Data Transformation

Pegah Abedini

11/12/2021

chapter 3 :

Data Transformation with dplyr

Visualization is an important tool for insight generation, but it is rare that you get the data in exactly the right form you need. Often you'll need to create some **new variables** or **summaries**, or maybe you just want to **rename the variables** or **reorder** the observations in order to make the data a little easier to work with. You'll learn how to do all that (and more!) in this chapter, which will teach you how to transform your data using the **dplyr package** and a new dataset on flights departing New York City in 2013.

In this chapter we're going to focus on how to use the **dplyr package**, another core member of the **tidyverse**. We'll illustrate the key ideas using data from the **nycflights13 package**, and use **ggplot2** to help us understand the data.

```
library("nycflights13")
library("tidyverse")
```

```
## -- Attaching packages --------------------------------------- tidyverse
1.3.1 --
```

```
## v ggplot2 3.3.5     v purrr   0.3.4
## v tibble  3.1.4     v dplyr   1.0.7
## v tidyr   1.1.4     v stringr 1.4.0
## v readr   2.0.1     v forcats 0.5.1
```

```
## -- Conflicts ------------------------------------------
tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Take careful note of the conflicts message that's printed when you load the tidyverse. It tells you that dplyr overwrites some functions in base R. If you want to use the base version of these functions after loading dplyr, you'll need to use their full names: stats::filter() and stats::lag().

nycflights13 package :

```
flights
```

```
## # A tibble: 336,776 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
##  6  2013     1     1      554            558        -4      740
728
##  7  2013     1     1      555            600        -5      913
854
##  8  2013     1     1      557            600        -3      709
723
##  9  2013     1     1      557            600        -3      838
846
## 10  2013     1     1      558            600        -2      753
745
## # ... with 336,766 more rows, and 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>
```

to see all :

```
view(flights)
```

It prints differently because it's a tib- ble. Tibbles are data frames, but slightly tweaked to work better in the tidyverse. For now, you don't need to worry about the differences; we'll come back to tibbles in more detail in Part II.

You might also have noticed the row of three- (or four-) letter abbreviations under the column names. These describe the type of eac variable:

- int stands for integers.
- dbl stands for doubles, or real numbers.
- chr stands for character vectors, or string.
- dttm stands for date-times (a date + a time).

There are three other common types of variables that aren't used in this dataset but you'll encounter later in the book:

- lgl stands for logical, vectors that contain only TRUE or FALSE.

- fctr stands for factors, which R uses to represent categorical variables with fixed possible values.
- date stands for dates.

dplyr Basics

five key dplyr functions :

- Pick observations by their values (filter()).
- Reorder the rows (arrange()).
- Pick variables by their names (select()).
- Create new variables with functions of existing variables(mutate()).
- Collapse many values down to a single summary (summarize())
- group_by()

All verbs work similarly:

1. The first argument is a data frame.
2. The subsequent arguments describe what to do with the data frame, using the variable names.
3. The result is a new data frame.

Filter Rows with filter()

filter() allows you to subset observations based on their values.The first argument is the name of the data frame. The second and ubsequent arguments are the expressions that filter the data frame.For example, we can select all flights on January 1st with:

```
filter(flights, month == 1, day == 1)
```

```
## # A tibble: 842 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
##  6  2013     1     1      554            558        -4      740
728
##  7  2013     1     1      555            600        -5      913
854
##  8  2013     1     1      557            600        -3      709
723
##  9  2013     1     1      557            600        -3      838
846
## 10  2013     1     1      558            600        -2      753
745
## # ... with 832 more rows, and 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>
```

When you run that line of code, dplyr executes the filtering operation and returns a new data frame. dplyr functions never modify their inputs, so if you want to save the result, you'll need to use the assignment operator, <-:

```
jan1 <- filter(flights, month == 1, day == 1)
```

R either prints out the results, or saves them to a variable. If you want to do both, you can wrap the assignment in parentheses:

```
(dec25 <- filter(flights, month == 12, day == 25))
```

```
## # A tibble: 719 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    12    25      456            500        -4      649
651
##  2  2013    12    25      524            515         9      805
814
##  3  2013    12    25      542            540         2      832
850
##  4  2013    12    25      546            550        -4     1022
1027
##  5  2013    12    25      556            600        -4      730
745
##  6  2013    12    25      557            600        -3      743
752
##  7  2013    12    25      557            600        -3      818
831
##  8  2013    12    25      559            600        -1      855
856
##  9  2013    12    25      559            600        -1      849
855
## 10  2013    12    25      600            600         0      850
```

```
846
## # ... with 709 more rows, and 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>
```

###Comparisons To use filtering effectively, you have to know how to select the observations that you want using the comparison operators. R provides the standard suite: >, >=, <, <=, != (not equal), and == (equal).

```
filter(flights, month == 1)
```

```
## # A tibble: 27,004 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
##  6  2013     1     1      554            558        -4      740
728
##  7  2013     1     1      555            600        -5      913
854
##  8  2013     1     1      557            600        -3      709
723
##  9  2013     1     1      557            600        -3      838
846
## 10  2013     1     1      558            600        -2      753
745
## # ... with 26,994 more rows, and 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>
```

common mistake

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

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

```
1/49 * 49 == 1
```

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

use near()

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

## [1] TRUE

near(1 / 49 * 49, 1)

## [1] TRUE
```

Logical Operators

```
filter(flights, month == 11 | month == 12)

## # A tibble: 55,403 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    11     1        5           2359         6      352
345
##  2  2013    11     1       35           2250       105      123
2356
##  3  2013    11     1      455            500        -5      641
651
##  4  2013    11     1      539            545        -6      856
827
##  5  2013    11     1      542            545        -3      831
855
##  6  2013    11     1      549            600       -11      912
923
##  7  2013    11     1      550            600       -10      705
659
##  8  2013    11     1      554            600        -6      659
701
##  9  2013    11     1      554            600        -6      826
827
## 10  2013    11     1      554            600        -6      749
751
## # ... with 55,393 more rows, and 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>
```

The order of operations doesn't work like English. You can't write filter(flights, month == 11 | 12), which you might literally translate into "finds all flights that departed in November or December." Instead it finds all months that equal 11 | 12, an expression that evaluates to TRUE. In a numeric context (like here), TRUE becomes one, so this finds all flights in January, not November or December. This is quite confusing!

A useful shorthand for this problem is x %in% y.This will select every row where x is one of the values in y. We could use it to rewrite the preceding code:

```
nov_dec <- filter(flights, month %in% c(11, 12))
```

De Morgan's law: !(x & y) is the same as !x | !y, and !(x | y) is the same as !x & !y. For example, if you wanted to find flights that weren't delayed (on arrival or departure) by more than two hours, you could use either of the following two filters:

```
filter(flights, !(arr_delay > 120 | dep_delay > 120))
```

```
## # A tibble: 316,050 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
##  6  2013     1     1      554            558        -4      740
728
##  7  2013     1     1      555            600        -5      913
854
##  8  2013     1     1      557            600        -3      709
723
##  9  2013     1     1      557            600        -3      838
846
## 10  2013     1     1      558            600        -2      753
745
## # ... with 316,040 more rows, and 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>
```

```
filter(flights, arr_delay <= 120, dep_delay <= 120)
```

```
## # A tibble: 316,050 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
##  6  2013     1     1      554            558        -4      740
728
##  7  2013     1     1      555            600        -5      913
854
##  8  2013     1     1      557            600        -3      709
723
##  9  2013     1     1      557            600        -3      838
846
## 10  2013     1     1      558            600        -2      753
745
## # ... with 316,040 more rows, and 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>
```

Missing Values

One important feature of R that can make comparison tricky is missing values, or NAs ("not availables"). NA represents an unknown value so missing values are "contagious"; almost any operationinvolving an unknown value will also be unknown:

```
NA > 5
```

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

```
10 == NA
```

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

```
NA + 10
```

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

```
NA / 2
```

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

The most confusing result is this one:

```
NA == NA
```

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

It's easiest to understand why this is true with a bit more context:

Let x be Mary's age. We don't know how old she is.

x <- NA

Let y be John's age. We don't know how old he is.

y <- NA

Are John and Mary the same age?

x == y

[1] NA

We don't know!

If you want to determine if a **value is missing**, use is.na():

```
x <- NA
is.na(x)

## [1] TRUE
```

filter() only includes rows where the condition is TRUE; it excludes both FALSE and NA values. If you want to preserve missing values, ask for them explicitly:

df <- tibble(x = c(1, NA, 3))

filter(df, x > 1)

A tibble: 1 × 1

x

3

filter(df, is.na(x) | x > 1)

A tibble: 2 × 1

x

NA

3

Arrange Rows with arrange()

arrange() works similarly to filter() except that instead of selecting rows, it changes their order. It takes a data frame and a set of column names (or more complicated expressions) to order by.

```
arrange(flights, year, month, day)

## # A tibble: 336,776 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
##  6  2013     1     1      554            558        -4      740
728
##  7  2013     1     1      555            600        -5      913
854
##  8  2013     1     1      557            600        -3      709
723
##  9  2013     1     1      557            600        -3      838
846
## 10  2013     1     1      558            600        -2      753
745
## # ... with 336,766 more rows, and 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>
```

Use desc() to reorder by a column in descending order:

```
arrange(flights, desc(arr_delay))
```

```
## # A tibble: 336,776 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     9      641            900      1301     1242
1530
##  2  2013     6    15     1432           1935      1137     1607
2120
##  3  2013     1    10     1121           1635      1126     1239
1810
##  4  2013     9    20     1139           1845      1014     1457
2210
##  5  2013     7    22      845           1600      1005     1044
1815
##  6  2013     4    10     1100           1900       960     1342
2211
##  7  2013     3    17     2321            810       911      135
1020
##  8  2013     7    22     2257            759       898      121
```

```
1026
##  9  2013    12     5      756           1700       896     1058
2020
## 10  2013     5     3     1133           2055       878     1250
2215
## # ... with 336,766 more rows, and 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>
```

Missing values are always sorted at the end:

```
df <- tibble(x = c(5, 2, NA))
arrange(df, x)
```

```
## # A tibble: 3 x 1
##       x
##   <dbl>
## 1     2
## 2     5
## 3    NA
```

```
arrange(df, desc(x))
```

```
## # A tibble: 3 x 1
##       x
##   <dbl>
## 1     5
## 2     2
## 3    NA
```

Select Columns with select()

It's not uncommon to get datasets with hundreds or even thousands of variables. In this case, the first challenge is often narrowing in on the variables you're actually interested in. select() allows you to rapidly zoom in on a useful subset using operations based on the names of the variables.

select() is not terribly useful with the flight data because we only have 19 variables, but you can still get the general idea:

```
select(flights, year, month, day)
```

```
## # A tibble: 336,776 x 3
##     year month   day
##    <int> <int> <int>
##  1  2013     1     1
##  2  2013     1     1
##  3  2013     1     1
##  4  2013     1     1
##  5  2013     1     1
##  6  2013     1     1
```

```
##  7  2013     1     1
##  8  2013     1     1
##  9  2013     1     1
## 10  2013     1     1
## # ... with 336,766 more rows

select(flights, year:day)

## # A tibble: 336,776 x 3
##     year month   day
##    <int> <int> <int>
##  1  2013     1     1
##  2  2013     1     1
##  3  2013     1     1
##  4  2013     1     1
##  5  2013     1     1
##  6  2013     1     1
##  7  2013     1     1
##  8  2013     1     1
##  9  2013     1     1
## 10  2013     1     1
## # ... with 336,766 more rows

select(flights, -(year:day))

## # A tibble: 336,776 x 16
##    dep_time sched_dep_time dep_delay arr_time sched_arr_time arr_delay
carrier
##       <int>          <int>     <dbl>    <int>          <int>     <dbl>
<chr>
##  1      517            515         2      830            819        11 UA
##  2      533            529         4      850            830        20 UA
##  3      542            540         2      923            850        33 AA
##  4      544            545        -1     1004           1022       -18 B6
##  5      554            600        -6      812            837       -25 DL
##  6      554            558        -4      740            728        12 UA
##  7      555            600        -5      913            854        19 B6
##  8      557            600        -3      709            723       -14 EV
##  9      557            600        -3      838            846        -8 B6
## 10      558            600        -2      753            745         8 AA
## # ... with 336,766 more rows, and 9 more variables: flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance
<dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dttm>
```

There are a number of helper functions you can use within 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 Chapter 11.
- num_range("x", 1:3) matches x1, x2, and x3.

See ?select for more details.

select() can be used to rename variables, but it's rarely useful because it drops all of the variables not explicitly mentioned. Instead, use rename(), which is a variant of select() that keeps all the variables that aren't explicitly mentioned:

```
rename(flights, tail_num = tailnum)
```

```
## # A tibble: 336,776 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
##  6  2013     1     1      554            558        -4      740
728
##  7  2013     1     1      555            600        -5      913
854
##  8  2013     1     1      557            600        -3      709
723
##  9  2013     1     1      557            600        -3      838
846
## 10  2013     1     1      558            600        -2      753
745
## # ... with 336,766 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tail_num <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour
<dttm>
```

Another option is to use select() in conjunction with the every thing() helper. This is useful if you have a handful of variables you'd like to move to the start of the data frame:

```
select(flights, time_hour, air_time, everything())
```

```
## # A tibble: 336,776 x 19
##    time_hour           air_time  year month   day dep_time sched_dep_time
##    <dttm>                 <dbl> <int> <int> <int>    <int>          <int>
##  1 2013-01-01 05:00:00      227  2013     1     1      517            515
##  2 2013-01-01 05:00:00      227  2013     1     1      533            529
##  3 2013-01-01 05:00:00      160  2013     1     1      542            540
##  4 2013-01-01 05:00:00      183  2013     1     1      544            545
##  5 2013-01-01 06:00:00      116  2013     1     1      554            600
##  6 2013-01-01 05:00:00      150  2013     1     1      554            558
##  7 2013-01-01 06:00:00      158  2013     1     1      555            600
##  8 2013-01-01 06:00:00       53  2013     1     1      557            600
##  9 2013-01-01 06:00:00      140  2013     1     1      557            600
## 10 2013-01-01 06:00:00      138  2013     1     1      558            600
## # ... with 336,766 more rows, and 12 more variables: dep_delay <dbl>,
## #   arr_time <int>, sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
## #   flight <int>, tailnum <chr>, origin <chr>, dest <chr>, distance <dbl>,
## #   hour <dbl>, minute <dbl>
```

Add New Variables with mutate()

Besides selecting sets of existing columns, it's often useful to add new columns that are functions of existing columns. That's the job of mutate(). mutate() always adds new columns at the end of your dataset so we'll start by creating a narrower dataset so we can see the new variables. Remember that when you're in RStudio, the easiest way to see all the columns is View():

```
flights_sml <- select(flights,year:day,ends_with("delay"),distance,air_time)
flights_sml
```

```
## # A tibble: 336,776 x 7
##     year month   day dep_delay arr_delay distance air_time
##    <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl>
##  1  2013     1     1         2        11     1400      227
##  2  2013     1     1         4        20     1416      227
##  3  2013     1     1         2        33     1089      160
##  4  2013     1     1        -1       -18     1576      183
##  5  2013     1     1        -6       -25      762      116
##  6  2013     1     1        -4        12      719      150
##  7  2013     1     1        -5        19     1065      158
##  8  2013     1     1        -3       -14      229       53
##  9  2013     1     1        -3        -8      944      140
## 10  2013     1     1        -2         8      733      138
## # ... with 336,766 more rows
```

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

```
## # A tibble: 336,776 x 9
##     year month   day dep_delay arr_delay distance air_time  gain speed
##    <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl> <dbl> <dbl>
##  1  2013     1     1         2        11     1400      227     9  370.
```

```
##  2  2013     1     1         4         20     1416      227    16  374.
##  3  2013     1     1         2         33     1089      160    31  408.
##  4  2013     1     1        -1        -18     1576      183   -17  517.
##  5  2013     1     1        -6        -25      762      116   -19  394.
##  6  2013     1     1        -4         12      719      150    16  288.
##  7  2013     1     1        -5         19     1065      158    24  404.
##  8  2013     1     1        -3        -14      229       53   -11  259.
##  9  2013     1     1        -3         -8      944      140    -5  405.
## 10  2013     1     1        -2          8      733      138    10  319.
## # ... with 336,766 more rows
```

If you only want to keep the new variables,** use transmute()**:

```
transmute(flights,gain = arr_delay - dep_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
```

Useful Creation Functions

- Arithmetic operators +, -, *, /, ^

These are all vectorized, using the so-called “recycling rules.” If one parameter is shorter than the other, it will be automatically extended to be the same length. This is most useful when one of the arguments is a single number: air_time / 60, hours * 60 + minute, etc.

- Modular arithmetic %/% (integer division) and %% (remainder).For example, in the flights dataset, you can compute hour and minute from dep_time with:

```
transmute(flights,dep_time,hour = dep_time %/% 100,minute = dep_time %% 100)
```

```
## # A tibble: 336,776 x 3
##    dep_time  hour minute
##       <int> <dbl>  <dbl>
##  1      517     5     17
##  2      533     5     33
##  3      542     5     42
##  4      544     5     44
##  5      554     5     54
##  6      554     5     54
```

```
##  7      555     5     55
##  8      557     5     57
##  9      557     5     57
## 10      558     5     58
## # ... with 336,766 more rows
```

Logs log(), log2(), log10()

Logarithms are an incredibly useful transformation for dealing with data that ranges across multiple orders of magnitude. They also convert multiplicative relationships to additive, a feature we'll come back to in Part IV.

lead() and lag()

- lead function :lead function shifted our vector one element to the right side. cut off the first value and added an NA at the end.
- lag function : the lag function shifted our vector one element to the left. cut off the last value and appended an NA at the beginning.

x -lag(x)) / (x != lag(x)) / group_by()

```
(x <- 1:10)

##  [1]  1  2  3  4  5  6  7  8  9 10

lead(x)

##  [1]  2  3  4  5  6  7  8  9 10 NA

lag(x)

##  [1] NA  1  2  3  4  5  6  7  8  9
```

Cumulative

cumsum(x)

cumprod(x)

cummax(x)

cummin(x)

```
x <- 1:10
cumsum(x)

##  [1]  1  3  6 10 15 21 28 36 45 55

cummean(x)

##  [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5
```

Ranking

There are a number of ranking functions, but you should start with min_rank(). It does the most usual type of ranking.

```
y <- c(1, 2, 2, NA, 3, 4)
min_rank(y)

## [1]  1  2  2 NA  4  5

min_rank(desc(y))

## [1]  5  3  3 NA  2  1
```

Exercises

1. Find all flights that:
    a. Had an arrival delay of two or more hours

```
filter(flights, arr_delay >= 120)

## # A tibble: 10,200 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      811            630       101     1047
830
##  2  2013     1     1      848           1835       853     1001
1950
##  3  2013     1     1      957            733       144     1056
853
##  4  2013     1     1     1114            900       134     1447
1222
##  5  2013     1     1     1505           1310       115     1638
1431
##  6  2013     1     1     1525           1340       105     1831
1626
##  7  2013     1     1     1549           1445        64     1912
1656
##  8  2013     1     1     1558           1359       119     1718
1515
##  9  2013     1     1     1732           1630        62     2028
1825
## 10  2013     1     1     1803           1620       103     2008
1750
## # ... with 10,190 more rows, and 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>
```

    b. Flew to Houston The flights that flew to Houston are those flights where the destination (dest) is either “IAH” or “HOU”

```
filter(flights, dest == "IAH" | dest == "HOU")
```

```
## # A tibble: 9,313 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      623            627        -4      933
932
##  4  2013     1     1      728            732        -4     1041
1038
##  5  2013     1     1      739            739         0     1104
1038
##  6  2013     1     1      908            908         0     1228
1219
##  7  2013     1     1     1028           1026         2     1350
1339
##  8  2013     1     1     1044           1045        -1     1352
1351
##  9  2013     1     1     1114            900       134     1447
1222
## 10  2013     1     1     1205           1200         5     1503
1505
## # ... with 9,303 more rows, and 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>
```

c.Were operated by United, American, or Delta The carrier code for Delta is “DL”, for American is “AA”, and for United is “UA”. Using these carriers codes, we check whether carrier is one of those.

```
filter(flights, carrier %in% c("AA", "DL", "UA"))
```

```
## # A tibble: 139,504 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      554            600        -6      812
837
```

```
##  5  2013     1     1      554            558        -4      740
728
##  6  2013     1     1      558            600        -2      753
745
##  7  2013     1     1      558            600        -2      924
917
##  8  2013     1     1      558            600        -2      923
937
##  9  2013     1     1      559            600        -1      941
910
## 10  2013     1     1      559            600        -1      854
902
## # ... with 139,494 more rows, and 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>
```

e. Arrived more than two hours late, but didn't leave late

Flights that arrived more than two hours late, but didn't leave late will have an arrival delay of more than 120 minutes (arr_delay > 120) and a non-positive departure delay (dep_delay <= 0).

```
filter(flights, arr_delay > 120, dep_delay <= 0)
```

```
## # A tibble: 29 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    27     1419           1420        -1     1754
1550
##  2  2013    10     7     1350           1350         0     1736
1526
##  3  2013    10     7     1357           1359        -2     1858
1654
##  4  2013    10    16      657            700        -3     1258
1056
##  5  2013    11     1      658            700        -2     1329
1015
##  6  2013     3    18     1844           1847        -3       39
2219
##  7  2013     4    17     1635           1640        -5     2049
1845
##  8  2013     4    18      558            600        -2     1149
850
##  9  2013     4    18      655            700        -5     1213
950
## 10  2013     5    22     1827           1830        -3     2217
2010
```

```
## # ... with 19 more rows, and 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>
```

Grouped Summaries with summarize()

```
summarize(flights, delay = mean(dep_delay, na.rm = TRUE))

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

summarize () is not terribly useful unless we pair it with group_by (). This changes the unit of analysis from the complete dataset to indi- vidual groups. For example,if we applied exactly the same code to a data frame grouped by date,we get the average delay per date:

```
by_day <- group_by(flights, year, month, day)
summarize(by_day, delay = mean(dep_delay, na.rm = TRUE))

## `summarise()` has grouped output by 'year', 'month'. You can override
using the `.groups` argument.

## # A tibble: 365 x 4
## # Groups:   year, month [12]
##     year month   day delay
##    <int> <int> <int> <dbl>
##  1  2013     1     1 11.5
##  2  2013     1     2 13.9
##  3  2013     1     3 11.0
##  4  2013     1     4  8.95
##  5  2013     1     5  5.73
##  6  2013     1     6  7.15
##  7  2013     1     7  5.42
##  8  2013     1     8  2.55
##  9  2013     1     9  2.28
## 10  2013     1    10  2.84
## # ... with 355 more rows
```

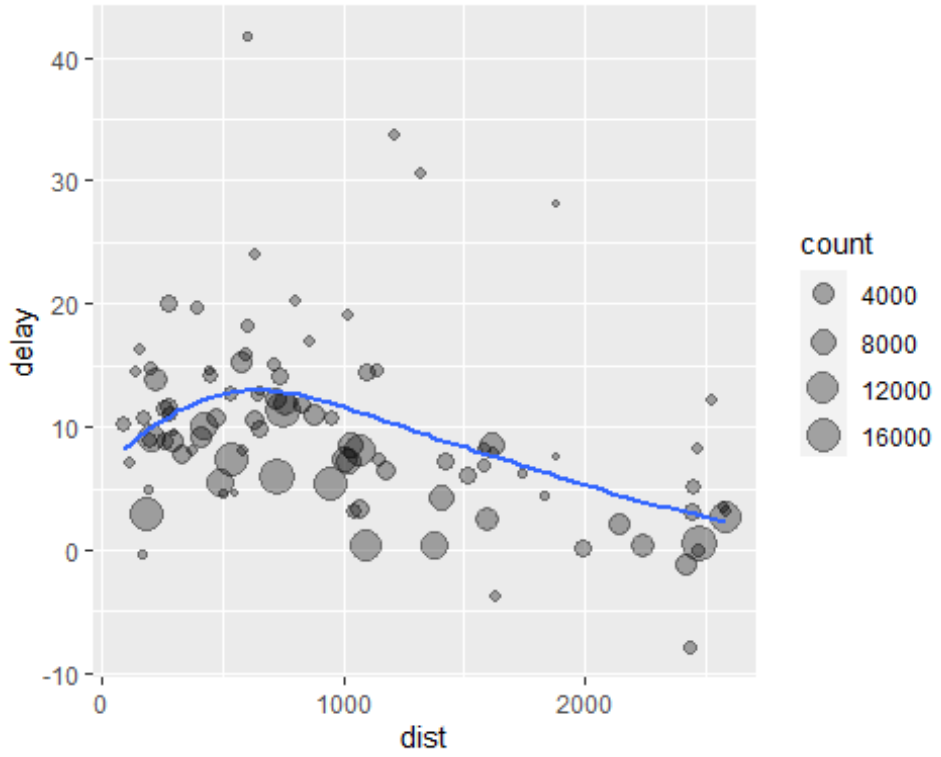
Together group_by() and summarize() provide one of the tools that you'll use most commonly when working with dplyr: grouped summaries. But before we go any further with this, we need to introduce a powerful new idea: the pipe.

Combining Multiple Operations with the Pipe

Imagine that we want to explore the relationship between the distance and average delay for each location. Using what you know about dplyr, you might write code like this:

```
by_dest <- group_by(flights, dest)
delay <- summarize(by_dest,count = n(),dist = mean(distance, na.rm = TRUE),
delay = mean(arr_delay, na.rm = TRUE))
```

```
delay <- filter(delay, count > 20, dest != "HNL")
ggplot(data = delay, mapping = aes(x = dist, y = delay)) +
geom_point(aes(size = count), alpha = 1/3) +
geom_smooth(se = FALSE)

## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



This code is a little frustrating to write because we have to give each intermediate data frame a name, even though we don't care about it. Naming things is hard, so this slows down our analysis. There's another way to tackle the same problem with the pipe,* %>% *:

```
delays <- flights %>%
group_by(dest) %>%
summarize(count = n(),dist = mean(distance, na.rm = TRUE), delay =
mean(arr_delay, na.rm = TRUE)) %>%
filter(count > 20, dest != "HNL")
```

This focuses on the transformations, not what's being transformed, which makes the code easier to read. You can read it as a series of imperative statements: group, then summarize, then filter. As sug- gested by this reading, a good way to pronounce %>% when reading code is "then."

Behind the scenes, x %>% f(y) turns into f(x, y), and x %>% f(y) %>% g(z) turns into g(f(x, y), z), and so on. You can use the pipe to rewrite multiple operations in a way that you can read left-toright, top-to-bottom. We'll use piping frequently from now on because it considerably improves the readability of code, and we'll come back to it in more detail in Chapter 14.

Working with the pipe is one of the key criteria for belonging to the tidyverse. The only exception is ggplot2: it was written before the pipe was discovered. Unfortunately, the next iteration of ggplot2, ggvis, which does use the pipe, isn't ready for prime time yet.

Missing Values

You may have wondered about the na.rm argument we used earlier. What happens if we don't set it?

```
flights %>%
group_by(year, month, day) %>%
summarize(mean = mean(dep_delay))

## `summarise()` has grouped output by 'year', 'month'. You can override
using the `.groups` argument.

## # A tibble: 365 x 4
## # Groups:   year, month [12]
##     year month   day  mean
##    <int> <int> <int> <dbl>
##  1  2013     1     1    NA
##  2  2013     1     2    NA
##  3  2013     1     3    NA
##  4  2013     1     4    NA
##  5  2013     1     5    NA
##  6  2013     1     6    NA
##  7  2013     1     7    NA
##  8  2013     1     8    NA
##  9  2013     1     9    NA
## 10  2013     1    10    NA
## # ... with 355 more rows
```

why?

use na.rm=TRUE

```
flights %>%
group_by(year, month, day) %>%
summarize(mean = mean(dep_delay, na.rm = TRUE))

## `summarise()` has grouped output by 'year', 'month'. You can override
using the `.groups` argument.

## # A tibble: 365 x 4
## # Groups:   year, month [12]
##     year month   day  mean
##    <int> <int> <int> <dbl>
##  1  2013     1     1 11.5
##  2  2013     1     2 13.9
##  3  2013     1     3 11.0
##  4  2013     1     4  8.95
##  5  2013     1     5  5.73
```

```
##  6  2013     1     6  7.15
##  7  2013     1     7  5.42
##  8  2013     1     8  2.55
##  9  2013     1     9  2.28
## 10  2013     1    10  2.84
## # ... with 355 more rows
```

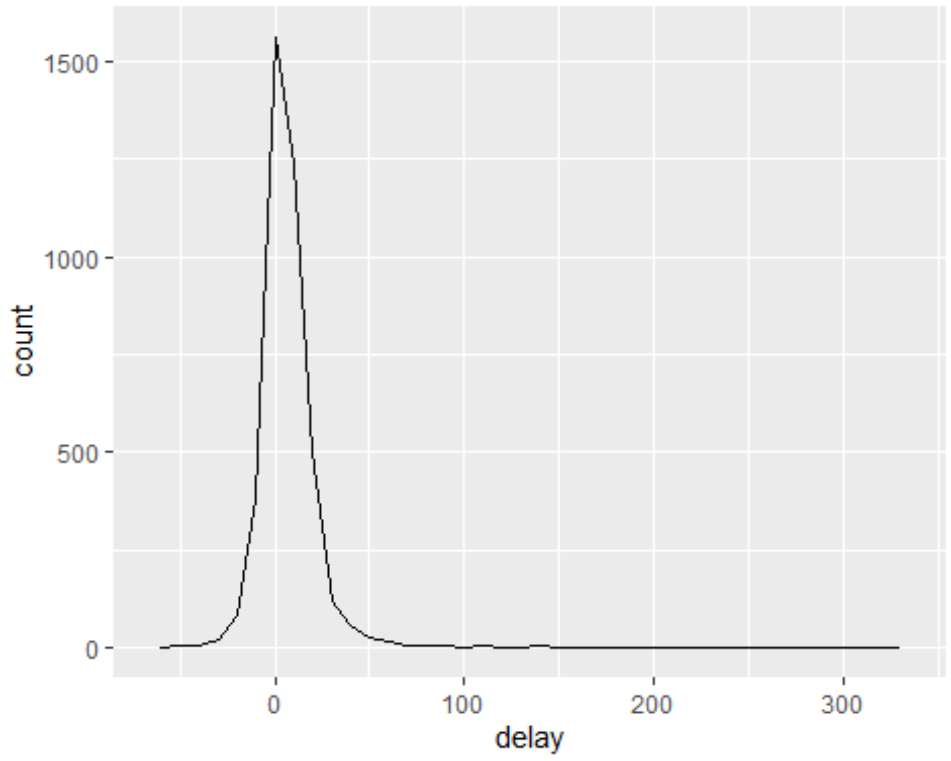
In this case, where missing values represent cancelled flights,, we could also tackle the problem by first removing the cancelled flights. We'll save this dataset so we can reuse it in the next few examples:

```
not_cancelled <- flights %>%
filter(!is.na(dep_delay), !is.na(arr_delay))

not_cancelled %>%
group_by(year, month, day) %>%
summarize(mean = mean(dep_delay))

## `summarise()` has grouped output by 'year', 'month'. You can override
using the `.groups` argument.

## # A tibble: 365 x 4
## # Groups:   year, month [12]
##     year month   day  mean
##    <int> <int> <int> <dbl>
##  1  2013     1     1 11.4
##  2  2013     1     2 13.7
##  3  2013     1     3 10.9
##  4  2013     1     4  8.97
##  5  2013     1     5  5.73
##  6  2013     1     6  7.15
##  7  2013     1     7  5.42
##  8  2013     1     8  2.56
##  9  2013     1     9  2.30
## 10  2013     1    10  2.84
## # ... with 355 more rows
```

Count

Whenever you do any aggregation, it's always a good idea to include either a count (n()), or a count of nonmissing values (sum(!is.na(x))). That way you can check that you're not drawing conclusions based on very small amounts of data.

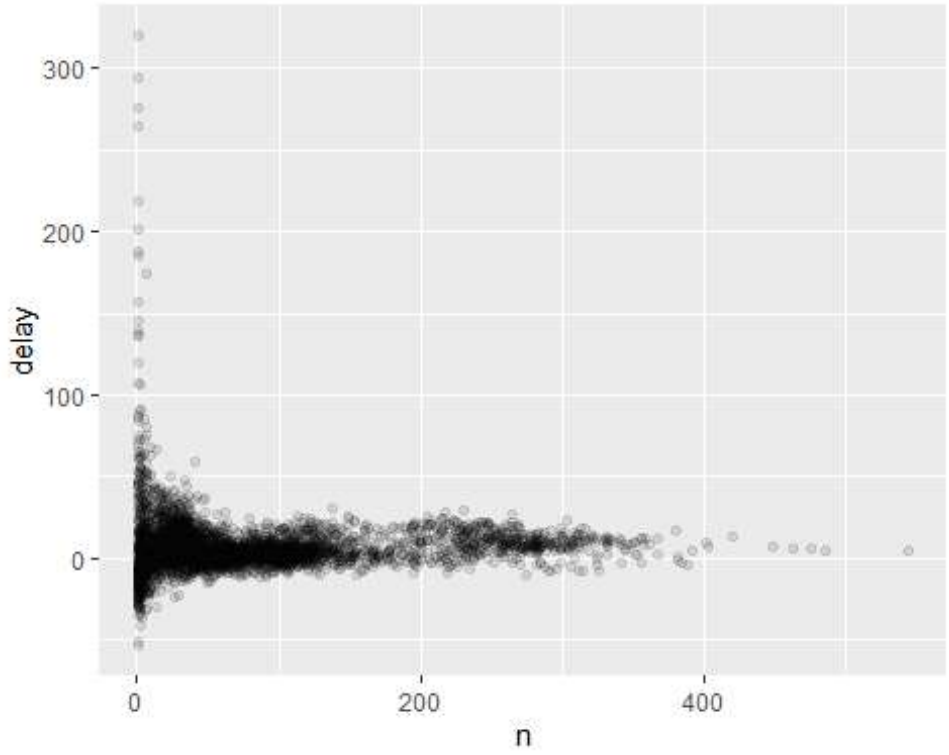
```
delays <- not_cancelled %>%
group_by(tailnum) %>%
summarize(delay = mean(arr_delay))
ggplot(data = delays, mapping = aes(x = delay)) +
geom_freqpoly(binwidth = 10)
```



Wow, there are some planes that have an average delay of 5 hours (300 minutes)!

The story is actually a little more nuanced. We can get more insight if we draw a scatterplot of number of flights versus average delay:

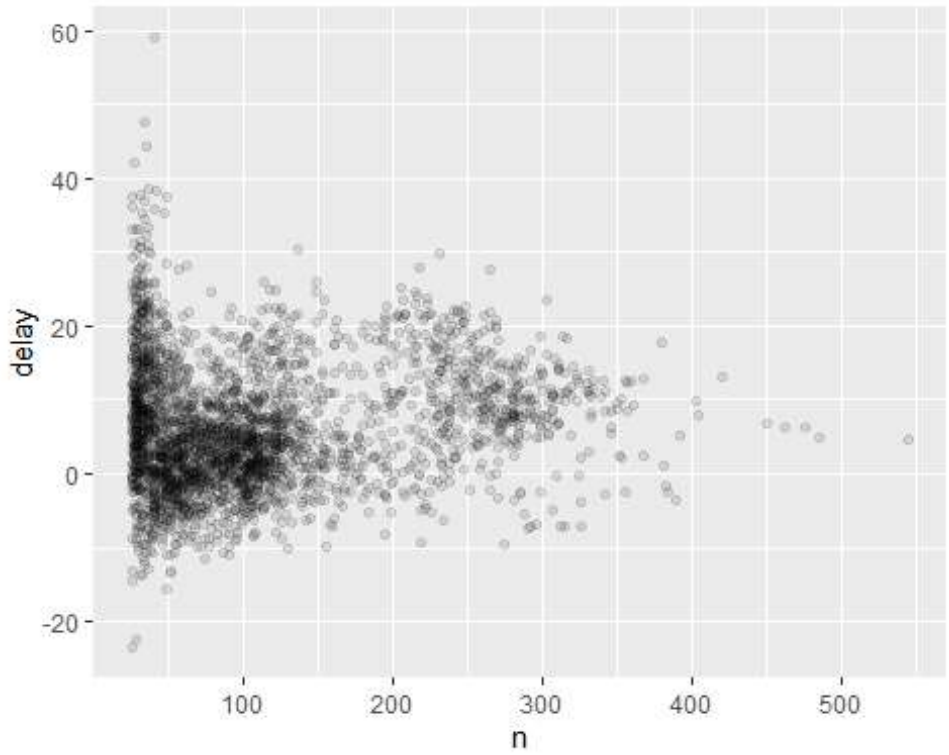
```
delays <- not_cancelled %>%
group_by(tailnum) %>%
summarize(delay = mean(arr_delay, na.rm = TRUE),n = n())
ggplot(data = delays, mapping = aes(x = n, y = delay)) +
geom_point(alpha = 1/10)
```



Not surprisingly, there is much greater variation in the average delay when there are few flights. The shape of this plot is very characteristic: whenever you plot a mean (or other summary) versus group size, you'll see that the variation decreases as the sample size increases.

When looking at this sort of plot, it's often useful to filter out the groups with the smallest numbers of observations, so you can see more of the pattern and less of the extreme variation in the smallest groups. This is what the following code does.

```
delays %>%
filter(n > 25) %>%
ggplot(mapping = aes(x = n, y = delay)) +
geom_point(alpha = 1/10)
```

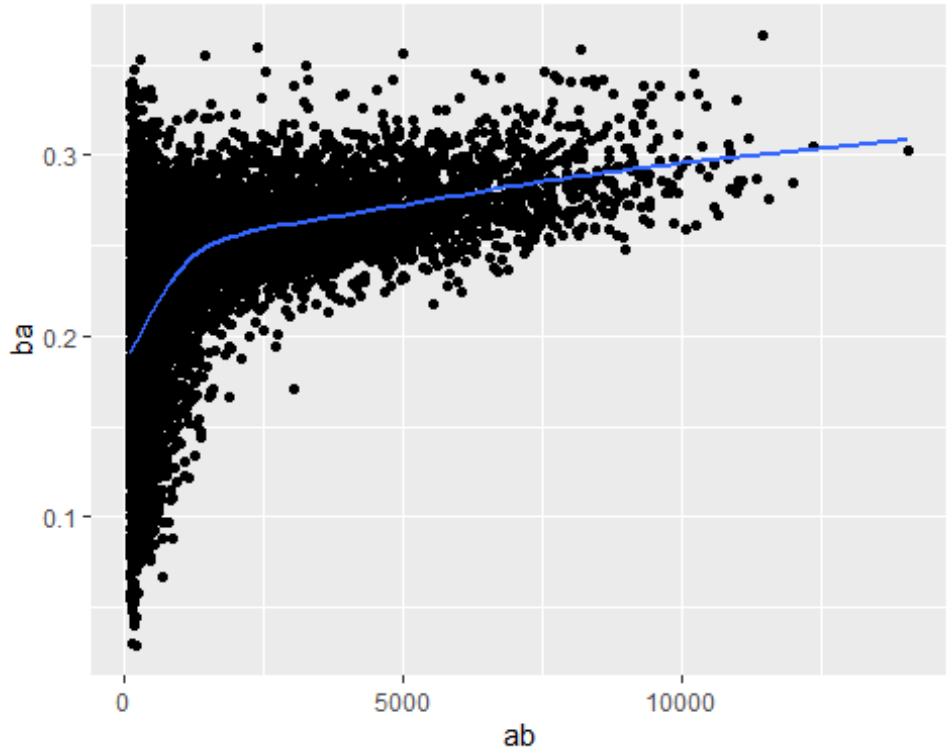


Lahman package

When I plot the skill of the batter (measured by the batting average, ba) against the number of opportunities to hit the ball (measured by at bat, ab), you see two patterns:

- As above, the variation in our aggregate decreases as we get more data points.
- There's a positive correlation between skill (ba) and opportunities to hit the ball (ab). This is because teams control who gets to play, and obviously they'll pick their best players:

```
batting <- as_tibble(Lahman::Batting)
batters <- batting %>%
group_by(playerID) %>%
summarize(
ba = sum(H, na.rm = TRUE) / sum(AB, na.rm = TRUE),
ab = sum(AB, na.rm = TRUE))
batters %>%
filter(ab > 100) %>%
ggplot(mapping = aes(x = ab, y = ba)) +
geom_point() +
geom_smooth(se = FALSE)

## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```



Useful Summary Functions

Measures of location : We've used mean(x), but median(x) is also useful. The mean is the sum divided by the length; the median is a value where 50% of x is above it, and 50% is below it.

```
not_cancelled %>%
group_by(year, month, day) %>%
summarize(
# average delay:
avg_delay1 = mean(arr_delay),
# average positive delay:
avg_delay2 = mean(arr_delay[arr_delay > 0])
)
```

```
## `summarise()` has grouped output by 'year', 'month'. You can override
using the `.groups` argument.
```

```
## # A tibble: 365 x 5
## # Groups:   year, month [12]
##     year month   day avg_delay1 avg_delay2
##    <int> <int> <int>      <dbl>      <dbl>
##  1  2013     1     1     12.7         32.5
##  2  2013     1     2     12.7         32.0
##  3  2013     1     3      5.73        27.7
##  4  2013     1     4     -1.93        28.3
##  5  2013     1     5     -1.53        22.6
```

```
##  6  2013     1     6      4.24         24.4
##  7  2013     1     7     -4.95         27.8
##  8  2013     1     8     -3.23         20.8
##  9  2013     1     9     -0.264        25.6
## 10  2013     1    10     -5.90         27.3
## # ... with 355 more rows
```

Measures of spread sd(x), IQR(x), mad(x)

The mean squared deviation, or standard deviation or sd for short, is the standard measure of spread. The interquartile range IQR() and median absolute deviation mad(x) are robust equivalents that may be more useful if you have outliers:

```
not_cancelled %>%
group_by(dest) %>%
summarize(distance_sd = sd(distance)) %>%
arrange(desc(distance_sd))
```

```
## # A tibble: 104 x 2
##    dest  distance_sd
##    <chr>       <dbl>
##  1 EGE         10.5
##  2 SAN         10.4
##  3 SFO         10.2
##  4 HNL         10.0
##  5 SEA          9.98
##  6 LAS          9.91
##  7 PDX          9.87
##  8 PHX          9.86
##  9 LAX          9.66
## 10 IND          9.46
## # ... with 94 more rows
```

Measures of rank min(x), quantile(x, 0.25), max(x) Quantiles are a generalization of the median. For example, quan tile(x, 0.25) will find a value of x that is greater than 25% of the values, and less than the remaining 75%:

```
not_cancelled %>%
group_by(year, month, day) %>%
summarize(
first = min(dep_time),
last = max(dep_time))
```

```
## `summarise()` has grouped output by 'year', 'month'. You can override
using the `.groups` argument.
```

```
## # A tibble: 365 x 5
## # Groups:   year, month [12]
##     year month   day first  last
##    <int> <int> <int> <int> <int>
##  1  2013     1     1   517  2356
```

```
##  2  2013     1     2    42  2354
##  3  2013     1     3    32  2349
##  4  2013     1     4    25  2358
##  5  2013     1     5    14  2357
##  6  2013     1     6    16  2355
##  7  2013     1     7    49  2359
##  8  2013     1     8   454  2351
##  9  2013     1     9     2  2252
## 10  2013     1    10     3  2320
## # ... with 355 more rows
```

Measures of position first(x),last(x)

```
not_cancelled %>%
group_by(year, month, day) %>%
summarize(
first_dep = first(dep_time),
last_dep = last(dep_time)
)
```

```
## `summarise()` has grouped output by 'year', 'month'. You can override
using the `.groups` argument.
```

```
## # A tibble: 365 x 5
## # Groups:   year, month [12]
##     year month   day first_dep last_dep
##    <int> <int> <int>     <int>    <int>
##  1  2013     1     1       517     2356
##  2  2013     1     2        42     2354
##  3  2013     1     3        32     2349
##  4  2013     1     4        25     2358
##  5  2013     1     5        14     2357
##  6  2013     1     6        16     2355
##  7  2013     1     7        49     2359
##  8  2013     1     8       454     2351
##  9  2013     1     9         2     2252
## 10  2013     1    10         3     2320
## # ... with 355 more rows
```

Counts

To count the number of non-missing values, use*sum(!is.na(x))*. To count the number of distinct (unique) values, use *n_distinct(x)*

```
# Which destinations have the most carriers?
not_cancelled %>%
group_by(dest) %>%
summarize(carriers = n_distinct(carrier)) %>%
arrange(desc(carriers))
```

```
## # A tibble: 104 x 2
##    dest  carriers
```

```
##    <chr>    <int>
##  1 ATL          7
##  2 BOS          7
##  3 CLT          7
##  4 ORD          7
##  5 TPA          7
##  6 AUS          6
##  7 DCA          6
##  8 DTW          6
##  9 IAD          6
## 10 MSP          6
## # ... with 94 more rows
```

```
not_cancelled %>%
count(dest)
```

```
## # A tibble: 104 x 2
##    dest      n
##    <chr> <int>
##  1 ABQ     254
##  2 ACK     264
##  3 ALB     418
##  4 ANC       8
##  5 ATL   16837
##  6 AUS    2411
##  7 AVL     261
##  8 BDL     412
##  9 BGR     358
## 10 BHM     269
## # ... with 94 more rows
```

```
not_cancelled %>%
count(tailnum, wt = distance)
```

```
## # A tibble: 4,037 x 2
##    tailnum      n
##    <chr>    <dbl>
##  1 D942DN    3418
##  2 N0EGMQ  239143
##  3 N10156  109664
##  4 N102UW   25722
##  5 N103US   24619
##  6 N104UW   24616
##  7 N10575  139903
##  8 N105UW   23618
##  9 N107US   21677
## 10 N108UW   32070
## # ... with 4,027 more rows
```

sum(x > 10), mean(y == 0) When used with numeric functions, TRUE is converted to 1 and FALSE to 0. This makes sum() and mean() very useful: sum(x) gives the number of TRUEs in x

What proportion of flights are delayed by more than an hour?

```
not_cancelled %>%
group_by(year, month, day) %>%
summarize(hour_perc = mean(arr_delay > 60))

## `summarise()` has grouped output by 'year', 'month'. You can override
using the `.groups` argument.

## # A tibble: 365 x 4
## # Groups:   year, month [12]
##     year month   day hour_perc
##    <int> <int> <int>     <dbl>
##  1  2013     1     1    0.0722
##  2  2013     1     2    0.0851
##  3  2013     1     3    0.0567
##  4  2013     1     4    0.0396
##  5  2013     1     5    0.0349
##  6  2013     1     6    0.0470
##  7  2013     1     7    0.0333
##  8  2013     1     8    0.0213
##  9  2013     1     9    0.0202
## 10  2013     1    10    0.0183
## # ... with 355 more rows
```