Analyzing Data

Ben Schmidt

2015-01-27

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. 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 `data.frame` 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)
z = divide_by(y,10)
z
```

[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.

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 rules 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 %>% 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. You will probably have an easier time coding if you just use=all of the time. 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. We'll encounter this in the advanced functions section.

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"))
summary(fabFour)
```

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 oldest individuals.

```
crews %>% filter(as.numeric(year)<1860) %>% arrange(Age) %>% head
```

If you want to sort in descending order, 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(as.numeric(year)<1860) %>% arrange(-Age) %>% 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.

```
crews %>% summarize(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))
```

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))
```

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 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 you should know about 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.

Exercises: filtering and summarizing

1. Using summarize, find the minimum age in the set. (Remember, the function to find a minimum is `min`.)
2. Using `filter`: what is the name of that youngest person? When did he or she sail?

3. How many sailors left on Barks between 1850 and 1880? Chain together `filter` and `summarize` with the special `n()` function. Note that this has a number of different conditions in the filter statement. You could build several filters in a row: but you can also include multiple filters by separating them with commas. For instance, `filter(school=="Northeastern",year==2015)` might be a valid filter on some dataset (though not this one.)

4. Question 3 told you how many sailors left on barks in those years. How many distinct voyages left? The variable `Voyage.number` identifies distinct voyages in this set. (This may require reading some documentation: reach out to me or a classmate if you can't figure it out. There are at least two ways: one involves using the `dplyr` function `distinct` before summarizing, and the second involves using the functions `length` and `unique` in your call to `summarize`.)

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.

That makes the sort of summaries one can run much more useful. For example, suppose we want to know not the total number of sailors in the data set, but the number residing in each city. To do so, we set the grouping to `Residence`.

```
crews %>% group_by(Residence)
```

Now when we run a summary operation, it will give the counts. For example, to get the number of people residing in each town by descending order, we could run the following function.

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

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)
```

Select and mutate

One last useful pair of functions in dplyr are `select` and `mutate`. Select simply lets you specify which columns you want: mutate lets you *change* it. We'll get more into mutate later.

Filtering, Summarizing, and Grouping Exercises: Population data

**Note: these exercises are quite cumulative: you'll probably be building up a single chain over the course of the set. Make use of cut and paste!

1. Read in the data frame `tidied` that you made in the last problem set, and select only the columns for city, state, year, and population.

2. Because the original data was not tidy, you probably have a lot of zeroes in there. Remove them.

3. The `slice` operator is much like `head`, but lets you specify precisely the positions you want. `slice(1:10)` will give you the first ten rows of a frame; `slice(4)` will give you the fourth row of a frame.
   Using `slice`: What were the ten largest cities in the United States in 1820?

4. `slice` works *by group*. Using `slice` and `group_by`, what was the third largest city in the United States for every census year?

5. Define a medium-sized city as any city with over 10,000 population. What **state** had the most medium-sized cities in 1900? (Remember the `n()` function get the count inside a summary.)

6. Tricky: create a list of the **states** with the most medium-sized cities for each year from 1790 to 2010. (Note that `group_by` can use multiple variables at the same time. You can either use `rank` or `arrange` with `slice` after grouping by the appropriate variables. Note that arrange respects existing groups: you may need to use the `ungroup` function even after a `summarize` call.)

7. Boston has been the largest city in its state in every census. What other cities have been the largest in their state every census since 1790?

8. (Tough) What state has had the most different cities as its largest? What are they? (Although you can answer both these questions with a single pipeline, it might be easier to do it with two.)

9. (Tough) For question 8, you found states which had the same largest city since 1790. But most states weren't even in the USA in 1790. Find, instead, the states that have the same largest city for every appearance they make in the census lists.

Mutating