Post02

Joe Kiefer

December 3, 2017

Intro

Topic: Cleaning, reshaping, and wrangling data with the tidyr() and reshape2() packages

Description: This post will demonstrate how to use the tidyr() and reshape2() packages to clean, aggregate, reshape, and prepare raw data files and tables in order to perform a more robust analysis. In-text citations will be of the format ("Name of Article") since much of the info comes from articles/tutorials written by the same author.

Background

Since we have spent a significant amount of time manipulating and cleaning raw data, I was interested in learning about some of the conceptual theory behind how data is structured in tables and frames for analysis. This led me to the concept of "long" vs "wide" data, as well as how raw data can be altered and shaped to reflect these two forms for different reasons. I found that the two packages presented in this post can make interacting with the large number of columns present in our in-class data much more efficient, and can help cut out some of the intermediate steps of identifying, selecting/slicing, and editing the columns in our data frames.

tidyr() and reshape2()

The reshape2() package allows the user to move data tables back and forth from "wide" to "long" (concepts explained later) format using the cast() and melt() functions ("Package'reshape2'"). Before doing this, tidyr() can help to make the data more reliable and presentable by reassigning missing values, and restructuring raw data so it matches R data frame parameters ("tidyr 0.3.0"). It can be paired with packages such as dplyr() for more succint, concise code ("Package'tidyr'").

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

```
## Warning: package 'reshape2' was built under R version 3.4.3
```

```
library('tidyr')
```

```
## Warning: package 'tidyr' was built under R version 3.4.3
```

```
##
## Attaching package: 'tidyr'
```

```
## The following object is masked from 'package:reshape2':
##
##     smiths
```

```
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
```

Basic operations, tidyr()

fill() function

The fill() funtion of tidyr() can help with ensuring that repeated data is correctly rendered into R data frames when imported from external locations ("tidyr 0.3.0"). For example, if you are importing data from Excel, some users will leave cells blank to indicate that the answer from the preceding cells is carrying to the cells below (I have experienced this in some extracurricular clubs, where we frequently deal with attendance and "yes/no" answer spreadsheets) ("tidyr 0.3.0"). The fill() function will use the preceding response in a column to fill the consequent rows with that response until it encounters another, unique response ("tidyr 0.3.0").

As shown below, it fills rows with "Banquet" until "Retreat", "Yes" until it reaches "No", etc.

```
banquetatt <- dplyr::data_frame(
  event = c("Banquet", NA, NA, NA, NA, NA, NA, "Retreat", NA, NA, NA),
  resp = c("Yes", NA, "No", NA, "Maybe", NA, NA, "Yes", NA, "No", NA)
)
banquetatt
```

```
## # A tibble: 11 x 2
##       event  resp
##       <chr> <chr>
##   1 Banquet   Yes
##   2    <NA>  <NA>
##   3    <NA>    No
##   4    <NA>  <NA>
##   5    <NA> Maybe
##   6    <NA>  <NA>
##   7    <NA>  <NA>
##   8 Retreat   Yes
##   9    <NA>  <NA>
##  10    <NA>    No
##  11    <NA>  <NA>
```

```
banquetatt %>% fill(event, resp)
```

```
## # A tibble: 11 x 2
##       event  resp
##       <chr> <chr>
##   1 Banquet   Yes
##   2 Banquet   Yes
##   3 Banquet    No
##   4 Banquet    No
##   5 Banquet Maybe
##   6 Banquet Maybe
##   7 Banquet Maybe
##   8 Retreat   Yes
##   9 Retreat   Yes
##  10 Retreat    No
##  11 Retreat    No
```

replace_na()

When missing values are present, the replace_na() function is an easy way to define what NA should be replaced with in a data frame ("Package'tidyr'"). This is done by calling list() within replace_na() and defining what value NA in each column should take ("tidyr 0.3.0").

A simple example using numbers and "yes/no/maybe":

```
df <- dplyr::data_frame(
  x = c(57, 6, NA, 7, NA, 23),
  y = c("Yes", NA, "No", NA, NA, "No")
)
df
```

```
## # A tibble: 6 x 2
##       x     y
##   <dbl> <chr>
## 1    57   Yes
## 2     6  <NA>
## 3    NA    No
## 4     7  <NA>
## 5    NA  <NA>
## 6    23    No
```

```
df %>% replace_na(list(x = 0, y = "Maybe"))
```

```
## # A tibble: 6 x 2
##       x     y
##   <dbl> <chr>
## 1    57   Yes
## 2     6 Maybe
## 3     0    No
## 4     7 Maybe
## 5     0 Maybe
## 6    23    No
```

Basic operations, reshape2()

"Wide" vs "Long" data

Conceptually, there are many different ways to arrange data in table form depending on which values are most important, and how the user wants the values to appear aesthetically compared to other data points ("4 data wrangling"). As we have seen throughout this class, most raw data is read into R in a "wide" format. This means that row names correspond to a large number of columns, with numerical and character values displayed within the table.

For example, all the player names, teams, salaries, games played, and games started from the nba player stats data (nbastat):

```r
nba <- read.csv(file = 'https://raw.githubusercontent.com/ucb-stat133/stat133-fall-2017/master/data/nba2017-player
-statistics.csv')
nbastat <- nba[ ,c(1,2,5,8,9)]
head(nbastat, n = 15)
```

```
##                Player Team   Salary GP GS
## 1          Al Horford  BOS 26540100 68 68
## 2        Amir Johnson  BOS 12000000 80 77
## 3       Avery Bradley  BOS  8269663 55 55
## 4  Demetrius Jackson  BOS  1450000  5  0
## 5        Gerald Green  BOS  1410598 47  0
## 6       Isaiah Thomas  BOS  6587132 76 76
## 7         Jae Crowder  BOS  6286408 72 72
## 8         James Young  BOS  1825200 29  0
## 9        Jaylen Brown  BOS  4743000 78 20
## 10      Jonas Jerebko  BOS  5000000 78  6
## 11      Jordan Mickey  BOS  1223653 25  1
## 12       Kelly Olynyk  BOS  3094014 75  6
## 13       Marcus Smart  BOS  3578880 79 24
## 14       Terry Rozier  BOS  1906440 74  0
## 15       Tyler Zeller  BOS  8000000 51  5
```

This demonstrates “wide” data, with multiple columns showing different variables and corresponding values for each row in the table (“4 data wrangling”).

Different types of data visualization and analysis may require different types of tables. For example, while we have used wide data forms almost exclusively in class, very advanced, high-level analysis of data through packages such as ggplot2 actually tends to work better with data that is in long form (“4 data wrangling”). Overall, long structures are much better for grouping and identifying the actual measured values attributed to the predetermined “ID” values (usually factors/characters such as month, day, year, people, etc.), while wide forms are better for observing distribution and spread of data (“4 data wrangling”). For example, Sharon Machlis in her article gives the example that instead of having a wide data frame with Quarter 1, 2, 3 and 4 as separate columns, it may be more efficient to group Quarter variables together in one column with all of their values displayed in another column for easier graphing and manipulation (i.e. not having to identify each column as dat$Quarter1, and not having to created multiple new tables to attain the same clean data) (“4 data wrangling”).

melt()

The melt() function is useful for converting data frames, tables, matrices, etc. into a “long” format (“Reshape and aggregate”). While there are certain functions in tidy() that can perform a function similar to melt() (i.e. the gather() function), they require more manipulation of parameters and cannot be used to reshape data in arrays or matrices (“tidyr vs reshape2”).

We can again see the “wide” version of the nbastat table:

```r
head(nbastat)
```

```
##               Player Team   Salary GP GS
## 1         Al Horford  BOS 26540100 68 68
## 2       Amir Johnson  BOS 12000000 80 77
## 3      Avery Bradley  BOS  8269663 55 55
## 4 Demetrius Jackson  BOS  1450000  5  0
## 5       Gerald Green  BOS  1410598 47  0
## 6      Isaiah Thomas  BOS  6587132 76 76
```

Then, very easily, we can use melt() to turn this into a “long” form table:

```r
lnba <- melt(nbastat, id.vars = c('Player', 'Team'))
head(lnba, n = 30)
```

```
##                Player Team variable    value
## 1          Al Horford  BOS   Salary 26540100
## 2        Amir Johnson  BOS   Salary 12000000
## 3       Avery Bradley  BOS   Salary  8269663
## 4   Demetrius Jackson  BOS   Salary  1450000
## 5        Gerald Green  BOS   Salary  1410598
## 6       Isaiah Thomas  BOS   Salary  6587132
## 7         Jae Crowder  BOS   Salary  6286408
## 8         James Young  BOS   Salary  1825200
## 9        Jaylen Brown  BOS   Salary  4743000
## 10      Jonas Jerebko  BOS   Salary  5000000
## 11      Jordan Mickey  BOS   Salary  1223653
## 12       Kelly Olynyk  BOS   Salary  3094014
## 13       Marcus Smart  BOS   Salary  3578880
## 14       Terry Rozier  BOS   Salary  1906440
## 15       Tyler Zeller  BOS   Salary  8000000
## 16      Channing Frye  CLE   Salary  7806971
## 17      Dahntay Jones  CLE   Salary    18255
## 18     Deron Williams  CLE   Salary   259626
## 19  Derrick Williams  CLE   Salary   268029
## 20        Edy Tavares  CLE   Salary     5145
## 21      Iman Shumpert  CLE   Salary  9700000
## 22         J.R. Smith  CLE   Salary 12800000
## 23        James Jones  CLE   Salary  1551659
## 24         Kay Felder  CLE   Salary   543471
## 25         Kevin Love  CLE   Salary 21165675
## 26        Kyle Korver  CLE   Salary  5239437
## 27       Kyrie Irving  CLE   Salary 17638063
## 28       LeBron James  CLE   Salary 30963450
## 29 Richard Jefferson  CLE   Salary  2500000
## 30  Tristan Thompson  CLE   Salary 15330435
```

```
tail(lnba, n = 5)
```

```
##                Player Team variable value
## 1319 Marquese Chriss  PHO       GS    75
## 1320    Ronnie Price  PHO       GS     0
## 1321      T.J. Warren  PHO       GS    59
## 1322       Tyler Ulis  PHO       GS    15
## 1323  Tyson Chandler  PHO       GS    46
```

To produce the long table (lnba), we melt the table “nbastat” by identifying our “ID variables” (Player and Team) using “id.vars =” (“An Introduction to reshape2”). This tells the function that the data should be sorted according to these two variables. Then, by default, the function combines the rest of the columns (Salary, GP, GS) into two columns of “variable” (the variable names of the non-ID columns in order) and “value” (the corresponding numerical value of the variable for the certain player on the certain team in that row) (“An Introduction to reshape2”).

cast()

The cast() verb performs the opposite function of melt(), and can return data back to a wide format if necessary. This can done in a vector/array/matrix format with acast(), or in data frame form with dcast().

```
wnba <- dcast(lnba, formula = Player + Team ~ variable)
head(wnba)
```

```
##             Player Team   Salary GP GS
## 1     A.J. Hammons  DAL   650000 22  0
## 2     Aaron Brooks  IND  2700000 65  0
## 3     Aaron Gordon  ORL  4351320 80 72
## 4    Adreian Payne  MIN  2022240 18  0
## 5 Al-Farouq Aminu  POR  7680965 61 25
## 6       Al Horford  BOS 26540100 68 68
```

The parameters are more complex for cast(). First, a formula needs to be identified so the function knows which variables to leave as ID, which to turn into measurement variables, and what to use as the values in the table (“An Introduction to reshape2”). In the code above, the data being reshaped is the long nba table (lnba), with the formula identifying that “Player” and “Team” remain ID variables (Player + Team). The term following the “~” symbol identifies which variables will become the columns in the new table, where each unique input (Salary vs GP) will form a new column (“An Introduction to reshape2”). By default, if coded correctly, the function will assume that you want to fill the wide table with the numerical values in the “value” column of the long table (“An Introduction to reshape2”). If necessary, this can be explicitly specified by using the “value.var =” command (shown next in the full example) (“An Introduction to reshape2”).

Full example

In order to demonstrate a possible use of tidyr() and reshape2(), we will start with a raw table of the data defined above from “nbastat”:

```
head(nbastat, n = 20)
```

```
##              Player Team   Salary GP GS
## 1         Al Horford  BOS 26540100 68 68
## 2       Amir Johnson  BOS 12000000 80 77
## 3      Avery Bradley  BOS  8269663 55 55
## 4  Demetrius Jackson  BOS  1450000  5  0
## 5       Gerald Green  BOS  1410598 47  0
## 6      Isaiah Thomas  BOS  6587132 76 76
## 7        Jae Crowder  BOS  6286408 72 72
## 8        James Young  BOS  1825200 29  0
## 9       Jaylen Brown  BOS  4743000 78 20
## 10     Jonas Jerebko  BOS  5000000 78  6
## 11     Jordan Mickey  BOS  1223653 25  1
## 12      Kelly Olynyk  BOS  3094014 75  6
## 13      Marcus Smart  BOS  3578880 79 24
## 14      Terry Rozier  BOS  1906440 74  0
## 15      Tyler Zeller  BOS  8000000 51  5
## 16     Channing Frye  CLE  7806971 74 15
## 17     Dahntay Jones  CLE    18255  1  0
## 18    Deron Williams  CLE   259626 24  4
## 19  Derrick Williams  CLE   268029 25  0
## 20       Edy Tavares  CLE     5145  1  0
```

First, we can convert the data from “wide” to “long” format using melt():

```
longnba <- melt(nbastat,
 id.vars = c("Player", "Team"),
 variable.name =   'Variables',
 value.name = "Number")

head(longnba, n = 20)
```

```
##              Player Team Variables   Number
## 1         Al Horford  BOS    Salary 26540100
## 2       Amir Johnson  BOS    Salary 12000000
## 3      Avery Bradley  BOS    Salary  8269663
## 4  Demetrius Jackson  BOS    Salary  1450000
## 5       Gerald Green  BOS    Salary  1410598
## 6      Isaiah Thomas  BOS    Salary  6587132
## 7        Jae Crowder  BOS    Salary  6286408
## 8        James Young  BOS    Salary  1825200
## 9       Jaylen Brown  BOS    Salary  4743000
## 10     Jonas Jerebko  BOS    Salary  5000000
## 11     Jordan Mickey  BOS    Salary  1223653
## 12      Kelly Olynyk  BOS    Salary  3094014
## 13      Marcus Smart  BOS    Salary  3578880
## 14      Terry Rozier  BOS    Salary  1906440
## 15      Tyler Zeller  BOS    Salary  8000000
## 16     Channing Frye  CLE    Salary  7806971
## 17     Dahntay Jones  CLE    Salary    18255
## 18    Deron Williams  CLE    Salary   259626
## 19  Derrick Williams  CLE    Salary   268029
## 20       Edy Tavares  CLE    Salary     5145
```

We now have a “long” data frame of only 4 columns, with the “Player” names and “Team” as ID variables, all of the measured variable names in the “Variables” column, and each of their corresponding values in the “Number” column.

Since this is raw data, there may be missing values (NA) in the data. Thus, we will use tidyr() to replace the missing values with 0 using replace_na():

```
longnba2 <- longnba %>% replace_na(list(Number = 0))
head(longnba2, n = 15)
```

```
##              Player Team Variables   Number
## 1         Al Horford  BOS    Salary 26540100
## 2       Amir Johnson  BOS    Salary 12000000
## 3      Avery Bradley  BOS    Salary  8269663
## 4  Demetrius Jackson  BOS    Salary  1450000
## 5       Gerald Green  BOS    Salary  1410598
## 6      Isaiah Thomas  BOS    Salary  6587132
## 7        Jae Crowder  BOS    Salary  6286408
## 8        James Young  BOS    Salary  1825200
## 9       Jaylen Brown  BOS    Salary  4743000
## 10     Jonas Jerebko  BOS    Salary  5000000
## 11     Jordan Mickey  BOS    Salary  1223653
## 12      Kelly Olynyk  BOS    Salary  3094014
## 13      Marcus Smart  BOS    Salary  3578880
## 14      Terry Rozier  BOS    Salary  1906440
## 15      Tyler Zeller  BOS    Salary  8000000
```

.

Now let us assume that we have gotten the above table in “long” form as raw data, having been downloaded from Excel or Google Sheets. In that case, you may have to use the fill() function to ensure that repeat values such as Salary or team names are filling the necessary rows in the table. Using the cast function, we can take our raw table and create a "wide" table using the dcast() function of reshape2():

```
widenba <- dcast(longnba2,
  formula = Player + Team ~ Variables,
  value.var = "Number")

head(widenba, n = 20)
```

```
##             Player Team   Salary GP GS
## 1      A.J. Hammons  DAL   650000 22  0
## 2      Aaron Brooks  IND  2700000 65  0
## 3      Aaron Gordon  ORL  4351320 80 72
## 4     Adreian Payne  MIN  2022240 18  0
## 5   Al-Farouq Aminu  POR  7680965 61 25
## 6        Al Horford  BOS 26540100 68 68
## 7      Al Jefferson  IND 10230179 66  1
## 8     Alan Anderson  LAC  1315448 30  0
## 9     Alan Williams  PHO   874636 47  0
## 10       Alec Burks  UTA 10154495 42  0
## 11     Alex Abrines  OKC  5994764 68  6
## 12         Alex Len  PHO  4823621 77 34
## 13  Alex Poythress  PHI    31969  6  1
## 14    Alexis Ajinca  NOP  4600000 39 15
## 15     Allen Crabbe  POR 18500000 79  7
## 16     Amir Johnson  BOS 12000000 80 77
## 17  Andre Drummond  DET 22116750 81 81
## 18   Andre Iguodala  GSW 11131368 76  0
## 19   Andre Roberson  OKC  2183072 79 79
## 20 Andrew Harrison  MEM   945000 72 18
```

As we can see, this has taken the long data and given us a wide table in the same fomr as the original "nbastat" table. Notice the use of the "value.var =" command to explicitly define where the table values will come from, just in case the function does not correctly identify the "Number" column by default.

We have now been able to reshape some raw data with melt(), replace missing values with replace_na(), turn the clean data into a "long" form table for easier graphing and other purposes, and then reshape the table again to the "wide" form, thus brining us back to the original table (absent the same type of sorting, which can be done as desired).

Useful applications

These packages are very useful for cleaning, restructuring, and preparing raw data files for analysis with other R commands and plots. This is especially true for switching data tables between "long" and "wide" forms without having to use extremely long, tedious lines of codes to extract certain columns, combine/slice them, and then create new tables with the original corresponding values. Through tidyr(), it is also possible to ensure date from external sources such as Excel can be easily integrated into RStudio's format. In short, these two packages help to eliminate some of the most inefficient middle steps of data wrangling with base R commands, and can help make data much easier to plot and display in charts, tables, etc.

Limitations

While these two packages are very powerful and useful for reshaping and wrangling tough, untidy raw data, they are limited by their ability to process only variables with numerical values when melting and casting columns into long/wide form. Both tidyr() and reshape2() perform similar functions, but the output may be different and confusing at times due to differences in function structure. For example, gather() and melt() are very similar conceptually, but gather() will more often return errors that values will be dropped if the attributes are not of the same length, and also requires specific ID input code to work properly ("tidyr vs reshape2"). In a similar way, without defined ID inputs for reshape2() code, the package will assume any factor/character column is an ID column by default, which can produce unwanted/misrepresented results ("tidyr vs reshape2"). Overall, these packages do take some time to undestand, but certain commands from each used together are very effective at performing efficient, successful data wrangling.

Extra examples/info

Any of the references included below offer a wealth of information on how these packages work, as well as more advance wrangling techniques and countless examples. Especially useful are the CRAN package descriptions that show all of the possible commands and parameters. reshape2: https://cran.r-project.org/web/packages/reshape2/reshape2.pdf tidyr: https://cran.r-project.org/web/packages/tidyr/tidyr.pdf

References

Anderson, Sean. "An Introduction to reshape2." seananderson.ca, 19 Oct. 2013. http://seananderson.ca/2013/10/19/reshape.html. Accessed 2 Dec 2017.

Giudici, Alberto. "How to reshape data in R: tidyr vs reshape2." r-bloggers.com, 20 Jun. 2016. https://www.r-bloggers.com/how-to-reshape-data-in-r-tidyr-vs-reshape2/. Accessed 1 Dec 2017.

Machlis, Sharon. "4 data wrangling tasks in R for advanced beginners." computerworld.com, 14 Apr. 2015. https://www.computerworld.com/article/2486425/business-intelligence/business-intel ligence-4-data-wrangling-tasks-in-r-for-advanced-beginners.html?page=6 Accessed 1 Dec 2017.

"Reshape and aggregate data with the R package reshape2." r-bloggers.com, 31 Oct. 2013. https://www.r-bloggers.com/reshape-and-aggregate-data-with-the-r-package-reshape2/ . Accessed 1 Dec 2017.

Wickham, Hadley. "Package 'reshape2'." cran.r-project.org, 22 Oct. 2016. https://cran.r-project.org/web/packages/reshape2/reshape2.pdf. Accessed 2 Dec 2017.

Wickham, Hadley. "Package 'tidyr'." cran.r-project.org, 17 Oct. 2017. https://cran.r-project.org/web/packages/reshape2/reshape2.pdf. Accessed 1 Dec 2017.

Wickham, Hadley. "tidyr 0.3.0." blog.rstudio.com, 13 Sep. 2015. https://blog.rstudio.com/2015/09/13/tidyr-0-3-0/. Accessed 2 Dec 2017.