Yijia-Qiao-Post02

Yijia Qiao

12/2/2017

Data Formats Transformation and Better Data Visualization

Outline of this Post

Motivation

As Professor Gaston has introduced us two important methods of contacting data, Data Manipulation, and Data Visualization, we have already learned two basic and functional packages, `dplyr` and `ggplot2`. However, they both have some restrictions when we approch to more sophisticated data and want more interactive graphs. Fortunately, there are two packages available in cran library– `reshape2` and `plotly`.

1. `reshape2` provides a way to transform wide data structures to long forms, which are more easily to be analyzed in most functions.
2. `plotly` introduces a reader-friendly and interactive plotting methods. Although embedded in the file, the readers still can test and try their own values in order to understand the functions.

(All the datasets I used as examples are build-in R objects. You can feel free to copy and paste codes to your Rstudio and try the functions by yourself! ^^)

Preparation

1. Installation

```
# install `reshape2`
install.packages('reshape2')

#install `plotly`
# (install from CRAN)
install.packages("plotly")
# (install the latest development version (on GitHub) via devtools)
devtools::install_github("ropensci/plotly")
```

2. Getting Started

```
library('reshape2')
library('plotly')
```

Introduction

1. `reshape2`

`reshape2` is an `R` package written by Hadley Wickham that makes it easy to transform data between wide and long formats.

Wide data has a column for each variable. For example, this is wide-format data:

```
# airquality is a build-in R dataset.
head(airquality)
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
```

And below is a long-format data:

```
head(melt(airquality))
```

```
##   variable value
## 1    Ozone    41
## 2    Ozone    36
## 3    Ozone    12
## 4    Ozone    18
## 5    Ozone    NA
## 6    Ozone    28
```

Long-format data has a column for possible variable types and a column for the values of those variables. But we can give long-format data different levels of "longness". That is, thw long-format data does not necessarily have only two columns. For example, we could have ozone mesaurements for each day of the year (the detailed instructions will be in Discussion section).

When we say tidy data, in most situations, we mean long-format data. Long-format data is more professional and technical in data analysis. For example, `ggplot2` requires long-format data, `plyr` requires long-format data, and most modelling functions (such as `lm()`, `glm()`, and `gam()`) require long-format data. That is why we need `reshape2` to help transform data dormats.

2. `plotly`

`plotly` is an `R` package for creating interactive web graphics via the open source JavaScript graphing library `plotly.js`.

Discussion

1) Overview of `reshape2` package

There are two key functions in `reshape2`: `melt()` and `cast()`

- `melt()` – wide- to long-format data

`melt()` can deal with wide-format data and transform them into long-format data. By default, `melt()` has assumed that all columns with numeric values are variables with values. Often this is what we want. But, sometimes we want to know more about the values of ozone, solar.r, wind, and temp for each month and day. `melt()` has embedded commands to allow the operation.

First let me introduce the command "ID variables", `id.vars`. ID variables are the variables that identify individual rows of data.

```
names(airquality) <- tolower(names(airquality))
aql <- melt(airquality, id.vars = c("month", "day"))
head(aql)
```

```
##   month day variable value
## 1     5   1    ozone    41
## 2     5   2    ozone    36
## 3     5   3    ozone    12
## 4     5   4    ozone    18
## 5     5   5    ozone    NA
## 6     5   6    ozone    28
```

Also, we can control the column names of long-format data via `melt()`:

```
aql <- melt(airquality, id.vars = c("month", "day"),
  variable.name = "climate_variable",
  value.name = "climate_value")
head(aql)
```

```
##   month day climate_variable climate_value
## 1     5   1            ozone            41
## 2     5   2            ozone            36
## 3     5   3            ozone            12
## 4     5   4            ozone            18
## 5     5   5            ozone            NA
## 6     5   6            ozone            28
```

- `cast` – long- to wide-format data

  `cast` has many versions of functions for different transformation demands. For example, we can use `dcast()` to transform into `data.frame` objects. Also we can use `acast()` to get a vector, an array, or a martix.

In the following example, we use `dcast()` to recover the `aql` data we got before via `melt()` function.

```
aql <- melt(airquality, id.vars = c("month", "day"))
aqw <- dcast(aql, month + day ~ variable)
head(aqw)
```

```
##   month day ozone solar.r wind temp
## 1     5   1    41     190  7.4   67
## 2     5   2    36     118  8.0   72
## 3     5   3    12     149 12.6   74
## 4     5   4    18     313 11.5   62
## 5     5   5    NA      NA 14.3   56
## 6     5   6    28      NA 14.9   66
```

Note that `month + day ~ variable` represents that `month` and `day` are the ID variables, and we want a column for each; then we tell `dcast`

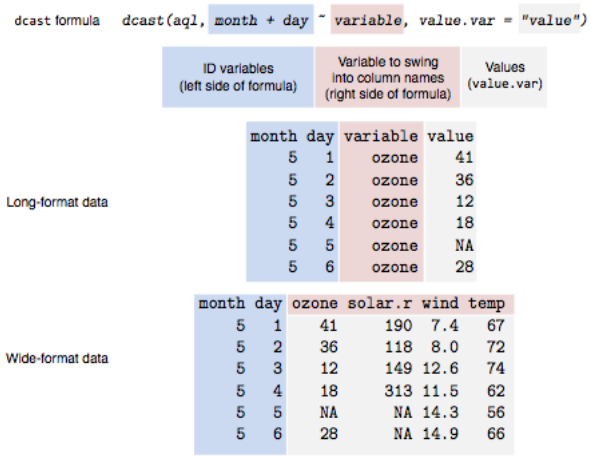
that `variable` describes the measured variables.

Then we compare the `aqw` with the original data format `airquality`.

```
head(airquality)
```

```
##   ozone solar.r wind temp month day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
```

The only difference is that those two datasets have different orders of columns. The scholar Sean C. Anderson has created a figure to illustrate the internal transformation of the data sets. Here is the figure:



Sean uses the blue shading to indicate ID variables that we want to represent individual rows, and the red shading represents variable names that we want to swing into column names, and the grey shading represents the data values that we want to fill in the cells with.

2) Overview of `plotly` package

- What is and interactive, web-based plot?

First, let us take a look at a function that calls interactive plots from our old friend, `ggplot2`. In `ggplot2`, `ggplotly()` converts your plots to an interactive, web-based version! It also provides sensible tooltips, which assists decoding of values encoded as visual properties in the plot.

```
library(ggplot2)
# `faithful` is a build-in R dataset
webplot <- ggplot(faithful, aes(x = eruptions, y = waiting)) +
  stat_density_2d(aes(fill = ..level..), geom = "polygon") +
  xlim(1, 6) + ylim(40, 100)
ggplotly(webplot)
```

Then let us see the `plotly` version of interactive plots. By default, Plotly for R runs locally in your web browser or in the R Studio viewer.

```
# `midwest` is a build-in R dataset
p <- plot_ly(midwest, x = ~percollege, color = ~state, type = "box")
p
```

- Cookbook for more `plotly`

  1> `plotly.js` supports some chart types that `ggplot2` doesn't (Refer to cheatsheet for more chart types). We can create any of these charts via `plot_ly()`.

```
# `volcano` is a build-in R dataset.
plot_ly(z = ~volcano, type = "surface")
```

2> `plotly` supports faceting. However, comparing to `ggplot2`, `plotly` cannot handle more than 9 facets.

```
# `diamonds` is a build-in R dataset
library(plotly)
set.seed(100)
d <- diamonds[sample(nrow(diamonds), 1000), ]
plot_ly(d, x = ~carat, y = ~price, color = ~carat,size = ~carat, text = ~paste("Clarity: ", clarity))
```

3> `plotly` has super fantastic graphin tools for 3D plots. Let us play with `mtcar` dataset.

```
mtcars$am[which(mtcars$am == 0)] <- 'Automatic'
mtcars$am[which(mtcars$am == 1)] <- 'Manual'
mtcars$am <- as.factor(mtcars$am)

p <- plot_ly(mtcars, x = ~wt, y = ~hp, z = ~qsec, color = ~am, colors = c('#BF382A', '#0C4B8E')) %>%
  add_markers() %>%
  layout(scene = list(xaxis = list(title = 'Weight'),
                      yaxis = list(title = 'Gross horsepower'),
                      zaxis = list(title = '1/4 mile time')))
p
```

We can change colors for each scales as well:

```
p <- plot_ly(mtcars, x = ~wt, y = ~hp, z = ~qsec,
        marker = list(color = ~mpg, colorscale = c('#FFE1A1', '#683531'), showscale = TRUE)) %>%
  add_markers() %>%
  layout(scene = list(xaxis = list(title = 'Weight'),
                      yaxis = list(title = 'Gross horsepower'),
                      zaxis = list(title = '1/4 mile time')),
         annotations = list(
           x = 1.13,
           y = 1.05,
           text = 'Miles/(US) gallon',
           xref = 'paper',
           yref = 'paper',
           showarrow = FALSE
         ))
p
```

4> R provides a way to animate plots. Both `ggplot2` and `plotly` can give us beautiful animated graphs.

First, let us see how `ggplotly()` works:

```
# `gapminder` is a build-in R dataset in package `gapminder` (make sure to install `gapminder` package first)
data(gapminder, package = "gapminder")
gg <- ggplot(gapminder, aes(gdpPercap, lifeExp, color = continent)) +
  geom_point(aes(size = pop, frame = year, ids = country)) +
  scale_x_log10()
ggplotly(gg)
```

Then let us see how `plotly` deals with key frame animation.

```
# `gapminder` is a build-in R dataset in package `gapminder` (make sure to install `gapminder` package first)
base <- gapminder %>%
  plot_ly(x = ~gdpPercap, y = ~lifeExp, size = ~pop,
          text = ~country, hoverinfo = "text") %>%
  layout(xaxis = list(type = "log"))

base %>%
  add_markers(color = ~continent, frame = ~year, ids = ~country) %>%
  animation_opts(1000, easing = "elastic", redraw = FALSE) %>%
  animation_button(
    x = 1, xanchor = "right", y = 0, yanchor = "bottom"
  ) %>%
  animation_slider(
    currentvalue = list(prefix = "YEAR ", font = list(color="red"))
  )
```

Notice that there is not a default “play” button shown on `plotly` results. That is because `plotly` has more powerful commands for interactive, web-based plots. We can use `animation_button()`, `animation_slider()` and others to customize our plots.

Take Home Messages

- There exist some differences between Long-format and Wide-format data. Long-format data makes functions and `ggplot2` graphing packages easier to deal with data. In other words, long-format data is more technical.
- `reshape2` provides two key functions to transform data formats: `melt()` and `cast`.
- Besides `ggplot2`, there is another powerful graphing package called `plotly`. `plotly` provides an interactive and web-based method for data visualization.

References

1. Sean C. Anderson’s blog for `reshape2`
2. Hadley’s `reshape2` package
3. R Markdown Cheatsheet
4. 3D Scatterplots in R
5. `plotly` cheatsheet
6. Create Interactive Web Graphics via ‘plotly.js’ by Carson Sievert
7. Getting Started with Plotly for R
8. Plotly for R