Code

STAT 209: Lab 13

Plotting With SQL

SQL vs `dplyr`

Now that you (are on your way to) know(ing) two different data wrangling languages (`dplyr` and `SQL`), it's worth spending a minute thinking about their relative strengths and weaknesses. Here are a few strengths of each one:

`dplyr`

1. Runs locally
2. Functional (Each wrangling verb has a clear input and output; whether this is a strength depends on your attitude toward functional programming, I suppose)
3. Integrated into the larger `tidyverse`
4. More flexible: can combine standard verbs with arbitrary R code, incorporate code in custom functions

`SQL`

1. More efficient for large data
2. Natural language syntax (Also a matter of taste; code is probably easier on the eyes, but I find it harder to pull apart than the functional style of R; but your mileage may vary)
3. More widely used(?) in industry (though I'm not completely sure if this is still true as far as *data-wrangling*. `dplyr` is much more popular now than it was even three or four years ago)

There are two big reasons we can't rely exclusively on SQL, however: (1) it doesn't support plotting, and (2) it doesn't support statistical modeling. So if we want to do more than show some tables, we need to be able to pass the results of our SQL queries back into R, so we can create graphs and (though we're not focusing on this in this class) models.

Goal

Use SQL together with `ggplot` to produce visualizations from large datasets.

In particular, we will try to verify the following claim from the FiveThirtyEight article here (https://fivethirtyeight.com/features/fastest-airlines-fastest-airports/):

“In 2014, the 6 million domestic flights the U.S. government tracked required an extra 80 million minutes to reach their destinations. The majority of flights – 54 percent – arrived ahead of schedule in 2014. (The 80 million minutes figure cited earlier is a net number. It consists of about 115 million minutes of delays minus 35 million minutes saved from early arrivals.)”

as well as to reproduce the graphic therein (shown below).

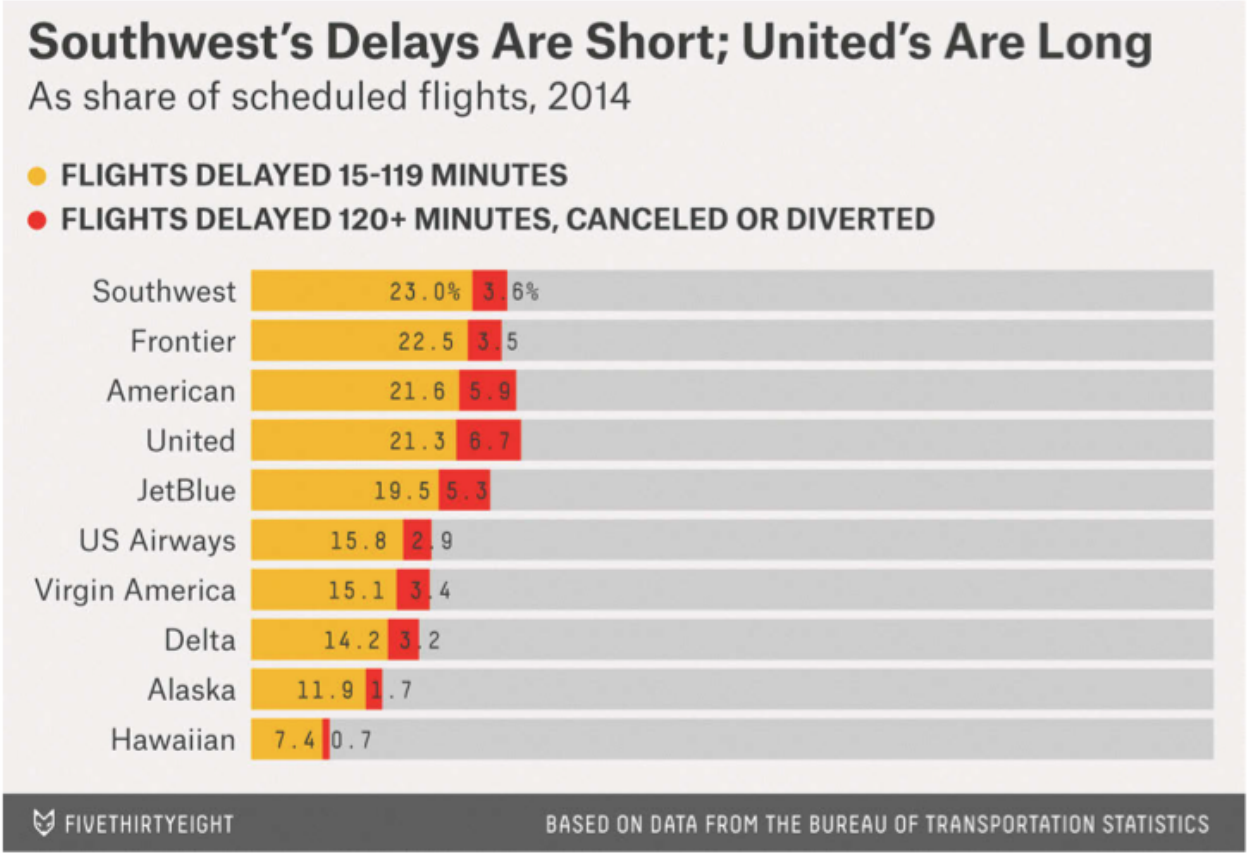


Figure: Reproduced from MDSR p. 291

Setup Boilerplate and Reference (Repeated from Lab 12b)

We need to load packages and set up a connection to the server again.

Code:

Code

For convenience here is the list of basic verbs again:

`SELECT` allows you to list the columns, or functions operating on columns, that you want to retrieve. This is an analogous operation to the `select()` verb in `dplyr`, potentially combined with `mutate()`.

`FROM` specifies the table where the data are.

`JOIN` allows you to stitch together two or more tables using a key. This is analogous to the `join()` commands in `dplyr`.

`WHERE` allows you to filter the records according to some criteria. This is an analogous operation to the `filter()` verb in `dplyr`.

`GROUP BY` allows you to aggregate the records according to some shared value. This is an analogous operation to the `group_by()` verb in `dplyr`.

`HAVING` is like a `WHERE` clause that operates on the result set—not the records themselves. This is analogous to applying a second `filter()` command in `dplyr`, after the rows have already been aggregated.

`ORDER BY` is exactly what it sounds like—it specifies a condition for ordering the rows of the result set. This is analogous to the `arrange()` verb in `dplyr`.

`LIMIT` restricts the number of rows in the output. This is similar to the R command `head()`, but somewhat more versatile.

Image Source: Baumer et al. *Modern Data Science with R*.

Remember: Verbs lower in the list must appear *after* verbs higher in the list when constructing queries.

Here's the `dplyr` to `SQL` translation summary again:

Concept	SQL	R

Filter by rows & columns	`SELECT col1, col2`
	`FROM a`
	`WHERE col3 = 'x'`
		`a %>%`
		`filter(col3 == "x")`
		`%>%`
		`select(col1, col2)`

Aggregate by rows	`SELECT id, sum(col1)`
	`FROM a`
	`GROUP BY id`
		`a %>%`
		`group_by(id) %>%`
		`summarize(sum(col1))`

Combine two tables	`SELECT *`
	`FROM a`
	`JOIN b ON a.id = b.id`
		`a %>%`
		`inner_join(b, by =`
		`c("id" = "id"))`

Table 12.1: Equivalent commands in SQL and R, where a and b are SQL tables and R `data.frames`.

Image Source: Baumer et al. *Modern Data Science with R*

And, it bears repeating, in all caps this time:

IMPORTANT: ALWAYS `LIMIT` YOUR QUERIES, LEST YOU TRY TO FETCH HUNDREDS OF MILLIONS OF RECORDS AND BREAK EVERYTHING FOR EVERYONE!

One last reminder: to designate a code chunk “SQL”, use `{sql connection=db}` in the chunk options (where `db` is whatever you named your connection in a previous R code chunk) in place of the `r` that is usually there.

Extracting the necessary summary statistics

Exercise 1 Before diving into writing any queries, examine the quote and the graph together with your group and work together to sketch out what the summarized data table(s) should look like to verify the quote and reproduce the graph. Note that flights are only classified as a “late arrival” if the delay is 15 minutes or more. What are the rows, what columns are needed; which of these correspond to columns in the `flights` data (use `DESCRIBE` to see them)? For columns that represent summary statistics, how can these be computed (write down the computation steps in sentences; not in code yet). Working backwards like this can be a good way to approach a wrangling+visualization problem.

Sample solution:

Code

Code

Exercise 2 Take the `flights` dataset, and write an SQL query to calculate the necessary summary information. A tip: you can use commands of the form `if(<condition>, <value if true>, <value if false>)` in SQL to create a variable that has one value if a condition is met and another value if it isn’t (the angle brackets indicate placeholders, and shouldn’t actually be typed). **Note: The result should just be one row, but include a `LIMIT` clause anyway, just in case the result isn’t what you intend. Note: I found that doing this on all the flights in 2014 takes a really long time, since new variables are being computed for each one. To save time, each member of your group can compute the quantities of interest for three or four days spread throughout 2014 (you might use your birthdays, though if they’re all in the same season this will skew the results), multiplying count variables by 365 to extrapolate to the year. The group can average together their results.**

Sample solution:

Code

1 records

num_flights	early_pct	min_late	min_early	net_delay

num_flights	early_pct	min_late	min_early	net_delay
5308560	0.2771	203.9667	-15.169	188.7977

My birthday is in the winter so there are more delays, but if you average, you should notice that the numbers still don't quite match what's in the quote, and not just because of the sampling error involved.

The total minutes early come close, but the total minutes late is way under what FiveThirtyEight reports. It turns out, when you read FiveThirtyEight's methodology, that *cancelled* flights have `arr_delay = 0` in the data, and so these aren't contributing to the statistics we've computed; but these flights obviously hold travelers up.

FiveThirtyEight did some modeling to estimate an `arr_delay` number for cancelled flights; hence the discrepancy. We won't try to reproduce what they did; instead as an approximation, we will consider cancelled flights to be delayed by 4.5 hours (following another quote in the article suggesting a "quick and dirty" estimate of 4-5 hours for each cancelled flight).

Exercise 3 Revise your query to add 270 minutes to each canceled flight.

Sample Solution:

Code

1 records

num_flights	early_pct	min_late	min_early	net_delay
5308560	0.2771	324.8876	-15.169	188.7977

Now let's create the dataset for the graph. We're going to need to pull the results into R, but let's first write the query in SQL to confirm that we get what we want.

Exercise 4 Write an SQL query to produce a data table with the summary statistics needed for the graph. Don't worry about the "gather" step to get the data into the "tidy" format; we'll do that in R later. Hint: for the labels, you'll want the `name` field from the `carriers` data table. Note that the graph is sorted in descending order by percentage of short delays.

Sample Solution:

Code

Displaying records 1 - 10

carrier	name	short_delay_pct	long_delay_pct
FL	AirTran Airways Corporation	0.5418	0.0605
WN	Southwest Airlines Co.	0.5392	0.2330
F9	Frontier Airlines Inc.	0.4128	0.5174

carrier	name	short_delay_pct	long_delay_pct
UA	United Air Lines Inc.	0.4035	0.2326
OO	SkyWest Airlines Inc.	0.3306	0.2141
US	US Airways Inc.	0.3292	0.0396
MQ	Envoy Air	0.3208	0.1615
VX	Virgin America	0.3195	0.0888
DL	Delta Air Lines Inc.	0.2931	0.0648
EV	ExpressJet Airlines Inc.	0.2808	0.2601

Bringing the result set into R

We're now done with the SQL part of the process!

Now that we have a small dataset, we can turn it into an R data frame and do our finishing wrangling touches in `dplyr` and our visualization in `ggplot2`.

Exercise 5 Once you have your SQL query working, create an R string that contains the text of the query. You can call it whatever you want; I will refer to this variable below as `query`.

Sample Solution:

Code

Exercise 6 Having defined the `query` string, create an R data frame that contains the relevant information with `db %>% dbGetQuery(query) %>% collect()`. The use of `collect()` here brings the actual data from the table into memory; not just a pointer to a "view" of the data on the remote server, so don't do this until you know that your query produces the small result set you want.

Sample Solution:

Code

```
##    carrier                       name short_delay_pct long_delay_pct
## 1       FL AirTran Airways Corporation          0.5418         0.0605
## 2       WN      Southwest Airlines Co.          0.5392         0.2330
## 3       F9      Frontier Airlines Inc.          0.4128         0.5174
## 4       UA       United Air Lines Inc.          0.4035         0.2326
## 5       OO       SkyWest Airlines Inc.          0.3306         0.2141
## 6       US             US Airways Inc.          0.3292         0.0396
## 7       MQ                  Envoy Air          0.3208         0.1615
## 8       VX             Virgin America          0.3195         0.0888
## 9       DL        Delta Air Lines Inc.          0.2931         0.0648
## 10      EV    ExpressJet Airlines Inc.          0.2808         0.2601
## 11      AA      American Airlines Inc.          0.2647         0.0487
## 12      B6             JetBlue Airways          0.2535         0.6416
## 13      AS        Alaska Airlines Inc.          0.1480         0.0072
## 14      HA      Hawaiian Airlines Inc.          0.0860         0.0054
```

Cleaning up the labels

Getting the airline names to display as they are in the graph will require some string manipulation. For one thing, we want to strip away the formal company identifiers like `Co.` and `Inc.`. Moreover, we don't really need the `Airlines` and `Airways` bits.

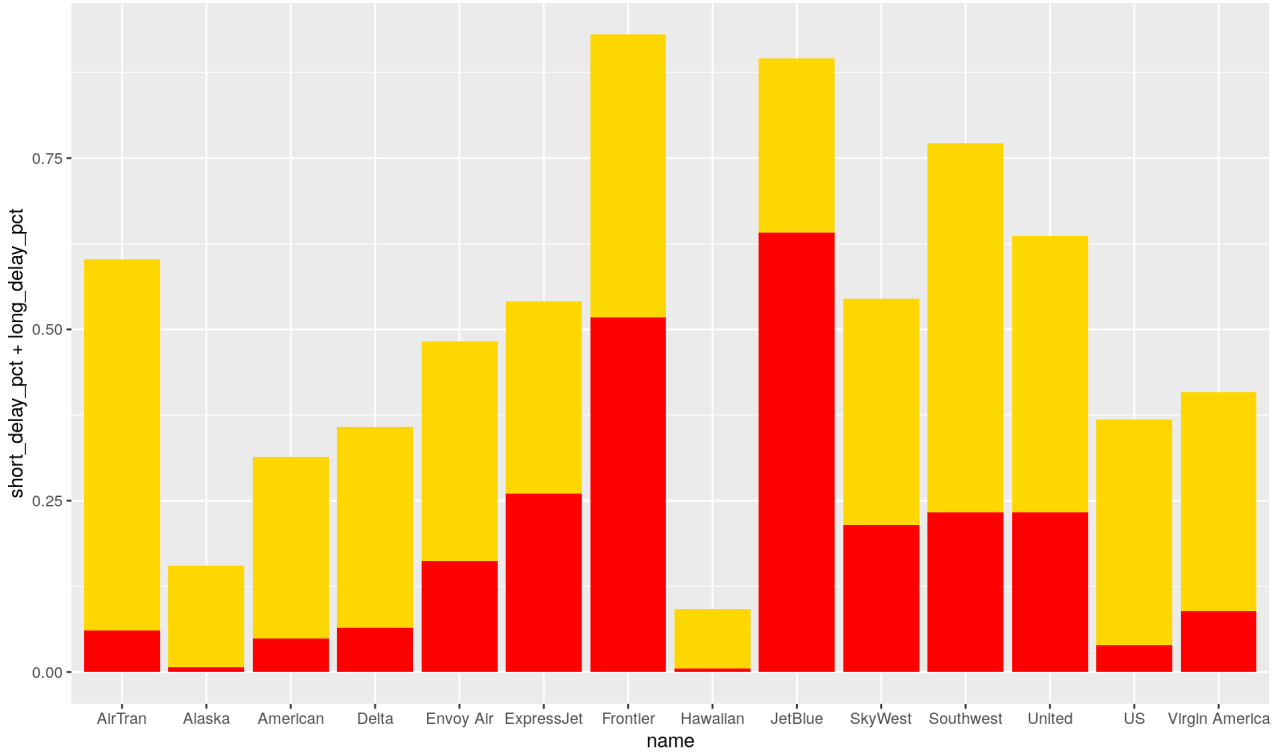
We can use the `gsub()` function to find these substrings and replace them. The syntax for this uses what's called "regular expressions"; essentially a pattern matching language that lets us define a variety of more or less general "wildcards". Here are the `gsub()` commands we want.

Code

```
##    carrier           name short_delay_pct long_delay_pct
## 1       FL        AirTran          0.5418         0.0605
## 2       WN      Southwest          0.5392         0.2330
## 3       F9       Frontier          0.4128         0.5174
## 4       UA         United          0.4035         0.2326
## 5       OO        SkyWest          0.3306         0.2141
## 6       US             US          0.3292         0.0396
## 7       MQ      Envoy Air          0.3208         0.1615
## 8       VX Virgin America          0.3195         0.0888
## 9       DL          Delta          0.2931         0.0648
## 10      EV     ExpressJet          0.2808         0.2601
## 11      AA       American          0.2647         0.0487
## 12      B6        JetBlue          0.2535         0.6416
## 13      AS         Alaska          0.1480         0.0072
## 14      HA       Hawaiian          0.0860         0.0054
```

You'll notice that you get more airlines than appear in the graph. We could do some consolidation, but we'll set this aside for now.

Code



`gather()ing` the percentages

For visualizing this data, we'll want to create a single `pct` column representing percentages of both delay categories, and a `delay_type` column indicating whether the percentage refers to short or long delays. This is a job for `gather()`.

Remember that `gather()` has a `key=` argument that gives the name of the new grouping variable; a `value=` column that gives the name of the new merged quantitative variable, and unnamed arguments listing the column headings in the "wide" data that we are merging.

Exercise 7 Execute an appropriate `gather()`

Sample Solution:

Code

```
##    carrier           name length delay_pct
## 1       HA       Hawaiian  short    0.0860
## 2       HA       Hawaiian   long    0.0054
## 3       AS         Alaska  short    0.1480
## 4       AS         Alaska   long    0.0072
## 5       B6        JetBlue  short    0.2535
## 6       B6        JetBlue   long    0.6416
## 7       AA       American  short    0.2647
## 8       AA       American   long    0.0487
## 9       EV     ExpressJet  short    0.2808
## 10      EV     ExpressJet   long    0.2601
## 11      DL          Delta  short    0.2931
## 12      DL          Delta   long    0.0648
## 13      VX Virgin America  short    0.3195
## 14      VX Virgin America   long    0.0888
## 15      MQ      Envoy Air  short    0.3208
## 16      MQ      Envoy Air   long    0.1615
## 17      US             US  short    0.3292
## 18      US             US   long    0.0396
## 19      OO        SkyWest  short    0.3306
## 20      OO        SkyWest   long    0.2141
## 21      UA         United  short    0.4035
## 22      UA         United   long    0.2326
## 23      F9       Frontier  short    0.4128
## 24      F9       Frontier   long    0.5174
## 25      WN      Southwest  short    0.5392
## 26      WN      Southwest   long    0.2330
## 27      FL        AirTran  short    0.5418
## 28      FL        AirTran   long    0.0605
```

Creating the graph

Most of the work is done now; all that remains is to actually produce the bar graph. Some tips here: don't forget to use `stat="identity"` with `geom_bar()`, since we have precomputed the percentages.

You may need to reorder the label variable so the airlines appear in descending order of short delays. You can do this with `reorder(<variable to sort>, <variable to sort on>, max)`. This iteratively applies the `max` function to the values of the `<variable to sort on>`, and then finds the matching value of the `<variable to sort>`, and puts it first in the order. Then it does it again with the values that remain. Etc.

Construct the barplot with the bars oriented vertically first, then use `+ coord_flip()` in your `ggplot` chain to turn it sideways.

Use `scale_y_continuous()` to set the limits on the "y" axis. Load `ggthemes` and use `theme_fivethirtyeight()` to get closer to the aesthetic they use.

Exercise 8 Go for it!

Getting credit

Post your graph to `#lab13` , along with the honor pledge, in Slack by Tuesday.