CHESTER ISMAY AND ALBERT Y. KIM

A MODERN DIVE INTO THE LANGUAGE OF SCIENCE

Contents

1
Prerequisites

This book was written using the **bookdown** R package from Yihui Xie. In order to follow along and run the code in this book on your own, you'll need to have access to R (and preferably RStudio). You can find more information on both of these with a simple Google search for "R" and for "RStudio" and in Appendix A - Chapter 7.

We will keep a running list of R packages you will need to have installed to complete the analysis as well here in the `needed_pkgs` character vector. You can check if you have all of the needed packages installed by running all of the lines below. The last lines including the `if` will install them as needed (i.e., download their needed files from the internet to your hard drive).

You can run the `library` function on them to load them into your current analysis. Prior to each analysis where a package is needed, you will see the corresponding `library` function in the text.

```
needed_pkgs <- c("nycflights13", "dplyr", "ggplot2", "knitr",
  "devtools", "ggplot2", "webshot")
new.pkgs <- needed_pkgs[!(needed_pkgs %in% installed.packages())]

if (length(new.pkgs)) {
  install.packages(new.pkgs, repos = "http://cran.rstudio.com")
}
# Check that phantomjs is installed
if(is.null(webshot:::find_phantom()))
  webshot::install_phantomjs()
```

1.1 Colophon

The source of the book is available at and was built with versions of R packages given here:

```
devtools::session_info(needed_pkgs)
```

```
## Session info -------------------------------------------------------------

##  setting  value
```

```
##  version  R version 3.3.0 (2016-05-03)
##  system   x86_64, darwin13.4.0
##  ui       X11
##  language (EN)
##  collate  en_US.UTF-8
##  tz       America/Chicago
##  date     2016-07-31

## Packages -----------------------------------------------------------------

##  package      * version  date       source
##  assertthat     0.1      2013-12-06 CRAN (R 3.3.0)
##  BH             1.60.0-2 2016-05-07 CRAN (R 3.3.0)
##  colorspace     1.2-6    2015-03-11 CRAN (R 3.3.0)
##  curl           0.9.7    2016-04-10 CRAN (R 3.3.0)
##  DBI            0.4-1    2016-05-08 CRAN (R 3.3.0)
##  devtools       1.12.0   2016-06-24 CRAN (R 3.3.0)
##  dichromat      2.0-0    2013-01-24 CRAN (R 3.3.0)
##  digest         0.6.9    2016-01-08 CRAN (R 3.3.0)
##  dplyr        * 0.5.0    2016-06-24 CRAN (R 3.3.0)
##  evaluate       0.9      2016-04-29 CRAN (R 3.3.0)
##  formatR        1.4      2016-05-09 CRAN (R 3.3.0)
##  ggplot2      * 2.1.0    2016-03-01 CRAN (R 3.3.0)
##  git2r          0.15.0   2016-05-11 CRAN (R 3.3.0)
##  gtable         0.2.0    2016-02-26 CRAN (R 3.3.0)
##  highr          0.6      2016-05-09 CRAN (R 3.3.0)
##  httr           1.2.1    2016-07-03 CRAN (R 3.3.0)
##  jsonlite       1.0      2016-07-01 CRAN (R 3.3.0)
##  knitr        * 1.13     2016-05-09 CRAN (R 3.3.0)
##  labeling       0.3      2014-08-23 CRAN (R 3.3.0)
##  lazyeval       0.2.0    2016-06-12 CRAN (R 3.3.0)
##  magrittr       1.5      2014-11-22 CRAN (R 3.3.0)
##  markdown       0.7.7    2015-04-22 CRAN (R 3.3.0)
##  MASS           7.3-45   2016-04-21 CRAN (R 3.3.0)
##  memoise        1.0.0    2016-01-29 CRAN (R 3.3.0)
##  mime           0.5      2016-07-07 CRAN (R 3.3.0)
##  munsell        0.4.3    2016-02-13 CRAN (R 3.3.0)
##  nycflights13 * 0.2.0    2016-04-30 CRAN (R 3.3.0)
##  openssl        0.9.4    2016-05-25 CRAN (R 3.3.0)
##  plyr           1.8.4    2016-06-08 CRAN (R 3.3.0)
##  R6             2.1.2    2016-01-26 CRAN (R 3.3.0)
##  RColorBrewer   1.1-2    2014-12-07 CRAN (R 3.3.0)
##  Rcpp           0.12.6   2016-07-19 CRAN (R 3.3.0)
##  reshape2       1.4.1    2014-12-06 CRAN (R 3.3.0)
##  rstudioapi     0.6      2016-06-27 CRAN (R 3.3.0)
```

```
##  scales         0.4.0    2016-02-26 CRAN (R 3.3.0)
##  stringi        1.1.1    2016-05-27 CRAN (R 3.3.0)
##  stringr        1.0.0    2015-04-30 CRAN (R 3.3.0)
##  tibble         1.1      2016-07-04 CRAN (R 3.3.0)
##  webshot        0.3.2    2016-06-23 CRAN (R 3.3.0)
##  whisker        0.3-2    2013-04-28 CRAN (R 3.3.0)
##  withr          1.0.2    2016-06-20 CRAN (R 3.3.0)
##  yaml           2.1.13   2014-06-12 CRAN (R 3.3.0)
```

2
Introduction

2.1 Preamble

- Coggle Diagrams
- Lean on visualizations as much as possible first to introduce summary measures
- We will focus on the triad: computational, data, and inferential thinking.
  - Literate programming / literate data science
  - "Programs must be written for people to read, and only incidentally for machines to execute." - Hal Abelson
- Discuss the role of data analysis in the sciences
- Explain why programming and data science help scientific knowledge grow

2.2 Data/science pipeline

2.3 Reproducibility

2.4 Who is this book for? (Target)

Students taking a traditional intro stats class in a small college environment using RStudio preferably RStudio Server.

We assume no prerequisites: no calculus and no prior programming experience.

2.5 Algorithmic Thinking

Despite what you may think, computers are stupid. You need to explicitly tell it everything it needs to do; if you make even a slight mistake, it will cry.

To think about further

You can label chapter and section titles using `{#label}` after them, e.g., we can reference Chapter 2. If you do not manually label them, there will be automatic labels anyway, e.g., Chapter **??**.

Figures and tables with captions will be placed in `figure` and `table` environments, respectively.

```
par(mar = c(4, 4, .1, .1))
plot(pressure, type = 'b', pch = 19)
```

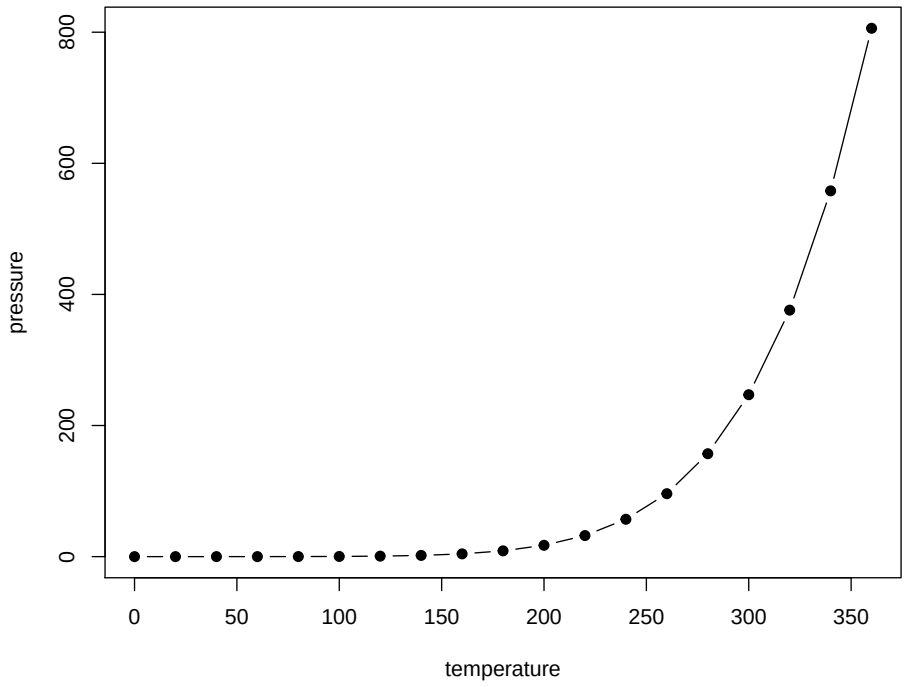


Figure 2.1: Here is a nice figure!

Reference a figure by its code chunk label with the `fig:` prefix, e.g., see Figure 2.1. Similarly, you can reference tables generated from `knitr::kable()`, e.g., see Table 2.1.

```
knitr::kable(
  head(iris, 20), caption = 'Here is a nice table!',
  booktabs = TRUE
)
```

Table 2.1: Here is a nice table!

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa
4.6	3.4	1.4	0.3	setosa
5.0	3.4	1.5	0.2	setosa
4.4	2.9	1.4	0.2	setosa
4.9	3.1	1.5	0.1	setosa
5.4	3.7	1.5	0.2	setosa
4.8	3.4	1.6	0.2	setosa
4.8	3.0	1.4	0.1	setosa
4.3	3.0	1.1	0.1	setosa
5.8	4.0	1.2	0.2	setosa
5.7	4.4	1.5	0.4	setosa
5.4	3.9	1.3	0.4	setosa
5.1	3.5	1.4	0.3	setosa
5.7	3.8	1.7	0.3	setosa
5.1	3.8	1.5	0.3	setosa

3
Tidy data

Need to give big picture question here and set up how this chapter ties in to chapters to come.

Force dataframes into tibbles using `as_tibble`? That way `head` isn't needed since printing will be nice by default.

You have surely heard the word "tidy" in your life:

- "Tidy up your room!"
- "Please write your homework in a tidy way so that it is easier to grade and to provide feedback."
- Marie Kondo's best-selling book *The Life-Changing Magic of Tidying Up: The Japanese Art of Decluttering and Organizing*
- "I am not by any stretch of the imagination a tidy person, and the piles of unread books on the coffee table and by my bed have a plaintive, pleading quality to me - 'Read me, please!' " - Linda Grant

So what does it mean for your data to be **tidy**? Put simply: it means that your data is organized. But it's more than just that. It means that your data follows the same standard format making it easy for others to find elements of your data, to manipulate and transform your data, and for our purposes continuing with the common theme: it makes it easier to visualize your data and the relationships between different variables in your data.

3.1 What is tidy data?

We will follow Hadley Wickham's definition of **tidy data** here (Wickham, 2014):

> A dataset is a collection of values, usually either numbers (if quantitative) or strings (if qualitative). Values are organised in two ways. Every value belongs to a variable and an observation. A variable contains all values that measure the same underlying attribute (like height, temperature, duration) across units. An observation contains all values measured on the same unit (like a person, or a day, or a race) across attributes.
>
> Tidy data is a standard way of mapping the meaning of a dataset to its structure. A dataset is messy or tidy depending on how rows, columns and tables are matched up with observations, variables and types. In **tidy data**:

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

Reading over this definition, you can begin to think about datasets that won't follow this nice format.

Learning check

(LC3.1) Give an example dataset that doesn't follow this format.

- What features of this dataset might make it difficult to visualize?
- How could the dataset be tweaked to make it **tidy**?

3.2 The `nycflights13` datasets

We likely have all flown on airplanes or know someone that has. Air travel has become an ever-present aspect of our daily lives. If you live in or are visiting a relatively large city and you walk around that city's airport, you see gates showing flight information from many different airlines. And you will frequently see that some flights are delayed because of a variety of conditions. Are there ways that we can avoid having to deal with these flight delays?

We'd all like to arrive at our destinations on time whenever possible. (Unless you secretly love hanging out at airports. If you are one of these people, pretend for the moment that you are very much anticipating being at your final destination.) Hadley Wickham (herein just referred to as "Hadley") created multiple datasets containing information about departing flights from the New York City area in 2013 (Wickham, 2016). We will begin by loading in one of these datasets, the `flights` dataset, and getting an idea of its structure:

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

The `library` function here loads the R package `nycflights13` into the current R environment in which you are working. (Note that you'll get an error if you try to load this package in and it hasn't been installed. Check Chapter 2 to make sure the package has been downloaded to your computer.) The next line of code `data(flights)` loads in the `flights` dataset that is stored in the `nycflights13` package.

This dataset and most others presented in this book will be in the `data.frame` format in R. dataframes are ways to look at collections of variables that are tightly coupled together. We next begin with a couple useful R functions to get a sense for what the `flights` dataset looks like:

```
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 <time>
```

Learning check

(LC3.2) What does the `head` function give us? Why might it be a useful function to run on a dataset you have been presented with?

(LC3.3) What do you think the `tail` function would give us for the `flights` dataset?

(LC3.4) What does any *ONE* row in this dataset refer to?

- A. Data on an airline
- B. Data on a flight
- C. Data on an airport
- D. Data on multiple flights

We see that the `head` function gives us the first six rows (by default) for this dataset. This can give us an idea of what to expect our dataset to look like. For example, we see the different **variables** listed in the columns and we see that there are different types of variables. Some of the variables like `distance`, `day`, and `arr_delay` are what we will call **quantitative** variables. These variables vary in a numerical way. Other variables here are **categorical**.

Note that if you look in the leftmost portion near the `##` of the R output, you will see a column of numbers. These are the row numbers of the dataset. If you glance across a row with the same number, say row 5, you can get an idea of what each row correspond to. In other words, this will allow you to identify what object is being referred to in a given row. This is often called the **observational unit**. The **observational unit** in this example is an individual flight departing New York City in 2013.

Note: Frequently the first thing you should do when given a dataset is to

- identify the observation unit,
- specify the variables, and
- give the types of variables you are presented with.

```
str(flights)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    336776 obs. of  19 variables:
##  $ year          : int  2013 2013 2013 2013 2013 2013 2013 2013 2013 2013 ...
##  $ month         : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ day           : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ dep_time      : int  517 533 542 544 554 554 555 557 557 558 ...
##  $ sched_dep_time: int  515 529 540 545 600 558 600 600 600 600 ...
##  $ dep_delay     : num  2 4 2 -1 -6 -4 -5 -3 -3 -2 ...
##  $ arr_time      : int  830 850 923 1004 812 740 913 709 838 753 ...
##  $ sched_arr_time: int  819 830 850 1022 837 728 854 723 846 745 ...
##  $ arr_delay     : num  11 20 33 -18 -25 12 19 -14 -8 8 ...
##  $ carrier       : chr  "UA" "UA" "AA" "B6" ...
##  $ flight        : int  1545 1714 1141 725 461 1696 507 5708 79 301 ...
##  $ tailnum       : chr  "N14228" "N24211" "N619AA" "N804JB" ...
##  $ origin        : chr  "EWR" "LGA" "JFK" "JFK" ...
##  $ dest          : chr  "IAH" "IAH" "MIA" "BQN" ...
##  $ air_time      : num  227 227 160 183 116 150 158 53 140 138 ...
##  $ distance      : num  1400 1416 1089 1576 762 ...
##  $ hour          : num  5 5 5 5 6 5 6 6 6 6 ...
##  $ minute        : num  15 29 40 45 0 58 0 0 0 0 ...
##  $ time_hour     : POSIXct, format: "2013-01-01 05:00:00" ...
```

Learning check

(LC3.5) What are some examples in this dataset of **categorical** variables? What makes them different than **quantitative** variables?

(LC3.6) What does `int`, `num`, and `chr` mean in the output above?

(LC3.7) How many different columns are in this dataset?

(LC3.8) How many different rows are in this dataset?

Another way to view the properties of a dataset is to use the `str` function ("str" is short for "structure"). This will give you the first few entries of each variable in a row after the variable. In addition, the type of the variable is given immediately after the `:` following each variable's name. Here, `int` and `num` refer to quantitative variables. In contrast, `chr` refers to categorical

variables. One more type of variable is given here with the `time_hour` variable: **POSIXct**. As you may suspect, this variable corresponds to a specific date and time of day.

Another nice feature of R is the help system. You can get help in R by simply entering a question mark before the name of a function or an object and you will be presented with a page showing the documentation. Note that this output help file is omitted here but can be accessed here on page 3 of the PDF document.

```
?flights
```

Another aspect of tidy data is a description of what each variable in the dataset represents. This helps others to understand what your variable names mean and what they correspond to. If we look at the output of `?flights`, we can see that a description of each variable by name is given.

An important feature to **ALWAYS** include with your data is the appropriate units of measurement. We'll see this further when we work with the `dep_delay` variable in Chapter 4. (It's in minutes, but you'd get some really strange interpretations if you thought it was in hours or seconds. UNITS MATTER!)

3.3 How is `flights` tidy?

We see that `flights` has a rectangular shape with each row corresponding to a different flight and each column corresponding to a characteristic of that flight. This matches exactly with how Hadley defined tidy data:

1. Each variable forms a column.
2. Each observation forms a row.

But what about the third property?

3. Each type of observational unit forms a table.

We identified earlier that the observational unit in the `flights` dataset is an individual flight. And we have shown that this dataset consists of 336776 flights with 19 variables. In other words, some rows of this dataset don't refer to a measurement on an airline or on an airport. They specifically refer to characteristics/measurements on a given **flight** from New York City in 2013.

By contrast, also included in the `nycflights13` package are datasets with different observational units (Wickham, 2016):

- `weather`: hourly meteorological data for each airport
- `planes`: construction information about each plane
- `airports`: airport names and locations
- `airlines`: translation between two letter carrier codes and names

You may have been asking yourself what `carrier` refers to in the `str(flights)` output above. The `airlines` dataset provides a description of this with each airline being the observational unit:

```
data(airlines)
airlines
```

```
## # A tibble: 16 x 2
##    carrier                        name
##      <chr>                       <chr>
## 1       9E           Endeavor Air Inc.
## 2       AA      American Airlines Inc.
## 3       AS        Alaska Airlines Inc.
## 4       B6             JetBlue Airways
## 5       DL        Delta Air Lines Inc.
## 6       EV    ExpressJet Airlines Inc.
## 7       F9      Frontier Airlines Inc.
## 8       FL AirTran Airways Corporation
## 9       HA      Hawaiian Airlines Inc.
## 10      MQ                  Envoy Air
## 11      OO       SkyWest Airlines Inc.
## 12      UA       United Air Lines Inc.
## 13      US             US Airways Inc.
## 14      VX              Virgin America
## 15      WN      Southwest Airlines Co.
## 16      YV          Mesa Airlines Inc.
```

```
library(knitr)
kable(airlines)
```

carrier	name
9E	Endeavor Air Inc.
AA	American Airlines Inc.
AS	Alaska Airlines Inc.
B6	JetBlue Airways
DL	Delta Air Lines Inc.
EV	ExpressJet Airlines Inc.
F9	Frontier Airlines Inc.
FL	AirTran Airways Corporation
HA	Hawaiian Airlines Inc.
MQ	Envoy Air
OO	SkyWest Airlines Inc.
UA	United Air Lines Inc.
US	US Airways Inc.
VX	Virgin America
WN	Southwest Airlines Co.
YV	Mesa Airlines Inc.

Note that R by default will print out the object when only its name is given as we have

done here with `airlines`. If we'd prefer to print a dataframe out in a clean format we can use the `kable` function in the `knitr` R package.

3.4 Normal forms of data

The datasets included in the `nycflights13` package are in a form that minimizes redundancy of data. We will see that there are ways to *merge* (or *join*) the different tables together easily. We are capable of doing so because each of the tables have *keys* in common to relate one to another. This is an important property of **normal forms** of data. The process of decomposing dataframes into less redundant tables without losing information is called **normalization**. More information is available on Wikipedia.

We saw an example of this above with the `airlines` dataset. While the `flights` dataframe could also include a column with the names of the airlines instead of the carrier code, this would be repetitive since there is a unique mapping of the carrier code to the name of the airline/carrier.

Below an example is given showing how to **join** the `airlines` dataframe together with the `flights` dataframe by linking together the two datasets via a common **key** of `"carrier"`. Note that this "joined" dataframe is assigned to a new dataframe called `joined_flights`. The structure of `joined_flights` is also given so that you can see a few elements of the new column `name` that has been added. (We will see in Chapter 5 ways to change `name` to a more descriptive variable name.)

```
library(dplyr)
joined_flights <- inner_join(x = flights, y = airlines, by = "carrier")
str(joined_flights)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    336776 obs. of  20 variables:
##  $ year          : int  2013 2013 2013 2013 2013 2013 2013 2013 2013 2013 ...
##  $ month         : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ day           : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ dep_time      : int  517 533 542 544 554 554 555 557 557 558 ...
##  $ sched_dep_time: int  515 529 540 545 600 558 600 600 600 600 ...
##  $ dep_delay     : num  2 4 2 -1 -6 -4 -5 -3 -3 -2 ...
##  $ arr_time      : int  830 850 923 1004 812 740 913 709 838 753 ...
##  $ sched_arr_time: int  819 830 850 1022 837 728 854 723 846 745 ...
##  $ arr_delay     : num  11 20 33 -18 -25 12 19 -14 -8 8 ...
##  $ carrier       : chr  "UA" "UA" "AA" "B6" ...
##  $ flight        : int  1545 1714 1141 725 461 1696 507 5708 79 301 ...
##  $ tailnum       : chr  "N14228" "N24211" "N619AA" "N804JB" ...
##  $ origin        : chr  "EWR" "LGA" "JFK" "JFK" ...
##  $ dest          : chr  "IAH" "IAH" "MIA" "BQN" ...
##  $ air_time      : num  227 227 160 183 116 150 158 53 140 138 ...
##  $ distance      : num  1400 1416 1089 1576 762 ...
##  $ hour          : num  5 5 5 5 6 5 6 6 6 6 ...
```

```
##  $ minute        : num  15 29 40 45 0 58 0 0 0 0 ...
##  $ time_hour     : POSIXct, format: "2013-01-01 05:00:00" ...
##  $ name          : chr  "United Air Lines Inc." "United Air Lines Inc." "American Airlines Inc." "JetBl
```

More discussion about joining dataframes together will be given in Chapter 5. We will see there that the names of the columns to be linked need not match as they did here with `"carrier"`.

Review questions

(RQ3.1) What are common characteristics of "tidy" datasets?

(RQ3.2) What makes "tidy" datasets useful for organizing data?

(RQ3.3) What would the code `kable(head(flights))` produce?

(RQ3.4) How many variables are presented in the table below? What does each row correspond to? (**Hint:** You may not be able to answer both of these questions immediately but take your best guess.)

students	faculty
4	2
6	3

(RQ3.5) The confusion you may have encountered in Question 4 is a common one those that work with data are commonly presented with. This dataset is not tidy. Actually, the dataset in Question 4 has three variables not the two that were presented. Make a guess as to what these variables are and present a tidy dataset instead of this untidy one given in Question 4.

(RQ3.6) The actual data presented in Question 4 is given below in tidy data format:

role	Sociology?	Type of School
student	TRUE	Public
student	TRUE	Public
student	TRUE	Public
student	TRUE	Public
student	FALSE	Public
student	FALSE	Public
student	FALSE	Private
student	FALSE	Private
student	FALSE	Private
student	FALSE	Private
faculty	TRUE	Public
faculty	TRUE	Public
faculty	FALSE	Public
faculty	FALSE	Private
faculty	FALSE	Private

- What does each row correspond to?
- What are the different variables in this dataframe?
- The `Sociology?` variable is known as a logical variable. What types of values does a logical variable take on?

(RQ3.7) What are some advantages of data in normal forms? What are some disadvantages?

3.5 What's to come?

In Chapter 4, we will further explore the distribution of a variable in a related dataset to `flights`: the `temp` variable in the `weather` dataset. We'll be interested in understanding how this variable varies in relation to the values of other variables in the dataset. We will see that visualization is often a powerful tool in helping us see what is going on in a dataset. It will be a useful way to expand on the `str` function we have seen here for tidy data.

Last updated:

```
## [1] "Sunday, July 31, 2016 17:21:41 CDT"
```

4
Visualizing Data

In Chapter 3, we discussed the importance of datasets being **tidy**. You will see in examples here why having a tidy dataset helps us immensely with plotting our data. We will focus on using Hadley's `ggplot2` package in doing so, which was developed to work specifically on datasets that are **tidy**. It provides an easy way to customize your plots and is based on data visualization theory given in *The Grammar of Graphics* (Wilkinson, 2005).

Graphics provide a nice way for us to get a sense for how quantitative variables compare in terms of their center and their spread. It also helps us to identify patterns and outliers in our data. We will see that a common extension of these ideas is to compare the distribution (i.e., what the spread of a variable looks like) as we go across the levels of a different categorical variable.

4.1 Five Named Graphs - The FNG

For our purposes here, we will be working with five different types of graphs. (Note that we will use a lot of different words here in regards to plotting - "graphs", "plots", and "charts" are all ways to discuss a resulting graphic. You can think of them as all being synonyms.) These five plots are:

- histograms
- boxplots
- barplots
- scatter-plots
- line-graphs

With this toolbox of plots, you can visualize just about any type of variable thrown at you. We will discuss some other variations of these but with the FNG in your repertoire you can do big things! Something we will also stress here is that certain plots only work for categorical/logical variables and others only for quantitative variables. You'll want to quiz yourself often on which plot makes sense with a given problem set-up.

We now introduce another dataframe in the `nycflights13` package introduced in Chapter 3.

```
library(nycflights13)
data(weather)
```

4.2 Histograms

Our focus now turns to the `temp` variable in this `weather` dataset. We would like to visualize what the 26130 temperatures look like. Looking over the `weather` dataset[1] and running `?weather`, we can see that the `temp` variable corresponds to hourly temperature (in Fahrenheit) recordings at weather stations near airports in New York City. We could just produce points where each of the different values appears on something similar to a number line:

[1] To view a dataset in spreadsheet format in RStudio, you can run the `View()` function with the dataframe as its argument.

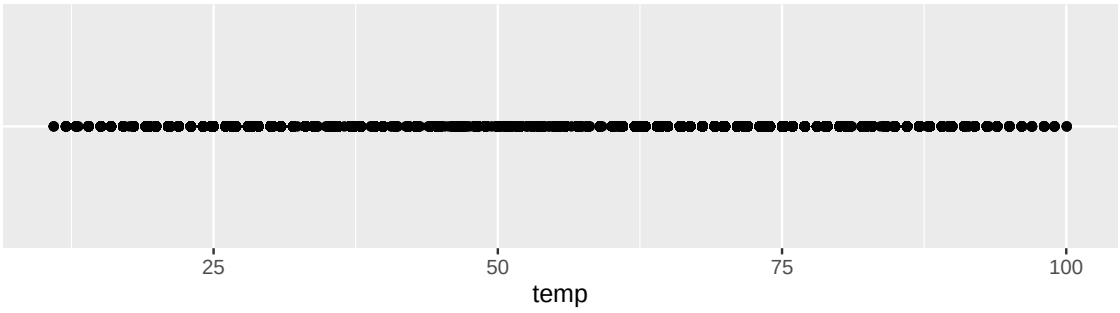


Figure 4.1: Strip Plot of Hourly Temperature Recordings from NYC in 2013

This gives us a general idea of how the values of `temp` differ. We see that temperatures vary from around 11 up to 100 degrees Fahrenheit. The area between 40 and 60 degrees appears to have more points plotted than outside that range.

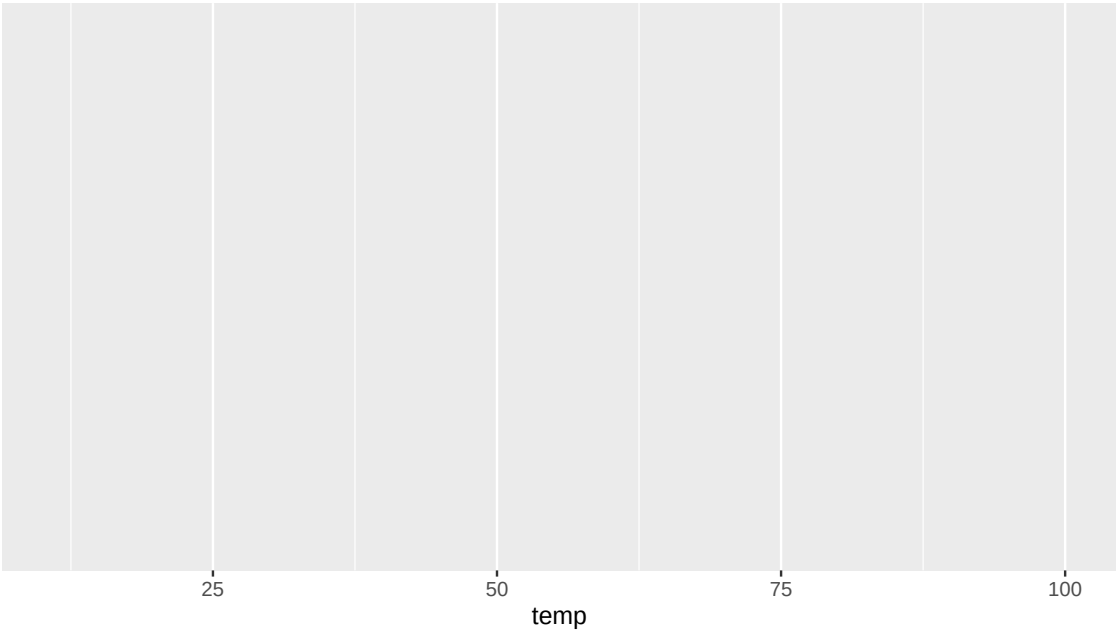
What is commonly produced instead of this strip plot is a plot known as a **histogram**. The **histogram** show how many elements of the variable fall in specified **bins**. These **bins** may correspond to between 0-10°F, 10-20°F, etc.

To produce a histogram, we introduce the Hadley's `ggplot2` package (Wickham and Chang, 2016). We will use the `ggplot` function which expects at a bare minimal as arguments

- the dataframe where the variables exist and
- the names of the variables to be plotted.

The names of the variables will be entered into the `aes` function as arguments where `aes` stands for "aesthetics".

```
ggplot(data = weather, mapping = aes(x = temp))
```



The plot given above is not a histogram, but the output does show us a bit of what is going on with `ggplot(data = weather, mapping = aes(x = temp))`. It is producing a backdrop onto which we will "paint" elements.

We next proceed by adding a layer—hence, the use of the `+` symbol—to the plot to produce a histogram. (Note also here that we don't have to specify the `data =` and `mapping =` text in our function calls. This is covered in more detail in Appendix A - Chapter 7.)

```
ggplot(data = weather, mapping = aes(x = temp)) +
  geom_histogram()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

```
## Warning: Removed 1 rows containing non-finite values (stat_bin).
```

We have the power to specify how many bins we would like to put the data into as an argument in the `geom_histogram` function. By default, this is chosen to be 30 somewhat arbitrarily and we have received a warning above our plot that this was done. We also notice here that another warning about 1 missing value is given. This value is omitted from the plot. This warning is ignored for future customizations of the plot. (**Discuss missing values here?**)

```
ggplot(data = weather, mapping = aes(x = temp)) +
  geom_histogram(bins = 60)
```

We can tweak the plot a little more by specifying the width of the bins (instead of how many bins to divide the variable into) by using the `binwidth` argument in the `geom_histogram` function. We can also add some color to the plot by invoking the `fill` and `color` arguments. A listing of all of the built-in colors to R by name and color is available here.

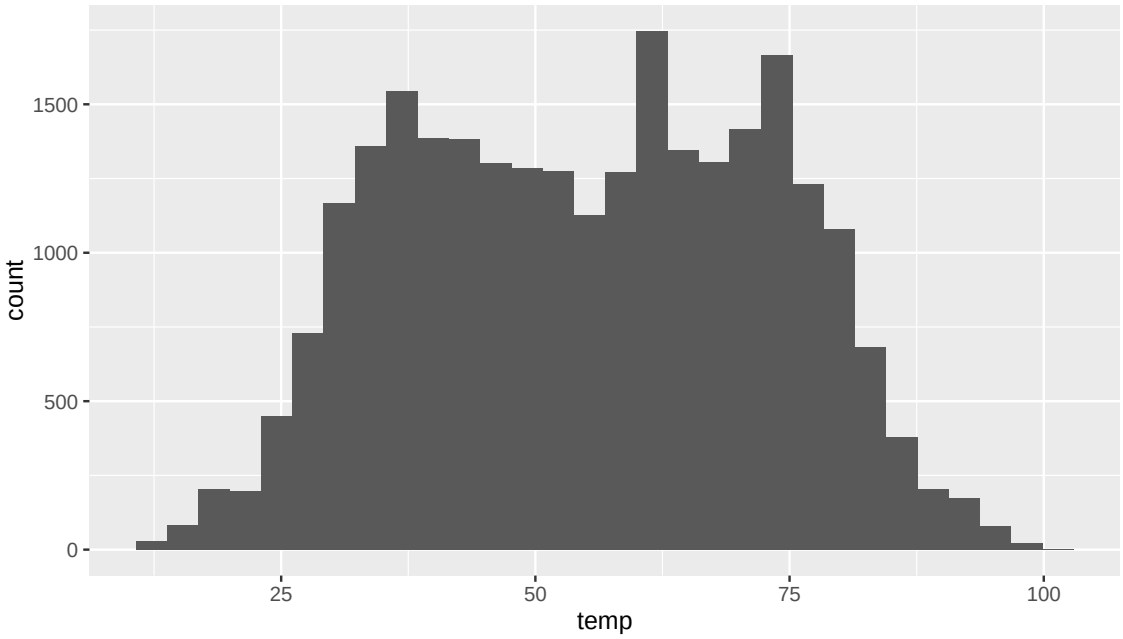


Figure 4.2: Histogram of Hourly Temperature Recordings from NYC in 2013

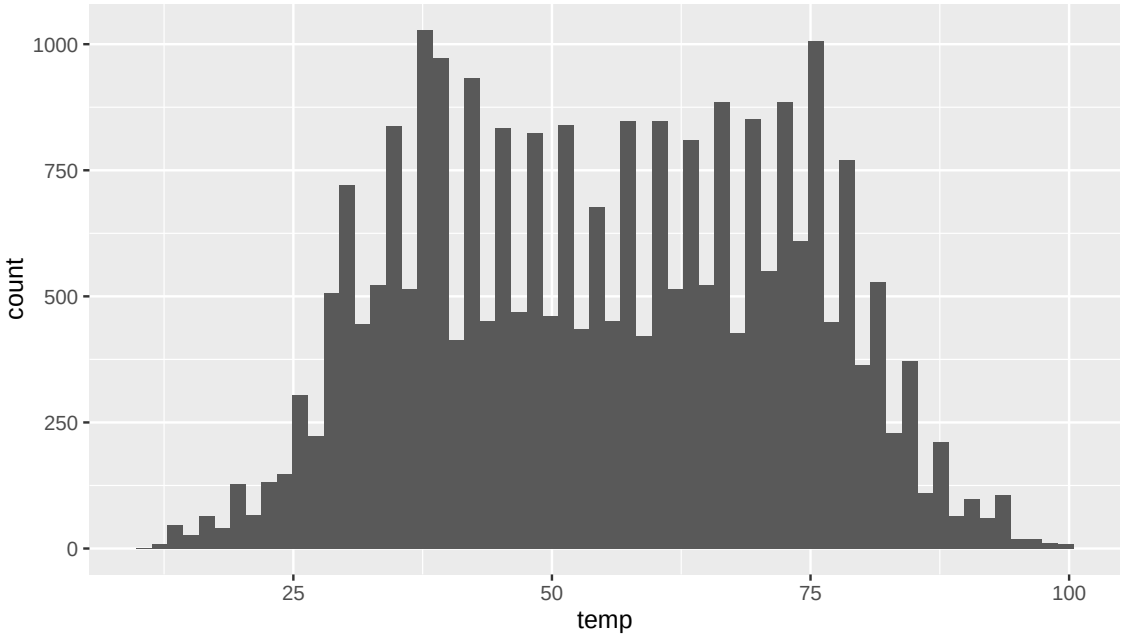


Figure 4.3: Histogram of Hourly Temperature Recordings from NYC in 2013 - 60 Bins

```
ggplot(data = weather, mapping = aes(x = temp)) +
  geom_histogram(binwidth = 10, color = "white", fill = "forestgreen")
```

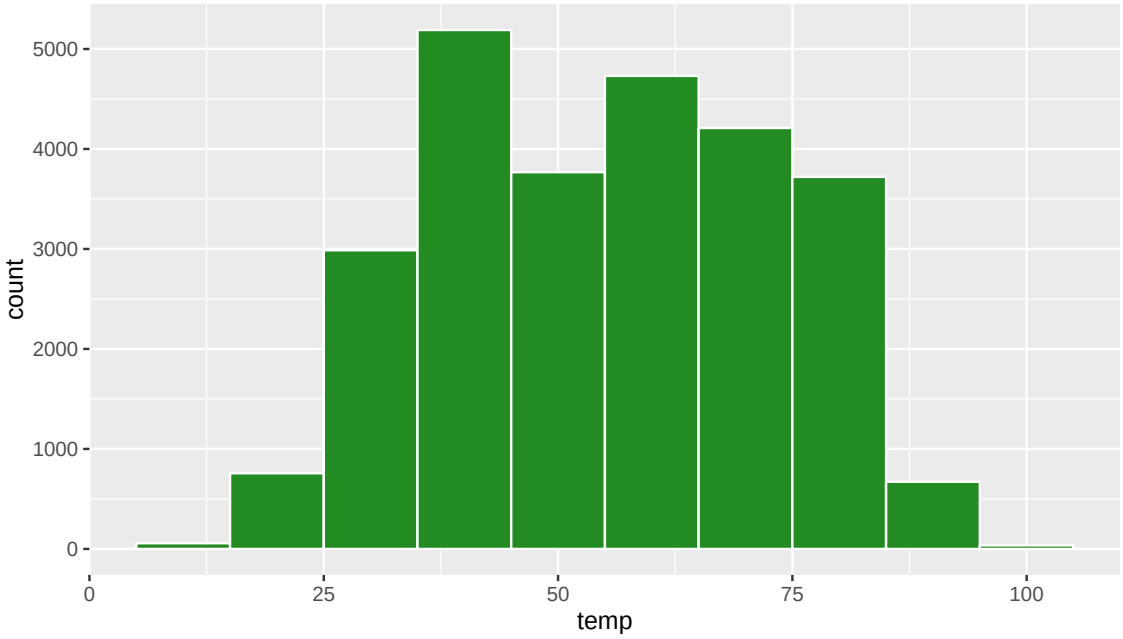


Figure 4.4: Histogram of Hourly Temperature Recordings from NYC in 2013 - Binwidth = 10

Learning check

(LC4.1) What does changing the number of bins from 30 to 60 tell us about the distribution of temperatures?

(LC4.2) Would you classify the distribution of temperatures as symmetric or skewed?

(LC4.3) What would you guess is the "center" value in this distribution? Why did you make that choice?

(LC4.4) Is this data spread out greatly from the center or is it close? Why?

4.2.1 Continuous data summaries

The `temp` variable is a **continuous** quantitative variable (frequently just called a **continuous variable**). "A variable is **continuous** if you can arrange its values in order and an infinite number of unique values can exist between any two values of the variable"(Wickham and Grolemund, 2016). Some common examples of continuous variables are time and height.

Between any two times there are an infinitely many number of time units that fall between them.

It is often easier to think about quantitative variables that are not continuous to help us better understand continuity. The best example is counts. If we are looking to count the number of flights that depart on a given day from New York City, this variable would not be continuous. It falls on a **discrete** scale.

We can examine some summary information about this `temp` variable. To do so, we introduce the `summary` function. (We will see in Chapter 5 how to use the `summarize` function in the `dplyr` package to produce similar results.)

The syntax here is a little different than what we have seen before. (A further discussion about R syntax is available in Appendix A - Chapter 7). Here, `summary` is the function and it is expecting an object to be summarized as its argument. The object here is the `temp` variable in the `weather` dataframe. To focus on just this one variable `temp` in `weather`, we separate them by the dollar sign symbol `$`. Order matters here: the dataframe comes before the `$` and the variable/column name comes after.

```
summary(weather$temp)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##   10.94   39.92   55.04   55.20   69.98  100.00       1
```

This tells us what is known as the **five-number summary** for our variable as well as the **mean** value of the variable. More information on both of these concepts is given in Appendix A - Chapter 7.

This `summary` gives us some numerical summaries of our temperature variable. The minimum recorded temperature is 10.94 degrees Fahrenheit and the maximum is 100.04 degrees Fahrenheit. We have one missing value denoted as an `NA` in the observations of this variable. The median Fahrenheit temperature of 55.04 and mean of 55.2035149 are quite close. This is a property of symmetric distributions.

The last two entries given by `summary` correspond to the 25^{th} percentile and the 75^{th} percentile. If we sorted all of the temperatures in increasing order, we would see that 25% of them would fall below 39.92 and that 75% of them would fall below 69.98. This implies that the middle 50% of data values lie between 39.92 and 69.98 degrees Fahrenheit.

Introduce standard deviation here?

4.2.2 Summary

Histograms provide a useful way of looking at how ONE continuous variable varies. They allow us to answer questions such as

- Are there values far away from the center? These are commonly called **outliers** and can frequently be easily identified on a histogram.
- Are most values close to the center? If so, the spread of the variable is small. If not, the spread is large.
- How spread out are the values? One measure of this spread is **standard deviation** discussed above.

The histogram show how many entries fall in different groupings of this variable. Another common property of distributions is symmetry and as we saw it is quite easily identified by looking over the histogram produced from the variable's values.

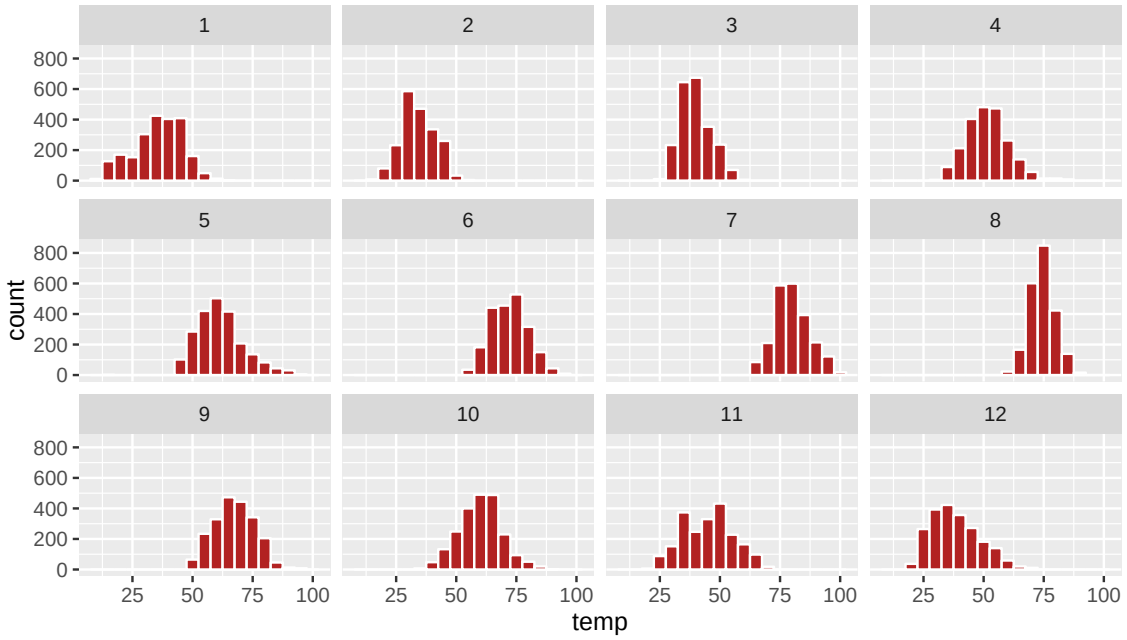
4.3 Boxplots

Histograms can also be produced to compare the distribution of a variable over another variable. Suppose we were interested in looking at how the temperature recordings we saw in the last section varied by month. This is what is meant by "the distribution of a variable over another variable": `temp` is one variable and `month` is the other variable.

4.3.1 Faceting

In order to look at histograms of `temp` for each month, we introduce a new concept called **facetting**. Faceting is used when we'd like to create small multiples of the same plot over a different categorical variable. By default, all of the small multiples will have the same vertical axis. An example will help here. We will discuss the concept of faceting in further detail in Section 4.4.

```
ggplot(data = weather, mapping = aes(x = temp)) +
  geom_histogram(binwidth = 5, color = "white", fill = "firebrick") +
  facet_wrap(~month)
```



As we might expect, the temperature tends to increase as summer approaches and then decrease as winter approaches.

Learning check

(LC4.5) What other things do you notice about the faceted plot above? How does a faceted plot help us see how relationships between two variables?

(LC4.6) What do the numbers 1-12 correspond to in the plot above? What about 25, 50, 75, 100?

(LC4.7) What could be done to make the faceted plot above more readable? (Focus on tweaking the histograms and not on making a different type of plot here.)

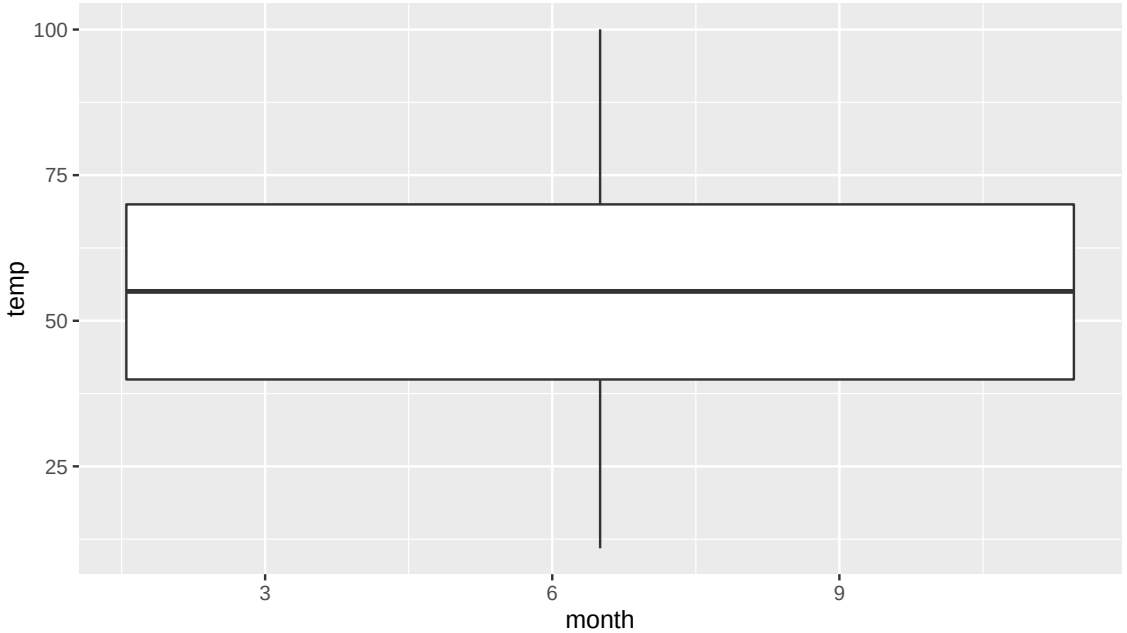
(LC4.8) For which types of datasets would these types of faceted plots not work well in comparing relationships between variables? Draw or give an example.

Histograms can provide a way to compare distributions across groups as we see above when we looked at temperature over months. Frequently, a plot called a **boxplot** (also called a **side-by-side boxplot**) is done instead. The **boxplot** uses the information provided in the **five-number summary** referred to in the previous section when we used the `summary` function. It gives a way to compare this summary information across the different levels of a group. Let's create a boxplot to compare the monthly temperatures as we did above with the faceted histograms.

```
ggplot(data = weather, mapping = aes(x = month, y = temp)) +
  geom_boxplot()
```

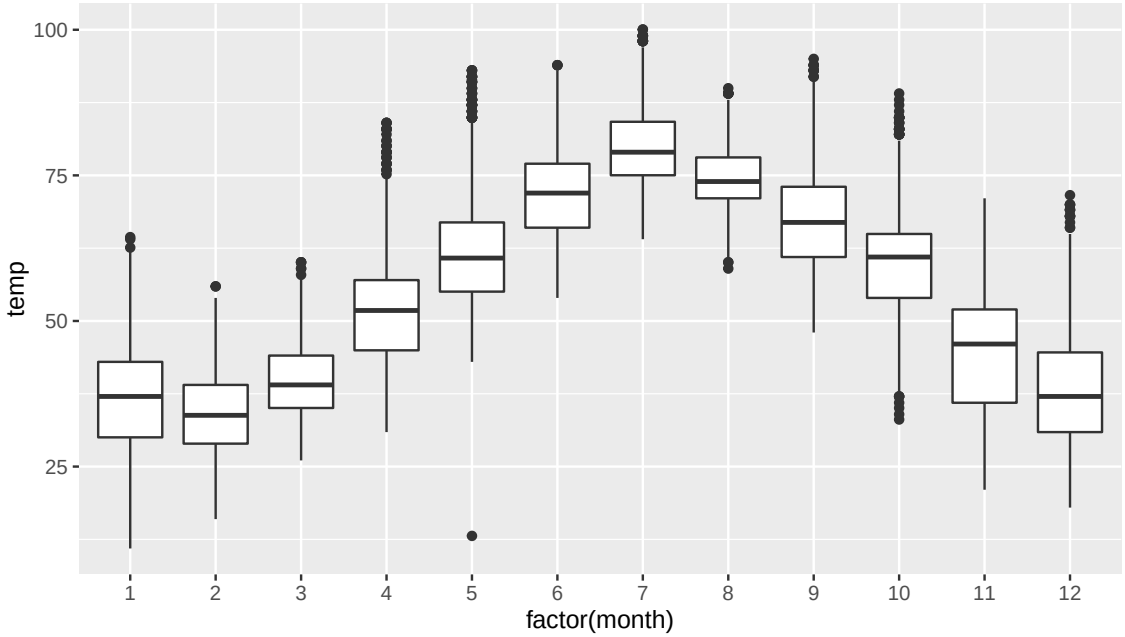
```
## Warning: Continuous x aesthetic -- did you forget aes(group=...)?
```

```
## Warning: Removed 1 rows containing non-finite values (stat_boxplot).
```



Note the first warning that is given here. (The second one corresponds to missing values in the dataframe and it is turned off on subsequent plots.) This plot does not look like what we were expecting. We were expecting to see the distribution of temperatures for each month (so 12 different boxplots). This gives us the overall boxplot without any other groupings. We can get around this by introducing a new function for our `x` variable.

```
ggplot(data = weather, mapping = aes(x = factor(month), y = temp)) +
  geom_boxplot()
```



We have introduced a new function called `factor()` here. One of the things this function does is to convert a numeric value like `month` (1, 2, ..., 12) into a categorical variable. The "box" part of this plot represents the 25^{th} percentile, the median (50^{th} percentile), and the 75^{th} percentile. The dots correspond to **outliers**. (The specific formulation for these outliers is discussed in Appendix A - Chapter 7.) The lines show how the data varies that is not in the center 50% defined by the first and third quantiles. Longer lines correspond to more variability and shorter lines correspond to less variability.

Learning check

(LC4.9) What does the dot at the bottom of the plot for May correspond to? Explain what might have occurred in May to produce this point.

(LC4.10) Which months have the highest variability in temperature? What reasons do you think this is?

(LC4.11) We looked at the distribution of a continuous variable over a categorical variable here with this boxplot. Why can't we look at the distribution of one continuous variable over the distribution of another continuous variable? Say temperature across pressure, for example?

(LC4.12) Boxplots provide a simple way to identify outliers. Why may outliers be easier to identify when looking at a boxplot instead of a faceted histogram?

4.3.2 Summary

Boxplots provide a way to compare and contrast the distribution of ONE quantitative variable across multiple levels of ONE categorical variable. One can easily look to see where the median falls across the different groups by looking at the center line in the box. You can also see how spread out the variable is across the different groups by looking at the width of the box and also how far out the lines stretch from the box. Lastly, outliers are even more easily identified when looking at a boxplot than when looking at a histogram.

4.4 Barplots

Both histograms and boxplots represent ways to visualize the variability of continuous variables. Another common task is to present the distribution of a categorical variable. This is a simpler task since we will be interested in how many elements from our data fall into the different categories of the categorical variable. We need not bin the data or identify the different quantiles for categorical variables.

Frequently, the best way to visualize these different counts (also known as **frequencies**) is via a barplot. Consider the distribution of airlines that flew out of New York City in 2013. This can be plotted by invoking the `geom_bar` function in `ggplot2`:

```
ggplot(data = flights, mapping = aes(x = carrier)) +
  geom_bar()
```

Recall the `airlines` dataset discussed in Chapter 3.

```
library(knitr)
data(airlines)
kable(airlines)
```

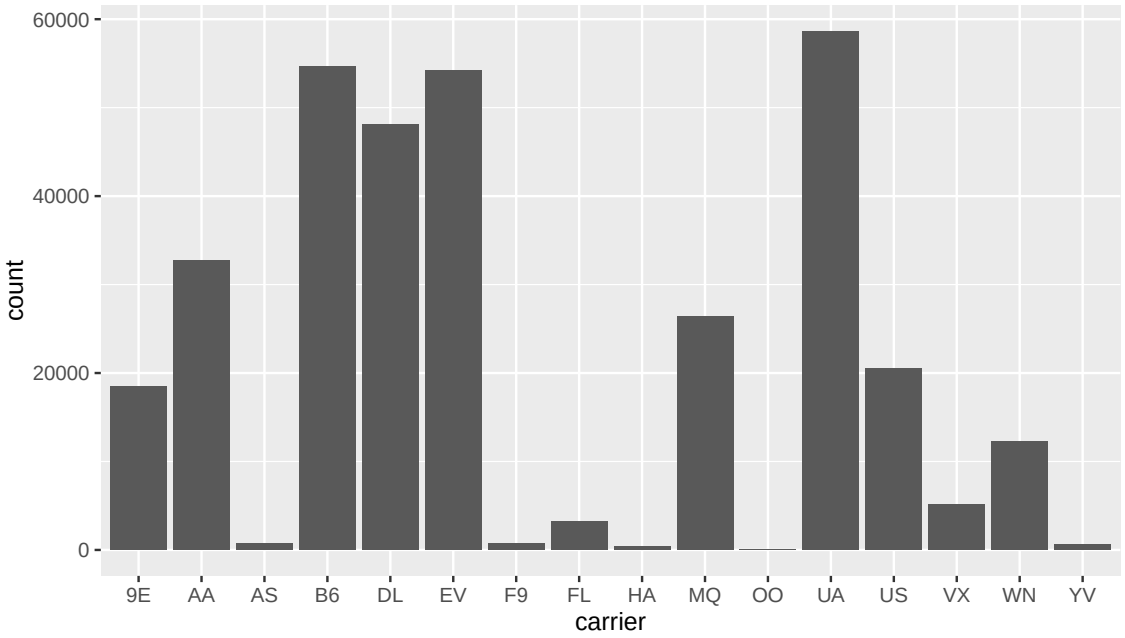


Figure 4.5: Number of flights departing NYC in 2013 by airline

carrier	name
9E	Endeavor Air Inc.
AA	American Airlines Inc.
AS	Alaska Airlines Inc.
B6	JetBlue Airways
DL	Delta Air Lines Inc.
EV	ExpressJet Airlines Inc.
F9	Frontier Airlines Inc.
FL	AirTran Airways Corporation
HA	Hawaiian Airlines Inc.
MQ	Envoy Air
OO	SkyWest Airlines Inc.
UA	United Air Lines Inc.
US	US Airways Inc.
VX	Virgin America
WN	Southwest Airlines Co.
YV	Mesa Airlines Inc.

We see that United Air Lines, JetBlue Airways, and ExpressJet Airlines had the most flights depart New York City in 2013. To get the actual number of flights by each airline we can use the `table` function on the `carrier` variable in `flights`:

```
flights_table <- table(flights$carrier)
flights_table
```

```
##
##    9E    AA    AS    B6    DL    EV    F9    FL    HA    MQ    OO    UA
## 18460 32729   714 54635 48110 54173   685  3260   342 26397    32 58665
##    US    VX    WN    YV
## 20536  5162 12275   601
```

More information on the use of this `$` syntax is available in Chapter 3 and in Appendix A - Chapter 7.

Learning check

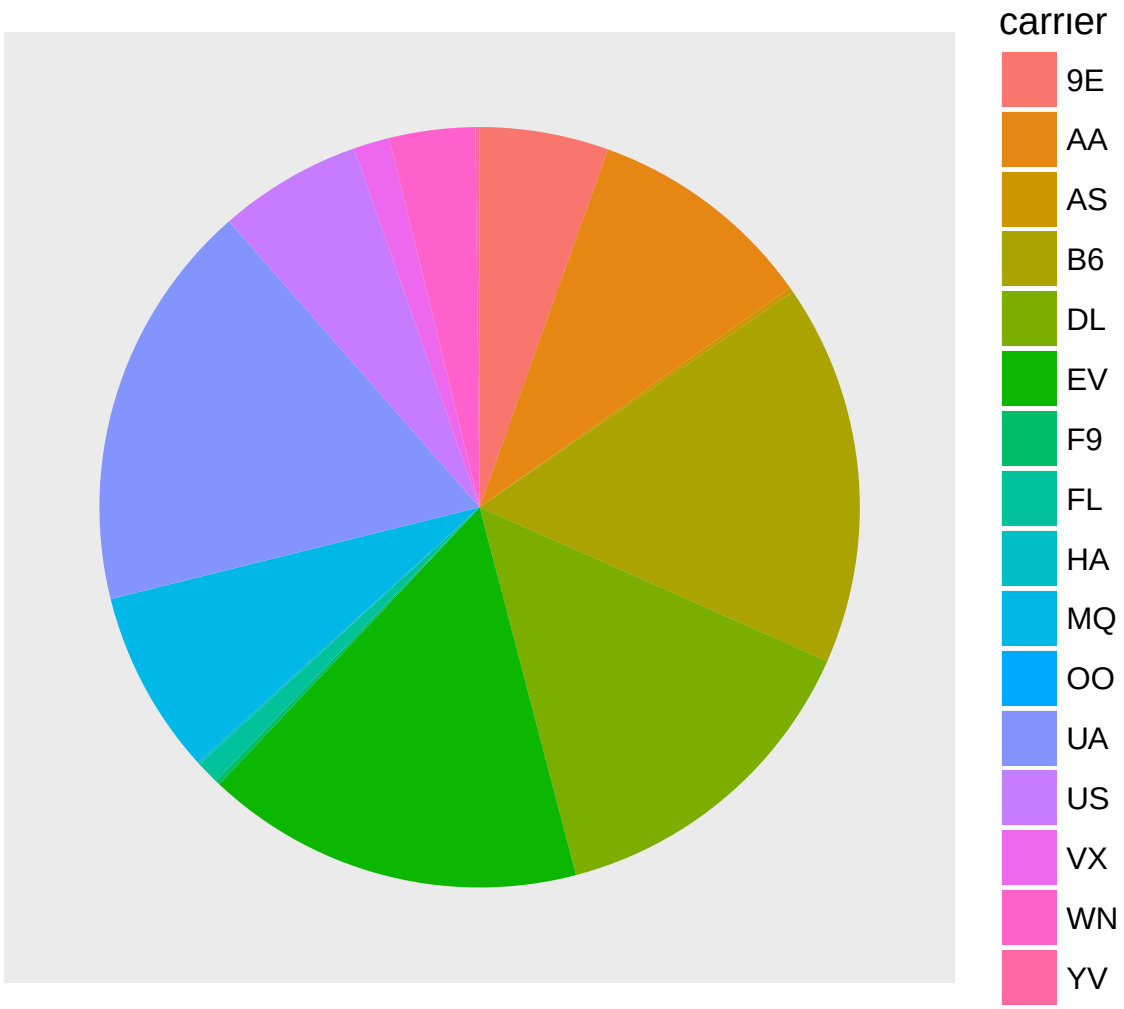
(LC4.13) Why are histograms inappropriate for visualizing categorical variables?
(LC4.14) What is the difference between histograms and barplots?
(LC4.15) How many Envoy Air flights departed NYC in 2013?
(LC4.16) What was the seventh highest airline in terms of departed flights from NYC in 2013?

4.4.1 Must avoid pie charts!

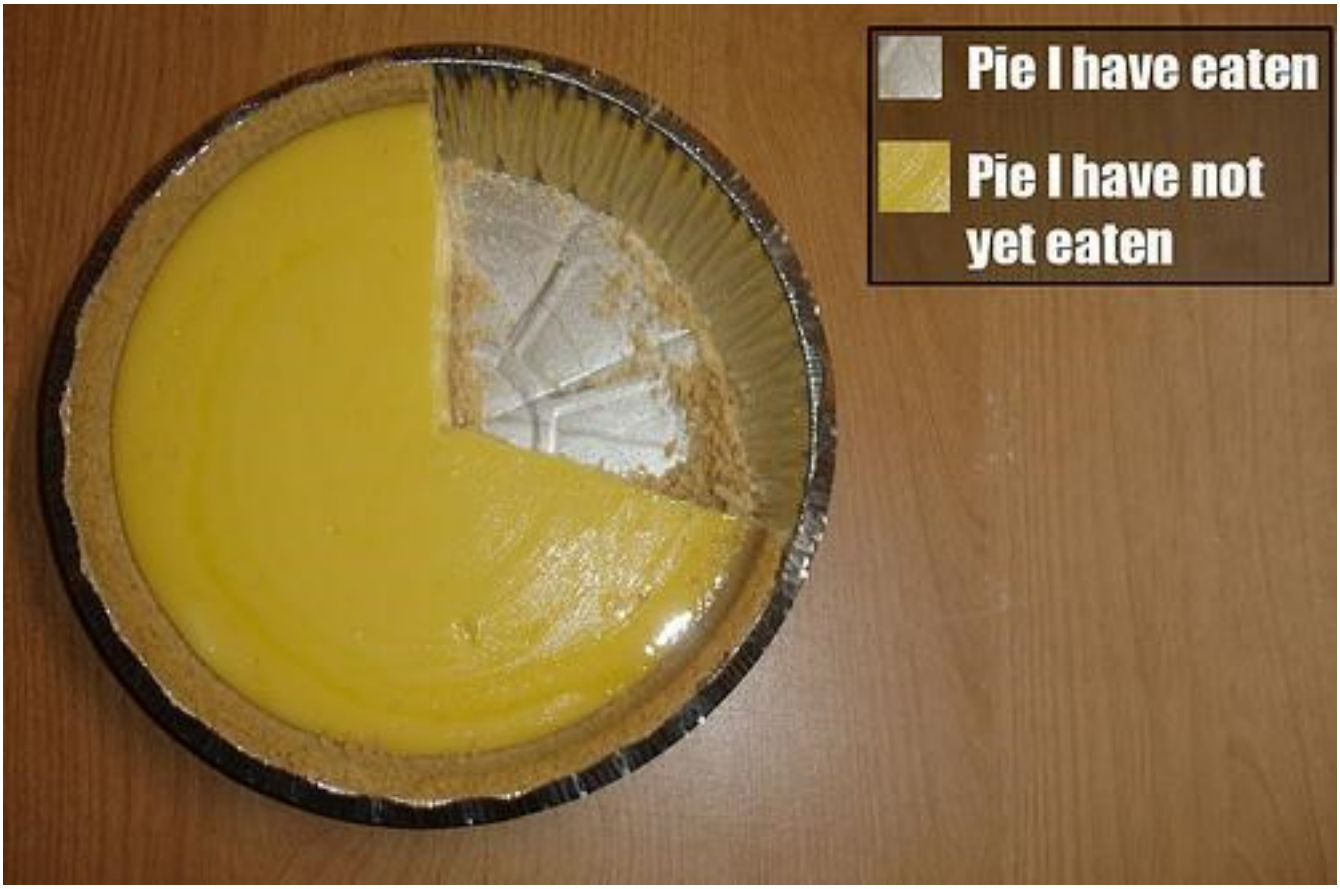
Unfortunately, one of the most common plots seen today for categorical data is the pie chart. While they may see harmless enough, they actually present a problem in that humans are unable to judge angles well. As Naomi Robbins describes in her book "Creating More Effective Graphs", we overestimate angles greater than 90 degrees and we underestimate angles less than 90 degrees. In other words, it is difficult for us to determine relative size of one piece of the pie compared to another.

Let's examine our previous barplot example on the number of flights departing NYC by airline. This time we will use a pie chart. As you review this chart, try to identify

- how much larger the portion of the pie is for ExpressJet Airlines (`EV`) compared to US Airways (`US`),
- what the third largest carrier is in terms of departing flights, and
- how many carriers have fewer flights than United Airlines (`UA`)?



While it is quite easy to look back at the barplot to get the answer to these questions, it's quite difficult to get the answers correct when looking at the pie graph. Barplots can always present the information in a way that is easier for the eye to determine relative position. There may be one exception from Nathan Yau at FlowingData.com but we will leave this for the reader to decide:



Learning check

(LC4.17) Why should pie charts be avoided and replaced by barplots?
(LC4.18) What is your opinion as to why pie charts continue to be used?

4.4.2 Using barplots to compare two variables

Barplots are the go-to way to visualize the frequency of different categories of a categorical variable. They make it easy to order the counts and to compare one group's frequency to another. Another use of barplots (unfortunately, sometimes inappropriately and confusingly) is to compare two categorical variables together. Let's examine the distribution of outgoing flights from NYC by `carrier` and `airport`.

We begin by getting the names of the airports in NYC that were included in the `flights` dataset. Remember from Chapter 3 that this can be done by using the `inner_join` function in the `dplyr` package.

```
library(dplyr)
flights_namedports <- inner_join(flights, airports, by = c("origin" = "faa"))
str(flights_namedports)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    336776 obs. of  25 variables:
##  $ year          : int  2013 2013 2013 2013 2013 2013 2013 2013 2013 2013 ...
##  $ month         : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ day           : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ dep_time      : int  517 533 542 544 554 554 555 557 557 558 ...
##  $ sched_dep_time: int  515 529 540 545 600 558 600 600 600 600 ...
##  $ dep_delay     : num  2 4 2 -1 -6 -4 -5 -3 -3 -2 ...
##  $ arr_time      : int  830 850 923 1004 812 740 913 709 838 753 ...
##  $ sched_arr_time: int  819 830 850 1022 837 728 854 723 846 745 ...
##  $ arr_delay     : num  11 20 33 -18 -25 12 19 -14 -8 8 ...
##  $ carrier       : chr  "UA" "UA" "AA" "B6" ...
##  $ flight        : int  1545 1714 1141 725 461 1696 507 5708 79 301 ...
##  $ tailnum       : chr  "N14228" "N24211" "N619AA" "N804JB" ...
##  $ origin        : chr  "EWR" "LGA" "JFK" "JFK" ...
##  $ dest          : chr  "IAH" "IAH" "MIA" "BQN" ...
##  $ air_time      : num  227 227 160 183 116 150 158 53 140 138 ...
##  $ distance      : num  1400 1416 1089 1576 762 ...
##  $ hour          : num  5 5 5 5 6 5 6 6 6 6 ...
##  $ minute        : num  15 29 40 45 0 58 0 0 0 0 ...
##  $ time_hour     : POSIXct, format: "2013-01-01 05:00:00" ...
##  $ name          : chr  "Newark Liberty Intl" "La Guardia" "John F Kennedy Intl" "John F Kennedy Intl"
##  $ lat           : num  40.7 40.8 40.6 40.6 40.8 ...
##  $ lon           : num  -74.2 -73.9 -73.8 -73.8 -73.9 ...
##  $ alt           : int  18 22 13 13 22 18 18 22 13 22 ...
##  $ tz            : num  -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 ...
##  $ dst           : chr  "A" "A" "A" "A" ...
```

We see that `name` now corresponds to the name of the airport as referenced by the `origin` variable. We will now plot `carrier` as the horizontal variable. When we specify `geom_bar`, it will specify `count` as being the vertical variable. A new addition here is `fill = name`. Look over what was produced from the plot to get an idea of what this argument gives.

```
ggplot(data = flights_namedports, mapping = aes(x = carrier, fill = name)) +
  geom_bar()
```

This plot is what is known as a **stacked barplot**. While simple to make, it often leads to many problems.

Learning check

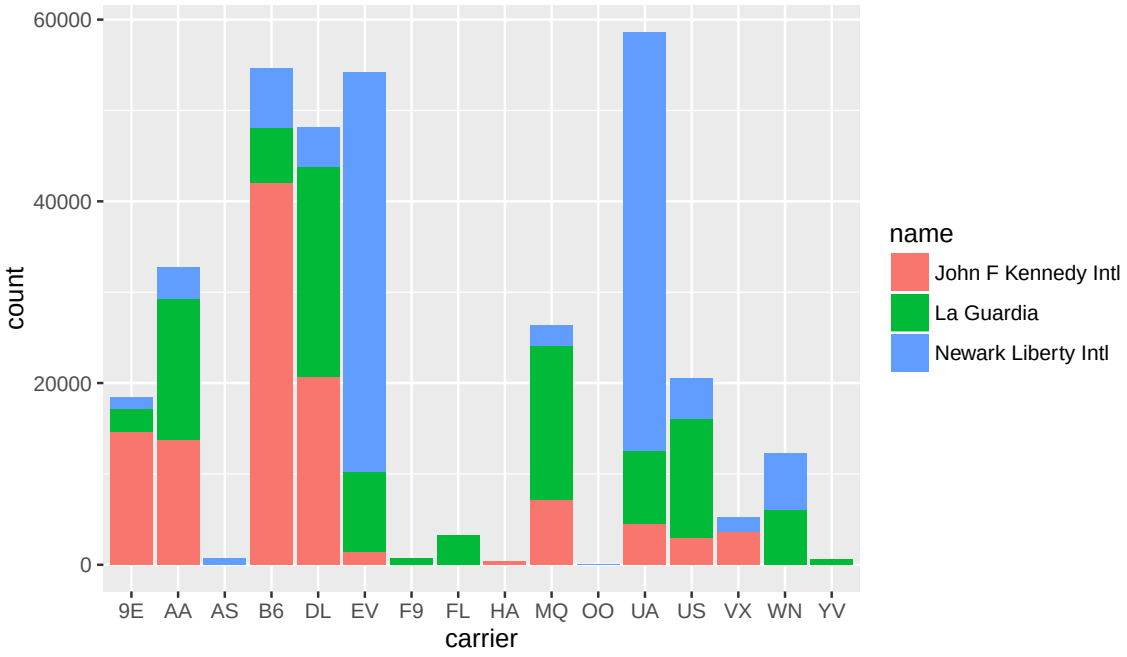


Figure 4.6: Stacked barplot comparing the number of flights by carrier and airport

(LC4.19) What kinds of questions are not easily answered by looking at the above figure?

(LC4.20) What can you say, if anything, about the relationship between airline and airport in NYC in 2013 in regards to the number of departing flights?

Another variation on the **stacked barplot** is the **side-by-side barplot**.

```
ggplot(data = flights_namedports, mapping = aes(x = carrier, fill = name)) +
  geom_bar(position = "dodge")
```

Learning check

(LC4.21) Why might the side-by-side barplot be preferable to a stacked barplot in this case?

(LC4.22) What are the disadvantages of using a side-by-side barplot, in general?

Lastly, an often preferred type of barplot is the **faceted barplot**. We already saw this concept of faceting and small multiples in Subsection 4.3.1. This gives us a nicer way to compare the distributions across both `carrier` and airport/`name`.

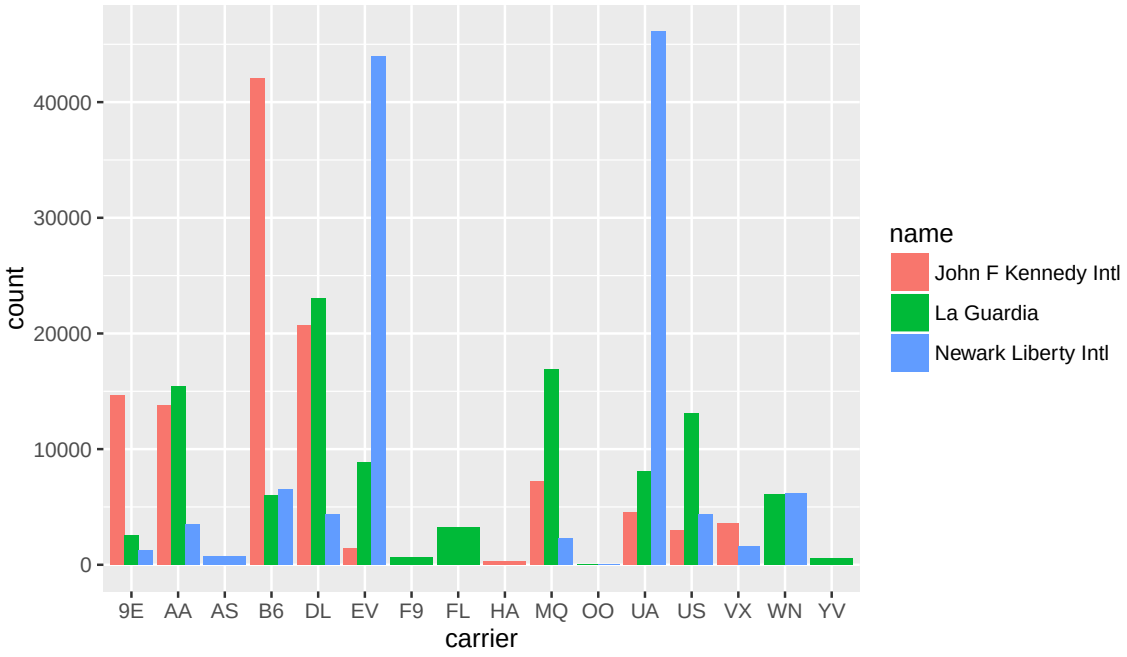


Figure 4.7: Side-by-side barplot comparing the number of flights by carrier and airport

```
ggplot(data = flights_namedports, mapping = aes(x = carrier, fill = name)) +
  geom_bar() +
  facet_grid(name ~ .)
```

Note how the `facet_grid` function arguments are written here. We are wanting the names of the airports vertically and the `carrier` listed horizontally. As you may have guessed, this argument and other *formulas* of this sort in R are in `y ~ x` order. We will see more examples of this in Chapter 6.

Learning check

(LC4.23) Why is the faceted barplot preferred to the side-by-side and stacked barplots in this case?

(LC4.24) What information about the different carriers at different airports is more easily seen in the faceted barplot?

4.4.3 Summary

Barplots are the preferred way of displaying categorical variables. They are easy-to-understand and to make comparisons across groups of a categorical variable. When dealing with more than

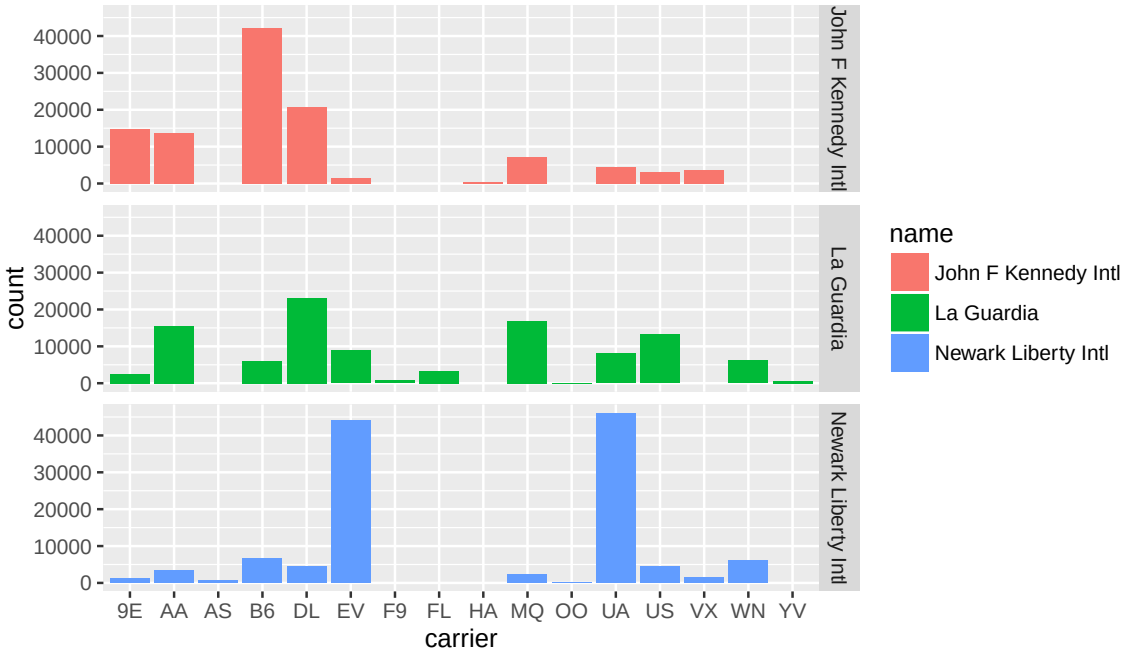


Figure 4.8: Faceted barplot comparing the number of flights by carrier and airport

one categorical variable, faceted barplots are frequently preferred over side-by-side or stacked barplots. Stacked barplots are sometimes nice to look at, but it is quite difficult to compare across the levels since the sizes of the bars are all of different sizes. Side-by-side barplots can provide an improvement on this, but the issue about comparing across groups still must be dealt with.

4.5 Scatter-plots

We have seen that boxplots are most appropriate when plotting the distribution of ONE continuous variable across different levels/groups of ONE categorical variable. Barplots (preferably the faceted type) are best when looking at the distribution of ONE categorical variable across different levels of another categorical variable. But what if we are looking to investigate the relationship between TWO continuous variables? What is commonly produced is the well-known **scatter-plot**, which shows the points corresponding to the values of each of the variables scattered around.

We will now investigate arrival delays (the vertical "y" axis variable) versus departure delays (the horizontal "x" axis variable) for Alaska Airlines flights leaving NYC in 2013. Notice the new function that is invoked here: `filter`, which resides in the `dplyr` package. You will see many more examples using this function in Chapter 5. The `filter` function goes through the dataframe specified (`flights` here) and selects only those rows which meet the condition given (`carrier == "AS"` here).

```
alaska_flights <- filter(flights, carrier == "AS")
ggplot(alaska_flights, aes(x = dep_delay, y = arr_delay)) +
```

```
  geom_point()
```

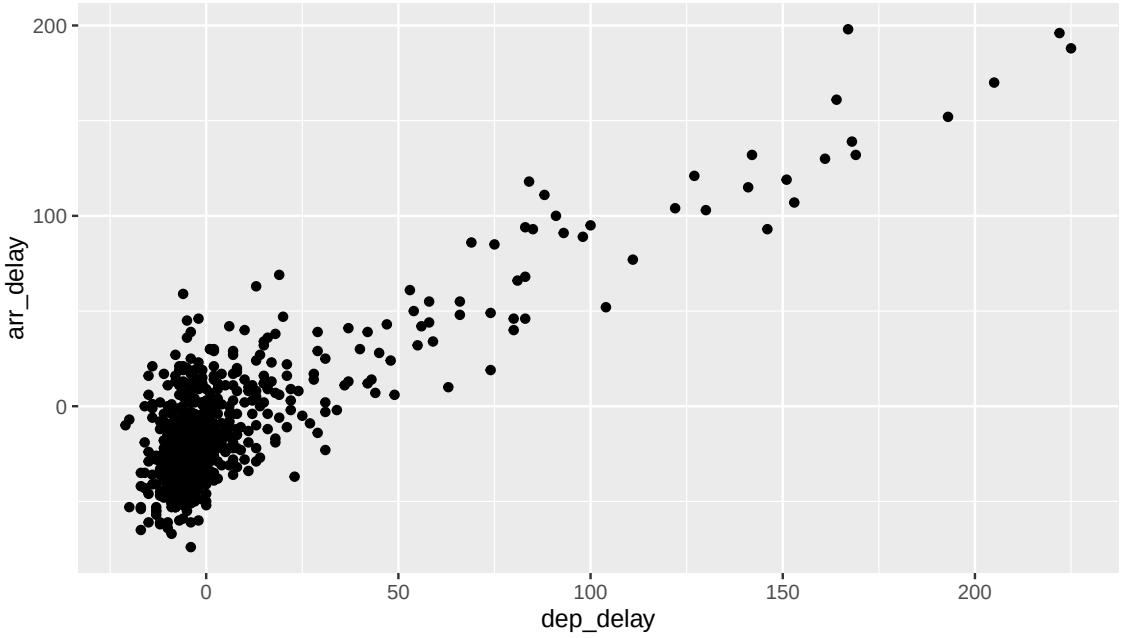


Figure 4.9: Arrival Delays vs Departure Delays for Alaska Airlines flights from NYC in 2013

We see that a positive relationship exists between `dep_delay` and `arr_delay`: as departure delays increase, arrival delays tend to also increase. We also note that the majority of points fall near the point (0, 0) here. There is a large mass of points clustered there.

Learning check

(LC4.25) What are some practical reasons why `dep_delay` and `arr_delay` have a positive relationship?

(LC4.26) What variables (not necessarily in the `flights` dataframe) would you expect to have a negative correlation (i.e. a negative relationship) with `dep_delay`? Why? Remember that we are focusing on continuous variables here.

(LC4.27) Why do you believe there is a cluster of points near (0, 0)?

- What does (0, 0) correspond to in terms of the Alaskan flights?

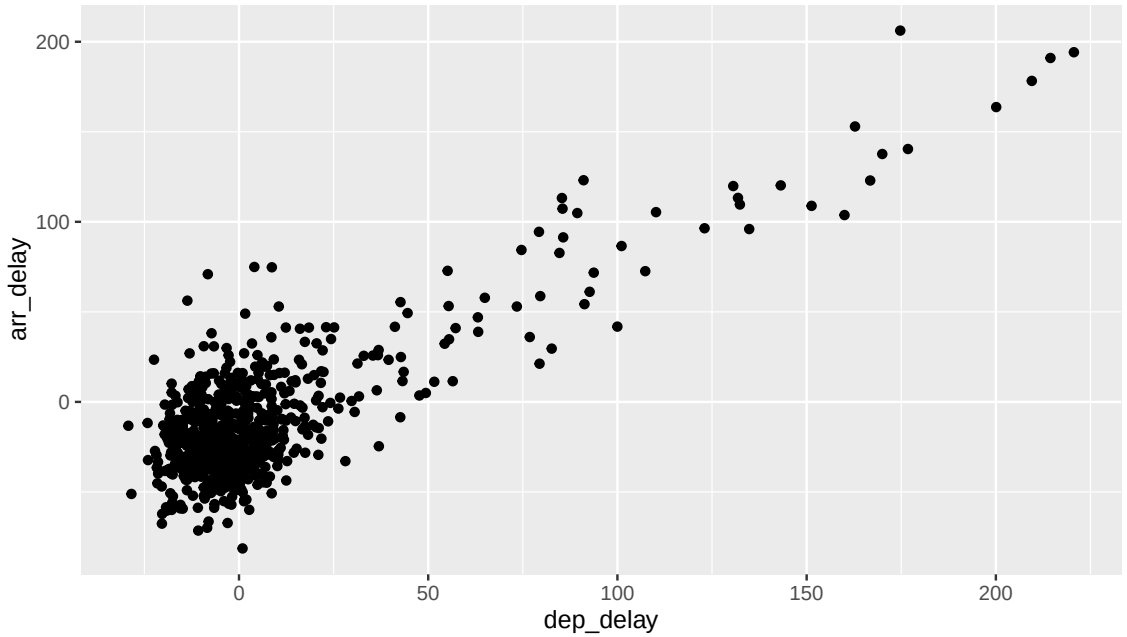
(LC4.28) What are some other features of the plot that stand out to you?

4.5.1 Jittering

The large mass of points near (0, 0) can cause some confusion. This is the result of a phenomenon called **over-plotting**. As one may guess, this corresponds to values being plotted on top of each other *over* and *over* again. It is often difficult to know just how many values are plotted in this way when looking at a basic scatter-plot as we have here.

One way of relieving this issue of **over-plotting** is to **jitter** the points a bit. In other words, we are going to add just a bit of random noise to the points to better see them and remove some of the over-plotting. You can think of "jittering" as shaking the points a bit on the plot. Instead of using `geom_point`, we use `geom_jitter` to perform this shaking and specify around how much jitter to add with the `width` and `height` arguments. This corresponds to how hard you'd like to shake the plot in units corresponding to those for both the horizontal and vertical variables (minutes here).

```
ggplot(alaska_flights, aes(x = dep_delay, y = arr_delay)) +
  geom_jitter(width = 30, height = 30)
```



This has helps us a little bit in getting a sense for the over-plotting, but with a relatively large dataset like this one (714 flights), it is often useful to change the transparency of the points as seen in the next section.

4.5.2 Setting transparency

One of the arguments that can be changed with `geom_point` is `alpha`. By default, this value is set to `1`. We can change this value to a smaller fraction to change the transparency of the points in the plot:

```
ggplot(alaska_flights, aes(x = dep_delay, y = arr_delay)) +
  geom_point(alpha = 0.2)
```

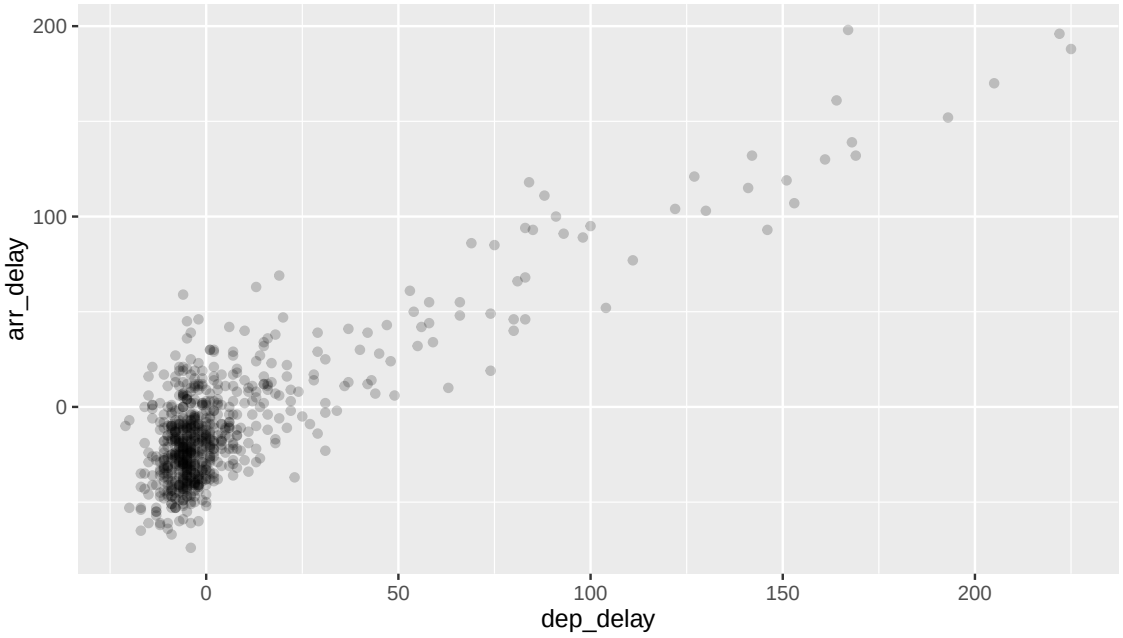


Figure 4.10: Arrival Delays vs Departure Delays for Alaska Airlines flights from NYC in 2013 - alpha=0.2

We can also specify the `alpha` argument in `geom_jitter`:

```
ggplot(alaska_flights, aes(x = dep_delay, y = arr_delay)) +
  geom_jitter(width = 30, height = 30, alpha = 0.3)
```

Learning check

(LC4.29) Why is setting the `alpha` argument value useful with scatter-plots?

- What further information does it give you that a regular scatter-plot cannot?

(LC4.30) After viewing the 4.10 above, give a range of arrival times and departure times that occur most frequently?

- How has that region changed compared to when you observed the same plot without the `alpha = 0.2` set in 4.9?

Maybe include a shading of the points by another variable example here for multivariate thinking?

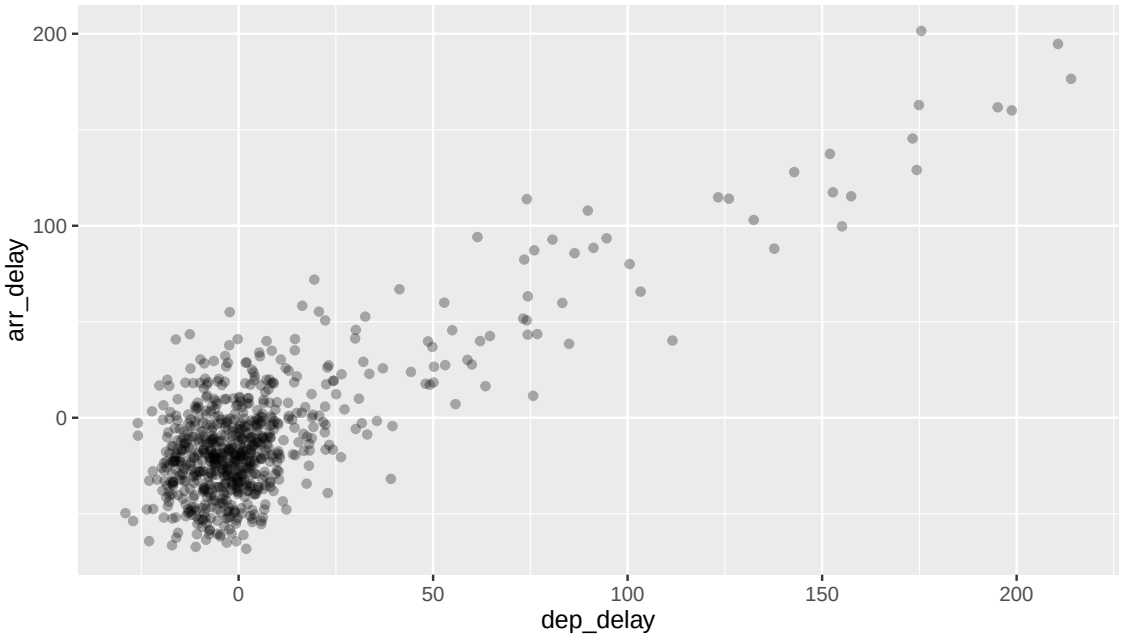


Figure 4.11: Arrival Delays vs Departure Delays for Alaska Airlines flights from NYC in 2013 - jitter and alpha added

4.5.3 Summary

Scatter-plots may be the most used plot today and they can provide an immediate way to see the trend in one variable versus another. Remember that they only make sense when plotting a continuous variable versus a continuous variable though. If you try to create a scatter-plot where either one of the two variables is not quantitative, you will get strange results. Be careful!

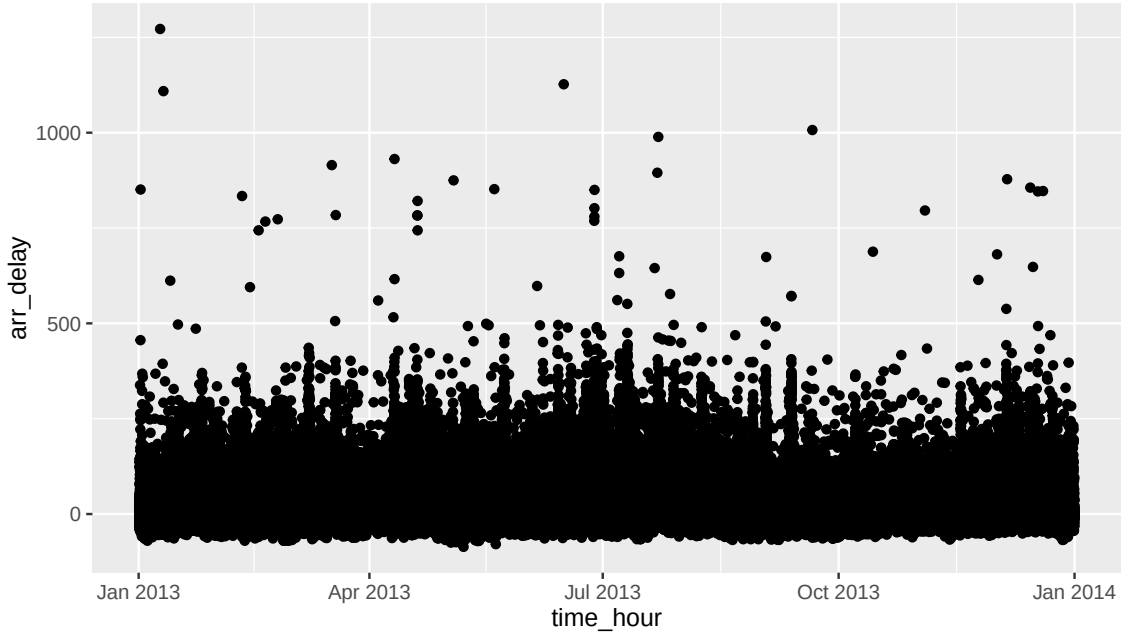
With medium to large datasets, you may need to tweak arguments in both `geom_jitter` and the `alpha` parameter in order to get a good feel for relationships in your data. This tweaking is often a fun part of data visualization since you'll have the chance to see different relationships come about as you make subtle changes to your plots.

4.6 Line-graphs

The last of the FNG is a line-graph. They are most frequently used when the horizontal axis is time. Time represents a variable that is connected together by each day following the previous day. In other words, time has a natural ordering. Line-graphs should be avoided when there is not a clear ordering to the explanatory ("x" variable).

We are interested in exploring the arrival delays by day throughout the year of 2013 from outgoing flights from New York City. If we plotted all of these values, we obtain the following scatter-plot:

```
ggplot(flights, aes(x = time_hour, y = arr_delay)) +
  geom_point()
```



We see that this plot is difficult to understand based on the sheer number of points plotted. We see some outlier points with more than 500 minutes of arrival delay, but even some jittering and transparency is not going to help us here.

Instead of plotting all of the values for each hour for all flights, it might make sense to plot the average value for each day in terms of arrival delays. Of course, we also need to address which average we should use: mean or median. With there being some outliers here, we have chosen to use the median arrival delay (since the mean is heavily influence by outliers).

You may think that this is a difficult task but the `group_by` and `summarize` functions make this a breeze. You'll see more examples using these two functions in Chapter 5. Here we will create a new variable, which corresponds to the month and day combined, from the `time_hour` column using the `mutate` function and create a new dataframe called `flights_day`. We also give the `str` of this new dataset so you can see the new variable added `date`:

```
flights_day <- mutate(flights, date = as.Date(time_hour))
str(flights_day)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    336776 obs. of  20 variables:
##  $ year          : int  2013 2013 2013 2013 2013 2013 2013 2013 2013 2013 ...
##  $ month         : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ day           : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ dep_time      : int  517 533 542 544 554 554 555 557 557 558 ...
##  $ sched_dep_time: int  515 529 540 545 600 558 600 600 600 600 ...
##  $ dep_delay     : num  2 4 2 -1 -6 -4 -5 -3 -3 -2 ...
##  $ arr_time      : int  830 850 923 1004 812 740 913 709 838 753 ...
##  $ sched_arr_time: int  819 830 850 1022 837 728 854 723 846 745 ...
##  $ arr_delay     : num  11 20 33 -18 -25 12 19 -14 -8 8 ...
##  $ carrier       : chr  "UA" "UA" "AA" "B6" ...
```

```
##  $ flight        : int  1545 1714 1141 725 461 1696 507 5708 79 301 ...
##  $ tailnum       : chr  "N14228" "N24211" "N619AA" "N804JB" ...
##  $ origin        : chr  "EWR" "LGA" "JFK" "JFK" ...
##  $ dest          : chr  "IAH" "IAH" "MIA" "BQN" ...
##  $ air_time      : num  227 227 160 183 116 150 158 53 140 138 ...
##  $ distance      : num  1400 1416 1089 1576 762 ...
##  $ hour          : num  5 5 5 5 6 5 6 6 6 6 ...
##  $ minute        : num  15 29 40 45 0 58 0 0 0 0 ...
##  $ time_hour     : POSIXct, format: "2013-01-01 05:00:00" ...
##  $ date          : Date, format: "2013-01-01" ...
```

```
flights_summarized <- flights_day %>% group_by(date) %>%
  summarize(median_arr_delay = median(arr_delay, na.rm = TRUE))
head(flights_summarized)
```

```
## # A tibble: 6 x 2
##         date median_arr_delay
##       <date>            <dbl>
## 1 2013-01-01                3
## 2 2013-01-02                4
## 3 2013-01-03                1
## 4 2013-01-04               -8
## 5 2013-01-05               -7
## 6 2013-01-06               -1
```

You will see the "pipe" operator `%>%` explained in more detail in Chapter 5, but you can read it as "and then". Here, we take the `flights_day` dataframe that we just created and then group it together by `date`. This goes through the dataframe and puts together all rows that have `2013-01-01` together, all rows that have `2013-01-02` together, ..., and all rows that have `2013-12-31` together. And then it looks at the median value of `arr_delay` over each one of the days. You can get a glimpse of the first few rows of this new dataset above since we invoked the `head` function on it.

Note also that there are missing values in this data set so we need to exclude them from the analysis. This is why the `na.rm = TRUE` argument is invoked. Many functions require this extra specification so it's always a good idea to run a `?median` or `?mean` before you try to run the function. Or you can always run it afterwards as well when you get strange results.

Now getting back to our line-graph. We want to plot the median arrival delay over all airlines on all days in 2013 from departing flights in NYC. This syntax should look similar to what we have seen before with plots involving `ggplot`. Notice that we are using the `flights_summarized` dataset here and not the `flights_day` or `flights` dataframes.

```
ggplot(data = flights_summarized, aes(x = date, y = median_arr_delay)) +
  geom_line()
```

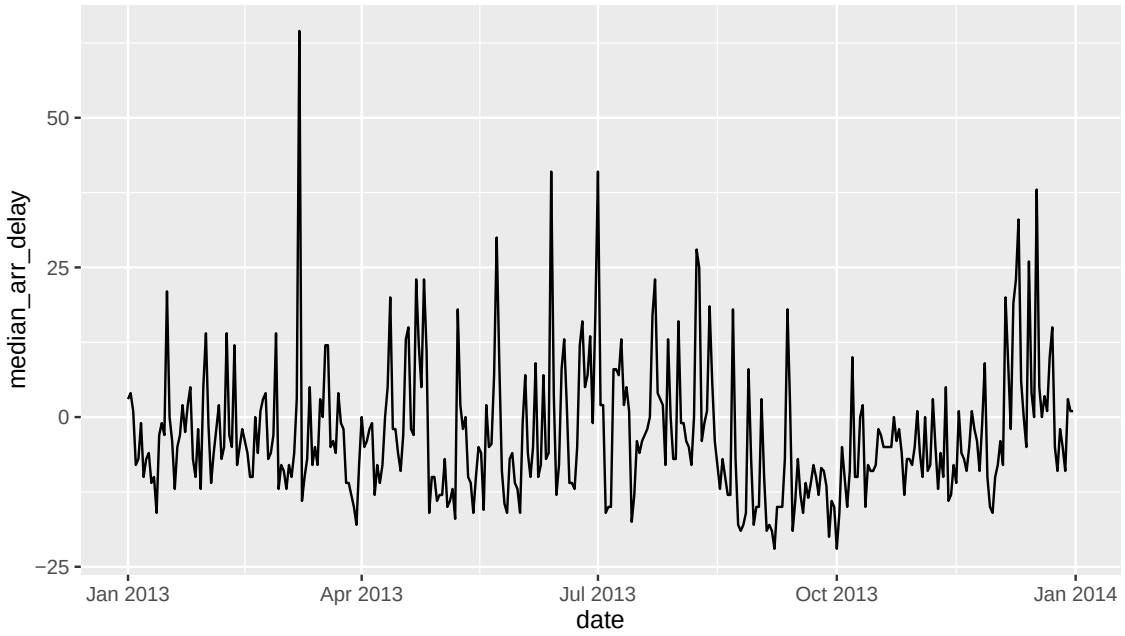
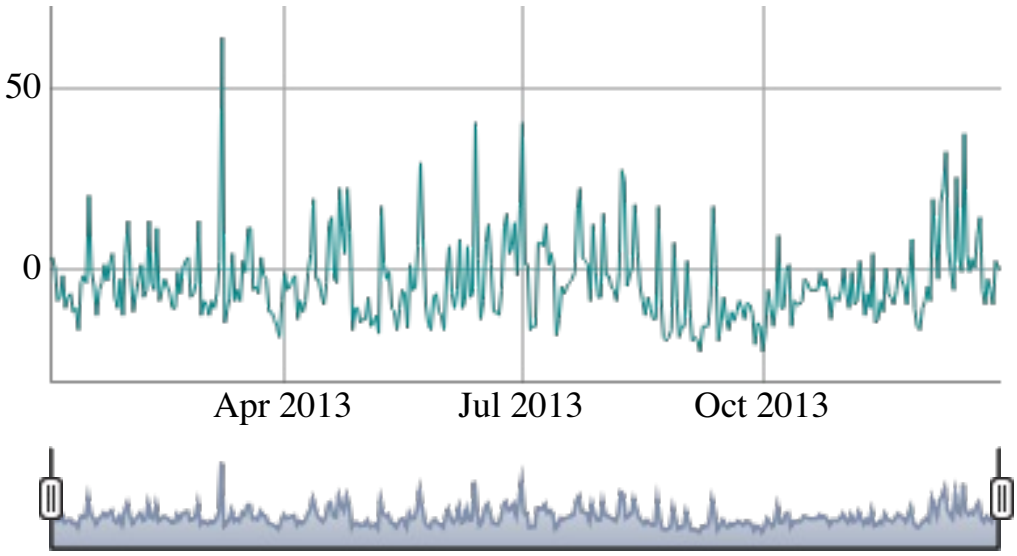


Figure 4.12: Line-graph of median arrival delay for flights leaving NYC in 2013 versus day of the year

4.6.1 Interactive line-graphs

Another useful tool for viewing line-graphs such as this is the `dygraph` function in the `dygraphs` package in combination with the `dyRangeSelector` function. This allows us to zoom in on a selected range and get an interactive plot for us to work with:

```
library(dygraphs)
rownames(flights_summarized) <- flights_summarized$date
flights_summarized <- select(flights_summarized, -date)
dyRangeSelector(dygraph(flights_summarized))
```



The syntax here is a little different than what we have covered so far. The `dygraph` function is expecting for the dates to be given as the `rownames` of the object. We then remove the `date` variable from the `flights_summarized` dataframe since it is accounted for in the `rownames`. Lastly, we run the `dygraph` function on the new dataframe that only contains the median arrival delay as a column and then provide the ability to have a selector to zoom in on the interactive plot via `dyRangeSelector`. (Note that this plot will only be interactive in the HTML version of this book.)

Include this interactive dygraphs stuff here or in Appendix B?

Include bad example of when a line-graph would be invalid?

Learning check

(LC4.31) Why should line-graphs be avoided when there is not a clear ordering of the horizontal axis?

(LC4.32) Why are line-graphs frequently used when time is the explanatory variable?

(LC4.33) Why did we use the `flights_summarized` dataframe to produce the line-graph in Figure 4.12 instead of `flights` or `flights_day`?

(LC4.34) Are the largest median arrival delays where you expected them to occur on the line-graph above in Figure **??**?

(LC4.35) Use the interactive line-graph to determine the highest median arrival delay for flights from NYC in 2013. What date was it and what do you think contributed to it?

4.6.2 Summary

Line-graphs provide a useful tool for viewing a continuous variable that is plotted versus time. We need to be careful to not be too entrenched in using line-graphs whenever we wish though. They only make sense when the explanatory variable (the one on the explanatory variable) has a natural ordering. We can mislead our audience if that isn't the case.

4.7 Brief Review of The Grammar of Graphics

You have seen all of the major pieces behind "The Grammar of Graphics" which serves as the basis for the `ggplot2` package. Here is a summary of each part:

May need some tweaking: `http://www.ling.upenn.edu/~joseff/avml2012/`

Review questions

(RQ4.1)

- Have a variety of bad plots with data for the readers and have readers create better plots with `ggplot2`
- Have sample datasets to work with from problem statements
  - Identify the appropriate plot to address the questions of interest

4.8 What's to come?

Last updated:

```
## [1] "Sunday, July 31, 2016 17:22:26 CDT"
```

5
Manipulating Data

- Going through a lot of examples with `dplyr` here
  - Definitely want to introduce them to the pipe
  - Show why it is so much better than nesting/temporary variables
  - Show how to get the summary statistics mentioned in Viz across groups with `group_by` and `summarize`
  - Show how to rename variables such as the `name` column from `airlines`
  - Get number of elements in specific columns similar to `table()` function with `dplyr`
  - Nick Horton's slides from JSM
- Database normalization
- `https://en.wikipedia.org/wiki/Database_normalization`
- Add importance of joining tables
- Make sure to refer back to plots in the viz chapter and how the material here relates to answering those questions
- We'll also need to address how this leads into inference

6
Inference

Topics

- (random) Sampling: representativeness/generalizability/bias
-

7

Appendix A: R and RStudio Basics

- What is R? What is RStudio?
- Screenshots of RStudio frames?
- Installing R and RStudio directions with screenshots
- Give an introduction into using R
- Mean, median, standard deviation, five-number summary, distribution
- Some content to cover:
  - data structures (vectors, lists, data frames, matrices)
  - indexing/subsetting
  - functions (default arguments)
  - Case matters in R!
  - Why do some arguments require quotations and others don't?
- RMarkdown chunk options

8
Appendix B: Intermediate R

8.1 Sorted barplots

Building upon the example in Section 4.4:

```
library(nycflights13)
flights_table <- table(flights$carrier)
flights_table
```

```
##
##    9E    AA    AS    B6    DL    EV    F9    FL    HA    MQ    OO    UA
## 18460 32729   714 54635 48110 54173   685  3260   342 26397    32 58665
##    US    VX    WN    YV
## 20536  5162 12275   601
```

More information on the use of this `$` syntax is available in Chapter 3 and in Appendix A - Chapter 7.

We can sort this table from highest to lowest counts by using the `sort` function:

```
sorted_flights <- sort(flights_table, decreasing = TRUE)
names(sorted_flights)
```

```
##  [1] "UA" "B6" "EV" "DL" "AA" "MQ" "US" "9E" "WN" "VX" "FL" "AS" "F9"
## [14] "YV" "HA" "OO"
```

It is often preferred for barplots to be ordered corresponding to the heights of the bars. This allows the reader to more easily compare the ordering of different airlines in terms of departed flights (Robbins, 2013). We can also much more easily answer questions like "How many airlines have more departing flights than Southwest Airlines?".

We can use the sorted table giving the number of flights defined as `sorted_flights` to **reorder** the `carrier`.

```
ggplot(data = flights, mapping = aes(x = carrier)) +
  geom_bar() +
  scale_x_discrete(limits = names(sorted_flights))
```

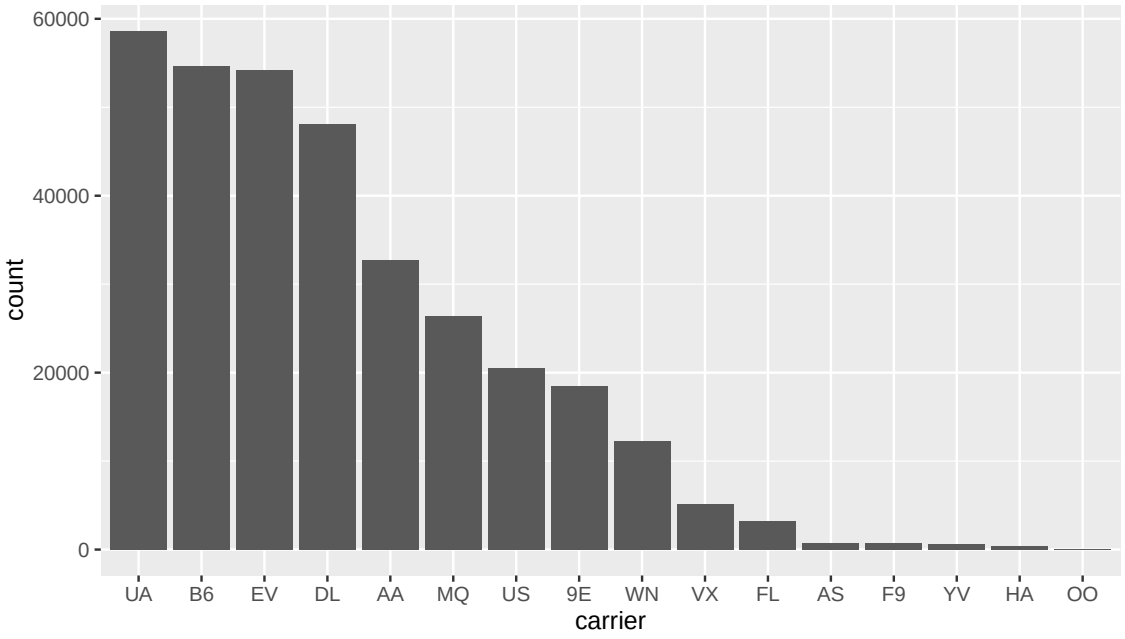


Figure 8.1: Number of flights departing NYC in 2013 by airline - Descending numbers

The last addition here specifies the values of the horizontal `x` axis on a discrete scale to correspond to those given by the entries of `sorted_flights`.

** ** What are three specific questions that can be more easily answered by looking at Figure 4.6 instead of Figure 4.5?

- Changing the labels of a plot (x-axis, y-axis)
- Changing the theme
- Adding `code_folding` and `code_download` to YAML

9
Bibliography

Robbins, N. (2013). *Creating More Effective Graphs*. Chart House.

Wickham, H. (2014). Tidy data. *Journal of Statistical Software*, Volume 59(Issue 10).

Wickham, H. (2016). *nycflights13: Flights that Departed NYC in 2013*. R package version 0.2.0.

Wickham, H. and Chang, W. (2016). *ggplot2: An Implementation of the Grammar of Graphics*. R package version 2.1.0.

Wickham, H. and Grolemund, G. (2016). *R for Data Science*. O'Reilly.

Wilkinson, L. (2005). *The Grammar of Graphics (Statistics and Computing)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.