6

dplyr: Manipulating Data Frames

To work with data, we need a place to store it in R. Our default setting is to store data in data frames in a *tidy* format**. When we work with properly formatted data frames, each data frame can be thought of as a collection of observations with each observation in its own row and each recorded variable (e.g. measurement) represented in a column; a generic dataframe with M rows and $N+1$ columns (i.e. N variables plus an `id` column) is shown in Table 6.1.

** Data will not always be stored in a way that is amenable to analysis. Typically, we will get our data into a tidy format - such that each row represents an observation and each column represents an attribute or property of that observation

Table 6.1: Think of a data frame as consisting of rows of observations and columns of variables.

observationID	variable1	variable2	variable...	variableN
1	xx	xx	xx	xx
2	xx	xx	xx	xx
⋮	⋮	⋮	⋮	⋮
M	xx	xx	xx	xx

We can look at a subset of the built-in `mtcars` dataset as a more tangible example of a tidy dataframe:

carID	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2
Camaro Z28	13.3	8	350.0	245	3.73	3.840	15.41	0	0	3	4
Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2

where each row represents a particular car and the recorded data associated with each car organized by column. Note that an `ID` column is also just an attribute of that observation.

There are infinite ways data can be non-tidy. Two non-tidy examples, assuming the observational unit is still just one car, might look like this:

carID	measure	value
Mazda RX4	mpg	21.0
Honda Civic	mpg	30.4
Camaro Z28	mpg	13.3
Volvo 142E	mpg	21.4
Mazda RX4	cyl	6.0
Honda Civic	cyl	4.0
Camaro Z28	cyl	8.0
Volvo 142E	cyl	4.0

where one particular observational unit is on multiple rows or alternatively, like this:

carID	mpg	eng:cyl_disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6 - 160	110	3.90	2.620	16.46	0	1	4	4
Honda Civic	30.4	4 - 75.7	52	4.93	1.615	18.52	1	1	4	2
Camaro Z28	13.3	8 - 350	245	3.73	3.840	15.41	0	0	3	4
Volvo 142E	21.4	4 - 121	109	4.11	2.780	18.60	1	1	4	2

where multiple variables (i.e. `cyl` and `disp`) might be stored together in one column. See (`https://tidyr.tidyverse.org/articles/tidy-data.html`) for further examples of messy data.

6.1 Tibbles

While R's data frame object has been the long running standard placeholder for data, we will often convert data frame's into `tibble` objects. The `as_tibble` function from the `dplyr` package does this conversion for us:

Remember, to load a package into an R session using `library(packageName)`, you must first have the package installed on your system. To do this for the `dplyr` package, you would execute `install.packages("dplyr")`.

```
## make the dplyr package and its function available
## in the current R session.
library(dplyr)

## convert the built in mtcars dataframe to a tibble
as_tibble(mtcars)
```

```
## # A tibble: 32 x 11
##      mpg   cyl  disp    hp  drat    wt  qsec
##   * <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
##   1  21.0    6.  160.  110.  3.90  2.62  16.5
##   2  21.0    6.  160.  110.  3.90  2.88  17.0
##   3  22.8    4.  108.   93.  3.85  2.32  18.6
##   4  21.4    6.  258.  110.  3.08  3.22  19.4
##   5  18.7    8.  360.  175.  3.15  3.44  17.0
##   6  18.1    6.  225.  105.  2.76  3.46  20.2
##   7  14.3    8.  360.  245.  3.21  3.57  15.8
```

```
## 8  24.4    4.  147.   62.  3.69  3.19  20.0
## 9  22.8    4.  141.   95.  3.92  3.15  22.9
## 10 19.2    6.  168.  123.  3.92  3.44  18.3
## # ... with 22 more rows, and 4 more
## #   variables: vs <dbl>, am <dbl>,
## #   gear <dbl>, carb <dbl>
```

The main advantage of a tibble is that when it is printed out, it will not try to print all the data. By default, tibble's show the first 10 rows of data and as many columns as will fit on your screen. Once we start working with datasets that have tens of thousands of rows and dozens of columns, you will then appreciate the tibble. In this book, we will use the terms *tibble* and *data frame* synonymously because they differ only very slightly in their behavior and most of the time Internet resources will almost exclusively refer to data frames as the object for data storage - the *tibble* terminology is far less ubiquitous.

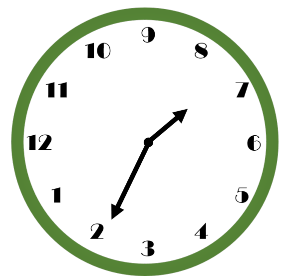


Figure 6.1: What time does this clock read? A non-standardized design increases the amount of thinking time required to get the answer. After significant deliberation, hopefully you see that the time is 7:11.

6.2 Reducing Cognitive Load

Data manipulation is not a natural human task - there is definitely some mental gymnastics required. To simplify the cognitive load, we will adopt a standard way of thinking about data manipulation. These standards will reduce the cognitive load, i.e. thinking time, required of your brain as you get data into more useful forms. An example standard that you might take for granted is that used by clock makers. But see the example in Figure 6.1 and you will quickly realize, by example, how important standards can be in aiding your thinking (example from Norman, 2013).

When it comes to data manipulation, the standards we will learn in this chapter are implemented in the `dplyr` package. The `dplyr` package simplifies our thought process in regards to data manipulation by reducing our possible operations to five main verbs and one adverb. It then facilitates the chaining of these operations to accomplish even the most difficult of data manipulation tasks. The five main verbs are:

The dplyr package is part of the tidyverse (https://www.tidyverse.org/) ecosystem of packages. These packages are all designed to reduce cognitive load through a set of well-thought out standards which share an underlying design philosophy and structure.

1. `filter()`: select subset of rows (i.e. observations). See Figure 6.2.
2. `arrange()`: reorder rows
3. `select()`: select subset of columns (i.e. variables). See Figure 6.3.
4. `mutate()`: create new columns that are functions of existing columns. See Figure 6.4.
5. `summarize()`: collapse data into a single row. See Figure 6.5.

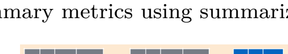
Figure 6.2: Filtering data to get a subset of rows.

Figure 6.3: Selecting data to get a subset of columns.

Figure 6.4: Create new columns that are functions of existing columns using mutate.

Figure 6.5: Collapsing data into summary metrics using summarize.

These verbs are useful on their own, but they become really powerful when you apply them to groups of observations within a dataset. In dplyr, you do this with the `group_by()` function. It breaks down



a dataset into specified groups of rows. When you then apply the verbs above on the resulting object they'll be automatically applied *by group*. Most importantly, all this is achieved by using the same exact syntax you'd use with an ungrouped object.

Grouping affects the verbs as follows:

- grouped *arrange()* orders first by the grouping variables and then by the variables of arrange
- grouped `summarize()` is perhaps the most powerful to combine with grouping. You use `summarize` with aggregate functions, which take a vector of values and return a single number. See Figure 6.6. There are many useful examples of aggregate functions in base R like `min()`, `max()`, `mean()`, `sum()`, `sd()`, `median()`, and `IQR()`. `dplyr` provides a handful of others:
  - `n()`: the number of observations in the current group
  - `n_distinct()`: the number of unique values in x.
  - `first(x), last(x)` and `nth(x, n)`: these work similarly to `x[1]`, `x[length(x)]`, and `x[n]`, but give you more control over the result if the value is missing.

Learning `dplyr` is perhaps easiest by example. Following the vignette that accompnaies the `dplyr` package (`https://cran.r-project.org/web/packages/dplyr/vignettes/dplyr.html`), we'll start with the built in `nycflights13` data frame. This dataset contains all 336,776 flights that departed from New York City in 2013 (see link in margin). A description of the dataset fields is available from the Bureau of Transportation Statistics.

Raw flight data is available at `http://www.transtats.bts.gov/Fields.asp?Table_ID=236`.

```
## Uncomment the install line if the package has not
## been installed on your computer.
#install.packages("nycflights13", dependencies = TRUE)
#install.packages("dplyr", dependencies = TRUE)
library(nycflights13)
library(dplyr)

## Show the dataframe in the RStudio envirnoment
flights = flights

## just fyi, there are lots of datasets already in R
data()
```

6.3 Filter Rows

`filter` allows you to select a subset of rows in a data frame. The first argument is the name of the data frame. The second and subsequent arguments are the expressions that filter the data frame:

For example, we can select all flights on January 1st with:

Take note of the double equal sign == which is used for logical comparison. The single equal sign = is only used for assignment purposes.

```
filter(flights, day == 1, month == 1)
```

```
## # A tibble: 842 x 19
##     year month   day dep_time sched_dep_time
##    <int> <int> <int>    <int>          <int>
##  1  2013     1     1      517            515
##  2  2013     1     1      533            529
##  3  2013     1     1      542            540
##  4  2013     1     1      544            545
##  5  2013     1     1      554            600
##  6  2013     1     1      554            558
##  7  2013     1     1      555            600
##  8  2013     1     1      557            600
##  9  2013     1     1      557            600
## 10  2013     1     1      558            600
## # ... with 832 more rows, and 14 more
## #   variables: 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>
```

Exercise 6.1. Create a new data frame called `uaFlights` that finds all flights where the carrier was United Airlines.

6.4 Arrange Rows

`arrange()` works similarly to `filter()` except that instead of filtering or selecting rows, it reorders them. It takes a data frame, and a set of column names (or more complicated expressions) to order by. If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns.

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

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time
##    <int> <int> <int>    <int>          <int>
##  1  2013     1     1      517            515
```

```
## 2  2013     1     1      533            529
## 3  2013     1     1      542            540
## 4  2013     1     1      544            545
## 5  2013     1     1      554            600
## 6  2013     1     1      554            558
## 7  2013     1     1      555            600
## 8  2013     1     1      557            600
## 9  2013     1     1      557            600
## 10 2013     1     1      558            600
## # ... with 336,766 more rows, and 14 more
## #   variables: 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>
```

Use `desc()` to order a column in descending order

```
arrange(flights, desc(arr_delay))
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time
##    <int> <int> <int>    <int>          <int>
## 1  2013     1     9      641            900
## 2  2013     6    15     1432           1935
## 3  2013     1    10     1121           1635
## 4  2013     9    20     1139           1845
## 5  2013     7    22      845           1600
## 6  2013     4    10     1100           1900
## 7  2013     3    17     2321            810
## 8  2013     7    22     2257            759
## 9  2013    12     5      756           1700
## 10 2013     5     3     1133           2055
## # ... with 336,766 more rows, and 14 more
## #   variables: 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>
```

6.5 Select() columns

Often you work with large datasets with many columns but only a few are actually of interest to you. `select()` allows you to rapidly zoom in on a useful subset:

```
# Select columns by name
select(flights, year, month, day)
```

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

```
# Select all columns between year and day (inclusive)
select(flights, year:day)
```

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

```
# Select all columns except those from year to day (inclusive)
select(flights, -(year:day))
```

```
## # A tibble: 336,776 x 16
##    dep_time sched_dep_time dep_delay
##       <int>          <int>     <dbl>
##  1      517            515        2.
##  2      533            529        4.
##  3      542            540        2.
##  4      544            545       -1.
##  5      554            600       -6.
##  6      554            558       -4.
##  7      555            600       -5.
##  8      557            600       -3.
##  9      557            600       -3.
## 10      558            600       -2.
## # ... with 336,766 more rows, and 13 more
## #   variables: arr_time <int>,
## #   sched_arr_time <int>, arr_delay <dbl>,
## #   carrier <chr>, flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>,
## #   time_hour <dttm>
```

A common use of `select()` is to find the values of a set of variables. This is particularly useful in conjunction with the `distinct()` verb (a shortcut variant for the `filter()` verb) which only returns the unique values in a table.

```
distinct(select(flights, tailnum))
```

```
## # A tibble: 4,044 x 1
##    tailnum
##    <chr>
##  1 N14228
##  2 N24211
##  3 N619AA
##  4 N804JB
##  5 N668DN
##  6 N39463
##  7 N516JB
##  8 N829AS
##  9 N593JB
## 10 N3ALAA
## # ... with 4,034 more rows
```

Exercise 6.2. Create a new data frame called `routes` that consists of two columns which contain all combinations of flight origin and flight destination in the original dataset. How many unique routes are there?

6.6 *Add new columns with mutate()*

Besides selecting sets of existing columns, it's often useful to add new columns that are functions of existing columns. This is the job of `mutate()`:

```
flightSpeedDF = select(flights, distance, air_time)
mutate(flightSpeedDF,
  speed = distance / air_time * 60)
```

```
## # A tibble: 336,776 x 3
##    distance air_time speed
##       <dbl>    <dbl> <dbl>
##  1    1400.     227.  370.
##  2    1416.     227.  374.
##  3    1089.     160.  408.
##  4    1576.     183.  517.
##  5     762.     116.  394.
##  6     719.     150.  288.
##  7    1065.     158.  404.
##  8     229.      53.  259.
##  9     944.     140.  405.
## 10     733.     138.  319.
## # ... with 336,766 more rows
```

6.7 *summarize() values*

The last verb is `summarize()`. It collapses a data frame into a single row.

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

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

6.8 *Commonalities*

You may have noticed that the syntax and function of all these verbs are very similar:

- The first argument is a data frame (e.g. `flights` or `flightsDF`).
- The subsequent arguments describe what to do with the data frame. Notice that you can refer to columns in the data frame directly without using $.
- The resulting output is a new data frame.

Together these properties make it easy to chain together multiple simple steps to achieve a complex result.

These five functions provide the basis of a language of data manipulation. At the most basic level, you can only alter a tidy data frame in five useful ways: you can reorder the rows (`arrange()`), pick observations (`filter()`) and variables (`select()`) of interest, add new variables (`mutate()`) that are functions of existing variables, or collapse (`summarize()`) many values into a summary . The remainder of the language comes from applying the five functions to different types of data. For example, I'll discuss how these functions work with grouped data.

6.9 Grouped Operations

Powerful data manipulation is enabled by the combination of the `group_by()` and `summarize()` functions as the summarize operation will collapse each group of data. For example, we could use these to find the number of planes and the number of flights that go to each possible destination:

```
## create a new dataframe that is organized by groups
destinations = group_by(flights, dest)
## summarize the rows of the grouped data frame
destDF = summarize(destinations,
  planes = n_distinct(tailnum), # unique planes
  flights = n() # number of flights
)
destDF
```

```
## # A tibble: 105 x 3
##    dest  planes flights
##    <chr>  <int>   <int>
##  1 ABQ      108     254
##  2 ACK       58     265
##  3 ALB      172     439
##  4 ANC        6       8
##  5 ATL     1180   17215
##  6 AUS      993    2439
##  7 AVL      159     275
```

```
##   8 BDL      186     443
##   9 BGR       46     375
## 10 BHM       45     297
## # ... with 95 more rows
```

Exercise 6.3. Create a new data fram called `sortDestDF` that orders (i.e. arranges the `destDF` dataframe in descending order of popularity (i.e. number of flights from NYC to that destination) to discover the most popular places people from New York City fly to.

6.10 Chaining with %>%

In the previous examples, we sometimes had to save results to intermediate dataframes and then do subsequent analysis on the newly created dataframe. For example, if using the original `flights` data frame we wanted to find the destination airports that had the fastest average (mean) flight speed, we could do the following:

```
# create new data frame (df) with three columns extracted from flights data frame
lightSpeedDF = select(flights, distance, air_time, dest)
# create new data frame with additional column representing speed
lightSpeedDF2 = mutate(lightSpeedDF,
  speed = distance / air_time * 60)
# create new data frame that has hidden groupings by destination
lightSpeedDF3 = group_by(lightSpeedDF2, dest)
# create new data frame that summarizes speed for each destination group
lightSpeedDF4 = summarize(lightSpeedDF3, avgSpeed = mean(speed, na.rm = TRUE))
# print out a sorted data frame - note that this does not create a new data frame
# as there is no assignment operator (i.e. '=')
arrange(lightSpeedDF4, desc(avgSpeed))
```

```
## # A tibble: 105 x 2
##    dest  avgSpeed
##    <chr>    <dbl>
##   1 ANC      490.
##   2 BQN      487.
##   3 SJU      486.
##   4 HNL      484.
##   5 PSE      481.
##   6 STT      479.
##   7 LAX      453.
##   8 SAN      451.
##   9 SMF      451.
## 10 LGB      450.
## # ... with 95 more rows
```

This becomes challenging code. It creates several data frames that we are not interested in (e.g. `lightSpeedDF2`) and is difficult to read. Alternatively, we can leverage the chain operator, `%>%`. This operator inserts an R object as the first argument of a function. In mathematical terms, x `%>%` $f(y)$ is interpreted as $f(x, y)$ - the x gets inserted as the first argument of the function. In other words, instead of continually writing functions of the form:

As an example of the chaining operator in mathematical notation, assume $f(x, y) = 2 * x + y^2$. Then, $f(1, 3) = 2 * 1 + 3^2 = 11$. To write this with chaining, we have 1 `%>%` $f(3)$. Notice that this would be different than 3 `%>%` $f(1)$; in this case, 3 `%>%` $f(1) = f(3, 1) = 7$.

```
newDF1 = filter(oldDF, arguments1)
newDF2 = arrange(newDF1, arguments2)
```

We can now go directly to the data frame we want:

```
newDF2 = oldDF %>%
  filter(arguments1) %>%
  arrange(arguments2)
```

without all of the intermediate data frames being created.

To find the fastest average flight speed destination:

```
flights %>%
  select(distance, air_time, dest) %>%
  mutate(speed = distance / air_time * 60) %>%
  group_by(dest) %>%
  summarize(avgSpeed = mean(speed, na.rm = TRUE)) %>%
  arrange(desc(avgSpeed))
```

and we can see this is much more succinct than the original method without chaining:

```
## # A tibble: 105 x 2
##    dest  avgSpeed
##    <chr>    <dbl>
##  1 ANC       490.
##  2 BQN       487.
##  3 SJU       486.
##  4 HNL       484.
##  5 PSE       481.
##  6 STT       479.
##  7 LAX       453.
##  8 SAN       451.
##  9 SMF       451.
## 10 LGB       450.
## # ... with 95 more rows
```

Exercise 6.4. Use the chaining operator, `%>%` to find which of the New York City airports experience the highest average departure delay.

6.11 Cheatsheets And Some Variants of The Five Verbs

Tip: Print out the Data Transormation Cheat Sheet and place the pages on the wall behind your computer. [Data Transofrmation Cheat Sheet][`https://github.com/rstudio/cheatsheets/raw/master/data-transformation.pdf`]

RStudio has done a wonderful job consolidating the most useful `dplyr` workflows into a two-page cheatsheet (see the "Data Transformation Cheat Sheet" at (`https://www.rstudio.com/resources/cheatsheets/`)). While for teaching purposes, there are five main verbs and one helper verb (i.e. `group_by()`), the cheat sheet reveals that there are some variants of those verbs; these provide shortcuts to common workflows. A table of commonly used variants is given here:

Primary Verb	Variant	Variant Description
`filter()`	`distinct()`	Remove all duplicate rows
`filter()`	`sample_n()`	Keep a random sample of `n` rows. Specify `n` using size argument (e.g. `flights %>% sample_n(size = 10)`).
`filter()`	`sample_frac()`	Keep a random sample of `frac` rows. Specify `frac` using size argument (e.g. `flights %>% sample_frac(size = 0.5)`).
`filter()`	`top_n`	Keep the top `n` rows from each group based on `wt`. (e.g. `lightSpeeedDF4 %>% top_n(n = 3, wt = speed)`)
`group_by()`	`ungroup()`	Remove all groupings from a data frame.
`mutate()`	`rename()`	Renames a column.

and more variants can be found on the "Data Transformation Cheat Sheet".

6.12 Getting Help

Users are encouraged to browse the resources available at (`https://dplyr.tidyverse.org/`) and (`http://r4ds.had.co.nz/`). When using google or youTube, make sure your search term includes the word `dplyr`. For example, searching for "selecting rows using `dplyr`" is preferable to "selecting rows in r". In R, there is often ten different ways to do the same thing. This book attempts to show one way that works well and is easier to do. Often we will find that the best way to do things is to use packages from the tidyverse (`https://www.tidyverse.org/`) - these include `dplyr` and `ggplot2` (for visualization) along with others that will make our lives easier.

6.13 Hadley Wickham

Tip: Add the term *Wickham* to any google search regarding data manipulation or visualization in R. It will likely improve the results!

When it comes to making R accessible for business analytics, one of the most influential contributors to this open source project has been Hadley Wickham (`http://hadley.nz/`). He is Chief Scientist at RStudio and according to his website, he "[builds] tool that make data science easier, faster, and more fun." I could not agree more, thanks Hadley!

7

dplyr: Data Manipulation For Insight

I just got word that the CEO of ZappTech is thinking about hiring our consulting firm. Apparently, his category managers are refusing to talk to one another; acting as if the four product categories are isolated kingdoms.

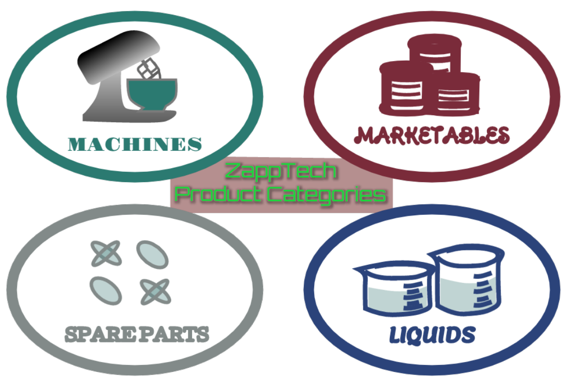


Figure 7.1: Are ZappTech's product categories sharing the same service level standards?

He is convinced that ZappTech's customers shop across multiple categories and thinks they expect the same level of customer service regardless of the product categories represented on their order. Since he doesn't trust his own team to put effort towards integrating management of the categories, the CEO has provided us data and asked us to investigate two questions: 1) Does service level (measured by on-time shipments) vary across product categories? and 2) how often do orders include products from more than one product category.

We will use provided data (available from the `causact` package) to answer the CEO's questions. In the process, we will learn more about data manipulation. The previous chapter presented tidy data frames as the starting point for data manipulation. In this chapter, the data is given in a slightly less than ideal format - we will learn to overcome this hurdle with some computational tricks, further exploring `dplyr` functions, and introducing the `lubridate` package for working with dates/times.

To answer the CEO's questions, we will approach the data analysis in four phases:

1. Data Loading: Make the data available in an data frame with all columns assoicated with the correct column class .
2. Lateness Calculation: In this phase, we will learn about the `lubridate` package in R and more rigourously define how to measure lateness.
3. Bring in product category information: In this phase, we will learn to merge the delivery information with the product category information using the join capabilities of dplyr.
4. Answer the CEO's questions: Does service level vary by product category? Do we ship items from multiple product categories?

Commonly used column classes include character, numeric, integer, and date

7.1 Data Loading and Cleaning

Data loading begins by getting the data into your R environment. Since the data is built into the `causact` package, we just need to load the package and then access the data.

The data from the CEO is in two data frames built into the `causact` package that accompanies this book: `delivDF` and `prodLineDF`. The following command will load the `delivDF` data into your environment.

Two very common data file formats in which data is exchanged are comma-delimeted files (`.csv`) and Microsoft Excel files (`.xlsx`). For importing and exporting `.csv` files, use the `read_csv()` and `write_csv()` functions from the `readr` package. For Excel files use the `read_excel()` and `write_excel` functions from the `readxl` package. Both packages are part of the tidyverse collection of packages (`https://www.tidyverse.org/packages/`).

```
# make the causact package available in this R session
library("causact")

# uncomment below line to show datasets that are part of the package
# data(package = "causact")

# load/unhide the dataset from the causact package
data("delivDF")
```

Remember, to install the `causact` package, go to (`https://github.com/flyaflya/causact`) and follow the installation instructions. Lubridate is available on CRAN and can be installed in the usual way.

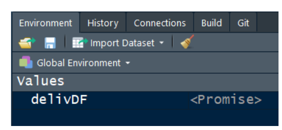


Figure 7.2: The delivDF dataframe details are not shown yet. The `<promise>` is an indication that once you try to use this R object, then and only then, will R load the object into the environment.

Figure 7.2 shows the updated environment tab in RStudio after running the above. Notice that it remains unclear what is contained within the `delivDF` data frame; the `<Promise>` description feels incomplete. This is due to something called *lazy loading* which refrains from completely loading the object into memory until the object is used. Thus, to see the details of `delivDF`, we just need to use it somehow. One way is to access the object by name - which prints the object to the `console` pane:

```
delivDF
```

```
## # A tibble: 117,790 x 5
##    shipID plannedShipDate actualShipDate
```

```
##     <chr>  <chr>          <chr>
##   1 10001  11/6/2013      10/4/2013
##   2 10002  10/15/2013     10/4/2013
##   3 10003  10/25/2013     10/7/2013
##   4 10004  10/14/2013     10/8/2013
##   5 10005  10/14/2013     10/8/2013
##   6 10006  10/14/2013     10/8/2013
##   7 10007  10/14/2013     10/8/2013
##   8 10008  10/14/2013     10/8/2013
##   9 10008  10/14/2013     10/8/2013
##  10 10008  10/14/2013     10/8/2013
## # ... with 117,780 more rows, and 2 more
## #   variables: partID <chr>, quantity <int>
```

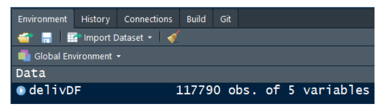


Figure 7.3: (ref:envData2

Notice that the environment tab has been updated (see Figure 7.3) to tell us that this is a data frame consisting of 117,790 observations and 5 variables. When we printed the tibble object to the console, the first 10 observations get printed and we see the class of the five columns. To the experienced eye, one notes that `plannedShipDate` and `actualShipDate` are `character` class objects (i.e. `<chr>`). As R novices, this is understandably overlooked, but it worth learning now that the `class` of an object determines what we can do with it. For example, the `character` class usually stores text information, also known as a *string* in computer lingo. As such, the following command trying to see the difference in planned versus ship dates for the first observation lacks meaning; afterall, what does it mean to subtract one word from another?:

```
delivDF$actualShipDate[1] - delivDF$plannedShipDate[1]
```

```
## Error in delivDF$actualShipDate[1] - delivDF$plannedShipDate[1]: non-numeric argument to binary opera
```

The resulting error suggests that R expects numbers to be subtracted; since text is not a number, there is no logical way to make this function work. So, we need another function to convert from character class to something that recognizes dates. And consistent with a theme we have been learning, we just need to find the right function; as usual the right function is in a package from the tidyverse. In this case, we use the `ymd` function from the `lubridate` package.

Getting R to agree that your data contains the dates and times you think it does can be tricky. `lubridate` simplifies that. Identify the order in which the year, month, and day appears in your character vector of dates. Now arrange the letters `y`, `m`, and `d` in the corresponding order. This arrangement is the name of the function in `lubridate` that will parse your dates. The dates in `delivDF` are given in month-day-year order; hence the `mdy` function will convert the column from character to date class. (See (`https://lubridate.tidyverse.org/`) for more details.)

```
# Uncomment this line to install the lubridate package
# install.packages("lubridate")
library("lubridate")

# Create new data frame to represent cleaned data
```

```
shipDF = delivDF
shipDF$plannedShipDate = mdy(shipDF$plannedShipDate)
shipDF$actualShipDate = mdy(shipDF$actualShipDate)

# Print updated tibble
shipDF
```

```
## # A tibble: 117,790 x 5
##    shipID plannedShipDate actualShipDate
##    <chr>  <date>          <date>
##  1 10001  2013-11-06      2013-10-04
##  2 10002  2013-10-15      2013-10-04
##  3 10003  2013-10-25      2013-10-07
##  4 10004  2013-10-14      2013-10-08
##  5 10005  2013-10-14      2013-10-08
##  6 10006  2013-10-14      2013-10-08
##  7 10007  2013-10-14      2013-10-08
##  8 10008  2013-10-14      2013-10-08
##  9 10008  2013-10-14      2013-10-08
## 10 10008  2013-10-14      2013-10-08
## # ... with 117,780 more rows, and 2 more
## #   variables: partID <chr>, quantity <int>
```

With date classes in place, we can now take advantage of date arithmetic and `lubridate` functions. For example, how many days late was the first line item shipped?

```
shipDF$actualShipDate[1] - shipDF$plannedShipDate[1]
```

```
## Time difference of -33 days
```

The answer is it was not shipped late, it was actually shipped 33 days ahead of the planned ship date.

Other date operations include these:

```
# print today's date
# (or if seeing this in print,
# the date this page was last edited)
today()
```

```
## [1] "2018-10-14"
```

```
# create a date object
thisDay = today()
```

```r
# extract information about a date
year(thisDay)
```

```
## [1] 2018
```

```r
month(thisDay)
```

```
## [1] 10
```

```r
day(thisDay)
```

```
## [1] 14
```

```r
wday(thisDay)
```

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

```r
mday(thisDay)
```

```
## [1] 14
```

```r
yday(thisDay)
```

```
## [1] 287
```

We can add a second argument, label = TRUE, to display the name of the weekday (represented as an ordered factor):

Notice that the `wday` function returns a `factor` variable (i.e. its class is the `factor` class). Factors are used to describe categorical variables with a fixed and known set of levels. They are often tricky to deal with. For historical context, see here: (`https://simplystatistics.org/2015/07/24/stringsasfactors-an-unauthorized-biography/`)

```r
# Return day of the week using an interpretable label
wday(thisDay, label = TRUE)
```

```
## [1] Sun
## 7 Levels: Sun < Mon < Tue < Wed < ... < Sat
```

For more date functions, see the RStudio "Dates and Times Cheat Sheet" ((`https://github.com/rstudio/cheatsheets/raw/master/lubridate.pdf`)) and information about the `lubridate` package at the tidyverse website ((`https://lubridate.tidyverse.org/`)).

7.2 Lateness Calculation

Now that our data is loaded and cleaned, we want to determine whether a particular delivery of an order is late or not. Let's revisit the data we have regarding deliveries:

```
shipDF
```

```
## # A tibble: 117,790 x 5
##    shipID plannedShipDate actualShipDate
##    <chr>  <date>          <date>
##  1 10001  2013-11-06      2013-10-04
##  2 10002  2013-10-15      2013-10-04
##  3 10003  2013-10-25      2013-10-07
##  4 10004  2013-10-14      2013-10-08
##  5 10005  2013-10-14      2013-10-08
##  6 10006  2013-10-14      2013-10-08
##  7 10007  2013-10-14      2013-10-08
##  8 10008  2013-10-14      2013-10-08
##  9 10008  2013-10-14      2013-10-08
## 10 10008  2013-10-14      2013-10-08
## # ... with 117,780 more rows, and 2 more
## #   variables: partID <chr>, quantity <int>
```

and also revisit the CEO's first question:

1. Does service level (measured by on-time shipments) vary across product categories?

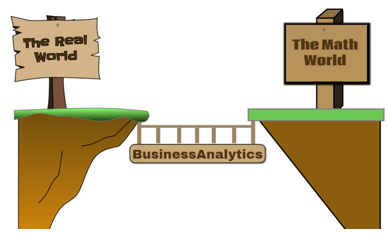


Figure 7.4: Time to traverse the business analytics bridge.

It is time to cross the business analytics bridge (Figure 7.4) Notice that the data we have refers to shipments and parts. Notice that the CEO's question talk about on-time shipments. We need to be mathematically precise in translating the CEO's real-world concerns to mathematical calculations; did he really mean shipments, or perhaps orders, or maybe even partID's? As an analyst, it is your job to *form an opinion* and *validate that opinion* with your stakeholder about how you plan to translate real-world concerns into mathematical constructs. **Do not immediately** fire off an email everytime you have a question; spend some time thinking and researching the issue before you make yourself look silly by asking simplistic questions that waste time. Also, when thinking about an issue, adopting the **customer's perspective** is often a good starting point.

After deliberating, forming an opinion, and validating that opinion, here is what is discovered about measuring lateness at ZappTech:

- The lateness calculation would ideally look at customer orders (i.e. `orderID`), but since we do not have that data and it is rare that an order gets broken into mulitple shipments, using `shipID` as the observational unit should give a good estimate/proxy of on-time order performance.

- Measuring lateness using quantity does not make sense for ZappTech. Some products, like latex gloves, get ordered by the hundreds whereas machines get ordered one or two at a time.
- Measuring lateness by `partID` might make sense for evaluating inventory policies on specific parts, but for now talking about lateness by `shipID` is preferable.
- For each unique shipID, if `actualShipDate > plannedShipDate`, then the shipID is considered late. (Note: it has been verified that each shipID has one and only one actualShipDate).

With these assumptions about measuring lateness, we can now rigorously define what it means to be late in mathematical and computational terms.

7.2.1 *Using dplyr to compute lateness*

Each line in `delivDF` represents a unique `shipID/partID` combination. Since the `partID` information is unnecessary, we create a new data frame to isolate just the shipment information:

```
# load dpyr package for select() and distinct()
library("dplyr")

# create new data frame for just shipID date info
shipDateDF = shipDF %>%
  select(shipID,plannedShipDate,actualShipDate) %>%
  distinct()  ## get unique rows to avoid double-counting

shipDateDF
```

```
## # A tibble: 23,339 x 3
##    shipID plannedShipDate actualShipDate
##    <chr>  <date>          <date>
##  1 10001  2013-11-06      2013-10-04
##  2 10002  2013-10-15      2013-10-04
##  3 10003  2013-10-25      2013-10-07
##  4 10004  2013-10-14      2013-10-08
##  5 10005  2013-10-14      2013-10-08
##  6 10006  2013-10-14      2013-10-08
##  7 10007  2013-10-14      2013-10-08
##  8 10008  2013-10-14      2013-10-08
##  9 10009  2013-10-14      2013-10-08
## 10 10010  2013-10-14      2013-10-08
## # ... with 23,329 more rows
```

Now, add a column to capture lateness.

`ifelse()` is a function from base R (i.e. no need to load a package). The function tests a logical condition in its first argument. If the test is `TRUE`, `ifelse()` returns the second argument. If the test is `FALSE`, `ifelse()` returns the third argument. The function is vectorized - it takes a vector as input and outputs a vector. In contrast, an aggregate function (like `sum()`) will take a vector of input and output a scalar (i.e. a single element). For more on vectorized functions, see (`http://www.noamross.net/blog/2014/4/16/vectorization-in-r--why.html`).

```
shipDateDF = shipDateDF %>%
  mutate(lateFlag = ifelse(actualShipDate > plannedShipDate,TRUE,FALSE))

shipDateDF
```

```
## # A tibble: 23,339 x 4
##    shipID plannedShipDate actualShipDate
##    <chr>  <date>          <date>
##  1 10001  2013-11-06      2013-10-04
##  2 10002  2013-10-15      2013-10-04
##  3 10003  2013-10-25      2013-10-07
##  4 10004  2013-10-14      2013-10-08
##  5 10005  2013-10-14      2013-10-08
##  6 10006  2013-10-14      2013-10-08
##  7 10007  2013-10-14      2013-10-08
##  8 10008  2013-10-14      2013-10-08
##  9 10009  2013-10-14      2013-10-08
## 10 10010  2013-10-14      2013-10-08
## # ... with 23,329 more rows, and 1 more
## #   variable: lateFlag <lgl>
```

And now, take advantage of the fact that R treats logical (`TRUE/FALSE`) values as numbers when used with numeric functions. `TRUE` is converted to 1 and `FALSE` converted to 0. Thus, as a simple example of this, we have:

```
# make a vector of 2 - TRUE values and 3 FALSE values
logicalVector = c(TRUE,TRUE,FALSE,FALSE,FALSE)

## return # of TRUE values
sum(logicalVector)
```

```
## [1] 2
```

```
## return proprtion of TRUE values
mean(logicalVector)
```

```
## [1] 0.4
```

```
## coerce logical vector to numeric vector
as.numeric(logicalVector)
```

```
## [1] 1 1 0 0 0
```

For calculating late shipments, the following code collapses the data on 23,339 shipments into two rows: one for on-time shipments (i.e. `lateFlag = FALSE`) and one for late shipments (i.e. `lateFlag = TRUE`):

```
shipDateDF %>%
  group_by(lateFlag) %>%
  summarize(countLate = n()) %>%
  mutate(proportion = countLate / sum(countLate))
```

```
## # A tibble: 2 x 3
##   lateFlag countLate proportion
##   <lgl>        <int>      <dbl>
## 1 FALSE        21399     0.917
## 2 TRUE          1940     0.0831
```

where the last `mutate` seems almost magical, but amazingly works.** We now have a lateness calculation complete, 8.31% of shipments are being delivered later than planned.

** To calculate the new `proportion` column, the 2-element `countLate` vector (i.e. 21399,1940) is divided by the aggregated 1-element `sum(countLate)` vector (i.e. 21399+1940). In R, when two unequal length vectors are arithmetically combined, the shorter vector is recycled so that it has the same length as the longer vector. Thus, `c(1,2,3) + c(4,5) = c(1+4,2+5,3+4) = c(5,7,7)`. See (`http://r4ds.had.co.nz/vectors.html#scalars-and-recycling-rules`) for more info. And in this case, we get two elements returned (i.e. 21399 / (21399+1940) , 1940 / (21399+1940)).

7.3 Bringing in Product Category Information

The information contained in `delivDF` did not include product category information. This information happens to be in another table:

```
library("causact")
data(prodLineDF)
prodLineDF
```

```
## # A tibble: 12,026 x 3
##    partID      productLine prodCategory
##    <chr>       <chr>       <chr>
##  1 part0a7f7c6 line7a      Machines
##  2 part84778b6 line7a      Machines
##  3 part330b1c9 line6d      Machines
##  4 parta4ebc9b line6d      Machines
##  5 partcf299b0 line6d      Machines
##  6 partfbc80a  line6d      Machines
##  7 partc986d3f line6d      Machines
##  8 part38c7896 line6d      Machines
##  9 partc39b72f line6d      Machines
## 10 partd8ab54  line6d      Machines
## # ... with 12,016 more rows
```

So now, we want to calculate lateness by product category, but the product category information is in `prodLineDF` and the actual/planned shipment data is in `delivDF`. How might we combine the information from these two tables?

In `dplyr`, there are many ways to integrate two data frames, we will focus on the one we need, called a *left join*. For the moment, let's ignore our need to combine shipping data with product category information and just learn about how a *left join* works.

7.3.1 A Left Join

For more information on methods of joining data frames, see (`http://r4ds.had.co.nz/relational-data.html#mutating-joins`)

The `left_join()` function includes all observations from one data frame and appends matching columns from another data frame. This is a commonly used join because it ensures you don't lose observations from your primary data frame. Let's see this in action using two simple data frames; one contains job title information and the other contains hourly salary:

```
employeeDF = tibble(name = c("Adam","Bob","Charlie"),
                    title = c("Server I","Innkeeper III","Server II"))
employeeDF
```

```
## # A tibble: 3 x 2
##   name    title
##   <chr>   <chr>
## 1 Adam    Server I
## 2 Bob     Innkeeper III
## 3 Charlie Server II
```

```
salaryDF = tibble(
  title = c("Server I","Server II", "Server III","Innkeeper I","Innkeeper II",
            "Innkeeper III","Bartender I","Bartender II"),
  hourlySalary = c(11,14,17,21,26,32,12,13)
  )
salaryDF
```

```
## # A tibble: 8 x 2
##   title         hourlySalary
##   <chr>                <dbl>
## 1 Server I               11.
## 2 Server II              14.
## 3 Server III             17.
## 4 Innkeeper I            21.
## 5 Innkeeper II           26.
## 6 Innkeeper III          32.
## 7 Bartender I            12.
## 8 Bartender II           13.
```

Given these two tables, we can find the hourly salary for each of the three employees using `left_join()`, a `dplyr` function. One could write `left_join(employeeDF,salaryDF)`, but with experience you will find it more elegant and intuitive to use the chaining operator from `dplyr` as shown:

```
employeeDF %>% left_join(salaryDF)
```

```
## # A tibble: 3 x 3
##   name    title          hourlySalary
##   <chr>   <chr>                 <dbl>
## 1 Adam    Server I                11.
## 2 Bob     Innkeeper III           32.
## 3 Charlie Server II               14.
```

Behind the scenes, `dplyr()` knows to combine the data frames based on any commonly labeled column names. In the above example, this was the `title` column; for all records in `employeeDF` append the columns of `salaryDF` by using the `title` column for matching rows of the data frames. Arguments to the `left_join()` function can be used to further control this behavor (see (`http://r4ds.had.co.nz/relational-data.html#join-by`)).

7.3.2 Combining Shipment Data With Product Category Data

Having learned to do a `left_join()`, we are equipped to get product category information and shipment information into one data frame. The first data frame, the primary one, will be shipment information since we want to know about *shipped* partID's, not necessarily all the partID's in `prodLineDF`.

You will notice that some of the product category information is listed as `NA`. In R, missing values are represented by the symbol NA (not available) . Impossible values (e.g., dividing by zero) are represented by the symbol `NaN` (not a number). These `NA`'s do have meaning though. Specifically, these are `partID`'s in the original data that we do not know the product category for.

```
catDF = shipDF %>%
  left_join(prodLineDF) %>%
  # NA prodCategory are for partID's that are
  # note really parts.  Used for shipping or
  # service fees, so we can safely get rid of them
  filter(!is.na(prodCategory))

catDF
```

```
## # A tibble: 98,207 x 7
##    shipID plannedShipDate actualShipDate partID      quantity productLine prodCategory
##    <chr>  <date>          <date>         <chr>          <int> <chr>       <chr>
##  1 10001  2013-11-06      2013-10-04     part92b16c5        6 line4b      Machines
##  2 10002  2013-10-15      2013-10-04     part66983b         2 linea3      Marketables
```

```
##   3 10003  2013-10-25      2013-10-07     part8e36f25        1 line90      Machines
##   4 10004  2013-10-14      2013-10-08     part30f5de0        1 linea3      Marketables
##   5 10005  2013-10-14      2013-10-08     part9d64d35        6 line9b      Machines
##   6 10006  2013-10-14      2013-10-08     part6cd6167       15 linec1      Marketables
##   7 10007  2013-10-14      2013-10-08     parta4d5fd1        2 line55      SpareParts
##   8 10008  2013-10-14      2013-10-08     part08cadf5        1 line4b      Machines
##   9 10009  2013-10-14      2013-10-08     part5cc4989       10 linec1      Marketables
## 10 10010  2013-10-14      2013-10-08     part912ae4c        1 line4b      Machines
## # ... with 98,197 more rows
```

7.4 Answering the CEO's Questions

With all the data we need to answer questions in one data frame, we proceed.

7.4.1 1) Does service level (measured by on-time shipments) vary across product categories?

```
catDF %>%
  select(shipID, plannedShipDate,actualShipDate,prodCategory) %>%
  distinct() %>% ##only maintain one row per shipID/prodCategory combination
                 ##otherwise, you will have one row per shipID/partID combo
  mutate(lateFlag = ifelse(actualShipDate > plannedShipDate,TRUE,FALSE)) %>%
  group_by(prodCategory,lateFlag) %>%
  summarize(countLate = n()) %>%
  mutate(proportion = countLate / sum(countLate)) %>%
  arrange(lateFlag,proportion)
```

```
## # A tibble: 8 x 4
## # Groups:   prodCategory [4]
##   prodCategory lateFlag countLate proportion
##   <chr>        <lgl>        <int>      <dbl>
## 1 Machines     FALSE         3124     0.810
## 2 SpareParts   FALSE         4561     0.842
## 3 Liquids      FALSE         6512     0.919
## 4 Marketables  FALSE        13910     0.919
## 5 Marketables  TRUE          1222     0.0808
## 6 Liquids      TRUE           575     0.0811
## 7 SpareParts   TRUE           857     0.158
## 8 Machines     TRUE           732     0.190
```

From the above analysis, we find that there does seem to be discrepancies between on-time shipments by product category. Machines has the most late shipments (19%), SpareParts(15.8%) is next, and the remaining two, Liquids (8.1%) and Marketables (8.1%) have similar performance.

7.4.2 *How often do orders include products from more than one product category?*

Now, we answer how often do orders(i.e. shipments) include products from other product categories using `group_by()` and `summarize()`?

```
# find # of categories included on each shipID
numCatDF = catDF %>%
  select(shipID, plannedShipDate,actualShipDate,prodCategory) %>%
  distinct() %>% ##only maintain one row per shipID/prodCategory combination
  group_by(shipID) %>%
  summarize(numCategories = n())

# print out summary of numCategories column
numCatDF %>%
  group_by(numCategories) %>%
  summarize(numShipID = n()) %>%
  mutate(percentOfShipments = numShipID / sum(numShipID))
```

```
## # A tibble: 4 x 3
##   numCategories numShipID percentOfShipments
##           <int>     <int>              <dbl>
## 1             1     17993             0.775
## 2             2      3261             0.141
## 3             3       842             0.0363
## 4             4      1113             0.0480
```

The answer to this second question is about 22.5% of orders contain more than one product category. So, in conclusion, 22.5% of orders have more than one product category on them and yes, it does seem that the product categories are managed differently.

> Recommendation to the CEO: Machines and SpareParts have different performance characteristics. The assumption of whether this is bad or good should be validated, but if ZappTech wants to achieve a uniform service level across product categories than more work should be done. Hire our team and we can help dig deeper!

7.5 *Notes about Data Wrangling from Twitter*

Data cleaning/wrangling/munging is the process of transforming raw data into a more valuable and useful format. In real-world data analysis, much of your time will be spent doing this. While consuming this textbook, much of the data will be presented in a format that is amenable for analysis; the real-world is much less thoughtful in this regard. Please see this collection of tweets to give you an idea about real-world data wrangling:

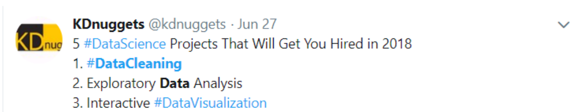


Figure 7.5: Some insightful tweets about data cleaning/wrangling.

Figure 7.6: Some insightful tweets about data cleaning/wrangling.

Figure 7.7: Some insightful tweets about data cleaning/wrangling.

Do not be surprised or think that you are wasting time when data wrangling. This is a necessary - albeit frustrating at times - part of the BAW.

7.6 *Getting Help*

For dealing with factors, see McNamara and Horton (2018). For more notes on logical vectors see (`https://bookdown.org/ndphillips/YaRrr/logical-indexing.html`).

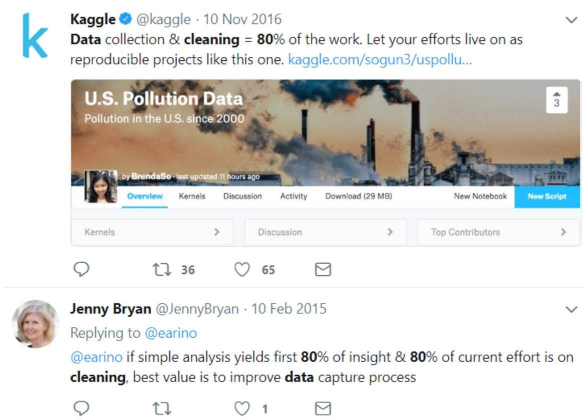


Figure 7.8: Some insightful tweets about data cleaning/wrangling.