Data Transformation with dplyr

Dr. Austin Brown

Kennesaw State University

Table of Contents

Introduction

- One of the most important aspects of a data analyst/scientist/applied statistician's job is getting the right data into the right format in order for it to be appropriately analyzed.
- For example, we may only need to work with a subset of rows which meet certain condition(s).
- While there are multiple ways to subset/transform data using R, the modern approach frequently employs a `tidyverse` package called "`dplyr`."

Filtering Rows

- We learned in the last class how to filter rows based on a single condition using `filter`. For today, let's use the `nycflights13::flights` dataframe. This dataframe contains lots of information about flights departing NYC in 2013.

```
flights <- nycflights13::flights
```

Filtering Rows

- Now, let's say I just want to focus my attention on those which departed in January. To do this, I can use the code below:

```
library(tidyverse)

jan_flights <- flights |>
  filter(month == 1)
```

Filtering Rows

- Okay, this is straightforward enough, but what if we wanted to select all flights on a particular day of the year? Say Fourth of July?
- Notice here, the ampersand serves as the "and" operator. If we use "and" then this means that `filter` will only select those rows which meet *both* conditions.

```
foj_flights <- flights |>
  filter(month == 7 & day == 4)
```

Filtering Rows

- Obviously, there are likely instances when the comparison we're wanting to make isn't just equals (==). R offers all of the main comparison operators that come standard in any type of programming language:

Symbol	Comparison
>	Greater Than
<	Less Than
>=	Greater Than or Equal to
<=	Less Than or Equal to
!=	Not Equal to
==	Equal to

Filtering Rows

- As shown in the Fourth of July example, we also have logical operators, like “and,” “or” and “not.”

<table>
<tr><th>Symbol</th><th>Operator</th><th>Example</th></tr>
<tr><td><code>&</code></td><td>And</td><td><code>month == 7 & month == 4</code></td></tr>
<tr><td><code>|</code></td><td>Or</td><td><code>month == 1 | month == 2</code></td></tr>
<tr><td><code>!</code></td><td>Not</td><td><code>month != 2</code></td></tr>
</table>

Filtering Rows

- So let's look at some examples of using this in action. Let's say I want to track flights over the holiday season so I want to filter just those flights who departed in November or December. There are a couple of different approaches:

```
holiday_cheer <- flights |>
  filter(month == 11 | month == 12)

holiday_cheer1 <- flights |>
  filter(month %in% c(11,12))
```

Filtering Rows

- We could also use some of our comparison operators. Perhaps we want to see which flights had either a 2+ hour arrival delay or a 2+ hour departure delay.

```
late_flights <- flights |>
  filter(arr_delay >= 120 | dep_delay >= 120)

late_flights1 <- flights |>
  filter(!(arr_delay < 120 & dep_delay < 120))
```

Filtering Rows

- Obviously to this point, we've been filtering based on numeric values, but there are likely lots of instances when we'd want to filter based on character/categorical variables, too. Let's say we want to filter those rows which had flights that left in April and had destinations of DEN, ATL, and DFW.

```r
new_df <- flights |>
  filter(month == 4 &
           dest %in% c("ATL","DEN","DFW"))
```

Filtering Rows

- Sometimes when you're doing more complex filtering, it can be useful to separate out each filtering operation, just to keep it clear to you what's going on. So in the previous example, we can call two `filter` functions and pipe (|>) them together. This may make things easier down the road, especially if debugging is necessary.

```
new_df1 <- flights |>
  filter(month == 4) |>
  filter(dest %in% c("ATL","DEN","DFW"))
```

Filtering Rows

- One thing that is important to point out is how `filter` handles `NA` values. By default, whatever logical argument you've entered into the `filter` function, it will return only those rows for which the argument is TRUE. `NA` values will not be returned unless you explicitly ask for them.

```r
## Won't Return NA ##
df <- tibble(x = c(1,NA,3))
df |> filter(x > 1)
```

```
# A tibble: 1 x 1
      x
  <dbl>
1     3
```

Filtering Rows

```
## Will Return NA ##
df |> filter(is.na(x) | x > 1)
```

```
# A tibble: 2 x 1
      x
  <dbl>
1    NA
2     3
```

Arranging Rows

- In some instances, we may have a need to order or arrange our dataframe. We can do this using `arrange`. Let's say we want to arrange our rows in descending order by the length of the arrival delay.

```
arr_delay_order <- flights |>
  arrange(desc(arr_delay))
```

Arranging Rows

- We can also arrange rows based on multiple columns. Here, the function will arrange on the leftmost column first, and then arrange within the first arranged column. So as a simple example:

```r
ex <- tibble(x = c(0,12,40,13,60,55),
             y = c("A","A","B","B","C","C"))
ex |> arrange(x,y)
```

```
# A tibble: 6 x 2
      x y
  <dbl> <chr>
1     0 A
2    12 A
3    13 B
4    40 B
5    55 C
6    60 C
```

Arranging Rows

- But if we flip the order, and order y in descending order, we get a different result

```
ex |>
  arrange(desc(y),x)
```

```
# A tibble: 6 x 2
      x y
  <dbl> <chr>
1    55 C
2    60 C
3    13 B
4    40 B
5     0 A
6    12 A
```

Finding Distinct Rows

- Occassionally, we need to know how many distinct rows we have in a dataset. To do this, we can make use of the `distinct` function.
- Suppose we want to remove duplicate rows, if any exist, from the `flights` dataframe:

Finding Distinct Rows

```r
flights |>
  distinct()
```

```
# A tibble: 336,776 x 19
    year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
   <int> <int> <int>    <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
 1  2013     1     1      517        515       2     830     819      11 UA
 2  2013     1     1      533        529       4     850     830      20 UA
 3  2013     1     1      542        540       2     923     850      33 AA
 4  2013     1     1      544        545      -1    1004    1022     -18 B6
 5  2013     1     1      554        600      -6     812     837     -25 DL
 6  2013     1     1      554        558      -4     740     728      12 UA
 7  2013     1     1      555        600      -5     913     854      19 B6
 8  2013     1     1      557        600      -3     709     723     -14 EV
 9  2013     1     1      557        600      -3     838     846      -8 B6
10  2013     1     1      558        600      -2     753     745       8 AA
# ... with 336,766 more rows, 9 more variables: flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dttm>, and abbreviated variable names
#   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
#   5: arr_delay
# i Use `print(n = ...)` to see more rows, and `colnames()` to see all variable names
```

Finding Distinct Rows

- Or maybe we want to find all unique pairs of origins and destinations:

```
flights |>
  distinct(origin,dest)
```

```
# A tibble: 224 x 2
   origin dest
   <chr>  <chr>
 1 EWR    IAH
 2 LGA    IAH
 3 JFK    MIA
 4 JFK    BQN
 5 LGA    ATL
 6 EWR    ORD
 7 EWR    FLL
 8 LGA    IAD
 9 JFK    MCO
10 LGA    ORD
# ... with 214 more rows
# i Use `print(n = ...)` to see more rows
```

Finding Distinct Rows

- Notice in the prior example, only the columns `origin` and `dest` were retained.

- If we wanted to keep all the columns, we can use the code:

```
flights |>
  distinct(origin,dest,.keep_all=T)
```

Finding Distinct Rows

```
# A tibble: 224 x 19
    year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
   <int> <int> <int>    <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
 1  2013     1     1      517        515       2     830     819      11 UA
 2  2013     1     1      533        529       4     850     830      20 UA
 3  2013     1     1      542        540       2     923     850      33 AA
 4  2013     1     1      544        545      -1    1004    1022     -18 B6
 5  2013     1     1      554        600      -6     812     837     -25 DL
 6  2013     1     1      554        558      -4     740     728      12 UA
 7  2013     1     1      555        600      -5     913     854      19 B6
 8  2013     1     1      557        600      -3     709     723     -14 EV
 9  2013     1     1      557        600      -3     838     846      -8 B6
10  2013     1     1      558        600      -2     753     745       8 AA
# ... with 214 more rows, 9 more variables: flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dttm>, and abbreviated variable names
#   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
#   5: arr_delay
# i Use `print(n = ...)` to see more rows, and `colnames()` to see all variable names
```

Selecting Columns

- We saw last week how we can use the `select` function to pick out specific columns that we'd like to use for future analysis. For example, if we want to work with just the destination, arrival delay and departure delay columns, we can do so by:

```
d_a_d <- flights |>
  select(dest,arr_delay,dep_delay)
```

Selecting Columns

- Now, this method of manually entering column names might not be too bad if we're just selecting a handful of columns. But if we have a really large dataset, this can become quite cumbersome. There are a couple of tricks we can employ that are especially useful if the columns you want (or don't want) appear sequentially in the dataframe:

```
## Select all columns b/w year and day (inclusive) ##
y_m_d <- flights |>
  select(year:day)
## Select all columns except ymd (inclusive) ##
ae_ymd <- flights |>
  select(!year:day)
```

Selecting Columns

- Suppose we were performing some analysis where we wanted just the numeric or character columns, similar to the `keep _NUMERIC_` call in a SAS data step.

```
num_cols <- flights |>
  select(where(is.numeric))
```

Selecting Columns

- If our column names follow particular naming conventions, we can use “helper” functions to aid in the selection process:
- `starts_with("abc")` will select columns whose names, you guessed it, start with the character string, “abc”
- `ends_with("xyz")` does the same thing except with those columns whose names end with “xyz”
- `contains("arr")` selects those columns whose names match the regular expression specified in the function
- `num_range("x",1:3)` will select those columns named `x1`, `x2`, and `x3`

Creating New Variables Using Mutate

- There are many, many times when we need to create a new column that is a function of existing columns, even something as summing two columns.

- For example, in our flights data, we have a variable called `air_time`, which is what it sounds like and is recorded in minutes. Suppose we wanted to create a new variable to estimate the average MPH during a particular flight. Well this is where `mutate` comes into play:

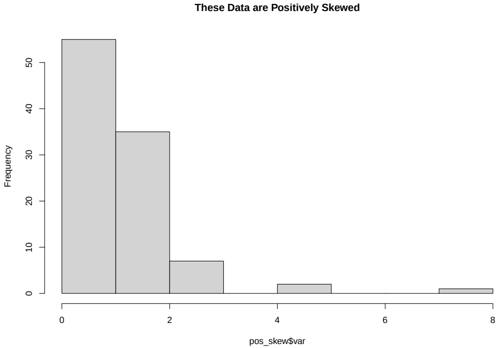
```
flights1 <- flights |>
  select(year:day,
                ends_with("delay"),
                distance,
                air_time
                ) |>
  mutate(speed = (distance/air_time)*60)
```

Create New Variables Using Mutate

- There are several useful functions which can aid in creating new columns.
- Obviously, we have our regular arithmetic operators: $+$, -, *, $/$, $\hat{}$. These can be used in conjunction with other R functions, such as `mean`, `sd`, `max`, `min`, etc.
- For example, let's say we want to standardize a variable.

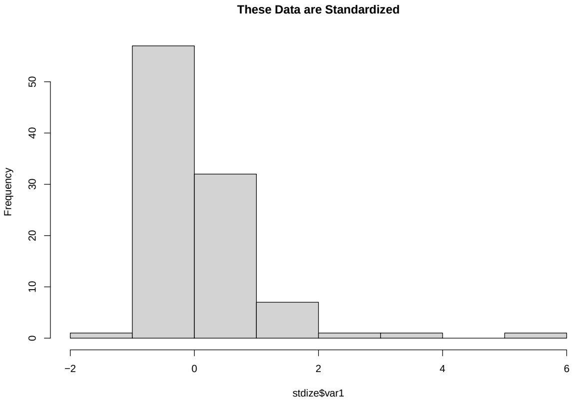
Create New Variables Using Mutate

```
## Positively Skewed Data ##
set.seed(123)
pos_skew <- tibble(var = rexp(100))
hist(pos_skew$var,main="These Data are Positively Skewed")
```



Create New Variables Using Mutate

```
stdize <- pos_skew |>
  mutate(var1 = (var - mean(var))/sd(var))
hist(stdize$var1,main="These Data are Standardized")
```



Grouped Summaries with Summarize

- Many, many times in research or industry scenarios, we are tasked with finding numerical summaries of data, whether those are grouped or otherwise. We can use the `summarize` function in conjunction with the `group_by` function to aid us in these aims.

- Let's say we want to find the mean of the `dep_delay` variable.

```
flights |>
  summarize(delay = mean(dep_delay,na.rm=T))
```

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

Grouped Summaries with Summarize

- Obviously, this previous example is not terribly efficient because we could easily obtain the same unconditional mean by using `mean(dep_delay,na.rm=T)`.
- The power of `summarize` really comes when using it in conjunction with `group_by`. For those of you who are old school R users, this is effectively the same as `aggregate` except a little handier as it can be used with other `tidyverse` functions in a seamless manner.
- Let's say we want to know the average departure delay by month.

Grouped Summaries with Summarize

```
flights |>
  group_by(month) |>
  summarize(mean_delay = mean(dep_delay,na.rm=T))
```

```
# A tibble: 12 x 2
   month mean_delay
   <int>      <dbl>
 1     1      10.0
 2     2      10.8
 3     3      13.2
 4     4      13.9
 5     5      13.0
 6     6      20.8
 7     7      21.7
 8     8      12.6
 9     9       6.72
10    10       6.24
11    11       5.44
12    12      16.6
```

Grouped Summaries with Summarize

- Now, what's cool about the piping function is that we can do lots of things in a single operation. Maybe we want to see the top five months in terms of longest mean departure delays:

```
flights |>
  group_by(month) |>
  summarize(mean_delay = mean(dep_delay,na.rm=T)) |>
  arrange(desc(mean_delay)) |>
  head(5)
```

```
# A tibble: 5 x 2
  month mean_delay
  <int>      <dbl>
1     7       21.7
2     6       20.8
3    12       16.6
4     4       13.9
5     3       13.2
```

Grouped Summaries with Summarize

- In another example, there is an extensive repository of baseball data in the `Lahman` package in R. Using the `Lahman::Batting` dataset, let's find the number of players in each season since 1990 who have hit more than 50 homeruns.
- In other words, what is the frequency, per year since 1990, of players who hit more than 50 homeruns.

Grouped Summaries with Summarize

```
bat <- Lahman::Batting
bat |>
  filter(yearID >= 1990) |>
  select(yearID,HR) |>
  filter(HR > 50) |>
  group_by(yearID) |>
  summarize(count = n()) |>
  arrange(desc(count))
```

Grouped Summaries with Summarize

- There is a lot of functionality and capability with the `summarize` function to provide a good deal of information in a cleaner coding format than some other R tricks.

- For example, let's say we want to count up the unique (i.e., distinct) number of airline carriers for a given destination

```
flights |>
  group_by(dest) |>
  summarize(carriers = n_distinct(carrier)) |>
  arrange(desc(carriers))
```