Lecture 4: Data Wrangling: Part 2

Stats 32: Introduction to R for Undergraduates

Harrison Li

April 11, 2024

Agenda

Reading: Sections 3.3, 3.4, 3.6, 3.7

Summary statistics

Summary statistics

Given a *numeric* vector, R provides several built in mathematical functions to *summarize* the data, i.e. return a single number that captures some aspect of the data vector:

- `length()`: computes the number of entries
- `sum()`: computes the sum of all the entries
- `mean()`: computes the average entry value, equivalent to sum divided by length
- `median()`: computes the median entry
- `min()`: computes the minimum entry
- `max()`: computes the maximum entry
- `sd()`: computes the (sample) standard deviation of the entries

Summary statistics

```r
my_vector <- c(0, 3, 3, -2)
max(my_vector)
```

```
## [1] 3
```

```r
length(my_vector)
```

```
## [1] 4
```

```r
sum(my_vector)
```

```
## [1] 4
```

```r
mean(my_vector)
```

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

```r
sum(my_vector)/length(my_vector)
```

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

```r
sd(my_vector)
```

```
## [1] 2.44949
```

summarise()

`summarise()`

The dplyr verb `summarise()` takes in a tibble and returns a *new* tibble that shows the result of applying the summary statistic(s) of your choice to any column(s) in your tibble.

Its syntax is similar to that of `mutate()`. The difference is that `summarise()` (without a grouping variable, which we will see later) returns a tibble with just 1 row. Each column in this tibble is a single numerical summary of a column in the original tibble.

summarise()

Let's read in the abalone data again, and get the length of the longest abalone, as well as the mean and standard deviation of Rings.

```r
library(tidyverse)
abalone <- read_csv("abalone.csv")
summary_tibble <- abalone %>%
  summarise(Longest_Abalone=max(Length),
            Mean_Rings=mean(Rings),
            Rings_SD=sd(Rings))
summary_tibble
```

```
## # A tibble: 1 x 3
##   Longest_Abalone Mean_Rings Rings_SD
##             <dbl>      <dbl>    <dbl>
## 1           0.815       9.97     3.21
```

`group_by()`

`group_by()`

Often, you do not want to summarize all the rows in a tibble; you want a breakdown based on another *grouping variable*.

For instance, suppose you have a dataset with the heights of randomly sampled adults around the world. You might be interested in knowing the average height of the adults you sampled, broken down by country.

In this case, you want to *group* your data *by* country, the grouping variable. The `dplyr` verb `group_by()` does just that. It doesn't cause any externally visible changes to your tibble. However, the internal structure is changed, as will be seen if you follow up with `summarise()`. Your summary will now have one row for each value of the grouping variable!

`group_by()`

Suppose *long* abalones are defined as those with `Length` > 0.5, and *short* abalones are those with `Length` $<= 0.5$.

How can we find the number of long and short abalones in our data, as well as the average heights in each length category?

From the help page, we see that `group_by()` takes in `.data` as its first argument (like any other `dplyr` verb). Then, it requires us to specify the variables to group by (i.e. the grouping variables).

group_by()

The abalone tibble doesn't explicitly tell us whether an abalone is long or short, so we can't immediately use group_by().

However, this information can be deduced from the Length column. Thus, we proceed by using mutate() to create a new column in the tibble called Height_Type, which specifies whether the abalone is in fact long or short. We store this in a new variable abalone_enhanced. Note the use of the ifelse() function, which can be useful:

```r
abalone_enhanced <- abalone %>%
  mutate(Height_Type=ifelse(Length > 0.5, "long", "short"))
```

group_by() with summarise(): a power duo

We can now call group_by() and summarise() on abalone_enhanced, using Height_Type as the grouping variable.

```r
abalone_enhanced %>%
  group_by(Height_Type) %>%
  summarise(Average_Height=mean(Height),
            Quantity=n())
```

```
## # A tibble: 2 x 3
##   Height_Type Average_Height Quantity
##   <chr>                <dbl>    <int>
## 1 long                0.164       187
## 2 short               0.0992      113
```

Note: n() is a special function with no arguments that can be used in summarise() to count the number of observations (in each group, when the tibble is grouped).

Categorical and quantitative variables

Typically, we want the grouping variable to be *categorical*, that is, a variable that takes on one of a few possible values, as opposed to a *quantitative* variable that takes on numeric values on a continuous, ordered spectrum.

Are the following variables categorical or quantitative?

- Wind speed
- Wind direction
- Income
- Eye color
- Number of bedrooms for condos in a small apartment building

Categorical and quantitative variables

What happens if we try to group by a quantitative variable?

```
abalone %>%
  group_by(Length) %>%
  summarise(Average_Height=mean(Height))
```

```
## # A tibble: 97 x 2
##    Length Average_Height
##     <dbl>          <dbl>
##  1  0.175          0.04
##  2  0.19           0.04
##  3  0.21           0.05
##  4  0.215          0.03
##  5  0.23           0.06
##  6  0.235          0.065
##  7  0.255          0.06
##  8  0.265          0.065
##  9  0.27           0.08
## 10  0.275          0.055
## # i 87 more rows
```

Joins

Joins

What if the information you want is spread across two different tibbles? **Joins** are the solution, enabling you to “join” together two tibbles into one.

In order to join two tibbles, they must have at least one column in common, sometimes called a *key variable*.

Joins

Let's load in the `nycflights13` package, and take a look at the included tibbles `airports` and `flights`:

```r
library(nycflights13)
head(airports, 5)
```

```
## # A tibble: 5 x 8
##   faa   name                             lat   lon   alt    tz dst   tzone
##   <chr> <chr>                          <dbl> <dbl> <dbl> <dbl> <chr> <chr>
## 1 04G   Lansdowne Airport               41.1 -80.6  1044    -5 A     America/New~
## 2 06A   Moton Field Municipal Airport  32.5 -85.7   264    -6 A     America/Chi~
## 3 06C   Schaumburg Regional             42.0 -88.1   801    -6 A     America/Chi~
## 4 06N   Randall Airport                 41.4 -74.4   523    -5 A     America/New~
## 5 09J   Jekyll Island Airport           31.1 -81.4    11    -5 A     America/New~
```

```r
head(flights, 5)
```

```
## # A tibble: 5 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
## 1  2013     1     1      517            515         2      830            819
## 2  2013     1     1      533            529         4      850            830
## 3  2013     1     1      542            540         2      923            850
## 4  2013     1     1      544            545        -1     1004           1022
## 5  2013     1     1      554            600        -6      812            837
## # i 11 more variables: 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>
```

Joins

We notice that the `flights` tibble has information about a number of flights departing from the three major NYC area airports in 2013.

However, the only information about the flight destination is the FAA code in the column `dest`. We might want to have more information, e.g. the full name of the destination airport.

```r
head(flights$dest)
```

```
## [1] "IAH" "IAH" "MIA" "BQN" "ATL" "ORD"
```

Key variables

Luckily, the `airports` tibble contains information including both their FAA codes and full names.

Since both `flights` and `airports` have FAA codes, the FAA codes are a **key variable** that enables us to join these tibbles.

Note the FAA codes are stored in differently named columns: `flights$dest` and `airports$faa`.

Left join

The result of a **left** join of `flights` onto `airports` will be an "augmented tibble" with at least one row for each row in `flights`, but more columns.

The augmented tibble has all columns in `flights` and all columns in `airports`, with one exception: it has no column called `faa`, since the join absorbs that into the matching `dest` column.

It would be redundant to have both `faa` and `dest`.

Left join

For each row in `flights`, in the left joined tibble we will have one row for each row in `airports` that has the same FAA code. It is probably easier to understand this fully with some worked examples - see the lab.

Left joins are carried out with the `dplyr` function `left_join()`. The `by()` argument specifies the key variable(s).

Left join

```r
flights %>%
  left_join(y=airports, by=c("dest"="faa")) %>%
  select(tailnum, origin, dest, name)
```

```
## # A tibble: 336,776 x 4
##    tailnum origin dest  name
##    <chr>   <chr>  <chr> <chr>
##  1 N14228  EWR    IAH   George Bush Intercontinental
##  2 N24211  LGA    IAH   George Bush Intercontinental
##  3 N619AA  JFK    MIA   Miami Intl
##  4 N804JB  JFK    BQN   <NA>
##  5 N668DN  LGA    ATL   Hartsfield Jackson Atlanta Intl
##  6 N39463  EWR    ORD   Chicago Ohare Intl
##  7 N516JB  EWR    FLL   Fort Lauderdale Hollywood Intl
##  8 N829AS  LGA    IAD   Washington Dulles Intl
##  9 N593JB  JFK    MCO   Orlando Intl
## 10 N3ALAA  LGA    ORD   Chicago Ohare Intl
## # i 336,766 more rows
```

Left join

Above, we used a **left** join of `flights` onto `airports`. This means we are guaranteed to have at least one row in the resulting tibble for each row in `flights`.

If there were multiple rows with the same FAA code in `airports`, we could potentially have multiple rows for each row in `flights`. Not the case for this example.

If, for some row in `flights`, the entry for `dest` is an FAA code not in `airports`, we will return `NA` for all of the columns corresponding to those in `airports`.

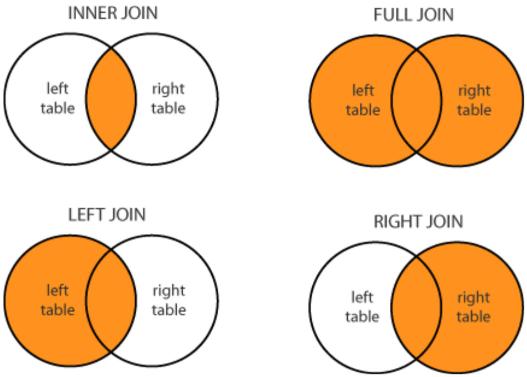
For example, consider row 4 in the result of the left join previous slide. The airport `BQN` is not in `airports`, so there is an `NA` in the `dest` column for that row.

Left, right, inner, full (outer) joins

While left joins can handle most use cases, it's useful to be aware of the other types of joins.

- Right join: Return at least one row for each row in the **second** tibble. In general, left joining tibble A onto tibble B is the same as right joining tibble B onto tibble A.
- Inner join: Only return rows where there is a match between both tibbles. In other words, take the result of a left (or right) join, and get rid of the rows with NA in any of the new columns. Inner joining A onto B is the same as inner joining B onto A.
- Full (outer) join: Return at least one row for each row in the first tibble and at least one row for each row in the right tibble, with NAs as necessary. Again, the order doesn't matter.

Left, right, inner, full (outer) joins


INNER JOIN
left table
right table
FULL JOIN
left table
right table
LEFT JOIN
left table
right table
RIGHT JOIN
left table
right table