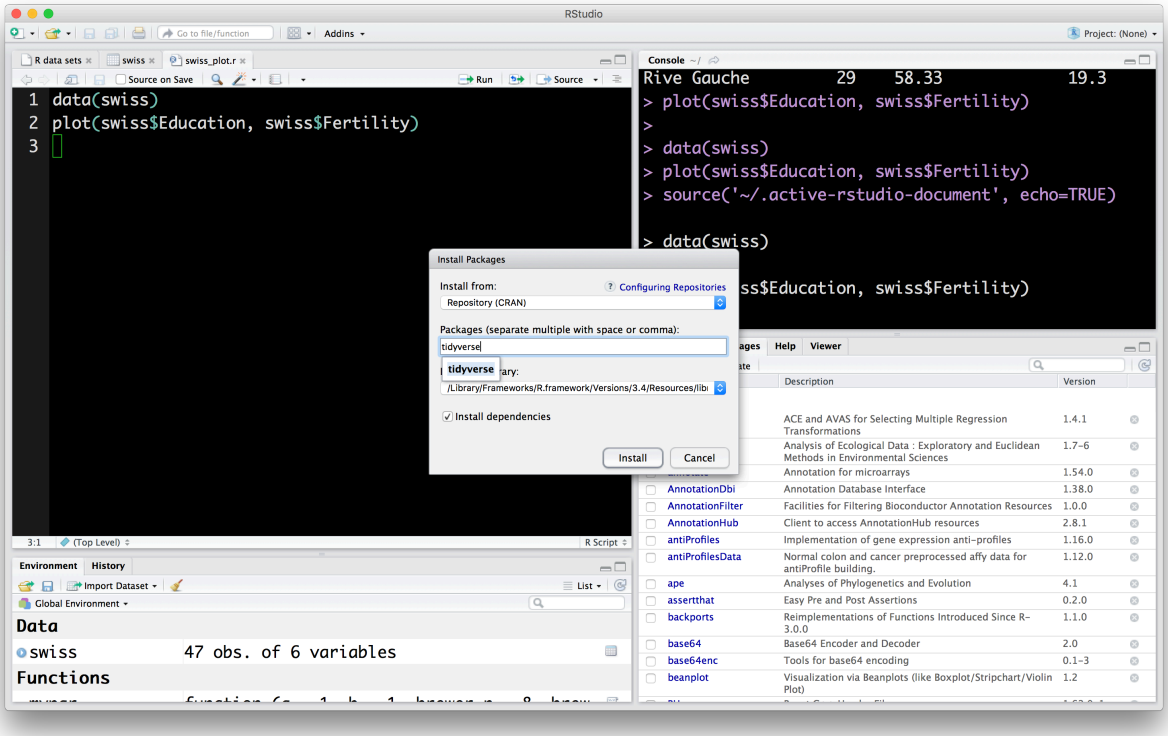


Figure 4.11:

On your own: Install the following additional packages which we will use later on: `rvest`, `stringr`, `nycflights13` and `broom`.

4.5 Finishing your setup

Go to the Google form as instructed and complete your exit ticket.

Chapter 5

R Principles

Now that we have our tools ready, let's start doing some analysis. First, let's go over some principles of R as a data analysis environment. R is a computational environment for data analysis. It is designed around a *functional* language, as opposed to *procedural* languages like Java or C, that has desirable properties for the type of operations and workflows that are frequently performed in the course of analyzing datasets. In this exercise we will start learning some of those desirable properties while performing an analysis of a real dataset.

5.1 Some history

R is an offspring of S, a language created in AT&T Labs by John Chambers (now at Stanford) and others in 1976 with the goal of creating an environment for statistical computing and data analysis. The standard for the language in current use was settled in 1998. That same year, "S" won the ACM Software System award, awarded to software systems "that have a lasting influence, reflected in contributions to concepts, in commercial acceptance, or both".

In 1991, Robert Gentleman and Ross Ihaka created R to provide an open source implementation of the S language and environment. They also redesigned the language to enforce lexical scoping rules. It has been maintained by the R core group since 1997, and in 2015 an R consortium, including Microsoft, Google, and others, was created.

Along with Python it is one of the most popular environments for data analysis (e.g., figure below from KDNuggets 2016 software survey)

We use it for this class because we find that besides it being a state-of-the-art data analysis environment, it provides a clean end-to-end platform for teaching material across the data management-modeling-communication spectrum that we study in class.

5.2 Additional R resources

Resources for learning and reading about R are listed in our here. Of note are the swirl project and DataCamp's [introduction to R] course.

One of the biggest strengths of the R ecosystem is the variety and quality of packages for data analysis available. R uses a package system (like Python and Ruby for instance). Packages are divided into two classes: **base** which are packages installed when R is installed, includes packages for basic statistics, computing with probability distributions, plotting and graphics, matrix manipulations and other), all other packages are available in CRAN. We will be using a fair number of these packages through the course of the semester.

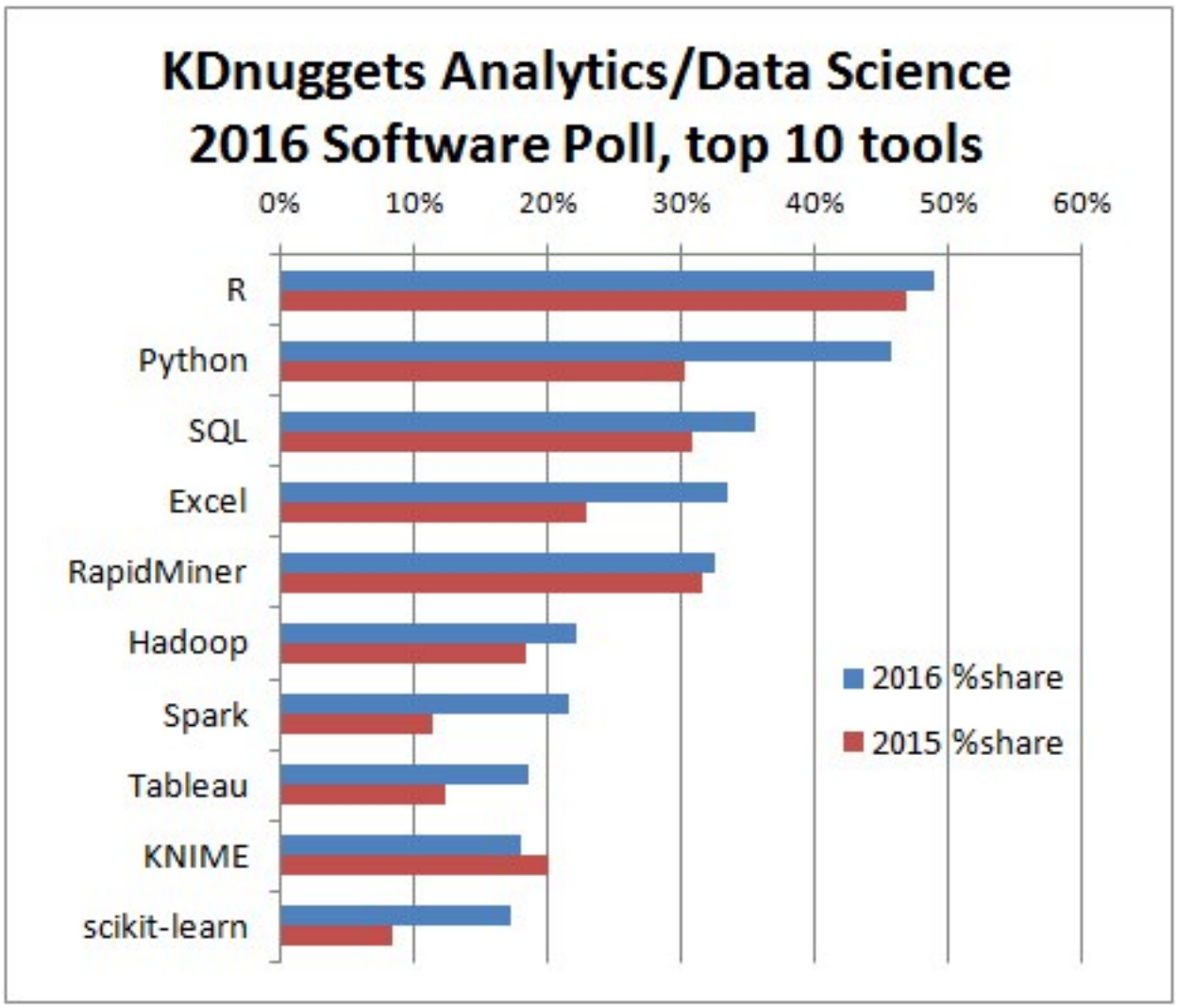

KDnuggets Analytics/Data Science
2016 Software Poll, top 10 tools
0%
10%
20%
30%
40%
50%
60%
R
Python
SQL
Excel
RapidMiner
Hadoop
Spark
Tableau
KNIME
scikit-learn
2016 %share
2015 %share

Figure 5.1:

5.3 Literate Programming

One last note before we get started. R has great support for literate programming, where source code that contains both code, the result of evaluating that code, and text explaining that code co-exist in a single document. This is extremely valuable in data analysis, as many choices made by data analysts are worth explaning in text, and interpretation of the results of analyses can co-exist with the computations used in that analysis. This document you are reading contains both text and code. In class, we will use Rmarkdown for this purpose.

5.4 A data analysis to get us going

I'm going to do a very simple analysis of Baltimore crime to show off R. We'll use data downloaded from Baltimore City's awesome open data site (this was downloaded a couple of years ago so if you download now, you will get different results).

The repository for this particular data is here. https://data.baltimorecity.gov/Crime/BPD-Arrests/3i3v-ibrt

5.5 Getting data

We've prepared the data previously into a comma-separated value file (`.csv` file). In this format, each line contains *attribute* values (separated by commas) for one *entity* in our dataset. Which we can download and load into our R environment.

The `read_csv` command is part of the `readr` R package and allows you to read a dataset stored in a csv file. This function is extremely versatile, and you can read more about it by using the standard help system in R: `?read_csv`. Now, the result of running calling this function is the data itself, so, by running the function in the console, the result of the function is printed.

5.6 Variables and Value

To make use of this dataset we want to assign the result of calling `read.csv` (i.e., the dataset) to a variable:

```
library(tidyverse)
arrest_tab <- read_csv("data/BPD_Arrests.csv")
```

```
## Parsed with column specification:
## cols(
##   arrest = col_integer(),
##   age = col_integer(),
##   sex = col_character(),
##   race = col_character(),
##   arrestDate = col_character(),
##   arrestTime = col_time(format = ""),
##   arrestLocation = col_character(),
##   incidentOffense = col_character(),
##   incidentLocation = col_character(),
##   charge = col_character(),
##   chargeDescription = col_character(),
##   district = col_character(),
```

```
##   post = col_integer(),
##   neighborhood = col_character(),
##   `Location 1` = col_character()
## )
```

Now we can ask what *type* of value is stored in the `arrest_tab` variable:

```
class(arrest_tab)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

The `data.frame` is a workhorse data structure in R. It encapsulates the idea of *entities* (in rows) and *attribute values* (in columns). We can ask other features of this dataset:

```
# This is a comment in R, by the way

# How many rows (entities) does this dataset contain?
nrow(arrest_tab)
```

```
## [1] 104528
```

```
# How many columns (attributes)?
ncol(arrest_tab)
```

```
## [1] 15
```

```
# What are the names of those columns?
colnames(arrest_tab)
```

```
##  [1] "arrest"           "age"              "sex"
##  [4] "race"             "arrestDate"       "arrestTime"
##  [7] "arrestLocation"   "incidentOffense"  "incidentLocation"
## [10] "charge"           "chargeDescription" "district"
## [13] "post"             "neighborhood"     "Location 1"
```

Now, in Rstudio you can view the data frame using `View(arrest_tab)`.

5.7 Indexing

A basic operation in data analysis is selecting subsets of a dataset. For that we can use a few alternative options for *indexing* into datasets.

```
# to obtain the value in the first row, fifth column:
arrest_tab[1,5]
```

```
## # A tibble: 1 x 1
##   arrestDate
##        <chr>
## 1 01/01/2011
```

```
# note that indexing in R is 1-based, not 0-based, so the first row is indexed by 1

# now we want to do a bit more, so let's say we want the value in the fifth column of our dataset for t
arrest_tab[1:10,5]
```

```
## # A tibble: 10 x 1
##    arrestDate
##         <chr>
##  1 01/01/2011
##  2 01/01/2011
##  3 01/01/2011
##  4 01/01/2011
##  5 01/01/2011
##  6 01/01/2011
##  7 01/01/2011
##  8 01/01/2011
##  9 01/01/2011
## 10 01/01/2011
```

```
# similarly, to obtain the value in the first five columns of the first row
arrest_tab[1,1:5]
```

```
## # A tibble: 1 x 5
##     arrest   age    sex   race arrestDate
##      <int> <int> <fctr> <fctr>      <chr>
## 1 11126858    23      B      M 01/01/2011
```

```
# what is the class of the value when we subset a single column?
class(arrest_tab[1:10,5])
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

```
# what is the class of the value when we subset a single row?
class(arrest_tab[1,1:5])
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

```
# what do we get with this indexing?
arrest_tab[1:10,1:5]
```

```
## # A tibble: 10 x 5
##      arrest   age    sex   race arrestDate
##       <int> <int> <fctr> <fctr>      <chr>
##  1 11126858    23      B      M 01/01/2011
##  2 11127013    37      B      M 01/01/2011
##  3 11126887    46      B      M 01/01/2011
##  4 11126873    50      B      M 01/01/2011
##  5 11126968    33      B      M 01/01/2011
##  6 11127041    41      B      M 01/01/2011
##  7 11126932    29      B      M 01/01/2011
##  8 11126940    20      W      M 01/01/2011
##  9 11127051    24      B      M 01/01/2011
## 10 11127018    53      B      M 01/01/2011
```

We can index any set of rows or columns by constructing *vectors* of integers. In fact, the slice notation `:` is essentially doing that for a sequence of consecutive indices. You should think of vectors as lists of values with the same class.

If we want non-consecutive indices we have other options (e.g., the `c` function, for "concatenate")

```
# non-consecutive indices using c
arrest_tab[c(2,4,7,10), 1:5]
```

```
## # A tibble: 4 x 5
##     arrest   age    sex   race arrestDate
##      <int> <int> <fctr> <fctr>      <chr>
## 1 11127013    37      B      M 01/01/2011
## 2 11126873    50      B      M 01/01/2011
## 3 11126932    29      B      M 01/01/2011
## 4 11127018    53      B      M 01/01/2011
```

```
# here's a fun one, when we call columns for a subset of rows
arrest_tab[c(2,4,7,10), ]
```

```
## # A tibble: 4 x 15
##     arrest   age    sex   race arrestDate arrestTime    arrestLocation
##      <int> <int> <fctr> <fctr>      <chr>     <time>             <chr>
## 1 11127013    37      B      M 01/01/2011   00:01:00  2000 Wilkens Ave
## 2 11126873    50      B      M 01/01/2011   00:04:00 2100 Ashburton St
## 3 11126932    29      B      M 01/01/2011   00:05:00   800 N Monroe St
## 4 11127018    53      B      M 01/01/2011   00:15:00 3300 Woodland Ave
## # ... with 8 more variables: incidentOffense <fctr>,
## #   incidentLocation <chr>, charge <chr>, chargeDescription <chr>,
## #   district <chr>, post <int>, neighborhood <chr>, `Location 1` <chr>
```

```
# there is also the `seq` function, to create sequences
arrest_tab[seq(from=1,to=10), seq(1,10)]
```

```
## # A tibble: 10 x 10
##      arrest   age    sex   race arrestDate arrestTime       arrestLocation
##       <int> <int> <fctr> <fctr>      <chr>     <time>                <chr>
##  1 11126858    23      B      M 01/01/2011   00:00:00                 <NA>
##  2 11127013    37      B      M 01/01/2011   00:01:00     2000 Wilkens Ave
##  3 11126887    46      B      M 01/01/2011   00:01:00    2800 Mayfield Ave
##  4 11126873    50      B      M 01/01/2011   00:04:00    2100 Ashburton St
##  5 11126968    33      B      M 01/01/2011   00:05:00      4000 Wilsby Ave
##  6 11127041    41      B      M 01/01/2011   00:05:00     2900 Spellman Rd
##  7 11126932    29      B      M 01/01/2011   00:05:00      800 N Monroe St
##  8 11126940    20      W      M 01/01/2011   00:05:00      5200 Moravia Rd
##  9 11127051    24      B      M 01/01/2011   00:07:00 2400 Gainsdbourgh Ct
## 10 11127018    53      B      M 01/01/2011   00:15:00    3300 Woodland Ave
## # ... with 3 more variables: incidentOffense <fctr>,
## #   incidentLocation <chr>, charge <chr>
```

```
# that is equivalent to
arrest_tab[1:10,1:10]
```

```
## # A tibble: 10 x 10
##      arrest   age    sex   race arrestDate arrestTime       arrestLocation
##       <int> <int> <fctr> <fctr>      <chr>     <time>                <chr>
##  1 11126858    23      B      M 01/01/2011   00:00:00                <NA>
##  2 11127013    37      B      M 01/01/2011   00:01:00     2000 Wilkens Ave
##  3 11126887    46      B      M 01/01/2011   00:01:00    2800 Mayfield Ave
##  4 11126873    50      B      M 01/01/2011   00:04:00    2100 Ashburton St
##  5 11126968    33      B      M 01/01/2011   00:05:00      4000 Wilsby Ave
##  6 11127041    41      B      M 01/01/2011   00:05:00     2900 Spellman Rd
##  7 11126932    29      B      M 01/01/2011   00:05:00      800 N Monroe St
##  8 11126940    20      W      M 01/01/2011   00:05:00      5200 Moravia Rd
##  9 11127051    24      B      M 01/01/2011   00:07:00 2400 Gainsdbourgh Ct
## 10 11127018    53      B      M 01/01/2011   00:15:00    3300 Woodland Ave
## # ... with 3 more variables: incidentOffense <fctr>,
## #   incidentLocation <chr>, charge <chr>
```

```
# with the `seq` function you can do more sophisticated things like select only entries in odd rows (1,
head(arrest_tab[seq(from=1,to=nrow(arrest_tab),by=2), ])
```

```
## # A tibble: 6 x 15
##     arrest   age    sex   race arrestDate arrestTime       arrestLocation
##      <int> <int> <fctr> <fctr>      <chr>     <time>                <chr>
## 1 11126858    23      B      M 01/01/2011   00:00:00                <NA>
## 2 11126887    46      B      M 01/01/2011   00:01:00    2800 Mayfield Ave
## 3 11126968    33      B      M 01/01/2011   00:05:00      4000 Wilsby Ave
## 4 11126932    29      B      M 01/01/2011   00:05:00      800 N Monroe St
## 5 11127051    24      B      M 01/01/2011   00:07:00 2400 Gainsdbourgh Ct
## 6 11127057    28      B      M 01/01/2011   00:15:00    3300 Woodland Ave
## # ... with 8 more variables: incidentOffense <fctr>,
## #   incidentLocation <chr>, charge <chr>, chargeDescription <chr>,
## #   district <chr>, post <int>, neighborhood <chr>, `Location 1` <chr>
```

Now, since columns have names, we can also use strings (and vectors of strings) to index data frames.

```
# single column
arrest_tab[1:10, "age"]
```

```
## # A tibble: 10 x 1
##      age
##    <int>
##  1    23
##  2    37
##  3    46
##  4    50
##  5    33
##  6    41
##  7    29
##  8    20
##  9    24
## 10    53
```

```
# multiple columns
arrest_tab[1:10, c("age", "sex", "race")]
```

```
## # A tibble: 10 x 3
##      age    sex   race
##    <int> <fctr> <fctr>
##  1    23      B      M
##  2    37      B      M
##  3    46      B      M
##  4    50      B      M
##  5    33      B      M
##  6    41      B      M
##  7    29      B      M
##  8    20      W      M
##  9    24      B      M
## 10    53      B      M
```

If we wanted a single named column from a data frame there's a special operator $ to index:

```
# first ten values of the age column
arrest_tab$age[1:10]
```

```
##  [1] 23 37 46 50 33 41 29 20 24 53
```

```
# EXERCISE
# try using three different ways of selecting rows 20 to 30 # of the "sex" column
```

In addition to integer indices or names, we can use vectors of logical values for indexing.

```
# rows 2,4,7 and 10 using logical indices
arrest_tab[c(FALSE,TRUE,FALSE,TRUE,FALSE,FALSE,TRUE,FALSE,FALSE,TRUE,rep(FALSE,nrow(arrest_tab)-10)),]
```

```
## # A tibble: 4 x 15
##     arrest   age    sex   race arrestDate arrestTime    arrestLocation
##      <int> <int> <fctr> <fctr>      <chr>     <time>             <chr>
## 1 11127013    37      B      M 01/01/2011   00:01:00  2000 Wilkens Ave
## 2 11126873    50      B      M 01/01/2011   00:04:00 2100 Ashburton St
## 3 11126932    29      B      M 01/01/2011   00:05:00   800 N Monroe St
## 4 11127018    53      B      M 01/01/2011   00:15:00 3300 Woodland Ave
## # ... with 8 more variables: incidentOffense <fctr>,
## #   incidentLocation <chr>, charge <chr>, chargeDescription <chr>,
## #   district <chr>, post <int>, neighborhood <chr>, `Location 1` <chr>
```

```
# now here's a fun one, if we only wanted odd rows
head(arrest_tab[c(TRUE,FALSE),])
```

```
## # A tibble: 6 x 15
##     arrest   age    sex   race arrestDate arrestTime       arrestLocation
##      <int> <int> <fctr> <fctr>      <chr>     <time>                <chr>
## 1 11126858    23      B      M 01/01/2011   00:00:00                 <NA>
```

```
## 2 11126887    46      B      M 01/01/2011   00:01:00    2800 Mayfield Ave
## 3 11126968    33      B      M 01/01/2011   00:05:00      4000 Wilsby Ave
## 4 11126932    29      B      M 01/01/2011   00:05:00      800 N Monroe St
## 5 11127051    24      B      M 01/01/2011   00:07:00 2400 Gainsdbourgh Ct
## 6 11127057    28      B      M 01/01/2011   00:15:00    3300 Woodland Ave
## # ... with 8 more variables: incidentOffense <fctr>,
## #   incidentLocation <chr>, charge <chr>, chargeDescription <chr>,
## #   district <chr>, post <int>, neighborhood <chr>, `Location 1` <chr>
```

The last example shows one of the most common gotchas in R. Indices are recycled. For instance if selecting rows, if you pass a logical vector that's shorter than the number of rows in the data frame, the vector will be recycled as many times as necessary to match the number of rows in the dataset. Now, why is this useful, because a pithy index vector can let you select easily. Why is this bad, because errors in code can go easily unnoticed. So in this case, the price of ease of use is paid by the programmer by having to think a lot more carefully about their code (this is a theme in R programming…)

The utility of logical indexing is that now we can select rows based on a property of its values for a given column

```
# select rows for entities younger than 21 years old
head(arrest_tab[arrest_tab$age < 21, ])
```

```
## # A tibble: 6 x 15
##     arrest   age    sex   race arrestDate arrestTime     arrestLocation
##      <int> <int> <fctr> <fctr>      <chr>     <time>              <chr>
## 1 11126940    20      W      M 01/01/2011   00:05:00    5200 Moravia Rd
## 2 11126942    20      B      M 01/01/2011   01:22:00 1900 Ashburton Ave
## 3 11126941    20      W      M 01/01/2011   02:00:00  300 S Bentalou St
## 4 11126955    20      B      M 01/01/2011   02:20:00     900 Myrtle Ave
## 5 11127139    19      W      M 01/01/2011   02:22:00    4500 Erdman Ave
## 6 11127003    20      B      F 01/01/2011   02:49:00  2400 Brentwood St
## # ... with 8 more variables: incidentOffense <fctr>,
## #   incidentLocation <chr>, charge <chr>, chargeDescription <chr>,
## #   district <chr>, post <int>, neighborhood <chr>, `Location 1` <chr>
```

```
# notice that the value of expression `arrest_tab$age < 21` # is a logical vector

# select entities (arrests) occuring in Mount Washington,
# a specific neighborhood in Baltimore
head(arrest_tab[arrest_tab$neighborhood == "Mount Washington",])
```

```
## # A tibble: 6 x 15
##   arrest   age    sex   race arrestDate arrestTime arrestLocation
##    <int> <int> <fctr> <fctr>      <chr>     <time>          <chr>
## 1     NA    NA     NA     NA       <NA>         NA           <NA>
## 2     NA    NA     NA     NA       <NA>         NA           <NA>
## 3     NA    NA     NA     NA       <NA>         NA           <NA>
## 4     NA    NA     NA     NA       <NA>         NA           <NA>
## 5     NA    NA     NA     NA       <NA>         NA           <NA>
## 6     NA    NA     NA     NA       <NA>         NA           <NA>
## # ... with 8 more variables: incidentOffense <fctr>,
## #   incidentLocation <chr>, charge <chr>, chargeDescription <chr>,
## #   district <chr>, post <int>, neighborhood <chr>, `Location 1` <chr>
```

```
# how about arrests where subjects are under 21 in Mount  Washington?
# use a logical `and` operator
indices <- arrest_tab$age < 21 & arrest_tab$neighborhood == "Mount Washington"
```

5.8 Exploration

R has built-in functions that help easily obtain summary information about datasets. For instance:

```
summary(arrest_tab$sex)
```

```
##     A     B     H     I     U     W  NA's
##   242 87268     1   218  1749 15048     2
```

```
summary(arrest_tab$race)
```

```
##     F     M  NA's
## 19431 85095     2
```

```
# well that seems problematic
# let's rename columns to correct that
colnames(arrest_tab)[3:4] <- c("race", "sex")
```

We can also ask other useful type of summaries

```
# What is the average age in arrests?
mean(arrest_tab$age)
```

```
## [1] 33.19639
```

```
# Median age?
median(arrest_tab$age)
```

```
## [1] 30
```

```
# what types of offenses are there
summary(arrest_tab$incidentOffense)
```

```
##                   Unknown Offense                       87-Narcotics
##                             38649                              24744
##                 4E-Common Assault           87O-Narcotics (Outside)
##                              6739                               6515
##               97-Search & Seizure                           79-Other
##                              3670                               3461
##                  24-Towed Vehicle           6C-Larceny- Shoplifting
##                              2994                               1849
##              4C-Agg. Asslt.- Oth.                  55A-Prostitution
##                              1556                               1398
##               4B-Agg. Asslt.- Cut              55-Disorderly Person
```

```
##                          1195                          923
##                115-Trespassing           5A-Burg. Res. (Force)
##                           871                          847
##         75-Destruct. Of Property            4D-Agg. Asslt.- Hand
##                           686                          618
##          61-Person Wanted On War            3B-Robb Highway (Ua)
##                           482                          410
##                54-Armed Person                 7A-Stolen Auto
##                           394                          392
##             4A-Agg. Asslt.- Gun           6D-Larceny- From Auto
##                           356                          336
##              6J-Larceny- Other            3AF-Robb Hwy-Firearm
##                           312                          277
##            49-Family Disturbance           26-Recovered Vehicle
##                           253                          249
##          6G-Larceny- From Bldg.                   20A-Followup
##                           248                          246
##                    78-Gambling           5D-Burg. Oth. (Force)
##                           219                          196
##               3K-Robb Res. (Ua)           111-Protective Order
##                           176                          152
##             4F-Assault By Threat                  109-Loitering
##                           152                          131
##                       117-Fto          5C-Burg. Res. (Noforce)
##                           130                          122
##              3AK-Robb Hwy-Knife           5B-Burg. Res. (Att.)
##                           102                          102
##            81-Recovered Property    108-Liquor Law/Open Container
##                            94                           92
##              3D-Robb Comm. (Ua)                2A-Rape (Force)
##                            92                           89
##             6E-Larceny- Auto Acc     112-Traffic Related Incident
##                            85                           81
##              23-Unauthorized Use              88-Unfounded Call
##                            81                           80
##           3AO-Robb Hwy-Other Wpn           7C-Stolen Veh./Other
##                            79                           78
##                2H-Indecent Exp.                      1A-Murder
##                            72                           64
##         71-Sex Offender Registry        48-Involuntary Detention
##                            64                           57
##                   114-Hindering        3AJF-Robb Carjack-Firearm
##                            56                           56
##                2F-Placing Hands              73-False Pretense
##                            54                           54
##         6B-Larceny- Purse Snatch                    95-Exparte
##                            51                           51
##            3CF-Robb Comm-Firearm       3JF-Robb Residence-Firearm
##                            43                           43
##                56-Missing Person               98-Child Neglect
##                            41                           41
##               3P-Robb Misc. (Ua)              58-Injured Person
##                            35                           35
##                   85-Mental Case           3BJ-Robb Carjack(Ua)
```

```
##                              34                              33
##           5F-Burg. Oth. (Noforce)              6F-Larceny- Bicycle
##                              31                              31
##              2G-Sodomy/Perverson          3JK-Robb Residence-Knife
##                              30                              30
##              3CK-Robb Comm-Knife                 80-Lost Property
##                              25                              25
##                2B-Rape (Attempt)           29-Driving While Intox.
##                              21                              20
##             3NF-Robb Misc-Firearm              5E-Burg. Oth. (Att.)
##                              20                              18
##                2D-Statutory Rape      3JO-Robb Residence-Other Wpn
##                              17                              17
##           67-Child Abuse-Physical               103-Dead On Arrival
##                              17                              15
##           3CO-Robb Comm-Other Wpn               3NK-Robb Misc-Knife
##                              15                              15
##                     113-Littering                           39-Fire
##                              14                              14
##             76-Child Abuse-Sexual         8AO-Arson Sin Res Str-Occ
##                              14                              14
##                   96-Stop & Frisk 116-Public Urination / Defecation
##                              14                              13
##                110-Summons Served               20H-Traffic Control
##                              12                              11
##           3AJK-Robb Carjack-Knife       3GF-Robb Conv Store-Firearm
##                              11                              11
##               106-Custody Dispute                52A-Animal Cruelty
##                              10                              10
##                  70A-Ill. Dumping            83-Discharging Firearm
##                              10                              10
##           3H-Robb Conv. Stor.(Ua)           3NO-Robb Misc-Other Wpn
##                               9                               8
##              93-Abduction - Other                         (Other)
##                               8                             101
```

```
# what does summary looks like for continuous attributes?
summary(arrest_tab$age)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     0.0    23.0    30.0    33.2    43.0    87.0
```

Combining this type of summary with our indexing strategies we learned previously we can ask more specific questions

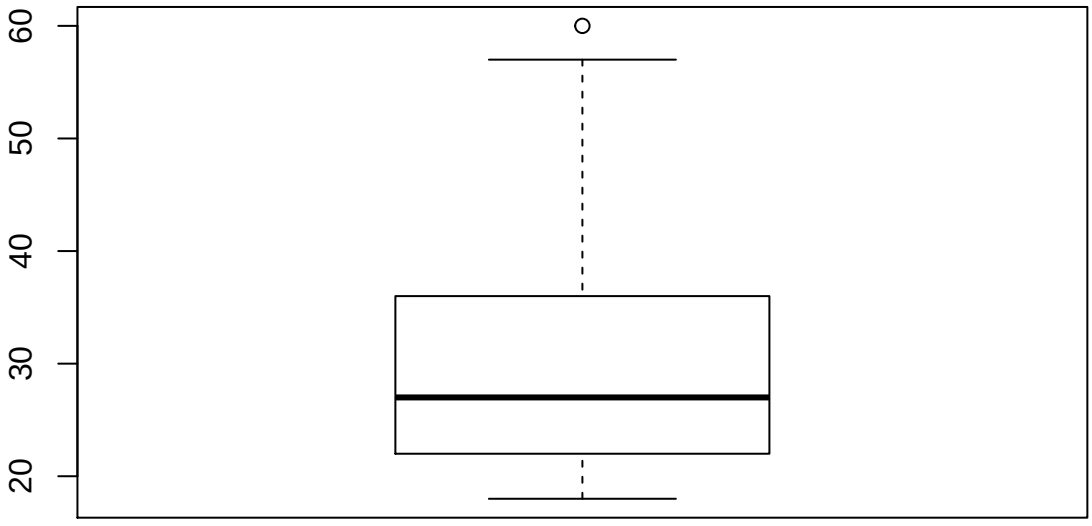
```
# What is the average age for arrests in Mount Washington?
mount_washington_index <- arrest_tab$neighborhood == "Mount Washington"

mean(arrest_tab$age[mount_washington_index], na.rm=TRUE)
```

```
## [1] 31.10345
```

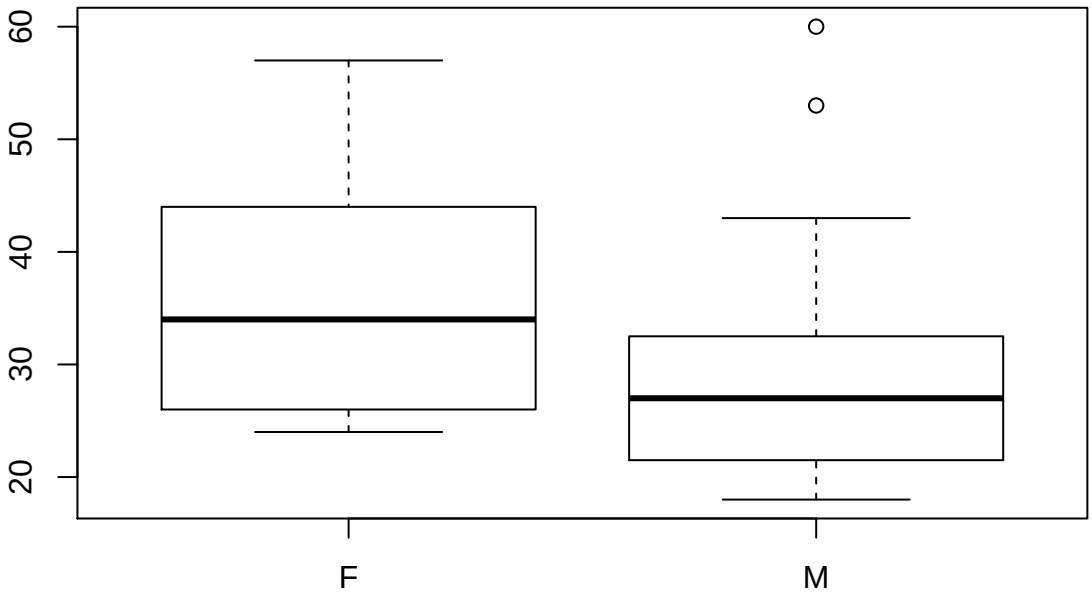
```
# How about the number of arrests in Mount Washington _stratified_ by race and sex?
table(arrest_tab$race[mount_washington_index], arrest_tab$sex[mount_washington_index])
```

```
##
##      F  M
##   A  0  0
##   B  4 14
##   H  0  0
##   I  0  0
##   U  0  0
##   W  2  9
```

```
# how about a graphical summary of arrest ages in Mount Washington?
# we'll use a boxplot
boxplot(arrest_tab$age[mount_washington_index])
```



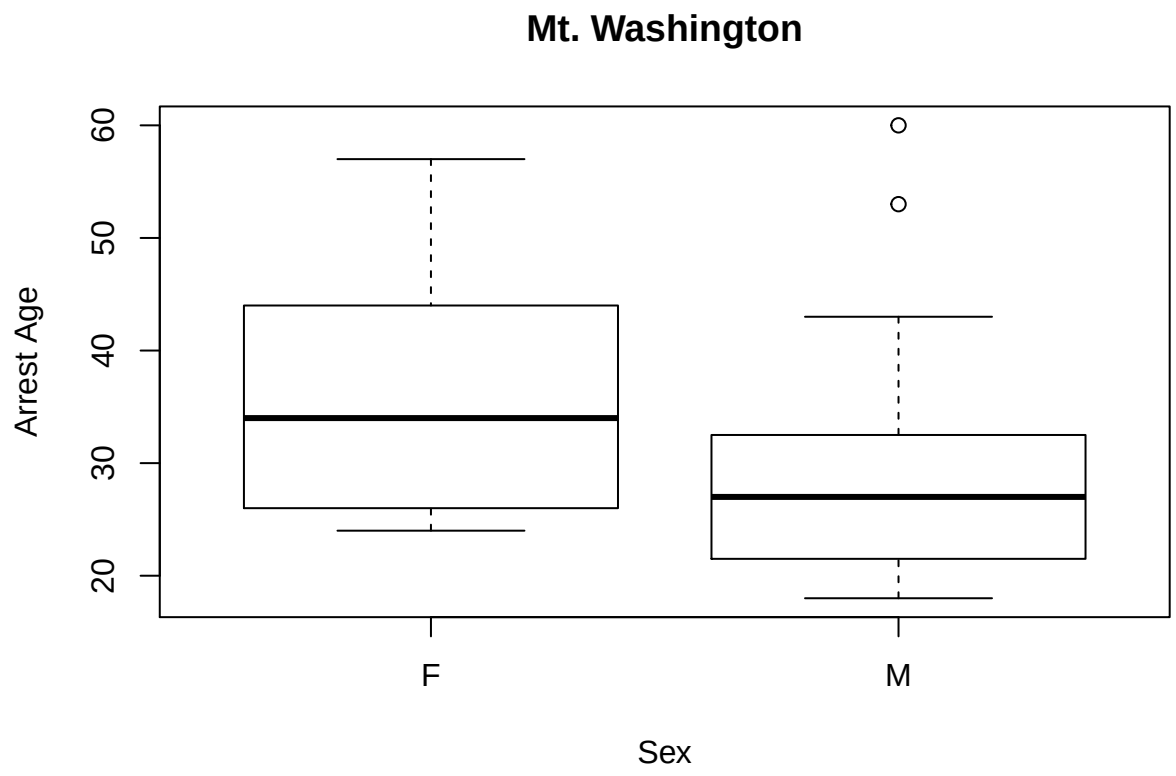

```
# can we do the same stratified by sex?
boxplot(arrest_tab$age[mount_washington_index]~arrest_tab$sex[mount_washington_index])
```



This used a very useful notation in R: the tilde, ~ which we will encounter in a few different places. One way of thinking about that abstractly is, do something with this attribute, as a function (or depending on, stratified by, conditioned on) this other attribute. For instance, "plot `age` as a function of sex" in our example.

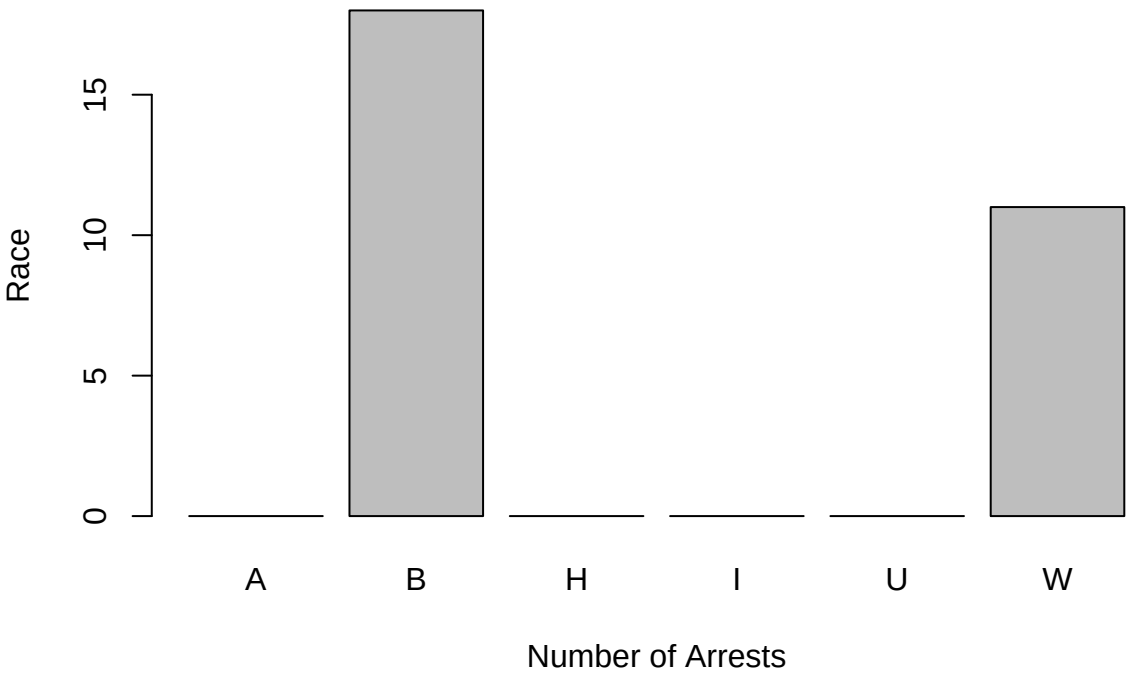
Let's write code that's a little cleaner for that last plot, and let's also make the plot a bit more useful by adding a title and axis labels:

```
mount_washington_tab <- arrest_tab[mount_washington_index,]
boxplot(mount_washington_tab$age~mount_washington_tab$sex,
        main="Mt. Washington",
        xlab="Sex", ylab="Arrest Age")
```



Here's one more useful plot:

```
barplot(table(mount_washington_tab$race),
        xlab="Number of Arrests",
        ylab="Race")
```



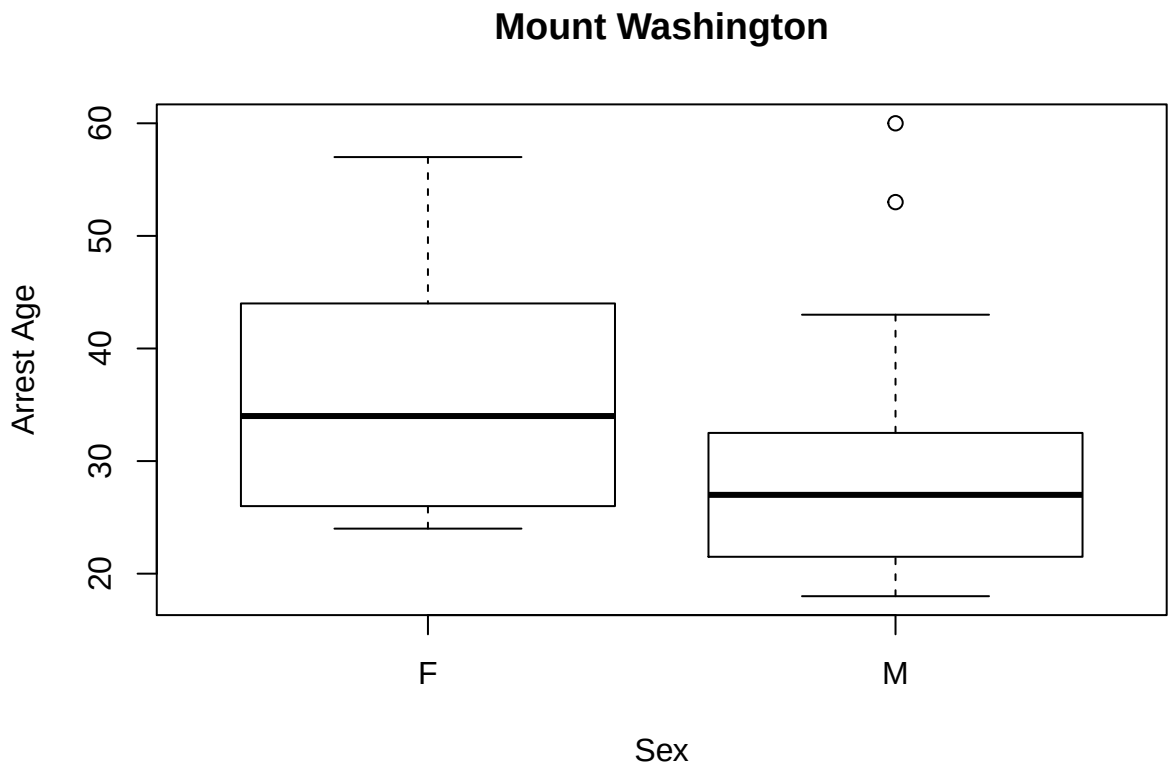
5.9 Functions

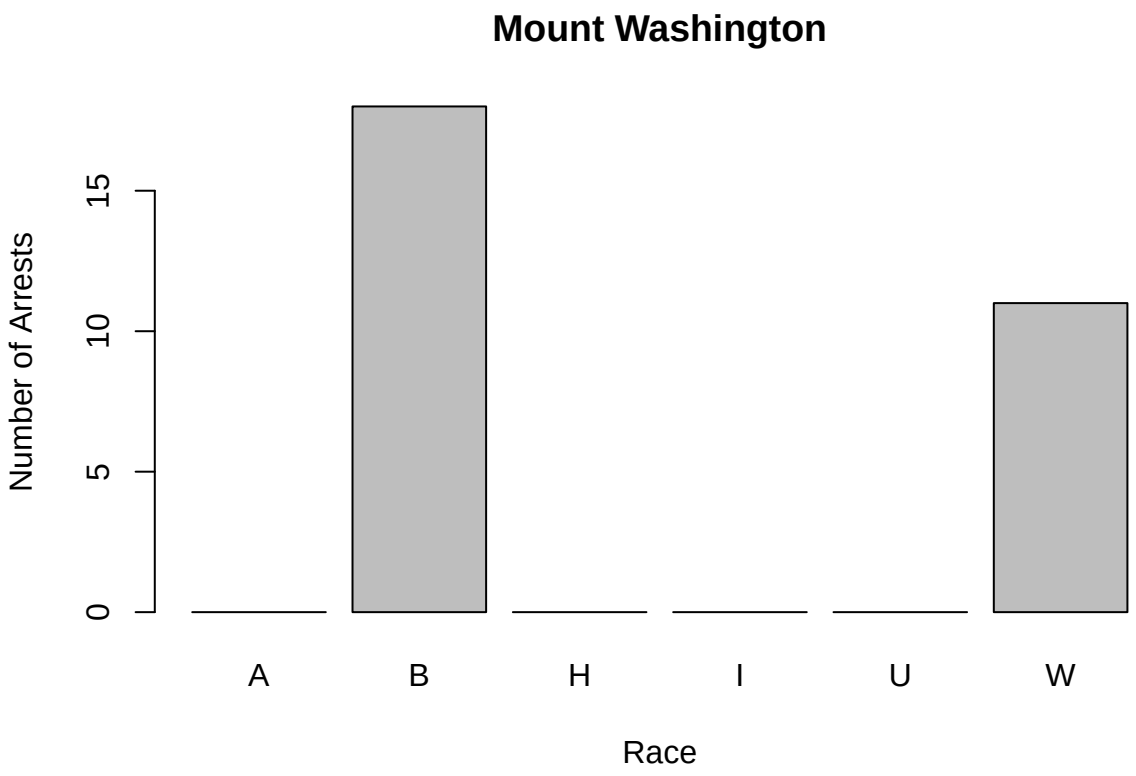
Now suppose we wanted to do a similar analysis for other neighborhoods. In that case we should encapsulate the summaries and plots we want to do in a function:

```
analyze_neighborhood <- function(neighborhood) {
  neighborhood_index <- arrest_tab$neighborhood == neighborhood
  neighborhood_tab <- arrest_tab[neighborhood_index,]

    boxplot(neighborhood_tab$age~neighborhood_tab$sex,
            main = neighborhood,
            xlab = "Sex", ylab="Arrest Age")

    barplot(table(neighborhood_tab$race),
            main = neighborhood,
            xlab = "Race", ylab="Number of Arrests")
}
```

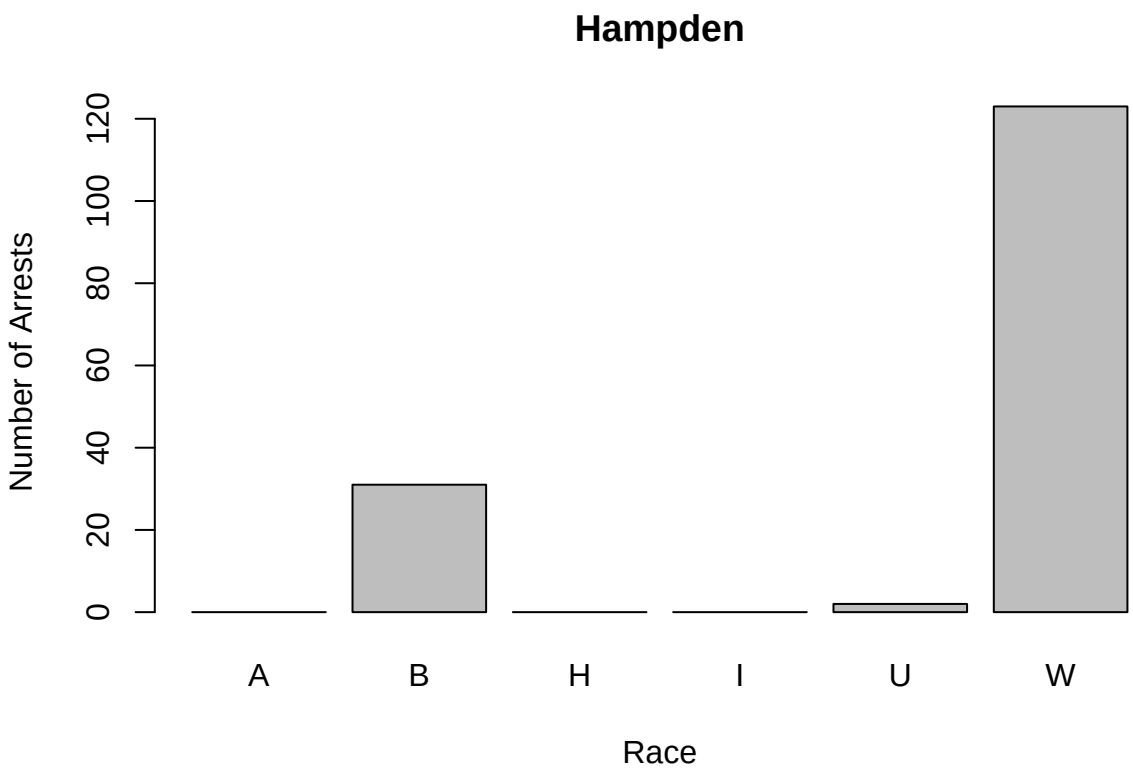
Now we can use that function to make our plots for specific neighborhoods

```
analyze_neighborhood("Mount Washington")
```






```
analyze_neighborhood("Hampden")
```



5.10 A note on data types

This dataset contains data of types commonly found in data analyses

- Numeric (continuous): A numeric measurement (e.g., height)
- Numeric (discrete): Usually obtained from counting, think only integers (e.g., `age` which is measured in years)
- Categorical: One of a possible set of values (e.g., `sex`)
- Datetime: Date and time of some event or observation (e.g., `arrestDate`, `arrestTime`)
- geolocation: Latitude and Longitude of some event or observation (e.g., `Location.`)

The distinction between continuous and discrete is a bit tricky since measurements that have finite precision must be discrete. So, the difference really comes up when we build statistical models of datasets for analysis. For now, think of discrete data as the result of counting, and continuous data the result of some physical measurement.

We said that R is designed for data analysis. My favorite example of how that manifests itself is the `factor` datatype. If you look at your dataset now, `arrest_tab$sex` is a vector of strings:

```
class(arrest_tab$sex)
```

```
## [1] "factor"
```

```
summary(arrest_tab$sex)
```

```
##     F     M  NA's
## 19431 85095     2
```

However, as a measurement, or attribute, it should only take one of two values (or three depending on how you record missing, unknown or unspecified). So, in R, that categorical data type is called a *factor*. Notice what the `summary` function does after turning the `sex` attribute into a *factor*:

```
arrest_tab$sex <- factor(arrest_tab$sex)
summary(arrest_tab$sex)
```

```
##     F     M  NA's
## 19431 85095     2
```

This distinction shows up in many other places where functions have very different behavior when called on a vector of strings and when called on a factor (e.g., functions that make plots, or functions that learn statistical models).

One last note, the possible values a *factor* can take are called *levels*:

```
levels(arrest_tab$sex)
```

```
## [1] "F" "M"
```

Exercise: you should transform the `race` attribute into a factor as well. How many levels does it have?

5.11 Thinking in vectors

In data analysis the *vector* is probably the most fundamental data type (other than basic numbers, strings, etc.). Why? Consider getting data about one attribute, say height, for a group of people. What do you get, an array of numbers, all in the same unit (say feet, inches or centimeters). How about their name? Then you get an array of strings. Abstractly, we think of vectors as arrays of values, all of the same *class* or datatype.

In our dataset, each column, corresponding to an attribute, is a vector:

```
# the 'str' function gives a bit more low-level information about objects
str(arrest_tab$Location)
```

```
## Warning: Unknown or uninitialised column: 'Location'.
```

```
##  NULL
```

R (and other data analysis languages) are designed to operate on vectors easily. For example, frequently we want to do some kind of transformation to a data attribute, say record age in months rather than years. Then we would perform the **same operation** for every value in the corresponding vector:

```
age_in_months <- arrest_tab$age * 12
```

In a language that doesn't support this type of vectorized operation, you would use a loop, or similar construct, to perform this operation.

Another type of transformation frequently done is to combine attributes into a single attribute. Suppose we wanted to combine the `arrestLocation` and `neighborhood` attributes into an `address` attribute:

```
# remember you can always find out what a function does by using ?paste
head(paste(arrest_tab$arrestLocation, arrest_tab$neighborhood, sep=", "))
```

```
## [1] "NA, NA"
## [2] "2000 Wilkens Ave, Carrollton Ridge"
## [3] "2800 Mayfield Ave, Belair-Edison"
## [4] "2100 Ashburton St, Panway/Braddish Avenue"
## [5] "4000 Wilsby Ave, Pen Lucy"
## [6] "2900 Spellman Rd, Cherry Hill"
```

Here the `paste` function concatenates strings element-wise: the first string in `arrestLocation` is concatenated with the first string in `neighborhood`, etc.

Arithmetic operations have the same element-wise operation:

```
# add first 10 odd numbers to first 10 even numbers
seq(1, 20, by=2) + seq(2, 20, by=2)
```

```
##  [1]  3  7 11 15 19 23 27 31 35 39
```

5.12 Lists vs. vectors

We saw that vectors are arrays of values, all of the same *class*. R also allows arrays of values that have different *class* or datatype. These are called *lists*. Here is a list containing a string, and a couple of numbers:

```
my_list <- list("Hector", 40, 71)
my_list
```

```
## [[1]]
## [1] "Hector"
##
## [[2]]
## [1] 40
##
## [[3]]
## [1] 71
```

Indexing in lists uses different syntax from the indexing we saw before. To index an element in a list we would use a double-bracket `[[`.

```
my_list[[1]]
```

```
## [1] "Hector"
```

In contrast, the single bracket `[` indexes a *part* of the list, and thus returns another list.

```
my_list[1]
```

```
## [[1]]
## [1] "Hector"
```

That way we can use slice notation and other operations we saw when indexing vectors as before, but we get lists as results.

```
my_list[1:2]
```

```
## [[1]]
## [1] "Hector"
##
## [[2]]
## [1] 40
```

List elements can have names as well:

```
named_list <- list(person="Hector", age=40, height=71)
named_list
```

```
## $person
## [1] "Hector"
##
## $age
## [1] 40
##
## $height
## [1] 71
```

Which we can use to index elements as well (both with `[[` and `$`)

```
named_list[["person"]]
```

```
## [1] "Hector"
```

```
named_list$person
```

```
## [1] "Hector"
```

Lists can hold arbitrary objects as elements. For example you can have a vector of strings as an element in a list

```
my_list <- list(person=c("Hector", "Ringo", "Paul", "John"), 40, 71)
my_list
```

```
## $person
## [1] "Hector" "Ringo"  "Paul"   "John"
##
## [[2]]
## [1] 40
##
## [[3]]
## [1] 71
```

Now, we come to a momentous occassion in understanding R. `data.frame`s are special instances of *lists*! But, in this case, every element in the list is a vector, and all vectors have exactly the same length. So `arrest_tab$age` indexes the named element `age` in the list `arrest_tab`!

The pattern of *applying* functions to entries in vectors also holds for elements in lists. So, if we want to calculate smallest value for every attribute in our dataset, we could do something like this:

```
sapply(arrest_tab, function(v) sort(v)[1])
```

```
##                                          arrest
##                                       "11126858"
##                                             age
##                                             "0"
##                                            race
##                                             "1"
##                                             sex
##                                             "1"
##                                       arrestDate
##                                      "01/01/2011"
##                                       arrestTime
##                                             "0"
##                                   arrestLocation
##                                      "0 20Th St"
##                                  incidentOffense
##                                             "1"
##                                 incidentLocation
##                          "*** District Detail ***"
##                                           charge
##                                        "1 0002"
##                                 chargeDescription
## "Abduct Child Under 12 || Abduct Child Under 12"
##                                         district
##                                        "CENTRAL"
##                                             post
##                                            "111"
##                                    neighborhood
##                                          "Abell"
##                                       Location 1
##                "(39.2000327685, -76.5555163146)"
```

5.13 Making the process explicit with pipes

We've discussed the idea of thinking about data analysis work in terms of "pipelines", where we start from data of a certain shape (e.g., a `data.frame`) and apply transformations (functions) to obtain data that contains the computation we want. Consider the following example seen in class:

What is the mean age of males arrested in the SOUTHERN district?

We can frame the answer to this question as a series of data transformations to get the answer we are looking for:

```
# filter data to observations we need
index_vector <- arrest_tab$sex == "M" & arrest_tab$district == "SOUTHERN"
tmp <- arrest_tab[index_vector,]

# select the attribute/column we need
tmp <- tmp[["age"]]

# compute statistic required
mean(tmp, na.rm=TRUE)
```

```
## [1] 32.28481
```

Let's rewrite this using functions to illustrate the point

```
filter_data <- function(data) {
  index_vector <- data$sex == "M" & data$district == "SOUTHERN"
  data[index_vector,]
}

select_column <- function(data, column) {
  data[[column]]
}

tmp <- filter_data(arrest_tab)
tmp <- select_column(tmp, "age")
mean(tmp, na.rm=TRUE)
```

```
## [1] 32.28481
```

So, this pattern of *data->transform->data* becomes clearer when written that way.

The `dplyr` package introduces *syntactic sugar* to make this explicit. We can write the above snippet using the "pipe" operator `%>%`:

```
arrest_tab %>%
  filter_data() %>%
  select_column("age") %>%
  mean(na.rm=TRUE)
```

```
## [1] 32.28481
```

The `%>%` binary operator takes the value to its **left** and inserts it as the first argument of the function call to its **right**. So the expression `LHS %>% f(another_argument)` is **equivalent** to the expression `f(LHS, another_argument)`. We will see this pattern extensively in class because it explicitly presents the way we want to organize many of our data analysis tasks.

Chapter 6

Ingesting data

Now that we have a better understanding of the R data analysis language, we turn to the first significant challenge in data analysis, getting data into R in a shape that we can use to start our analysis. We will look at two types of data ingestion: *structured ingestion*, where we read data that is already structured, like a comma separated value (CSV) file, and *scraping* where we obtain data from text, usually in websites.

6.1 Structured ingestion

6.1.1 CSV files (and similar)

We saw in a previous chapter how we can use the `read_csv` file to read data from a CSV file into a data frame. Comma separated value (CSV) files are structured in a somewhat regular way, so reading into a data frame is straightforward. Each line in the CSV file corresponds to an observation (a row in a data frame). Each line contains values separated by a comma (`,`), corresponding to the variables of each observation.

This ideal principle of how a CSV file is constructed is frequently violated by data contained in CSV files. To get a sense of how to deal with these cases look at the documentation of the `read_csv` function. For instance:

- the first line of the file may or may not contain the names of variables for the data frame (`col_names` argument).
- strings are quoted using `'` instead of `"` (`quote` argument)
- missing data is encoded with a non-standard code, e.g., `-` (`na` argument)
- values are separated by a character other than `,` (`read_delim` function)
- file may contain header information before the actual data so we have to skip some lines when loading the data (`skip` argument)

You should read the documentation of the `read_csv` function to appreciate the complexities it can maneuver when reading data from structured text files.

```
?read_csv
```

6.1.2 Excel spreadsheets

Often you will need to ingest data that is stored in an Excel spreadsheet. The `readxl` package is used to do this. The main function for this package is the `read_excel` function. It contains similar arguments to the `read_csv` function we saw above.

On your own: Use the `read_excel` function to parse migration data from the 2009 INEGI national survey contained in file `data/Migracion_interna_eua.xls`.

6.2 Scraping

Often, data we want to use is hosted as part of HTML files in webpages. The markup structure of HTML allows to parse data into tables we can use for analysis. Let's use the Rotten Tomatoes ratings webpage for Diego Luna as an example:

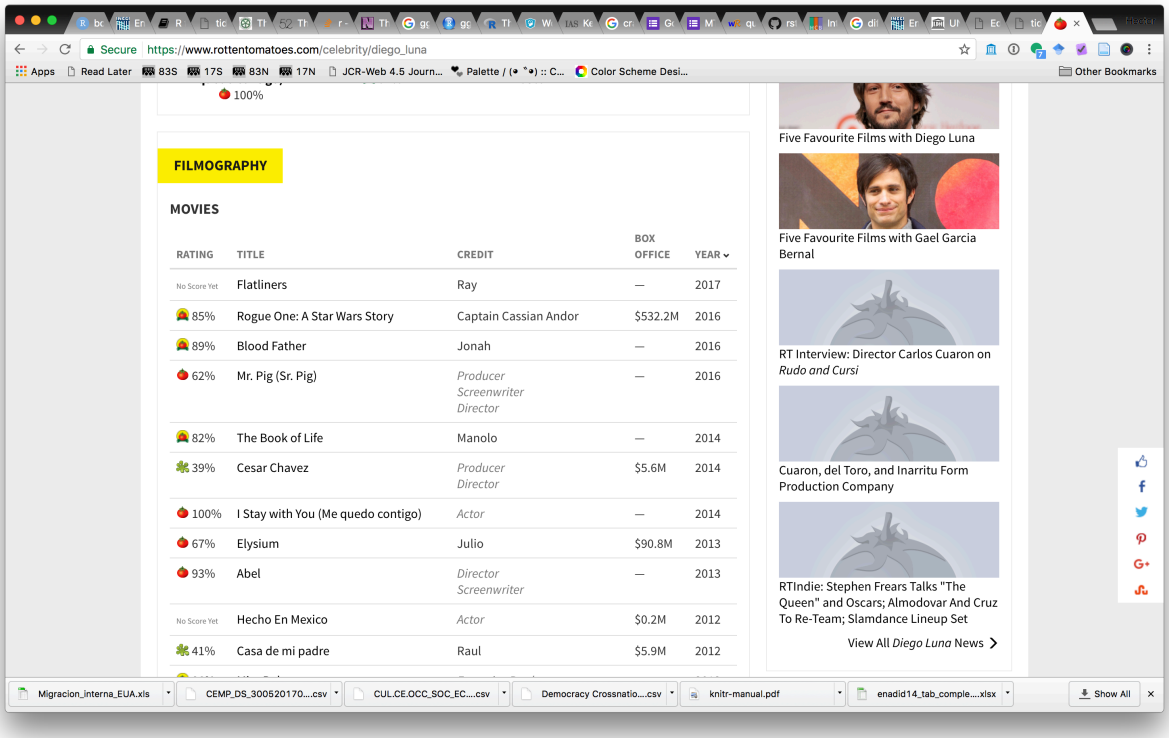


Figure 6.1:

We can scrape ratings for his movies from this page. To do this we need to figure out how the HTML page's markup can help us write R expressions to find this data in the page. Most web browsers have facilities to show page markup. In Google Chrome, you can use `View>Developer>Developer Tools`, and inspect the page markdown to find where the data is contained. In this example, we see that the data we want is in a `<table>` element in the page, with id `filmographyTbl`.

Now that we have that information, we can use the `rvest` package to scrape this data:

```
library(rvest)
```

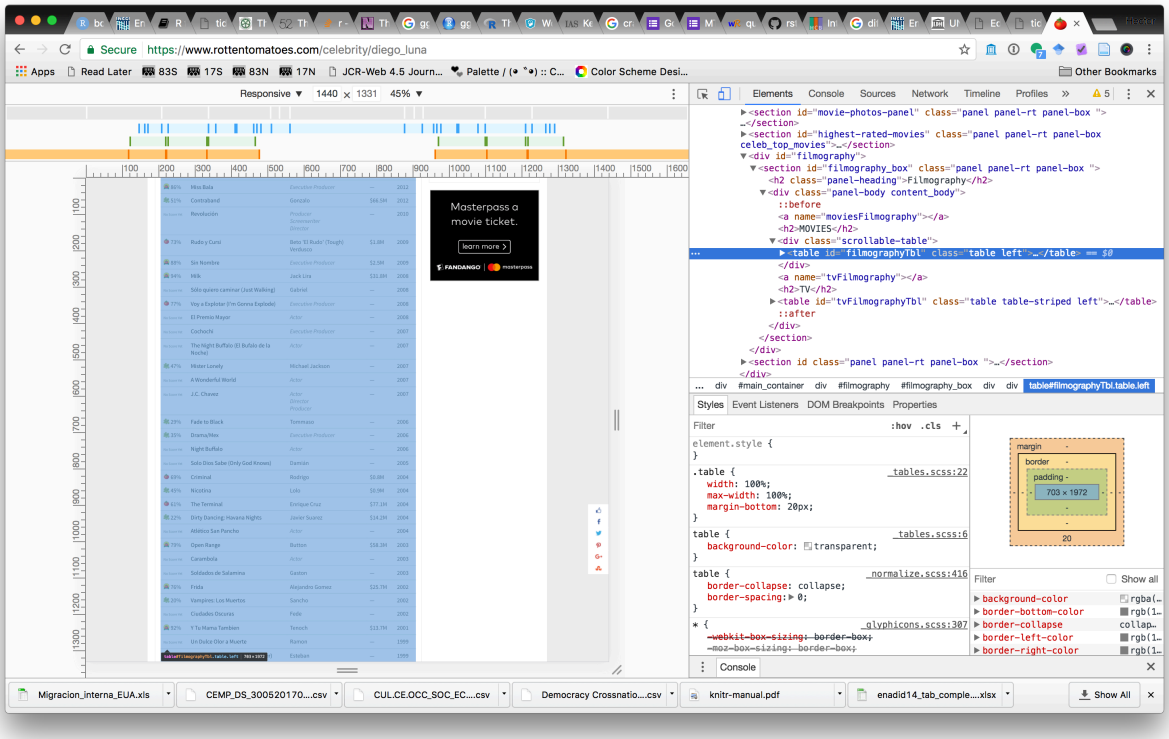


Figure 6.2:

```
url <- "https://www.rottentomatoes.com/celebrity/diego_luna"

dl_tab <- url %>%
  read_html() %>%
  html_node("#filmographyTbl") %>%
  html_table()

head(dl_tab)
```

```
##         RATING                        TITLE
## 1 No Score Yet                   Flatliners
## 2          85% Rogue One: A Star Wars Story
## 3          89%                 Blood Father
## 4          62%           Mr. Pig (Sr. Pig)
## 5          82%            The Book of Life
## 6          39%                 Cesar Chavez
##
## 1
## 2
## 3
## 4 Producer\n                                \n                    \n
## 5
## 6                                                                    Pr
##   BOX OFFICE YEAR
## 1          - 2017
```

```
## 2    $532.2M 2016
## 3          - 2016
## 4          - 2016
## 5          - 2014
## 6      $5.6M 2014
```

The main two functions we used here are `html_node` and `html_table`. `html_node` finds elements in the HTML page according to some selection criteria. Since we want the element with `id=filmographyTbl` we use the `#` selection operation since that corresponds to selection by id. Once the desired element in the page is selected, we can use the `html_table` function to parse the element's text into a data frame.

On your own: If you wanted to extract the TV filmography from the page, how would you change this call?

On your own: We can get movie budget and gross information from this page: http://www.the-numbers.com/movie/budgets/all. Write R code to scrape the budget data from that page.

Chapter 7

Tidying data

This section is concerned with common problems in data preparation, namely use cases commonly found in raw datasets that need to be addressed to turn messy data into tidy data. These would be operations that you would perform on data obtained as a csv file from a collaborator or data repository, or as the result of scraping data from webpages or other sources. We derive many of our ideas from the paper Tidy Data by Hadley Wickham. Associated with that paper we will use two very powerful R libraries `tidyr` and `dplyr` which are extremely useful in writing scripts for data cleaning, preparation and summarization. A basic design principle behind these libraries is trying to effectively and efficiently capture very common use cases and operations performed in data cleaning. The paper frames these use cases and operations which are them implemented in software.

7.1 Tidy Data

Here we assume we are working with a data model based on rectangular data structures where

1. Each attribute (or variable) forms a column
2. Each entity (or observation) forms a row
3. Each type of entity (observational unit) forms a table

Here is an example of a tidy dataset:

```
library(nycflights13)
head(flights)
```

```
## # A tibble: 6 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>
## 1  2013     1     1      517            515         2      830
## 2  2013     1     1      533            529         4      850
## 3  2013     1     1      542            540         2      923
## 4  2013     1     1      544            545        -1     1004
## 5  2013     1     1      554            600        -6      812
## 6  2013     1     1      554            558        -4      740
## # ... with 12 more variables: sched_arr_time <int>, arr_delay <dbl>,
```

```
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
## #   time_hour <dttm>
```

it has one observation per row, a single variable per column. Notice only information about flights are included here (e.g., no airport information other than the name) in these observations.

7.2 Common problems in messy data

The set of common operations we will study are based on these common problems found in datasets. We will see each one in detail:

- Column headers are values, not variable names (gather)
- Multiple variables stored in one column (split)
- Variables stored in both rows and column (rotate)
- Multiple types of observational units are stored in the same table (normalize)
- Single observational unit stored in multiple tables (join)

We are using data from Hadley's paper found in github. It's included directory `data`:

```
data_dir <- "data"
```

7.2.1 Headers as values

The first problem we'll see is the case where a table header contains values. At this point we will introduce the `dplyr` package, which we'll use extensively in this course. It is an extremely powerful and efficient way of manipulating tidy data. It will serve as the core of our data manipulation knowledge after this course.

`dplyr` defines a slightly different way of using data.frames. The `tbl_df` function converts a standard R data.frame into a `tbl_df` defined by `dplyr`. One nice thing it does, for example, is print tables in a much friendlier way.

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

pew <- read_csv(file.path(data_dir, "pew.csv"))
```

```
## Parsed with column specification:
## cols(
##   religion = col_character(),
##   `<$10k` = col_integer(),
##   `$10-20k` = col_integer(),
##   `$20-30k` = col_integer(),
##   `$30-40k` = col_integer(),
##   `$40-50k` = col_integer(),
```

```
##   `$50-75k` = col_integer(),
##   `$75-100k` = col_integer(),
##   `$100-150k` = col_integer(),
##   `>150k` = col_integer(),
##   `Don't know/refused` = col_integer()
## )
```

```
pew
```

```
## # A tibble: 18 x 11
##                   religion `<$10k` `$10-20k` `$20-30k`
##                      <chr>   <int>     <int>     <int>
##  1                Agnostic      27        34        60
##  2                 Atheist      12        27        37
##  3                Buddhist      27        21        30
##  4                Catholic     418       617       732
##  5      Don't know/refused      15        14        15
##  6        Evangelical Prot     575       869      1064
##  7                   Hindu       1         9         7
##  8 Historically Black Prot     228       244       236
##  9       Jehovah's Witness      20        27        24
## 10                  Jewish      19        19        25
## 11           Mainline Prot     289       495       619
## 12                  Mormon      29        40        48
## 13                  Muslim       6         7         9
## 14                Orthodox      13        17        23
## 15         Other Christian       9         7        11
## 16            Other Faiths      20        33        40
## 17   Other World Religions       5         2         3
## 18            Unaffiliated     217       299       374
## # ... with 7 more variables: `$30-40k` <int>, `$40-50k` <int>,
## #   `$50-75k` <int>, `$75-100k` <int>, `$100-150k` <int>, `>150k` <int>,
## #   `Don't know/refused` <int>
```

This table has the number of survey respondents of a specific religion that report their income within some range. A tidy version of this table would consider the *variables* of each observation to be `religion, income, frequency` where `frequency` has the number of respondents for each religion and income range. The function to use in the `tidyr` package is `gather`:

```
tidy_pew <- gather(pew, income, frequency, -religion)
tidy_pew
```

```
## # A tibble: 180 x 3
##                   religion income frequency
##                      <chr>  <chr>     <int>
##  1                Agnostic  <$10k        27
##  2                 Atheist  <$10k        12
##  3                Buddhist  <$10k        27
##  4                Catholic  <$10k       418
##  5      Don't know/refused  <$10k        15
##  6        Evangelical Prot  <$10k       575
##  7                   Hindu  <$10k         1
##  8 Historically Black Prot  <$10k       228
```

```
##  9       Jehovah's Witness  <$10k        20
## 10                  Jewish  <$10k        19
## # ... with 170 more rows
```

This says: gather all the columns from the `pew` (except `religion`) into key-value columns `income` and `frequency`. This table is much easier to use in other analyses.

Another example: this table has a row for each song appearing in the billboard top 100. It contains track information, and the date it entered the top 100. It then shows the rank in each of the next 76 weeks.

```
billboard <- read_csv(file.path(data_dir, "billboard.csv"))
```

```
## Parsed with column specification:
## cols(
##   .default = col_integer(),
##   artist = col_character(),
##   track = col_character(),
##   time = col_time(format = ""),
##   date.entered = col_date(format = ""),
##   wk66 = col_character(),
##   wk67 = col_character(),
##   wk68 = col_character(),
##   wk69 = col_character(),
##   wk70 = col_character(),
##   wk71 = col_character(),
##   wk72 = col_character(),
##   wk73 = col_character(),
##   wk74 = col_character(),
##   wk75 = col_character(),
##   wk76 = col_character()
## )
```

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

```
billboard
```

```
## # A tibble: 317 x 81
##     year         artist                  track     time date.entered
##    <int>          <chr>                  <chr>   <time>       <date>
##  1  2000          2 Pac Baby Don't Cry (Keep... 04:22:00   2000-02-26
##  2  2000        2Ge+her The Hardest Part Of ... 03:15:00   2000-09-02
##  3  2000   3 Doors Down              Kryptonite 03:53:00   2000-04-08
##  4  2000   3 Doors Down                   Loser 04:24:00   2000-10-21
##  5  2000       504 Boyz           Wobble Wobble 03:35:00   2000-04-15
##  6  2000           98^0 Give Me Just One Nig... 03:24:00   2000-08-19
##  7  2000        A*Teens           Dancing Queen 03:44:00   2000-07-08
##  8  2000        Aaliyah           I Don't Wanna 04:15:00   2000-01-29
##  9  2000        Aaliyah               Try Again 04:03:00   2000-03-18
## 10  2000 Adams, Yolanda           Open My Heart 05:30:00   2000-08-26
## # ... with 307 more rows, and 76 more variables: wk1 <int>, wk2 <int>,
## #   wk3 <int>, wk4 <int>, wk5 <int>, wk6 <int>, wk7 <int>, wk8 <int>,
## #   wk9 <int>, wk10 <int>, wk11 <int>, wk12 <int>, wk13 <int>, wk14 <int>,
## #   wk15 <int>, wk16 <int>, wk17 <int>, wk18 <int>, wk19 <int>,
```

```
## #   wk20 <int>, wk21 <int>, wk22 <int>, wk23 <int>, wk24 <int>,
## #   wk25 <int>, wk26 <int>, wk27 <int>, wk28 <int>, wk29 <int>,
## #   wk30 <int>, wk31 <int>, wk32 <int>, wk33 <int>, wk34 <int>,
## #   wk35 <int>, wk36 <int>, wk37 <int>, wk38 <int>, wk39 <int>,
## #   wk40 <int>, wk41 <int>, wk42 <int>, wk43 <int>, wk44 <int>,
## #   wk45 <int>, wk46 <int>, wk47 <int>, wk48 <int>, wk49 <int>,
## #   wk50 <int>, wk51 <int>, wk52 <int>, wk53 <int>, wk54 <int>,
## #   wk55 <int>, wk56 <int>, wk57 <int>, wk58 <int>, wk59 <int>,
## #   wk60 <int>, wk61 <int>, wk62 <int>, wk63 <int>, wk64 <int>,
## #   wk65 <int>, wk66 <chr>, wk67 <chr>, wk68 <chr>, wk69 <chr>,
## #   wk70 <chr>, wk71 <chr>, wk72 <chr>, wk73 <chr>, wk74 <chr>,
## #   wk75 <chr>, wk76 <chr>
```

Challenge: This dataset has values as column names. Which column names are values? How do we tidy this dataset?

7.2.2 Multiple variables in one column

The next problem we'll see is the case when we see multiple variables in a single column. Consider the following dataset of tuberculosis cases:

```
tb <- read_csv(file.path(data_dir, "tb.csv"))
```

```
## Parsed with column specification:
## cols(
##   .default = col_integer(),
##   iso2 = col_character()
## )
```

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

```
tb
```

```
## # A tibble: 5,769 x 22
##    iso2  year   m04  m514  m014 m1524 m2534 m3544 m4554 m5564   m65    mu
##   <chr> <int> <int> <int> <int> <int> <int> <int> <int> <int> <int> <int>
##  1    AD  1989    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
##  2    AD  1990    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
##  3    AD  1991    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
##  4    AD  1992    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
##  5    AD  1993    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
##  6    AD  1994    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
##  7    AD  1996    NA    NA     0     0     0     4     1     0     0    NA
##  8    AD  1997    NA    NA     0     0     1     2     2     1     6    NA
##  9    AD  1998    NA    NA     0     0     0     1     0     0     0    NA
## 10    AD  1999    NA    NA     0     0     0     1     1     0     0    NA
## # ... with 5,759 more rows, and 10 more variables: f04 <int>, f514 <int>,
## #   f014 <int>, f1524 <int>, f2534 <int>, f3544 <int>, f4554 <int>,
## #   f5564 <int>, f65 <int>, fu <int>
```

This table has a row for each year and strain of tuberculosis (given by the first two columns). The remaining columns state the number of cases for a given demographic. For example, `m1524` corresponds to males

between 15 and 24 years old, and `f1524` are females age 15-24. As you can see each of these columns has two variables: `sex` and `age`.

Challenge: what else is untidy about this dataset?

So, we have to do two operations to tidy this table, first we need to use `gather` the tabulation columns into a `demo` and `n` columns (for demographic and number of cases):

```
tidy_tb <- gather(tb, demo, n, -iso2, -year)
tidy_tb
```

```
## # A tibble: 115,380 x 4
##     iso2  year  demo     n
##    <chr> <int> <chr> <int>
##  1    AD  1989   m04    NA
##  2    AD  1990   m04    NA
##  3    AD  1991   m04    NA
##  4    AD  1992   m04    NA
##  5    AD  1993   m04    NA
##  6    AD  1994   m04    NA
##  7    AD  1996   m04    NA
##  8    AD  1997   m04    NA
##  9    AD  1998   m04    NA
## 10    AD  1999   m04    NA
## # ... with 115,370 more rows
```

Next, we need to `separate` the values in the `demo` column into two variables `sex` and `age`

```
tidy_tb <- separate(tidy_tb, demo, c("sex", "age"), sep=1)
tidy_tb
```

```
## # A tibble: 115,380 x 5
##     iso2  year   sex   age     n
##  * <chr> <int> <chr> <chr> <int>
##  1    AD  1989     m    04    NA
##  2    AD  1990     m    04    NA
##  3    AD  1991     m    04    NA
##  4    AD  1992     m    04    NA
##  5    AD  1993     m    04    NA
##  6    AD  1994     m    04    NA
##  7    AD  1996     m    04    NA
##  8    AD  1997     m    04    NA
##  9    AD  1998     m    04    NA
## 10    AD  1999     m    04    NA
## # ... with 115,370 more rows
```

This calls the `separate` function on table `tidy_db`, separating the `demo` variable into variables `sex` and `age` by separating each value after the first character (that's the `sep` argument).

We can put these two commands together in a pipeline:

```
tidy_tb <- tb %>%
  gather(demo, n, -iso2, -year)  %>%
  separate(demo, c("sex", "age"), sep=1)
tidy_tb
```

```
## # A tibble: 115,380 x 5
##    iso2  year   sex   age     n
##  * <chr> <int> <chr> <chr> <int>
##  1    AD  1989     m    04    NA
##  2    AD  1990     m    04    NA
##  3    AD  1991     m    04    NA
##  4    AD  1992     m    04    NA
##  5    AD  1993     m    04    NA
##  6    AD  1994     m    04    NA
##  7    AD  1996     m    04    NA
##  8    AD  1997     m    04    NA
##  9    AD  1998     m    04    NA
## 10    AD  1999     m    04    NA
## # ... with 115,370 more rows
```

7.2.3 Variables stored in both rows and columns

This is the messiest, commonly found type of data. Let's take a look at an example, this is daily weather data from for one weather station in Mexico in 2010.

```
weather <- read_csv(file.path(data_dir, "weather.csv"))
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   id = col_character(),
##   year = col_integer(),
##   month = col_integer(),
##   element = col_character(),
##   d9 = col_character(),
##   d12 = col_character(),
##   d18 = col_character(),
##   d19 = col_character(),
##   d20 = col_character(),
##   d21 = col_character(),
##   d22 = col_character(),
##   d24 = col_character()
## )
```

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

```
weather
```

```
## # A tibble: 22 x 35
##         id  year month element    d1    d2    d3    d4    d5    d6    d7
##      <chr> <int> <int>   <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
##  1 MX17004  2010     1    tmax    NA    NA    NA    NA    NA    NA    NA
##  2 MX17004  2010     1    tmin    NA    NA    NA    NA    NA    NA    NA
##  3 MX17004  2010     2    tmax    NA  27.3  24.1    NA    NA    NA    NA
##  4 MX17004  2010     2    tmin    NA  14.4  14.4    NA    NA    NA    NA
##  5 MX17004  2010     3    tmax    NA    NA    NA    NA  32.1    NA    NA
##  6 MX17004  2010     3    tmin    NA    NA    NA    NA  14.2    NA    NA
```

```
##  7 MX17004  2010     4    tmax    NA    NA    NA    NA    NA    NA    NA
##  8 MX17004  2010     4    tmin    NA    NA    NA    NA    NA    NA    NA
##  9 MX17004  2010     5    tmax    NA    NA    NA    NA    NA    NA    NA
## 10 MX17004  2010     5    tmin    NA    NA    NA    NA    NA    NA    NA
## # ... with 12 more rows, and 24 more variables: d8 <dbl>, d9 <chr>,
## #   d10 <dbl>, d11 <dbl>, d12 <chr>, d13 <dbl>, d14 <dbl>, d15 <dbl>,
## #   d16 <dbl>, d17 <dbl>, d18 <chr>, d19 <chr>, d20 <chr>, d21 <chr>,
## #   d22 <chr>, d23 <dbl>, d24 <chr>, d25 <dbl>, d26 <dbl>, d27 <dbl>,
## #   d28 <dbl>, d29 <dbl>, d30 <dbl>, d31 <dbl>
```

So, we have two rows for each month, one with maximum daily temperature, one with minimum daily temperature, the columns starting with `d` correspond to the day in the where the measurements were made.

Challenge: How would a tidy version of this data look like?

```
weather %>%
  gather(day, value, d1:d31, na.rm=TRUE) %>%
  spread(element, value)
```

```
## # A tibble: 33 x 6
##         id  year month   day  tmax  tmin
##  *   <chr> <int> <int> <chr> <chr> <chr>
##  1 MX17004  2010     1   d30  27.8  14.5
##  2 MX17004  2010     2   d11  29.7  13.4
##  3 MX17004  2010     2    d2  27.3  14.4
##  4 MX17004  2010     2   d23  29.9  10.7
##  5 MX17004  2010     2    d3  24.1  14.4
##  6 MX17004  2010     3   d10  34.5  16.8
##  7 MX17004  2010     3   d16  31.1  17.6
##  8 MX17004  2010     3    d5  32.1  14.2
##  9 MX17004  2010     4   d27  36.3  16.7
## 10 MX17004  2010     5   d27  33.2  18.2
## # ... with 23 more rows
```

The new function we've used here is `spread`. It does the inverse of `gather` it spreads columns `element` and `value` into separate columns.

7.2.4 Multiple types in one table

Remember that an important aspect of tidy data is that it contains exactly one kind of observation in a single table. Let's see the billboard example again after the `gather` operation we did before:

```
tidy_billboard <- billboard %>%
  gather(week, rank, wk1:wk76, na.rm=TRUE)
tidy_billboard
```

```
## # A tibble: 5,307 x 7
##     year         artist                  track     time date.entered
##  * <int>          <chr>                  <chr>   <time>       <date>
##  1  2000          2 Pac Baby Don't Cry (Keep... 04:22:00   2000-02-26
##  2  2000        2Ge+her The Hardest Part Of ... 03:15:00   2000-09-02
##  3  2000   3 Doors Down              Kryptonite 03:53:00   2000-04-08
```

```
##  4  2000   3 Doors Down                    Loser 04:24:00   2000-10-21
##  5  2000       504 Boyz            Wobble Wobble 03:35:00   2000-04-15
##  6  2000           98^0 Give Me Just One Nig... 03:24:00   2000-08-19
##  7  2000        A*Teens            Dancing Queen 03:44:00   2000-07-08
##  8  2000        Aaliyah            I Don't Wanna 04:15:00   2000-01-29
##  9  2000        Aaliyah                Try Again 04:03:00   2000-03-18
## 10  2000 Adams, Yolanda            Open My Heart 05:30:00   2000-08-26
## # ... with 5,297 more rows, and 2 more variables: week <chr>, rank <chr>
```

Let's sort this table by track to see a problem with this table:

```
tidy_billboard <- tidy_billboard %>%
  arrange(track)
tidy_billboard
```

```
## # A tibble: 5,307 x 7
##     year artist                   track     time date.entered  week  rank
##    <int>  <chr>                    <chr>   <time>       <date> <chr> <chr>
##  1  2000  Nelly (Hot S**t) Country G... 04:17:00   2000-04-29   wk1   100
##  2  2000  Nelly (Hot S**t) Country G... 04:17:00   2000-04-29   wk2    99
##  3  2000  Nelly (Hot S**t) Country G... 04:17:00   2000-04-29   wk3    96
##  4  2000  Nelly (Hot S**t) Country G... 04:17:00   2000-04-29   wk4    76
##  5  2000  Nelly (Hot S**t) Country G... 04:17:00   2000-04-29   wk5    55
##  6  2000  Nelly (Hot S**t) Country G... 04:17:00   2000-04-29   wk6    37
##  7  2000  Nelly (Hot S**t) Country G... 04:17:00   2000-04-29   wk7    24
##  8  2000  Nelly (Hot S**t) Country G... 04:17:00   2000-04-29   wk8    24
##  9  2000  Nelly (Hot S**t) Country G... 04:17:00   2000-04-29   wk9    30
## 10  2000  Nelly (Hot S**t) Country G... 04:17:00   2000-04-29  wk10    36
## # ... with 5,297 more rows
```

We have a lot of repeated information in many of these rows (the artist, track name, year, title and date entered). The problem is that this table contains information about both tracks and rank in billboard. That's two different kinds of observations that should belong in two different tables in a tidy dataset.

Let's make a song table that only includes information about songs:

```
song <- tidy_billboard %>%
  select(artist, track, year, time, date.entered) %>%
  unique()
song
```

```
## # A tibble: 317 x 5
##           artist                   track  year     time date.entered
##            <chr>                   <chr> <int>   <time>       <date>
##  1         Nelly (Hot S**t) Country G...  2000 04:17:00   2000-04-29
##  2     Nu Flavor          3 Little Words  2000 03:54:00   2000-06-03
##  3  Jean, Wyclef                   911  2000 04:00:00   2000-10-07
##  4   Brock, Chad A Country Boy Can Su...  2000 03:54:00   2000-01-01
##  5  Clark, Terri       A Little Gasoline  2000 03:07:00   2000-12-16
##  6   Son By Four A Puro Dolor (Purest...  2000 03:30:00   2000-04-08
##  7 Carter, Aaron Aaron's Party (Come ...  2000 03:23:00   2000-08-26
##  8     Nine Days Absolutely (Story Of...  2000 03:09:00   2000-05-06
##  9    De La Soul               All Good?  2000 05:02:00   2000-12-23
```

```
## 10     Blink-182    All The Small Things  2000 02:52:00   1999-12-04
## # ... with 307 more rows
```

The `unique` function removes any duplicate rows in a table. That's how we have a single row for each song.

Next, we would like to remove all the song information from the rank table. But we need to do it in a way that still remembers which song each ranking observation corresponds to. To do that, let's first give each song an identifier that we can use to link songs and rankings. So, we can produce the final version of our song table like this:

```
song <- tidy_billboard %>%
  select(artist, track, year, time, date.entered) %>%
  unique() %>%
  mutate(song_id = row_number())
song
```

```
## # A tibble: 317 x 6
##            artist                    track  year     time date.entered
##             <chr>                    <chr> <int>   <time>       <date>
##  1          Nelly (Hot S**t) Country G...  2000 04:17:00   2000-04-29
##  2      Nu Flavor           3 Little Words  2000 03:54:00   2000-06-03
##  3   Jean, Wyclef                     911  2000 04:00:00   2000-10-07
##  4    Brock, Chad A Country Boy Can Su...  2000 03:54:00   2000-01-01
##  5   Clark, Terri        A Little Gasoline  2000 03:07:00   2000-12-16
##  6    Son By Four A Puro Dolor (Purest...  2000 03:30:00   2000-04-08
##  7 Carter, Aaron Aaron's Party (Come ...  2000 03:23:00   2000-08-26
##  8      Nine Days Absolutely (Story Of...  2000 03:09:00   2000-05-06
##  9     De La Soul               All Good?  2000 05:02:00   2000-12-23
## 10     Blink-182    All The Small Things  2000 02:52:00   1999-12-04
## # ... with 307 more rows, and 1 more variables: song_id <int>
```

The `mutate` function adds a new column to the table, in this case with column name `song_id` and value the row number the song appears in the table (from the `row_number` column).

Now we can make a rank table, we combine the tidy billboard table with our new song table using a `join` (we'll learn all about joins later). It checks the values on each row of the billboard table and looks for rows in the song table that have the exact same values, and makes a new row that combines the information from both tables.

```
tidy_billboard %>%
  left_join(song, c("artist", "year", "track", "time", "date.entered"))
```

```
## # A tibble: 5,307 x 8
##     year artist                    track     time date.entered  week  rank
##    <int>  <chr>                    <chr>   <time>       <date> <chr> <chr>
##  1  2000  Nelly (Hot S**t) Country G... 04:17:00   2000-04-29   wk1   100
##  2  2000  Nelly (Hot S**t) Country G... 04:17:00   2000-04-29   wk2    99
##  3  2000  Nelly (Hot S**t) Country G... 04:17:00   2000-04-29   wk3    96
##  4  2000  Nelly (Hot S**t) Country G... 04:17:00   2000-04-29   wk4    76
##  5  2000  Nelly (Hot S**t) Country G... 04:17:00   2000-04-29   wk5    55
##  6  2000  Nelly (Hot S**t) Country G... 04:17:00   2000-04-29   wk6    37
##  7  2000  Nelly (Hot S**t) Country G... 04:17:00   2000-04-29   wk7    24
##  8  2000  Nelly (Hot S**t) Country G... 04:17:00   2000-04-29   wk8    24
```

```
##  9  2000  Nelly (Hot S**t) Country G... 04:17:00   2000-04-29   wk9    30
## 10  2000  Nelly (Hot S**t) Country G... 04:17:00   2000-04-29  wk10    36
## # ... with 5,297 more rows, and 1 more variables: song_id <int>
```

That adds the `song_id` variable to the `tidy_billboard` table. So now we can remove the song information and only keep ranking information and the `song_id`.

```
rank <- tidy_billboard %>%
  left_join(song, c("artist", "year", "track", "time", "date.entered")) %>%
  select(song_id, week, rank)
rank
```

```
## # A tibble: 5,307 x 3
##    song_id  week  rank
##      <int> <chr> <chr>
##  1       1   wk1   100
##  2       1   wk2    99
##  3       1   wk3    96
##  4       1   wk4    76
##  5       1   wk5    55
##  6       1   wk6    37
##  7       1   wk7    24
##  8       1   wk8    24
##  9       1   wk9    30
## 10       1  wk10    36
## # ... with 5,297 more rows
```

Challenge: Let's do a little better job at tidying the billboard dataset:

1. When using `gather` to make the `week` and `rank` columns, remove any weeks where the song does not appear in the top 100. This is coded as missing (`NA`). See the `na.rm` argument to `gather`.

2. Make `week` a numeric variable (i.e., remove `wk`). See what the `extract_numeric` function does.

3. Instead of `date.entered` add a `date` column that states the actual date of each ranking. See how R deals with dates `?Date` and how you can turn a string into a `Date` using `as.Date`.

4. Sort the resulting table by date and rank.
5. Make new `song` and `rank` tables. `song` will now not have the `date.entered` column, and `rank` will have the new `date` column you have just created.

7.3 Data wrangling with `dplyr`

In previous lectures we discussed the `data.frame` to introduced the structure we usually see in a dataset before we start analysis:

1. Each attribute/variable forms a column
2. Each entity/(observational unit) forms a row
3. Each type of entity/(observation unit) forms a table

Although we did not explicitly mentioned number 3, in more complex datasets we want to make sure we divide different entity types into their respective table. We will discuss this in more detail when we see data models (in the database sense) later on. We will refer to data organized in this fashion as *tidy data*.

In this section we introduce operations and manipulations that commonly arise in analyses. We center our discussion around the idea that we are operating over tidy data, and we want to ensure that the operations we apply also generate tidy data as a result.

7.3.1 `dplyr`

We will use the `dplyr` package to introduce these oprations. I think it is one of the most beautiful tools created for data analysis. It clearly defines and efficiently implements most common data manipulation operations (verbs) one comes across in data analysis. It is built around tidy data principles. It also presents uniform treatment of multiple kinds of data sources (in memory files, partially loaded files, databases). It works best when used in conjuction with the non-standard *pipe* operator (`%>%`) first introduced by the `magrittr` package.

A complete introduction to `dplyr` is found here: http://cran.rstudio.com/web/packages/dplyr/vignettes/introduction.html

We will use a dataset of inbound and outbound flights to New York City as an example:

```
library(nycflights13)
data(flights)
```

7.4 Single-table manipulation

We will first look at operations that work over a single table at a time.

Single table verbs:

- `filter()` and `slice()`: subset observations (entities)
- `arrange()`: sort observations (entities)
- `select()` and `rename()`: subset variables (attributes)
- `distinct()`: make entities unique
- `mutate()` and `transmutate()`: add a new variable (attribute)
- `summarize()`: compute a summary statistics for one or more variables
- `sample_n()` and `sample_frac()`: sample observations from a data table

7.4.1 Subsetting Observations

The first fundamental operation we learned about early in this course is subsetting, or filtering, observations (entities, rows) in a dataset. Recall that we could subset by a set of indices (say, all even rows, this is used when splitting datasets to train and test statistical models). Much more useful is the ability to filter observations based on attribute values.

Figure 7.1:

```
# include only flights on United Airlines
flights %>% filter(carrier == "UA")

# select even samples, note function `n` defined by dplyr
flights %>% slice(seq(1, n(), by=2))
```

7.4.2 Subsetting Variables

Frequently, we may want to restrict a data analysis to a subset of variables (attributes, columns) to improve efficiency or interpretability.

Figure 7.2:

```
# select only month carrier and origin variables
flights %>% select(month, carrier, origin)
```

On large, complex, datasets the ability to perform this selection based on properties of column/attribute names is very powerful. For instance, in the `billboard` dataset we saw in a previous unit, we can select columns using partial string matching:

```
billboard %>%
  select(starts_with("wk"))
```

7.4.3 Creating New Variables

One of the most common operations in data analysis is to create new variables (attributes), based on other existing attributes.

These manipulations are used for transformations of existing single variables, for example, squaring a given varaible (`x -> x^2`), to make visualization or other downstream analysis more effective. In other cases, we may want to compute functions of existing variables to improve analysis or interpretation of a dataset.

Here is an example creating a new variable as a function of two existing variables

Figure 7.3:

```
# add new variable with total delay
flights %>% mutate(delay=dep_delay + arr_delay)
```

7.4.4 Summarizing Data

Much of statistical analysis, modeling and visualization is based on computing summaries (refered to as summary statistics) for variables (attributes), or other data features, of datasets. The `summarize` operation summarizes one variable (columns) over multiple observations (rowss) into a single value.

Figure 7.4:

```
# compute mean total delay across all flights
flights %>%
  mutate(delay = dep_delay + arr_delay) %>%
  summarize(mean_delay = mean(delay, na.rm=TRUE),
            min_delay = min(delay, na.rm=TRUE),
            max_delay = max(delay, na.rm=TRUE))
```

7.4.5 Grouping Data

Aggregation and summarization also go hand in hand with data grouping, where aggregates, or even variable transformations are performed *conditioned* on other variables. The notion of *conditioning* is fundamental and we will see it very frequently through the course. It is the basis of statistical analysis and Machine Learning models for regression and prediction, and it is essential in understanding the design of effective visualizations.

So the goal is to group observations (rows) with the same value of one or more variables (columns). In the `dplyr` implementation, the `group_by` function in essence annotates the rows of a data table as belonging to a specific group. When `summarize` is the applied onto this annotated data table, summaries are computed for each group, rather than the whole table.

```
# compute mean total delay per carrier
flights %>%
  mutate(delay = dep_delay + arr_delay) %>%
  group_by(carrier) %>%
  summarize(delay=mean(delay, na.rm=TRUE))
```

Figure 7.5:

7.5 Two-table manipulation

We saw above manipulations defined over single tables. In this section we look at efficient methods to combine data from multiple tables. The fundamental operation here is the `join`, which is a workhorse of database system design and impementation. The `join` operation combines rows from two tables to create a new single table, based on matching criteria specified over attributes of each of the two tables.

Consider the example of joining the `flights` and `airlines` table:

```
head(flights)
```

```
## # A tibble: 6 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>
## 1  2013     1     1      517            515         2      830
## 2  2013     1     1      533            529         4      850
## 3  2013     1     1      542            540         2      923
## 4  2013     1     1      544            545        -1     1004
## 5  2013     1     1      554            600        -6      812
## 6  2013     1     1      554            558        -4      740
## # ... with 12 more variables: sched_arr_time <int>, arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
## #   time_hour <dttm>
```

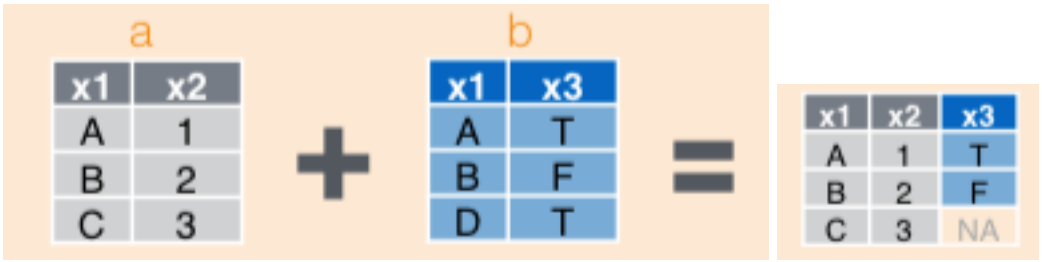
```
head(airlines)
```

```
## # A tibble: 6 x 2
##   carrier                    name
##     <chr>                   <chr>
## 1      9E        Endeavor Air Inc.
## 2      AA   American Airlines Inc.
## 3      AS     Alaska Airlines Inc.
## 4      B6           JetBlue Airways
## 5      DL     Delta Air Lines Inc.
## 6      EV ExpressJet Airlines Inc.
```

Here, we want to add airline information to each flight. We can do so by joining the attributes of the respective airline from the `airlines` table with the `flights` table based on the values of attributes `flights$carrier` and `airlines$carrier`. Specifically, every row of `flights` with a specific value for `flights$carrier`, is joined with the the corresponding row in `airlines` with the same value for `airlines$carrier`. We will see four different ways of performing this operation that differ on how non-matching observations are handled.

7.5.1 Left Join

In this case, all observations on left operand (LHS) are retained:

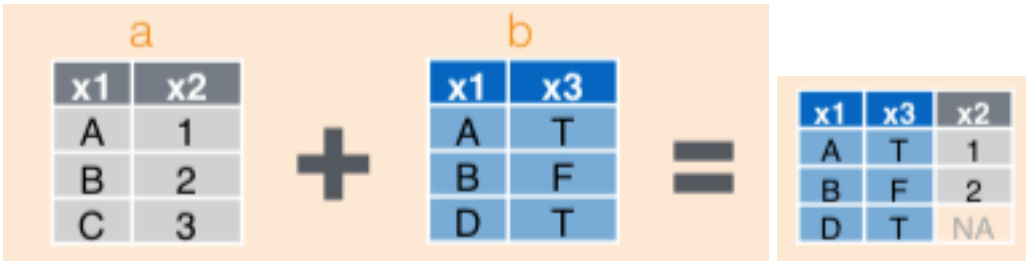



```
flights %>%
  left_join(airlines, by="carrier")
```

RHS variables for LHS observations with no matching RHS observations are coded as `NA`.

7.5.1.1 Right Join

All observations on right operand (RHS) are retained:

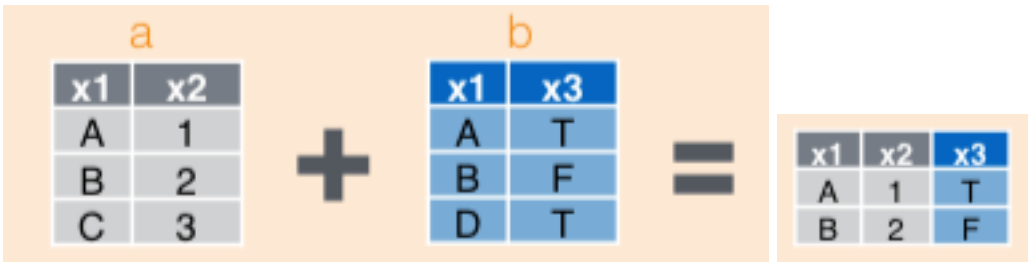



```
flights %>%
  right_join(airlines, by="carrier")
```

LHS variables for RHS observations with no matching LHS observations are coded as `NA`.

7.5.1.2 Inner Join

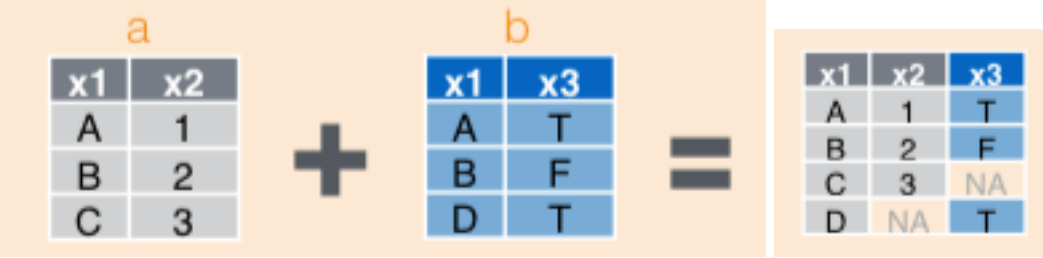
Only observations matching on both tables are retained




```
flights %>%
  inner_join(airlines, by="carrier")
```

7.5.1.3 Full Join

All observations are retained, regardless of matching condition




```
flights %>%
  full_join(airlines, by="carrier")
```

All values coded as `NA` for non-matching observations as appropriate.

7.5.2 Join conditions

All join operations are based on a matching condition:

```
flights %>%
  left_join(airlines, by="carrier")
```

specifies to join observations where `flights$carrier` equals `airlines$carrier`.

In this case, where no conditions are specified using the `by` argument:

```
flights %>%
  left_join(airlines)
```

a *natural join* is perfomed. In this case all variables with the same name in both tables are used in join condition.

You can also specify join conditions on arbitrary attributes using the `by` argument.

```
flights %>%
  left_join(airlines, by=c("carrier" = "name"))
```

7.5.3 Filtering Joins

We've just seen *mutating joins* that create new tables. *Filtering joins* use join conditions to filter a specific table.

```
flights %>% anti_join(airlines, by="carrier")
```

```
## # A tibble: 0 x 19
## # ... with 19 variables: year <int>, month <int>, day <int>,
## #   dep_time <int>, sched_dep_time <int>, dep_delay <dbl>, arr_time <int>,
## #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>, flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>,
## #   distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Filters the `flights` table to only include flights from airlines that *are not* included in the `airlines` table.

7.6 Final note on `dplyr`

- Very efficient implementation of these operations.
- More info: http://cran.rstudio.com/web/packages/dplyr/vignettes/introduction.html
- Cheatsheet: http://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf

7.7 Exercise

1. Clean up the Diego Luna ratings data scraped in the previous unit.

- Only keep movies that have a rating and that Diego Luna acts in
- Convert the rating to a numeric variable (use the `str_replace` and `type_convert` functions)

2. Clean up the movie budget data scraped in the previous unit.

- Remove rows with missing values
- Make the budget and revenue columns numeric and expressed in millions

3. Join the Diego Luna ratings and movie budget data using the movie title as the join variable
4. Think and implement ways of making the integration of these two datasets more robust.

Chapter 8

Visualizing data

We are now entering an important step of what we would want to do with a dataset before starting modeling using statistics or Machine Learning. We have seen manipulations and operations that prepare datasets into tidy (or normal form), compute summaries, and join tables to obtain organized, clean data tables that contain the observational units, or entities, we want to model statistically.

At this point, we want to perform *Exploratory Data Analysis* to better understand the data at hand, and help us make decisions about appropriate statistical or Machine Learning methods, or data transformations that may be helpful to do. Moreover, there are many instances where statistical data modeling is not required to tell a clear and convincing story with data. Many times an effective visualization can lead to convincing conclusions.

8.0.1 EDA (Exploratory Data Analysis)

The goal of EDA is to perform an initial exploration of attributes/variables across entities/observations. In this section, we will concentrate on exploration of single or pairs of variables. Later on in the course we will see *dimensionality reduction* methods that are useful in exploration of more than two variables at a time.

Ultimately, the purpose of EDA is to spot problems in data (as part of data wrangling) and understand variable properties like:

- central trends (mean)
- spread (variance)
- skew
- suggest possible modeling strategies (e.g., probability distributions)

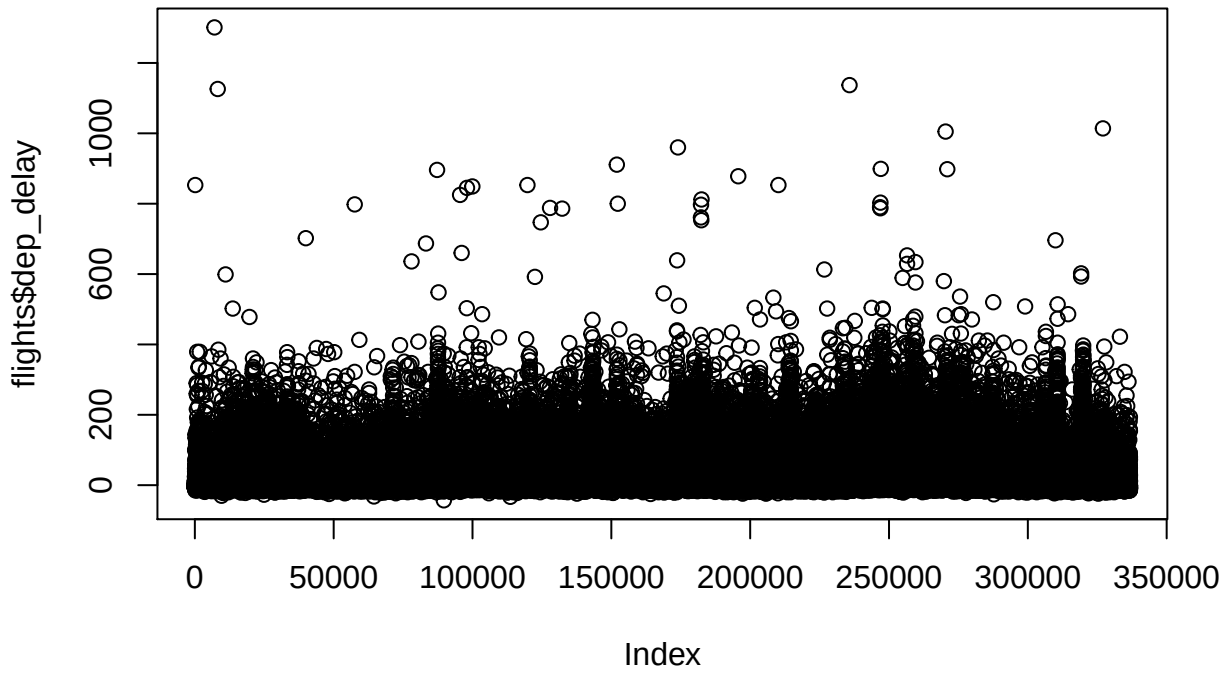
We also want to use EDA to understand relationships between pairs of variables, e.g. their correlation or covariance.

8.1 Visualization of single variables

Let's begin by using R's basic graphics capabilities which are great for creating quick plots especially for EDA. We will then see `ggplot2` that requires you to be a bit more thoughtful on data exploration that can lead to good ideas about analysis and modeling.

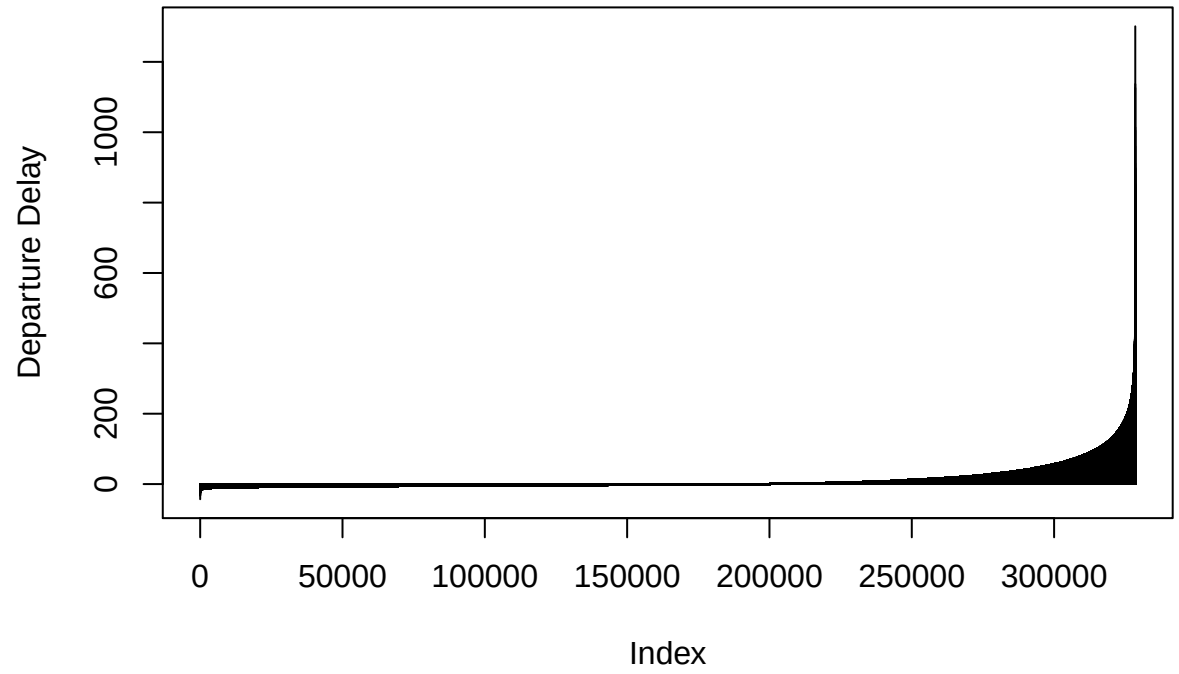
Let's start with a very simple visualization of the `dep_delay` variable in the flights dataset.

```
library(dplyr)
library(nycflights13)

plot(flights$dep_delay)
```



That's not particularly informative since there is no structure to this plot. Let's change the order of the points to depend on departure delay and change the graphical representation to make easier to see.

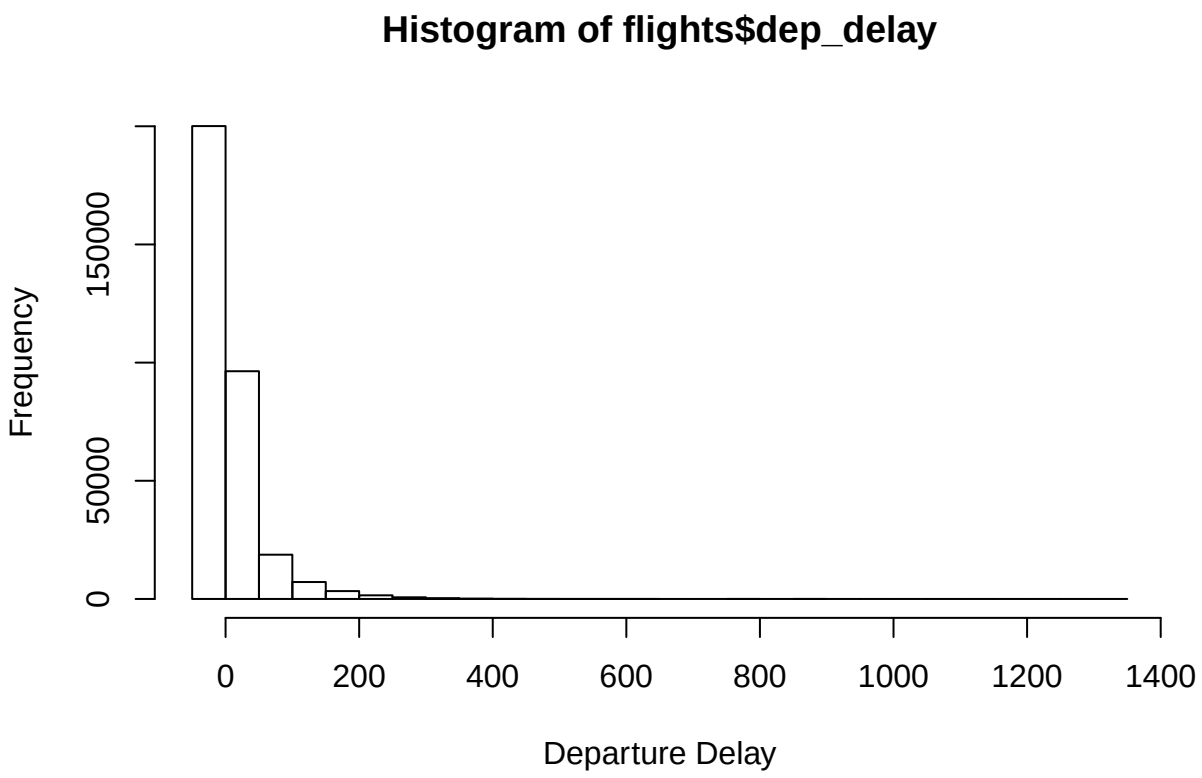
```
plot(sort(flights$dep_delay), type="h", ylab="Departure Delay")
```



What can we make of that plot now? Start thinking of *central tendency*, *spread* and *skew* as you look at that plot.

Let's now create a graphical summary of that variable to incorporate observations made from this initial plot. Let's start with a *histogram*: it divides the *range* of the `dep_delay` variable into **equal-sized** bins, then plots the number of observations within each bin. What additional information does this new visualization gives us about this variable?

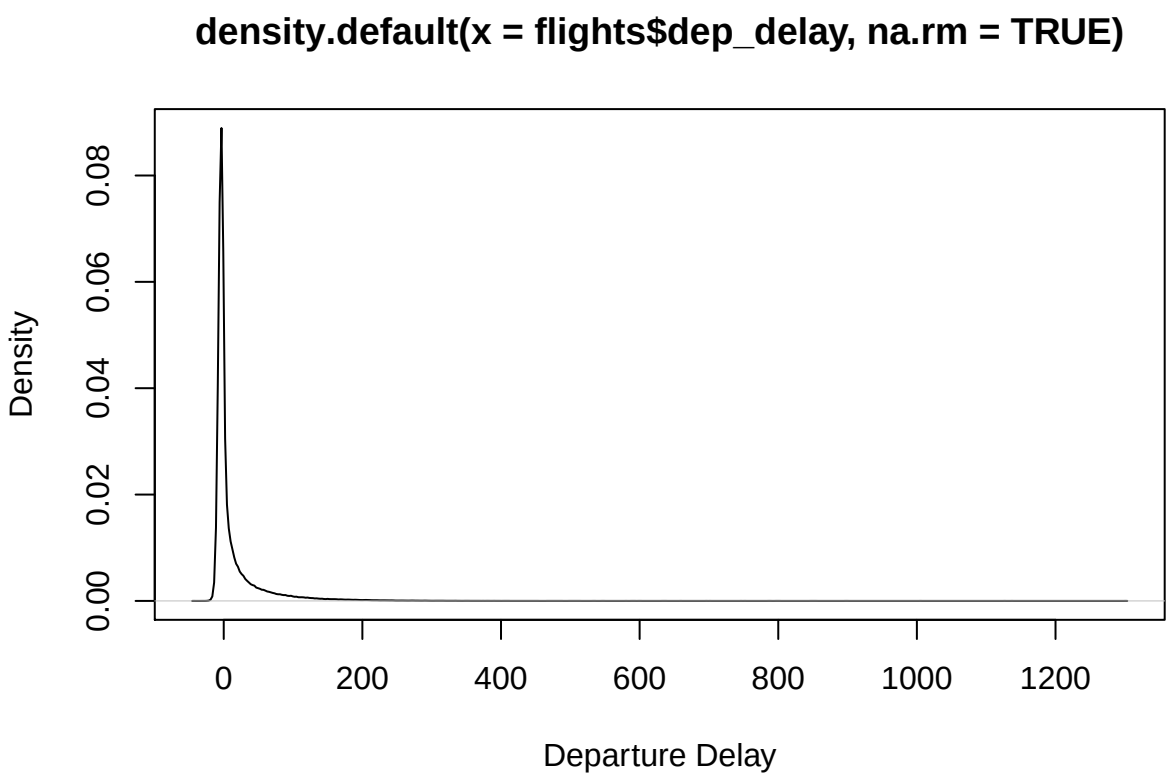
```
hist(flights$dep_delay, xlab="Departure Delay")
```



The `nclass` parameter controls the number of bins into which the `dep_delay` range is divided. Try changing that parameter and see what happens.

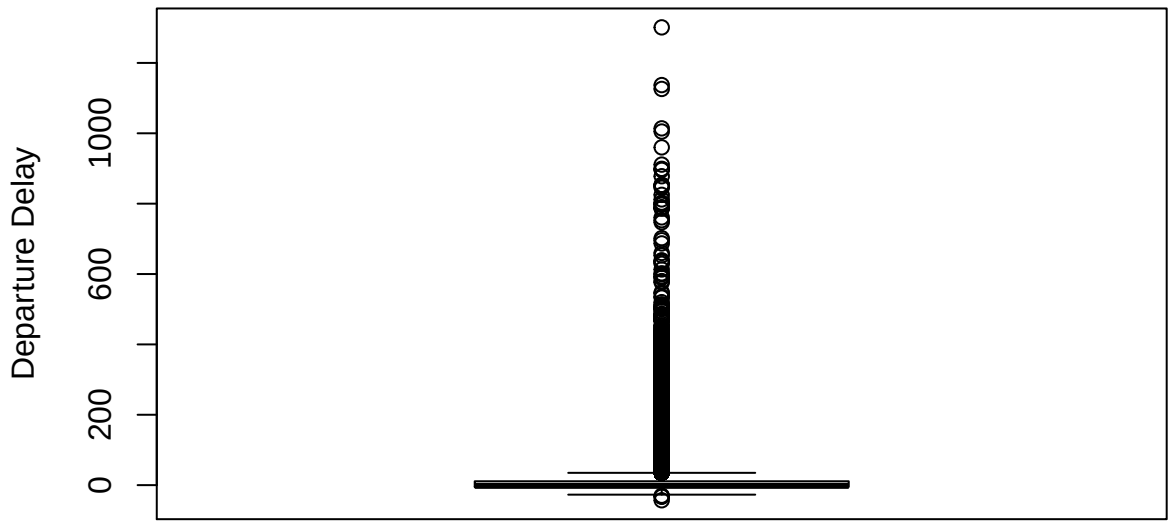
Now, we can (conceptually) make the bins as small as possible and get a smooth curve that describes the *distribution* of values of the `dep_delay` variable. We call this a *density* plot:

```
plot(density(flights$dep_delay, na.rm=TRUE), xlab="Departure Delay")
```



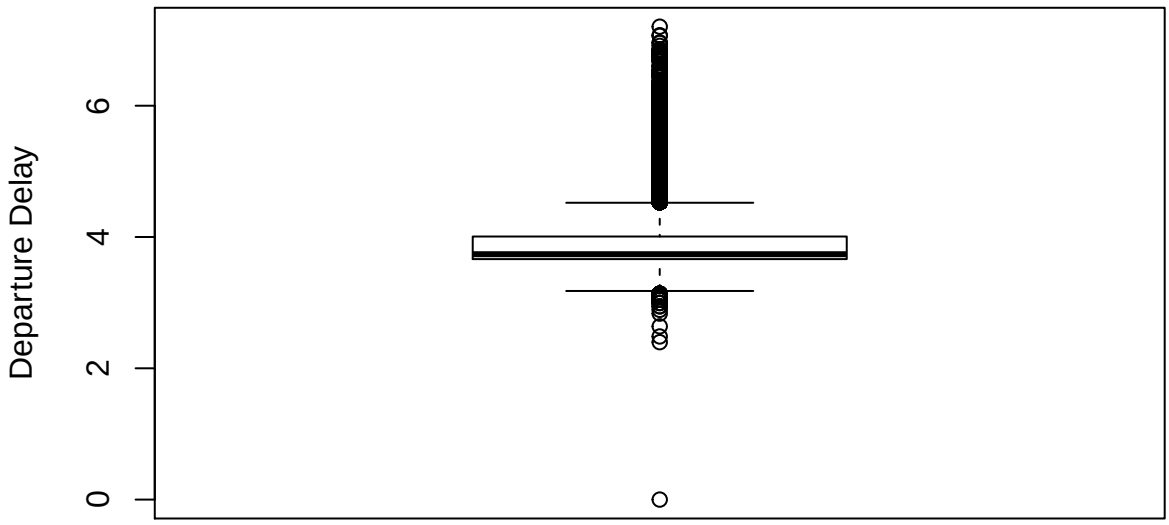
Now, one more very useful way of succintly graphically summarizing the distribution of a variable is using a `boxplot`.

```
boxplot(flights$dep_delay, ylab="Departure Delay")
```



That's not very clear to see, so let's do a transformation of this data to see this better:

```
boxplot(log(flights$dep_delay -
            min(flights$dep_delay,na.rm=TRUE)
            +1), ylab="Departure Delay")
```



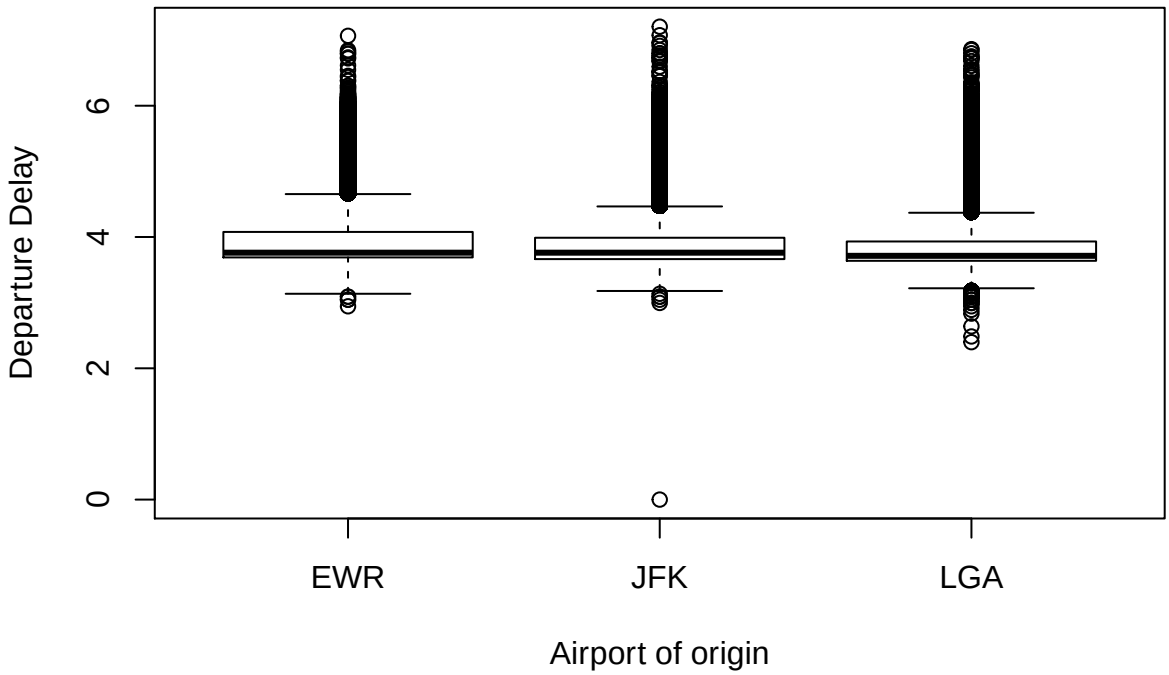
So what does this represent: (a) central tendency (using the median) is represented by the black line within the box, (b) spread (using inter-quartile range) is represented by the box and whiskers. (c) outliers (data that is *unusually* outside the spread of the data)

We will see more formal descriptions of these summary statistics in the next section, but you can see what we are trying to capture with them graphically.

8.1.1 Visualization of pairs of variables

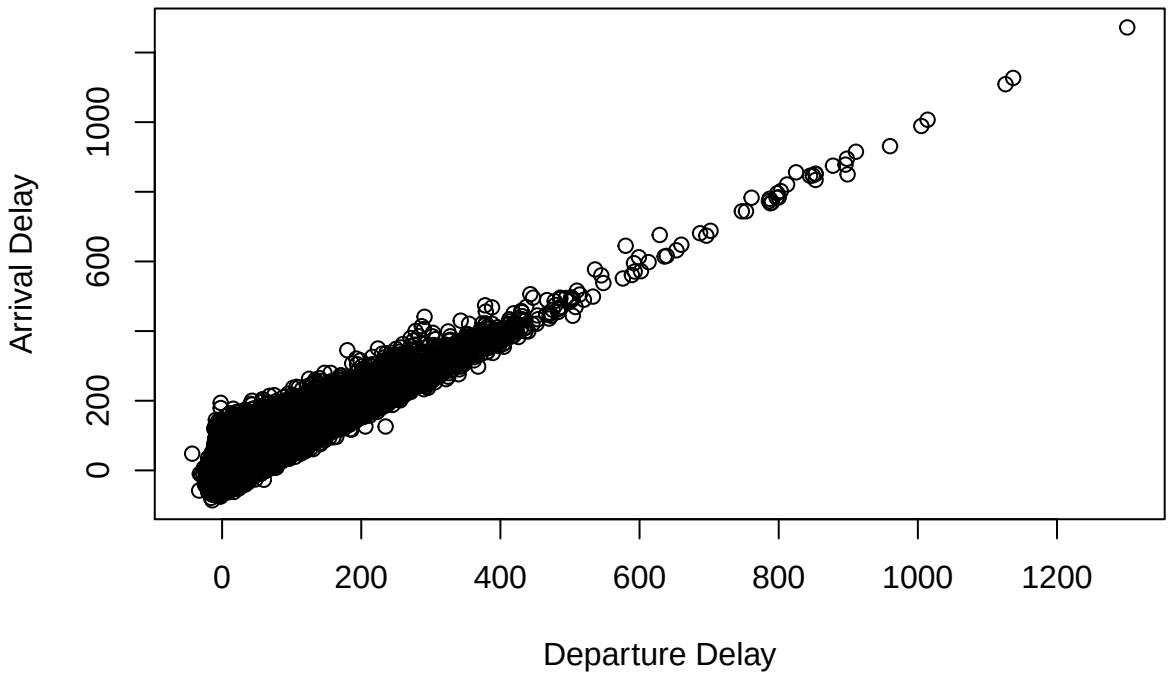
Now we can start looking at the relationship between pairs of variables. Suppose we want to see the relationship between `dep_delay`, a *numeric* variable, and `origin`, a *categorical* variable. The neat thing here is that we can start thinking about conditioning as we saw before. Here is how we can see a plot of the distribution of departure delays *conditioned* on origin airport.

```
boxplot(log(flights$dep_delay - min(flights$dep_delay, na.rm=TRUE) + 1) ~ flights$origin,
        ylab="Departure Delay", xlab="Airport of origin")
```



For pairs of continuous variables, the most useful visualization is the scatter plot. This gives an idea of how one variable varies conditioned on another variable.

```
plot(flights$dep_delay, flights$arr_delay, xlab="Departure Delay", ylab="Arrival Delay")
```



8.2 EDA with the grammar of graphics

While we have seen a basic repertoire of graphics it's easier to proceed if we have a bit more formal way of thinking about graphics and plots. Here is where we will use the *grammar of graphics* implemented in R by

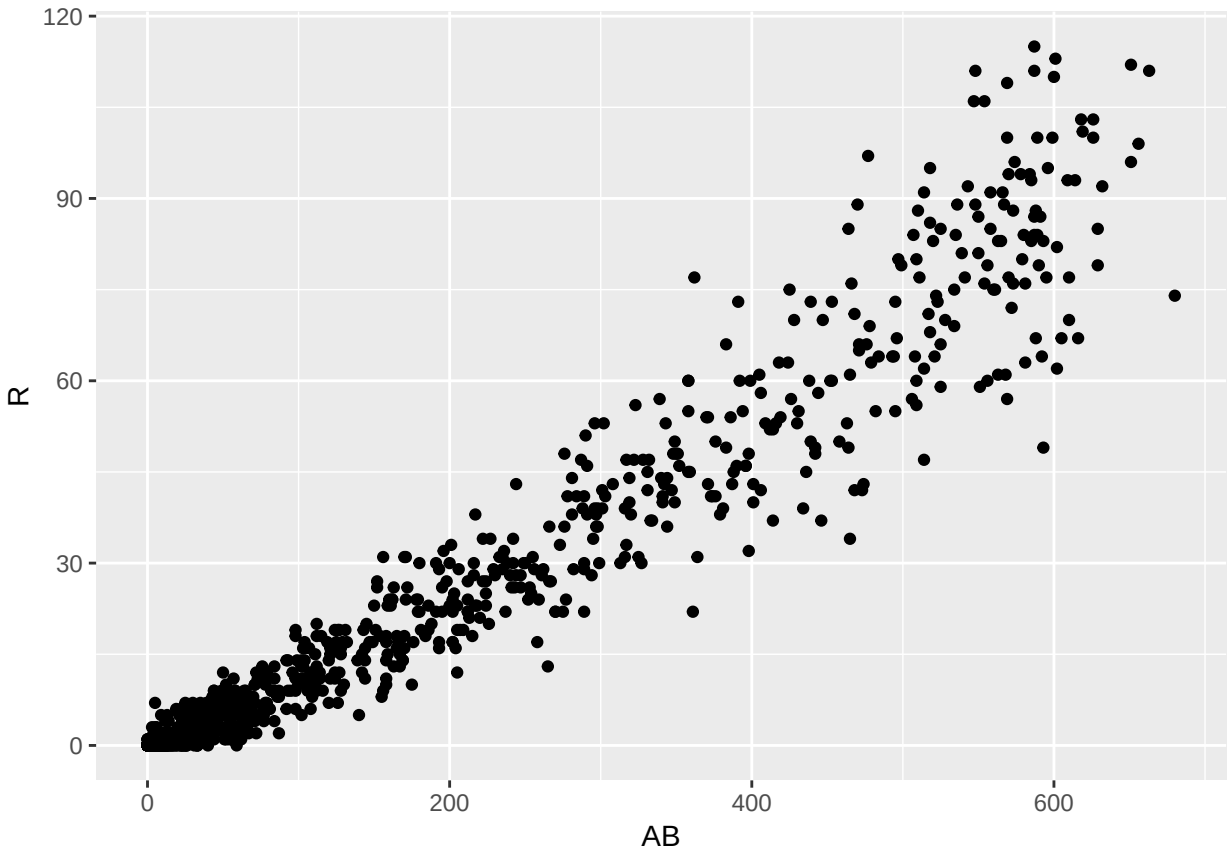
the package `ggplot2`.

The central premise is to characterize the building pieces behind plots:

1. The data that goes into a plot, works best when data is tidy
2. The mapping between data and *aesthetic* attributes
3. The *geometric* representation of these attributes

Let's start with a simple example:

```
library(dplyr)
library(ggplot2)
library(Lahman)
batting <- tbl_df(Batting)

# scatter plot of at bats vs. runs for 2010
batting %>%
  filter(yearID == "2010") %>%
  ggplot(aes(x=AB, y=R)) +
    geom_point()
```

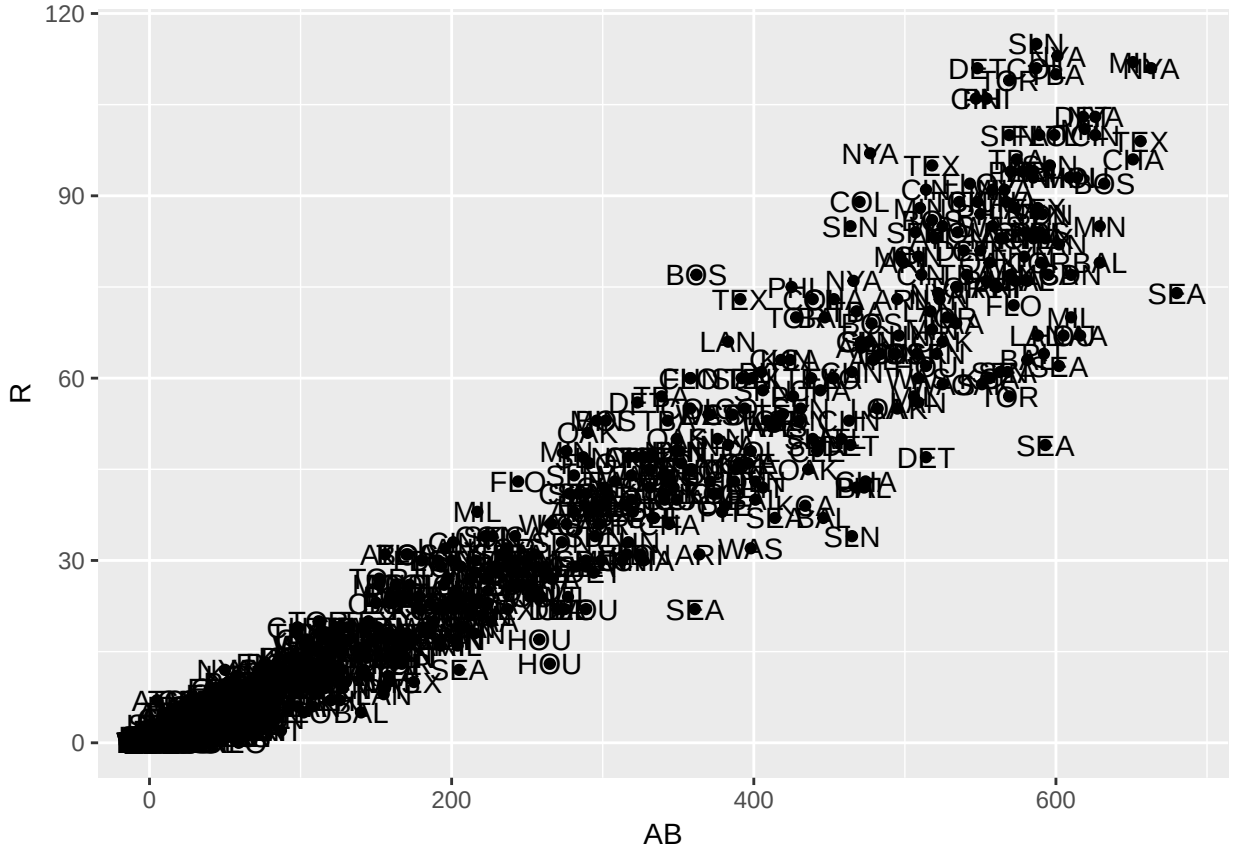


Data: Batting table filtering for year
Aesthetic attributes: - x-axis mapped to variables `AB` - y-axis mapped to variable `R`

Geometric Representation: points!

Now, you can cleanly distinguish the constituent parts of the plot. E.g., change the geometric representation

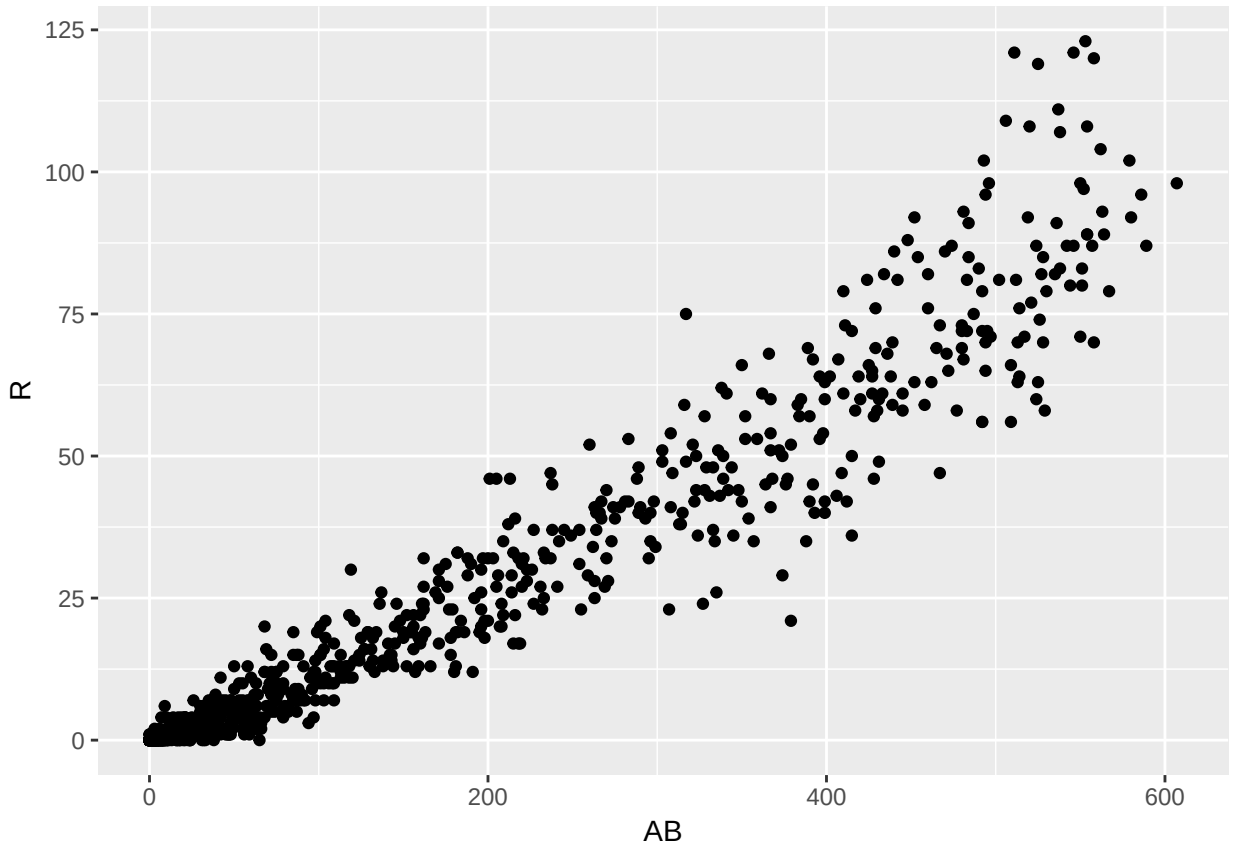
```
# scatter plot of at bats vs. runs for 2010
batting %>%
  filter(yearID == "2010") %>%
  ggplot(aes(x=AB, y=R, label=teamID)) +
    geom_text() +
    geom_point()
```



E.g., change the data.

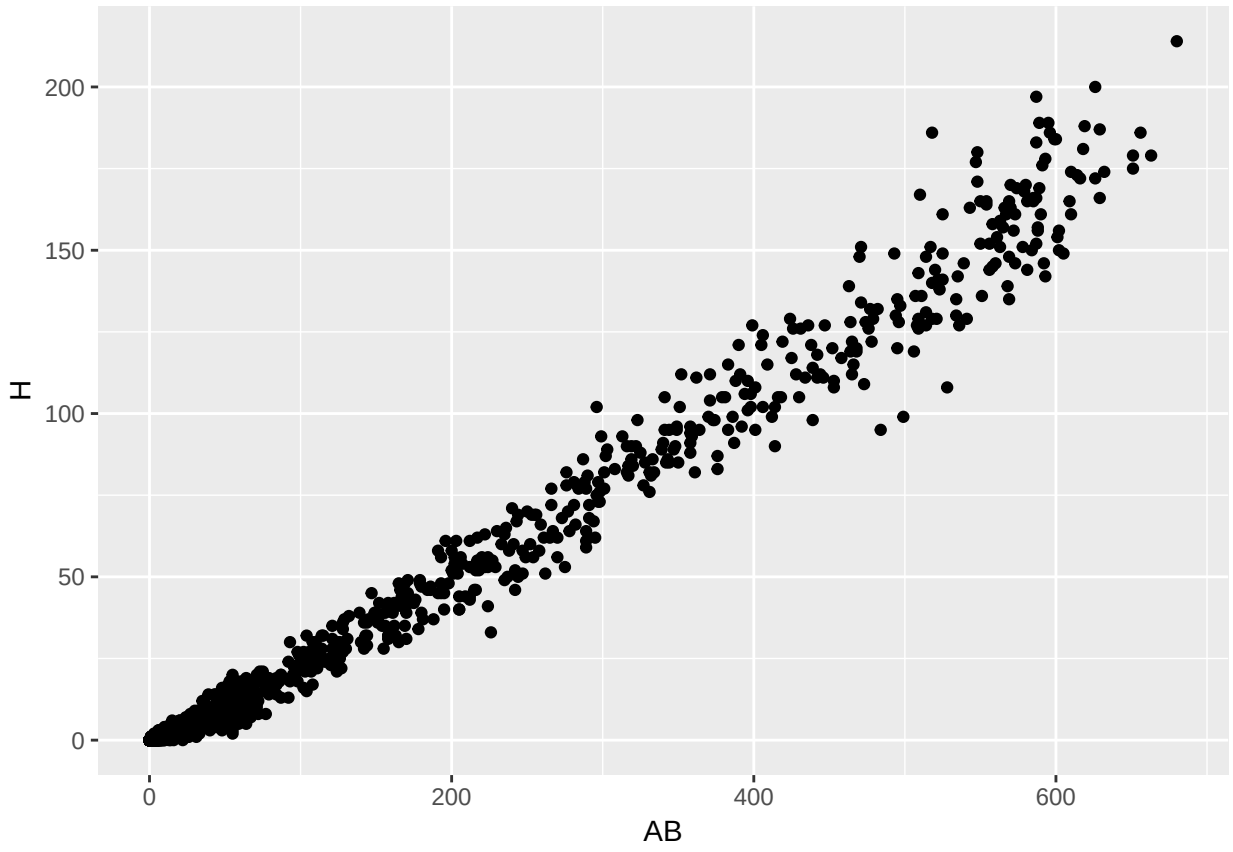
```
# scatter plot of at bats vs. runs for 1995
batting %>%
  filter(yearID == "1995") %>%
  ggplot(aes(x=AB, y=R)) +
    geom_point()
```

```
## Warning: Removed 279 rows containing missing values (geom_point).
```



E.g., change the aesthetic.

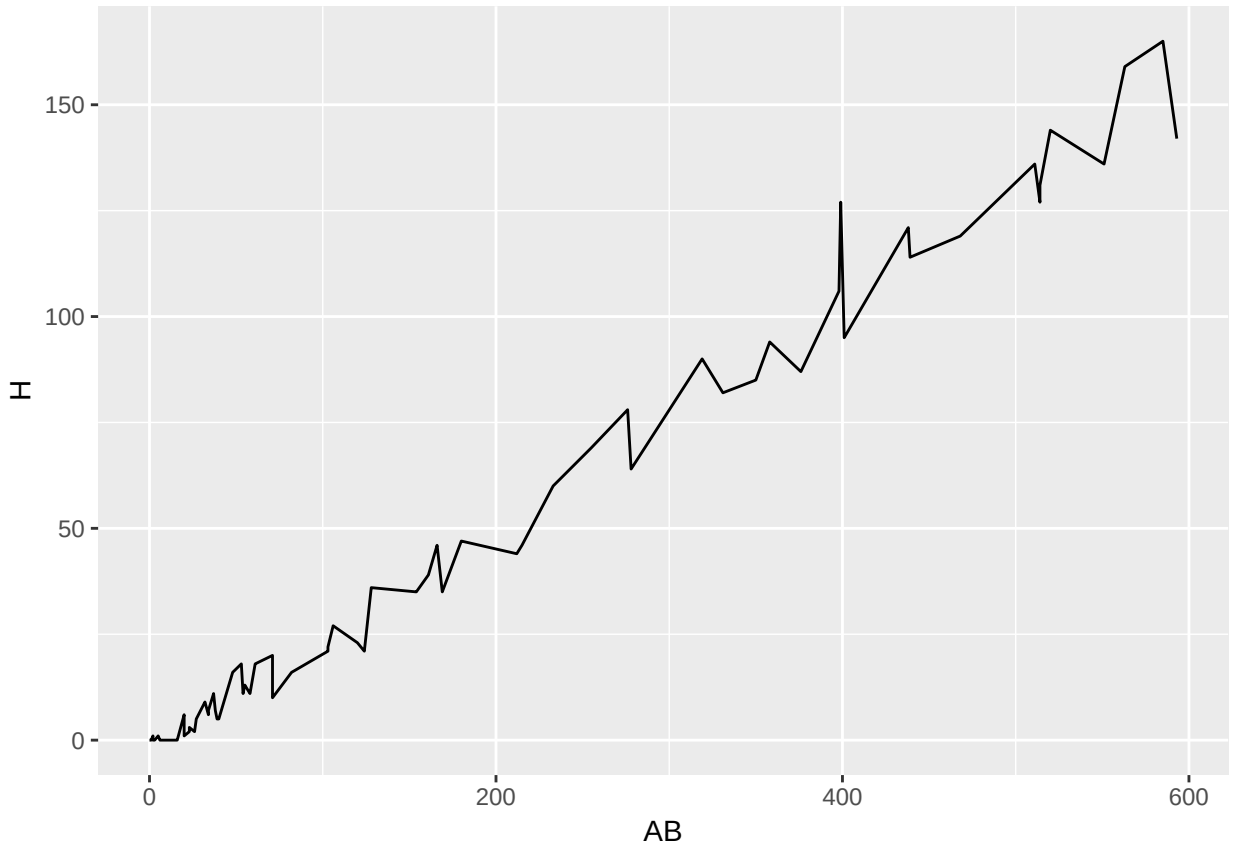
```
# scatter plot of at bats vs. hits for 2010
batting %>%
  filter(yearID == "2010") %>%
  ggplot(aes(x=AB, y=H)) +
    geom_point()
```



Let's make a line plot

What do we change? (data, aesthetic or geometry?)

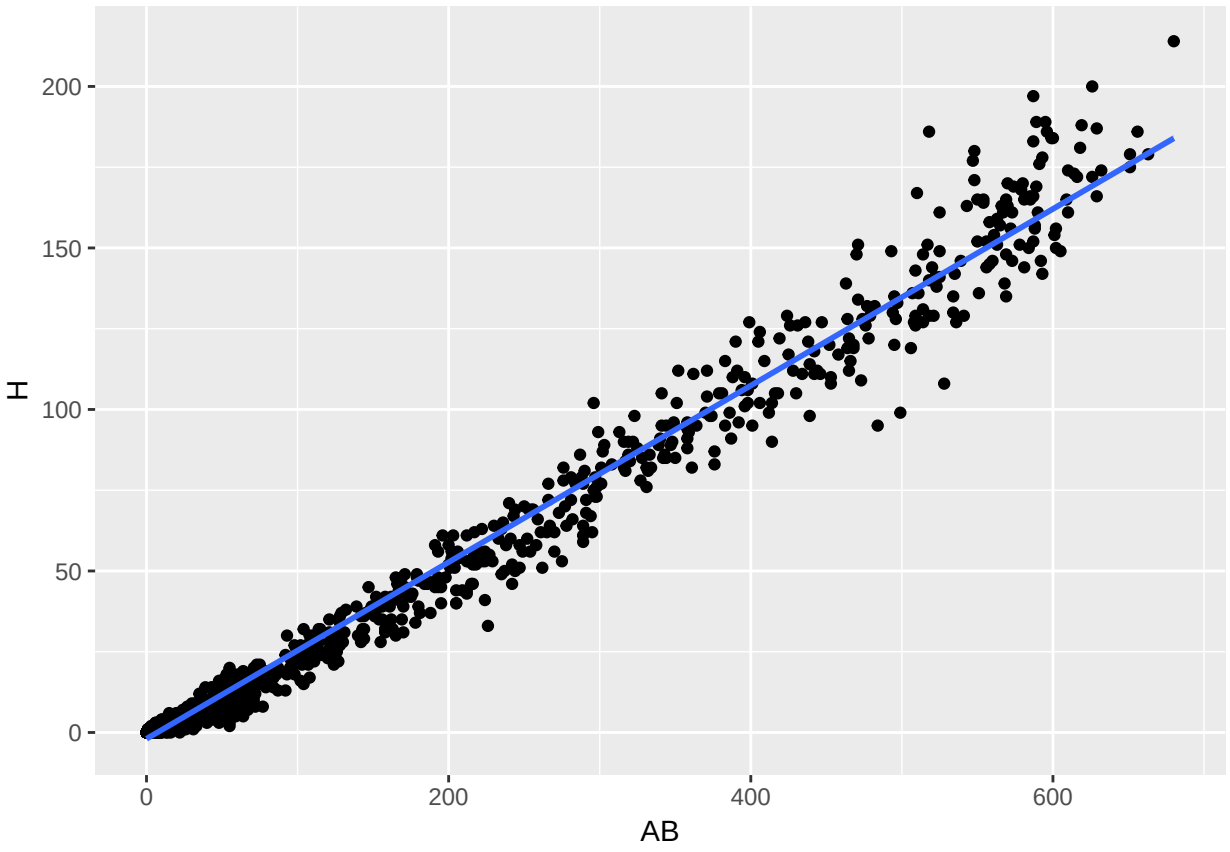
```
batting %>%
  filter(yearID == "2010") %>%
  sample_n(100) %>%
  ggplot(aes(x=AB, y=H)) +
  geom_line()
```



Let's add a regression line

What do we add? (data, aesthetic or geometry?)

```
batting %>%
  filter(yearID == "2010") %>%
  ggplot(aes(x=AB, y=H)) +
  geom_point() +
  geom_smooth(method=lm)
```

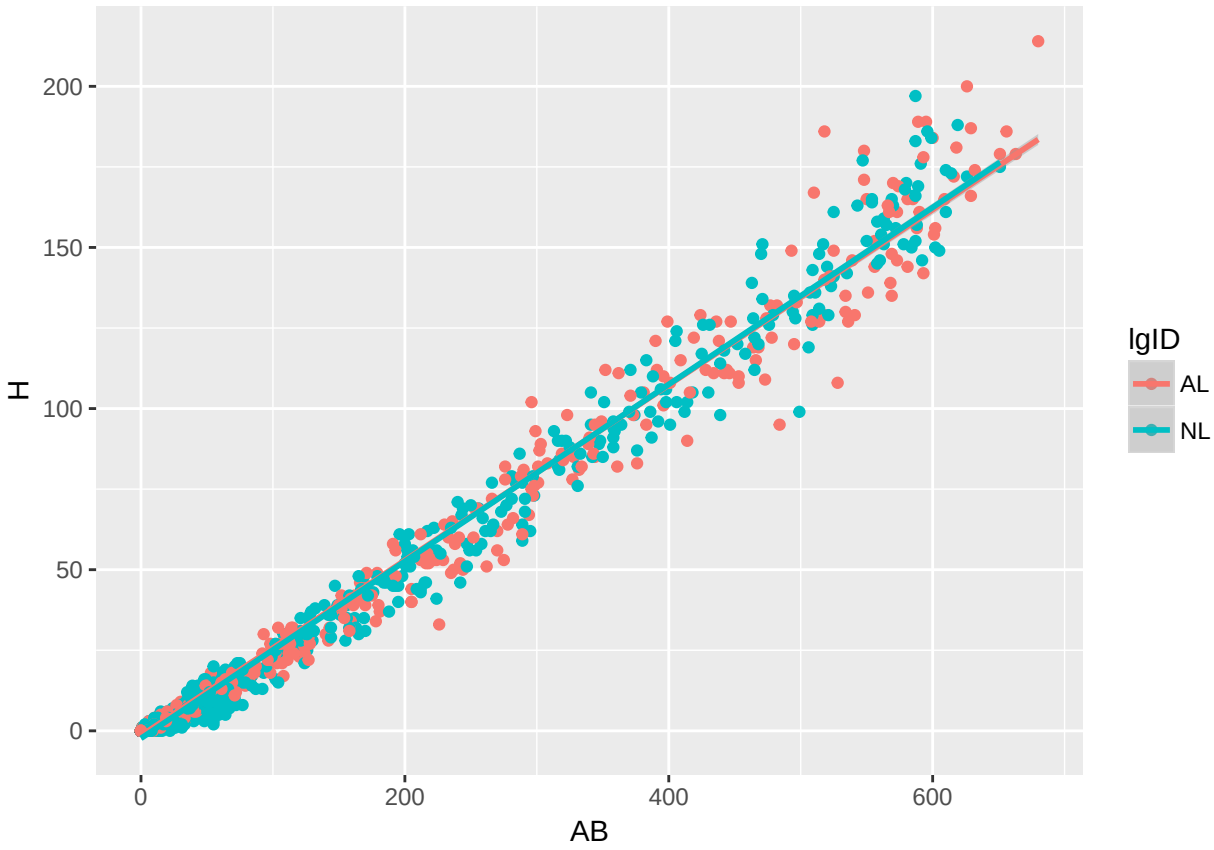


8.2.1 Other aesthetics

Using other aesthetics we can incorporate information from other variables.

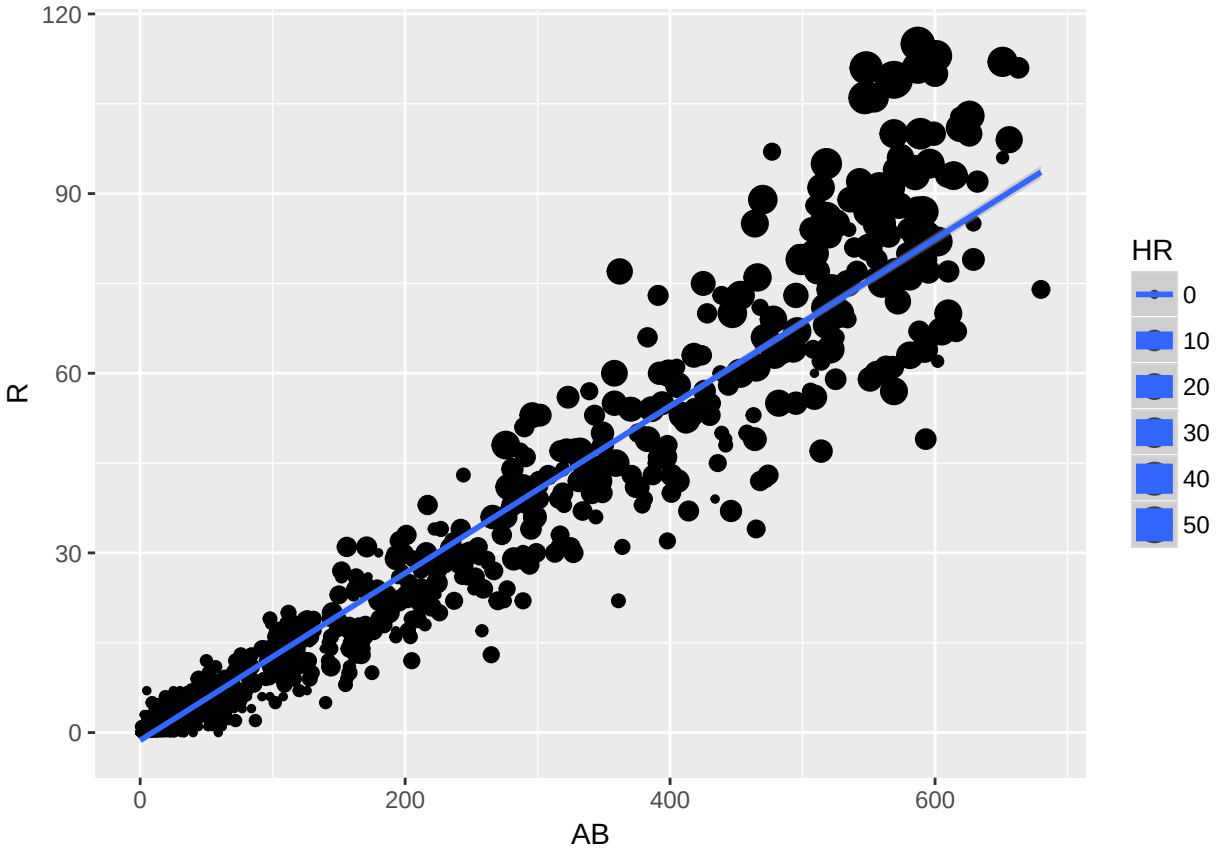
Color: color by categorical variable

```
batting %>%
  filter(yearID == "2010") %>%
  ggplot(aes(x=AB, y=H, color=lgID)) +
  geom_point() +
  geom_smooth(method=lm)
```



Size: size by (discrete) numeric variable

```
batting %>%
  filter(yearID == "2010") %>%
  ggplot(aes(x=AB, y=R, size=HR)) +
  geom_point() +
  geom_smooth(method=lm)
```



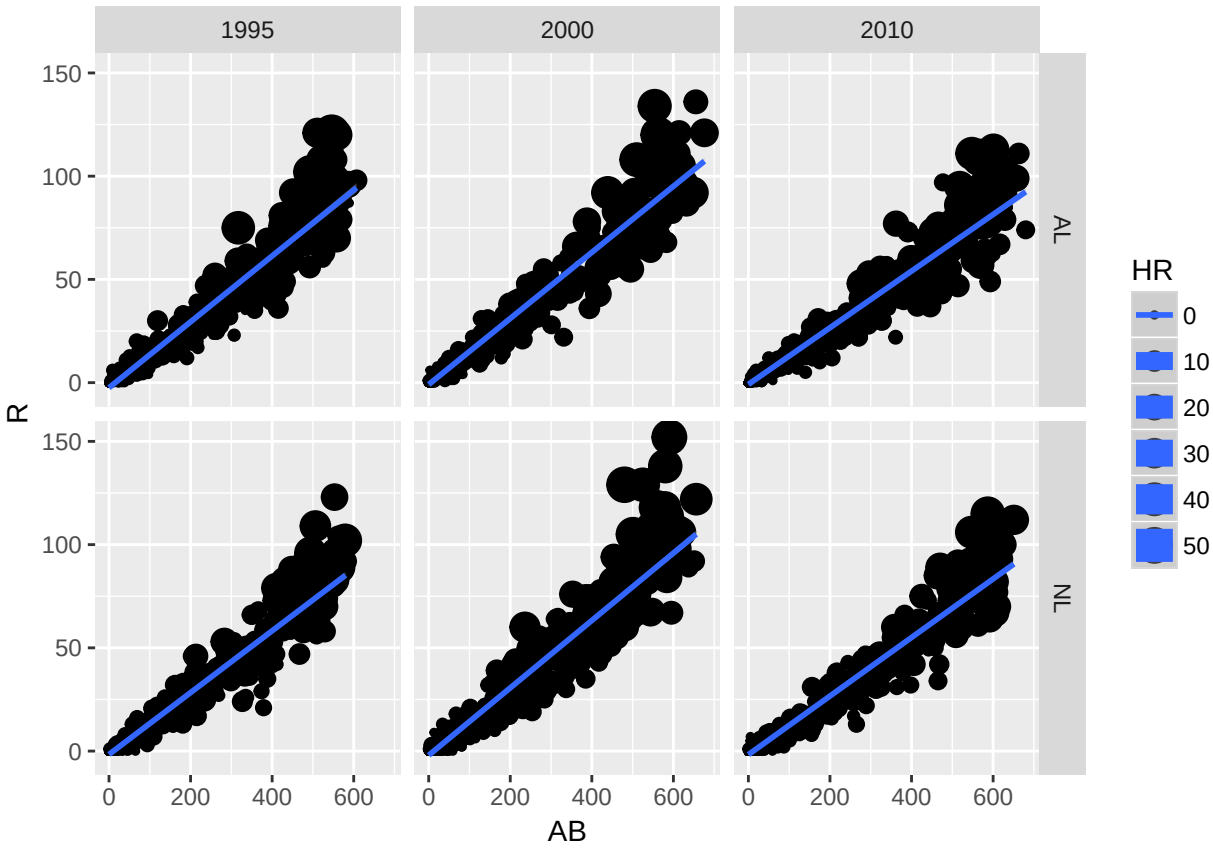
8.2.2 Faceting

The last major component of exploratory analysis called `faceting` in visualization, corresponds to `conditioning` in statistical modeling, we've seen it as the motivation of `grouping` when wrangling data.

```
batting %>%
  filter(yearID %in% c("1995", "2000", "2010")) %>%
  ggplot(aes(x=AB, y=R, size=HR)) +
  facet_grid(lgID~yearID) +
  geom_point() +
  geom_smooth(method=lm)
```

```
## Warning: Removed 279 rows containing non-finite values (stat_smooth).
```

```
## Warning: Removed 279 rows containing missing values (geom_point).
```



8.3 Exercise

Use `ggplot2` to make a scatter plot of domestic gross revenue vs. Rotten Tomato rating for Diego Luna's movies.

Chapter 9

Capstone Project 1

9.1 Project 1

- Write a script that includes all steps for scraping, cleaning, manipulating and making the Diego Luna movie analysis plot.
- Write a set of functions to perform the same analysis given an actor's name

9.2 Project 2

Download data from the ENADID and make a plot of internal vs. external migration patterns in Mexico from 1992 to 2014. Data for 2009 can be found here: http://www.beta.inegi.org.mx/proyectos/enchogares/especiales/enadid/2009/default.html. Data for other years is also available there. Write your analysis in an R script.

Chapter 10

Introduction to tidy regression analysis

Linear regression is a very elegant, simple, powerful and commonly used technique for data analysis.

10.1 Simple Regression

Let's start with the simplest linear model. The goal here is to analyze the relationship between a *continuous numerical* variable Y and another (*numerical* or *categorical*) variable X. We assume that in our population of interest the relationship between the two is given by a linear function:

$$Y = \beta_0 + \beta_1 X$$

Here is (simulated) data from an advertising campaign measuring sales and the amount spent in advertising. We think that sales are related to the amount of money spent on TV advertising:

$$\texttt{sales} \approx \beta_0 + \beta_1 \times \texttt{TV}$$

Given this data, we would say that we *regress* `sales` on `TV` when we perform this regression analysis. As before, given data we would like to estimate what this relationship is in the *population* (what is the population in this case?). What do we need to estimate in this case? Values for β_0 and β_1.

Let's take a look at some data. Here is data measuring characteristics of cars, including horsepower, weight, displacement, miles per gallon. Let's see how well a linear model captures the relationship between miles per gallon and weight

```
library(ISLR)
library(dplyr)
library(ggplot2)
library(broom)

data(Auto)

Auto %>%
  ggplot(aes(x=weight, y=mpg)) +
    geom_point() +
```

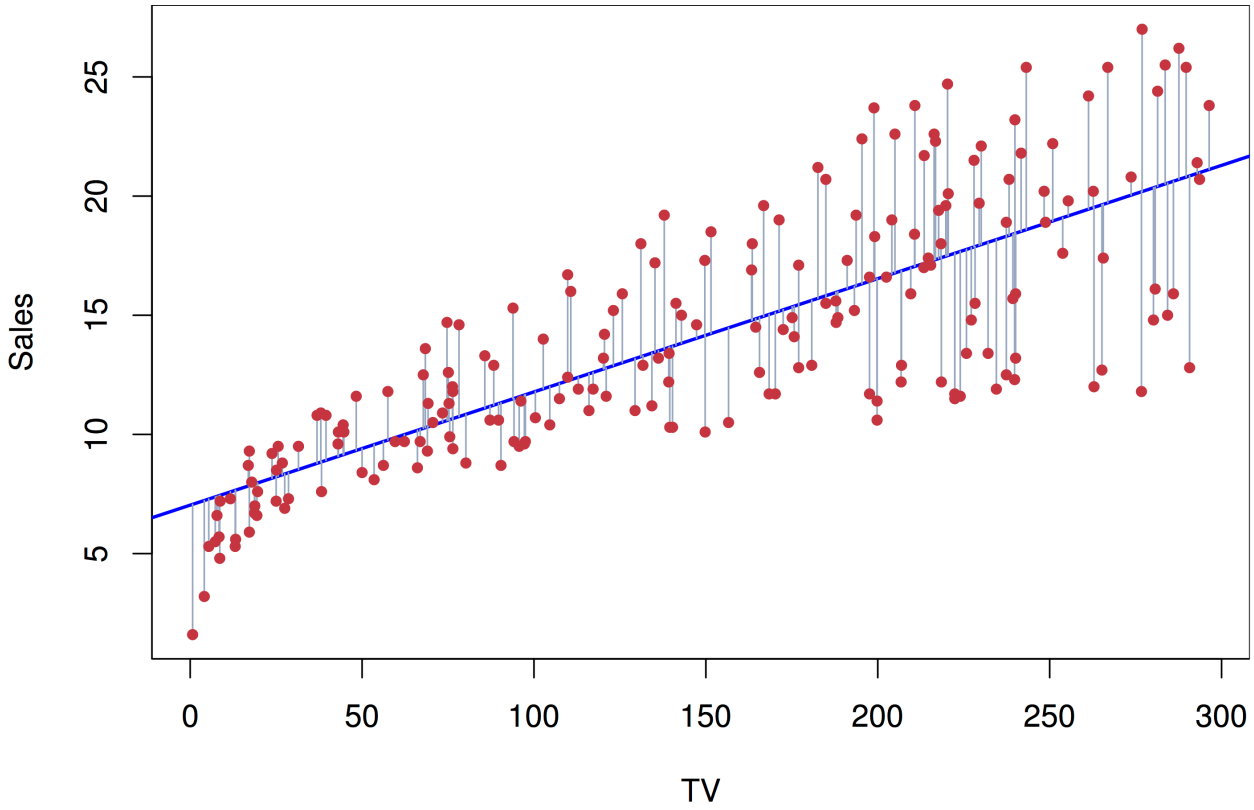
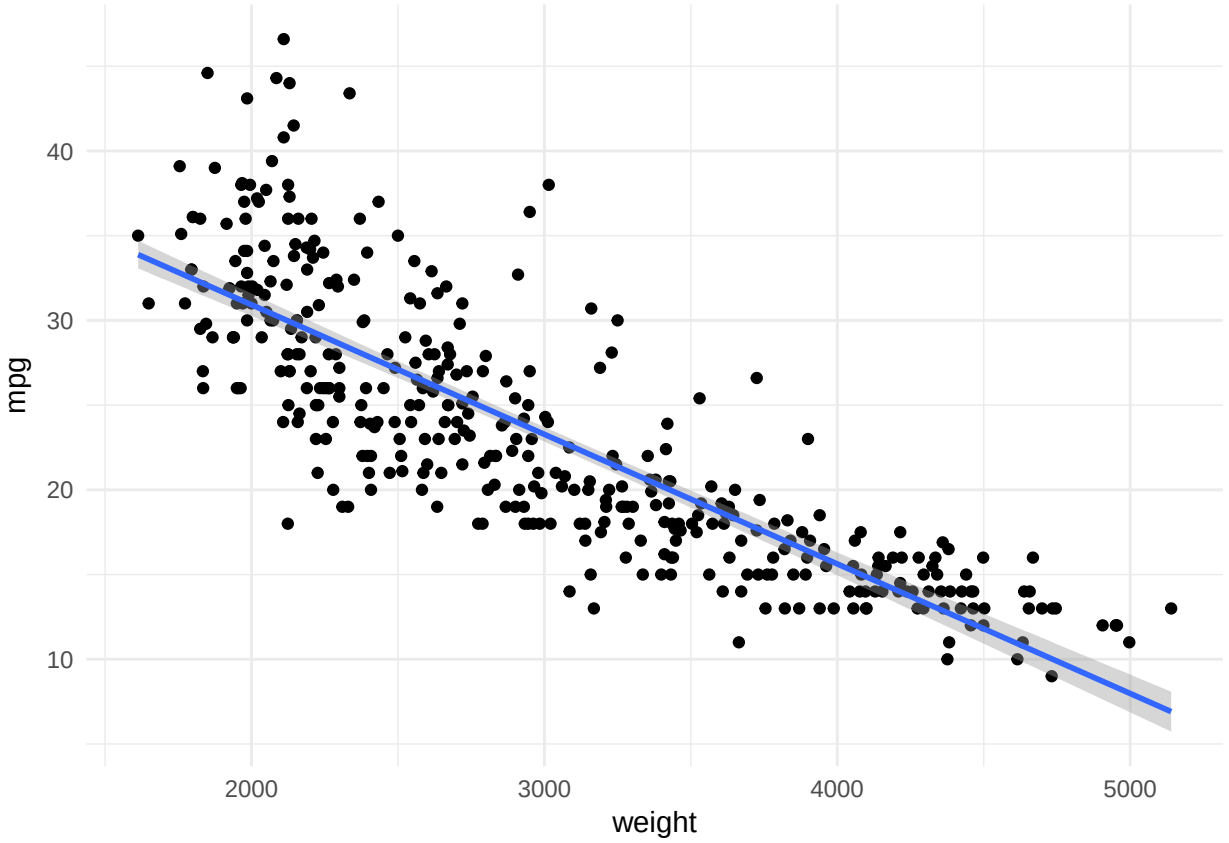


Figure 10.1:

```
    geom_smooth(method=lm) +
    theme_minimal()
```



In R, linear models are built using the `lm` function

```
auto_fit <- lm(mpg~weight, data=Auto)
auto_fit
```

```
##
## Call:
## lm(formula = mpg ~ weight, data = Auto)
##
## Coefficients:
## (Intercept)       weight
##   46.216525    -0.007647
```

This states that for this dataset $\hat{\beta}_0 = 46.2165245$ and $\hat{\beta}_1 = -0.0076473$. What's the interpretation? According to this model, a weightless car `weight=0` would run ≈ 46.22 *miles per gallon* on average, and, on average, a car would run ≈ 0.008 *miles per gallon* fewer for every extra *pound* of weight. Note, that the units of the outcome Y and the predictor X matter for the interpretation of these values.

10.2 Inference

Now that we have an estimate, we want to know how good of an estimate this is. An important point to understand is that like the sample mean, the regression line we learn from a specific dataset is an estimate. A different sample from the same population would give us a different estimate (regression line).

But, statistical theory tells us that, on average, we are close to population regression line (I.e., close to β_0 and β_1), that the spread around β_0 and β_1 is well approximated by a normal distribution and that the spread goes to zero as the sample size increases.

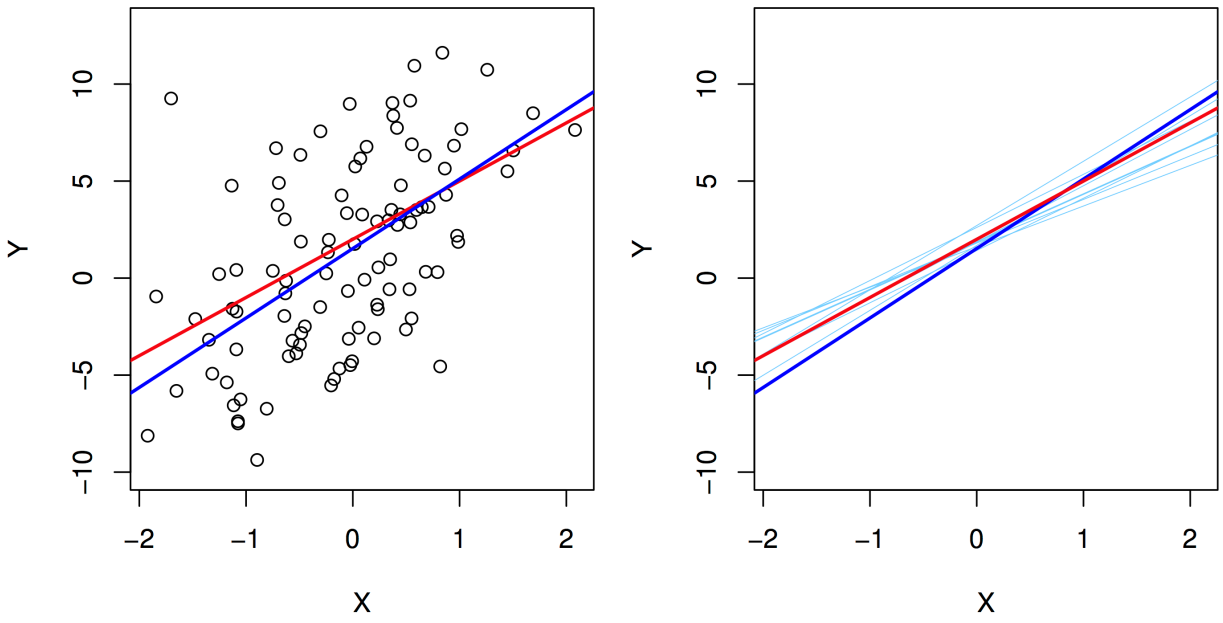


Figure 10.2:

10.2.1 Confidence Interval

We can construct a confidence interval to say how precise we think our estimates of the population regression line is. In particular, we want to see how precise our estimate of β_1 is, since that captures the relationship between the two variables.

```
auto_fit_stats <- auto_fit %>%
  tidy() %>%
  select(term, estimate, std.error)
auto_fit_stats
```

```
##          term     estimate    std.error
## 1 (Intercept) 46.216524549 0.7986724633
## 2      weight -0.007647343 0.0002579633
```

This `tidy` function is defined by the `broom` package, which is very handy to manipulate the result of learning models in a consistent manner. The `select` call removes some extra information that we will discuss shortly.

Given the confidence interval, we would say, "on average, a car runs $_{-0.0082} - 0.0076_{-0.0071}$ *miles per gallon* fewer per pound of weight.

10.2.2 The t-statistic and the t-distribution

We can also test a null hypothesis about this relationship: "there is no relationship between weight and miles per gallon", which translates to $\beta_1 = 0$. According to the statistical theory if this hypothesis is true then the

distribution of $\hat{\beta}_1$ is well approximated by $N(0, \text{se}(\hat{\beta}_1))$, and if we observe the estimated $\hat{\beta}_1$ is *too far* from 0 according to this distribution then we *reject* the hypothesis.

Now, there is a technicality here that is worth paying attention to. The normal approximation is good as sample size increases, but what about moderate sample sizes (say, less than 100)? The t distribution provides a better approximation of the sampling distribution of these estimates for moderate sample sizes, and it tends to the normal distribution as sample size increases.

The t distribution is commonly used in this testing situation to obtain the probability of rejecting the null hypothesis. It is based on the t-statistic

$$\frac{\hat{\beta}_1}{\text{se}(\hat{\beta}_1)}$$

You can think of this as a *signal-to-noise* ratio, or a standardizing transformation on the estimated parameter. Under the null hypothesis, it was shown that the t-statistic is well approximated by a t-distribution with $n-2$ *degrees of freedom* (we will get back to *degrees of freedom* shortly).

In our example, we get a t statistic and P-value as follows:

```
auto_fit_stats <- auto_fit %>%
  tidy()
auto_fit_stats
```

```
##          term     estimate    std.error statistic       p.value
## 1 (Intercept) 46.216524549 0.7986724633  57.86668 1.623069e-193
## 2      weight -0.007647343 0.0002579633 -29.64508 6.015296e-102
```

We would say: "We found a statistically significant relationship between weight and miles per gallon. On average, a car runs $_{-0.0082} - 0.0076_{-0.0071}$ *miles per gallon* fewer per pound of weight (t=-29.65, p-value $< 6.02 \times 10^{-102}$)."

10.2.3 Global Fit

Now, notice that we can make *predictions* based on our regression model, and that prediction should be better than a prediction with a simple average. We can use this comparison as a measure of how good of a job we are doing using our model to fit this data: how much of the variance of Y can we *explain* with our model. To do this we can calculate *total sum of squares*:

$$TSS = \sum_i (y_i - \overline{y})^2$$

(this is the squared error of a prediction using the sample mean of Y)

and the *residual sum of squares*:

$$RSS = \sum_i (y_i - \hat{y}_i)^2$$

(which is the squared error of a prediction using the linear model we learned)

The commonly used R^2 measure comparse these two quantities:

$$R^2 = \frac{\text{TSS} - \text{RSS}}{\text{TSS}} = 1 - \frac{\text{RSS}}{\text{TSS}}$$

These types of global statistics for the linear model can be obtained using the `glance` function in the `broom` package. In our example

```
auto_fit %>%
  glance() %>%
  select(r.squared, sigma, statistic, df, p.value)
```

```
##   r.squared    sigma statistic df       p.value
## 1 0.6926304 4.332712  878.8309  2 6.015296e-102
```

We will explain the the columns `statistic`, `df` and `p.value` when we discuss regression using more than a single predictor X.

10.3 Some important technicalities

We mentioned above that predictor X could be *numeric* or *categorical*. However, this is not precisely true. We can use a transformation to represent *categorical* variables. Here is a simple example:

Suppose we have a categorical variable `sex` with values `female` and `male`, and we want to show the relationship between, say `credit card balance` and `sex`. We can create a dummy variable x as follows:

$$x_i = \begin{cases} 1 \text{ if female} \\ 0 \text{o.w.} \end{cases}$$

and fit a model $y = \beta_0 + \beta_1 x$. What is the conditional expectation given by this model? If the person is male, then $y = \beta_0$, if the person is female, then $y = \beta_0 + \beta_1$. So, what is the interpretation of β_1? The average difference in credit card balance between females and males.

We could do a different encoding:

$$x_i = \begin{cases} +1 \text{ if female} \\ -1 \text{o.w.} \end{cases}$$

Then what is the interpretation of β_1 in this case?

Note, that when we call the `lm(y~x)` function and `x` is a factor with two levels, the first transformation is used by default. What if there are more than 2 levels? We need multiple regression, which we will see shortly.

10.4 Issues with linear regression

There are some assumptions underlying the inferences and predictions we make using linear regression that we should verify are met when we use this framework. Let's start with four important ones that apply to simple regression

10.4.1 Non-linearity of outcome-predictor relationship

What if the underlying relationship is not linear? We will see later that we can capture non-linear relationships between variables, but for now, let's concentrate on detecting if a linear relationship is a good approximation. We can use exploratory visual analysis to do this for now by plotting residuals $(y_i - \hat{y}_i)^2$ as a function of the fitted values $\hat{y}_i$.

The `broom` package uses the `augment` function to help with this task. It augments the input data used to learn the linear model with information of the fitted model for each observation

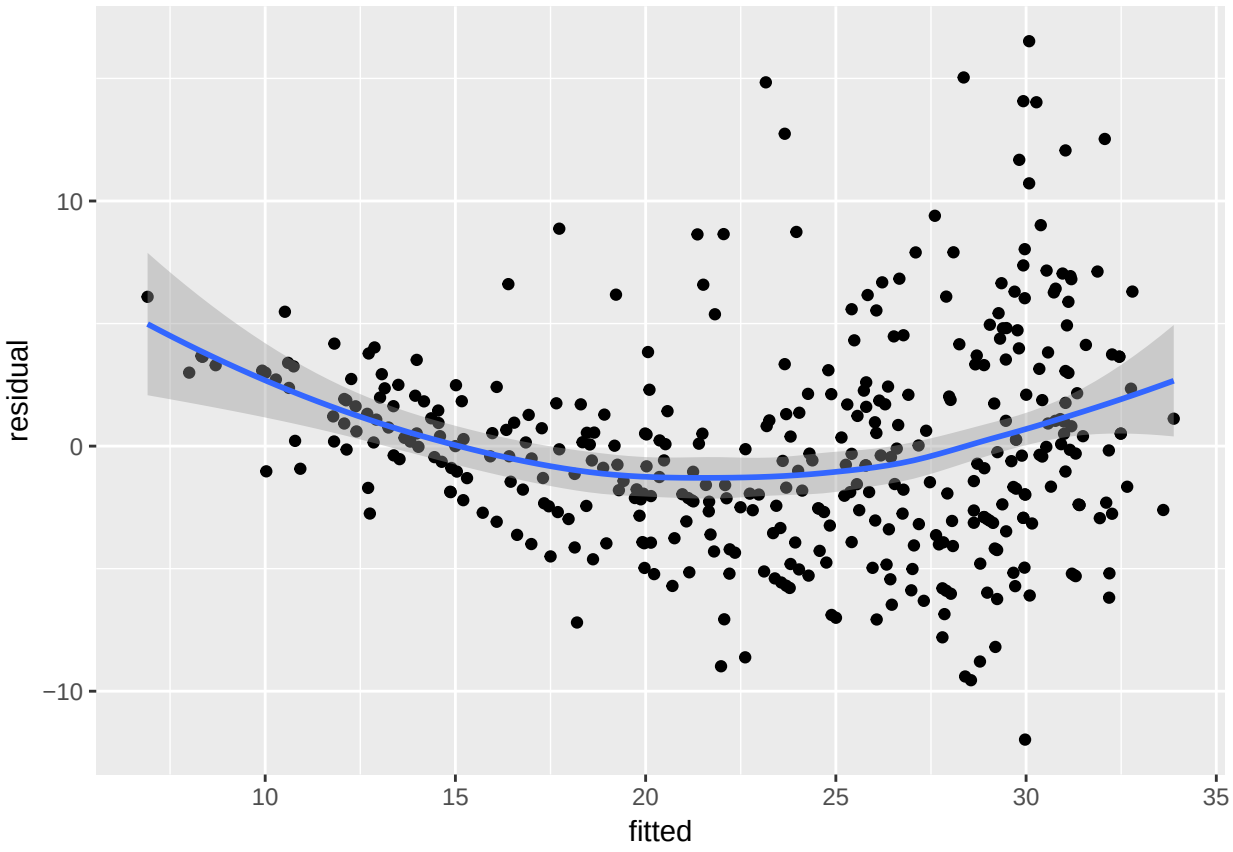
```
augmented_auto <- auto_fit %>%
  augment()
augmented_auto %>% head()
```

```
##   .rownames mpg weight  .fitted   .se.fit    .resid        .hat   .sigma
## 1         1  18   3504 19.42024 0.2575448 -1.420236 0.003533343 4.337678
## 2         2  15   3693 17.97489 0.2862653 -2.974889 0.004365337 4.335643
## 3         3  18   3436 19.94026 0.2487426 -1.940256 0.003295950 4.337158
## 4         4  16   3433 19.96320 0.2483756 -3.963198 0.003286232 4.333606
## 5         5  17   3449 19.84084 0.2503543 -2.840840 0.003338799 4.335878
## 6         6  15   4341 13.01941 0.4142337  1.980589 0.009140525 4.337104
##        .cooksd .std.resid
## 1 0.0001911753 -0.3283745
## 2 0.0010380292 -0.6881147
## 3 0.0003326721 -0.4485553
## 4 0.0013838821 -0.9162219
## 5 0.0007225022 -0.6567698
## 6 0.0009727164  0.4592282
```

With that we can make the plot we need to check for possible non-linearity

```
augmented_auto %>%
  ggplot(aes(x=.fitted,y=.resid)) +
    geom_point() +
    geom_smooth() +
    labs(x="fitted", y="residual")
```

```
## `geom_smooth()` using method = 'loess'
```



10.4.2 Correlated Error

For our inferences to be valid, we need residuals to be independent and identically distributed. We can spot non independence if we observe a trend in residuals as a function of the predictor X. Here is a simulation to demonstrate this:

In this case, our standard error estimates would be underestimated and our confidence intervals and hypothesis testing results would be biased.

10.4.3 Non-constant variance

Another violation of the iid assumption would be observed if the spread of residuals is not independent of the fitted values. Here is an illustration, and a possible fix using a log transformation on the outcome Y.

10.5 Multivariate Regression

Now that we've seen regression using a single predictor we'll move on to regression using multiple predictors. In this case, we use models of conditional expectation represented as linear functions of multiple variables.

In the case of our advertising example, this would be a model:

$$\texttt{sales} = \beta_0 + \beta_1 \times \texttt{TV} + \beta_2 \times \texttt{newspaper} + \beta_3 \times \texttt{facebook}$$

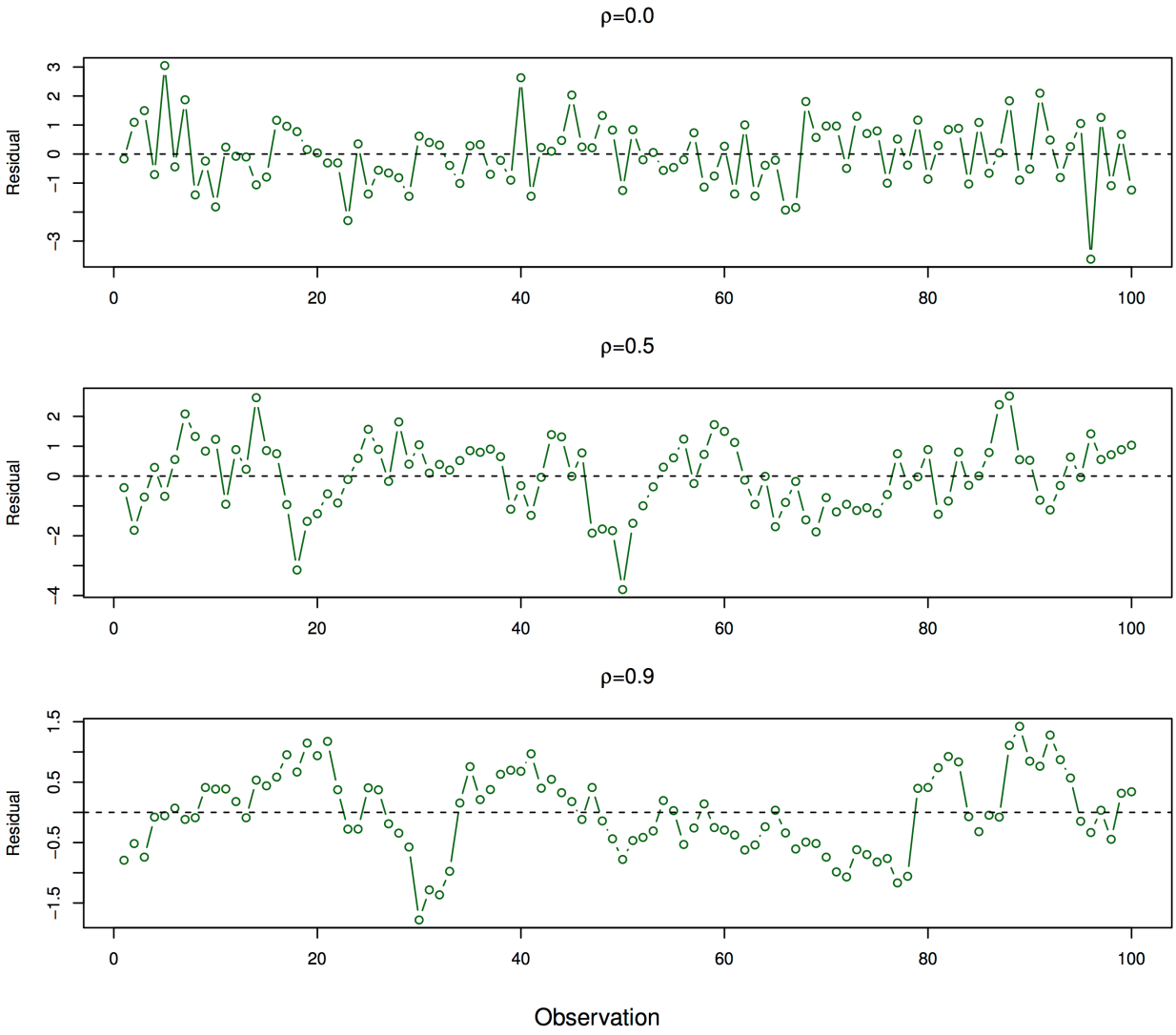

ρ=0.0
Residual
−3
−1
0
1
2
3
0
20
40
60
80
100
ρ=0.5
Residual
−4
−2
0
1
2
0
20
40
60
80
100
ρ=0.9
Residual
−1.5
−0.5
0.5
1.5
0
20
40
60
80
100
Observation

Figure 10.3:

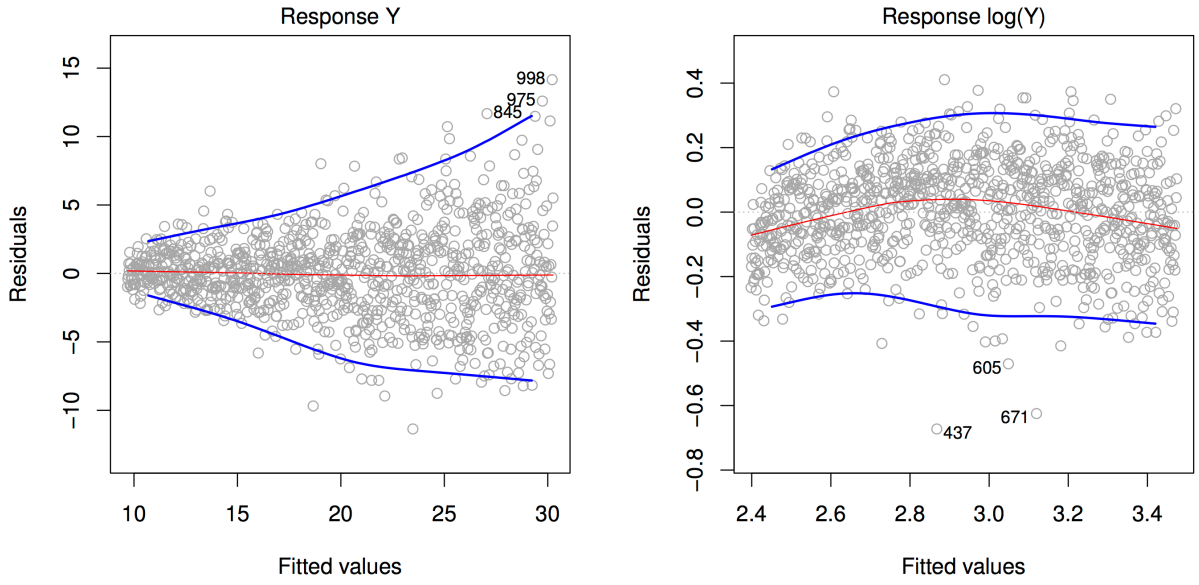


Figure 10.4:

These models let us make statements of the type: "holding everything else constant, sales increased on average by 1000 per dollar spent on Facebook advertising" (this would be given by parameter β_3 in the example model).

10.5.1 Estimation in multivariate regression

Continuing with our Auto example, we can build a model for miles per gallon using multiple predictors:

```
auto_fit <- lm(mpg~1+weight+cylinders+horsepower+displacement+year, data=Auto)
auto_fit
```

```
##
## Call:
## lm(formula = mpg ~ 1 + weight + cylinders + horsepower + displacement +
##     year, data = Auto)
##
## Coefficients:
##  (Intercept)        weight     cylinders    horsepower  displacement
##   -12.779493     -0.006524     -0.343690     -0.007715      0.006996
##         year
##     0.749924
```

From this model we can make the statement: "Holding everything else constant, cars run 0.76 miles per gallon more each year on average".

10.5.2 Statistical statements (cont'd)

Like simple linear regression, we can construct confidence intervals, and test a null hypothesis of no relationship ($\beta_j = 0$) for the parameter corresponding to each predictor. This is again nicely managed by the `broom` package:

```
auto_fit_stats <- auto_fit %>%
  tidy()
auto_fit_stats %>% knitr::kable()
```

term	estimate	std.error	statistic	p.value
(Intercept)	-12.7794934	4.2739387	-2.9900975	0.0029676
weight	-0.0065245	0.0005866	-11.1215621	0.0000000
cylinders	-0.3436900	0.3315619	-1.0365786	0.3005812
horsepower	-0.0077149	0.0107036	-0.7207702	0.4714872
displacement	0.0069964	0.0073095	0.9571736	0.3390787
year	0.7499243	0.0524361	14.3016700	0.0000000

In this case we would reject the null hypothesis of no relationship only for predictors `weight` and `year`. We would write the statement for year as follows:

"Holding everything else constant, cars run ${}_{0.65}0.75_{0.85}$ miles per gallon more each year on average (P-value$< 1e-16$)".

10.5.3 The F-test

We can make additional statements for multivariate regression: "is there a relationship between *any* of the predictors and the response?". Mathematically, we write this as $\beta_1 = \beta_2 = \cdots = \beta_p = 0$.

Under the null, our model for y would be estimated by the sample mean $\overline{y}$, and the error for that estimate is by total sum of squared error TSS. As before, we can compare this to the residual sum of squared error RSS using the F statistic:

$$\frac{(\text{TSS} - \text{RSS})/p}{\text{RSS}/(n-p-1)}$$

If this statistic is greater (enough) than 1, then we reject hypothesis that there is no relationship between response and predictors.

Back to our example, we use the `glance` function to compute this type of summary:

```
auto_fit %>%
  glance() %>%
  select(r.squared, sigma, statistic, df, p.value) %>%
  knitr::kable()
```

r.squared	sigma	statistic	df	p.value
0.8089093	3.433902	326.7965	6	0

In comparison with the linear model only using `weight`, this multivariate model explains *more of the variance* of `mpg`, but using more predictors. This is where the notion of *degrees of freedom* comes in: we now have a model with expanded *representational* ability.

However, the bigger the model, we are conditioning more and more, and intuitively, given a fixed dataset, have fewer data points to estimate conditional expectation for each value of the predictors. That means, that are estimated conditional expectation is less *precise*.

To capture this phenomenon, we want statistics that tradeoff how well the model fits the data, and the "complexity" of the model. Now, we can look at the full output of the `glance` function:

```
auto_fit %>%
  glance() %>%
  knitr::kable()
```

r.squared	adj.r.squared	sigma	statistic	p.value	df	logLik	AIC	BIC	deviance	df.residual
0.8089093	0.806434	3.433902	326.7965	0	6	-1036.81	2087.62	2115.419	4551.589	386

Columns `AIC` and `BIC` display statistics that penalize model fit with model size. The smaller this value, the better. Let's now compare a model only using `weight`, a model only using `weight` and `year` and the full multiple regression model we saw before.

```
lm(mpg~weight, data=Auto) %>%
  glance() %>%
  knitr::kable()
```

r.squared	adj.r.squared	sigma	statistic	p.value	df	logLik	AIC	BIC	deviance	df.residual
0.6926304	0.6918423	4.332712	878.8309	0	2	-1129.969	2265.939	2277.852	7321.234	390

```
lm(mpg~weight+year, data=Auto) %>%
  glance() %>%
  knitr::kable()
```

r.squared	adj.r.squared	sigma	statistic	p.value	df	logLik	AIC	BIC	deviance	df.residual
0.8081803	0.8071941	3.427153	819.473	0	3	-1037.556	2083.113	2098.998	4568.952	389

In this case, using more predictors beyond `weight` and `year` doesn't help.

10.5.4 Categorical predictors (cont'd)

We saw transformations for categorical predictors with only two values, and deferred our discussion of categorical predictors with more than two values. In our example we have the `origin` predictor, corresponding to where the car was manufactured, which has multiple values

```
Auto <- Auto %>%
  mutate(origin=factor(origin))
levels(Auto$origin)
```

```
## [1] "1" "2" "3"
```

As before, we can only use numerical predictors in linear regression models. The most common way of doing this is to create new dummy predictors to *encode* the value of the categorical predictor. Let's take a categorical variable `major` that can take values `CS`, `MATH`, `BUS`. We can encode these values using variables x_1 and x_2

$$x_1 = \begin{cases} 1 \text{ if MATH} \\ 0 \text{ o.w.} \end{cases}$$

$$x_2 = \begin{cases} 1 \text{ if BUS} \\ 0 \text{ o.w.} \end{cases}$$

Now let's build a model to capture the relationship between `salary` and `major`:

$$\texttt{salary} = \beta_0 + \beta_1 x_1 + \beta_2 x_2$$

What is the expected salary for a CS major? β_0.
For a MATH major? $\beta_0 + \beta_1$. For a BUS major? $\beta_0 + \beta_2$.

So, β_1 is the average difference in salary between MATH and CS majors. How can we calculate the average difference in salary between MATH and BUS majors? $\beta_1 - \beta_2$.

The `lm` function in R does this transformation by default when a variable has class `factor`. We can see what the underlying numerical predictors look like by using the `model_matrix` function and passing it the model formula we build:

```
extended_df <- model.matrix(~factor(origin), data=Auto) %>%
  as.data.frame() %>%
  mutate(origin = factor(Auto$origin))
```

```
extended_df %>%
  filter(origin == "1") %>% head()
```

```
##   (Intercept) factor(origin)2 factor(origin)3 origin
## 1           1               0               0      1
## 2           1               0               0      1
## 3           1               0               0      1
## 4           1               0               0      1
## 5           1               0               0      1
## 6           1               0               0      1
```

```
extended_df %>%
  filter(origin == "2") %>% head()
```

```
##   (Intercept) factor(origin)2 factor(origin)3 origin
## 1           1               1               0      2
## 2           1               1               0      2
## 3           1               1               0      2
## 4           1               1               0      2
## 5           1               1               0      2
## 6           1               1               0      2
```

```
extended_df %>%
  filter(origin == "3") %>% head()
```

```
##   (Intercept) factor(origin)2 factor(origin)3 origin
## 1           1               0               1      3
## 2           1               0               1      3
## 3           1               0               1      3
## 4           1               0               1      3
## 5           1               0               1      3
## 6           1               0               1      3
```

10.6 Interactions in linear models

The linear models so far include *additive* terms for a single predictor. That let us made statemnts of the type "holding everything else constant…". But what if we think that a pair of predictors *together* have a relationship with the outcome. We can add these *interaction* terms to our linear models as products:

Consider the advertising example:

$$\texttt{sales} = \beta_0 + \beta_1 \times \texttt{TV} + \beta_2 \times \texttt{facebook} + \beta_3 \times (\texttt{TV} \times \texttt{facebook})$$

If β_3 is positive, then the effect of increasing TV advertising money is increased if facebook advertising is also increased.

When using categorical variables, interactions have an elegant interpretation. Consider our car example, and suppose we build a model with an interaction between `weight` and `origin`. Let's look at what the numerical predictors look like:

```
Auto$origin <- factor(Auto$origin)
extended_df <- model.matrix(~weight+origin+weight:origin, data=Auto) %>%
as.data.frame() %>%
mutate(origin = factor(Auto$origin))

extended_df %>%
filter(origin == "1") %>% head()
```

```
##   (Intercept) weight origin2 origin3 weight:origin2 weight:origin3 origin
## 1           1   3504       0       0              0              0      1
## 2           1   3693       0       0              0              0      1
## 3           1   3436       0       0              0              0      1
## 4           1   3433       0       0              0              0      1
## 5           1   3449       0       0              0              0      1
## 6           1   4341       0       0              0              0      1
```

```
extended_df %>%
filter(origin == "2") %>% head()
```

```
##   (Intercept) weight origin2 origin3 weight:origin2 weight:origin3 origin
## 1           1   1835       1       0           1835              0      2
## 2           1   2672       1       0           2672              0      2
## 3           1   2430       1       0           2430              0      2
## 4           1   2375       1       0           2375              0      2
## 5           1   2234       1       0           2234              0      2
## 6           1   2123       1       0           2123              0      2
```

```
extended_df %>%
filter(origin == "3") %>% head()
```

```
##   (Intercept) weight origin2 origin3 weight:origin2 weight:origin3 origin
## 1           1   2372       0       1              0           2372      3
## 2           1   2130       0       1              0           2130      3
## 3           1   2130       0       1              0           2130      3
## 4           1   2228       0       1              0           2228      3
## 5           1   1773       0       1              0           1773      3
## 6           1   1613       0       1              0           1613      3
```

So what is the expected miles per gallon for a car with `origin == 1` as a function of weight?

$$\texttt{mpg} = \beta_0 + \beta_1 \times \texttt{weight}$$

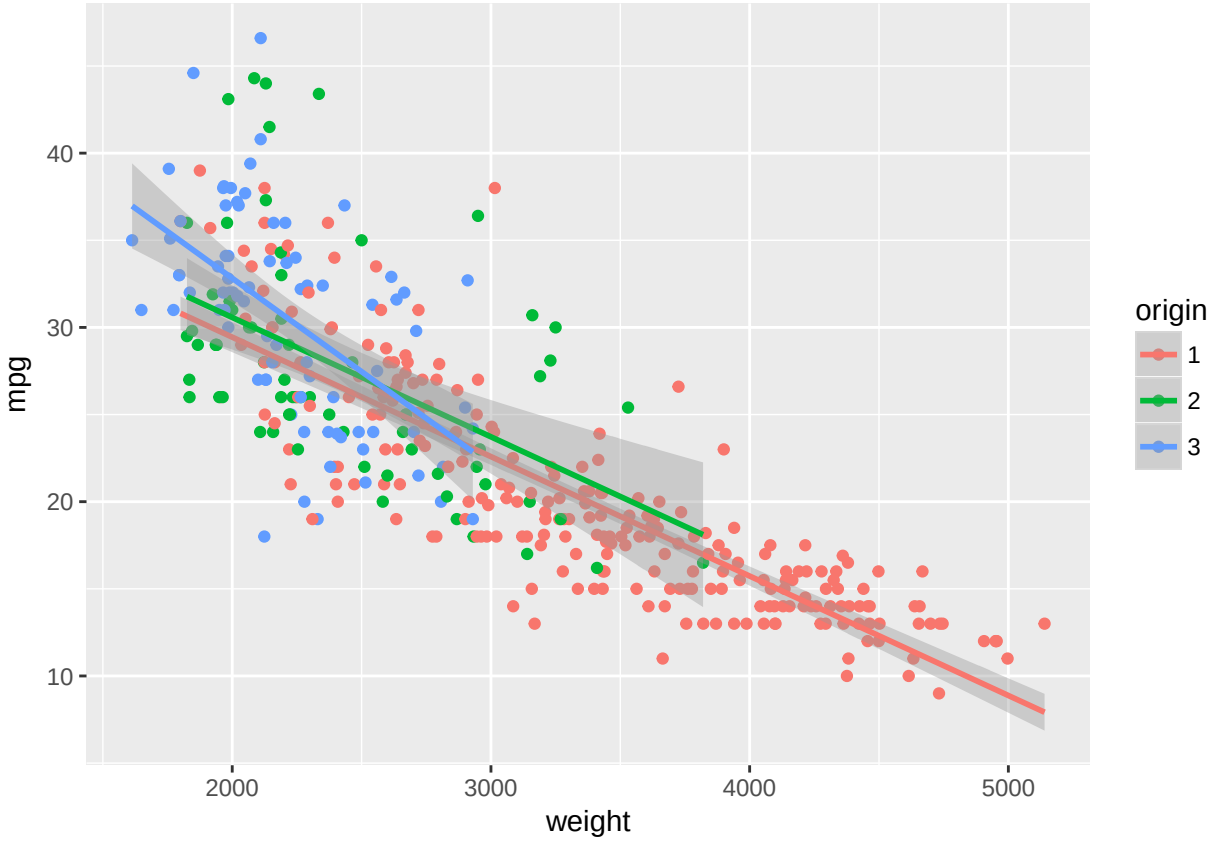
Now how about a car with `origin == 2`?

$$\texttt{mpg} = \beta_0 + \beta_1 \times \texttt{weight} + \beta_2 + \beta_4 \times \texttt{weight}$$

Now think of the graphical representation of these lines. For `origin == 1` the intercept of the regression line is β_0 and its slope is β_1. For `origin == 2` the intercept of the regression line is $\beta_0 + \beta_2$ and its slope is $\beta_1 + \beta_4$.

`ggplot` does this when we map a factor variable to a aesthetic, say color, and use the `geom_smooth` method:

```
Auto %>%
ggplot(aes(x=weight, y=mpg, color=origin)) +
geom_point() +
geom_smooth(method=lm)
```



The intercept of the three lines seem to be different, but the slope of `origin == 3` looks different (decreases faster) than the slopes of `origin == 1` and `origin == 2` that look very similar to each other.

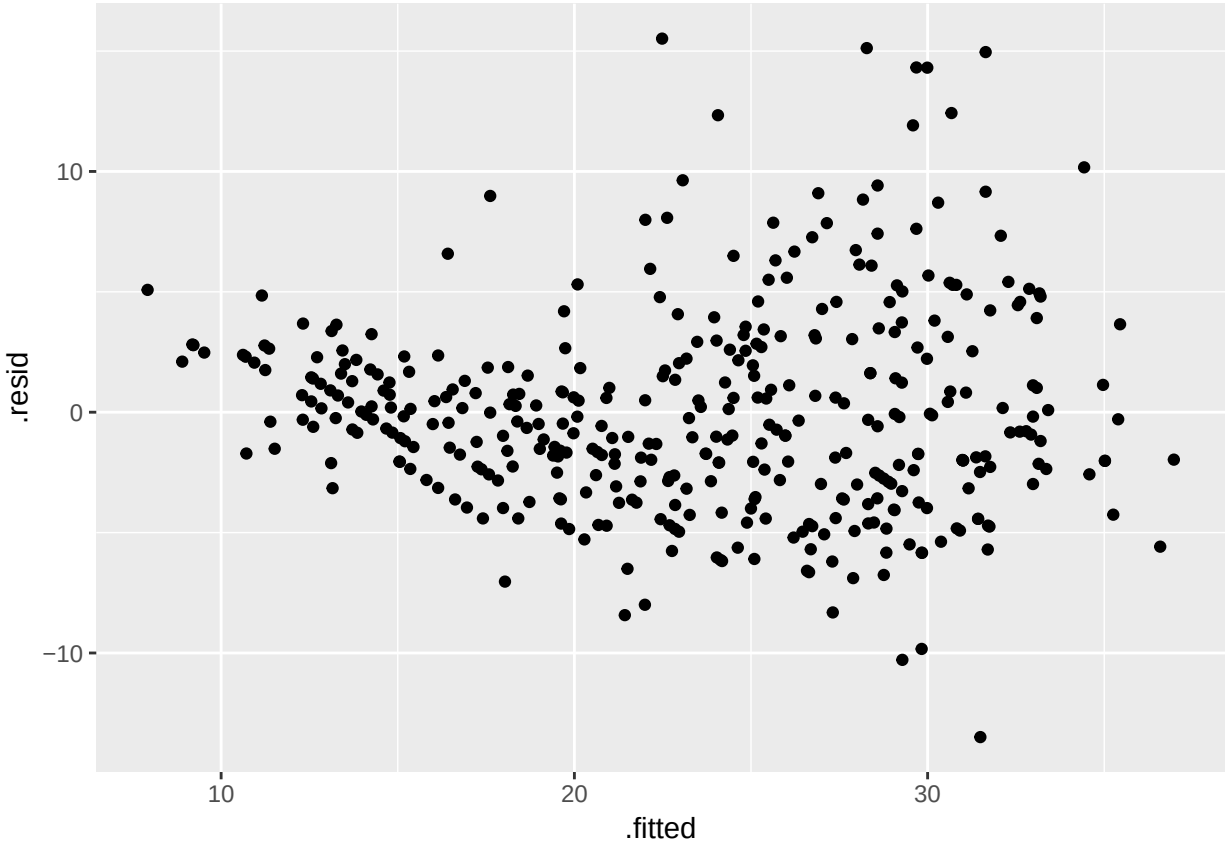
Let's fit the model and see how much statistical confidence we can give to those observations:

```
auto_fit <- lm(mpg~weight*origin, data=Auto)
auto_fit_stats <- auto_fit %>%
  tidy()
auto_fit_stats %>% knitr::kable()
```

term	estimate	std.error	statistic	p.value
(Intercept)	43.1484685	1.1861118	36.3780794	0.0000000
weight	-0.0068540	0.0003423	-20.0204971	0.0000000
origin2	1.1247469	2.8780381	0.3908033	0.6961582
origin3	11.1116815	3.5743225	3.1087518	0.0020181
weight:origin2	0.0000036	0.0011106	0.0032191	0.9974332
weight:origin3	-0.0038651	0.0015411	-2.5079723	0.0125521

So we can say that for `origin == 3` the relationship between `mpg` and `weight` is different but not for the other two values of `origin`. Now, there is still an issue here because this could be the result of a poor fit from a linear model, it seems none of these lines do a very good job of modeling the data we have. We can again check this for this model:

```
auto_fit %>%
  augment() %>%
  ggplot(aes(x=.fitted, y=.resid)) +
  geom_point()
```



The fact that residuals are not centered around zero suggests that a linear fit does not work well in this case.

10.7 Additional issues with linear regression

We saw previously some issues with linear regression that we should take into account when using this method for modeling. Multiple linear regression introduces an additional issue that is extremely important to consider when interpreting the results of these analyses: collinearity.

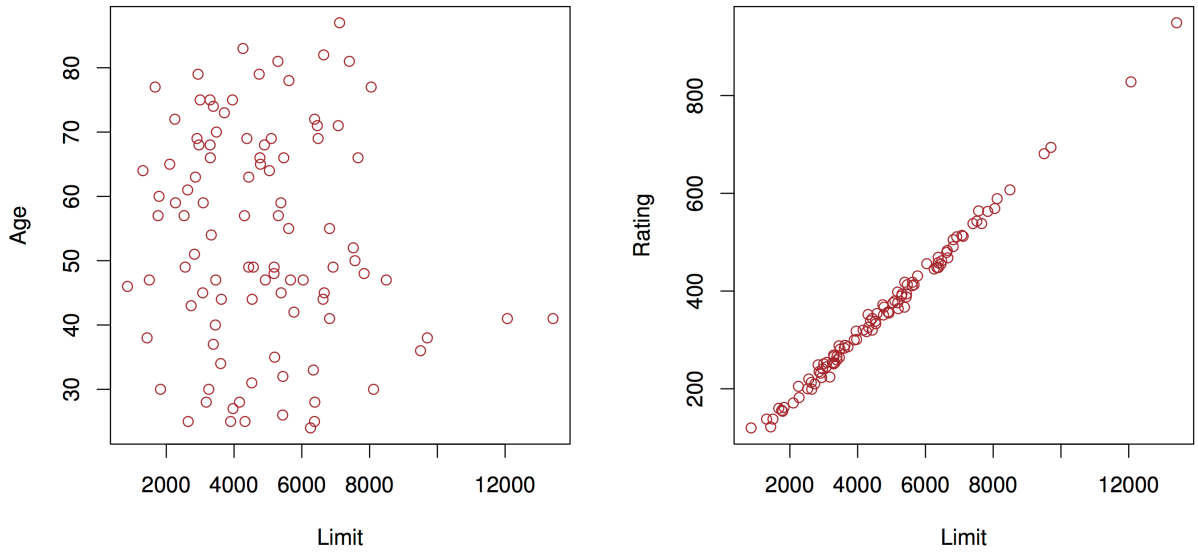


Figure 10.5:

In this example, you have two predictors that are very closely related. In that case, the set of β's that minimize RSS may not be unique, and therefore our interpretation is invalid. You can identify this potential problem by regressing predictors onto each other. The usual solution is to fit models only including one of the colinear variables.

10.8 Exercise

Here you will practice and experiment with linear regression using data from gapminder.org. I recommend spending a little time looking at material there, it is quite an informative site.

We will use a subset of data provided by gapminder provided by Jennifer Bryan described in it's github page.

The following commands load the dataset

```
library(gapminder)
data(gapminder)
```

For this exercise you will explore how life expectancy has changed over 50 years across the world, and how economic measures like gross domestic product (GDP) are related to it.

Exercise 1: *Make a scatter plot of life expectancy across time.*

Question 1: *Is there a general trend (e.g., increasing or decreasing) for life expectancy across time? Is this trend linear? (answering this qualitatively from the plot, you will do a statistical analysis of this question shortly)*

A slightly different way of making the same plot is looking at the distribution of life expectancy across countries as it changes over time:

```
library(dplyr)
library(ggplot2)

gapminder %>%
  ggplot(aes(x=factor(year), y=lifeExp)) +
    geom_violin() +
    labs(title="Life expectancy over time",
         x = "year",
         y = "life expectancy")
```

This type of plot is called a *violin plot*, and it displays the distribution of the variable in the y-axis for each value of the variable in the x-axis.

Question 2: *How would you describe the distribution of life expectancy across countries for individual years? Is it skewed, or not? Unimodal or not? Symmetric around it's center?*

Based on this plot, consider the following questions.

Question 3: *Suppose I fit a linear regression model of life expectancy vs. year (treating it as a continuous variable), and test for a relationship between year and life expectancy, will you reject the null hypothesis of no relationship? (do this without fitting the model yet. I am testing your intuition.)*

Question 4: *What would a violin plot of residuals from the linear model in Question 3 vs. year look like? (Again, don't do the analysis yet, answer this intuitively)*

Question 5: *According to the assumptions of the linear regression model, what **should** that violin plot look like?*

Exercise 2: *Fit a linear regression model using the `lm` function for life expectancy vs. year (as a continuous variable). Use the `broom::tidy` to look at the resulting model.*

Question 6: *On average, by how much does life expectancy increase every year around the world?*

Question 7: *Do you reject the null hypothesis of no relationship between year and life expectancy? Why?*

Exercise 3: *Make a violin plot of residuals vs. year for the linear model from Exercise 2 (use the `broom::augment` function).*

Question 8: *Does the plot of Excersize 3 match your expectations (as you answered Question 4)?*

Exercise 4: *Make a boxplot (or violin plot) of model residuals vs. continent.*

Question 9: *Is there a dependence between model residual and continent? If so, what would that suggest when performing a regression analysis of life expectancy across time?*

Exercise 5: *Use `geom_smooth(method=lm)` in ggplot as part of a scatter plot of life expectancy vs. year, grouped by continent (e.g., using the `color` aesthetic mapping).*

Question 10: *Based on this plot, should your regression model include an interaction term for continent **and** year? Why?*

Exercise 6: *Fit a linear regression model for life expectancy including a term for an interaction between continent and year. Use the `broom::tidy` function to show the resulting model.*

Question 11: *Are all parameters in the model significantly different from zero? If not, which are not significantly different from zero?*

Question 12: *On average, by how much does life expectancy increase each year for each continent? (Provide code to answer this question by extracting relevant estimates from model fit)*

Exercise 7: *Use the `anova` function to perform an F-test that compares how well two models fit your data: (a) the linear regression models from Exercise 2 (only including year as a covariate) and (b) Exercise 6 (including interaction between year and continent).*

Question 13: *Is the interaction model significantly better than the year-only model? Why?*

Exercise 8: *Make a residuals vs. year violin plot for the interaction model. Comment on how well it matches assumptions of the linear regression model. Do the same for a residuals vs. fitted values model.* (You should use the `broom::augment` function).

Chapter 11

Text analysis

In this section we will learn about text analysis using R and the tidy data framework. We will be using the `tidytext` package, and material from the excellent textbook "Text Mining with R": http://tidytextmining.com/tidytext.html. Specifically, we will work through the first two chapters of the book "The tidy text format" and "Sentiment analysis with tidy data". You can use the notes from the link above.

Chapter 12

Capstone 2 Project

- Use the `kmeans` function to cluster movies by rating and domestic gross revenue. Use the `broom` package to tidy the result of using `kmeans` and incorprate this into your analysis script
- Use ggplot2 to make the domestic gross vs. rating scatter plot. Use color to encode the result of the kmeans algorithm.
- Annotate the plot with movie titles
- Add a regression analysis of revenue vs. rating to address the question "do movies with high ratings tend to have bigger revenues"?

Chapter 13

Publishing Analyses with R

Rstudio has created an impressive eco-system for publishing with R. It has concentrated around two systems: Rmarkdown for creating documents that include both text and data analysis code, publishable in multiple formats, and shiny, a framework for creating interactive html applications that allow users to explore data analyses.

13.1 RMarkdown

Rstudio has provided substantial tutorials and documentation for Rmarkdown: http://rmarkdown.rstudio.com/

We will go over parts of the tutorial included there: http://rmarkdown.rstudio.com/lesson-1.html

and create an HTML report of the movie analysis we have been working on.

13.2 Shiny

Like RMarkdown, Rstudio provides great tutorials and documentation for shiny: https://shiny.rstudio.com/

We will go over parts of the shiny tutorial: https://shiny.rstudio.com/tutorial/lesson1/

We will create an application for our movie analysis as an example.

Chapter 14

Teaching with R

The publishing and sharing capabilities provided by R and Rstudio are outstanding resources for teaching with and about R.

14.1 Course materials

Using Rmarkdown as the main authoring tool for course materials has several advantages:

- Text discussing analyses, including plots and results, exists in the same document containing code to perform analysis.
- Analyses are reproducible since documents are executable
- Students can interact with teaching materials by downloading and re-executing or modifying analysis code
- Can create lecture notes and slides, reusing code and text
- Can use bookdown https://bookdown.org/ to create lecture note collection publishable in different formats
- Placing materials in github https://github.com makes it possible for students to download and modify materials. Can also incorporate feedback and modifications from others.

14.2 Assignments

- Students can submit Rmarkdown (or rendered Rmarkdown) for assignments to include both code used in analysis and text discussing analyses.
- Assignments can be handed out in Rmarkdown asking both conceptual and analysis questions. Students can write text (with mathematical notation if necessary) for the former, and R code for the latter.

14.3 Free and open source

- R and Rstudio are free and open source and remain so after students are done with courses.

- Many commercial products are free to use for student use, but must be purchased after students are done.
- This can be limiting for many students, especially those working in subject areas where purchasing expensive software is not a priority

14.4 Wide range of activities

- Besides allowing students to interact with code, using shiny as a teaching tool can be very powerful.
- Students often find creating interactive applications rewarding
- Wide range of analysis tools, covering many subject areas, or covering many technologies, allow students to carry out activities they find specifically motivating and interesting.

Chapter 15

Day 3 Capstone Project

Complete and publish a shiny application for our movie analysis.