Lab 7: Data cleaning with dplyr

1 Introduction

When working with data you must:

1. Figure out what you want to do.
2. Precisely describe what you want in the form of a computer program.
3. Execute the code.

The dplyr package makes each of these steps as fast and easy as possible by:

1. Elucidating the most common data manipulation operations, so that your options are helpfully constrained when thinking about how to tackle a problem.
2. Providing simple functions that correspond to the most common data manipulation verbs, so that you can easily translate your thoughts into code.
3. Using efficient data storage backends, so that you spend as little time waiting for the computer as possible.

The goal of this lesson is to introduce you to the basic tools that dplyr provides, and show how you to apply them to data frames. You must have the `dplyr`, `nycflights13` and `hflights` packages installed before you can proceed. Here is one method to load multiple libraries.

```
lapply(c("dplyr", "nycflights13", "hflights"), require, character.only=TRUE)
```

2 The 5 basic verbs

The `dplyr` package contains five key data manipulation functions, also called verbs:

- `select()`: Returns a subset of the columns.
- `filter()`: Returns a subset of the rows.
- `arrange()`: Reorders the rows according to single or multiple variables.
- `mutate()`: Adds columns from existing data.
- `summarise()`: Reduces each group to a single row by calculating aggregate measures.

You will learn how to use these verbs in the `dplyr Swirl` lesson. Complete that now. *Note: There is one spot with a bug where it will throw an error. You can simply hit "Enter" and continue on with the lesson to bypass the error.*

2.1 Practice

For this exercise, at each step use the assignment operator `<-` to store the results into a new data table and use that data in the next step. You will use the `hflights` data set which is data on flights departing from two Houston airports in 2011.

1. Use `select()` to extract the following variables: `Distance`, and `AirTime`.

```r
a <- select(________, Distance, ________)
```

2. Use `filter()` to select only the flights who traveled exactly 501 miles.

```r
b <- filter(a , ______ == 501)
```

3. Use `mutate()` to create a new variable `Speed` that calculates speed of the plane as Distance/AirTime*60.

```r
c <- mutate(b , Speed = _____ / ______ * __)
```

4. Use `arrange()` to order the previous data frame by ascending `AirTime`.

```r
d <- arrange(___ , ______)
```

5. Use `slice()` to select the flight with the shortest time in the air. (*Hint: the shorteset flight should be in row 1 after you sort by ascending AirTime*). How fast did this plane go?

```r
slice(d, __)
```

3 Additional Helper functions

`dplyr` comes with 6 helper functions that can help you select variables. These functions find groups of variables to select, based on their names. They can only be used inside `select()`. Here are 3 of the most common.

- `starts_with("X")`: every name that starts with "X".

```r
select(planes, starts_with("tail"))
```

```
## Source: local data frame [3,322 x 1]
##
##    tailnum
## 1   N10156
## 2   N102UW
## 3   N103US
## 4   N104UW
## 5   N10575
## 6   N105UW
## 7   N107US
## 8   N108UW
## 9   N109UW
## 10  N110UW
## ..     ...
```

- `ends_with("X")`: every name that ends with "X".

```r
select(planes, ends_with("er"))
```

```
## Source: local data frame [3,322 x 1]
##
##        manufacturer
## 1           EMBRAER
## 2  AIRBUS INDUSTRIE
## 3  AIRBUS INDUSTRIE
## 4  AIRBUS INDUSTRIE
## 5           EMBRAER
## 6  AIRBUS INDUSTRIE
## 7  AIRBUS INDUSTRIE
## 8  AIRBUS INDUSTRIE
## 9  AIRBUS INDUSTRIE
## 10 AIRBUS INDUSTRIE
## ..              ...
```

- `contains("X")`: every name that contains "X".

```r
select(planes, contains("eng"))
```

```
## Source: local data frame [3,322 x 2]
##
##    engines    engine
## 1        2 Turbo-fan
## 2        2 Turbo-fan
## 3        2 Turbo-fan
## 4        2 Turbo-fan
## 5        2 Turbo-fan
## 6        2 Turbo-fan
## 7        2 Turbo-fan
## 8        2 Turbo-fan
## 9        2 Turbo-fan
## 10       2 Turbo-fan
## ..     ...       ...
```

Watch out: Surround character strings with quotes when you pass them to a helper function, but do not surround variable names with quotes if you are not passing them to a helper function.

```r
select(planes, contains("engine"))
```

```
## Source: local data frame [3,322 x 2]
##
##    engines    engine
## 1        2 Turbo-fan
## 2        2 Turbo-fan
## 3        2 Turbo-fan
## 4        2 Turbo-fan
## 5        2 Turbo-fan
## 6        2 Turbo-fan
## 7        2 Turbo-fan
## 8        2 Turbo-fan
## 9        2 Turbo-fan
## 10       2 Turbo-fan
## ..     ...       ...
```

```
select(planes, "engine")
```

```
## Error: All select() inputs must resolve to integer column positions.
## The following do not:
## *  "engine"
```

```
select(planes, engine)
```

```
## Source: local data frame [3,322 x 1]
##
##       engine
## 1  Turbo-fan
## 2  Turbo-fan
## 3  Turbo-fan
## 4  Turbo-fan
## 5  Turbo-fan
## 6  Turbo-fan
## 7  Turbo-fan
## 8  Turbo-fan
## 9  Turbo-fan
## 10 Turbo-fan
## ..       ...
```

3.1 Practice

Still using the `hflights` data table, select columns with the following conditions:

1. All varibles that start with the letter `D`.
2. All variables that end with `Num`.
3. All variables that contain an the word `Taxi` or `Delay`.

4 Grouped Operations

The above verbs are useful, but they become really powerful when you combine them with the idea of "group by", repeating the operation individually on groups of observations within the dataset. In dplyr, you use the `group_by()` function to describe how to break a dataset down into groups of rows. You can then use the resulting object in exactly the same functions as above; they'll automatically work "by group" when the input is a grouped.

Let's demonstrate how some of these functions work after grouping the flights data set by month. First we'll create a new data set that is grouped by month.

```
by_month <- group_by(flights, month)
```

- If we want to sort the data, `arrange()` orders first by grouping variables, then by the sorting variable.

```
how_long <- arrange(by_month, distance)
how_long
```

```
## Source: local data frame [336,776 x 16]
## Groups: month
##
##    year month day dep_time dep_delay arr_time arr_delay carrier tailnum
## 1  2013     1   3     2127        -2     2222        -2      EV  N13989
## 2  2013     1   4     1240        40     1333        27      EV  N14972
## 3  2013     1   4     1829       134     1937       136      EV  N15983
## 4  2013     1   4     2128        -1     2218        -6      EV  N27962
## 5  2013     1   5     1155        -5     1241       -25      EV  N14902
## 6  2013     1   6     2125        -4     2224         0      EV  N22909
## 7  2013     1   7     2124        -5     2212       -12      EV  N33182
## 8  2013     1   8     2127        -3     2304        39      EV  N11194
## 9  2013     1   9     2126        -3     2217        -7      EV  N17560
## 10 2013     1  10     2133         4     2223        -1      EV  N12569
## ..  ...   ... ...      ...       ...      ...       ...     ...     ...
## Variables not shown: flight (int), origin (chr), dest (chr), air_time
##   (dbl), distance (dbl), hour (dbl), minute (dbl)
```

- Now that the data is sorted by shortest to longest distance, we can use `slice()` extract the shortest flight per month.

```
slice(how_long, 1)
```

```
## Source: local data frame [12 x 16]
## Groups: month
##
##    year month day dep_time dep_delay arr_time arr_delay carrier tailnum
## 1  2013     1   3     2127        -2     2222        -2      EV  N13989
## 2  2013     2   1     2128        -1     2216        -8      EV  N13969
## 3  2013     3   2     1926        -3     2011       -12      EV  N21144
## 4  2013     4   6     1948        -2     2034       -10      EV  N29917
## 5  2013     5   1      935        -5     1101         6      9E  N8710A
## 6  2013     6   1      849        49      946        38      9E  N8794B
## 7  2013     7  27       NA        NA       NA        NA      US
## 8  2013     8   1      933        -7     1045        -6      9E  N8800G
## 9  2013     9   1     2056        21     2142       -19      9E  N8794B
## 10 2013    10   1      838        -7      924       -34      9E  N830AY
## 11 2013    11   1      904        19     1040        42      9E  N8944B
## 12 2013    12   1     1942        -3     2037       -24      9E  N8580A
## Variables not shown: flight (int), origin (chr), dest (chr), air_time
##   (dbl), distance (dbl), hour (dbl), minute (dbl)
```

- The `summarise()` verb allows you to calculate summary statistics for each group. This is probably the most common function that is used in conjnction with `group_by`. For example, the average distance flown per month.

```
summarise(by_month, avg_airtime = mean(distance, na.rm=TRUE))
```

```
## Source: local data frame [12 x 2]
##
##    month avg_airtime
## 1      1    1006.844
## 2      2    1000.982
## 3      3    1011.987
## 4      4    1038.733
## 5      5    1040.913
## 6      6    1057.125
## 7      7    1058.596
## 8      8    1062.138
## 9      9    1041.250
## 10    10    1038.876
## 11    11    1050.305
## 12    12    1064.656
```

Or simply the total nubmer of flights per month.

```r
summarize(by_month, count=n())
```

```
## Source: local data frame [12 x 2]
##
##    month count
## 1      1 27004
## 2      2 24951
## 3      3 28834
## 4      4 28330
## 5      5 28796
## 6      6 28243
## 7      7 29425
## 8      8 29327
## 9      9 27574
## 10    10 28889
## 11    11 27268
## 12    12 28135
```

5 Chaining Operations

Consider the following group of operations that take the data set `flights`, and produce a final data set (`a4`) that contains only the flights where the daily average delay is greater than a half hour.

```r
a1 <- group_by(flights, year, month, day)
a2 <- select(a1, arr_delay, dep_delay)
a3 <- summarise(a2,
                arr = mean(arr_delay, na.rm = TRUE),
                dep = mean(dep_delay, na.rm = TRUE))
a4 <- filter(a3, arr > 30 | dep > 30)
head(a4)
```

```
## Source: local data frame [6 x 5]
## Groups: year, month
```

```
##
##   year month day      arr      dep
## 1 2013     1  16 34.24736 24.61287
## 2 2013     1  31 32.60285 28.65836
## 3 2013     2  11 36.29009 39.07360
## 4 2013     2  27 31.25249 37.76327
## 5 2013     3   8 85.86216 83.53692
## 6 2013     3  18 41.29189 30.11796
```

It does the trick, but what if you don't want to save all the intermediate results (`a1 - a3`)? Well these verbs are `function`, so they can be wrapped inside other functions to create a nesting type structure.

```
filter(
  summarise(
    select(
      group_by(flights, year, month, day),
      arr_delay, dep_delay
    ),
    arr = mean(arr_delay, na.rm = TRUE),
    dep = mean(dep_delay, na.rm = TRUE)
  ),
  arr > 30 | dep > 30
)
```

Woah, that is HARD to read! This is difficult to read because the order of the operations is from inside to out, and the arguments are a long way away from the function. To get around this problem, dplyr provides the `%>%` operator. `x %>% f(y)` turns into `f(x, y)` so you can use it to rewrite multiple operations so you can read from left-to-right, top-to-bottom:

```
flights %>%
  group_by(year, month, day) %>%
  select(arr_delay, dep_delay) %>%
  summarise(
    arr = mean(arr_delay, na.rm = TRUE),
    dep = mean(dep_delay, na.rm = TRUE)
  ) %>%
  filter(arr > 30 | dep > 30)
```

```
## Source: local data frame [49 x 5]
## Groups: year, month
##
##    year month day      arr      dep
## 1  2013     1  16 34.24736 24.61287
## 2  2013     1  31 32.60285 28.65836
## 3  2013     2  11 36.29009 39.07360
## 4  2013     2  27 31.25249 37.76327
## 5  2013     3   8 85.86216 83.53692
## 6  2013     3  18 41.29189 30.11796
## 7  2013     4  10 38.41231 33.02368
## 8  2013     4  12 36.04814 34.83843
## 9  2013     4  18 36.02848 34.91536
## 10 2013     4  19 47.91170 46.12783
## ..  ...   ... ...      ...      ...
```

Another way you can read this is by thinking "and then" when you see the `%>%` operator. So the above code takes the data set flighs
.. and then groups by day
.. and then selects the delay varibles
.. and then calculates the means
.. and then filters on a delay over half hour.

The same 4 steps that resulted in the `a4` data set, but without all the intermediate data saved! This can be **very important** when dealing with Big Data. `R` stores all data in memory, so if your little computer only has 2G of RAM and you're working with a data set that is 500M in size, your computers memory will be used up fast. `a1` takes 500M, `a2` another 500M, by now your computer is getting slow. Make another copy at `a3` and it gets worse, `a4` now likely won't even be able to be created because you'll be out of memory.

5.1 Practice

Use dplyr functions and the pipe operator (`%>%`) to transform the following English sentences into R code:

1. Take the `hflights` data set and then ...
2. Add a variable named `diff` that is the result of subtracting `TaxiIn` from `TaxiOut`, and then ...
3. Pick all of the rows whose `diff` value does not equal `NA`, and then ...
4. Summarise the data set with a value named `avg` that is the mean diff value.

5. Store the result in the variable `p`.

5.1.0.1 Challenge Question: Do you drive or fly?

You can answer sophisticated questions by combining the verbs of dplyr. Now you will examine whether it sometimes makes sense to drive instead of fly. You will begin by making a data set that contains relevant variables. Then, you will find flights whose equivalent average velocity is lower than the velocity when traveling by car.

1. Define a data set named `d` that contains just the `Dest`, `UniqueCarrier`, `Distance`, and `ActualElapsedTime` columns of `hflights` as well as two additional variables: `RealTime` and `mph.RealTime` should be created as the actual elapsed time plus 100 minutes. This will be an estimate of how much time a person spends getting from point A to point B while flying, including getting to the airport, security checks, etc. mph will be the miles per hour that a person on the flight traveled based on the RealTime of the flight.

2. On many highways you can drive at 70 mph. Continue with `d` to calculate the following variables:

- `n_less`, the number of flights whose mph value does not equal NA that traveled at less than 70 mph in real time
- `n_dest`, the number of destinations that were traveled to at less than 70 mph
- `min_dist`, the minimum distance of these flights
- `max_dist`, the maximum distance of these flights.

This suggests that some flights might be less efficient than driving in terms of speed. But is speed all that matters? Flying imposes burdens on a traveler that driving does not. For example, airplane tickets are very expensive. Air travelers also need to limit what they bring on their trip and arrange for a pick up or a drop off. Given these burdens we might demand that a flight provide a large speed advantage over driving.

Let's define preferable flights as flights that are 150% faster than driving, i.e. that travel 105 mph or greater in real time. Also, assume that cancelled or diverted flights are less preferable than driving.

Write an adapted version of the solution to the previous exercise in an all-in-one fashion (i.e. in a single expression without intermediary variables, using %>%) to find:

- `n_non` - the number of non-preferable flights in hflights,
- `p_non` - the percentage of non-preferable flights in hflights,
- `n_dest` - the number of destinations that non-preferable flights traveled to,
- `min_dist` - the minimum distance that non-preferable flights traveled,
- `max_dist` - the maximum distance that non-preferable flights traveled.

To maintain readability in this advanced exercise, start your operations with a `select()` function to retain only the five columns that will be needed for the subsequent calculation steps.

** Whew! That was intense! Congratulations on making it this far! It takes a lot of concentration and puzzle solving to come up with a solution to get from the data you have to the data you want so that you can answer complex questions. **

6 Additional Resources

The ability to manipulate, arrange and clean up data is an extremely important skill to have. It is advised that you review at least one other tutorial method for using dplyr if you are still uncomfortable with it. Remember, it is all about practice. The more you use it the easier it will become!

- Interactive Data Camp lesson. Data Camp is like Swirl on steriods. A very awesome learning tool.
- Dplyr vignette
- Hands-on dplyr tutorial for faster data manipulation in R You Tube video by Data School
- This video recording given by Hadley Wickham at the Summer Institute for Statistics for Big Data, Module on Visualizing Big Data in July 2015
- These video recordingspart1/part2 from Hadley's tutorial at UseR! 2014 conference at UCLA.
- Michael Levy's talk at the Davis R User's group, October 2014.