Data Cleaning

Jim Harner

5/30/2020

The `tidyr` package is part of the `tidyverse`. It provides a standardized way of storing data to ensure workflow operations.

```
library(dplyr, warn.conflicts = FALSE)
library(tidyr)
```

3.3 Data Cleaning with `tidyr`

In order for `dplyr` to work the data must be *tidy*, i.e., it must be structured as a data frame with certain characteristics.

This section extracts text and code from Hadley's Wickham's vignette for the `tidyr` package. Click on the link to his repo for his tidyr package to find the tidyr Tidy-data Vignette. For your convenience, the file for the vignette and the required data sets are in this section directory. More detailed discussions are given in his tidy data paper.

Hadley states that 80% of data analysis is spent on the cleaning and preparing data. Further, it must be repeated many times over the course of analysis as new problems come to light or new data is collected. His vignette and paper focuses on an important aspect of data cleaning: *data tidying*, i.e., structuring datasets to facilitate analysis.

The principles of tidy data provide a standard way to organize data values within a dataset. The *tidy data standard* has been designed to:

- facilitate initial exploration and analysis of the data, and
- simplify the development of data analysis tools, e.g., `dplyr` and `ggplot`, that work well together.

Current tools often require translation, i.e., you have to spend time *munging* the output from one tool so you can input it into another. Tidy datasets and tidy tools work hand in hand to make data analysis easier, allowing you to focus on the interesting domain problem, not on the uninteresting logistics of data.

3.3.1 Data Frames

We first look at the structure and semantics of data before more formally specifying how we make data tidy.

Data structure

Most *statistical datasets* can be represented in data frames made up of rows and columns. The columns (variables) are almost always labeled and the rows (observations) are sometimes labeled.

Unfortunately, the following code provides data in a format commonly seen.

```
preg <- read.csv("preg.csv", stringsAsFactors = FALSE)
preg
```

```
##           name treatmenta treatmentb
## 1   John Smith         NA         18
## 2     Jane Doe          4          1
```

```
## 3 Mary Johnson          6          7
```

The `preg` data does not follow the row-column format above, i.e, this data is not tidy! The `treatment` values (`a` and `b`) are the column names, i.e., the columns are not variables.

Data semantics

A *dataset* is a collection of values, usually either numbers (if *quantitative*) or strings (if *qualitative*). Values are organised in two ways. Every value belongs to:

- a *variable* containing all values that measure the same underlying attribute, e.g., height or temperature, across units;
- An *observation* containing all values measured on the same unit, e.g., a person or company, across attributes.

By convention, observations are stored in rows and variables are stored in columns. Without this convention, it would be impossible to define variable transformations and observation operations consistently.

Three verbs are the basis for tidying data (at least in most cases):

- `gather` takes multiple columns and turns them into key-value pairs;
- `spread` takes the key-value pairs and spreads them into multiple columns;
- `separate` pulls apart columns representing multiple variables.

In popular parlance `gather` makes "wide" data "long," whereas `spread` makes "long" data "wide."

The `gather` verb can make a tidy data frame for the pregnancy data with two `dplyr` helper verbs.

```
preg_df <- preg %>%
  gather(treatment, n, treatmenta:treatmentb) %>%
  mutate(treatment = gsub("treatment", "", treatment)) %>%
  arrange(name, treatment)
preg_df
```

```
##           name treatment  n
## 1     Jane Doe         a  4
## 2     Jane Doe         b  1
## 3   John Smith         a NA
## 4   John Smith         b 18
## 5 Mary Johnson         a  6
## 6 Mary Johnson         b  7
```

`gather()` takes the `treatmenta` and `treatmentb` columns and collapses them into *key-value pairs* with the key becoming `treatment` and the value becoming `n`, duplicating the other columns as needed. You use `gather()` when the column names are variable values—not variables. It is now clearer that the dataset contains 18 values, including one missing value, representing three variables (`name`, `treatment`, and `n`) and six observations.

The *experimental design* tells us something about the structure of the observations. In the pregnancy experiment, every combination of of `name` and `treatment` was measured, a *completely crossed design*. The experimental design also determines whether or not *missing values* can be safely dropped. There are two types of missing values:

- *simple missing values*: measurements that could have been made, or
- *structural missing values*: measurements that can't be made (e.g., the count of pregnant males).

The former missing values should be kept whereas the latter should be removed from the data set.

In a given analysis, there may be multiple levels of observation. For example, in a trial of new allergy medication we might have three observational types:

- demographic data collected from each person (age, sex, race),
- medical data collected from each person on each day (number of sneezes, redness of eyes), and
- meteorological data collected on each day (temperature, pollen count).

How is this data represented? Are the repeated measurements over days for each person spread across rows, e.g., to do multivariate modeling, or are days stretched out in a single column, e.g., to use multi-level models?

Tidy data

Tidy data is a standard way of mapping the meaning of a dataset to its structure. A dataset is *messy or tidy* depending on how *rows, columns and tables are matched up with observations, variables and types.* In *tidy data*:

- Each variable forms a column.
- Each observation forms a row.
- Each type of observational unit forms a table.

Thus, statisticians constructing data frames in R face the same issues as those faced by database designers. That is, tidy data is equivalent to *Codd's 3rd normal form* for relational databases, but with the constraints framed in statistical language. *Messy data* is any other other arrangement of the data. However, data analyses are done on a single data set—not connected tables in a relational database. Thus, data analysts must denormalize or merge the datasets back into one table.

Tidy data makes it easy for an analyst or a computer to extract needed variables or to construct new variables, e.g., `z = x / y`, because each column is a variable. Likewise, it is easy to compare groups of observations, e.g., `a` vs. `b`, because each observation is a row. Tidy data is particularly well suited for *vectorized* programming languages like R, because the layout ensures that values of different variables from the same observation are always paired.

The order of variables and observations does not affect the analysis, but a standardized way of representing data improves readability. One way of organizing variables is by their role in the analysis, i.e., variables are:

- *fixed* by the design of the experiment, or
- *measured* during the course of the experiment?

Fixed variables should come first, followed by measured variables, each ordered so that related variables are contiguous. Rows can then be ordered by the major fixed variable, breaking ties with the second and subsequent (fixed) variables.

3.3.2 Tidying Messy Datasets

The five most common problems with messy datasets are:

- Column headers are values, not variable names.
- Multiple variables are stored in one column.

- Variables are stored in both rows and columns.
- Multiple types of observational units are stored in the same table.
- A single observational unit is stored in multiple tables.

Surprisingly, most messy datasets, including types of messiness not explicitly described above, can be tidied with a small set of tools: `gather`, `separate` and `spread`.

Hadley provides a series of examples to remedy these common problems in his vignette. We will focus an a single example which illustrates gathering and spreading and the use of pipes to build a tidy workflow.

The most complicated form of messy data occurs when variables are stored in both rows and columns. The code below loads daily weather data from the Global Historical Climatology Network for one weather station (`MX17004`) in Mexico for five months in 2010.

```
weather <- as_tibble(read.csv("weather.csv", stringsAsFactors = FALSE))
weather
```

```
## # A tibble: 22 x 35
##    id     year month element    d1    d2    d3    d4    d5    d6    d7    d8
##    <chr> <int> <int> <chr>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
##  1 MX17...  2010     1 tmax       NA  NA    NA       NA  NA       NA    NA    NA
##  2 MX17...  2010     1 tmin       NA  NA    NA       NA  NA       NA    NA    NA
##  3 MX17...  2010     2 tmax       NA  27.3  24.1    NA  NA       NA    NA    NA
##  4 MX17...  2010     2 tmin       NA  14.4  14.4    NA  NA       NA    NA    NA
##  5 MX17...  2010     3 tmax       NA  NA    NA       NA  32.1    NA    NA    NA
##  6 MX17...  2010     3 tmin       NA  NA    NA       NA  14.2    NA    NA    NA
##  7 MX17...  2010     4 tmax       NA  NA    NA       NA  NA       NA    NA    NA
##  8 MX17...  2010     4 tmin       NA  NA    NA       NA  NA       NA    NA    NA
##  9 MX17...  2010     5 tmax       NA  NA    NA       NA  NA       NA    NA    NA
## 10 MX17...  2010     5 tmin       NA  NA    NA       NA  NA       NA    NA    NA
## # ... with 12 more rows, and 23 more variables: d9 <lgl>, d10 <dbl>, d11 <dbl>,
## #   d12 <lgl>, d13 <dbl>, d14 <dbl>, d15 <dbl>, d16 <dbl>, d17 <dbl>,
## #   d18 <lgl>, d19 <lgl>, d20 <lgl>, d21 <lgl>, d22 <lgl>, d23 <dbl>,
## #   d24 <lgl>, d25 <dbl>, d26 <dbl>, d27 <dbl>, d28 <dbl>, d29 <dbl>,
## #   d30 <dbl>, d31 <dbl>
```

It has variables:

- in individual columns (`id`, `year`, `month`),
- spread across columns (`d1-d31`) i.e., days, and
- spread along rows (`tmin`, `tmax`), i.e., minimum and maximum temperatures.

Months with fewer than 31 days have structural missing values for the last day(s) of the month.

To tidy this dataset we first gather the day columns while dropping the many missing values, structural or not, clean with `mutate` and `select`, arrange by the fixed variables, and spread by element (`tmax` and `tmin`).

```
weather_tbl <- weather %>%
  gather(day, value, d1:d31, na.rm = TRUE) %>%
  mutate(day = as.integer(readr::parse_number(day))) %>%
  select(id, year, month, day, element, value) %>%
  arrange(id, year, month, day) %>%
  spread(element, value)
weather_tbl
```

```
## # A tibble: 33 x 6
##    id       year month   day  tmax  tmin
##    <chr>   <int> <int> <int> <dbl> <dbl>
##  1 MX17004  2010     1    30  27.8  14.5
##  2 MX17004  2010     2     2  27.3  14.4
##  3 MX17004  2010     2     3  24.1  14.4
##  4 MX17004  2010     2    11  29.7  13.4
##  5 MX17004  2010     2    23  29.9  10.7
##  6 MX17004  2010     3     5  32.1  14.2
##  7 MX17004  2010     3    10  34.5  16.8
##  8 MX17004  2010     3    16  31.1  17.6
##  9 MX17004  2010     4    27  36.3  16.7
## 10 MX17004  2010     5    27  33.2  18.2
## # ... with 23 more rows
```

Note, we could reconstruct the non-structural missing values since we know how many days are in each month.

This example and the others in Hadley's vignette show the power of `tidyr` together with `dplyr` in cleaning data.