Random Data Set

24 April, 2015

Contents

This post will discuss a recent GitHub package I'm working on, **wakefield** to generate random data sets.

One of my more popular blog posts, Function To Generate A Random Data Set, was an early post about generating random data sets. Basically I had created a function to generate a random data set of various types of continuous and categorical columns. Optionally, the user could assign a certain percentage of cells in each column to missing values (`NA`). Often I find myself generating random data sets to test code/functions/models out on but rarely do I use that original random data generator. Why?

1. For one it's not in a package so it's not handy

2. It generates too many unrelated columns

Recently I had an idea inspired by Richie Cotton's **rebus** and Kevin Ushey & Jim Hester's **rex** regex based packages. Basically, these packages allow the user to utilize many little human readable regular expression chunks to build a larger desired regular expression. I thought, why not apply this concept to building a random data set. I'd make mini, modular random variable generating functions that the user passes to a `data.frame` like function and the result is a quick data set just as desired. I also like the way **dplyr** makes a `tbl_df` that prints only a few rows and limits the number of columns. So I made the output a `tbl_df` object and print accordingly.

1 Demo

1.1 Random Variable Functions

First we'll use the **pacman** package to grab and load the **wakefield** package from GitHub.

```
if (!require("pacman")) install.packages("pacman"); library(pacman)
p_load_gh("trinker/wakefield")
```

Then we'll look at a random variable generating function.

```
race(n=10)
```

```
##  [1] White    White    White    Black    White    Hispanic Black
##  [8] Asian    Hispanic White
## Levels: White Hispanic Black Asian Bi-Racial Native Other Hawaiian
```

```
attributes(race(n=10))
```

```
## $levels
## [1] "White"     "Hispanic"  "Black"     "Asian"     "Bi-Racial" "Native"
## [7] "Other"     "Hawaiian"
##
## $class
## [1] "variable" "factor"
##
## $varname
## [1] "Race"
```

A few more...

```
sex(10)
```

```
##  [1] Male   Female Male   Male   Male   Female Male   Male   Male
## Levels: Male Female
```

```
likert_7(10)
```

```
##  [1] Strongly Agree    Strongly Agree    Neutral
##  [4] Somewhat Agree    Disagree          Disagree
##  [7] Somewhat Disagree Neutral           Strongly Agree
## [10] Agree
## 7 Levels: Strongly Disagree < Disagree < ... < Strongly Agree
```

```
gpa(10)
```

```
##  [1] 3.00 3.67 2.67 3.33 3.00 4.00 3.00 3.00 3.67 3.00
```

```
dna(10)
```

```
##  [1] "Adenine"  "Thymine"  "Thymine"  "Thymine"  "Adenine"  "Cytosine"
##  [7] "Guanine"  "Thymine"  "Thymine"  "Guanine"
```

```
string(10, length = 5)
```

```
##  [1] "L3MPu" "tyTgQ" "mqBWh" "uGnch" "6KKZC" "DdLrw" "t2lEJ" "Hir6Y"
##  [9] "eE4v9" "oPb4u"
```

1.2 Random Data Frames

Ok so modular chunks great... but they get more powerful inside of the `r_data_frame` function. The user only needs to supply `n` once and the column names are auto-generated by the function (can be specified with `name =` prefix as usual). The call parenthesis are not even needed if no other arguments are passed.

```
set.seed(10)

r_data_frame(
    n = 500,
    id,
    race,
    age,
    smokes,
    marital,
    Start = hour,
    End = hour,
    iq,
    height,
    died
)
```

```
## Source: local data frame [500 x 10]
##
##     ID     Race Age Smokes       Marital    Start      End  IQ Height
## 1  001    White  33  FALSE       Married 00:00:00 00:00:00  95     62
## 2  002    White  35  FALSE Never Married 00:00:00 00:00:00  94     69
## 3  003    White  33  FALSE     Separated 00:00:00 00:00:00 112     71
## 4  004 Hispanic  24  FALSE       Married 00:00:00 00:00:00  97     65
## 5  005    White  21  FALSE Never Married 00:00:00 00:00:00  89     74
## 6  006    White  28  FALSE       Married 00:00:00 00:00:00  93     67
## 7  007    White  22  FALSE       Married 00:00:00 00:00:00 113     66
## 8  008    White  21  FALSE Never Married 00:00:00 00:00:00 115     69
## 9  009    White  23  FALSE      Divorced 00:00:00 00:00:00  85     74
## 10 010    White  34  FALSE      Divorced 00:00:00 00:00:00 110     71
## .. ...      ... ...    ...           ...      ...      ... ...    ...
## Variables not shown: Died (lgl)
```

This `r_data_frame` is pretty awesome and not my own. Josh O'Brien wrote the function as seen HERE. Pretty nifty trick. Josh thank you for your help with bringing to fruition the concept.

1.3 Missing Values

The original blog post provided a means for adding missing values. **wakefield** keeps this alive and adds more flexibility. It is no longer a part of the data generation process but a function, `r_na`, that is called after the data set has been generated. The user can specify which columns to add `NA`s to. By default column 1 is excluded. This works nicely within a **dplyr**/**magrittr** pipe line. ***Note***: ***dpyr*** *has an* `id` *function ad well so the prefix* `wakedfield::` *must be used for* `id`.

```
p_load(dplyr)
set.seed(10)

r_data_frame(
    n = 30,
    id,
    state,
    month,
    sat,
    minute,
    iq,
    zip_code,
    year,
    Scoring = rnorm,
    Smoker = valid,
    sentence
) %>%
    r_na(prob=.25)
```

```
## Source: local data frame [30 x 11]
##
##    ID      State     Month  SAT   Minute  IQ   Zip Year     Scoring Smoker
## 1  01    Georgia      July 1315 00:03:00 106    NA   NA          NA     NA
## 2  02    Florida  February 1492 00:04:00 107 87108 2007  0.64350004   TRUE
## 3  03       Ohio     March 1597     <NA>  83 58653 2012 -1.36030614   TRUE
## 4  04         NA  November 1518 00:07:00  NA 50381 1999 -0.19850611   TRUE
## 5  05 California      June 1362 00:08:00 111 58123 1996          NA  FALSE
## 6  06   New York September 1356     <NA>  87 18479 2010  2.06820961   TRUE
## 7  07         NA        NA   NA     <NA> 111 97135 2007 -0.30528475  FALSE
## 8  08    Florida  December 1324 00:15:00  NA 99438 2010  0.28124561   TRUE
## 9  09 Washington        NA 1468 00:16:00  97 97135 1996          NA   TRUE
## 10 10       Ohio      July   NA 00:20:00  NA 58123 2014  0.04636144     NA
## .. ..        ...       ...  ...      ... ...   ...  ...         ...    ...
## Variables not shown: Sentence (chr)
```

1.4 Default Data Set

There's still a default data set function, `r_data`, in case the functionality of the original random data generation function is missed or if you're in a hurry and aren't too picky about the data being generated.

```
set.seed(10)

r_data(1000)
```

```
## Source: local data frame [1,000 x 8]
##
##       ID     Race Age    Sex     Hour  IQ Height  Died
## 1   0001    White  32 Female 00:00:00  91     63 FALSE
## 2   0002    White  31   Male 00:00:00  92     69  TRUE
## 3   0003    White  23 Female 00:00:00  94     67 FALSE
## 4   0004 Hispanic  28 Female 00:00:00 102     63 FALSE
## 5   0005    White  29 Female 00:00:00 103     74  TRUE
## 6   0006    White  25   Male 00:00:00  96     68  TRUE
## 7   0007    White  26 Female 00:00:00 115     70 FALSE
## 8   0008    White  23   Male 00:00:00 119     66  TRUE
## 9   0009    White  32   Male 00:00:00 107     74  TRUE
## 10  0010    White  32   Male 00:00:00 104     71  TRUE
## ..   ...      ... ...    ...      ... ...    ...   ...
```

2 Future Direction

Where will the **wakefield** package go from here? Well this blog post is a measure of public interest. I use it and at this point it lives on GitHub. I'd like interest in two ways: (a) users and (b) contributors. Users make the effort worth while and provide feedback and suggested improvements. Contributors make maintenance easier.

There is one area of improvement I'd like to see in the `r_data_frame` (`r_list`) functions. I like that I don't have to specify an `n` for each variable/column. I also like that column names are auto generated. I also like that **dplyr**'s `data_frame` function allows me to create a variable `y` based on column `x`. So I can make columns that are correlated or any function of another column.

```
p_load(dplyr)
set.seed(10)

dplyr::data_frame(
    x = 1:10,
    y = x + rnorm(10)
)
```

```
## Source: local data frame [10 x 2]
##
##     x        y
## 1   1 1.018746
## 2   2 1.815747
## 3   3 1.628669
## 4   4 3.400832
## 5   5 5.294545
## 6   6 6.389794
## 7   7 5.791924
## 8   8 7.636324
## 9   9 7.373327
## 10 10 9.743522
```

The user can use the modular variable functions inside of `dplyr::data_frame` and have this functionality but the column name and `n` must explicit be passed to each variable.

```
set.seed(10)

dplyr::data_frame(
    ID = wakefield::id(n=10),
    Smokes = smokes(n=10),
    Sick = ifelse(Smokes, sample(5:10, 10, TRUE), sample(0:4, 10, TRUE)),
    Death = ifelse(Smokes, sample(0:1, 10, TRUE, prob = c(.2, .8)), sample(0:1, 10, TRUE, prob = c(.7,
)
```

```
## Source: local data frame [10 x 4]
##
##    ID Smokes Sick Death
## 1  01  FALSE    3     1
## 2  02  FALSE    2     0
## 3  03  FALSE    0     1
## 4  04  FALSE    2     0
## 5  05  FALSE    1     0
## 6  06  FALSE    2     1
## 7  07  FALSE    0     1
## 8  08  FALSE    1     0
## 9  09  FALSE    1     1
## 10 10  FALSE    4     0
```

I'd like to modify `r_data_frame` to continue to pass `n` and extract column names yet have the ability to make columns a function of other columns. Currently this is controlled by the `r_list` function that `r_data_frame` wraps.

3 Getting Involved

If you're interested in getting involved with use or contributing you can:

1. Install and use **wakefield**
2. Provide feedback via comments below
3. Provide feedback (bugs, improvements, and feature requests) via wakefield's Issues Page
4. Fork from GitHub and give a Pull Request

Thanks for reading, your feedback is welcomed.

**Get the R code for this post HERE* *Get a PDf version this post HERE