Data Frames and dplyr

David Gerard

2019-02-06

Learning Objectives

- Manipulating data frames.
- Calculating summary statistics.
- Using the basic functions of dplyr
- Chapter 5 of RDS

Background

- A data frame consists of variables along the columns and observations along the rows.
- For example, in the `msleep` data frame, the observations are animals and the the variables are properies of those animals (body weight, total sleep time, etc).
- **Data frames are the fundamental data type in most analyses.**
- Common operations on a data frame during an analysis:
  - Select specific variables (`select()`).
  - Select observational units by the values of some variables (`filter()`).
  - Create new variables from old variables (`mutate()`)
  - Reorder the observational units (`arrange()`)
  - Create summary statistics from many observational units (`summarize()`)
  - Group the observational units by the values of some variables (`group_by()`).
- As a taste, let's look at an example from the `flights` data frame from the nycflights13 package:

```r
library(nycflights13)
data("flights")
```

- Suppose we want calculate the average departure delay for the flights from carrier in the second half of the year. The steps would be
  1. Select only flights from the second half of the year.
  2. Group the flights by the carrier.
  3. Calculate the average departure delay time within each carrier.
- In base R, this operation would look like:

```r
flights2 <- flights[flights$month >= 7, ]
flights3 <- aggregate(dep_time ~ carrier, FUN = mean, data = flights2)
flights3
```

- In tidyverse, this looks like

```r
suppressPackageStartupMessages(library(tidyverse))
```

```r
flights %>%
  filter(month >= 7) %>%
```

```r
  group_by(carrier) %>%
  summarize(mean_dep = mean(dep_time, na.rm = TRUE))
```

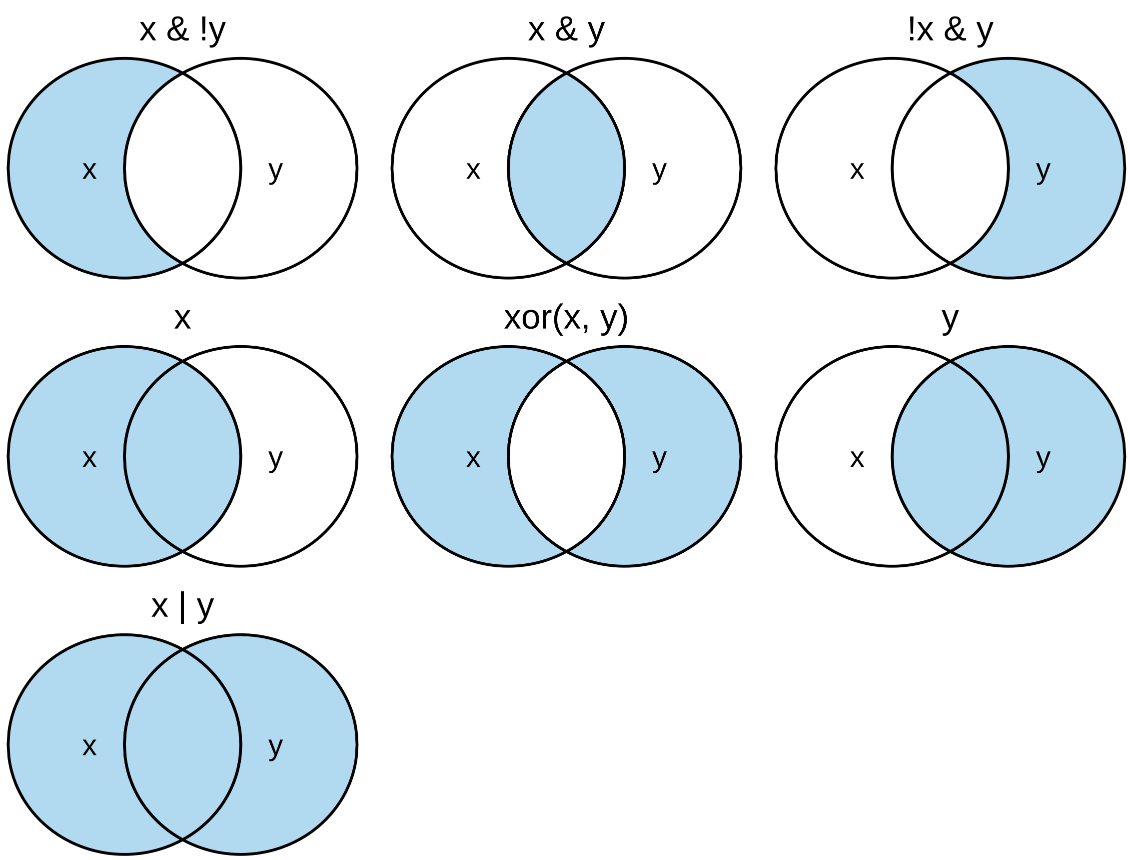
- In the tidyverse:
  - We get to use piping.
  - It's more expressive/clear.

Filter Rows Based on Variable Values

- In the tidyverse, we use the `filter()` function to select rows (observations) based on the values of some variables.
- You create logical conditions and the rows that satisfy these logical conditions (return `TRUE`) are selected.
- Let's extract all flights from new york that occurred in January.

```r
flights %>%
  filter(month == 1)
```

- You can filter based on more than two variables using logical operators.
- Graphical Depiction of Logical Operations:



- Let's get all flights that were both in January and from JFK.

```r
flights %>%
  filter(month == 1 & origin == "JFK")
```

- If you don't know what variable values are possible in a categorical variable, then you can try two things:

  1. `levels()` if the variable is a factor.
  2. `unique()` otherwise.

  ```
  unique(flights$origin)
  ```

  ```
  ## [1] "EWR" "LGA" "JFK"
  ```

- Because the *and* operator is the most used, `filter()` will also perform the and operation if you separate logical conditions by a comma.

  ```
  flights %>%
    filter(month == 1, origin == "JFK")
  ```

- You should still know the logical operators in case the filtering gets super complicated.

- Let's extract the January LGA flights and the December JFK flights.

  ```
  flights %>%
    filter((month == 1 & origin == "LGA") | (month == 12 & origin == "JFK"))
  ```

- **Exercise**: Extract all flights that either occur on odd months, or on odd days of even months.

- **Exercise** (RDS 5.2.4.1) Find all flights that satisfy the following conditions

  1. Had an arrival delay of two or more hours
  2. Flew to Houston (IAH or HOU)
  3. Were operated by United, American, or Delta
  4. Departed in summer (July, August, and September)
  5. Arrived more than two hours late, but didn't leave late

Missing Values

- **`filter()` will exclude observations with missing values**.

- If you want to extract those rows as well, you have to ask for them explicitly using `is.na()`.

  ```
  dfdat <- data.frame(x = c(1, NA, 2),
                      y = c(2, 4, 1))
  dfdat %>%
    filter(x == 1)
  ```

  ```
  ##   x y
  ## 1 1 2
  ```

  ```
  dfdat %>%
    filter(x == 1 | is.na(x))
  ```

  ```
  ##    x y
  ## 1  1 2
  ## 2 NA 4
  ```

- **You cannot use `NA == NA`**. If two observations are missing, then you don't know if they are equal, so R will return `NA` to this:

  ```
  NA == NA
  ```

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

near()

- Unless you explicitely tell it, R treats all numerics as floats.
- It's thus dangerous to use `==` for numerics.
- Instead, use the `near()` function.

```
sqrt(2) ^ 2
```

```
## [1] 2
```

```
sqrt(2) ^ 2 == 2
```

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

```
near(sqrt(2) ^ 2, 2)
```

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

- If a variable is an integer `<int>`, then it's OK to use `==`

```
twoint <- as.integer(sqrt(2) ^ 2)
twoint == 2
```

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

Arrange order of rows

- Use `arrange()` to order the rows by the value of a variable.

```
flights %>%
  arrange(dep_delay)
```

- The default is the arrange in **ascending** order. To arrange in descending order, use the `desc()` function.

```
flights %>%
  arrange(desc(dep_delay))
```

- If there are ties, then you can break the ties by arranging by another variable.

```
dfdat <- data.frame(x = c(1, 2, 1, 2),
                    y = c(2, 2, 1, 1))
dfdat
```

```
##   x y
## 1 1 2
## 2 2 2
## 3 1 1
## 4 2 1
```

```
dfdat %>%
  arrange(x)
```

```
##   x y
## 1 1 2
## 2 1 1
## 3 2 2
## 4 2 1
```

```r
dfdat %>%
  arrange(x, y)
```

```
##   x y
## 1 1 1
## 2 1 2
## 3 2 1
## 4 2 2
```

- Observations with missing values are always placed at the end (even when using the `desc()` function)

Select Specific Columns

- The `select()` function will extract variables and place them in a smaller data frame.
- Select specific variables

```r
flights %>%
  select(dep_delay, arr_delay)
```

- Select a range of variables with `:`

```r
flights %>%
  select(year:day)
```

- Select all variables except certain ones with `-`

```r
flights %>%
  select(-dep_delay, -arr_delay)
```

- Select all variables except within a range of columns.

```r
flights %>%
  select(-(year:day))
```

- Useful helper functions for `select()`:
  - `starts_with("abc")`: matches names that begin with `"abc"`.
  - `ends_with("xyz")`: matches names that end with `"xyz"`.
  - `contains("ijk")`: matches names that contain `"ijk"`.
  - `matches("(.)\\1")`: selects variables that match a regular expression. This one matches any variables that contain repeated characters. You'll learn more about regular expressions in strings.
  - `num_range("x", 1:3)`: matches `x1`, `x2`, and `x3`.

```r
flights %>%
  select(ends_with("delay"))
```

```r
flights %>%
  select(starts_with("dep"), year, month, day)
```

- **Exercise**: Select all variables that have anything to do with the arrival. Also keep the `year`, `month`, and `day`. Use as few characters as possible in your `select()` call.

Rename Variables

- Use `rename()` to rename a variable.

```
flights %>%
  rename(departureTime = dep_time)
```

Create New Variables

- The variables we have are usually not enough for an analysis.
  - Take a log-transformation of positive data to make associations more linear.
  - Create new features based on existing features.
- We can use `mutate()` to create new variables from old.

```
flights %>%
  mutate(gain = dep_delay - arr_delay,
         speed = distance / air_time * 60)
```

- If you only want to keep new variables, use `transmute()`

```
flights %>%
  transmute(gain = dep_delay - arr_delay,
            hours = air_time / 60,
            gain_per_hour = gain / hours)
```

```
## # A tibble: 336,776 x 3
##     gain hours gain_per_hour
##    <dbl> <dbl>         <dbl>
##  1    -9 3.78          -2.38
##  2   -16 3.78          -4.23
##  3   -31 2.67         -11.6
##  4    17 3.05           5.57
##  5    19 1.93           9.83
##  6   -16 2.5           -6.4
##  7   -24 2.63          -9.11
##  8    11 0.883         12.5
##  9     5 2.33           2.14
## 10   -10 2.3           -4.35
## # ... with 336,766 more rows
```

- **Exercise**: (RDS 3.5.2.1) Currently `dep_time` and `sched_dep_time` are convenient to look at, but hard to compute with because they're not really continuous numbers. Convert them to a more convenient representation of number of minutes since midnight. Hint: `%/%` is integer division and `%%` is remainder.

Summaries

- We can create summary statistics using the `summarize()` function.
- The following will calculate the mean departure delay time.

```
flights %>%
  summarize(mean_del = mean(dep_delay, na.rm = TRUE))
```

```
## # A tibble: 1 x 1
##   mean_del
##      <dbl>
## 1     12.6
```

- **Exercise**: What is the standard deviation of the departure delay time?

Grouped Summaries

- You can create a grouped data frame using the `group_by()` function.
- You define what variables to group the observational units by.
- Each unique combination of the values of the grouping variables will create a new group.
- Consider the data set:

```
dfdat <- tribble(~x,  ~y,  ~z,
                 "a", "c", 1,
                 "a", "d", 2,
                 "a", "c", 3,
                 "a", "c", 4,
                 "b", "c", 5,
                 "b", "d", 6,
                 "b", "c", 7)
dfdat
```

```
## # A tibble: 7 x 3
##   x     y         z
##   <chr> <chr> <dbl>
## 1 a     c         1
## 2 a     d         2
## 3 a     c         3
## 4 a     c         4
## 5 b     c         5
## 6 b     d         6
## 7 b     c         7
```

- If we group by the variable `x`, then there are two groups:
  i. Rows 1, 2, 3, 4 (corresponding to `"a"`)
  ii. Rows 5, 6, 7 (corresponding to `"b"`)
- If we group by the variable `y` then there are also two groups:
  i. Rows 1, 3, 4, 7, 5 (corresponding to `"c"`)
  ii. Rows 2, 6 (corresponding to `"d"`)
- If we group by both `x` and `y` then we have four groups:
  i. Rows 1, 3, 4 (corresponding to `"a"` and `"c"`)
  ii. Row 1 (corresponding to `"a"` and `"d"`)
  iii. Rows 5, 7 (corresponding to `"b"` and `"c"`)
  iv. Row 6 (corresponding to `"b"` and `"d"`)

```
dfdat %>%
  group_by(x) ->
  grouped_dfdat
attributes(grouped_dfdat)
```

```
## $names
## [1] "x" "y" "z"
##
## $row.names
## [1] 1 2 3 4 5 6 7
##
## $class
## [1] "grouped_df" "tbl_df"     "tbl"        "data.frame"
##
## $vars
## [1] "x"
##
## $drop
## [1] TRUE
##
## $indices
## $indices[[1]]
## [1] 0 1 2 3
##
## $indices[[2]]
## [1] 4 5 6
##
##
## $group_sizes
## [1] 4 3
##
## $biggest_group_size
## [1] 4
##
## $labels
##   x
## 1 a
## 2 b
```

- The grouping function is most useful to calculate summaries within each group.
- The `summarize()`, `filter()`, `arrange()`, `mutate()` functions will now all operate in a group-specific manner.
- Suppose we want to calculate the mean and standard deviation of the delays within each airport?

```r
flights %>%
  group_by(origin) %>%
  summarize(sd_del = sd(dep_delay, na.rm = TRUE),
            mean_del = mean(dep_delay, na.rm = TRUE))
```

- Or at a particular time of day within each airport:
- Suppose we want to calculate the mean and standard deviation of the delays within each airport?

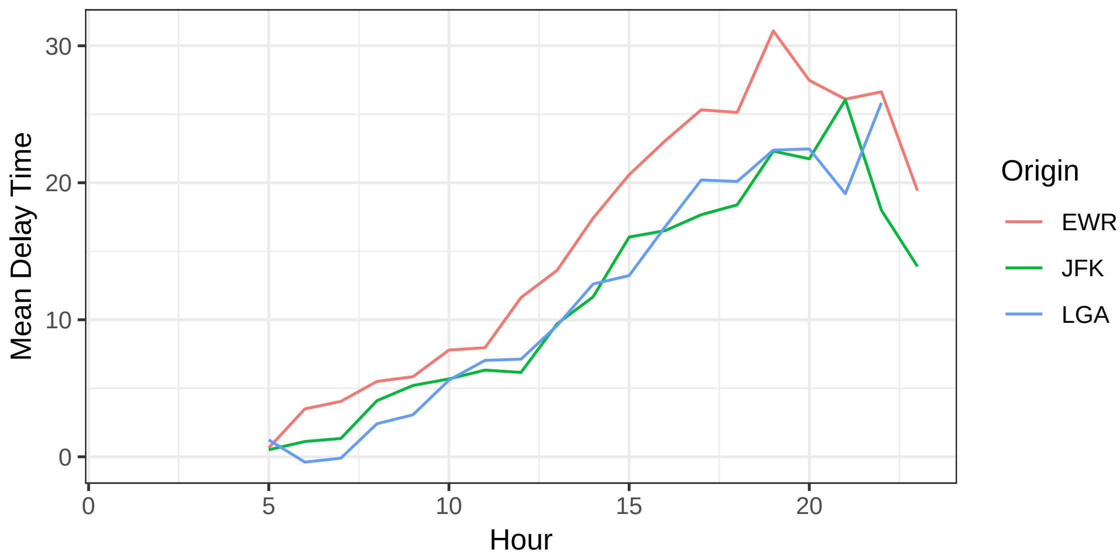
```r
flights %>%
  group_by(origin, hour) %>%
  summarize(sd_del = sd(dep_delay, na.rm = TRUE),
            mean_del = mean(dep_delay, na.rm = TRUE))
```

- We can save this output and feed into ggplot2

```
flights %>%
  group_by(origin, hour) %>%
  summarize(sd_del = sd(dep_delay, na.rm = TRUE),
            mean_del = mean(dep_delay, na.rm = TRUE)) ->
  sumdf

ggplot(data = sumdf, mapping = aes(x = hour, y = mean_del, col = origin)) +
  geom_line() +
  theme_bw() +
  xlab("Hour") +
  ylab("Mean Delay Time") +
  scale_color_discrete(name = "Origin")
```

```
## Warning: Removed 1 rows containing missing values (geom_path).
```



- The `n()` function will count the number of observational units in a group. **It is a good idea to always include this function in a `summarize()` call.

```
flights %>%
  group_by(origin, hour) %>%
  summarize(sd_del = sd(dep_delay, na.rm = TRUE),
            mean_del = mean(dep_delay, na.rm = TRUE),
            n = n())
```

- **Exercise**: Look at the number and proportion of cancelled flights per day. Is there a pattern? Is the proportion of cancelled flights related to the average delay? We'll define a flight to be canceled by `is.na(dep_delay) | is.na(arr_delay)`.

Select Specific Rows

- You can select sertain rows of a data frame using the `slice()` function.

```
flights %>%
  slice(c(1, 4, 6))
```

```r
flights %>%
  slice(10:n())
```