Analyzing Data

Ben Schmidt

2019-01-29

Introduction

Now that we know how to read in data, we are finally ready to analyze it.

Data "analysis," of course, can mean many different things. That's the whole point of this clalss! The analyses we'll conduct to begin with are extremely simple; they mostly include counting, grouping, and summarizing. Later in this course we'll come to more complicated operations. But in fact, simple algorithms of counting have major advantages.

1. **You** can understand them. There is a "black box" quality to many of the more advanced tools we'll be looking at later in the course: "summary statistics" are much easier to correct and change on the fly. They use, for the most part, math you learned in elementary school.
2. Your **readers** can understand them. It is a burden to have to spend five minutes explaining your algorithms at every talk, and when your work makes a field-specific contribution, you want scholars to focus on your argument and evidence, *not* on whether they trust your algorithm.

The pipeline strategy for exploratory data analysis.

Our core strategy will be a **pipeline** strategy where a single `tibble` object is passed through a series of saved steps until it gives you useful results.

`dplyr` exposes a number of functions that make this easy. But although they are more coherently bundled in this package, they are shared by a wide variety of data manipulation software. In particular, they closely resemble the operations of SQL, the most important data access language of all time.[1]

I'll be introducing these statements one by one, but first a refresher on the basic idea.

The "pipe" operator.

We're making heavy, heavy use of a new feature of R in this class: the so-called "pipe" operator, `%>%`. (The name of the package containing it is "magrittr - Ceci n'est pas un pipe," but for our purposes it's fine to call it a pipe.)

The idea of the pipe is to represent each transformation of data you make as a chain. If you wanted to define a process where you took a number, added 2, then multiplied by 6, and finally divided by ten. You could express that as a set of functions:

```
library(magrittr)

start = 1
x = multiply_by(start,6)
y = add(x,2)
```

[1] Note: if you have a slow computer or an extremely large data set, you can do all of these operations on data on disk. I recommend using `sqllite3` as the on-disk data storage engine: read the documentation at `?src_sqlite` for an explanation of how to read it in.

```
z = divide_by(y,10)
z
```

But you could also nest them inside each other. This requires you to read the functions from the inside-out: it's awkward.

```
divide_by(add(multiply_by(1,6),2),10)
```

With `magrittr`, you can use pipes to read the operation simply from left to right.

```
1 %>% multiply_by(6) %>% add(2) %>% divide_by(10)
```

These expressions can get quite long. Formatting your code will make it substantially more readable. There are a few guidelines for this:

1. Each line must end with a pipe-forward operator: if R hits a linebreak with a syntactically complete
2. If you pipe an operation to a function that has no arguments, you may omit the parentheses after the function.

So you might prefer to write the above in the following form. (Note that RStudio will automatically indent your code for you.)

```
1 %>%
  multiply_by(6) %>%
  add(2) %>%
  divide_by(10)
```

Filtering

"Filtering" is the operation where you make a dataset smaller based on some standards.

The ==,>,and < operators

The easiest way to filter is when you know exact which value you're looking for.

Passing data through the filter operator reduces it down to only those entries that match your criteria. In the previous section, you have noticed that the `summary` function on crews showed a single 82-year-old. By filtering to only persons aged 82, we can see who exactly that was.

```
crews = read_csv("../data/CrewlistCleaned.csv")
crews %>% filter(Age==82)
```

If we wished to case a slightly wider net, we could filter for sailors over the age of 70:

```
crews %>% filter(Age>65)
```

== and =

Warning: one of the *most* frequent forms of errors you will encounter in data analysis is confusing the `==` operator with the `=` one.

> Advanced note: in the early history of R, you *could not* in fact assign a variable by using the `=` sign. Instead, you would build an arrow out of two characters: `president <- "Washington"` This form of assignment is still used in R. If you're the sort of writer who sometimes starts sentences without having figured out the final clause, it can even be handy, because they can point in both directions: `"Washington" -> president Know to recognize this code when it appears. But most computer languages use=for assignment and==for equality, and`

so R now follows suit. I find it easier time coding if you just use=all of the time. Super-esoteric note: There is actually a third assignment operator, which introductory programmers will almost never encounter: the<<-‘ assigner, which makes variable assignments that move outside their home function.

The %in% operator

Almost as useful for humanities computing as == is the special operator %in%. == tests if two values are the same: %in% tests if the left hand sign is *part of* the right hand side. So for example, see the results of the following operations.

```
"New York"==c("Boston","New York","Philadelphia")

"New York" %in% c("Boston","New York","Philadelphia")
```

In the crews dataset, we could search to find, for example, any combination of names of interest.

```
fabFour = crews %>% filter(LastName %in% c("McCartney","Lennon","Harrison","Starr"))
fabFour %>% count(LastName)
```

In practice, %in% is most often useful for checking very large lists you already have loaded in.

Arranging

Frequently useful with filter is the function arrange. It *sorts* data. Using the head function from last week, we can, for example, first limit to ships that sailed before 1860, and then show the youngest individuals.

```
crews %>% filter(date<1860) %>% arrange(Age) %>% head
```

If you want to sort in descending order, to see the oldest, you can use the dplyr function desc to reverse the variable: but usually it's easiest to just put a negative sign in front of the variable you want sorted.

```
crews %>% filter(date < 1860) %>% arrange(Age %>% desc) %>% head
```

Summarizing

Looking at your individual data is sometimes sufficient: but usually, you want to know some aggregate conditions about it.

dplyr provides the summarize function to do this. Unlike the summary function we saw earlier, summarize doesn't do anything on its own; instead, it lets you specify the kinds of commands you want to run.

In dplyr, you pipe your results through to a summarize function and then run a *different function call* inside the parentheses. The simplest function call is n: it says how many rows there are. To run it, we use the function **summarize**, which reduces the whole dataset down.

```
crews %>% summarize(n = n())
```

But you can use any of the variables inside the frame as part of your function call: for instance, to find the average age in the set, you could run the following.

```
crews %>% summarize(mean_age = mean(Age))
```

This produces an error, because there is **missing data**. R provides a variety of ways of dealing with missing data: in dplyr, we can just put in a filter operation to make sure that we get the values we want back.

```
crews %>% filter(Age>0) %>% summarize(mean_age = mean(Age))
```

This looks like a single variable, but it's actually a frame. By using = to assign values inside summarize, we can summarize on a variety of statistics: average, mean, or oldest age.

```
crews %>%
  filter(Age>0) %>%
  summarize(
    average_age = mean(Age),
    median_age=median(Age),
    oldest_age=max(Age))
```

Finding functions for your task.

There are a bunch of functions you maybe haven't seen before here: `mean`, `median`, and `max`. From their names it should be clear what they do.

Some languages (Python, for example) work to reduce the number of functions in the set. R is not like these: it has so many functions that even experienced users sometimes stumble across ones they have not seen before. And libraries like `dplyr` provide still more.

So how do you find functions. The best place to start is by typing `??` into the console and then phrase you're looking for.

A few functions from base R that may be useful include:

- `length`
- `unique`
- `rank`
- `min`

And the various type-conversion functions:

- `as.character`
- `as.numeric`

The other way is by Googling. One of R's major flaws is that the name is so generic that it's hard to Google. The website "Stack Overflow" contains some of the most valuable information.

`group` and `summarize`.

The last element which enables all sorts of amazing analysis is **grouping**. Now that you know how to filter and summarize, you're ready for the most distinctive operation in dplyr: `group_by`. Unlike `filter` and `summarize`, `group_by` **doesn't change the data.** Instead, it does something more subtle; as it says, it *groups* the data for future operations.

In other words, it sets the units that you'll be working with. In Witmore's terms, it changes the *level of address* for the text.

The most basic idea is to use a grouping and then the `n` function to count the number of items in each bucket.

```
crews %>%
  group_by(Skin) %>%
  summarize(count=n()) %>%
  arrange(-count)
```

This is actually so common that there's a function, "count", that does it directly. But I recommend sticking with the more basic format for now, because it gets you in the habit of actually thinking about what happens.

```
crews %>%
  count(Skin)
```

You can group by multiple items.

```
crews %>%
  group_by(Skin, Hair) %>%
  summarize(count=n()) %>%
  arrange(-count)
```

Here is a very interesting point about the social construction of race, entirely in code. Try to figure out what it does! Try to understand what this code does, and what the data says!

Note that there's a new function in here, `mutate`. Like `summarize`, that adds a new column (or changes an existing one).

```
crews %>%
  group_by(Skin) %>%
  filter(!is.na(Skin)) %>%
  mutate(skin_total = n()) %>%
  filter(skin_total > 100) %>% # Why?
  group_by(Skin, Hair, skin_total) %>% # Why is `skin_total` here?
  summarize(combo_count=n()) %>%
  mutate(share = combo_count/skin_total) %>%
  arrange(-share) %>%
  filter(share > .1)
```

Read the line of code below. What does it do? Can you come up with any patterns or explanations for the results that you see here?

```
crews %>%
  filter(Age>0) %>%
  group_by(Skin) %>%
  summarize(averageAge=median(Age),count=n()) %>%
  filter(count>100) %>%
  arrange(averageAge)
```