Prediction and Machine Learning for Economics

Data cleaning & wrangling

Sebastian Fossati

University of Alberta | E593 | 2023

Outline

Gapminder data

We'll be working with data from Hans Rosling's Gapminder project.

An excerpt of these data can be accessed through an R package called `gapminder`, cleaned and assembled by Jenny Bryan at UBC.

In the console: `install.packages("gapminder")`

Load the package and data:

```
# load library
library(gapminder)
```

Gapminder data

The data frame we will work with is called `gapminder`, available once you have loaded the package. Let's see its structure:

```
# check out data
str(gapminder)
```

```
## tibble [1,704 x 6] (S3: tbl_df/tbl/data.frame)
##  $ country  : Factor w/ 142 levels "Afghanistan",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ continent: Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 3 3 3 3 3 ...
##  $ year     : int [1:1704] 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
##  $ lifeExp  : num [1:1704] 28.8 30.3 32 34 36.1 ...
##  $ pop      : int [1:1704] 8425333 9240934 10267083 11537966 13079460 14880372 128
##  $ gdpPercap: num [1:1704] 779 821 853 836 740 ...
```

What's interesting here?

The `gapminder` dataset contains **panel data** on life expectancy, population size, and GDP per capita for 142 countries since the 1950s.

Remarks:

- **factor** variables `country` and `continent`
  - factors are categorical data with an underlying numeric representation
- many observations: $n = 1704$ rows
- a nested/hierarchical structure: `year` in `country` in `continent`

Indices and dimensions

In base R, there are two main ways to access elements of objects: square brackets (`[]` or `[[]]`) and `$`. How you access an object depends on its *dimensions*.

Dataframes have *2* dimensions: **rows** and **columns**.

- square brackets allow us to numerically **subset** in the format of `object[row, column]`
- leaving the row or column place empty selects *all* elements of that dimension

Indices and dimensions

```r
gapminder[1,] # First row
```

```
## # A tibble: 1 x 6
##   country     continent  year lifeExp     pop gdpPercap
##   <fct>       <fct>     <int>   <dbl>   <int>     <dbl>
## 1 Afghanistan Asia       1952    28.8 8425333      779.
```

Indices and dimensions

The **colon operator** (`:`) generates a vector using the sequence of integers from its first argument to its second. `1:3` is equivalent to `c(1,2,3)`.

```
gapminder[1:3, 3:4] # first three rows, third and fourth column
```

```
## # A tibble: 3 x 2
##    year lifeExp
##   <int>   <dbl>
## 1  1952    28.8
## 2  1957    30.3
## 3  1962    32.0
```

Dataframes and names

Columns in dataframes can also be accessed using their names with the `$` extract operator. This will return the column as a vector:

```
gapminder$gdpPercap[1:10]
```

```
##  [1] 779.4 820.9 853.1 836.2 740.0 786.1 978.0 852.4 649.3 635.3
```

Note here I *also* used brackets to select just the first 10 elements of that column.

Dataframes and names

You can mix subsetting formats! In this case I provided only a single value (no column index) because **vectors** have *only one dimension* (length).

If you try to subset something and get a warning about "incorrect number of dimensions", check your subsetting!

Indexing by expression

We can also index using expressions—logical *tests*.

```r
gapminder[gapminder$year==1952, ]
```

```
## # A tibble: 142 x 6
##    country     continent  year lifeExp      pop gdpPercap
##    <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
##  1 Afghanistan Asia       1952    28.8  8425333      779.
##  2 Albania     Europe     1952    55.2  1282697     1601.
##  3 Algeria     Africa     1952    43.1  9279525     2449.
##  4 Angola      Africa     1952    30.0  4232095     3521.
##  5 Argentina   Americas   1952    62.5 17876956     5911.
##  6 Australia   Oceania    1952    69.1  8691212    10040.
##  7 Austria     Europe     1952    66.8  6927772     6137.
##  8 Bahrain     Asia       1952    50.9   120447     9867.
##  9 Bangladesh  Asia       1952    37.5 46886859      684.
## 10 Belgium     Europe     1952    68    8730405     8343.
## # ... with 132 more rows
```

How expressions work

What does `gapminder$year==1952` actually do?

```
head(gapminder$year==1952, 20) # display first 20 elements
```

```
##  [1]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [13]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

It returns a vector of `TRUE` or `FALSE` values.

When used with the subset operator (`[]`), elements for which a `TRUE` is given are returned while those corresponding to `FALSE` are dropped.

Logical operators

We used `==` for testing “equals”: `year == "1952"`.

There are many other logical operators:

- `!=`: not equal to
- `>`, `>=`, `<`, `<=`: less than, less than or equal to, etc.
- `%in%`: used with checking equal to one of several values

Or we can combine multiple logical conditions:

- `&`: both conditions need to hold (AND)
- `|`: at least one condition needs to hold (OR)
- `!`: inverts a logical condition (`TRUE` becomes `FALSE`, `FALSE` becomes `TRUE`)

Sidenote: missing values

Missing values are coded as `NA` entries without quotes:

```
vector_w_missing <- c(1, 2, NA, 4, 5, 6, NA)
```

Even one `NA` "poisons the well": You'll get `NA` out of your calculations unless you remove them manually or use the extra argument `na.rm = TRUE` in some functions:

```
mean(vector_w_missing)
```

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

```
mean(vector_w_missing, na.rm = TRUE)
```

```
## [1] 3.6
```

Finding missing values

You can't test for missing values by seeing if they "equal" (`==`) `NA`:

```
vector_w_missing == NA
```

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

But you can use the `is.na()` function:

```
is.na(vector_w_missing)
```

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

We can use subsetting to get the equivalent of `na.rm = TRUE`:

```
mean(vector_w_missing[!is.na(vector_w_missing)])
```

```
## [1] 3.6
```

- `!` *reverses* a logical condition, read the above as "subset *not* `NA`"

Outline

Tidyverse

R packages for data science

The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures.

Install the complete tidyverse with:

```
install.packages("tidyverse")
```

Tidyverse vs. base R

There is often a direct correspondence between a tidyverse command and its base R equivalent.

tidyverse	base
`?readr::read_csv`	`?utils::read.csv`
`?dplyr::if_else`	`?base::ifelse`
`?tibble::tibble`	`?base::data.frame`

Etc.

If you call up the above examples, you'll see that the tidyverse alternative typically offers some enhancements or other useful options (and sometimes restrictions) over its base counterpart.

Tidyverse packages

Let's load the tidyverse meta-package and check the output.

```
#
library(tidyverse)
```

We have loaded a number of packages (which could also be loaded individually): **ggplot2**, **tibble**, **dplyr**, etc.

Tidyverse packages (cont.)

The tidyverse actually comes with a lot more packages than those that are just loaded automatically.

```
tidyverse_packages()
```

```
##  [1] "broom"         "cli"           "crayon"        "dbplyr"       
##  [5] "dplyr"         "dtplyr"        "forcats"       "ggplot2"      
##  [9] "googledrive"   "googlesheets4" "haven"         "hms"          
## [13] "httr"          "jsonlite"      "lubridate"     "magrittr"     
## [17] "modelr"        "pillar"        "purrr"         "readr"        
## [21] "readxl"        "reprex"        "rlang"         "rstudioapi"   
## [25] "rvest"         "stringr"       "tibble"        "tidyr"        
## [29] "xml2"          "tidyverse"
```

It also includes a *lot* of dependencies upon installation.

Tidyverse packages (cont.)

Today I'm going to focus on two packages:

1. **dplyr**
2. **tidyr**

These are the workhorse packages for cleaning and wrangling data. They are thus the ones that you will likely make the most use of (alongside **ggplot2**).

Data cleaning and wrangling occupies an inordinate amount of time, no matter where you are in your research career.

Outline

dplyr

Loading dplyr

```
#
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union```
```

Wait, was that an error?

When you load packages in R that have functions sharing the same name as functions you already have, the more recently loaded functions overwrite the previous ones ("masks them").

Sometimes you may get a **warning message** when loading packages—usually because you aren't running the latest version of R:

```
Warning message:
package `gapminder' was built under R version 3.5.3
```

`magrittr` and pipes (`%>%`)

`dplyr` allows us to use `magrittr` operators (`%>%`) to "pipe" data between functions. So instead of nesting functions like this:

```
log(mean(gapminder$pop))
```

```
## [1] 17.2
```

We can pipe them like this:

```
gapminder$pop %>% mean() %>% log()
```

```
## [1] 17.2
```

Read this as, "send `gapminder$pop` to `mean()`, then send the output of that to `log()`."

Using pipes

```r
#
gapminder %>%
  filter(country == "Canada") %>%
  head(4)
```

```
## # A tibble: 4 x 6
##   country continent  year lifeExp      pop gdpPercap
##   <fct>   <fct>     <int>   <dbl>    <int>     <dbl>
## 1 Canada  Americas   1952    68.8 14785584    11367.
## 2 Canada  Americas   1957    70.0 17010154    12490.
## 3 Canada  Americas   1962    71.3 18985849    13462.
## 4 Canada  Americas   1967    72.1 20819767    16077.
```

Using pipes

Easier to read when you have each function on a separate line.

```
take_this_data %>%
  do_first_thing(with = this_value) %>%
  do_next_thing(using = that_value) %>% ...
```

Stuff to the left of the pipe is passed to the *first argument* of the function on the right. Other arguments go on the right in the function.

If you ever find yourself piping a function where data are not the first argument, use `.` in the data argument instead.

```
gapminder %>% lm(pop ~ year, data = .)
```

Pipe assignment

When creating a new object from the output of piped functions, place the assignment operator at the beginning.

```
lm_pop_year <- gapminder %>%
  filter(continent == "Americas") %>%
  lm(pop ~ year, data = .)
```

No matter how long the chain of functions is, assignment is always done at the top.

- this is just a stylistic convention, you *can* do assignment at the end of the chain

The Base R pipe

In R version 4.1.0, a base R pipe was introduced: `|>`

```
gapminder |> head(2)
```

```
## # A tibble: 2 x 6
##   country     continent  year lifeExp     pop gdpPercap
##   <fct>       <fct>     <int>   <dbl>   <int>     <dbl>
## 1 Afghanistan Asia       1952    28.8 8425333      779.
## 2 Afghanistan Asia       1957    30.3 9240934      821.
```

It works just like `%>%` but is simpler and faster.

Key `dplyr` verbs

There are five key `dplyr` verbs that you need to learn.

1. `filter`: Filter (i.e. subset) rows based on their values.
2. `arrange`: Arrange (i.e. reorder) rows based on their values.
3. `select`: Select (i.e. subset) columns by their names:
4. `mutate`: Create new columns.
5. `summarize`: Collapse multiple rows into a single summary value.

`filter()` data frames

We subset *rows* of data using logical conditions with `filter()`.

```
gapminder %>%
  filter(country == "Canada") %>%
  head(4)
```

```
## # A tibble: 4 x 6
##   country continent  year lifeExp      pop gdpPercap
##   <fct>   <fct>     <int>   <dbl>    <int>     <dbl>
## 1 Canada  Americas   1952    68.8 14785584    11367.
## 2 Canada  Americas   1957    70.0 17010154    12490.
## 3 Canada  Americas   1962    71.3 18985849    13462.
## 4 Canada  Americas   1967    72.1 20819767    16077.
```

Multiple conditions example

```r
gapminder %>%
  filter(country == "Canada" & year > 1980) %>%
  head(4)
```

```
## # A tibble: 4 x 6
##   country continent  year lifeExp      pop gdpPercap
##   <fct>   <fct>     <int>   <dbl>    <int>     <dbl>
## 1 Canada  Americas   1982    75.8 25201900    22899.
## 2 Canada  Americas   1987    76.9 26549700    26627.
## 3 Canada  Americas   1992    78.0 28523502    26343.
## 4 Canada  Americas   1997    78.6 30305843    28955.
```

Multiple conditions

And: &

```
gapminder %>%
  filter(country == "Canada" & year > 1980)
```

- give me rows where the country is Canada **and** the year is after 1980

Or: |

```
gapminder %>%
  filter(country == "Canada" | year > 1980)
```

- give me rows where the country is Canada **or** the year is after 1980… or **both**

`%in%` operator

We can use `%in%` like `==` but for matching *any element* in a vector.

```r
former_yugoslavia <-
  c("Bosnia and Herzegovina", "Croatia","Montenegro", "Serbia", "Slovenia")
yugoslavia <- gapminder %>%
  filter(country %in% former_yugoslavia)
tail(yugoslavia, 2)
```

```
## # A tibble: 2 x 6
##   country  continent  year lifeExp     pop gdpPercap
##   <fct>    <fct>     <int>   <dbl>   <int>     <dbl>
## 1 Slovenia Europe     2002    76.7 2011497    20660.
## 2 Slovenia Europe     2007    77.9 2009245    25768.
```

Sorting: arrange()

Along with filtering the data to see certain rows, we might want to sort it:

```
yugoslavia %>%
  arrange(year, desc(pop)) %>%
  head(4)
```

```
## # A tibble: 4 x 6
##   country                continent  year lifeExp     pop gdpPercap
##   <fct>                  <fct>     <int>   <dbl>   <int>     <dbl>
## 1 Serbia                 Europe     1952    58.0 6860147     3581.
## 2 Croatia                Europe     1952    61.2 3882229     3119.
## 3 Bosnia and Herzegovina Europe     1952    53.8 2791000      974.
## 4 Slovenia               Europe     1952    65.6 1489518     4215.
```

The data are sorted by ascending `year` and descending `pop`.

Keeping columns: `select()`

Not only can we subset rows, but we can include specific columns (and put them in the order listed) using **`select()`**.

```
yugoslavia %>%
  select(country, year, pop) %>%
  head(4)
```

```
## # A tibble: 4 x 3
##   country                  year     pop
##   <fct>                   <int>   <int>
## 1 Bosnia and Herzegovina  1952 2791000
## 2 Bosnia and Herzegovina  1957 3076000
## 3 Bosnia and Herzegovina  1962 3349000
## 4 Bosnia and Herzegovina  1967 3585000
```

Dropping columns: `select()`

We can instead drop only specific columns with `select()` using `-` signs:

```
yugoslavia %>%
  select(-continent, -pop, -lifeExp) %>%
  head(4)
```

```
## # A tibble: 4 x 3
##   country                 year gdpPercap
##   <fct>                  <int>     <dbl>
## 1 Bosnia and Herzegovina  1952      974.
## 2 Bosnia and Herzegovina  1957     1354.
## 3 Bosnia and Herzegovina  1962     1710.
## 4 Bosnia and Herzegovina  1967     2172.
```

Helper functions for `select()`

`select()` has a variety of helper functions like `starts_with()`, `ends_with()`, and `matches()`, or can be given a range of contiguous columns `startvar:endvar`.

See `?select` for details.

`select(where())`

An especially useful helper for select is `where()` which can be used for selecting columns based on functions that check column types.

```
gapminder %>%
  select(where(is.numeric)) %>%
  head(4)
```

```
## # A tibble: 4 x 4
##    year lifeExp      pop gdpPercap
##   <int>   <dbl>    <int>     <dbl>
## 1  1952    28.8  8425333      779.
## 2  1957    30.3  9240934      821.
## 3  1962    32.0 10267083      853.
## 4  1967    34.0 11537966      836.
```

Renaming columns with `select()`

We can rename columns using `select()`, but that drops everything that isn't mentioned:

```
yugoslavia %>%
  select(Life_Expectancy = lifeExp) %>%
  head(4)
```

```
## # A tibble: 4 x 1
##   Life_Expectancy
##             <dbl>
## 1            53.8
## 2            58.4
## 3            61.9
## 4            64.8
```

Safer: rename columns with `rename()`

`rename()` renames variables using the same syntax as `select()` without dropping unmentioned variables.

```
yugoslavia %>%
  select(country, year, lifeExp) %>%
  rename(Life_Expectancy = lifeExp) %>%
  head(4)
```

```
## # A tibble: 4 x 3
##   country                 year Life_Expectancy
##   <fct>                  <int>           <dbl>
## 1 Bosnia and Herzegovina  1952            53.8
## 2 Bosnia and Herzegovina  1957            58.4
## 3 Bosnia and Herzegovina  1962            61.9
## 4 Bosnia and Herzegovina  1967            64.8
```

`mutate()`

In `dplyr`, you can add new columns to a data frame using **`mutate()`**.

```
yugoslavia %>%
  filter(country == "Serbia") %>%
  select(year, pop, lifeExp) %>%
  mutate(pop_million = pop / 1000000) %>%
  head(2)
```

```
## # A tibble: 2 x 4
##    year     pop lifeExp pop_million
##   <int>   <int>   <dbl>       <dbl>
## 1  1952 6860147    58.0        6.86
## 2  1957 7271135    61.7        7.27
```

- you can create multiple variables in a single `mutate()` call by separating the expressions with commas

`ifelse()`

A common function used in `mutate()` is **`ifelse()`**. It returns a vector of values depending on a logical test.

```
ifelse(test = x==y, yes = first_value , no = second_value)
```

Output from `ifelse()` if `x==y` is...

- TRUE: `first_value`
- FALSE: `second_value`
- NA: NA

For example:

```
example <- c(1, 0, NA, -2)
ifelse(example > 0, "Positive", "Not Positive")
```

```
## [1] "Positive"     "Not Positive" NA             "Not Positive"
```

ifelse() example

```
yugoslavia %>%
  mutate(
    short_country =
      ifelse(country == "Bosnia and Herzegovina","B and H", as.character(country))
  ) %>%
  select(country, short_country, year, pop) %>%
  arrange(year, short_country) %>%
  head(2)
```

```
## # A tibble: 2 x 4
##   country                short_country  year     pop
##   <fct>                  <chr>         <int>   <int>
## 1 Bosnia and Herzegovina B and H        1952 2791000
## 2 Croatia                Croatia        1952 3882229
```

- `country` is a factor, use `as.character()` to convert to character

`case_when()`

`case_when()` performs multiple `ifelse()` operations at the same time. `case_when()` allows you to create a new variable with values based on multiple logical statements. This is useful for making categorical variables or variables from combinations of other variables.

case_when()

```
gapminder %>%
  mutate(
    gdpPercap_ordinal = case_when(
      gdpPercap <  700 ~ "low",
      gdpPercap >= 700 & gdpPercap < 800 ~ "moderate",
      TRUE ~ "high" )
  ) %>%
  slice(6:9) # get rows 6 through 9
```

```
## # A tibble: 4 x 7
##   country     continent  year lifeExp      pop gdpPercap gdpPercap_ordinal
##   <fct>       <fct>     <int>   <dbl>    <int>     <dbl> <chr>
## 1 Afghanistan Asia       1977    38.4 14880372      786. moderate
## 2 Afghanistan Asia       1982    39.9 12881816      978. high
## 3 Afghanistan Asia       1987    40.8 13867957      852. high
## 4 Afghanistan Asia       1992    41.7 16317921      649. low
```

General aggregation: `summarize()`

`summarize()` takes your column(s) of data and computes something using every row:

- count how many rows there are
- calculate the mean
- compute the sum
- obtain a minimum or maximum value

You can use any function in `summarize()` that aggregates *multiple values* into a *single value* (like `sd()`, `mean()`, or `max()`).

`summarize()` example

For the year 1982, let's get the *number of observations*, *total population*, *mean life expectancy*, and *range of life expectancy* for former Yugoslavian countries.

```
yugoslavia %>% filter(year==1982)
```

```
## # A tibble: 5 x 6
##   country                continent  year lifeExp     pop gdpPercap
##   <fct>                  <fct>     <int>   <dbl>   <int>     <dbl>
## 1 Bosnia and Herzegovina Europe     1982    70.7 4172693     4127.
## 2 Croatia                Europe     1982    70.5 4413368    13222.
## 3 Montenegro             Europe     1982    74.1  562548    11223.
## 4 Serbia                 Europe     1982    70.2 9032824    15181.
## 5 Slovenia               Europe     1982    71.1 1861252    17867.
```

`summarize()` example

For the year 1982, let's get the *number of observations*, *total population*, *mean life expectancy*, and *range of life expectancy* for former Yugoslavian countries.

```r
yugoslavia %>% filter(year==1982) %>%
  summarize(
    n_obs = n(),
    total_pop = sum(pop),
    mean_life_exp = mean(lifeExp),
    range_life_exp = max(lifeExp) - min(lifeExp)
  )
```

```
## # A tibble: 1 x 4
##   n_obs total_pop mean_life_exp range_life_exp
##   <int>     <int>         <dbl>          <dbl>
## 1     5  20042685          71.3           3.94
```

Avoiding repetition: `summarize(across())`

Maybe you need to calculate the mean and standard deviation of a bunch of columns. With **`across()`**, put the variables to compute over first (using `c()` or `select()` syntax) and put the functions to use in a `list()` after.

```
yugoslavia %>% filter(year==1982) %>%
  summarize(
    across(c(lifeExp, pop), list(avg = ~mean(.), sd = ~sd(.)))
  )
```

```
## # A tibble: 1 x 4
##   lifeExp_avg lifeExp_sd pop_avg   pop_sd
##         <dbl>      <dbl>   <dbl>    <dbl>
## 1        71.3       1.60 4008537 3237282.
```

Note it automatically names the summarized variables based on the names given in `list()`.

Too many (and)

It can get hard to read code with lots of **nested** functions–functions inside others.

Break things up when it gets confusing!

```
yugoslavia %>% filter(year==1982) %>%
  summarize(
    across(
      c(lifeExp, pop),
      list(avg = ~mean(.), sd  = ~sd(.))
    )
  )
```

RStudio also helps you by tracking parentheses: Put your cursor after a) and see!

Avoiding repetition

There are additional ways to use `across()` for repetitive operations:

- `across(everything())` will summarize / mutate *all* variables sent to it in the same way
- `across(where())` will summarize / mutate all variables that satisfy some logical condition

You can use all of these to avoid typing out the same code repeatedly!

`group_by()`

The special function **`group_by()`** changes how subsequent functions operate on the data, most importantly `summarize()`.

Functions after `group_by()` are computed *within each group* as defined by unique values of the variables given, rather than over all rows at once.

Typically the variables you group by will be integers, factors, or characters, and *not continuous real values*.

`group_by()` example

```
yugoslavia %>%
  group_by(year) %>%
  summarize(
    num_countries = n_distinct(country),
    total_pop = sum(pop)
  ) %>% head(3)
```

```
## # A tibble: 3 x 3
##     year num_countries total_pop
##    <int>         <int>     <int>
## 1  1952             5  15436728
## 2  1957             5  16314276
## 3  1962             5  17099107
```

Because we did `group_by()` with `year` then used `summarize()`, we get *one row per value of* `year`!

Window functions

Grouping can also be used with `mutate()` or `filter()` to give rank orders within a group, lagged values, and cumulative sums.

```
yugoslavia %>% select(country, year, pop) %>%
  filter(year >= 2002) %>%
  group_by(country) %>%
  mutate(lag_pop = lag(pop, order_by = year),
         pop_chg = pop - lag_pop) %>%
  head(4)
```

```
## # A tibble: 4 x 5
## # Groups:   country [2]
##   country                year     pop lag_pop pop_chg
##   <fct>                 <int>   <int>   <int>   <int>
## 1 Bosnia and Herzegovina  2002 4165416      NA      NA
## 2 Bosnia and Herzegovina  2007 4552198 4165416  386782
## 3 Croatia                 2002 4481020      NA      NA
## 4 Croatia                 2007 4493312 4481020   12292
```

Ungrouping

Multiple groups with `summarize()` will retain all but the last group.

```r
gapminder %>%
  group_by(continent, year) %>%
  summarize(mean_gdp = mean(gdpPercap)) %>%
  head(2)
```

```
## # A tibble: 2 x 3
## # Groups:   continent [1]
##   continent  year mean_gdp
##   <fct>     <int>    <dbl>
## 1 Africa     1952    1253.
## 2 Africa     1957    1385.
```

Ungrouping

Use `ungroup()` if you want to remove groups!

```
gapminder %>%
  group_by(continent, year) %>%
  summarize(mean_gdp = mean(gdpPercap)) %>%
  ungroup() %>%
  head(2)
```

```
## # A tibble: 2 x 3
##   continent  year mean_gdp
##   <fct>     <int>    <dbl>
## 1 Africa     1952    1253.
## 2 Africa     1957    1385.
```

Base R vs. `dplyr`

Two ways of calculating the same thing: which do you like better?

Classic R:

```
mean(swiss[swiss$Education > mean(swiss$Education), "Education"])
```

`dplyr`:

```
library(dplyr)
swiss %>%
  filter(Education > mean(Education)) %>%
  summarize(mean = mean(Education))
```

Outline

When do we need to join data?

Want to make columns using criteria too complicated for `ifelse()` or `case_when()`.

- we can work with small sets of variables then combine them back together

Combine data stored in separate data sets.

- often large surveys are broken into different data sets for each level (e.g. household, individual, neighborhood)

Joining in concept

We need to think about the following when we want to merge data frames `A` and `B`:

- which *rows* are we keeping from each data frame?
- which *columns* are we keeping from each data frame?
- which variables determine whether rows *match*?

Join types: rows and columns kept

There are many types of joins...

- `A %>% left_join(B)`: keep all rows from `A`, matched with `B` wherever possible (`NA` when not), keep columns from both `A` and `B`
- `A %>% right_join(B)`: keep all rows from `B`, matched with `A` wherever possible (`NA` when not), keep columns from both `A` and `B`
- `A %>% inner_join(B)`: keep only rows from `A` and `B` that match, keep columns from both `A` and `B`
- `A %>% full_join(B)`: keep all rows from both `A` and `B`, matched wherever possible (`NA` when not), keep columns from both `A` and `B`

Join types: rows and columns kept

There are many types of joins…

- `A %>% semi_join(B)`: keep rows from `A` that match rows in `B`, keep columns from only `A`
- `A %>% anti_join(B)`: keep rows from `A` that *don't* match a row in `B`, keep columns from only `A`

Usually `left_join()` does the job.

Matching criteria

We say rows should *match* because they have some columns containing the same value. We list these in a `by =` argument to the join.

Matching Behavior:

- no `by`: match using all variables in `A` and `B` that have identical names
- `by = c("var1", "var2", "var3")`: match on identical values of `var1`, `var2`, and `var3` in both `A` and `B`
- `by = c("Avar1" = "Bvar1", "Avar2" = "Bvar2")`: match identical values of `Avar1` variable in `A` to `Bvar1` variable in `B`, and `Avar2` variable in `A` to `Bvar2` variable in `B`

Note: If there are multiple matches, you'll get *one row for each possible combination* (except with `semi_join()` and `anti_join()`).

Joins example

We'll use some data sets that come bundled with the **nycflights13** package (inspect these data frames in your own computers).

```
library(nycflights13)
flights
planes
```

Joins example

Let's perform a left join on the flights and planes datasets.

- I'm going subset columns after the join, but only to keep text on the slide.

```
left_join(flights, planes) %>%
  select(year, month, day, dep_time, arr_time, carrier, flight, tailnum) %>%
  head(4)
```

```
## # A tibble: 4 x 8
##    year month   day dep_time arr_time carrier flight tailnum
##   <int> <int> <int>    <int>    <int> <chr>    <int> <chr>
## 1  2013     1     1      517      830 UA        1545 N14228
## 2  2013     1     1      533      850 UA        1714 N24211
## 3  2013     1     1      542      923 AA        1141 N619AA
## 4  2013     1     1      544     1004 B6         725 N804JB
```

Joins example

Note that dplyr made a reasonable guess about which columns to join on (i.e. columns that share the same name). It also told us its choices:

```
## Joining, by = c("year", "tailnum")
```

However, there's an obvious problem here: the variable "year" does not have a consistent meaning across our joining datasets!

- in one it refers to the *year of flight*, in the other it refers to *year of construction*

Joins example

We need to be more explicit in your join call by using the `by =` argument.

```
left_join(
    flights,
    planes %>% rename(year_built = year), # not necessary w/ below line, but helpful
    by = "tailnum" # be specific about the joining column
  ) %>%
  select(year, month, day, dep_time, arr_time, carrier, flight, tailnum, year_built)
  head(3)
```

```
## # A tibble: 3 x 9
##    year month   day dep_time arr_time carrier flight tailnum year_built
##   <int> <int> <int>    <int>    <int> <chr>    <int> <chr>        <int>
## 1  2013     1     1      517      830 UA        1545 N14228        1999
## 2  2013     1     1      533      850 UA        1714 N24211        1998
## 3  2013     1     1      542      923 AA        1141 N619AA        1990
```

Outline

tidyr

Initial spot checks

First things to check after loading new data:

- did the last rows/columns from the original file make it in?
- are the column names in good shape?
- are there “decorative” blank rows or columns to remove?
- how are missing values represented: `NA`, `" "` (blank), `.` (period), `999`?
- are there character data (e.g. ZIP codes with leading zeroes) being incorrectly represented as numeric or vice versa?

Some data to work with

Let's generate some data.

```
stocks <-
  data.frame(
    time = as.Date('2009-01-01') + 0:1,
    X = rnorm(2, 0, 1),
    Y = rnorm(2, 0, 2),
    Z = rnorm(2, 0, 4)
  )
stocks
```

```
##         time       X      Y     Z
## 1 2009-01-01 -1.2071  2.169 1.716
## 2 2009-01-02  0.2774 -4.691 2.024
```

Tidy version

```r
#
stocks %>%
  pivot_longer(-time, names_to = "stock", values_to = "price")
```

```
## # A tibble: 6 x 3
##   time       stock  price
##   <date>     <chr>  <dbl>
## 1 2009-01-01 X     -1.21
## 2 2009-01-01 Y      2.17
## 3 2009-01-01 Z      1.72
## 4 2009-01-02 X      0.277
## 5 2009-01-02 Y     -4.69
## 6 2009-01-02 Z      2.02
```

Tidy data

Tidy data (aka “long data”) are such that:

1. The values for a single observation are in their own row.
2. The values for a single variable are in their own column.
3. There is only one value per cell.

Tidy data

Why do we want tidy data?

- easier to understand many rows than many columns
- required for plotting in `ggplot2`
- required for many types of statistical procedures (e.g. hierarchical or mixed effects models)
- fewer issues with missing values and “imbalanced” repeated measures data

Key `tidyr` verbs

1. `pivot_longer`: Pivot wide data into long format (i.e. “melt”). (updated version of `tidyr::gather`)
2. `pivot_wider`: Pivot long data into wide format (i.e. “cast”). (updated version of `tidyr::spread`)
3. `separate`: Separate (i.e. split) one column into multiple columns.
4. `unite`: Unite (i.e. combine) multiple columns into one.

`pivot_longer()`

`pivot_longer()` takes a set of columns and pivots them down to make two new columns (which you can name yourself):

- a `name` column that stores the original column names
- a `value` with the values in those original columns

```
tidy_stocks <-
  stocks %>%
  pivot_longer(-time, names_to = "stock", values_to = "price")
```

pivot_wider()

`pivot_wider()` inverts `pivot_longer()` by taking two columns and pivoting them up into multiple columns.

```
tidy_stocks %>%
  pivot_wider(names_from = "stock", values_from = "price")
```

```
## # A tibble: 2 x 4
##   time             X     Y     Z
##   <date>       <dbl> <dbl> <dbl>
## 1 2009-01-01 -1.21    2.17  1.72
## 2 2009-01-02  0.277 -4.69   2.02
```

pivot_wider()

```r
tidy_stocks %>%
  pivot_wider(names_from = "time", values_from = "price")
```

```
## # A tibble: 3 x 3
##   stock `2009-01-01` `2009-01-02`
##   <chr>        <dbl>        <dbl>
## 1 X            -1.21        0.277
## 2 Y             2.17       -4.69
## 3 Z             1.72        2.02
```

Note that the second example—which has combined different pivoting arguments—has effectively transposed the data.

`separate()`

`separate()` pulls apart one column into multiple columns (common after `pivot_longer()` where values are embedded in column names).

```
economists <-
  data.frame(name = c("Adam.Smith", "Paul.Samuelson", "Milton.Friedman"))
economists %>%
  separate(name, c("first_name", "last_name")) %>%
  head(1)
```

```
##   first_name last_name
## 1       Adam     Smith
```

This command is pretty smart. But to avoid ambiguity, you can also specify the separation character with `separate(..., sep=".")`.

separate()

A related function is `separate_rows`, for splitting up cells that contain multiple fields or observations (a frustratingly common occurrence with survey data).

```
jobs <-
  data.frame(
    name = c("Jack", "Jill"),
    occupation = c("Homemaker", "Philosopher, Philanthropist, Troublemaker")
  )
jobs
```

```
##   name                                occupation
## 1 Jack                                 Homemaker
## 2 Jill Philosopher, Philanthropist, Troublemaker
```

separate()

```r
# split out Jill's various occupations into different rows
jobs %>% separate_rows(occupation)
```

```
## # A tibble: 4 x 2
##   name  occupation
##   <chr> <chr>
## 1 Jack  Homemaker
## 2 Jill  Philosopher
## 3 Jill  Philanthropist
## 4 Jill  Troublemaker
```

unite()

Let’s generate some data.

```r
gdp <-
  data.frame(
    yr = rep(2016, times = 4),
    mnth = rep(1, times = 4),
    dy = 1:4,
    gdp = rnorm(4, mean = 100, sd = 2)
  )
gdp
```

```
##     yr mnth dy   gdp
## 1 2016    1  1 98.85
## 2 2016    1  2 98.91
## 3 2016    1  3 98.87
## 4 2016    1  4 98.22
```

`unite()`

`unite()` will combine multiple columns into one.

```
# combine "yr", "mnth", and "dy" into one "date" column
gdp %>% unite(date, c("yr", "mnth", "dy"), sep = "-")
```

```
##       date   gdp
## 1 2016-1-1 98.85
## 2 2016-1-2 98.91
## 3 2016-1-3 98.87
## 4 2016-1-4 98.22
```

unite()

Note that `unite` will automatically create a character variable. You can see this better if we convert it to a tibble.

```
gdp_u <- gdp %>%
  unite(date, c("yr", "mnth", "dy"), sep = "-") %>%
  as_tibble()
gdp_u
```

```
## # A tibble: 4 x 2
##   date        gdp
##   <chr>     <dbl>
## 1 2016-1-1   98.9
## 2 2016-1-2   98.9
## 3 2016-1-3   98.9
## 4 2016-1-4   98.2
```

unite()

If you want to convert it to something else (e.g. date or numeric) then you will need to modify it using `mutate`.

```
library(lubridate)
gdp_u %>% mutate(date = ymd(date))
```

```
## # A tibble: 4 x 2
##   date          gdp
##   <date>      <dbl>
## 1 2016-01-01   98.9
## 2 2016-01-02   98.9
## 3 2016-01-03   98.9
## 4 2016-01-04   98.2
```

Other `tidyr` goodies

Use `crossing` to get the full combination of a group of variables (Base R alternative: `expand.grid`).

```
crossing(side=c("left", "right"), height=c("top", "bottom"))
```

```
## # A tibble: 4 x 2
##   side  height
##   <chr> <chr>
## 1 left  bottom
## 2 left  top
## 3 right bottom
## 4 right top
```

See `?expand` and `?complete` for more specialized functions that allow you to fill in (implicit) missing data or variable combinations in existing data frames.

Outline

Example: Google mobility indexes

This is an example from another working paper of mine. As the title suggests, we want to plot some Google mobility indexes.

We'll need some additional packages.

```
# packages
library(tsibble)
library(lubridate)
```

Example: Google mobility indexes

The file containing Google mobility data is quite large, We'll use the function `read_csv()` from the `readr` package (part of the tidyverse) as it is very fast.

Next, I will immediately filter the data for Canada.

```
# Google mobility data
mobility_data <-
  read_csv("data/Global_Mobility_Report.csv",col_names = TRUE) %>%
  filter(country_region == "Canada")
```

Example: Google mobility indexes

Let's get data for some provinces and four of the indexes.

```
# sub-region codes
region_code <- c('CA-BC', 'CA-AB', 'CA-ON', 'CA-QC')
#
data <- mobility_data %>%
  filter(iso_3166_2_code %in% region_code) %>%
  select(
    date,
    province = iso_3166_2_code,
    retail = retail_and_recreation_percent_change_from_baseline,
    transit = transit_stations_percent_change_from_baseline,
    workplace = workplaces_percent_change_from_baseline,
    residential = residential_percent_change_from_baseline
  )
```

Example: Google mobility indexes

Let's take a look.

```
#
head(data, 6)
```

```
## # A tibble: 6 x 6
##   date       province retail transit workplace residential
##   <date>     <chr>     <dbl>   <dbl>     <dbl>       <dbl>
## 1 2020-02-15 CA-AB         7       7        -2          -
1
## 2 2020-02-16 CA-AB        10       3        -3          -
2
## 3 2020-02-17 CA-AB        -7     -40       -67          15
## 4 2020-02-18 CA-AB        -1      -9        -5           2
## 5 2020-02-19 CA-AB         3      -7        -1           1
## 6 2020-02-20 CA-AB         7      -6        -1           0
```

Example: Google mobility indexes

Now let's clean things up a bit.

```
# tidy
data_daily <- data %>%
  pivot_longer(
    c("retail","transit","workplace","residential"),
    names_to = "index",
    values_to = "value"
  ) %>%
  group_by(province, index)
```

Example: Google mobility indexes

Let's take a look.

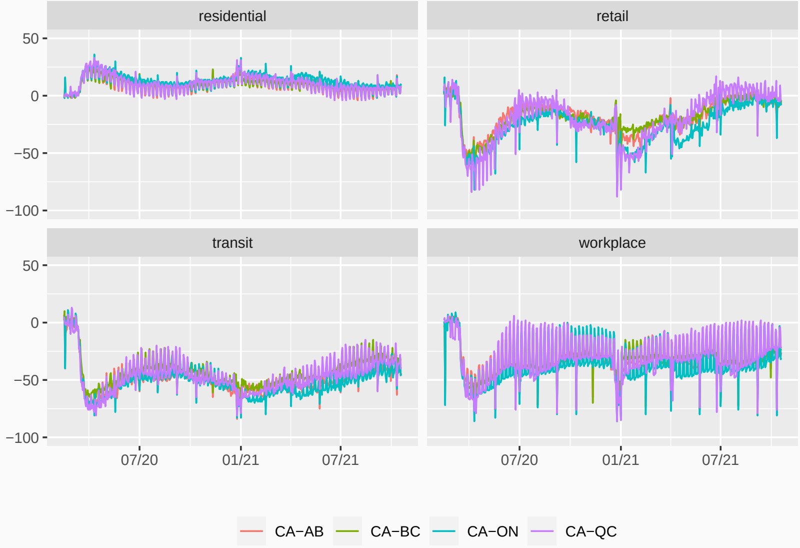
```r
#
head(data_daily, 6)
```

```
## # A tibble: 6 x 4
## # Groups:   province, index [4]
##   date       province index       value
##   <date>     <chr>    <chr>       <dbl>
## 1 2020-02-15 CA-AB    retail          7
## 2 2020-02-15 CA-AB    transit         7
## 3 2020-02-15 CA-AB    workplace      -2
## 4 2020-02-15 CA-AB    residential    -1
## 5 2020-02-16 CA-AB    retail         10
## 6 2020-02-16 CA-AB    transit         3
```

Example: Google mobility indexes

```r
# now we can plot the daily indexes
p1 <-
  ggplot(data_daily, aes(x = date, y = value, color = province)) +
  geom_line(size = .5) +
  scale_y_continuous(name = "", limits = c(-100,50), breaks = c(50,0,-50,-100)) +
  scale_x_date(name = "", date_breaks = "6 month", date_labels = "%m/%y") +
  theme(legend.position = "bottom", legend.title = element_blank()) +
  facet_wrap( ~ index)
```

Example: Google mobility indexes



Example: Google mobility indexes

Let's transform the data from daily to weekly.

```
#
data_weekly <- data_daily %>%
  mutate(y_week = yearweek(date)) %>%
  group_by(province, index, y_week) %>%
  summarize(
    date = first(date),
    y_week = first(y_week),
    province = first(province),
    index = first(index),
    value = mean(value)
  ) %>%
  arrange(province, index)
```

Example: Google mobility indexes

Let's take a look.

```
#
head(data_weekly, 6)
```

```
## # A tibble: 6 x 5
## # Groups:   province, index [1]
##   province index       y_week date       value
##   <chr>    <chr>       <week> <date>     <dbl>
## 1 CA-AB    residential 2020 W07 2020-02-15 -1.5
## 2 CA-AB    residential 2020 W08 2020-02-17  2.43
## 3 CA-AB    residential 2020 W09 2020-02-24 -0.429
## 4 CA-AB    residential 2020 W10 2020-03-02  0.429
## 5 CA-AB    residential 2020 W11 2020-03-09  2.86
## 6 CA-AB    residential 2020 W12 2020-03-16 14.1
```

Example: Google mobility indexes