Analyzing external data set

Ethan Davis, Keaton Wilson, Ellen Bledsoe

Learning Outcomes

- Students will be able to use some functions of the tidyverse: `select`, `filter`, `mutate`, `summarize` and the pipe `%>%`.
- Students will be able to explore and summarize complex, real-world data

Functions used in this lesson

- `read_csv()`
- `select()`
- `filter()`
- `glimpse()`
- `unique()`
- `length()`
- `group_by()`
- `summarize()`
- `mutate()`
- `arrange()`

More practice exploring and summarizing data

```r
# load the tidyverse
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.4     v readr     2.1.5
v forcats   1.0.0     v stringr   1.5.1
v ggplot2   3.5.1     v tibble    3.2.1
v lubridate 1.9.3     v tidyr     1.3.1
v purrr     1.0.2
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to becom
```

To review the **tidyverse** syntax, we're going to use a real data set on weather measurements in Antarctica. It contains weather measurements from many climate stations over the course of a year.

```r
# read in the data file
# `read_csv()` is part of the `tidyverse` and gives us nice options when reading in data
stationData <- read_csv("station-data.csv")
```

```
Rows: 139160 Columns: 12
-- Column specification --------------------------------------------------------
Delimiter: ","
chr  (1): station_id
dbl (11): year, day, month, running_day, hour, temp, pressure, wind_speed, w...

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```r
# Let's take a look at the climate data.
stationData
```

```
# A tibble: 139,160 x 12
    year   day month running_day  hour  temp pressure wind_speed wind_direction
   <dbl> <dbl> <dbl>       <dbl> <dbl> <dbl>    <dbl>      <dbl>          <dbl>
```

```
 1  2018     1     1           1     0 -29.5     629         5.1            247
 2  2018     1     1           1   300 -27.4     629.        5.8            236
 3  2018     1     1           1   600 -25.5     629.        5.5            228
 4  2018     1     1           1   900 -24.9     629.        5.8            219
 5  2018     1     1           1  1200 -25       630.        3.9            230
 6  2018     1     1           1  1500 -27.5     630.        3.4            242
 7  2018     1     1           1  1800 -30.3     630.        3.3            259
 8  2018     1     1           1  2100 -30.1     630.        3.8            243
 9  2018     2     1           2     0 -28.8     630.        5.1            238
10  2018     2     1           2   300 -26.4     630.        4.9            235
# i 139,150 more rows
# i 3 more variables: humidity <dbl>, delta_t <dbl>, station_id <chr>
```

Unlike the data we've seen so far, this data set is *BIG*. While we could reasonably look at the data in our team data set, it would be impossible for us to look at all 139,160 rows of this climate data. This is where R and the tidyverse can be really powerful!

The `tidyverse` offers a great function called `glimpse()` that lets us take a quick look at the data set. (Other similar functions are `str()` and `head()`)

```
# explore the data set with glimpse
glimpse(stationData)
```

```
Rows: 139,160
Columns: 12
$ year           <dbl> 2018, 2018, 2018, 2018, 2018, 2018, 2018, 2018, 2018, 2~
$ day            <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3~
$ month          <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
$ running_day    <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3~
$ hour           <dbl> 0, 300, 600, 900, 1200, 1500, 1800, 2100, 0, 300, 600, ~
$ temp           <dbl> -29.5, -27.4, -25.5, -24.9, -25.0, -27.5, -30.3, -30.1,~
$ pressure       <dbl> 629.0, 628.9, 628.8, 629.1, 629.5, 629.7, 629.7, 630.1,~
$ wind_speed     <dbl> 5.1, 5.8, 5.5, 5.8, 3.9, 3.4, 3.3, 3.8, 5.1, 4.9, 5.5, ~
$ wind_direction <dbl> 247, 236, 228, 219, 230, 242, 259, 243, 238, 235, 230, ~
$ humidity       <dbl> 60.9, 62.0, 64.0, 63.0, 61.0, 64.0, 64.0, 66.6, 62.0, 6~
$ delta_t        <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,~
$ station_id     <chr> "ag4q3h", "ag4q3h", "ag4q3h", "ag4q3h", "ag4q3h", "ag4q~
```

```
# str(data)
str(stationData)
```

```
spc_tbl_ [139,160 x 12] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
 $ year          : num [1:139160] 2018 2018 2018 2018 2018 ...
 $ day           : num [1:139160] 1 1 1 1 1 1 1 1 2 2 ...
 $ month         : num [1:139160] 1 1 1 1 1 1 1 1 1 1 ...
 $ running_day   : num [1:139160] 1 1 1 1 1 1 1 1 2 2 ...
 $ hour          : num [1:139160] 0 300 600 900 1200 1500 1800 2100 0 300 ...
 $ temp          : num [1:139160] -29.5 -27.4 -25.5 -24.9 -25 -27.5 -30.3 -30.1 -28.8 -26.4
 $ pressure      : num [1:139160] 629 629 629 629 630 ...
 $ wind_speed    : num [1:139160] 5.1 5.8 5.5 5.8 3.9 3.4 3.3 3.8 5.1 4.9 ...
 $ wind_direction: num [1:139160] 247 236 228 219 230 242 259 243 238 235 ...
 $ humidity      : num [1:139160] 60.9 62 64 63 61 64 64 66.6 62 64 ...
 $ delta_t       : num [1:139160] NA NA NA NA NA NA NA NA NA NA ...
 $ station_id    : chr [1:139160] "ag4q3h" "ag4q3h" "ag4q3h" "ag4q3h" ...
 - attr(*, "spec")=
  .. cols(
  ..   year = col_double(),
  ..   day = col_double(),
  ..   month = col_double(),
  ..   running_day = col_double(),
  ..   hour = col_double(),
  ..   temp = col_double(),
  ..   pressure = col_double(),
  ..   wind_speed = col_double(),
  ..   wind_direction = col_double(),
  ..   humidity = col_double(),
  ..   delta_t = col_double(),
  ..   station_id = col_character()
  .. )
 - attr(*, "problems")=<externalptr>
```

```r
# head(data, n) gives us the first n rows
head(stationData, 10)
```

```
# A tibble: 10 x 12
    year   day month running_day  hour  temp pressure wind_speed wind_direction
   <dbl> <dbl> <dbl>       <dbl> <dbl> <dbl>    <dbl>      <dbl>          <dbl>
 1  2018     1     1           1     0 -29.5     629         5.1            247
 2  2018     1     1           1   300 -27.4     629.        5.8            236
 3  2018     1     1           1   600 -25.5     629.        5.5            228
 4  2018     1     1           1   900 -24.9     629.        5.8            219
 5  2018     1     1           1  1200 -25       630.        3.9            230
 6  2018     1     1           1  1500 -27.5     630.        3.4            242
 7  2018     1     1           1  1800 -30.3      630.        3.3            259
 8  2018     1     1           1  2100 -30.1      630.        3.8            243
 9  2018     2     1           2     0 -28.8      630.        5.1            238
10  2018     2     1           2   300 -26.4      630.        4.9            235
# i 3 more variables: humidity <dbl>, delta_t <dbl>, station_id <chr>
```

`select()` columns

Now let's introduce a new function, `select()`. The utility of select isn't entirely new to you. It operates in a similar way to the `$`, but with a bit more power. Select allows us to pick out specific columns from our data. You can use names or their position in the data frame. Unlike with `$`, with `select`, we can easily select multiple columns.

First, a quick reminder of how to use `$`

```
# column selection in base R (using $)
stationData$year
```

Now let's see how the same can be accomplished with `select()`. The first argument in the function is the data frame. Any following arguments are the columns we want to select.

`select(.data, column(s))`

```
# first argument is the data frame, then the columns
select(stationData, year)
# multiple columns
#select columns by name
select(stationData, year, month, temp, station_id)
# select adjacent columns
select(stationData, year:hour)
# ignore columns (everything but)
select(stationData, -humidity)
```

Notice that unlike using `$`, using `select()` returns a formatted data frame. With larger data sets, this becomes especially useful.

Let's practice!

Write a line of code to select the following columns from the `stationData`: `temp`, `wind_speed`, `humidity` . Then, select everything but the `wind_direction` column.

```
#select
select(stationData, temp, wind_speed, humidity)
#select
select(stationData, -wind_direction)
```

`filter()` rows

To review, `filter()` allows you filter rows by certain conditions. Rows the meet the conditions are kept, while rows that do not are "filtered" out.

Let's first review relational operators, which make `filter()` work. The ones that are relevant for today are `==` ,`!=` , `>` , and `>=`. You can think about these operators as true or false questions.

`x == y` - "is x equal to y?"

`x != y` - "is x unequal to y?"

`x > y` - "is x greater than y?"

`x >= y` - "is x greater than or equal to y?"

These expressions will evaluate to either `TRUE` or `FALSE`. Here are some examples:

```
# does 2 equal 2?
2==2
```

```
[1] TRUE
```

```
# is x unequal to y?
x<-3
y<-4
x!=y
```

```
[1] TRUE
```

```
# is x greater than y?
x>y
```

```
[1] FALSE
```

```
#is x equal to y?
x==y
```

```
[1] FALSE
```

```
# is "apple" equal to "orange"?
x<-"apple"
y<-"orange"
x==y
```

```
[1] FALSE
```

```
"a">"b"
```

```
[1] FALSE
```

This language of true and false will help us create conditions on which to filter our data.

With a sufficiently large data set though, it may be difficult to determine what to filter *on*.

The `unique()` function tells us all the unique values in an vector/column. When you have an enormous data set like this, you won't be able to gather all the possible values of a variable manually. For instance, we might like to know: "What years are represented in our data?", or "What are the IDs of the various stations?"

```
# get unique values of the year column
unique(stationData$year)

# get unique values of the month column
unique(stationData$month)
```

```
# get unique values of the station_id column
unique(stationData$station_id)
```

Another useful function we'll introduce here is `length()`. Length simply counts the number of entries in a column or vector. We can use it in combination with `unique()` to find how many unique values exist in a column. So, we could then answer the question: "How many stations are there?"

```
# get unique values of the station_id column
length(unique(stationData$station_id))
```

```
[1] 52
```

Now, filter the data so that it only contains one station.

```
# filter for a single station
filter(stationData, station_id=="ag4q3h")

# worth noting here that we haven't saved any of this. We ought to write to a new object
ag <- filter(stationData, station_id=="ag4q3h")
```

Let's practice using `select()` and `filter()`

Working with the climate data, construct a small set of code that does the following:

1. Slims down the full data frame to one that contains the columns `month`, `temp` and `station_id`. Assign this to an object called `slim`.
2. Filters the data for `ag4q3h` with an average temperature less than -22 degrees.
3. Name this new data frame `cold`

```
# not piped
slim <- select(stationData, month, temp, station_id)

slim

cold <- filter(slim, station_id=="ag4q3h", temp < -22)

cold
```

The pipe %>%

You can use the pipe operator to chain tidyverse functions together. You can think of the pipe as automatically sending the output from the first line into the next line as the input.

This is helpful for a lot of reasons, including:

1. removing the clutter of creating a lot of intermediate objects in your work space, which reduces the chance of errors caused by using the wrong input object
2. makes things more human-readable (in addition to computer-readable)

Now, let's try the same as above, but using the pipe:

```
#with pipe
stationData %>%
  select(temp, station_id, month) %>%
  filter(station_id=="ag4q3h", temp < -22)
```

```
# A tibble: 2,872 x 3
    temp station_id month
   <dbl> <chr>      <dbl>
 1 -29.5 ag4q3h         1
 2 -27.4 ag4q3h         1
 3 -25.5 ag4q3h         1
 4 -24.9 ag4q3h         1
 5 -25   ag4q3h         1
 6 -27.5 ag4q3h         1
 7 -30.3 ag4q3h         1
 8 -30.1 ag4q3h         1
 9 -28.8 ag4q3h         1
10 -26.4 ag4q3h         1
# i 2,862 more rows
```

Let's practice!

In small groups, use pipes to create a new data frame called `singleStation` that includes the following:

- the columns `day`, `temp`, `station_id`
- only rows with temperatures that are greater than -20 degrees

```
singleStation <- stationData %>%
  select(day, temp , station_id) %>%
  filter(temp > -20)

singleStation
```

```
# A tibble: 52,736 x 3
     day  temp station_id
   <dbl> <dbl> <chr>
 1   356 -18.4 ag4q3h
 2   356 -17.7 ag4q3h
 3   356 -18.4 ag4q3h
 4   356 -19.8 ag4q3h
 5   357 -19.2 ag4q3h
 6   357 -18.7 ag4q3h
 7   357 -18.6 ag4q3h
 8   357 -19.6 ag4q3h
 9   358 -18.5 ag4q3h
10   358 -18.8 ag4q3h
# i 52,726 more rows
```

Creating new variables with `mutate()`

Sometimes our data aren't in exactly the format we want. For example, we might want our temperature data in Fahrenheit instead of Celsius.

The `tidyverse` has a function called `mutate()` that lets us create a new column. Often, we want to apply a function to the entire column or perform some type of calculation, such as converting temp from F to C.

To help us out, here is the equation for converting: `Fahrenheit = Celcius * (9/5) + 32`

```r
# create a new column for temps in Fahrenheit
mutate(stationData, temp_f = temp * (9/5)+32)
```

```
# A tibble: 139,160 x 13
    year   day month running_day  hour  temp pressure wind_speed wind_direction
   <dbl> <dbl> <dbl>       <dbl> <dbl> <dbl>    <dbl>      <dbl>          <dbl>
 1  2018     1     1           1     0 -29.5     629         5.1            247
 2  2018     1     1           1   300 -27.4     629.        5.8            236
 3  2018     1     1           1   600 -25.5     629.        5.5            228
 4  2018     1     1           1   900 -24.9     629.        5.8            219
 5  2018     1     1           1  1200 -25       630.        3.9            230
 6  2018     1     1           1  1500 -27.5     630.        3.4            242
 7  2018     1     1           1  1800 -30.3     630.        3.3            259
 8  2018     1     1           1  2100 -30.1     630.        3.8            243
 9  2018     2     1           2     0 -28.8     630.        5.1            238
10  2018     2     1           2   300 -26.4     630.        4.9            235
# i 139,150 more rows
# i 4 more variables: humidity <dbl>, delta_t <dbl>, station_id <chr>,
#   temp_f <dbl>
```

Understanding data through `summarize()`

Like we have talked about in previous classes, one of the best ways for us to understand our data is through what we call summary statistics such as the mean, standard deviation, etc.

Summary statistics are particularly useful for large data sets, because we cannot navigate the data manually. Along with data visualization (covered soon!), summary statistics are essential to understanding large data sets.

```
# first attempt at mean and sd of average temperature
stationData %>%
  summarize(mean_temp=mean(temp, na.rm=TRUE), sd_temp=sd(temp, na.rm=TRUE))
```

```
# A tibble: 1 x 2
  mean_temp sd_temp
      <dbl>   <dbl>
1     -25.3    15.3
```

The above gives us an impression of the climate of the whole region contained by all of the stations. However, this wouldn't really help us choose an ideal location for us to inhabit, or tell us which areas we might avoid.

Grouping data by variable values gives us a clearer picture of specific subsets of the data. For instance, we might like to know the average measurements for specific locations.

The `group_by()` function lets us do this. It is most often used in combination with `summarize()`.

We can use this method to calculate the mean temperatures of each station_id instead of the overall mean of the entire data set.

Next, use `arrange()` function, which can rearrange our data in numerical order by a specific column. For instance, we could use it with the code below to find the stations with the highest and lowest average temperatures.

```
#group_by summarize arrange
stationData %>%
  group_by(station_id) %>%
  summarize(mean_temp=mean(temp, na.rm=TRUE)) %>%
  arrange(mean_temp)
```

```
# A tibble: 52 x 2
   station_id mean_temp
   <chr>          <dbl>
 1 fujq3h         -52.6
 2 dc2q3h         -51.7
 3 ag4q3h         -51.0
 4 jasq3h         -49.8
 5 rlsq3h         -44.6
 6 d85q3h         -40.5
 7 balq3h         -36.8
 8 pdaq3h         -35.8
 9 mizq3h         -29.8
10 kmsq3h         -29.3
# i 42 more rows
```

One great thing about `group_by()` is that you can give it multiple columns. This allows us to have groups within groups. For instance, we could organize our data by `station_id` and then break up data for each station by `month`. That's a mouthful, so let's just take a look at it:

```r
#group_by multiple columns
stationData %>%
  group_by(station_id, month) %>%
  summarize(mean_temp=mean(temp, na.rm=TRUE))
```

```
`summarise()` has grouped output by 'station_id'. You can override using the
`.groups` argument.
```

```
# A tibble: 571 x 3
# Groups:   station_id [52]
   station_id month mean_temp
   <chr>      <dbl>     <dbl>
 1 ag4q3h         1     -31.4
 2 ag4q3h         2     -43.0
 3 ag4q3h         3     -54.9
 4 ag4q3h         4     -60.7
 5 ag4q3h         5     -57.1
 6 ag4q3h         6     -58.6
 7 ag4q3h         7     -60.9
 8 ag4q3h         8     -61.2
 9 ag4q3h         9     -63.6
10 ag4q3h        10     -51.8
# i 561 more rows
```

What are good/bad columns to group by? Try summarizing the mean temperature, grouping by different combinations of columns. Talk with your neighbors about which columns might or might not work well.