Note 16. tidyverse: dplyr

Introduction to Statistical Programming

Chanmin Kim

Department of Statistics
Sungkyunkwan University

2022 Spring

Introduction

“Small data”

Figure 1:

iris data

```
#> # A tibble: 150 x 5
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Spec
#>          <dbl>       <dbl>        <dbl>       <dbl> <fct
#> 1          5.1         3.5          1.4         0.2 seto
#> 2          4.9         3            1.4         0.2 seto
#> 3          4.7         3.2          1.3         0.2 seto
#> 4          4.6         3.1          1.5         0.2 seto
#> # ... with 146 more rows
```

Boston housing data

```
#> # A tibble: 506 x 14
#>       crim    zn indus  chas   nox    rm   age   dis   ra
#> *   <dbl> <dbl> <dbl> <int> <dbl> <dbl> <dbl> <dbl> <int
#> 1 0.00632    18  2.31     0 0.538  6.58  65.2  4.09
#> 2 0.0273      0  7.07     0 0.469  6.42  78.9  4.97
#> 3 0.0273      0  7.07     0 0.469  7.18  61.1  4.97
#> 4 0.0324      0  2.18     0 0.458  7.00  45.8  6.06
#> # ... with 502 more rows, and 3 more variables: black <d
#> #   medv <dbl>
```

Prostate Cancer Data

```
#> # A tibble: 97 x 9
#>    lcavol lweight   age  lbph   svi   lcp gleason pgg45
#>     <dbl>   <dbl> <dbl> <dbl> <dbl> <dbl>   <dbl> <dbl>
#> 1 -0.580    2.77    50 -1.39     0 -1.39       6     0 -
#> 2 -0.994    3.32    58 -1.39     0 -1.39       6     0 -
#> 3 -0.511    2.69    74 -1.39     0 -1.39       7    20 -
#> 4 -1.20     3.28    58 -1.39     0 -1.39       6     0 -
#> # ... with 93 more rows
```

dplyr package

dplyr is a grammar of data manipulation, providing a consistent set of verbs that help you solve the most common data manipulation challenges:

- `mutate()` adds new variables that are functions of existing variables
- `select()` picks variables based on their names.
- `filter()` picks cases based on their values.
- `summarise()` reduces multiple values down to a single summary.
- `arrange()` changes the ordering of the rows.

Installation

```
# (recommended) The easiest way to get dplyr is to install
# tidyverse:
install.packages("tidyverse")

# Alternatively, install just dplyr:
install.packages("dplyr")
```

Working with data

When working with data you must:

- Figure out what you want to do.
- Describe those tasks in the form of a computer program.
- Execute the program.

Doing it with `dplyr`

The dplyr package makes these steps fast and easy:

- By constraining your options, it helps you think about your data manipulation challenges.
- It provides simple "verbs", functions that correspond to the most common data manipulation tasks, to help you translate your thoughts into code.
- It uses efficient backends, so you spend less time waiting for the computer.

Using `dplyr` in action

Data: nycflights13

- To explore the basic data manipulation verbs of dplyr, we'll use `nycflights13::flights`.
- This dataset contains all 336776 flights that departed from New York City in 2013.
- The data comes from the US Bureau of Transportation Statistics, and is documented in `?nycflights13`

```
library(nycflights13)
dim(flights)
#> [1] 336776     19
flights
#> # A tibble: 336,776 x 19
#>    year month   day dep_time sched_dep_time dep_delay a
#>   <int> <int> <int>    <int>          <int>     <dbl>
#> 1  2013     1     1      517            515         2
#> 2  2013     1     1      533            529         4
#> 3  2013     1     1      542            540         2
#> 4  2013     1     1      544            545        -1
```

A short note on tibble

- Note that `nycflights13::flights` is a "tibble," a modern reimagining of the data frame.
- It's particularly useful for large datasets because it only prints the first few rows.
- You can learn more about tibbles at http://tibble.tidyverse.org; in particular you can convert data frames to tibbles with `as_tibble()`.

We will learn about the tools that have been developed to handle large datasets later.

Single table verbs

Dplyr aims to provide a function for each basic verb of data manipulation:

- `filter()` to select cases based on their values.
- `arrange()` to reorder the cases.
- `select()` and `rename()` to select variables based on their names.
- `mutate()` and `transmute()` to add new variables that are functions of existing variables.
- `summarise()` to condense multiple values to a single value.
- `sample_n()` and `sample_frac()` to take random samples.

Filter rows with `filter()`

- `filter()` allows you to select a subset of rows in a data frame.
- Like all single verbs, the first argument is the tibble (or data frame).
- The second and subsequent arguments refer to variables within that data frame, selecting rows where the expression is `TRUE`.

Example: `filter()`

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 ar
#>   <int> <int> <int>    <int>          <int>     <dbl>
#> 1  2013     1     1      517            515         2
#> 2  2013     1     1      533            529         4
#> 3  2013     1     1      542            540         2
#> 4  2013     1     1      544            545        -1
#> # ... with 838 more rows, and 12 more variables: sched_
#> #   arr_delay <dbl>, carrier <chr>, flight <int>, tailn
#> #   origin <chr>, dest <chr>, air_time <dbl>, distance
#> #   minute <dbl>, time_hour <dttm>
```

Compare it with the base R code

It is rougly equivalent to this base R code:

```
flights[flights$month == 1 & flights$day == 1, ]
```

Arrange rows with `arrange()`

- `arrange()` works similarly to `filter()` except that instead of filtering or selecting rows, it reorders them.
- It takes a data frame (or anything in the similar class), and a set of column names (or more complicated expressions) to order by.
- If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns:

```
arrange(flights, year, month, day)
#> # A tibble: 336,776 x 19
#>    year month   day dep_time sched_dep_time dep_delay ar
#>   <int> <int> <int>    <int>          <int>     <dbl>
#> 1  2013     1     1      517            515         2
#> 2  2013     1     1      533            529         4
#> 3  2013     1     1      542            540         2
#> 4  2013     1     1      544            545        -1
#> # ... with 336,772 more rows, and 12 more variables: sch
```

Select columns with `select()`

- Often you work with large datasets with many columns but only a few are actually of interest to you.
- `select()` allows you to rapidly zoom in on a useful subset using operations that usually only work on numeric variable positions:

```
# Select columns by name
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
#> # ... with 336,772 more rows
# Select all columns between year and day (inclusive)
select(flights, year:day)
```

One step up with `select()`

- There are a number of helper functions you can use within `select()`, like `starts_with()`, `ends_with()`, `matches()` and `contains()`.
- You can rename variables with `select()` by using named arguments:

```
select(flights, tail_num = tailnum)
#> # A tibble: 336,776 x 1
#>   tail_num
#>   <chr>
#> 1 N14228
#> 2 N24211
#> 3 N619AA
#> 4 N804JB
#> # ... with 336,772 more rows
```

- But because `select()` drops all the variables not explicitly mentioned, it's not that useful. Instead, use `rename()`:


HL7219
Airbus A300
HL7463
BOEING 747

```
rename(flights, tail_num = tailnum)
#> # A tibble: 336,776 x 19
#>    year month   day dep_time sched_dep_time dep_delay ar
#>   <int> <int> <int>    <int>          <int>     <dbl>
#> 1  2013     1     1      517            515         2
#> 2  2013     1     1      533            529         4
#> 3  2013     1     1      542            540         2
#> 4  2013     1     1      544            545        -1
#> # ... with 336,772 more rows, and 12 more variables: sch
#> #   arr_delay <dbl>, carrier <chr>, flight <int>, tail_n
#> #   origin <chr>, dest <chr>, air_time <dbl>, distance <
#> #   minute <dbl>, time_hour <dttm>
```

Add new columns with `mutate()`

Besides selecting sets of existing columns, it's often useful to add new columns that are functions of existing columns. This is the job of `mutate()`:

```
mutate(flights, gain = arr_delay - dep_delay, speed = dista
#> # A tibble: 336,776 x 21
#>    year month   day dep_time sched_dep_time dep_delay ar
#>   <int> <int> <int>    <int>          <int>     <dbl>
#> 1  2013     1     1      517            515         2
#> 2  2013     1     1      533            529         4
#> 3  2013     1     1      542            540         2
#> 4  2013     1     1      544            545        -1
#> # ... with 336,772 more rows, and 14 more variables: sch
#> #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnu
#> #   origin <chr>, dest <chr>, air_time <dbl>, distance
#> #   minute <dbl>, time_hour <dttm>, gain <dbl>, speed <d
```

`dplyr::mutate()` is similar to the base `transform()`, but allows you to refer to columns that you've just created:

```
mutate(flights, gain = arr_delay - dep_delay, gain_per_hour
#> # A tibble: 336,776 x 21
#>    year month   day dep_time sched_dep_time dep_delay ar
#>   <int> <int> <int>    <int>          <int>     <dbl>
#> 1  2013     1     1      517            515         2
#> 2  2013     1     1      533            529         4
#> 3  2013     1     1      542            540         2
#> 4  2013     1     1      544            545        -1
#> # ... with 336,772 more rows, and 14 more variables: sch
#> #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnu
#> #   origin <chr>, dest <chr>, air_time <dbl>, distance <
#> #   minute <dbl>, time_hour <dttm>, gain <dbl>, gain_per
```

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

```
transmute(flights, gain = arr_delay - dep_delay, gain_per_h
#> # A tibble: 336,776 x 2
#>    gain gain_per_hour
#>   <dbl>         <dbl>
#> 1     9          2.38
#> 2    16          4.23
#> 3    31         11.6
#> 4   -17         -5.57
#> # ... with 336,772 more rows
```

Summarise values with `summarise()`

The last verb is `summarise()`. It collapses a data frame to a single row.

```
summarise(flights, delay = mean(dep_delay, na.rm = TRUE))
#> # A tibble: 1 x 1
#>   delay
#>   <dbl>
#> 1  12.6
```

It's not that useful until we learn the `group_by()` – but we will learn.

Randomly sample rows with `sample_n()` and `sample_frac()`

You can use `sample_n()` and `sample_frac()` to take a random sample of rows: use `sample_n()` for a fixed number and `sample_frac()` for a fixed fraction.

```
sample_n(flights, 10)
#> # A tibble: 10 x 19
#>    year month   day dep_time sched_dep_time dep_delay ar
#>   <int> <int> <int>    <int>          <int>     <dbl>
#> 1  2013    10     1      822            825        -3
#> 2  2013     8     2      712            715        -3
#> 3  2013     5    10     1309           1315        -6
#> 4  2013    10    28     2002           1930        32
#> # ... with 6 more rows, and 12 more variables: sched_ar
#> #   arr_delay <dbl>, carrier <chr>, flight <int>, tailn
#> #   origin <chr>, dest <chr>, air_time <dbl>, distance
#> #   minute <dbl>, time_hour <dttm>
sample_frac(flights, 0.01)
```

Commonalities

You may have noticed that the syntax and function of all these verbs are very similar:

- The first argument is a data frame.
- The subsequent arguments describe what to do with the data frame. You can refer to columns in the data frame directly without using $.
- The result is a new data frame

Together these properties make it easy to chain together multiple simple steps to achieve a complex result.

Five pillars of of data manipulation

At the most basic level, you can only alter a tidy data frame in five useful ways:

- you can reorder the rows (`arrange()`)
- pick observations and variables of interest (`filter()` and `select()`)
- add new variables that are functions of existing variables (`mutate()`)
- collapse many values to a summary (`summarise()`).

The remainder of the language comes from applying the five functions to different types of data.

Grouped operations

- The dplyr verbs become very powerful when you apply them to groups of observations within a dataset.
- In dplyr, you do this with the `group_by()` function.
- It breaks down a dataset into specified groups of rows.
- When you then apply the verbs above on the resulting object they'll be automatically applied "by group".

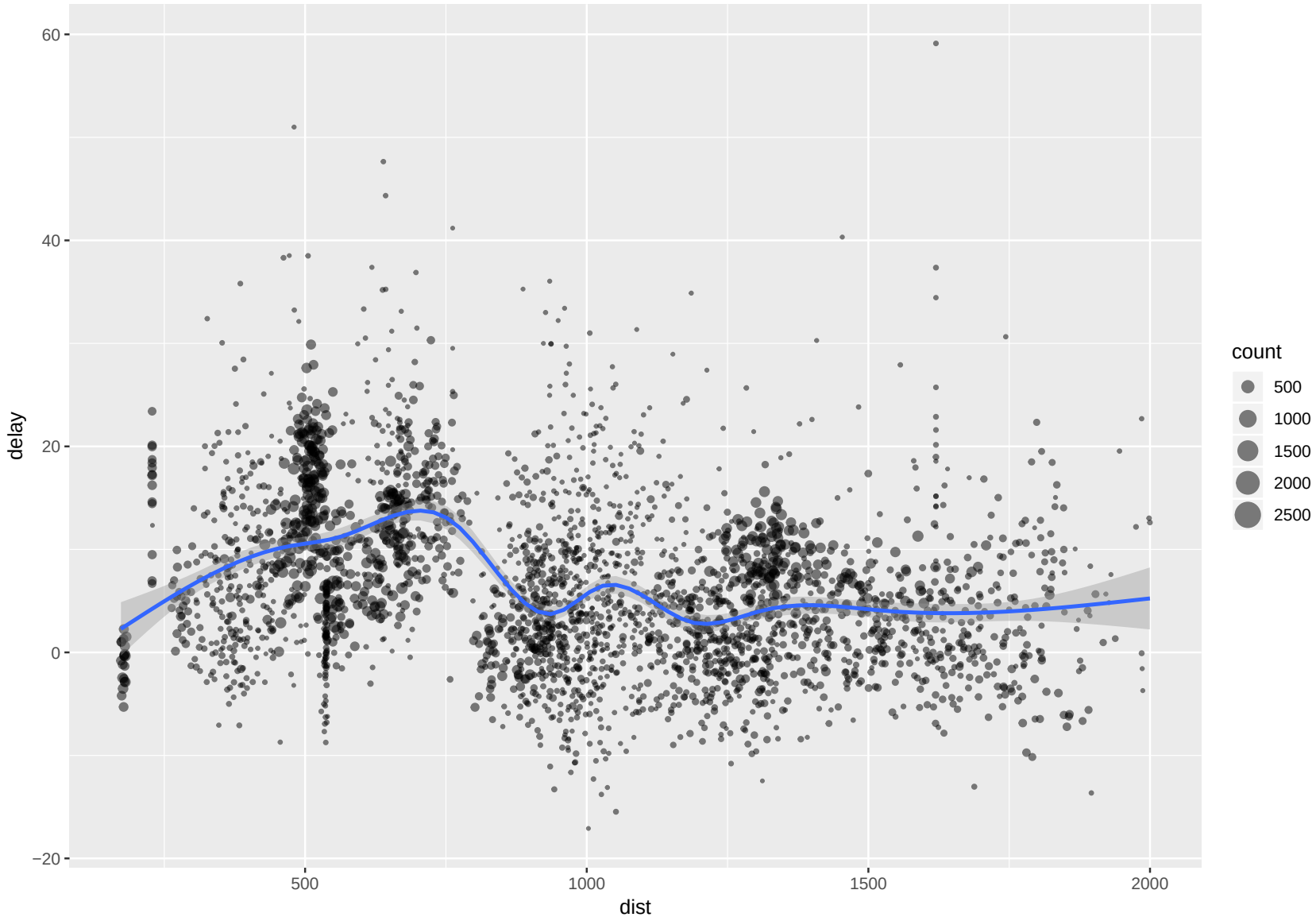
Grouping affects the verbs

- grouped `select()` is the same as ungrouped `select()`, except that grouping variables are always retained.
- grouped `arrange()` is the same as ungrouped; unless you set `.by_group = TRUE`, in which case it orders first by the grouping variables
- `mutate()` and `filter()` are most useful in conjunction with window functions (like `rank()`, or `min(x) == x`). They are described in detail in `vignette("window-functions")`.
- `sample_n()` and `sample_frac()` sample the specified number/fraction of rows in each group.
- `summarise()` computes the summary for each group.

Example

In the following example, we split the complete dataset into individual planes and then summarise each plane by counting the number of flights (`count = n()`) and computing the average distance (`dist = mean(distance, na.rm = TRUE)`) and arrival delay (`delay = mean(arr_delay, na.rm = TRUE)`).

```
by_tailnum <- group_by(flights, tailnum)
delay <- summarise(by_tailnum,
  count = n(), # we will learn about this in the next slid
  dist = mean(distance, na.rm = TRUE),
  delay = mean(arr_delay, na.rm = TRUE))
delay <- filter(delay, count > 20, dist < 2000)
```


60
40
20
0
−20
delay
500
1000
1500
2000
dist
count
500
1000
1500
2000
2500

Summarise with the summaries

- You use `summarise()` with **aggregate functions**, which take a vector of values and return a single number.
- There are many useful examples of such functions in base R like `min()`, `max()`, `mean()`, `sum()`, `sd()`, `median()`, and `IQR()`.
- dplyr provides a handful of others:
  - `n()`: the number of observations in the current group
  - `n_distinct(x)`:the number of unique values in `x`.
  - `first(x)`, `last(x)` and `nth(x, n)` - these work similarly to `x[1]`, `x[length(x)]`, and `x[n]` but give you more control over the result if the value is missing.

For example, we could use these to find the number of planes and the number of flights that go to each possible destination:

```
destinations <- group_by(flights, dest)
summarise(destinations, planes = n_distinct(tailnum), fligh
#> # A tibble: 105 x 3
#>    dest  planes flights
#>    <chr>  <int>   <int>
#> 1 ABQ      108     254
#> 2 ACK       58     265
#> 3 ALB      172     439
#> 4 ANC        6       8
#> # ... with 101 more rows
```

Recursive bottom-up

When you group by multiple variables, each summary peels off one level of the grouping. That makes it easy to progressively roll-up a dataset:

```
daily <- group_by(flights, year, month, day)
(per_day <- summarise(daily, flights = n()))
#> # A tibble: 365 x 4
#> # Groups:   year, month [?]
#>    year month   day flights
#>   <int> <int> <int>   <int>
#> 1  2013     1     1     842
#> 2  2013     1     2     943
#> 3  2013     1     3     914
#> 4  2013     1     4     915
#> # ... with 361 more rows
(per_month <- summarise(per_day, flights = sum(flights)))
#> # A tibble: 12 x 3
#> # Groups:   year [?]
#>    year month flights
```

Mutating operations

- Mutate semantics are quite different from selection semantics.
- `select()` expects column names or positions:

```
df <- select(flights, year:dep_time)
```

- Whereas `select()` expects column names or positions, `mutate()` expects *column vectors*.
- For `mutate()` on the other hand, column symbols represent the actual column vectors stored in the tibble.

```
mutate(df, "year", 2)
#> # A tibble: 336,776 x 6
#>    year month   day dep_time `"year"`   `2`
#>   <int> <int> <int>    <int> <chr>    <dbl>
#> 1  2013     1     1      517 year         2
#> 2  2013     1     1      533 year         2
#> 3  2013     1     1      542 year         2
#> 4  2013     1     1      544 year         2
#> # ... with 336,772 more rows
```

`mutate()` gets length-1 vectors that it interprets as new columns in the data frame.

More sensible examples:

```
mutate(df, new_year = year + 10)
#> # A tibble: 336,776 x 5
#>    year month   day dep_time new_year
#>   <int> <int> <int>    <int>    <dbl>
#> 1  2013     1     1      517     2023
#> 2  2013     1     1      533     2023
#> 3  2013     1     1      542     2023
#> 4  2013     1     1      544     2023
#> # ... with 336,772 more rows
mutate(df, sqrt_dep_time = sqrt(dep_time))
#> # A tibble: 336,776 x 5
#>    year month   day dep_time sqrt_dep_time
#>   <int> <int> <int>    <int>          <dbl>
#> 1  2013     1     1      517           22.7
#> 2  2013     1     1      533           23.1
#> 3  2013     1     1      542           23.3
#> 4  2013     1     1      544           23.3
#> # ... with 336,772 more rows
```

Readablity

The dplyr API is functional in the sense that function calls don't have side-effects. You must always save their results. This doesn't lead to particularly elegant code, especially if you want to do many operations at once. You either have to do it step-by-step:

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

Or if you don't want to name the intermediate results, you need to wrap the function calls inside each other:

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

Better readability with piping

- This is difficult to read because the order of the operations is from inside to out.
- dplyr provides the %>% operator from magrittr. `x %>% f(y)` turns into `f(x, y)` so you can use it to rewrite multiple operations that you can read left-to-right, top-to-bottom:

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

We will learn more about piping operator (%>%).