"there is art to data science"

NBME useRs

VOL.II. . . No.3 *AUGUST 9, 2021* PRICELESS

Welcome to the SPECIAL PSYCHOMETRIC SUMMER INTERN EDITION of the NBME useRs newsletter! After a short introduction on coding best practices, this newsletter will have R package reviews that were kindly contributed by the summer psychometric interns, as well as package reviews suggested or inspired by their work. The summer interns actually had so many good suggestions for things to cover that I'll have to spread the information over two newsletters. I sincerely appreciate their help, especially given their other working demands.

Syntax Style Guide

At some level there is no penalty for incidental decisions when programming. Most of the time when you are writing / reading syntax there is an over-arching goal of achieving some desired outcome, which can be something like estimating some model parameter (item difficulties, individual ability levels, identifying enemy item probabilities), transforming or matching data across different sources, producing a set of weights for an automated scoring model, etc.

Does it matter how you style your syntax? Are there naming and style conventions? As you have probably have guessed, many people believe the answer is yes.

Hadley Wickham developed the tidyverse style guide based on the original Google R style guide, but now the new Google R style guide is based on Hadley's guide. I recommend reading Hadley's guide when you have the time. It doesn't take long and is definitely worth it. Nonetheless, I've summarized some of the main points below. I didn't realize it at the time, but when I took my first `R` course my instructor made us conform to most of these maxims.

Naming Objects, Functions, and Variables

The names of most objects and variables should be succinct, descriptive, lowercase, and separated by a hyphen (`-`) or an underscore (`_`); the latter is sometimes called 'snake_case'. The use of periods (`.`) are discouraged because R functionality sometimes implements the period in some of its processes (e.g., when merging data with duplicate variables names but differing values, one will be `var.x` and the other `var.y`), so if you are also using periods it can create some redundancies / unexpected situations. While rare, when it happens it will completely break your code.

The use of camel case naming, where words are smashed together and differentiated by uppercase letters (`likeThis` or `LikeThis`), is discouraged because it encourages long names and doesn't promote succinct descriptive naming. (Admittedly I violate this rule sometimes because `CamelNamesAreFun` to look at.)

Practically, wouldn't it be helpful to have consistent naming conventions across departments / the organization? This way you would already know the formatting of the examinee ID variable (for example), so there isn't any need for syntax to convert variable names to execute functions or merge data.

`snake_case`

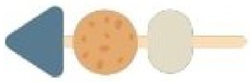
Pros: Concise when it consists of a few words.
Cons: Redundant as hell when it gets longer.
`push_something_to_first_queue, pop_what, get_whatever…`

`PascalCase`

Pros: Seems neat.
`GetItem, SetItem, Convert, ...`
Cons: Barely used. (why?)

`camelCase`

Pros: Widely used in the programmer community.
Cons: Looks ugly when a few methods are n-worded.
`push, reserve, beginBuilding, ...`



`skewer-case`

Pros: Easy to type.
`easier-than-capitals, easier-than-underscore, …`
Cons: Any sane language freaks out when you try it.

`SCREAMING_SNAKE_CASE`

Pros: Can demonstrate your anger with text.
Cons: Makes your eyes deaf.
`LOOK_AT_THIS, LOOK_AT_THAT, LOOK_HERE_YOU_MORON, …`

`nocase`

Pros: Looks professional.
Cons: Misleading af.
`supersexyhippothalamus,` **`bool`** `pencilislarge, ...`

`aNarCHY`

Pros: Can live outside of the law.
Cons: Can be out of a job.

Functions, "if" family statements

These should be well spaced, include an explicit return call, and have the first curly bracket on the initial line. In addition, `<-` is encouraged as the assignment operator to differentiate it from situations that use `==` to test inequality. While it may seem like overkill to include an explicit return statement, it's computationally cheap and worth the clarity.

```
# Good
simple_function <- function(x, y) {
  return(x * y)
}

# Bad (rumored Bill Gates style - parts of it, at least)
simpleFunction = function(x,y)
{
  x*y
}
```

Just Don't

Don't make implicit (untested) assumptions that your code will maintain the same type of object, especially when using functions / arguments that may change the type of object you're working with. Below, the top function returns a `logical` (binary) output, whereas the second provides an `integer` output. When both objects are converted to `numeric`, you get different results (second value given after each section of code - `1` and `3` respectively).

```
x <- c(1, 2, 3)

# Good
if (length(x) > 0) {
  print(length(x) > 0)
  as.numeric(length(x) > 0)
}
```

```
## [1] TRUE
```

```
## [1] 1
```

```
# Bad
if (length(x)) { # In a limited sense, also checks length(x) > 0
  print(length(x))
  as.numeric(length(x))
}
```

```
## [1] 3
```

```
## [1] 3
```

More Style

These are only a few examples and they are specific to the R language. In preparing this newsletter I found many more examples that I thought were interesting, such as the two packages Hadley Wickham recommends for styling - `styler` and `linter`, interesting blog posts 1 and 2, and US Geological Survey Suggestions.

R Package Reviews

R package: `igraph`

Guest Contributor: Ni Bei

igraph pdf vignette
A great package to visualize networks, and calculate descriptive network statistics.
Suggested audience: Researchers and People Interested in Graphing.

This package is a staring point for researchers interested in network analysis, especially in graphing people interactions and idea/concept transitions. It provides step-by-step instructions on how to format network data, and how to visualize them in multiple ways.

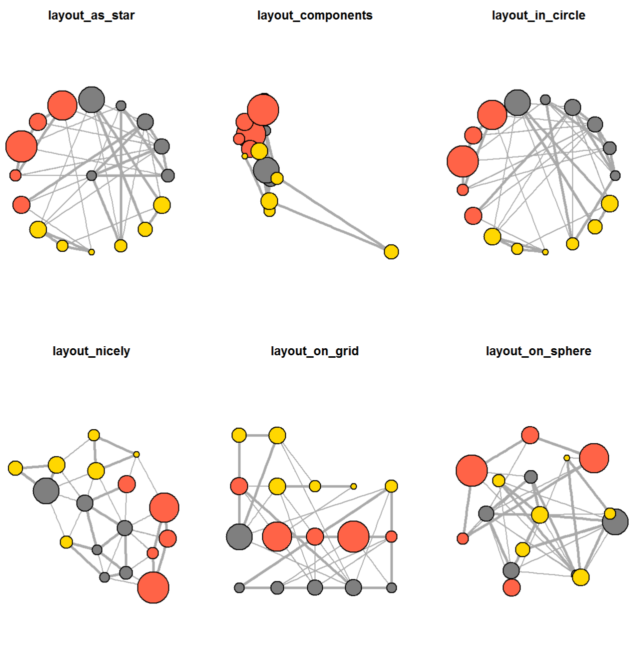


Figure 1: Some examples of beautiful igraph figures.

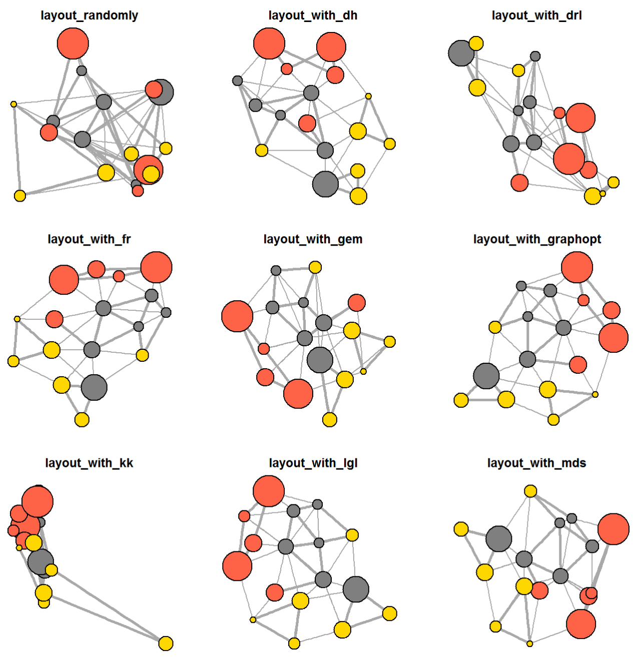


Figure 2: Some *more* examples of beautiful igraph figures.

R package: `reticulate`

Guest Contributor: Merve Sarac

reticulate pdf vignette 1 vignette 2 vignette 3 vignette 4
An efficient package that allows for high-performance interoperability between R and Python.
Suggested audience: An R developer that uses Python for some of their work or a data scientist that uses both the R and Python languages.

The reticulate package offers an extensive set of tools for interoperability between Python and R. The package translates between R and Python objects, calls and/or sources Python scripts, imports Python modules, and integrates Python interactively within R. If you are a user of both powerful languages, the reticulate package significantly simplifies your workflow. It enables data scientists/researchers who speak different languages to collaborate optimally on a project. It may also help a predominant R user comfortably transition from R to Python for some of their work, if interested/needed.

As Merve's summary above implies, organizations, teams, or even individual users don't need to pick a singular language to accomplish necessary tasks. Sure, it does help if multiple people on a team all code in the same language because you can work on the same code / proofread each other's work, but in a strict sense it isn't absolutely necessary. The people at R Studio recognized this fact when they started building the ability for R Studio IDE to handle and process *both* R and Python syntax.

The R Studio people recognize the benefit to using both languages, and even see it as supporting R Studio's charter as a Public Benefits Corporation. To that end, R Studio has made (and continues to make) several improvements to the R Studio environment to make it easier to use for Python users (e.g., 1, 2). To learn more about effectively using R and Python together, I encourage you to read this informative blog posted titled *Debunking the Myths of R vs. Python* and to watch this webinar called *Building Effective Data Science Teams*.

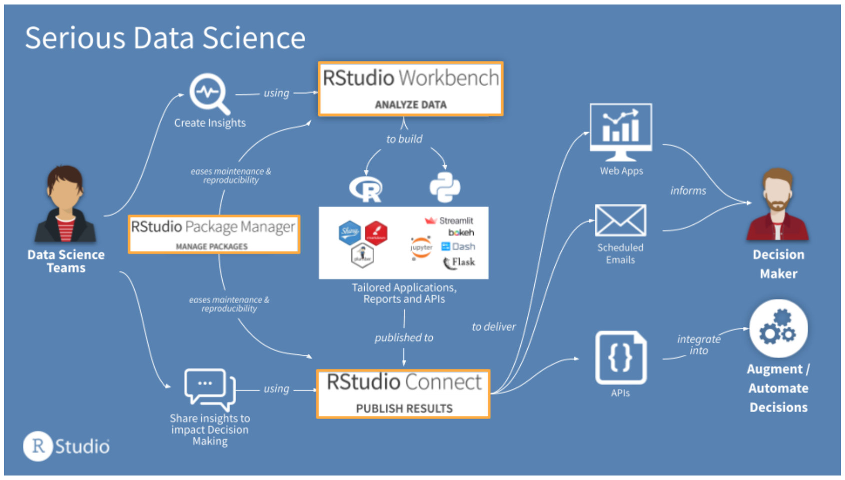


Figure 3: From *Debunking the Myths of R vs. Python*

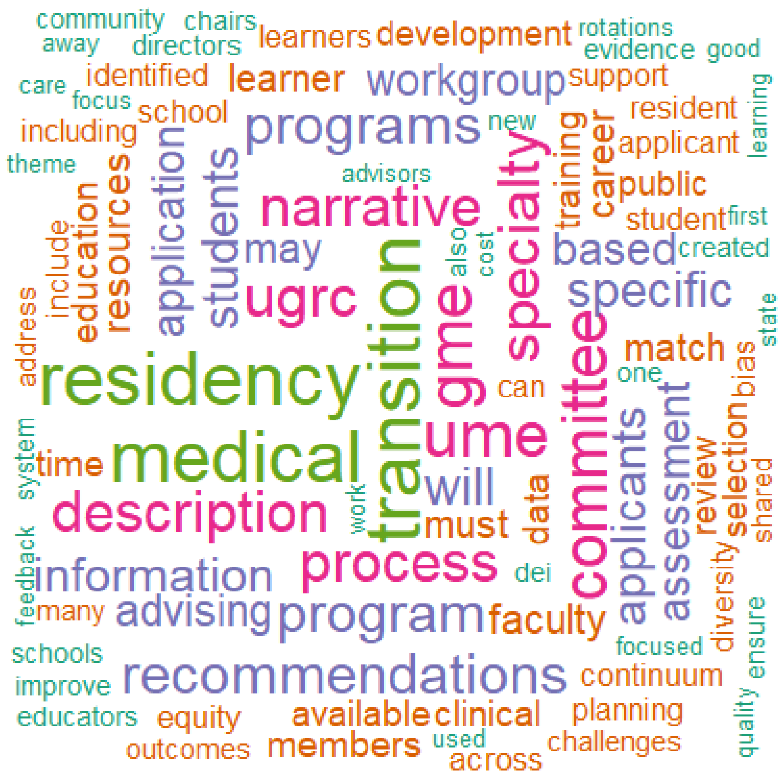
R package: `tm`

Guest Contributor: Magdalen Beiting-Parrish

tm pdf vignette 1 vignette 2 vignette 3
A simple and easy-to-use package for cleaning/processing text for deeper text mining work.
Suggested audience: People new to text mining/text pre-processing.

The tm package is a very simple package that has all of the functionality you need to turn text data into a Document Term Matrix (DTM) or Term Document Matrix (TDM) and also process the data to remove additional whitespace, convert everything to lower case, and reduce all words to their stems. The tm package has a small lexicon of stop words built in, but it also allows you to add in a few additional stop words as a short vector within the command line or even import a file that contains your custom stop words library. It is good for a novice text miner because it doesn't have too many options or places where things can go wrong. It is very limited; however, and for more advanced text cleaning/pre-processing, the qdap package is certainly better.

Interested researchers should also consult the Text Mining with R: A Tidy Approach book by Julia Silge and David Robinson that is freely available online, or also available for purchase from the publisher or Amazon if you prefer to work with a physical copy.



Wordcloud of the Preliminary Recommendations of the Undergraduate Medical Education to Graduate Medical Education Review Committee created using the tm, textreadr, pdftools, and wordcloud R packages.

R package: ggdist

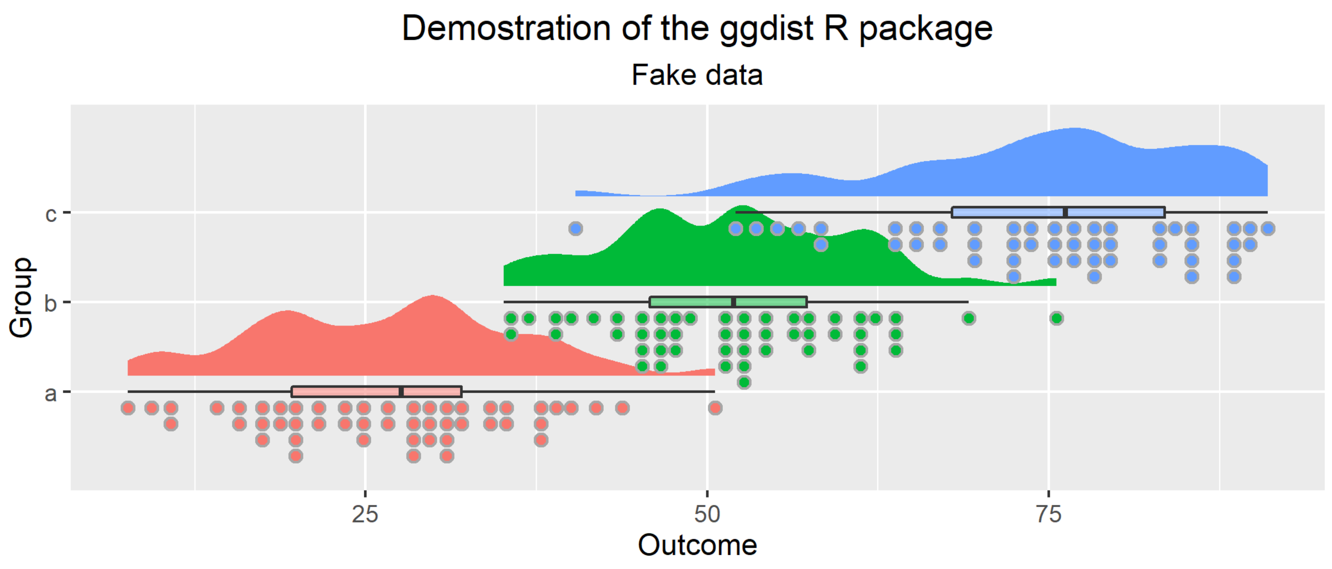
ggdist pdf vignette
A ggplot extension package to enhance density plots.
Suggested audience: Anyone looking for a snazzy plot to compare groups on some variable.

Makes "raincloud" plots - which consists of stacking a density plot, a boxplot, and then a histogram (with dots instead of bars). Pretty simple, but effective. I haven't completely jumped into everything the package can do - just followed the vignette to learn how to make the plot.

```
library(ggdist); library(ggplot2); set.seed(2872021)

fakedata2 <- data.frame(
  outcome <- c(rnorm(50, 25, 10), rnorm(50, 50, 10), rnorm(50, 75, 10)),
  groupz <- c(rep("a", times = 50), rep("b", times = 50), rep("c", times = 50)))

  ggplot(fakedata2, aes(x = factor(groupz), y = outcome, fill = factor(groupz))) +
  stat_halfeye(adjust = 0.5, justification = -.2,
               .width = 0, point_color = NA) +
  xlab("Group") + ylab("Outcome") +
  ggtitle("Demostration of the ggdist R package",
          subtitle = "Fake data") +
  theme(legend.position = "none") +
  theme(plot.title = element_text(hjust= 0.5)) +
  theme(plot.subtitle = element_text(hjust= 0.5)) +
  geom_boxplot(width = .12, outlier.color = NA, alpha = 0.5) +
  stat_dots(justification = 1.1, side = "left") + coord_flip()
```


Demostration of the ggdist R package
Fake data
Group
c
b
a
Outcome