11 - Tidy Data

Data and Information Engineering

SYS 2202 | Fall 2019

11-tidy.pdf

Contents

Required Packages and Data

```
library(tidyverse)
```

Some images and quotes taken from our textbook R4DS.

1 Tidy Data

"Happy families are all alike; every unhappy family is unhappy in its own way." - Leo Tolstoy

"Tidy datasets are all alike, but every messy dataset is messy in its own way." - Hadley Wickham

The textbook has some examples of tidy and untidy data

```
library(tidyverse)
data(package="tidyr")
# table1, table2, table3, table4a, table4b
```

1.1 Get the Rate (cases/population)

For each table, calculate the `rate = cases/population`.

1.1.1 Table 1

```
table1
#> # A tibble: 6 x 4
#>   country      year  cases population
#>   <chr>       <int>  <int>      <int>
#> 1 Afghanistan  1999    745   19987071
#> 2 Afghanistan  2000   2666   20595360
#> 3 Brazil       1999  37737  172006362
#> 4 Brazil       2000  80488  174504898
#> 5 China        1999 212258 1272915272
#> 6 China        2000 213766 1280428583
```

Your Turn #1

What `dplyr` function can be used to create the `rate` column of `table1`?

1.1.2 Table 2

```
table2
#> # A tibble: 12 x 4
#>   country      year type            count
#>   <chr>       <int> <chr>           <int>
#> 1 Afghanistan  1999 cases             745
#> 2 Afghanistan  1999 population   19987071
#> 3 Afghanistan  2000 cases            2666
#> 4 Afghanistan  2000 population   20595360
#> 5 Brazil       1999 cases           37737
#> 6 Brazil       1999 population  172006362
#> # ... with 6 more rows
```

Your Turn #2

What needs to be done to calculate the rate (by country and year) of `table2`?
Hint: what constitutes an *observation*, and what are the *variables*? Another way to consider is by identifying the *primary key(s)* of the table.

1.1.3 Table 3

```
table3
#> # A tibble: 6 x 3
#>   country      year rate
#> * <chr>       <int> <chr>
#> 1 Afghanistan  1999 745/19987071
#> 2 Afghanistan  2000 2666/20595360
#> 3 Brazil       1999 37737/172006362
#> 4 Brazil       2000 80488/174504898
#> 5 China        1999 212258/1272915272
#> 6 China        2000 213766/1280428583
```

Your Turn #3

What needs to be done to actually calculate the rate in table 3?

1.1.4 Tables 4a and 4b

```
#-- The data are in two different tables
table4a    # number of cases
#> # A tibble: 3 x 3
#>   country     `1999` `2000`
#> * <chr>        <int>  <int>
#> 1 Afghanistan    745   2666
#> 2 Brazil       37737  80488
#> 3 China       212258 213766
table4b    # population
#> # A tibble: 3 x 3
#>   country         `1999`     `2000`
#> * <chr>            <int>      <int>
#> 1 Afghanistan   19987071   20595360
#> 2 Brazil       172006362  174504898
#> 3 China       1272915272 1280428583
```

Your Turn #4

What needs to be done to calculate the rate from tables 4a and 4b?
Hint: The info is split between two tables. Would it help if each table was in a different form?

1.2 Why Tidy Data?

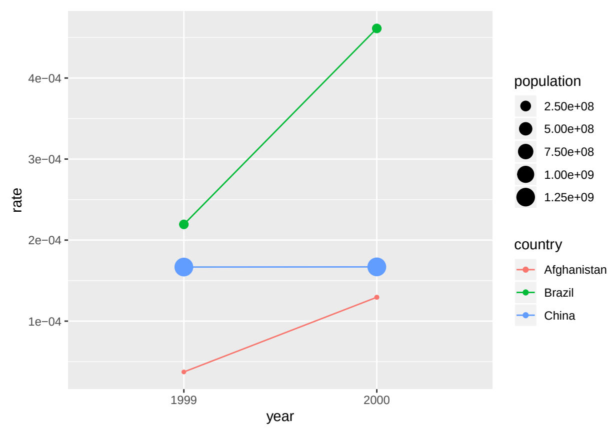
- Tidy data (in form of a data frame) is usually the best form for analysis
  - some exceptions are for modeling (e.g., matrix manipulations and algorithms)
- For presentation of data (e.g., in tables), non-tidy form can often do better
- the functions in `tidyr` usually allow us to covert from non-tidy to tidy for analysis and also from tidy to non-tidy for presentation

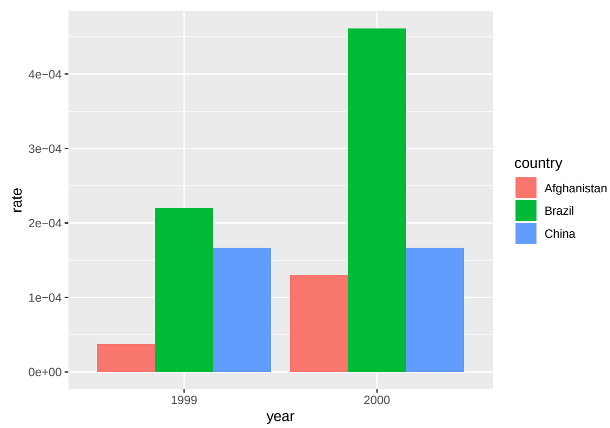
2 Main `tidyr` functions

function	description
`spread()`	Spreads a pair of key:value columns into a set of tidy columns
`gather()`	Gather takes multiple columns and collapses into key-value pairs, duplicating all other columns as needed. You use gather() when you notice that you have columns that are not variables
`separate()`	turns a single character column into multiple columns
`unite()`	paste together multiple columns into one (reverse of `separate()`)

Tidy data is often the form we want for further analysis. For example, here are some basic plots that would be difficult to make in the untidy versions.

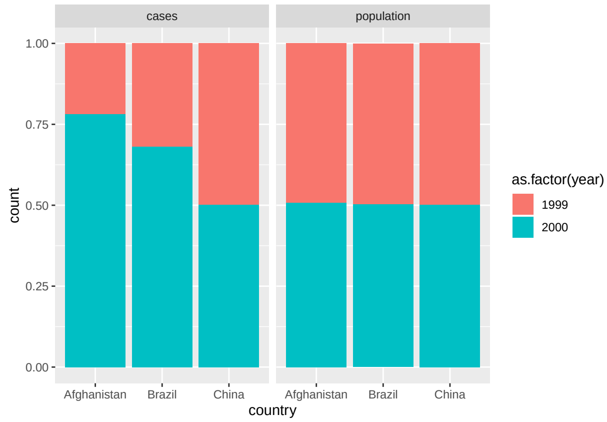
```
tidy_table = table1 %>% mutate(rate=cases/population)

#- line plot
ggplot(tidy_table, aes(x=as.factor(year), y=rate, color=country, group=country)) +
  geom_line() + geom_point(aes(size=population)) + xlab("year")

#- bar plot
ggplot(tidy_table, aes(x=as.factor(year), y=rate, fill=country)) +
  geom_bar(stat="identity", position="dodge") + xlab("year")
```





One exception is if we want to facet (or group) by `type` column(s). Then `table2` is better.

```
ggplot(table2, aes(x=country, y=count, fill=as.factor(year))) +
  geom_bar(stat="identity", position="fill") + facet_wrap(~type)
```



The tidyr package provides functionality to convert to and from tidy data, which can greatly speed up analysis and help structure your thinking.

2.1 gather() into long form

The gather() function collects a set of column names and places them into a single "key" column. It also collects the field of cells associated with those columns and places them into a single value column.

In the example from 12.3.1 R4DS, table4a (cases) and table4b (population) are gathered into two columns: year and value.

```
table4a
#> # A tibble: 3 x 3
#>   country     `1999` `2000`
#> * <chr>        <int>  <int>
#> 1 Afghanistan    745   2666
#> 2 Brazil       37737  80488
#> 3 China       212258 213766
(tidy4a = gather(table4a, key="year", value="cases", 2:3))
#> # A tibble: 6 x 3
#>   country     year   cases
#>   <chr>       <chr>  <int>
#> 1 Afghanistan 1999     745
#> 2 Brazil      1999   37737
#> 3 China       1999  212258
#> 4 Afghanistan 2000    2666
#> 5 Brazil      2000   80488
#> 6 China       2000  213766
```

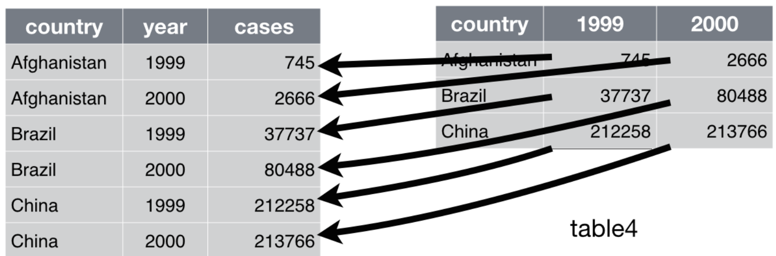


Figure 1: Gathering `table4` into a tidy form.

The function is:

```
gather(
    data = <data frame>,
    key = <name of new key column>,
    value = <name of new value column>,
    ... = <specification of columns to gather>,
    <optional.args>)
```

where the specification of columns could be by name, index, or any method allowed by the `?dplyr::select()` function.

Your Turn #5

1. For tidying table4a, how were the columns to gather specified (in the example above)?
2. What would be an alternative way to specify them?
3. Tidy up table4b.

```
table4b
#> # A tibble: 3 x 3
#>   country         `1999`     `2000`
#> * <chr>            <int>      <int>
#> 1 Afghanistan   19987071   20595360
#> 2 Brazil       172006362  174504898
#> 3 China       1272915272 1280428583
```

4. Calculate the disease rate.

2.2 `spread()` into wide form

The `spread()` function is the opposite of `gather()` and converts two columns (one key, one value) into a set of columns (one new column for every unique key value).

The `table2` can be *spread* into a tidy format

```
table2
#> # A tibble: 12 x 4
#>   country      year type           count
#>   <chr>       <int> <chr>          <int>
#> 1 Afghanistan  1999 cases            745
```

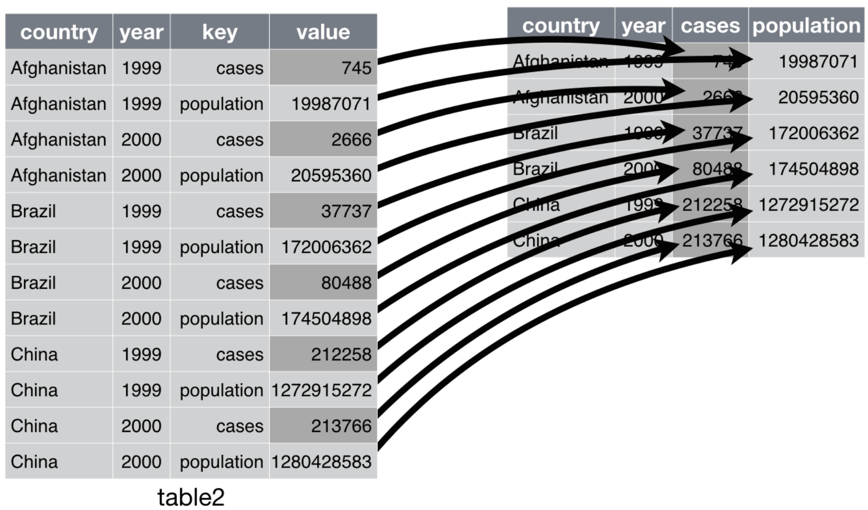


Figure 2: Spreading `table2` makes it tidy.

```
#> 2 Afghanistan  1999 population  19987071
#> 3 Afghanistan  2000 cases           2666
#> 4 Afghanistan  2000 population  20595360
#> 5 Brazil       1999 cases          37737
#> 6 Brazil       1999 population 172006362
#> # ... with 6 more rows
unique(table2$type)
#> [1] "cases"      "population"
spread(table2, key=type, value=count)
#> # A tibble: 6 x 4
#>   country      year  cases population
#>   <chr>       <int>  <int>      <int>
#> 1 Afghanistan  1999    745   19987071
#> 2 Afghanistan  2000   2666   20595360
#> 3 Brazil       1999  37737  172006362
#> 4 Brazil       2000  80488  174504898
#> 5 China        1999 212258 1272915272
#> 6 China        2000 213766 1280428583
```

Notice that 2 extra columns were added (`cases` and `population`) according the unique values in `type`.

The function is:

```
spread(
   data = <data frame>,
   key = <unquoted name of key column>,
   value = <unquoted name of value column>,
   fill = <the value to replace NA's>,
   convert = <logical. Convert (parse) the new columns.>
   <optional.args>)
```

2.3 separate()

The `separate()` function pulls apart one column into multiple columns, by splitting wherever the separator (`sep=`) character appears.

In `table3`, the *equation* for the rate is given, but not the calculated value. One approach is to use the `separate()` function from `tidyr` to separate this one column into two which gives us `table1`.

```
table3
#> # A tibble: 6 x 3
#>   country      year rate
#> * <chr>       <int> <chr>
#> 1 Afghanistan  1999 745/19987071
#> 2 Afghanistan  2000 2666/20595360
#> 3 Brazil       1999 37737/172006362
#> 4 Brazil       2000 80488/174504898
#> 5 China        1999 212258/1272915272
#> 6 China        2000 213766/1280428583
separate(table3, rate, into=c("cases", "population"), sep="/", convert=TRUE) %>%
  mutate(rate=cases/population)
#> # A tibble: 6 x 5
#>   country      year  cases population      rate
#>   <chr>       <int>  <int>      <int>     <dbl>
#> 1 Afghanistan  1999    745   19987071 0.0000373
#> 2 Afghanistan  2000   2666   20595360 0.000129
#> 3 Brazil       1999  37737  172006362 0.000219
#> 4 Brazil       2000  80488  174504898 0.000461
#> 5 China        1999 212258 1272915272 0.000167
#> 6 China        2000 213766 1280428583 0.000167
```

Notice that we used the optional arguments `sep="/"` and `convert=TRUE`.

```
separate(
   data = <data frame>,
   col = <unquoted name column to separate>,
   into = <names of new columns (character vector)>,
   sep = <the separator>,
   remove = <logical. remove original column?>
   convert = <logical. Convert (parse) the new columns.>
   <optional.args>)
```

The `separate()` functions is also useful for extracting date and time elements.

Consider the following data that has date and event information.

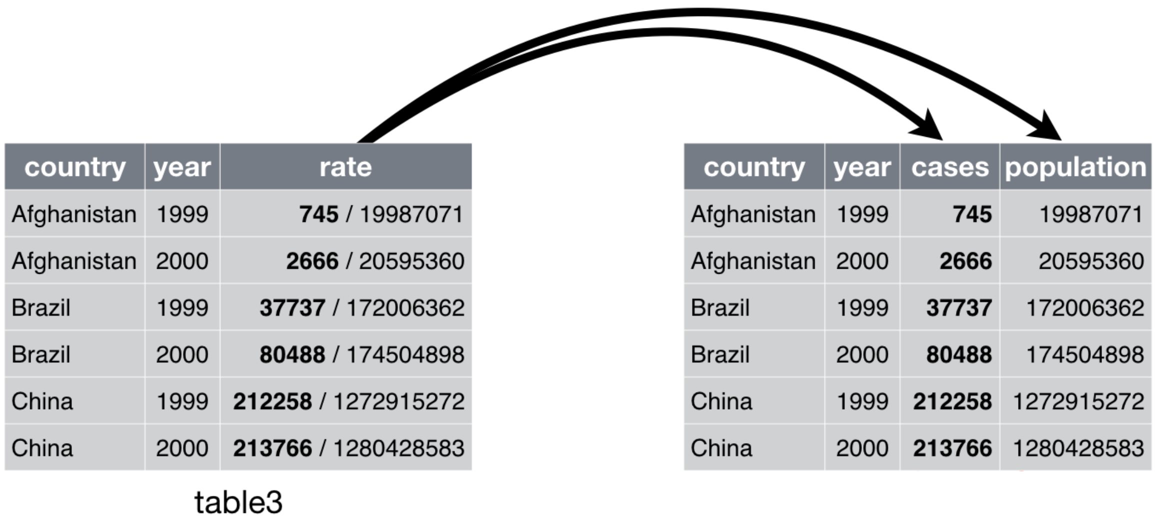


Figure 3: Separating `table3` makes it tidy.

```
url = "https://raw.githubusercontent.com/mdporter/SYS2202/master/data/date-event.csv"
(df = read_csv(url))
#> # A tibble: 100 x 2
#>   date       event
#>   <date>     <chr>
#> 1 2016-01-16 D
#> 2 2016-03-29 D
#> 3 2016-01-17 B
#> 4 2016-05-16 A
#> 5 2016-04-13 C
#> 6 2016-03-29 B
#> # ... with 94 more rows
```

We want to know the distribution of event type by *day of the month*. One way to get this information is with the `separate()` function. The `separate()` function will split up a character column, according to some pattern, into multiple new columns. It essentially does a `str_split` and then adds the new columns into the data frame.

Here is the result with default settings

```
separate(df, col=date, into=c("year", "month", "day"), sep="-")
#> # A tibble: 100 x 4
#>   year  month day   event
#>   <chr> <chr> <chr> <chr>
#> 1 2016  01    16    D
#> 2 2016  03    29    D
#> 3 2016  01    17    B
#> 4 2016  05    16    A
#> 5 2016  04    13    C
#> 6 2016  03    29    B
#> # ... with 94 more rows
```

Notice a few things:

- The original `date` column was removed. We can keep it in with the argument `remove=FALSE`
- The new columns are still *character* vectors. If we want them to be numeric, we can set `convert=TRUE`, which attempt to convert the columns to the appropriate type.

This produces the following:

```
separate(df, col=date, into=c("year", "month", "day"), sep="-",
         remove=FALSE, convert=TRUE)
#> # A tibble: 100 x 5
#>   date        year month   day event
#>   <date>     <int> <int> <int> <chr>
#> 1 2016-01-16  2016     1    16 D
#> 2 2016-03-29  2016     3    29 D
#> 3 2016-01-17  2016     1    17 B
#> 4 2016-05-16  2016     5    16 A
#> 5 2016-04-13  2016     4    13 C
#> 6 2016-03-29  2016     3    29 B
#> # ... with 94 more rows
```

Your Turn #6

1. Find the counts per `day`
2. Convert the data to make the following table

day	A	B	C	D
1	2	0	0	2
2	1	2	0	1
4	0	0	1	1
5	0	0	0	2
6	0	2	1	2
7	2	3	2	0
8	0	0	1	1
9	0	0	1	0
...	...	...	...	...

2.4 `unite()`

The `unite()` function is the opposite of `separate()` and will recombine multiple columns.

```
df %>%
separate(col=date, into=c("year", "month", "day"), sep="-",
         remove=FALSE, convert=TRUE) %>%
  unite(col="USdate", month, day, year, sep="/")
#> # A tibble: 100 x 3
#>   date       USdate    event
#>   <date>     <chr>     <chr>
#> 1 2016-01-16 1/16/2016 D
#> 2 2016-03-29 3/29/2016 D
#> 3 2016-01-17 1/17/2016 B
#> 4 2016-05-16 5/16/2016 A
#> 5 2016-04-13 4/13/2016 C
#> 6 2016-03-29 3/29/2016 B
#> # ... with 94 more rows
```

3 Missing Data

3.1 Missing Values

Changing the representation of a dataset brings up an important subtlety of missing values. Surprisingly, a value can be missing in one of two possible ways:

- *Explicitly*, i.e., flagged with `NA`.
- *Implicitly*, i.e., simply not present in the data.

In the previous example (`date-event`), there is some implicit missing data. What is missing, and what should be the value of the missing data?

```
df %>%
separate(col=date, into=c("year", "month", "day"), sep="-",
         remove=FALSE, convert=TRUE) %>%
  count(day, event) %>% arrange(day, event)
#> # A tibble: 66 x 3
#>     day event     n
#>   <int> <chr> <int>
#> 1     1 A         2
#> 2     1 D         2
#> 3     2 A         1
#> 4     2 B         2
#> 5     2 D         1
#> 6     4 C         1
#> # ... with 60 more rows
```

Now to fill in missing days with `complete()`

```
df %>%
separate(col=date, into=c("year", "month", "day"), sep="-",
         remove=FALSE, convert=TRUE) %>%
  count(day, event) %>%
  complete(day=1:31, event=c('A','B', 'C', 'D'), fill=list(n=0L))
#> # A tibble: 124 x 3
#>     day event     n
#>   <int> <chr> <int>
#> 1     1 A         2
#> 2     1 B         0
#> 3     1 C         0
#> 4     1 D         2
#> 5     2 A         1
#> 6     2 B         2
#> # ... with 118 more rows
```

3.1.1 Functions to know

- `complete()`
- `fill()`

4 Your Turn

4.1 Problem 1: Tornado

Your Turn #7 : Tidy Tornadoes

The US Storm Prediction Center make severe weather data available from the website http://www.spc.noaa.gov/wcm/#data. This data is used by insurance companies to help with their claims evaluation and forecasting. A description of the data can be found http://www.spc.noaa.gov/wcm/data/SPC_severe_database_description.pdf.
Use the tornado event data (https://raw.githubusercontent.com/mdporter/ST597/master/data/tornado.csv), to calculate the number of tornadoes by *year* and *Fujita score* (`f`) and then use `spread()` to convert the results to a table. The final result should look like this

yr	EF-0	EF-1	EF-2	EF-3	EF-4	EF-5
2007	681	306	97	27	4	1
2008	997	515	158	56	11	1
2009	709	355	94	21	3	0
2010	776	351	129	42	17	0
2011	821	638	212	72	25	9
2012	577	242	100	32	5	0
2013	508	314	86	22	8	1
2014	478	325	76	20	7	0
2015	704	415	69	19	5	0

a. Import the tornado data from https://raw.githubusercontent.com/mdporter/SYS2202/master/data/tornado.csv.
b. Create a data frame with columns year (`yr`), Fujita score (`f`), and count (`n`).
c. Use `spread()` to convert to the required (untidy) table. Note: Some years have 0 EF-5 tornadoes.

4.2 Problem 2: Time of Day

Your Turn #8 : Time-of-Day

The goal of this task is to plot the estimated density of the time when tornadoes occur. The `time` column in the `tornado` data gives the time-of-day (24 hour clock, central time zone) when the tornado occurred. Ignoring the time zone issue, create a density plot of the fractional hour when tornadoes occur.

a. Use the `separate()` function to create three new columns (*hour*, *min*, *sec*) from the `time` column.
b. Add another column, named `time2`, that gives the fractional number of hours that a tornado occurred.
c. Generate a density plot of `time2`. Are there any differences by severity?

4.3 Problem 3: Pew Survey

Your Turn #9 : Pew Survey

Results from a pew survey were presented in a non-tidy (table) format where the column headers are *values* instead of *variable names*. That is, the data are in *wide* format, and we desire the *long* format. The data can be found https://raw.githubusercontent.com/tidyverse/tidyr/master/data-raw/relig_income.csv.

a. Load the data into R. The url to the raw data is https://raw.githubusercontent.com/tidyverse/tidyr/master/data-raw/relig_income.csv
b. What are the three variables in the data?
c. Use `gather()` to make the data *tidy* (i.e., long format, with one column for each variable).
d. Make a graphic from the long data comparing the distribution of income between `Catholic` and `Evangelical Prot`.

5 Other functions in `tidyr` package

function	description
`replace_na()`	Replace NA's with specific values
`fill()`	Fills missing values in using the previous entry. This is useful in the common output format where values are not repeated, they're recorded each time they change.
`extract()`	check out `separate()`, but allows different patterns
`expand()`	convert *implicit* missing values (i.e., missing rows) to *explicit* missing values (include rows with `NA`s)
`complete()`	good for tables (filling in missing with 0 counts)