**Analyzing A Leader Through Their Code**

It’s no secret that the R Programming Community has a number of leaders. It’s one of the draws that separates R from the rest of the pack! The leaders have earned their leadership position by making an impact through high-end data science and unselfishly giving back to the community. We can all learn from what they do. The key is dissecting their code bases to understand the tools and techniques they use.

What we’ve done is made a first step in figuring out why these individuals are successful through analyzing their most frequently used tools. We started with one standout, a true master of data science…

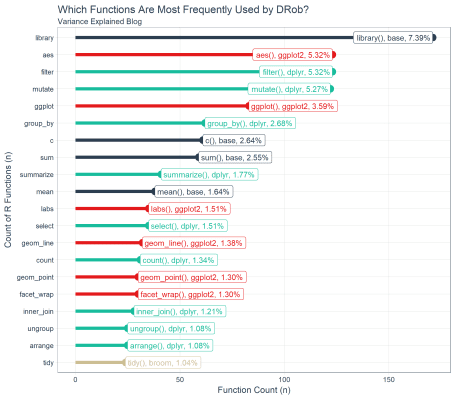
**Variance Explained Blog (Where We Got The Data)**

We used the rvest package to collect the code contained in each post. We started with one blog post involving mixture models of baseball statistics. We then extended it to all 58 articles to increase our confidence in what tools he frequently uses.

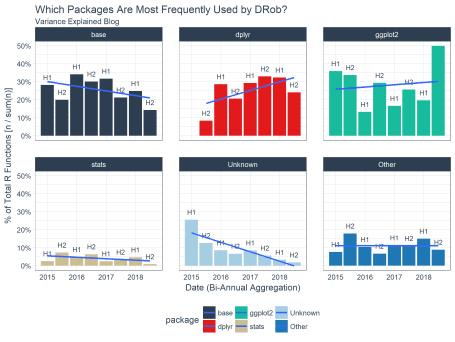
**80/20 Analysis For Learning R (What We Did With The Data)**

We output all of the high-usage functions and packages DRob regularly uses at the end of the article. This will be used in our next article developing a strategy to learn R efficiently.

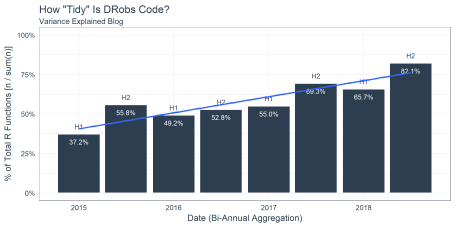
Our first graph helps us answer which functions DRob routinely uses. Here are his top 20 functions.



Our second graph helps us understand which packages are most frequently used based on the number of times functions within those packages appear in his code.



We noticed a theme that DRob is frequently using packages in the “tidyverse” (e.g. dplyr, ggplot2, etc). The third chart shows how “tidy” his code is by measuring the percentage of tidyverse functions vs non-tidyverse functions.



And, finally, we listed out the top 88 functions in our 80/20 analysis. These are the functions he used 80% of the time. Check out the below:

If we treat code on his Variance Explained blog as a text analysis, **we can find the most frequently used packages and functions that cover the majority of code he produces**. We can then use an 80/20 approach to determine which functions and packages are most used and therefore most important to master. We’ll split this analysis into two parts:

* Part 1: Web Scraping The Variance Explained Blog - Note this is a technical section showing how we retrieved the data. Novice learners may wish to skip this part.
* Part 2: Learning From Code - The analytical work is done here!

### Libraries

If you wish to follow along, please load the following libraries.

**library**(tidyverse)

**library**(tidyquant)

**library**(tibbletime)

**library**(rvest)

**library**(broom)

### Part 1: Web Scraping the Variance Explained Blog

This part of the analysis is **a how-to in web scraping**. We expose the process to collect the data, which use the rvest package and a number of custom functions to parse the text. We split this part into two steps:

* Step 1: Setup Code Parsing Functions
* Step 2: Web Scraping Variance Explained

#### Step 1: Setup Code Parsing and Utility Functions

This is a text analysis. As such, we are going to need to parse some text to extract function names, to determine which packages functions belong to, and to analyze the text with counts and percents. To do so, we create a few helper functions:

* count\_to\_pct(): Utility function to quickly convert counts to percentages. Works well with dplyr::count().
* parse\_function\_names(): Takes in text and returns function names.
* find\_functions\_in\_package(): Takes in a library or package name and returns all functions in the library or package.
* find\_loaded\_packages(): Detects which packages are loaded in the R system.
* map\_loaded\_package\_functions(): Maps the function names to the respective package.

##### Count To Percent

Function: Utility function to quickly convert counts to percentages. Works well with dplyr::count().

count\_to\_pct <- **function**(data, **...**, col = n) {

grouping\_vars\_expr <- quos(**...**)

col\_expr <- enquo(col)

data %>%

group\_by(!!! grouping\_vars\_expr) %>%

mutate(pct = (!! col\_expr) / sum(!! col\_expr)) %>%

ungroup()

}

Usage: Use dplyr::count() to retrieve counts by grouping variables. Then use count\_to\_pct() to quickly get percentages within groups. Exclude groups if overall percentages are desired. Note this example uses the mpg data set from ggplot2 package.

mpg %>%

count(manufacturer) %>%

count\_to\_pct() %>%

top\_n(5) %>%

arrange(desc(n))

## # A tibble: 5 x 3

## manufacturer n pct

## <chr> <int> <dbl>

## 1 dodge 37 0.158

## 2 toyota 34 0.145

## 3 volkswagen 27 0.115

## 4 ford 25 0.107

## 5 chevrolet 19 0.0812

##### Parse Function Names

Function: Takes in text and returns function names.

parse\_function\_names <- **function**(text, stop\_words = c("")) {

parser <- **function**(text, stop\_words) {

ret <- text %>%

str\_c(collapse = " ") %>%

str\_split("\\(") %>%

set\_names("text") %>%

as.tibble() %>%

slice(-n()) %>%

mutate(str\_split = map(text, str\_split, " ")) %>%

select(-text) %>%

unnest() %>%

mutate(function\_name = map\_chr(str\_split, ~ purrr::pluck(last(.x)))) %>%

select(function\_name) %>%

separate(function\_name, into = c("discard", "function\_name"),

sep = "(:::|::|\n)", fill = "left") %>%

select(-discard) %>%

mutate(function\_name = str\_replace\_all(function\_name,

pattern = "[^[:alnum:]\_\\.]", "")) %>%

filter(!(function\_name %**in**% stop\_words))

**return**(ret)

}

safe\_parser <- possibly(parser, otherwise = NA)

safe\_parser(text, stop\_words)

}

Usage: Parses the function name preceding “(“. Returns a tibble.

test\_text <- "my\_mean <- mean(1:10) some text base::sum(my\_mean) "

parse\_function\_names(test\_text)

## # A tibble: 2 x 1

## function\_name

## <chr>

## 1 mean

## 2 sum

##### Find Functions In Package

Function: Takes in a library or package name and returns all functions in the library or package.

find\_functions\_in\_package <- **function**(package) {

pkg\_text <- paste0("package:", package)

safe\_ls <- possibly(ls, otherwise = NA)

package\_functions <- safe\_ls(pkg\_text)

**if** (is.na(package\_functions[[1]])) **return**(package\_functions)

ret <- package\_functions %>%

as.tibble() %>%

rename(function\_name = value)

**return**(ret)

}

Usage: Takes in a package that is loaded via library(package\_name) (the package must be loaded for this to work properly). Returns a tibble of all functions in a package.

find\_functions\_in\_package("dplyr") %>% glimpse()

## Observations: 237

## Variables: 1

## $ function\_name <chr> "%>%", "add\_count", "add\_count\_", "add\_row"...

##### Find Loaded Packages

Function: Detects which packages are loaded in the R system.

find\_loaded\_packages <- **function**() {

ret <- search() %>%

list() %>%

set\_names("search") %>%

as.tibble() %>%

separate(search, into = c("discard", "keep"), sep = ":", fill = "right") %>%

select(keep) %>%

filter(!is.na(keep)) %>%

rename(package = keep) %>%

arrange(package)

**return**(ret)

}

Usage: Returns a tibble of loaded packages. Used in conjunction with map\_loaded\_package\_functions() to build a corpus of functions and associated packages, which is needed to determine which package the target function comes from.

find\_loaded\_packages() %>% glimpse()

## Observations: 28

## Variables: 1

## $ package <chr> "base", "bindrcpp", "broom", "datasets", "dplyr",...

##### Map Loaded Package Functions

Function: Maps the function names to the respective package.

map\_loaded\_package\_functions <- **function**(data, col) {

col\_expr <- enquo(col)

data %>%

mutate(function\_name = map(!! col\_expr, find\_functions\_in\_package)) %>%

mutate(is\_logical = map\_dbl(function\_name, is.logical)) %>%

filter(is\_logical != 1) %>%

select(-is\_logical) %>%

unnest()

}

Usage: Used in conjunction with find\_loaded\_packages(). Builds a corpus of package and function combinations for every package that is loaded. Returns a tibble of packages and associated functions.

find\_loaded\_packages() %>%

map\_loaded\_package\_functions(package) %>%

glimpse()

## Observations: 4,518

## Variables: 2

## $ package <chr> "base", "base", "base", "base", "base", "ba...

## $ function\_name <chr> "-", "-.Date", "-.POSIXt", "!", "!.hexmode"...

#### Step 2: Web Scraping Variance Explained

Now that we have the parsing and utility functions setup, we can begin the web scraping process to return the code on the [Variance Explained](http://varianceexplained.org/) blog. The first action is to setup our process for pulling the functions and packages for **a single post**. Once that process is defined, we can scale our web scraping to **ALL posts**!

##### Web Scrape A Single Blog Post

DRob has a number of posts that include code-throughs (walkthroughs using code). The code is contained within the HTML as a node named <code>. This makes it very easy to extract using the rvest package.



*Source: Understanding Mixture Models and Expectation-Maximization (Using Baseball Statistics), Variance Explained*

We can get the functions and packages used using the following code. It takes four steps:

1. Get a path for one of the articles. In our case, we chose DRob’s article on analyzing baseball stats with mixture models.
2. Create a corpus of all packages that are loaded in our R session. We will use this to determine which package the function that DRob uses comes from.
3. Use rvest functions read\_html() to read the HTML from the page. Then collect all nodes containing <code>. Then extract the text within those nodes using html\_text().
4. Run the text through our custom parse\_function\_names() function. This returns parsed function names. We still need the packages, which we can get by using left\_join() with our loaded\_functions\_tbl.

The final output is all of the functions and most of the package names for the functions that are used in this article! We use the glimpse() function to keep the output minimal.

# Assign one of the blog urls to a variable called path

path <- "http://varianceexplained.org/r/mixture-models-baseball/"

# Get the loaded functions (joined in last step)

loaded\_functions\_tbl <- find\_loaded\_packages() %>%

map\_loaded\_package\_functions(package)

# Read in HTML as text for all code attributes on the page

html\_code\_text <- read\_html(path) %>%

html\_nodes("code") %>%

html\_text()

# Parse function names and join with loaded functions

# Note that stats::filter and dplyr::filter conflict

# We replace any missing packages with "Unknown"

mixture\_models\_code\_tbl <- html\_code\_text %>%

parse\_function\_names() %>%

left\_join(loaded\_functions\_tbl) %>%

filter(!(function\_name == "filter" & !(package == "dplyr"))) %>%

mutate(package = case\_when(is.na(package) ~ "Unknown", TRUE ~ package))

mixture\_models\_code\_tbl %>% glimpse()

## Observations: 131

## Variables: 2

## $ function\_name <chr> "library", "library", "library", "library",...

## $ package <chr> "base", "base", "base", "base", "ggplot2", ...

From the output, we see that DRob used 131 functions in this particular article.

Next, we can then do a quick analysis to see what functions DRob used most frequently in this article. We can see that dplyr comes up quite frequently. The most popular function used is mutate(), which is used in this article about 11% of the time.

mixture\_models\_code\_tbl %>%

count(package, function\_name) %>%

count\_to\_pct() %>%

arrange(desc(n)) %>%

top\_n(5) %>%

knitr::kable()

| package | function\_name | n | pct |
| --- | --- | --- | --- |
| dplyr | mutate | 14 | 0.1068702 |
| base | sum | 9 | 0.0687023 |
| dplyr | group\_by | 9 | 0.0687023 |
| dplyr | filter | 7 | 0.0534351 |
| dplyr | ungroup | 6 | 0.0458015 |
| tidyr | crossing | 6 | 0.0458015 |

This is just one sample. We need more data to increase our confidence in which packages and functions are important.

##### Web Scrape All Blog Posts

Scaling to all blog posts is fairly easy with the purrr package. We need to do two things:

1. Web scrape all of the titles, dates, and paths for each of DRob’s articles using rvest.
2. Scale the analysis using purrr in combination with a custom function, build\_function\_names\_tbl\_from\_url\_path(), that we will create.

Web scraping the titles, dates, and paths (href) are again easy with the rvest package. On the [Variance Explained Posts](http://varianceexplained.org/posts/) page, we can again examine the HTML to find that the structure contains:

* Titles are stored in the article a nodes as text
* Dates are stored in the article p.datetime nodes as text
* Paths (href) are stored in the article a nodes as href attributes

We can extract this information using three web scrapings (one for title, dates, and hrefs), and then binding each together using bind\_cols(). We output the first six posts in a table using the head() function.

# Get the path to all of the posts

posts\_path <- "http://varianceexplained.org/posts/"

# Extract the post titles

titles\_vec <- read\_html(posts\_path) %>%

html\_node("#main") %>%

html\_nodes("article") %>%

html\_nodes("a") %>%

html\_text(trim = TRUE)

# Extract the post dates

dates\_vec <- read\_html(posts\_path) %>%

html\_node("#main") %>%

html\_nodes("article") %>%

html\_nodes("p.dateline") %>%

html\_text(trim = TRUE) %>%

mdy()

# Extract the post hrefs

hrefs\_vec <- read\_html(posts\_path) %>%

html\_node("#main") %>%

html\_nodes("article") %>%

html\_nodes("a") %>%

html\_attr("href")

# Bind the data together in a tibble

variance\_explained\_tbl <- bind\_cols(

title = titles\_vec,

date = dates\_vec,

href = hrefs\_vec)

# First six posts shown

variance\_explained\_tbl %>%

head() %>%

knitr::kable()

| title | date | href |
| --- | --- | --- |
| What digits should you bet on in Super Bowl squares? | 2018-02-04 | http://varianceexplained.org/r/super-bowl-squares/ |
| Exploring handwritten digit classification: a tidy analysis of the MNIST dataset | 2018-01-22 | http://varianceexplained.org/r/digit-eda/ |
| What’s the difference between data science, machine learning, and artificial intelligence? | 2018-01-09 | http://varianceexplained.org/r/ds-ml-ai/ |
| Advice to aspiring data scientists: start a blog | 2017-11-14 | http://varianceexplained.org/r/start-blog/ |
| Announcing “Introduction to the Tidyverse”, my new DataCamp course | 2017-11-09 | http://varianceexplained.org/r/intro-tidyverse/ |
| Don’t teach students the hard way first | 2017-09-21 | http://varianceexplained.org/r/teach-hard-way/ |

We now have all 58 of DRob’s posts (title, date, and href) and are now ready to scale! To simplify the process, we’ll create a custom function, build\_function\_names\_tbl\_from\_url\_path(), that combines several rvest operations from the previous section,

build\_function\_names\_tbl\_from\_url\_path <- **function**(path, loaded\_functions\_tbl) {

builder <- **function**(path, loaded\_functions\_tbl) {

read\_html(path) %>%

html\_nodes("code") %>%

html\_text() %>%

parse\_function\_names() %>%

left\_join(loaded\_functions\_tbl) %>%

filter(

!(function\_name == "filter" & !(package == "dplyr"))

) %>%

mutate(package = ifelse(is.na(package), "Unknown", package))

}

safe\_builder <- possibly(builder, otherwise = NA)

safe\_builder(path, loaded\_functions\_tbl)

}

We can test the function to see how it takes a path and a tibble of loaded\_functions\_tbl and returns the functions and packages.

path <- "http://varianceexplained.org/r/mixture-models-baseball/"

build\_function\_names\_tbl\_from\_url\_path(path, loaded\_functions\_tbl) %>%

glimpse()

## Observations: 131

## Variables: 2

## $ function\_name <chr> "library", "library", "library", "library",...

## $ package <chr> "base", "base", "base", "base", "ggplot2", ...

Next, we can scale this to all posts using map() functions from the purrr package. Several of the posts have no code and therefore return nested NA values. We filter them out by mapping is.logical. We unnest() the function\_name column to reveal the nested function names and packages.

variance\_explained\_tbl <- bind\_cols(

title = titles\_vec,

date = dates\_vec,

href = hrefs\_vec) %>%

mutate(

function\_name = map(href, build\_function\_names\_tbl\_from\_url\_path, loaded\_functions\_tbl),

is\_logical = map\_dbl(function\_name, is.logical)

) %>%

filter(is\_logical == 0) %>%

select(-is\_logical) %>%

unnest()

variance\_explained\_tbl %>% glimpse()

## Observations: 2,314

## Variables: 5

## $ title <chr> "What digits should you bet on in Super Bow...

## $ date <date> 2018-02-04, 2018-02-04, 2018-02-04, 2018-0...

## $ href <chr> "http://varianceexplained.org/r/super-bowl-...

## $ function\_name <chr> "library", "theme\_set", "theme\_light", "dir...

## $ package <chr> "base", "ggplot2", "ggplot2", "base", "purr...

Awesome - We now have all of the function names and most of the packages that DRob used ALL of his code on Variance Explained! Notice that the sample size has increased to 2314 functions extracted. This is a much larger sample size than before with the single Mixture Models post, which had 131 functions extracted.

### Part 2: Learning From Code

The question we need to answer is **“What Code Does An R Master Use To Perform Data Science?”** We can break this down into separate questions of interest:

#### Which Functions Are Most Frequently Used by DRob?

We can answer this question with by counting our package and function name frequencies, sorting, and taking the top 20, which gives us a subset of the most frequently used functions.

ve\_functions\_top\_20\_tbl <- variance\_explained\_tbl %>%

count(package, function\_name) %>%

count\_to\_pct() %>%

arrange(desc(n)) %>%

top\_n(20) %>%

mutate(function\_name = as\_factor(function\_name) %>% fct\_reorder(n)) %>%

arrange(desc(function\_name)) %>%

mutate(package = as\_factor(package))

ve\_functions\_top\_20\_tbl %>% glimpse()

## Observations: 20

## Variables: 4

## $ package <fct> base, ggplot2, dplyr, dplyr, ggplot2, dplyr...

## $ function\_name <fct> library, aes, filter, mutate, ggplot, group...

## $ n <int> 171, 123, 123, 122, 83, 62, 61, 59, 41, 38,...

## $ pct <dbl> 0.07389801, 0.05315471, 0.05315471, 0.05272...

We can visualize this data using ggplot2. We chose a lollipop style chart that extends lengthwise for the top 20, which shows off the number and percentage of total for each of the top 20 functions. We can see that base::library(), ggplot2::aes(), dplyr::filter(), and dplyr::mutate() are very frequently used by DRob. In fact, these four functions comprise 23.3% of his total functions. Unfortunately, aes() can’t be used alone (see below for how it’s used with the ggplot() function). **However, with knowledge of library() and the combination of filter() and mutate() from dplyr, a learner can understand 18% of DRob’s code!**

ve\_functions\_top\_20\_tbl %>%

ggplot(aes(x = n, y = function\_name, color = package)) +

geom\_segment(aes(xend = 0, yend = function\_name), size = 2) +

geom\_point(size = 4) +

geom\_label(aes(label = paste0(function\_name, "(), ", package, ", ", scales::percent(pct))),

hjust = "inward", size = 3.5) +

expand\_limits(x = 0) +

labs(

title = "Which Functions Are Most Frequently Used by DRob?",

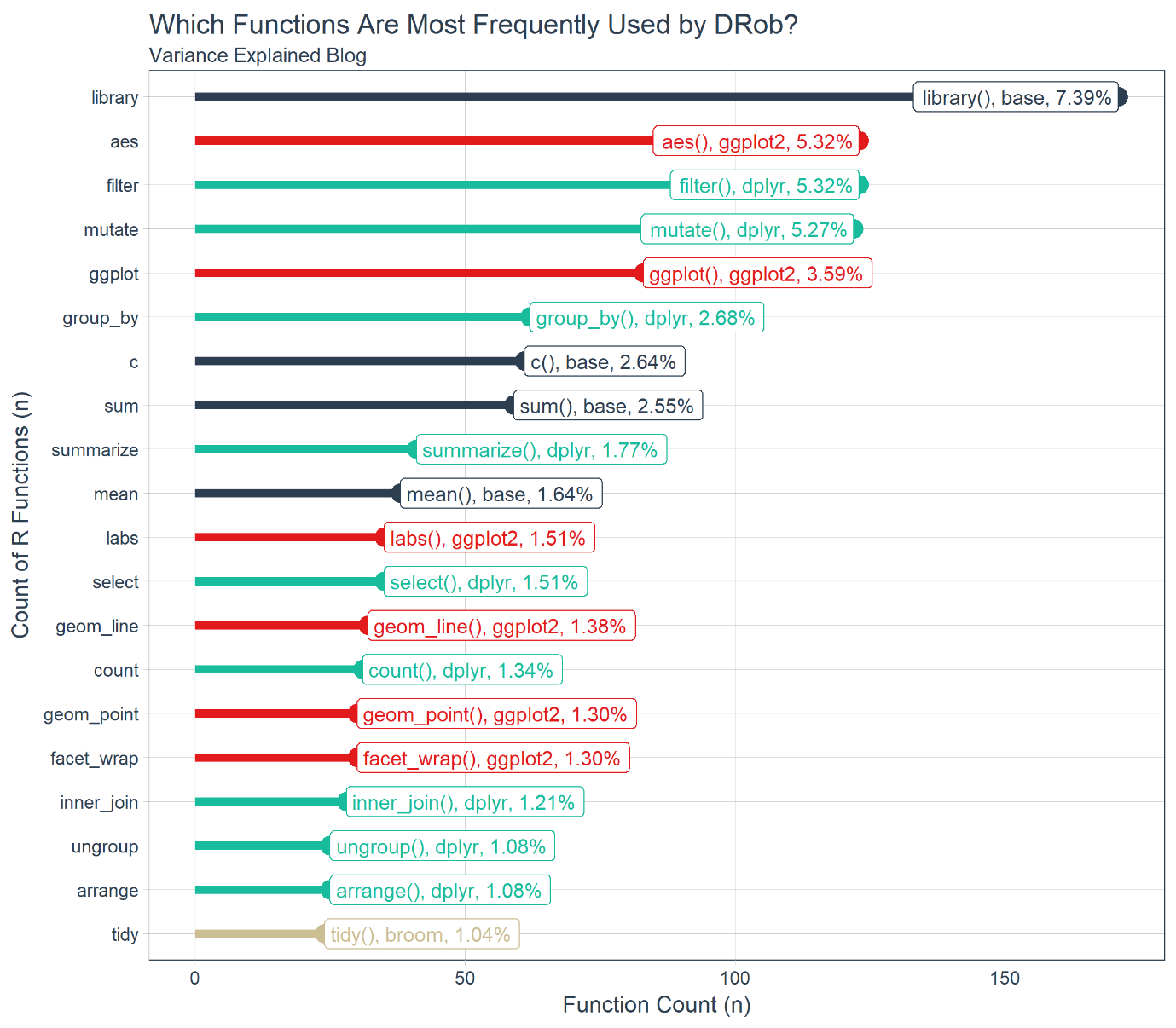
subtitle = "Variance Explained Blog",

x = "Function Count (n)", y = "Count of R Functions (n)") +

scale\_color\_tq() +

theme\_tq() +

theme(legend.position = "none")



#### Which Packages Are Most Frequently Used by DRob?

We can answer this question a number of ways, and we elect to make a time-based analysis to expose underlying trends within packages over time. The idea is that some packages may be used more frequently for specific reasons, and we aim to uncover the true trend of the packages which is not constant. We’ll use the tibbletime package to help out with the time-based analysis by aggregating (or grouping) the data by six-month intervals. Note that we lump (using fct\_lump()) all packages into six categories based on the top 5 packages and an extra column called “Other”. a label is made by pasting “H” with semester(date) to return the which half of the year the data is aggregated.

ve\_package\_frequency\_tbl <- variance\_explained\_tbl %>%

select(date, package, function\_name) %>%

mutate(package = as.factor(package) %>% fct\_lump(n = 5, other\_level = "Other")) %>%

arrange(date) %>%

as\_tbl\_time(index = date) %>%

collapse\_by(period = "6 m", clean = TRUE) %>%

count(date, package) %>%

count\_to\_pct(date) %>%

mutate(biannual = paste0("H", semester(date)))

ve\_package\_frequency\_tbl %>% glimpse()

## Observations: 47

## Variables: 5

## $ date <date> 2015-01-01, 2015-01-01, 2015-01-01, 2015-01-01,...

## $ package <fct> base, ggplot2, stats, Unknown, Other, base, dply...

## $ n <int> 22, 28, 2, 20, 6, 19, 8, 32, 7, 12, 17, 131, 110...

## $ pct <dbl> 0.28205128, 0.35897436, 0.02564103, 0.25641026, ...

## $ biannual <chr> "H1", "H1", "H1", "H1", "H1", "H2", "H2", "H2", ...

Next, we can visualize with ggplot2. **The total functions (column n in ve\_package\_frequency\_tbl) used are misleading since in some half years DRob posts less than in others**. We can normalize by switching to percentage of total functions by half year.

ve\_package\_frequency\_tbl %>%

ggplot(aes(date, n, fill = package)) +

geom\_bar(stat = "identity") +

geom\_text(aes(x = date, y = n, label = biannual),

vjust = -1, color = palette\_light()[[1]], size = 3) +

geom\_smooth(method = "lm", se = FALSE) +

facet\_wrap(~ package, ncol = 3) +

scale\_fill\_tq() +

theme\_tq() +

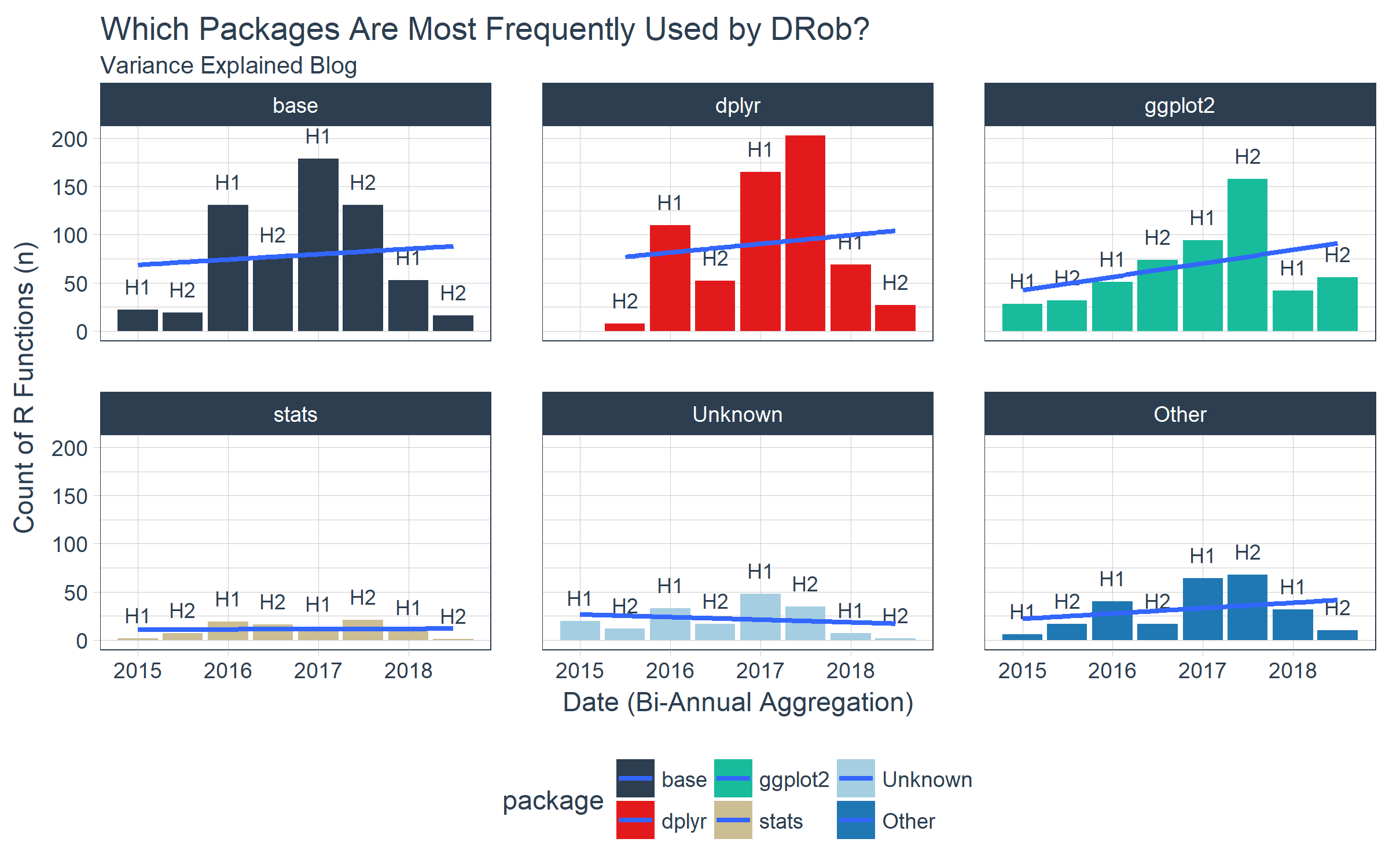
labs(

title = "Which Packages Are Most Frequently Used by DRob?",

subtitle = "Variance Explained Blog",

x = "Date (Bi-Annual Aggregation)", y = "Count of R Functions (n)"

)



We switch to a percentage of total functions (pct column in ve\_package\_frequency\_tbl) to get a better perspective on what trends are happening within posts over time. We see that DRob is trending in the direction of more dplyr and ggplot2 and using fewer “Unknown” packages, which are packages that I do not have currently loaded on my machine (e.g. not “tidyverse” or “base”). It’s clear that base, dplyr, and ggplot2 are DRob’s toolkits of choice.

ve\_package\_frequency\_tbl %>%

ggplot(aes(date, pct, fill = package)) +

geom\_bar(stat = "identity") +

geom\_text(aes(x = date, y = pct, label = biannual),

vjust = -1, color = palette\_light()[[1]], size = 3) +

geom\_smooth(method = "lm", se = FALSE) +

facet\_wrap(~ package, ncol = 3) +

scale\_y\_continuous(labels = scales::percent) +

scale\_fill\_tq() +

theme\_tq() +

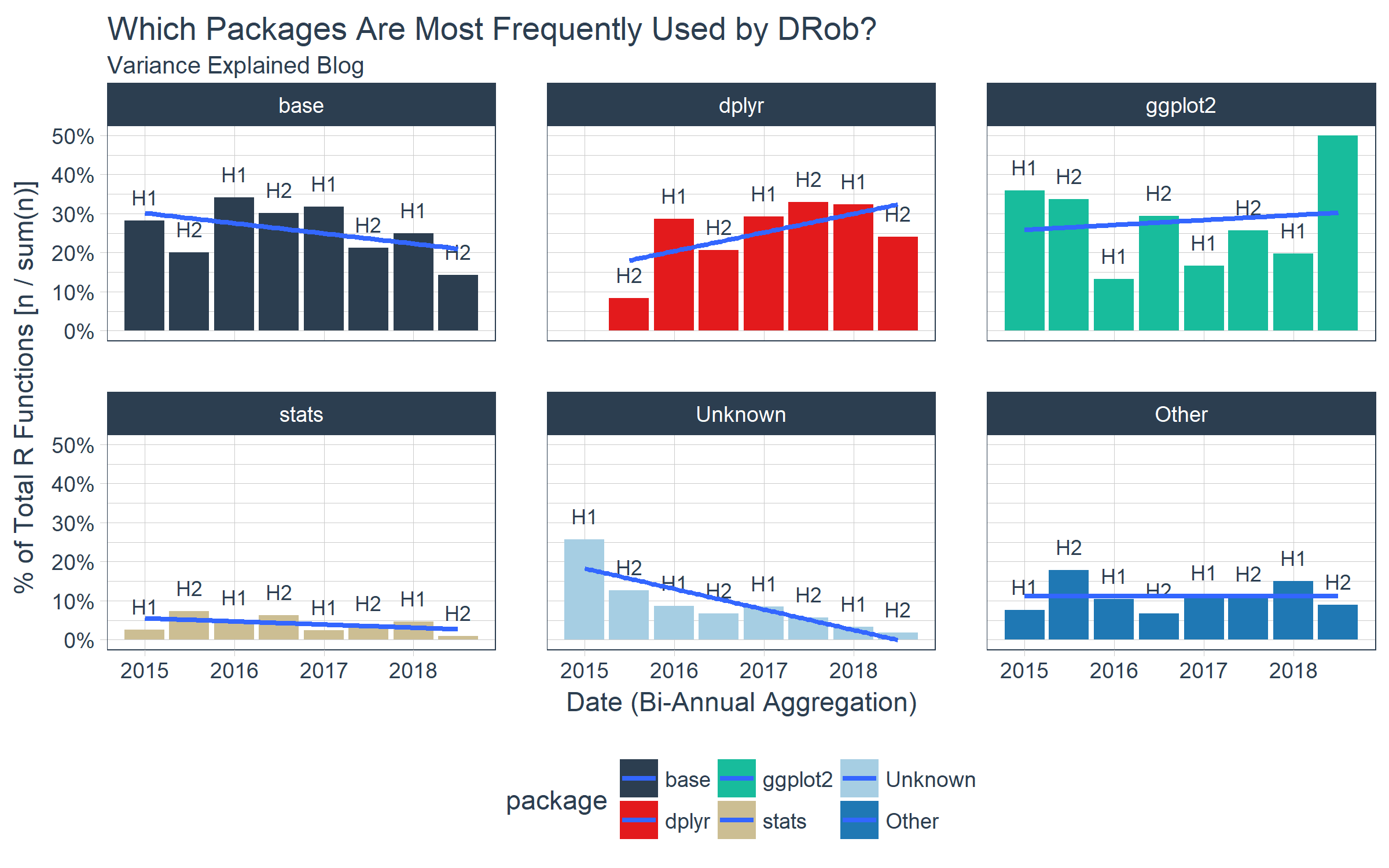
labs(

title = "Which Packages Are Most Frequently Used by DRob?",

subtitle = "Variance Explained Blog",

x = "Date (Bi-Annual Aggregation)", y = "% of Total R Functions [n / sum(n)]"

)



Finally, we can get the overall percentage of package usage by uncounting and recounting by package. We add a cumulative percentage column and see that we can almost get to 80% with just three package: dplyr, base, and ggplot2.

ve\_package\_frequency\_tbl %>%

uncount(weights = n) %>%

count(package) %>%

count\_to\_pct() %>%

arrange(desc(n)) %>%

mutate(pct\_cum = cumsum(pct)) %>%

knitr::kable()

| package | n | pct | pct\_cum |
| --- | --- | --- | --- |
| dplyr | 634 | 0.2739844 | 0.2739844 |
| base | 627 | 0.2709594 | 0.5449438 |
| ggplot2 | 535 | 0.2312014 | 0.7761452 |
| Other | 254 | 0.1097666 | 0.8859118 |
| Unknown | 174 | 0.0751945 | 0.9611063 |
| stats | 90 | 0.0388937 | 1.0000000 |

The tidyverse is a very popular set of packages that are developed specifically to do data science in an integrated and easy to understand way. Currently, the “tidyverse” consists of the following packages:

tidyverse\_packages(include\_self = F)

## [1] "broom" "cli" "crayon" "dplyr"

## [5] "dbplyr" "forcats" "ggplot2" "haven"

## [9] "hms" "httr" "jsonlite" "lubridate"

## [13] "magrittr" "modelr" "purrr" "readr"

## [17] "readxl\n(>=" "reprex" "rlang" "rstudioapi"

## [21] "rvest" "stringr" "tibble" "tidyr"

## [25] "xml2"

We can flag functions from the tidyverse package from DRob’s code base using the tidyverse\_packages() function. If functions are in a tidyverse package, the are flagged as “Yes” and otherwise “No”.

ve\_tidiness\_tbl <- variance\_explained\_tbl %>%

select(date, function\_name, package) %>%

mutate(tidy\_function = case\_when(

package %**in**% tidyverse\_packages() ~ "Yes",

TRUE ~ "No"))

ve\_tidiness\_tbl %>% glimpse()

## Observations: 2,314

## Variables: 4

## $ date <date> 2018-02-04, 2018-02-04, 2018-02-04, 2018-0...

## $ function\_name <chr> "library", "theme\_set", "theme\_light", "dir...

## $ package <chr> "base", "ggplot2", "ggplot2", "base", "purr...

## $ tidy\_function <chr> "No", "Yes", "Yes", "No", "Yes", "Yes", "Ye...

Here’s how easy it is to quickly see how tidy DRob is. About 60% of his functions are “tidyverse” functions.

ve\_tidiness\_tbl %>%

count(tidy\_function) %>%

count\_to\_pct() %>%

arrange(desc(n)) %>%

knitr::kable()

| tidy\_function | n | pct |
| --- | --- | --- |
| Yes | 1373 | 0.5933449 |
| No | 941 | 0.4066551 |

How has DRob’s “tidiness” changed over time? We’ll again call upon tibbletime to help transform the data using collapse\_by().

ve\_tidiness\_over\_time\_tbl <- ve\_tidiness\_tbl %>%

select(date, tidy\_function, function\_name, package) %>%

arrange(date) %>%

as\_tbl\_time(index = date) %>%

collapse\_by(period = "6 m", clean = TRUE) %>%

count(date, tidy\_function) %>%

count\_to\_pct(date) %>%

filter(tidy\_function == "Yes") %>%

mutate(biannual = paste0("H", semester(date)))

glimpse(ve\_tidiness\_over\_time\_tbl)

## Observations: 8

## Variables: 5

## $ date <date> 2015-01-01, 2015-07-01, 2016-01-01, 2016-0...

## $ tidy\_function <chr> "Yes", "Yes", "Yes", "Yes", "Yes", "Yes", "...

## $ n <int> 29, 53, 189, 133, 310, 427, 140, 92

## $ pct <dbl> 0.3717949, 0.5578947, 0.4921875, 0.5277778,...

## $ biannual <chr> "H1", "H2", "H1", "H2", "H1", "H2", "H1", "H2"

Here’s a fun fact… According to this graph, DRob is over twice as “tidy” now as when he started blogging in 2015. This should tell us that we really need to give the “tidyverse” a shot if we aren’t using it now.

ve\_tidiness\_over\_time\_tbl %>%

ggplot(aes(date, pct)) +

geom\_bar(stat = "identity", fill = palette\_light()[[1]], color = "white") +

geom\_text(aes(x = date, y = pct, label = biannual),

vjust = -1, color = palette\_light()[[1]], size = 3) +

geom\_text(aes(x = date, y = pct, label = scales::percent(pct)),

vjust = 2, color = "white", size = 3) +

geom\_smooth(method = "lm", se = FALSE) +

scale\_y\_continuous(labels = scales::percent) +

scale\_fill\_tq() +

theme\_tq() +

labs(

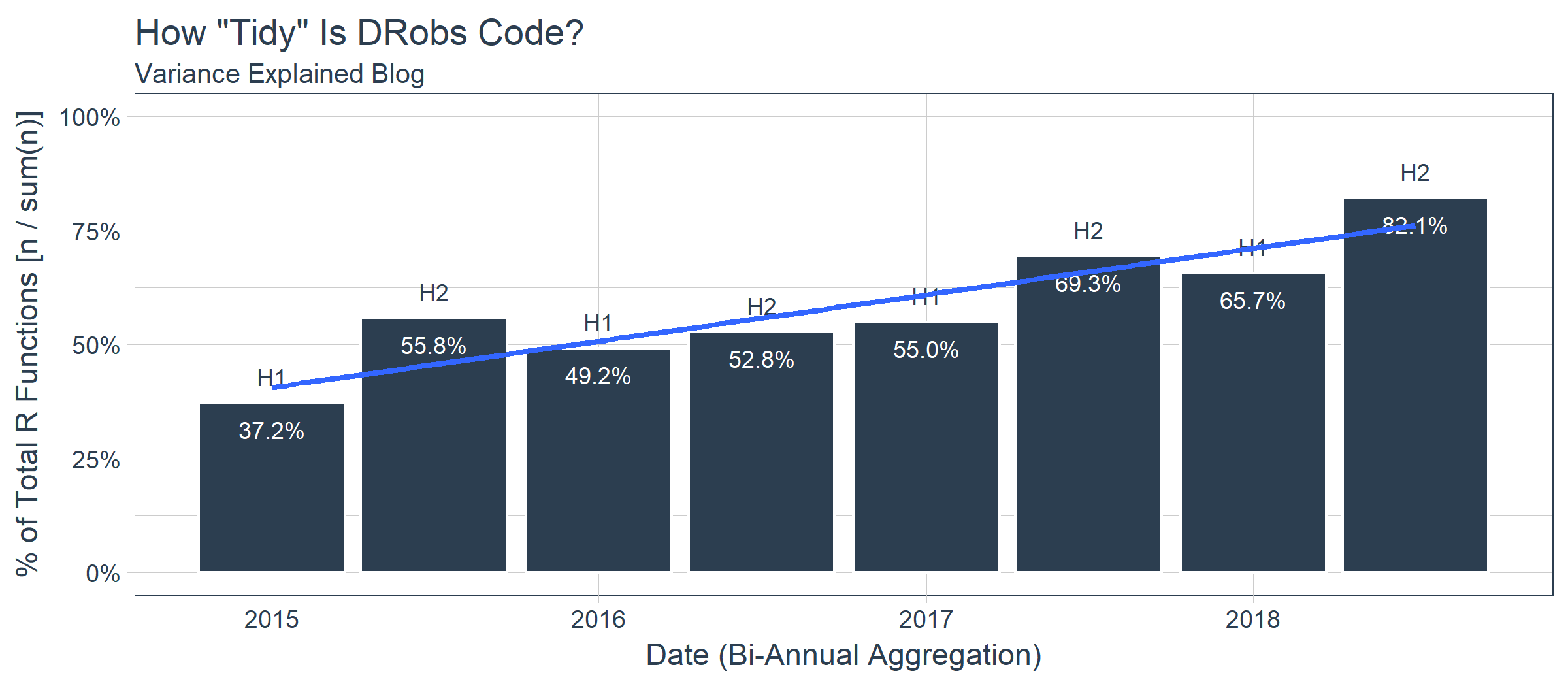
title = 'How "Tidy" Is DRobs Code?',

subtitle = "Variance Explained Blog",

x = "Date (Bi-Annual Aggregation)", y = "% of Total R Functions [n / sum(n)]"

) +

expand\_limits(y = 1)



#### Which Functions and Packages Should We Focus On For Learning R?

Now the million dollar question: What should we focus on if we are just starting out in R? We’ll use the 80/20 Rule, which boils down to which top functions build 80% of DRob’s code. Ideally this should be around 20% according to the rule. The question is actually really easy to answer using the cumsum() function from base. We can flag any cumulative percentages that are less than or equal to 80% as “high usage”.

ve\_eighty\_twenty\_tbl <- variance\_explained\_tbl %>%

count(package, function\_name) %>%

count\_to\_pct() %>%

arrange(desc(pct)) %>%

mutate(

pct\_cum = cumsum(pct),

high\_usage = case\_when(

pct\_cum <= 0.8 ~ "Yes",

TRUE ~ "No"

))

ve\_eighty\_twenty\_tbl %>% glimpse()

## Observations: 312

## Variables: 6

## $ package <chr> "base", "dplyr", "ggplot2", "dplyr", "ggplo...

## $ function\_name <chr> "library", "filter", "aes", "mutate", "ggpl...

## $ n <int> 171, 123, 123, 122, 83, 62, 61, 59, 41, 38,...

## $ pct <dbl> 0.073898012, 0.053154710, 0.053154710, 0.05...

## $ pct\_cum <dbl> 0.07389801, 0.12705272, 0.18020743, 0.23292...

## $ high\_usage <chr> "Yes", "Yes", "Yes", "Yes", "Yes", "Yes", "...

Next, we just count our high usage flags and turn the count to percent. We can see that 28.2% of functions create 80% of DRob’s code.

ve\_eighty\_twenty\_tbl %>%

count(high\_usage) %>%

count\_to\_pct(col = nn) %>%

knitr::kable()

| high\_usage | nn | pct |
| --- | --- | --- |
| No | 224 | 0.7179487 |
| Yes | 88 | 0.2820513 |

Finally, here are the functions by package that we should focus on if we are just starting out. Keep in mind this is just DRob and we may want to expand to other masters of data science to get an even better picture of the high usage functions.

ve\_eighty\_twenty\_tbl %>%

filter(high\_usage == "Yes") %>%

split(.$package)

## $base

## # A tibble: 23 x 6

## package function\_name n pct pct\_cum high\_usage

## <chr> <chr> <int> <dbl> <dbl> <chr>

## 1 base library 171 0.0739 0.0739 Yes

## 2 base c 61 0.0264 0.322 Yes

## 3 base sum 59 0.0255 0.347 Yes

## 4 base mean 38 0.0164 0.382 Yes

## 5 base function 23 0.00994 0.519 Yes

## 6 base list 18 0.00778 0.537 Yes

## 7 base seq 17 0.00735 0.552 Yes

## 8 base set.seed 14 0.00605 0.585 Yes

## 9 base seq\_len 12 0.00519 0.619 Yes

## 10 base log10 11 0.00475 0.634 Yes

## 11 base sample 11 0.00475 0.639 Yes

## 12 base cbind 10 0.00432 0.653 Yes

## 13 base log 10 0.00432 0.657 Yes

## 14 base is.na 9 0.00389 0.674 Yes

## 15 base min 9 0.00389 0.678 Yes

## 16 base cumsum 8 0.00346 0.720 Yes

## 17 base paste0 8 0.00346 0.724 Yes

## 18 base matrix 7 0.00303 0.741 Yes

## 19 base colSums 6 0.00259 0.768 Yes

## 20 base max 6 0.00259 0.770 Yes

## 21 base as.Date 5 0.00216 0.788 Yes

## 22 base replicate 5 0.00216 0.790 Yes

## 23 base t 5 0.00216 0.792 Yes

##

## $broom

## # A tibble: 2 x 6

## package function\_name n pct pct\_cum high\_usage

## <chr> <chr> <int> <dbl> <dbl> <chr>

## 1 broom tidy 24 0.0104 0.509 Yes

## 2 broom augment 7 0.00303 0.744 Yes

##

## $dplyr

## # A tibble: 20 x 6

## package function\_name n pct pct\_cum high\_usage

## <chr> <chr> <int> <dbl> <dbl> <chr>

## 1 dplyr filter 123 0.0532 0.127 Yes

## 2 dplyr mutate 122 0.0527 0.233 Yes

## 3 dplyr group\_by 62 0.0268 0.296 Yes

## 4 dplyr summarize 41 0.0177 0.365 Yes

## 5 dplyr select 35 0.0151 0.397 Yes

## 6 dplyr count 31 0.0134 0.439 Yes

## 7 dplyr inner\_join 28 0.0121 0.477 Yes

## 8 dplyr arrange 25 0.0108 0.488 Yes

## 9 dplyr ungroup 25 0.0108 0.499 Yes

## 10 dplyr n 23 0.00994 0.529 Yes

## 11 dplyr desc 15 0.00648 0.573 Yes

## 12 dplyr tbl\_df 14 0.00605 0.591 Yes

## 13 dplyr funs 11 0.00475 0.644 Yes

## 14 dplyr anti\_join 9 0.00389 0.682 Yes

## 15 dplyr mutate\_each 8 0.00346 0.727 Yes

## 16 dplyr rename 8 0.00346 0.731 Yes

## 17 dplyr top\_n 8 0.00346 0.734 Yes

## 18 dplyr data\_frame 7 0.00303 0.747 Yes

## 19 dplyr distinct 7 0.00303 0.750 Yes

## 20 dplyr do 6 0.00259 0.773 Yes

##

## $ggplot2

## # A tibble: 23 x 6

## package function\_name n pct pct\_cum high\_usage

## <chr> <chr> <int> <dbl> <dbl> <chr>

## 1 ggplot2 aes 123 0.0532 0.180 Yes

## 2 ggplot2 ggplot 83 0.0359 0.269 Yes

## 3 ggplot2 labs 35 0.0151 0.412 Yes

## 4 ggplot2 geom\_line 32 0.0138 0.426 Yes

## 5 ggplot2 facet\_wrap 30 0.0130 0.452 Yes

## 6 ggplot2 geom\_point 30 0.0130 0.465 Yes

## 7 ggplot2 geom\_histogram 14 0.00605 0.597 Yes

## 8 ggplot2 geom\_smooth 13 0.00562 0.603 Yes

## 9 ggplot2 scale\_y\_continuous 13 0.00562 0.608 Yes

## 10 ggplot2 theme 12 0.00519 0.624 Yes

## 11 ggplot2 theme\_set 10 0.00432 0.662 Yes

## 12 ggplot2 ylab 10 0.00432 0.666 Yes

## 13 ggplot2 element\_text 9 0.00389 0.686 Yes

## 14 ggplot2 geom\_bar 9 0.00389 0.690 Yes

## 15 ggplot2 geom\_text 9 0.00389 0.694 Yes

## 16 ggplot2 scale\_x\_log10 9 0.00389 0.697 Yes

## 17 ggplot2 theme\_bw 9 0.00389 0.701 Yes

## 18 ggplot2 geom\_tile 7 0.00303 0.753 Yes

## 19 ggplot2 geom\_abline 6 0.00259 0.775 Yes

## 20 ggplot2 geom\_vline 6 0.00259 0.778 Yes

## 21 ggplot2 geom\_boxplot 5 0.00216 0.794 Yes

## 22 ggplot2 geom\_hline 5 0.00216 0.796 Yes

## 23 ggplot2 theme\_void 5 0.00216 0.799 Yes

##

## $lubridate

## # A tibble: 1 x 6

## package function\_name n pct pct\_cum high\_usage

## <chr> <chr> <int> <dbl> <dbl> <chr>

## 1 lubridate round\_date 7 0.00303 0.756 Yes

##

## $purrr

## # A tibble: 1 x 6

## package function\_name n pct pct\_cum high\_usage

## <chr> <chr> <int> <dbl> <dbl> <chr>

## 1 purrr map 9 0.00389 0.705 Yes

##

## $stats

## # A tibble: 5 x 6

## package function\_name n pct pct\_cum high\_usage

## <chr> <chr> <int> <dbl> <dbl> <chr>

## 1 stats reorder 16 0.00691 0.566 Yes

## 2 stats qbeta 12 0.00519 0.630 Yes

## 3 stats lm 10 0.00432 0.670 Yes

## 4 stats dbeta 7 0.00303 0.759 Yes

## 5 stats rbeta 6 0.00259 0.780 Yes

##

## $stringr

## # A tibble: 1 x 6

## package function\_name n pct pct\_cum high\_usage

## <chr> <chr> <int> <dbl> <dbl> <chr>

## 1 stringr str\_detect 17 0.00735 0.559 Yes

##

## $tibble

## # A tibble: 1 x 6

## package function\_name n pct pct\_cum high\_usage

## <chr> <chr> <int> <dbl> <dbl> <chr>

## 1 tibble data\_frame 7 0.00303 0.762 Yes

##

## $tidyr

## # A tibble: 5 x 6

## package function\_name n pct pct\_cum high\_usage

## <chr> <chr> <int> <dbl> <dbl> <chr>

## 1 tidyr separate 18 0.00778 0.545 Yes

## 2 tidyr gather 15 0.00648 0.579 Yes

## 3 tidyr crossing 13 0.00562 0.614 Yes

## 4 tidyr unite 9 0.00389 0.709 Yes

## 5 tidyr unnest 9 0.00389 0.713 Yes

##

## $Unknown

## # A tibble: 5 x 6

## package function\_name n pct pct\_cum high\_usage

## <chr> <chr> <int> <dbl> <dbl> <chr>

## 1 Unknown percent\_format 9 0.00389 0.717 Yes

## 2 Unknown ebb\_fit\_prior 8 0.00346 0.738 Yes

## 3 Unknown unnest\_tokens 7 0.00303 0.765 Yes

## 4 Unknown dbetabinom.ab 6 0.00259 0.783 Yes

## 5 Unknown mcbind 6 0.00259 0.786 Yes

##

## $utils

## # A tibble: 1 x 6

## package function\_name n pct pct\_cum high\_usage

## <chr> <chr> <int> <dbl> <dbl> <chr>

## 1 utils head 11 0.00475 0.649 Yes

### Takeaways From Code

1. We are using quite a bit of base, dplyr, and ggplot2 code. **In fact, these three libraries account for 77.6% of his code on Variance Explained.**
2. DRob’s code is getting… tidier! **DRob is using approximately 80% tidyverse code in the most recent half-year of blogging.** This trend is increasing, although it will eventually top out. This compares to around 37% tidy code when he began blogging in 2015.
3. If DRob is getting tidier, which area is getting impacted the most? It’s the packages I’ve categorized as “Unknown”. These are non-tidyverse or pre-loaded packages. In other words, these are uncommonly used packages that may serve a specialized need. I do not currently have these loaded, which is why they are considered “Unknown”. It’s worth mentioning that base and stats libraries are declining slightly, but not to the extent that specialized packages are declining. **The bottom line - DRob is using less specialized packages and more tidyverse.**

### Analysis Risks

One point I have not discussed is that DRob is just one really good data scientist. His code is clearly representative of the tidyverse-style, which resonates with many future data scientists coming into the industry. If one wishes to emulate DRob, this is probably a good analysis to take and run with. However, it may make sense to also view other “masters” that exist as part of a future endeavor.

Another point is that we got 2,314 functions out of 58 posts. While this is by no means a small sample, we certainly may wish to increase the sample size to get more confidence in the most high usage functions. Personally, I’d like to see a 100X ratio between top functions and total observations, meaning the top 100 functions would be from at a minimum 10,000 functions. With that said, the analysis was performed accross a large sample of projects (58 posts less those that do not contain code) and multiple years which is another factor that improves confidence.

## Bonus: How to Learn R by Analyzing Your Code

We spoke a lot about analyzing DRob’s code, but with a few modifications you can apply this analysis to your own code stored in .R or .Rmd files! Here’s how with the fs package.

We’ll begin with a relatively large code base from a project I’m working on, which is a new course called **HR 201: Predicting Employee Attrition**.!

### Part 1: Extracting Your Functions From Your Code Base

First, load the fs package. This is a great package for working with the file system on you computer.

**library**(fs)

Next, collect the path for YOUR code base directory. I will use my R Project directory for the HR 201 Course.

dir\_path <- "../../../Business Science/Courses/Teachable/HR201\_Employee\_Turnover\_H2O/HR201\_Employee\_Turnover\_Project/"

Use a function called dir\_info() to retrieve the contents of the directory. Add the argument, recursive = TRUE, to collect all the files from the sub-directories. Use head() to return the first six rows only.

dir\_info(dir\_path, recursive = TRUE) %>%

head() %>%

knitr::kable()

Now that we see how dir\_info() works, we can use one more function called path\_file() to retrieve just the file portion of the path. We can then use the file name with str\_detect() to detect only files with “.R” or “.Rmd” at the end. We’ll create a tibble of the file names and paths.

rmd\_or\_r\_file\_paths\_tbl <- dir\_info(dir\_path, recursive = T) %>%

mutate(file\_name = path\_file(path)) %>%

select(file\_name, path) %>%

filter(str\_detect(file\_name, "(\\.R|\\.Rmd)$"))

rmd\_or\_r\_file\_paths\_tbl %>% knitr::kable()

Next, we can create a custom function called, build\_function\_names\_tbl\_from\_file\_path(), which is very similar function to the url builder before. The main difference is that the HTML extraction code is replaced with readLines().

build\_function\_names\_tbl\_from\_file\_path <- **function**(path, loaded\_functions\_tbl) {

builder <- **function**(path, loaded\_functions\_tbl) {

readLines(path) %>%

parse\_function\_names() %>%

left\_join(loaded\_functions\_tbl) %>%

filter(

!(function\_name == "filter" & !(package == "dplyr"))

) %>%

mutate(package = ifelse(is.na(package), "Unknown", package))

}

safe\_builder <- possibly(builder, otherwise = NA)

safe\_builder(path, loaded\_functions\_tbl)

}

We can test it with one of the file paths.

file\_path\_1 <- rmd\_or\_r\_file\_paths\_tbl$path[[1]]

file\_path\_1

## [1] "../../../Business Science/Courses/Teachable/HR201\_Employee\_Turnover\_H2O/HR201\_Employee\_Turnover\_Project/00\_Scripts/assess\_attrition.R"

Let’s see what it returns.

build\_function\_names\_tbl\_from\_file\_path(file\_path\_1, loaded\_functions\_tbl) %>%

glimpse()

## Observations: 57

## Variables: 2

## $ function\_name <chr> "library", "count", "function", "quos", "en...

## $ package <chr> "base", "dplyr", "base", "dplyr", "dplyr", ...

Great, it works identically to the web scraping version but with local file paths. We have 57 functions just in the first file.

We can scale it to all code in the code base using the file paths. The process is almost identical to the web scraping process.

local\_function\_names\_tbl <- rmd\_or\_r\_file\_paths\_tbl %>%

mutate(

function\_name = map(path, build\_function\_names\_tbl\_from\_file\_path, loaded\_functions\_tbl),

is\_logical = map\_dbl(function\_name, is.logical)

) %>%

filter(is\_logical != 1) %>%

select(file\_name, function\_name) %>%

unnest() %>%

left\_join(loaded\_functions\_tbl)

local\_function\_names\_tbl %>% glimpse()

## Observations: 1,422

## Variables: 3

## $ file\_name <fs::path> "assess\_attrition.R", "assess\_attritio...

## $ function\_name <chr> "library", "count", "function", "quos", "en...

## $ package <chr> "base", "dplyr", "base", "dplyr", "dplyr", ...

### Part 2: Analyzing Your Code

You can run through the same process with your code. Here are my top 20 functions.

local\_functions\_top\_20\_tbl <- local\_function\_names\_tbl %>%

count(package, function\_name) %>%

count\_to\_pct() %>%

arrange(desc(n)) %>%

top\_n(20) %>%

rowid\_to\_column(var = "rank")

local\_functions\_top\_20\_tbl %>%

ggplot(aes(x = n, y = fct\_reorder(function\_name, n), color = package)) +

geom\_segment(aes(xend = 0, yend = function\_name), size = 2) +

geom\_point(size = 4) +

geom\_label(aes(label = paste0(function\_name, "(), ", package, ", ", scales::percent(pct))),

hjust = "inward", size = 3.5) +

expand\_limits(x = 0) +

labs(

title = "Which Functions Are Most Frequently Used by DRob?",

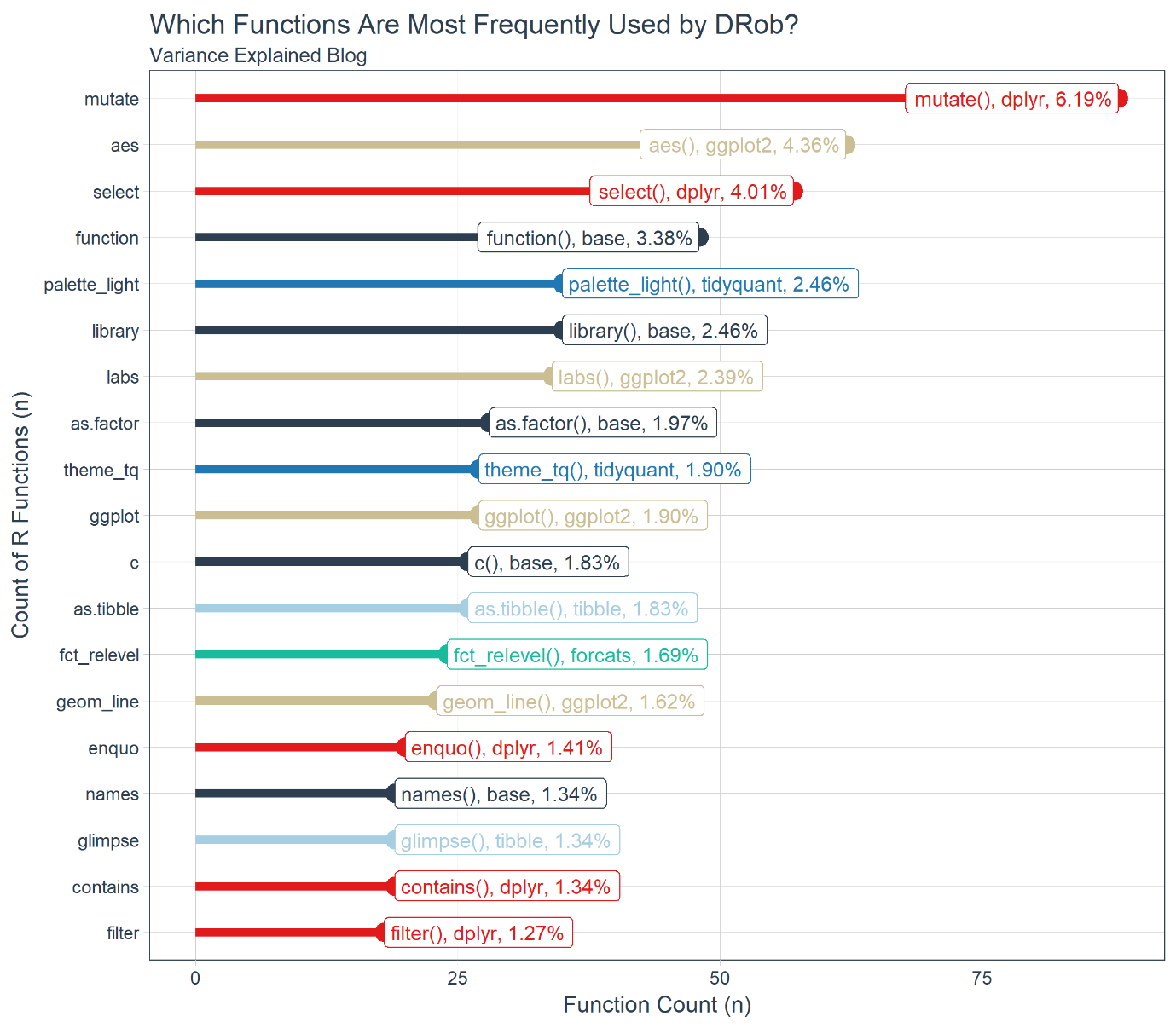
subtitle = "Variance Explained Blog",

x = "Function Count (n)", y = "Count of R Functions (n)") +

scale\_color\_tq() +

theme\_tq() +

theme(legend.position = "none")



#### Similarities

You can assess the similarities and differences between you and DRob. For example, I both use quite a bit of dplyr for data manipulation and ggplot2 for visualization.

local\_functions\_top\_20\_tbl %>%

filter(function\_name %**in**% ve\_functions\_top\_20\_tbl$function\_name) %>%

knitr::kable()

| rank | package | function\_name | n | pct |
| --- | --- | --- | --- | --- |
| 1 | dplyr | mutate | 88 | 0.0618847 |
| 2 | ggplot2 | aes | 62 | 0.0436006 |
| 3 | dplyr | select | 57 | 0.0400844 |
| 5 | base | library | 35 | 0.0246132 |
| 7 | ggplot2 | labs | 34 | 0.0239100 |
| 9 | ggplot2 | ggplot | 27 | 0.0189873 |
| 11 | base | c | 26 | 0.0182841 |
| 14 | ggplot2 | geom\_line | 23 | 0.0161744 |
| 20 | dplyr | filter | 18 | 0.0126582 |

#### Differences

I have a few differences related to my coding techniques. I do quite a bit of programming so base::function() is in fourth place and dplyr::enquo() (part of the new tidy eval framework) is in 15th place. I also have tidyquant::palette\_light() and tidyquant::theme\_tq() related to my preference for tidyquant ggplot2 themes.

local\_functions\_top\_20\_tbl %>%

filter(!function\_name %**in**% ve\_functions\_top\_20\_tbl$function\_name) %>%

knitr::kable()

| rank | package | function\_name | n | pct |
| --- | --- | --- | --- | --- |
| 4 | base | function | 48 | 0.0337553 |
| 6 | tidyquant | palette\_light | 35 | 0.0246132 |
| 8 | base | as.factor | 28 | 0.0196906 |
| 10 | tidyquant | theme\_tq | 27 | 0.0189873 |
| 12 | tibble | as.tibble | 26 | 0.0182841 |
| 13 | forcats | fct\_relevel | 24 | 0.0168776 |
| 15 | dplyr | enquo | 20 | 0.0140647 |
| 16 | base | names | 19 | 0.0133615 |
| 17 | dplyr | contains | 19 | 0.0133615 |
| 18 | dplyr | glimpse | 19 | 0.0133615 |
| 19 | tibble | glimpse | 19 | 0.0133615 |

And, we can also see how DRob’s top 20 differs from mine. Most of these functions are ones I use frequently, just not in my top 20. And, this is likely the case for DRob with the dissimilar functions in the table above.

ve\_functions\_top\_20\_tbl %>%

rowid\_to\_column(var = "rank") %>%

filter(!function\_name %**in**% local\_functions\_top\_20\_tbl$function\_name) %>%

knitr::kable()

| rank | package | function\_name | n | pct |
| --- | --- | --- | --- | --- |
| 6 | dplyr | group\_by | 62 | 0.0267934 |
| 8 | base | sum | 59 | 0.0254970 |
| 9 | dplyr | summarize | 41 | 0.0177182 |
| 10 | base | mean | 38 | 0.0164218 |
| 14 | dplyr | count | 31 | 0.0133967 |
| 15 | ggplot2 | geom\_point | 30 | 0.0129646 |
| 16 | ggplot2 | facet\_wrap | 30 | 0.0129646 |
| 17 | dplyr | inner\_join | 28 | 0.0121003 |
| 18 | dplyr | ungroup | 25 | 0.0108038 |
| 19 | dplyr | arrange | 25 | 0.0108038 |
| 20 | broom | tidy | 24 | 0.0103717 |

**Bonus: Analyze Your Code**

We show as a bonus how you can apply the custom scripts developed in this article to do an analysis on your code with the new fs package (short for file system). We briefly analyze our code base for the HR 201: Predicting Employee

**Our Hypothesis**

Learning R can be unnecessarily challenging if one focuses on learning everything immediately rather than applying a strategic approach. Our hypothesis is composed of two theories:

1. You do not need to learn everything to become effective
2. You can learn a lot by analyzing others’ code bases

Our quest is to prove these theories.

**80/20 Rule**

Trying to solve every aspect of a challenge is overwhelming and often not the best use of your time. The 80/20 Rule can often help in these situations. Generally speaking, the 80/20 Rule posits that roughly 20% of activities or tasks produce 80% of the results you want. The challenge of learning the R language falls perfectly into this model. The key is figuring out which areas are the highest value for your time.

**Learning From A Master Data Scientist**

To make an optimal strategy, we need data on what tools are being used to perform high-end data science. But, where can we get data? From a data scientist that regularly performs high-end data science publicly via the internet. A true master at the dark art of data science!

**Why we chose David Robinson (aka DRob):**

I met DRob at the EARL Boston conference last November. He had just given a stellar presentation on StackOverflow trends concluding that R was growing rapidly. What stuck out about his presentation was his ability to analyze a question or problem – This is a key skill in DS4B!

Many data scientists such as top Kaggle competitors focus on how to create a high end predictions (which is important) but it’s not representative of most real-world situations. It’s really unique to see a data scientist effectively using problem solving and critical thinking in combination with data science to learn about the problem. DRob did this very well.

I had checked out his blog before, but never picked up on the fact that he’s really doing data science that can be applied to business (although he’s mainly applying to other areas such as sports and politics). Some of my favorites are articles on statistics applications in baseball. His approaches are novel leaning on Bayesian A/B testing, hierarchical modeling, mixture models, and many other tools that are very useful in business analysis. Further, he employs problem solving and critical thinking, which are the same skills that are needed in DS4B.

**Analysis: Learning From A Master**

If we treat DRob’s code on his Variance Explained blog as a text analysis, **we can find the most frequently used packages and functions that cover the majority of code he produces**. We can then use an 80/20 approach to determine which functions and packages are most used and therefore most important to master. We’ll split this analysis into two parts:

* Part 1: Web Scraping The Variance Explained Blog – Note this is a technical section showing how we retrieved the data. Novice learners may wish to skip this part.
* Part 2: Learning From DRob’s Code – The analytical work is done here!

**Libraries**

If you wish to follow along, please load the following libraries.

library(tidyverse)

library(tidyquant)

library(tibbletime)

library(rvest)

library(broom)

**Part 1: Web Scraping the Variance Explained Blog**

This part of the analysis is **a how-to in web scraping**. We expose the process to collect the data, which use the rvest package and a number of custom functions to parse the text. We split this part into two steps:

* Step 1: Setup Code Parsing Functions
* Step 2: Web Scraping Variance Explained

**Step 1: Setup Code Parsing and Utility Functions**

This is a text analysis. As such, we are going to need to parse some text to extract function names, to determine which packages functions belong to, and to analyze the text with counts and percents. To do so, we create a few helper functions:

* count\_to\_pct(): Utility function to quickly convert counts to percentages. Works well with dplyr::count().
* parse\_function\_names(): Takes in text and returns function names.
* find\_functions\_in\_package(): Takes in a library or package name and returns all functions in the library or package.
* find\_loaded\_packages(): Detects which packages are loaded in the R system.
* map\_loaded\_package\_functions(): Maps the function names to the respective package.

**Count To Percent**

*Function*: Utility function to quickly convert counts to percentages. Works well with dplyr::count().

count\_to\_pct <- function(data, ..., col = n) {

grouping\_vars\_expr <- quos(...)

col\_expr <- enquo(col)

data %>%

group\_by(!!! grouping\_vars\_expr) %>%

mutate(pct = (!! col\_expr) / sum(!! col\_expr)) %>%

ungroup()

}

*Usage*: Use dplyr::count() to retrieve counts by grouping variables. Then use count\_to\_pct() to quickly get percentages within groups. Exclude groups if overall percentages are desired. Note this example uses the mpg data set from ggplot2 package.

mpg %>%

count(manufacturer) %>%

count\_to\_pct() %>%

top\_n(5) %>%

arrange(desc(n))

## # A tibble: 5 x 3

## manufacturer n pct

##

## 1 dodge 37 0.158

## 2 toyota 34 0.145

## 3 volkswagen 27 0.115

## 4 ford 25 0.107

## 5 chevrolet 19 0.0812

**Parse Function Names**

*Function*: Takes in text and returns function names.

parse\_function\_names <- function(text, stop\_words = c("")) {

parser <- function(text, stop\_words) {

ret <- text %>%

str\_c(collapse = " ") %>%

str\_split("\\(") %>%

set\_names("text") %>%

as.tibble() %>%

slice(-n()) %>%

mutate(str\_split = map(text, str\_split, " ")) %>%

select(-text) %>%

unnest() %>%

mutate(function\_name = map\_chr(str\_split, ~ purrr::pluck(last(.x)))) %>%

select(function\_name) %>%

separate(function\_name, into = c("discard", "function\_name"),

sep = "(:::|::|\n)", fill = "left") %>%

select(-discard) %>%

mutate(function\_name = str\_replace\_all(function\_name,

pattern = "[^[:alnum:]\_\\.]", "")) %>%

filter(!(function\_name %in% stop\_words))

return(ret)

}

safe\_parser <- possibly(parser, otherwise = NA)

safe\_parser(text, stop\_words)

}

*Usage*: Parses the function name preceding “(“. Returns a tibble.

test\_text <- "my\_mean <- mean(1:10) some text base::sum(my\_mean) "

parse\_function\_names(test\_text)

## # A tibble: 2 x 1

## function\_name

##

## 1 mean

## 2 sum

**Find Functions In Package**

*Function*: Takes in a library or package name and returns all functions in the library or package.

find\_functions\_in\_package <- function(package) {

pkg\_text <- paste0("package:", package)

safe\_ls <- possibly(ls, otherwise = NA)

package\_functions <- safe\_ls(pkg\_text)

if (is.na(package\_functions[[1]])) return(package\_functions)

ret <- package\_functions %>%

as.tibble() %>%

rename(function\_name = value)

return(ret)

}

*Usage*: Takes in a package that is loaded via library(package\_name) (the package must be loaded for this to work properly). Returns a tibble of all functions in a package.

find\_functions\_in\_package("dplyr") %>% glimpse()

## Observations: 237

## Variables: 1

## $ function\_name "%>%", "add\_count", "add\_count\_", "add\_row"...

**Find Loaded Packages**

*Function*: Detects which packages are loaded in the R system.

find\_loaded\_packages <- function() {

ret <- search() %>%

list() %>%

set\_names("search") %>%

as.tibble() %>%

separate(search, into = c("discard", "keep"), sep = ":", fill = "right") %>%

select(keep) %>%

filter(!is.na(keep)) %>%

rename(package = keep) %>%

arrange(package)

return(ret)

}

*Usage*: Returns a tibble of loaded packages. Used in conjunction with map\_loaded\_package\_functions() to build a corpus of functions and associated packages, which is needed to determine which package the target function comes from.

find\_loaded\_packages() %>% glimpse()

## Observations: 28

## Variables: 1

## $ package "base", "bindrcpp", "broom", "datasets", "dplyr",...

**Map Loaded Package Functions**

*Function*: Maps the function names to the respective package.

map\_loaded\_package\_functions <- function(data, col) {

col\_expr <- enquo(col)

data %>%

mutate(function\_name = map(!! col\_expr, find\_functions\_in\_package)) %>%

mutate(is\_logical = map\_dbl(function\_name, is.logical)) %>%

filter(is\_logical != 1) %>%

select(-is\_logical) %>%

unnest()

}

*Usage*: Used in conjunction with find\_loaded\_packages(). Builds a corpus of package and function combinations for every package that is loaded. Returns a tibble of packages and associated functions.

find\_loaded\_packages() %>%

map\_loaded\_package\_functions(package) %>%

glimpse()

## Observations: 4,518

## Variables: 2

## $ package "base", "base", "base", "base", "base", "ba...

## $ function\_name "-", "-.Date", "-.POSIXt", "!", "!.hexmode"...

**Step 2: Web Scraping Variance Explained**

The first action is to setup our process for pulling the functions and packages for **a single post**. Once that process is defined, we can scale our web scraping to **ALL posts**!

**Web Scrape A Single Blog Post**

DRob has a number of posts that include code-throughs (walkthroughs using code). The code is contained within the HTML as a node named . This makes it very easy to extract using the rvest package.



Mixure Model Baseball

library(dplyr)

library(tidyr)

library(Lahman)

library(ggplot2)

theme\_set(theme\_bw())

*# Identify those who have pitched at least three games*

pitchers **<-** Pitching **%>%**

group\_by(playerID) **%>%**

summarize(gamesPitched **=** **sum**(G)) **%>%**

filter(gamesPitched **>** 3)

*# in this setup, we're keeping some extra information for later in the post:*

*# a "bats" column and a "year" column*

career **<-** Batting **%>%**

filter(AB **>** 0, lgID **==** "NL", yearID **>=** 1980) **%>%**

group\_by(playerID) **%>%**

summarize(H **=** **sum**(H), AB **=** **sum**(AB), year **=** mean(yearID)) **%>%**

mutate(average **=** H **/** AB,

isPitcher **=** playerID **%in%** pitchers**$**playerID)

*# Add player names*

career **<-** Master **%>%**

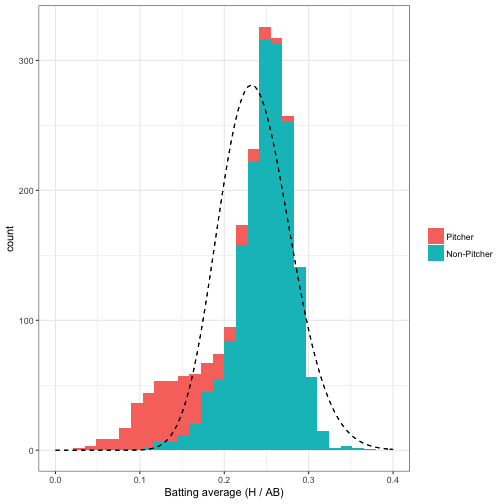
tbl\_df() **%>%**

dplyr**::**select(playerID, nameFirst, nameLast, bats) **%>%**

unite(name, nameFirst, nameLast, sep **=** " ") **%>%**

inner\_join(career, by **=** "playerID")

We’ve been filtering out pitchers in the previous posts, which make batting averages look roughly like a beta distribution. But when we leave them in, as I showed above, the data looks a lot less like a beta:



The dashed density curve represents the beta distribution we would naively fit to this data. We can see that unlike our earlier analysis, where we’d filtered out pitchers, the beta is not a good fit- but that it’s plausible that we could fit the data using two beta distributions, one for pitchers and one for non-pitchers.

In this example, we know which players are pitchers and which aren’t. But if we didn’t, we would need to assign each player to a distribution, or “cluster”, before performing shrinkage on it. In a real analysis it’s not realistic that we wouldn’t know which players are pitchers, but it’s an excellent illustrative example of a mixture model and of expectation-maximization algorithms.

### Expectation-maximization

The challenge of mixture models is that at the start, we don’t know which observations belong to which cluster, nor what the parameters of each distribution is. It’s difficult to solve these problems at the same time- so an expectation-maximization (EM) algorithm takes the jump of estimating them one at a time, and **alternating** between them.

The first thing to do in an EM clustering algorithm is to assign our clusters **randomly**:

set.seed(2016)

*# We'll fit the clusters only with players that have had at least 20 at-bats*

starting\_data **<-** career **%>%**

filter(AB **>=** 20) **%>%**

select(**-**year, **-**bats, **-**isPitcher) **%>%**

mutate(cluster **=** factor(sample(**c**("A", "B"), n(), replace **=** **TRUE**)))

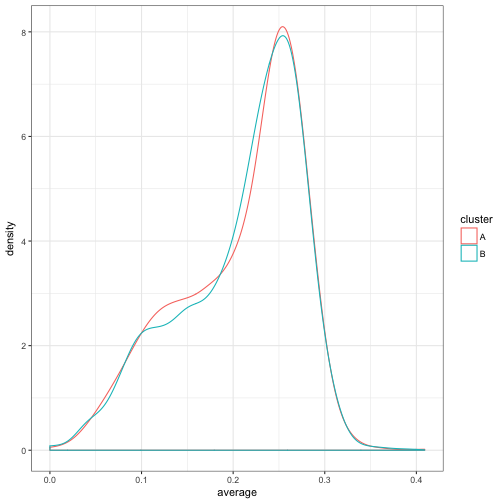
#### Maximization

Now that we’ve got cluster assignments, what do the densities of each cluster look like?

starting\_data **%>%**

ggplot(aes(average, color **=** cluster)) **+**

geom\_density()



Well, that doesn’t look like much of a division- they have basically the same density! That’s OK: one of the nice features of expectation-maximization is that we don’t actually have to start with good clusters to end up with a good result.

We’ll now write a function for fitting a beta-binomial distribution using maximum likelihood estimation (and the dbetabinom.ab function from the VGAM package). This is a process we’ve done in multiple posts before, including the appendix of one of the first ones. We’re just encapsulating it into a function.

library(VGAM)

fit\_bb\_mle **<-** **function**(x, n) {

*# dbetabinom.ab is the likelihood function for a beta-binomial*

*# using n, alpha and beta as parameters*

ll **<-** **function**(alpha, beta) {

**-sum**(dbetabinom.ab(x, n, alpha, beta, log **=** **TRUE**))

}

m **<-** stats4**::**mle(ll, start **=** **list**(alpha **=** 3, beta **=** 10), method **=** "L-BFGS-B",

lower **=** **c**(0.001, .001))

ab **<-** stats4**::**coef(m)

data\_frame(alpha **=** ab[1], beta **=** ab[2], number **=** **length**(x))

}

(The number column I added will be useful in the next step). For example, here are the alpha and beta chosen for the entire data as a whole:

fit\_bb\_mle(starting\_data**$**H, starting\_data**$**AB)

## # A tibble: 1 × 3

## alpha beta number

## <dbl> <dbl> <int>

## 1 12.8 45.5 3209

But now we’re working with a mixture model. This time, we’re going to fit the model within each of our (randomly assigned) clusters:

fits **<-** starting\_data **%>%**

group\_by(cluster) **%>%**

do(fit\_bb\_mle(.**$**H, .**$**AB)) **%>%**

ungroup()

fits

## # A tibble: 2 × 4

## cluster alpha beta number

## <fctr> <dbl> <dbl> <int>

## 1 A 12.1 43.3 1559

## 2 B 13.6 47.9 1650

Another component of this model is the prior probability that a player is in cluster A or cluster B, which we set to 50-50 when we were assigning random clusters. We can estimate our new iteration of this based on the total number of assignments in each group, which is why we included the number column:

fits **<-** fits **%>%**

mutate(prior **=** number **/** **sum**(number))

fits

## # A tibble: 2 × 5

## cluster alpha beta number prior

## <fctr> <dbl> <dbl> <int> <dbl>

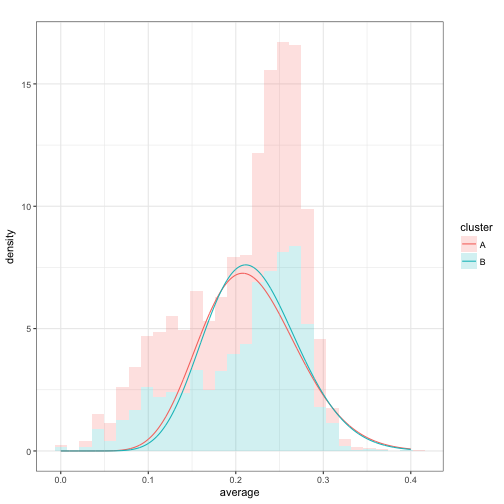
## 1 A 12.1 43.3 1559 0.486

## 2 B 13.6 47.9 1650 0.514

Much as the within-cluster densities only changed a little, the priors only changed a little as well. This was the maximization step: find the maximum likelihood parameters (in this case, two alpha/beta values, and a per-cluster probability), pretending we knew the assignments.

### Expectation

We now have a distribution for each cluster. It’s worth noting that these are pretty similar distributions, and that neither is a good fit to the data.



However, notice that due to a small random difference, cluster B is **slightly** more likely than cluster A for batting averages above about .2, and vice versa below .2.

Consider therefore that each player has a likelihood it would have been generated from cluster A, and a likelihood it would have been generated from cluster B (being sure to weight each by the prior probability of being in A or B):

crosses **<-** starting\_data **%>%**

select(**-**cluster) **%>%**

crossing(fits) **%>%**

mutate(likelihood **=** prior **\*** VGAM**::**dbetabinom.ab(H, AB, alpha, beta))

crosses

## # A tibble: 6,418 × 11

## playerID name H AB average cluster alpha beta

## <chr> <chr> <int> <int> <dbl> <fctr> <dbl> <dbl>

## 1 abbotje01 Jeff Abbott 11 42 0.2619 A 12.1 43.3

## 2 abbotje01 Jeff Abbott 11 42 0.2619 B 13.6 47.9

## 3 abbotji01 Jim Abbott 2 21 0.0952 A 12.1 43.3

## 4 abbotji01 Jim Abbott 2 21 0.0952 B 13.6 47.9

## 5 abbotku01 Kurt Abbott 475 1860 0.2554 A 12.1 43.3

## 6 abbotku01 Kurt Abbott 475 1860 0.2554 B 13.6 47.9

## 7 abbotky01 Kyle Abbott 3 31 0.0968 A 12.1 43.3

## 8 abbotky01 Kyle Abbott 3 31 0.0968 B 13.6 47.9

## 9 abercre01 Reggie Abercrombie 86 386 0.2228 A 12.1 43.3

## 10 abercre01 Reggie Abercrombie 86 386 0.2228 B 13.6 47.9

## # ... with 6,408 more rows, and 3 more variables: number <int>,

## # prior <dbl>, likelihood <dbl>

For example, consider Jeff Abbott, who got 11 hits out of 42 at-bats. He had a 4.35% chance of getting that if he were in cluster A, but a 4.76% chance if he were in cluster B. For that reason (even though it’s a small difference), we’ll put him in B. Similarly we’ll put Kyle Abbott in cluster A: 3/31 was more likely to come from that distribution.

We can do that for every player using group\_by and top\_n:

assignments **<-** starting\_data **%>%**

select(**-**cluster) **%>%**

crossing(fits) **%>%**

mutate(likelihood **=** prior **\*** VGAM**::**dbetabinom.ab(H, AB, alpha, beta)) **%>%**

group\_by(playerID) **%>%**

top\_n(1, likelihood) **%>%**

ungroup()

assignments

## # A tibble: 3,209 × 11

## playerID name H AB average cluster alpha beta

## <chr> <chr> <int> <int> <dbl> <fctr> <dbl> <dbl>

## 1 abbotje01 Jeff Abbott 11 42 0.2619 B 13.6 47.9

## 2 abbotji01 Jim Abbott 2 21 0.0952 B 13.6 47.9

## 3 abbotku01 Kurt Abbott 475 1860 0.2554 B 13.6 47.9

## 4 abbotky01 Kyle Abbott 3 31 0.0968 A 12.1 43.3

## 5 abercre01 Reggie Abercrombie 86 386 0.2228 B 13.6 47.9

## 6 abnersh01 Shawn Abner 110 531 0.2072 B 13.6 47.9

## 7 abreubo01 Bobby Abreu 1607 5395 0.2979 B 13.6 47.9

## 8 abreuto01 Tony Abreu 129 509 0.2534 B 13.6 47.9

## 9 acevejo01 Jose Acevedo 8 101 0.0792 A 12.1 43.3

## 10 aceveju01 Juan Acevedo 6 65 0.0923 A 12.1 43.3

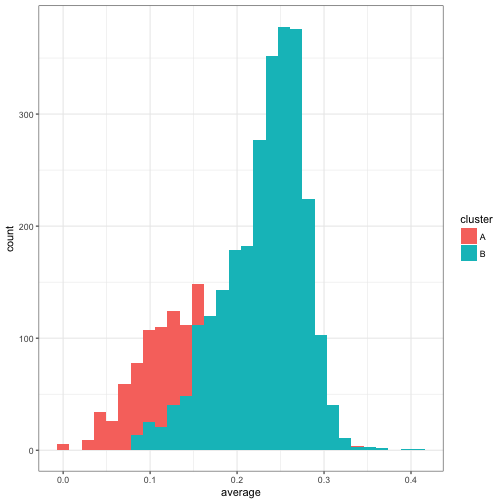
## # ... with 3,199 more rows, and 3 more variables: number <int>,

## # prior <dbl>, likelihood <dbl>

That’s the expectation step: **assigning each person to the most likely cluster**. How do our assignments look after that?

ggplot(assignments, aes(average, fill **=** cluster)) **+**

geom\_histogram()

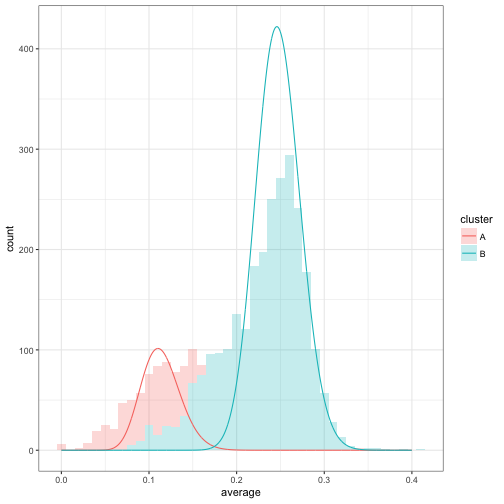


Something really important happened here: even though the two beta models we’d fit were very similar, we still split up the data rather neatly. Generally batters with a higher average ended up in cluster B, while batters with a lower average were in cluster A. (Note that due to B having a slightly higher prior probability, it was possible for players with a low average- but also a low AB- to be assigned to cluster B).

### Expectation-Maximization

The above two steps got to a better set of assignments than our original, random ones. But there’s no reason to believe these are as good as we can get. So we **repeat** the two steps, choosing new parameters for each distribution in the mixture and then making new assignments each time.

For example, now that we’ve reassigned each player’s cluster, we could re-fit the beta-binomial with the new assignments. Those distributions would look like this:



Unlike our first model fit, we can see that cluster A and cluster B have diverged a lot. Now we can take those parameters and perform a new estimation step. Generally we will do this multiple times, as an iterative process. This is the heart of an expectation-maximization algorithm, where we switch between assigning clusters (expectation) and fitting the model from those clusters (maximization).

set.seed(1337)

iterate\_em **<-** **function**(state, ...) {

fits **<-** state**$**assignments **%>%**

group\_by(cluster) **%>%**

do(mutate(fit\_bb\_mle(.**$**H, .**$**AB), number **=** nrow(.))) **%>%**

ungroup() **%>%**

mutate(prior **=** number **/** **sum**(number))

assignments **<-** assignments **%>%**

select(playerID**:**average) **%>%**

crossing(fits) **%>%**

mutate(likelihood **=** prior **\*** VGAM**::**dbetabinom.ab(H, AB, alpha, beta)) **%>%**

group\_by(playerID) **%>%**

top\_n(1, likelihood) **%>%**

ungroup()

**list**(assignments **=** assignments,

fits **=** fits)

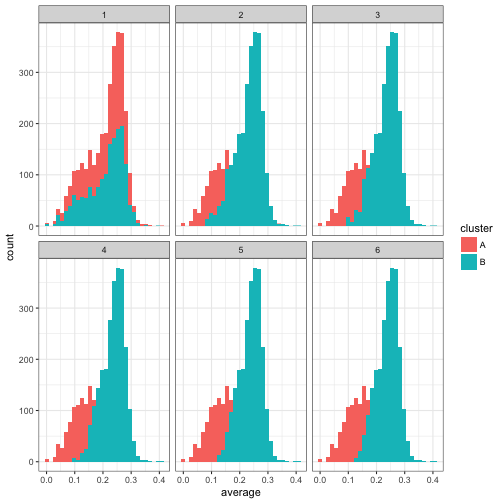
}

library(purrr)

iterations **<-** accumulate(1**:**5, iterate\_em, .init **=** **list**(assignments **=** starting\_data))

Here I used the accumulate function from the purrr package, which is useful for running data through the same function repeatedly and keeping intermediate states. I haven’t seen others use this tidy approach to EM algorithms, and there are [existing R approaches to mixture models](http://ase.tufts.edu/gsc/gradresources/guidetomixedmodelsinr/mixed%20model%20guide.html?utm_source=dlvr.it&utm_medium=twitter).[2](http://varianceexplained.org/r/mixture-models-baseball/#fn:mixture) But I like this approach both because it’s transparent about what we’re doing in each iteration, and because our iterations are now combined in a tidy format, which is convenient to summarize and visualize.

For example, how did our assignments change over the course of the iteration?



We notice that only the first few iterations led to a shift in the assignments, after which it appears to converge. Similarly, how did the estimated beta distributions change over these iterations?

fit\_iterations **<-** iterations **%>%**

map\_df("fits", .id **=** "iteration")

fit\_iterations **%>%**

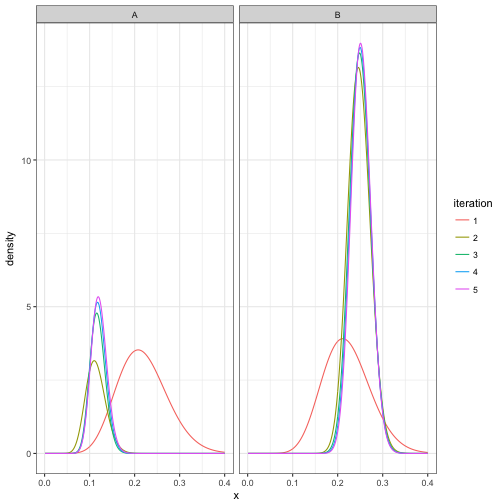
crossing(x **=** seq(.001, .4, .001)) **%>%**

mutate(density **=** prior **\*** dbeta(x, alpha, beta)) **%>%**

ggplot(aes(x, density, color **=** iteration, group **=** iteration)) **+**

geom\_line() **+**

facet\_wrap(**~** cluster)



This confirms that it took about three iterations to converge, and then stayed about the same after that. Also notice that in the process, cluster B got much more likely than cluster A, which makes sense since there are more non-pitchers than pitchers in the dataset.

### Assigning players to clusters

We now have some final parameters for each cluster:

## # A tibble: 2 × 6

## iteration cluster alpha beta number prior

## <chr> <fctr> <dbl> <dbl> <int> <dbl>

## 1 5 A 43.0 312 740 0.231

## 2 5 B 98.3 292 2469 0.769

How would we assign players to clusters, and get a posterior probability that the player belongs to that cluster? Well, let’s arbitrarily pick the six players that each batted exactly 100 times:

batter\_100 **<-** career **%>%**

filter(AB **==** 100) **%>%**

arrange(average)

batter\_100

## # A tibble: 6 × 8

## playerID name bats H AB year average isPitcher

## <chr> <chr> <fctr> <int> <int> <dbl> <dbl> <lgl>

## 1 dejesjo01 Jose de Jesus R 11 100 1990 0.11 TRUE

## 2 nicasju01 Juan Nicasio R 12 100 2012 0.12 TRUE

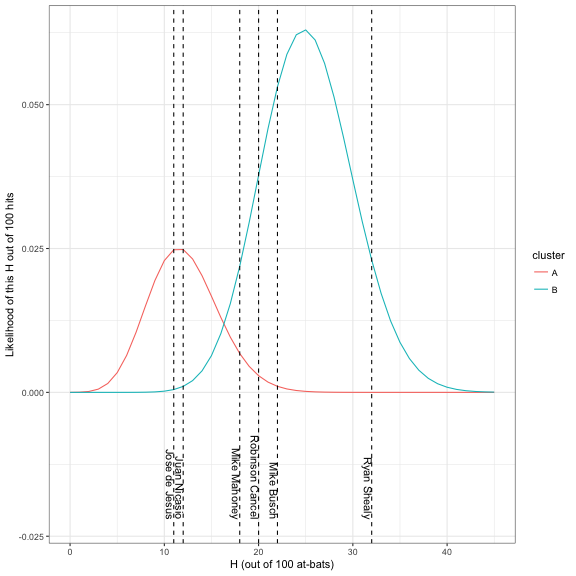
## 3 mahonmi02 Mike Mahoney R 18 100 2002 0.18 FALSE

## 4 cancero01 Robinson Cancel R 20 100 2007 0.20 FALSE

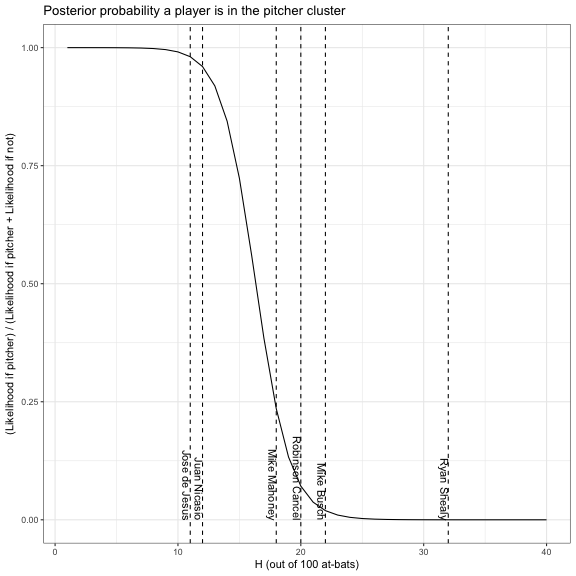
## 5 buschmi01 Mike Busch R 22 100 1996 0.22 FALSE

## 6 shealry01 Ryan Shealy R 32 100 2006 0.32 FALSE

Where would we classify each of them? Well, we’d consider the likelihood each would get the number of hits they did if they were a pitcher (cluster A) or a non-pitcher (cluster B):



By Bayes’ Theorem, we can simply use the ratio of one likelihood (say, A in red) to the sum of the two likelihoods to get the posterior probability:



Based on this, we feel confident that Juan Nicasio and Jose de Jesus are pitchers, and that the others probably aren’t. And we’d be right! (Check out the isPitcher column in the batter\_100 table above).

This allows us to assign all players in the dataset to one of the two clusters.

career\_likelihoods **<-** career **%>%**

filter(AB **>** 20) **%>%**

crossing(final\_parameters) **%>%**

mutate(likelihood **=** prior **\*** VGAM**::**dbetabinom.ab(H, AB, alpha, beta)) **%>%**

group\_by(playerID) **%>%**

mutate(posterior **=** likelihood **/** **sum**(likelihood))

career\_assignments **<-** career\_likelihoods **%>%**

top\_n(1, posterior) **%>%**

ungroup()

Since we know whether each player actually is a pitcher or not, we can also get a [confusion matrix](https://en.wikipedia.org/wiki/Confusion_matrix). How many pitchers were accidentally assigned to cluster B, and how many non-pitchers were assigned to cluster A? In this case we’ll look only at the ones for which we had at least 80% confidence in our classification.

career\_assignments **%>%**

filter(posterior **>** .8) **%>%**

count(isPitcher, cluster) **%>%**

spread(cluster, n)

## Source: local data frame [2 x 3]

## Groups: isPitcher [2]

##

## isPitcher A B

## \* <lgl> <int> <int>

## 1 FALSE 23 2026

## 2 TRUE 493 157

Not bad, considering the only information we used was the batting average- and note that we didn’t even use data on who were pitchers to train the model, but just let the clusters define themselves.

It looks like we were a lot more likely to call a pitcher a non-pitcher than vice versa. There’s a lot more we could do to examine this model, how well calibrated its posterior estimates are, and what kinds of pitchers may be mistaken for non-pitchers (e.g. good batters who pitched only a few times), but we won’t consider them in this post.

### Empirical bayes shrinkage with a mixture model

We’ve gone to all this work posterior probabilities of each player’s assignments. How can we use this in [empirical Bayes shrinkage](http://varianceexplained.org/r/empirical_bayes_baseball/), or with the other methods we’ve described in this series?

Well, consider that all of our other methods have worked because the posterior was another beta distribution (thanks to the beta being the conjugate prior of the binomial). However, now that each point might belong to one of two beta distributions, our posterior will be a mixture of betas. This mixture is made up of the posterior from each cluster, weighted by the probability the point belongs to that cluster.

For example, consider the six players who had exactly 100 at-bats. Their posterior distributions would look like this:

batting\_data **<-** career\_likelihoods **%>%**

ungroup() **%>%**

filter(AB **==** 100) **%>%**

mutate(name **=** paste0(name, " (", H, "/", AB, ")"),

name **=** reorder(name, H),

alpha1 **=** H **+** alpha,

beta1 **=** AB **-** H **+** beta)

batting\_data **%>%**

crossing(x **=** seq(0, .4, .001)) **%>%**

mutate(posterior\_density **=** posterior **\*** dbeta(x, alpha1, beta1)) **%>%**

group\_by(name, x) **%>%**

summarize(posterior\_density **=** **sum**(posterior\_density)) **%>%**

ggplot(aes(x, posterior\_density, color **=** name)) **+**

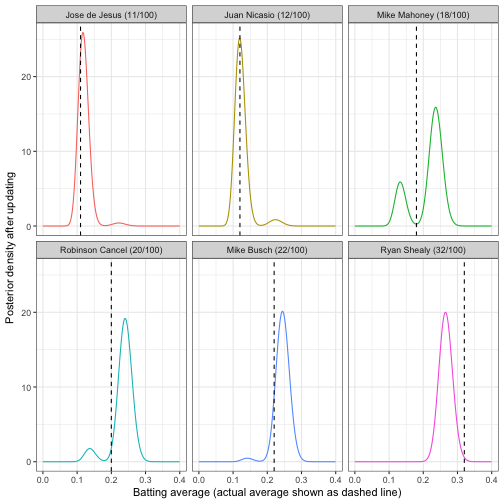
geom\_line(show.legend **=** **FALSE**) **+**

geom\_vline(aes(xintercept **=** average), data **=** batting\_data, lty **=** 2) **+**

facet\_wrap(**~** name) **+**

labs(x **=** "Batting average (actual average shown as dashed line)",

y **=** "Posterior density after updating")



For example, we are pretty sure that Jose de Jesus and Juan Nicasio are part of the “pitcher” cluster, so that makes up most of their posterior mass, and all of Ryan Shealy’s density is in the “non-pitcher” cluster. However, we’re pretty split on Mike Mahoney- he could be a pitcher who is unusually good at batting, or a non-pitcher who is unusually bad.

Can we perform shrinkage like we did in that early post? If our goal is still to find the mean of each posterior, then yes! Thanks to [linearity of expected value](https://en.wikipedia.org/wiki/Expected_value#Linearity), we can simply average the two distribution means, weighing each by the probability the player belongs to that cluster:

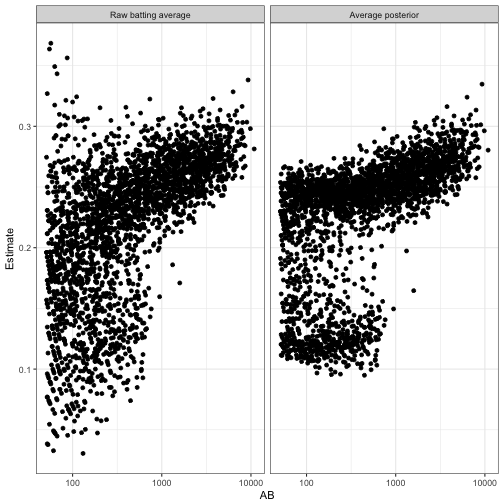
eb\_shrinkage **<-** career\_likelihoods **%>%**

mutate(shrunken\_average **=** (H **+** alpha) **/** (AB **+** alpha **+** beta)) **%>%**

group\_by(playerID) **%>%**

summarize(shrunken\_average **=** **sum**(posterior **\*** shrunken\_average))

For example, we are pretty sure that Jose de Jesus and Juan Nicasio are part of the “pitcher” cluster, which means they mostly get shrunken towards that center. We are quite certain Ryan Shealy is not a pitcher, so he’ll be updated based entirely on that distribution.



We can get the functions and packages used using the following code. It takes four steps:

1. Get a path for one of the articles. In our case, we chose DRob’s article on analyzing baseball stats with mixture models.
2. Create a corpus of all packages that are loaded in our R session. We will use this to determine which package the function that DRob uses comes from.
3. Use rvest functions read\_html() to read the HTML from the page. Then collect all nodes containing . Then extract the text within those nodes using html\_text().
4. Run the text through our custom parse\_function\_names() function. This returns parsed function names. We still need the packages, which we can get by using left\_join() with our loaded\_functions\_tbl.

The final output is all of the functions and most of the package names for the functions that are used in this article! We use the glimpse() function to keep the output minimal.

# Assign one of the blog urls to a variable called path

path <- "http://varianceexplained.org/r/mixture-models-baseball/"

# Get the loaded functions (joined in last step)

loaded\_functions\_tbl <- find\_loaded\_packages() %>%

map\_loaded\_package\_functions(package)

# Read in HTML as text for all code attributes on the page

html\_code\_text <- read\_html(path) %>%

html\_nodes("code") %>%

html\_text()

# Parse function names and join with loaded functions

# Note that stats::filter and dplyr::filter conflict

# We replace any missing packages with "Unknown"

mixture\_models\_code\_tbl <- html\_code\_text %>%

parse\_function\_names() %>%

left\_join(loaded\_functions\_tbl) %>%

filter(!(function\_name == "filter" & !(package == "dplyr"))) %>%

mutate(package = case\_when(is.na(package) ~ "Unknown", TRUE ~ package))

mixture\_models\_code\_tbl %>% glimpse()

## Observations: 131

## Variables: 2

## $ function\_name "library", "library", "library", "library",...

## $ package "base", "base", "base", "base", "ggplot2", ...

From the output, we see that DRob used 131 functions in this particular article.

Next, we can then do a quick analysis to see what functions DRob used most frequently in this article. We can see that dplyr comes up quite frequently. The most popular function used is mutate(), which is used in this article about 11% of the time.

mixture\_models\_code\_tbl %>%

count(package, function\_name) %>%

count\_to\_pct() %>%

arrange(desc(n)) %>%

top\_n(5) %>%

knitr::kable()

| **package** | **function\_name** | **n** | **pct** |
| --- | --- | --- | --- |
| dplyr | mutate | 14 | 0.1068702 |
| base | sum | 9 | 0.0687023 |
| dplyr | group\_by | 9 | 0.0687023 |
| dplyr | filter | 7 | 0.0534351 |
| dplyr | ungroup | 6 | 0.0458015 |
| tidyr | crossing | 6 | 0.0458015 |

This is just one sample. We need more data to increase our confidence in which packages and functions are important.

**Web Scrape All Blog Posts**

Scaling to all blog posts is fairly easy with the purrr package. We need to do two things:

1. Web scrape all of the titles, dates, and paths for each of DRob’s articles using rvest.
2. Scale the analysis using purrr in combination with a custom function, build\_function\_names\_tbl\_from\_url\_path(), that we will create.

Web scraping the titles, dates, and paths (href) are again easy with the rvest package. we can again examine the HTML to find that the structure contains:

* Titles are stored in the article a nodes as text
* Dates are stored in the article p.datetime nodes as text
* Paths (href) are stored in the article a nodes as href attributes

We can extract this information using three web scrapings (one for title, dates, and hrefs), and then binding each together using bind\_cols(). We output the first six posts in a table using the head() function.

# Get the path to all of the posts

posts\_path <- "http://varianceexplained.org/posts/"

# Extract the post titles

titles\_vec <- read\_html(posts\_path) %>%

html\_node("#main") %>%

html\_nodes("article") %>%

html\_nodes("a") %>%

html\_text(trim = TRUE)

# Extract the post dates

dates\_vec <- read\_html(posts\_path) %>%

html\_node("#main") %>%

html\_nodes("article") %>%

html\_nodes("p.dateline") %>%

html\_text(trim = TRUE) %>%

mdy()

# Extract the post hrefs

hrefs\_vec <- read\_html(posts\_path) %>%

html\_node("#main") %>%

html\_nodes("article") %>%

html\_nodes("a") %>%

html\_attr("href")

# Bind the data together in a tibble

variance\_explained\_tbl <- bind\_cols(

title = titles\_vec,

date = dates\_vec,

href = hrefs\_vec)

# First six posts shown

variance\_explained\_tbl %>%

head() %>%

knitr::kable()

| **title** | **date** | **href** |
| --- | --- | --- |
| What digits should you bet on in Super Bowl squares? | 2018-02-04 | http://varianceexplained.org/r/super-bowl-squares/ |
| Exploring handwritten digit classification: a tidy analysis of the MNIST dataset | 2018-01-22 | http://varianceexplained.org/r/digit-eda/ |
| What’s the difference between data science, machine learning, and artificial intelligence? | 2018-01-09 | http://varianceexplained.org/r/ds-ml-ai/ |
| Advice to aspiring data scientists: start a blog | 2017-11-14 | http://varianceexplained.org/r/start-blog/ |
| Announcing “Introduction to the Tidyverse”, my new DataCamp course | 2017-11-09 | http://varianceexplained.org/r/intro-tidyverse/ |
| Don’t teach students the hard way first | 2017-09-21 | http://varianceexplained.org/r/teach-hard-way/ |

We now have all 58 of DRob’s posts (title, date, and href) and are now ready to scale! To simplify the process, we’ll create a custom function, build\_function\_names\_tbl\_from\_url\_path(), that combines several rvest operations from the previous section.

build\_function\_names\_tbl\_from\_url\_path <- function(path, loaded\_functions\_tbl) {

builder <- function(path, loaded\_functions\_tbl) {

read\_html(path) %>%

html\_nodes("code") %>%

html\_text() %>%

parse\_function\_names() %>%

left\_join(loaded\_functions\_tbl) %>%

filter(

!(function\_name == "filter" & !(package == "dplyr"))

) %>%

mutate(package = ifelse(is.na(package), "Unknown", package))

}

safe\_builder <- possibly(builder, otherwise = NA)

safe\_builder(path, loaded\_functions\_tbl)

}

We can test the function to see how it takes a path and a tibble of loaded\_functions\_tbl and returns the functions and packages.

path <- "http://varianceexplained.org/r/mixture-models-baseball/"

build\_function\_names\_tbl\_from\_url\_path(path, loaded\_functions\_tbl) %>%

glimpse()

## Observations: 131

## Variables: 2

## $ function\_name "library", "library", "library", "library",...

## $ package "base", "base", "base", "base", "ggplot2", ...

Next, we can scale this to all posts using map() functions from the purrr package. Several of the posts have no code and therefore return nested NA values. We filter them out by mapping is.logical. We unnest() the function\_name column to reveal the nested function names and packages.

variance\_explained\_tbl <- bind\_cols(

title = titles\_vec,

date = dates\_vec,

href = hrefs\_vec) %>%

mutate(

function\_name = map(href, build\_function\_names\_tbl\_from\_url\_path, loaded\_functions\_tbl),

is\_logical = map\_dbl(function\_name, is.logical)

) %>%

filter(is\_logical == 0) %>%

select(-is\_logical) %>%

unnest()

variance\_explained\_tbl %>% glimpse()

## Observations: 2,314

## Variables: 5

## $ title "What digits should you bet on in Super Bow...

## $ date 2018-02-04, 2018-02-04, 2018-02-04, 2018-0...

## $ href "http://varianceexplained.org/r/super-bowl-...

## $ function\_name "library", "theme\_set", "theme\_light", "dir...

## $ package "base", "ggplot2", "ggplot2", "base", "purr...

Awesome – We now have all of the function names and most of the packages that DRob used ALL of his code on Variance Explained! Notice that the sample size has increased to 2314 functions extracted. This is a much larger sample size than before with the single Mixture Models post, which had 131 functions extracted.

**Part 2: Learning From DRob’s Code**

The question we need to answer is **“What Code Does An R Master Use To Perform Data Science?”** We can break this down into separate questions of interest:

**Which Functions Are Most Frequently Used by DRob?**

We can answer this question with by counting our package and function name frequencies, sorting, and taking the top 20, which gives us a subset of the most frequently used functions.

ve\_functions\_top\_20\_tbl <- variance\_explained\_tbl %>%

count(package, function\_name) %>%

count\_to\_pct() %>%

arrange(desc(n)) %>%

top\_n(20) %>%

mutate(function\_name = as\_factor(function\_name) %>% fct\_reorder(n)) %>%

arrange(desc(function\_name)) %>%

mutate(package = as\_factor(package))

ve\_functions\_top\_20\_tbl %>% glimpse()

## Observations: 20

## Variables: 4

## $ package base, ggplot2, dplyr, dplyr, ggplot2, dplyr...

## $ function\_name library, aes, filter, mutate, ggplot, group...

## $ n 171, 123, 123, 122, 83, 62, 61, 59, 41, 38,...

## $ pct 0.07389801, 0.05315471, 0.05315471, 0.05272...

We can visualize this data using ggplot2. We chose a lollipop style chart that extends lengthwise for the top 20, which shows off the number and percentage of total for each of the top 20 functions. We can see that base::library(), ggplot2::aes(), dplyr::filter(), and dplyr::mutate() are very frequently used by DRob. In fact, these four functions comprise 23.3% of his total functions. Unfortunately, aes() can’t be used alone (see below for how it’s used with the ggplot() function). **However, with knowledge of library() and the combination of filter() and mutate() from dplyr, a learner can understand 18% of DRob’s code!**

ve\_functions\_top\_20\_tbl %>%

ggplot(aes(x = n, y = function\_name, color = package)) +

geom\_segment(aes(xend = 0, yend = function\_name), size = 2) +

geom\_point(size = 4) +

geom\_label(aes(label = paste0(function\_name, "(), ", package, ", ", scales::percent(pct))),

hjust = "inward", size = 3.5) +

expand\_limits(x = 0) +

labs(

title = "Which Functions Are Most Frequently Used by DRob?",

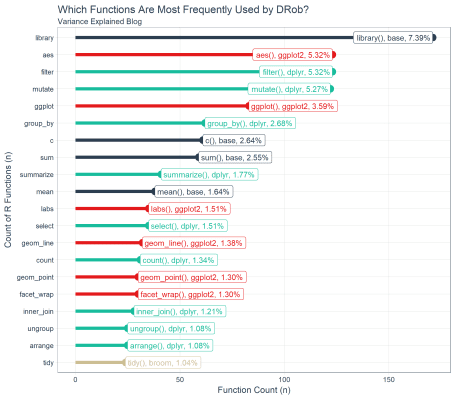
subtitle = "Variance Explained Blog",

x = "Function Count (n)", y = "Count of R Functions (n)") +

scale\_color\_tq() +

theme\_tq() +

theme(legend.position = "none")



**Which Packages Are Most Frequently Used by DRob?**

We can answer this question a number of ways, and we elect to make a time-based analysis to expose underlying trends within packages over time. The idea is that some packages may be used more frequently for specific reasons, and we aim to uncover the true trend of the packages which is not constant. We’ll use the tibbletime package to help out with the time-based analysis by aggregating (or grouping) the data by six-month intervals. Note that we lump (using fct\_lump()) all packages into six categories based on the top 5 packages and an extra column called “Other”. a label is made by pasting “H” with semester(date) to return the which half of the year the data is aggregated.

ve\_package\_frequency\_tbl <- variance\_explained\_tbl %>%

select(date, package, function\_name) %>%

mutate(package = as.factor(package) %>% fct\_lump(n = 5, other\_level = "Other")) %>%

arrange(date) %>%

as\_tbl\_time(index = date) %>%

collapse\_by(period = "6 m", clean = TRUE) %>%

count(date, package) %>%

count\_to\_pct(date) %>%

mutate(biannual = paste0("H", semester(date)))

ve\_package\_frequency\_tbl %>% glimpse()

## Observations: 47

## Variables: 5

## $ date 2015-01-01, 2015-01-01, 2015-01-01, 2015-01-01,...

## $ package base, ggplot2, stats, Unknown, Other, base, dply...

## $ n 22, 28, 2, 20, 6, 19, 8, 32, 7, 12, 17, 131, 110...

## $ pct 0.28205128, 0.35897436, 0.02564103, 0.25641026, ...

## $ biannual "H1", "H1", "H1", "H1", "H1", "H2", "H2", "H2", ...

Next, we can visualize with ggplot2. **The total functions (column n in ve\_package\_frequency\_tbl) used are misleading since in some half years DRob posts less than in others**. We can normalize by switching to percentage of total functions by half year.

ve\_package\_frequency\_tbl %>%

ggplot(aes(date, n, fill = package)) +

geom\_bar(stat = "identity") +

geom\_text(aes(x = date, y = n, label = biannual),

vjust = -1, color = palette\_light()[[1]], size = 3) +

geom\_smooth(method = "lm", se = FALSE) +

facet\_wrap(~ package, ncol = 3) +

scale\_fill\_tq() +

theme\_tq() +

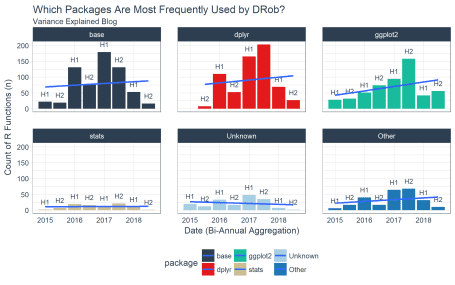
labs(

title = "Which Packages Are Most Frequently Used by DRob?",

subtitle = "Variance Explained Blog",

x = "Date (Bi-Annual Aggregation)", y = "Count of R Functions (n)"

)



We switch to a percentage of total functions (pct column in ve\_package\_frequency\_tbl) to get a better perspective on what trends are happening within posts over time. We see that DRob is trending in the direction of more dplyr and ggplot2 and using fewer “Unknown” packages, which are packages that I do not have currently loaded on my machine (e.g. not “tidyverse” or “base”). It’s clear that base, dplyr, and ggplot2 are DRob’s toolkits of choice.

ve\_package\_frequency\_tbl %>%

ggplot(aes(date, pct, fill = package)) +

geom\_bar(stat = "identity") +

geom\_text(aes(x = date, y = pct, label = biannual),

vjust = -1, color = palette\_light()[[1]], size = 3) +

geom\_smooth(method = "lm", se = FALSE) +

facet\_wrap(~ package, ncol = 3) +

scale\_y\_continuous(labels = scales::percent) +

scale\_fill\_tq() +

theme\_tq() +

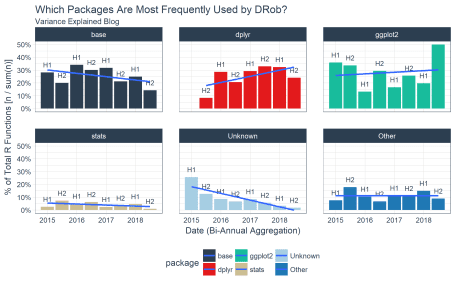
labs(

title = "Which Packages Are Most Frequently Used by DRob?",

subtitle = "Variance Explained Blog",

x = "Date (Bi-Annual Aggregation)", y = "% of Total R Functions [n / sum(n)]"

)



Finally, we can get the overall percentage of package usage by uncounting and recounting by package. We add a cumulative percentage column and see that we can almost get to 80% with just three package: dplyr, base, and ggplot2.

ve\_package\_frequency\_tbl %>%

uncount(weights = n) %>%

count(package) %>%

count\_to\_pct() %>%

arrange(desc(n)) %>%

mutate(pct\_cum = cumsum(pct)) %>%

knitr::kable()

| **package** | **n** | **pct** | **pct\_cum** |
| --- | --- | --- | --- |
| dplyr | 634 | 0.2739844 | 0.2739844 |
| base | 627 | 0.2709594 | 0.5449438 |
| ggplot2 | 535 | 0.2312014 | 0.7761452 |
| Other | 254 | 0.1097666 | 0.8859118 |
| Unknown | 174 | 0.0751945 | 0.9611063 |
| stats | 90 | 0.0388937 | 1.0000000 |

**How “Tidy” Is DRob’s Code?**

We saw in the package analysis that DRob is using quite a few “tidy” packages. We can extend the analysis to see how frequently he’s using “tidyverse” functions.

The tidyverse is a very popular set of packages that are developed specifically to do data science in an integrated and easy to understand way. Currently, the “tidyverse” consists of the following packages:

tidyverse\_packages(include\_self = F)

## [1] "broom" "cli" "crayon" "dplyr"

## [5] "dbplyr" "forcats" "ggplot2" "haven"

## [9] "hms" "httr" "jsonlite" "lubridate"

## [13] "magrittr" "modelr" "purrr" "readr"

## [17] "readxl\n(>=" "reprex" "rlang" "rstudioapi"

## [21] "rvest" "stringr" "tibble" "tidyr"

## [25] "xml2"

We can flag functions from the tidyverse package from DRob’s code base using the tidyverse\_packages() function. If functions are in a tidyverse package, the are flagged as “Yes” and otherwise “No”.

ve\_tidiness\_tbl <- variance\_explained\_tbl %>%

select(date, function\_name, package) %>%

mutate(tidy\_function = case\_when(

package %in% tidyverse\_packages() ~ "Yes",

TRUE ~ "No"))

ve\_tidiness\_tbl %>% glimpse()

## Observations: 2,314

## Variables: 4

## $ date 2018-02-04, 2018-02-04, 2018-02-04, 2018-0...

## $ function\_name "library", "theme\_set", "theme\_light", "dir...

## $ package "base", "ggplot2", "ggplot2", "base", "purr...

## $ tidy\_function "No", "Yes", "Yes", "No", "Yes", "Yes", "Ye...

Here’s how easy it is to quickly see how tidy DRob is. About 60% of his functions are “tidyverse” functions.

ve\_tidiness\_tbl %>%

count(tidy\_function) %>%

count\_to\_pct() %>%

arrange(desc(n)) %>%

knitr::kable()

| **tidy\_function** | **n** | **pct** |
| --- | --- | --- |
| Yes | 1373 | 0.5933449 |
| No | 941 | 0.4066551 |

How has DRob’s “tidiness” changed over time? We’ll again call upon tibbletime to help transform the data using collapse\_by().

ve\_tidiness\_over\_time\_tbl <- ve\_tidiness\_tbl %>%

select(date, tidy\_function, function\_name, package) %>%

arrange(date) %>%

as\_tbl\_time(index = date) %>%

collapse\_by(period = "6 m", clean = TRUE) %>%

count(date, tidy\_function) %>%

count\_to\_pct(date) %>%

filter(tidy\_function == "Yes") %>%

mutate(biannual = paste0("H", semester(date)))

glimpse(ve\_tidiness\_over\_time\_tbl)

## Observations: 8

## Variables: 5

## $ date 2015-01-01, 2015-07-01, 2016-01-01, 2016-0...

## $ tidy\_function "Yes", "Yes", "Yes", "Yes", "Yes", "Yes", "...

## $ n 29, 53, 189, 133, 310, 427, 140, 92

## $ pct 0.3717949, 0.5578947, 0.4921875, 0.5277778,...

## $ biannual "H1", "H2", "H1", "H2", "H1", "H2", "H1", "H2"

Here’s a fun fact… According to this graph, DRob is over twice as “tidy” now as when he started blogging in 2015. This should tell us that we really need to give the “tidyverse” a shot if we aren’t using it now.

ve\_tidiness\_over\_time\_tbl %>%

ggplot(aes(date, pct)) +

geom\_bar(stat = "identity", fill = palette\_light()[[1]], color = "white") +

geom\_text(aes(x = date, y = pct, label = biannual),

vjust = -1, color = palette\_light()[[1]], size = 3) +

geom\_text(aes(x = date, y = pct, label = scales::percent(pct)),

vjust = 2, color = "white", size = 3) +

geom\_smooth(method = "lm", se = FALSE) +

scale\_y\_continuous(labels = scales::percent) +

scale\_fill\_tq() +

theme\_tq() +

labs(

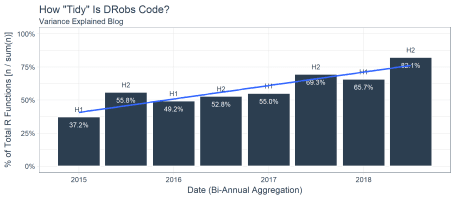
title = 'How "Tidy" Is DRobs Code?',

subtitle = "Variance Explained Blog",

x = "Date (Bi-Annual Aggregation)", y = "% of Total R Functions [n / sum(n)]"

) +

expand\_limits(y = 1)



**Which Functions and Packages Should We Focus On For Learning R?**

Now the million dollar question: What should we focus on if we are just starting out in R? We’ll use the 80/20 Rule, which boils down to which top functions build 80% of DRob’s code. Ideally this should be around 20% according to the rule. The question is actually really easy to answer using the cumsum() function from base. We can flag any cumulative percentages that are less than or equal to 80% as “high usage”.

ve\_eighty\_twenty\_tbl <- variance\_explained\_tbl %>%

count(package, function\_name) %>%

count\_to\_pct() %>%

arrange(desc(pct)) %>%

mutate(

pct\_cum = cumsum(pct),

high\_usage = case\_when(

pct\_cum <= 0.8 ~ "Yes",

TRUE ~ "No"

))

ve\_eighty\_twenty\_tbl %>% glimpse()

## Observations: 312

## Variables: 6

## $ package "base", "dplyr", "ggplot2", "dplyr", "ggplo...

## $ function\_name "library", "filter", "aes", "mutate", "ggpl...

## $ n 171, 123, 123, 122, 83, 62, 61, 59, 41, 38,...

## $ pct 0.073898012, 0.053154710, 0.053154710, 0.05...

## $ pct\_cum 0.07389801, 0.12705272, 0.18020743, 0.23292...

## $ high\_usage "Yes", "Yes", "Yes", "Yes", "Yes", "Yes", "...

Next, we just count our high usage flags and turn the count to percent. We can see that 28.2% of functions create 80% of DRob’s code.

ve\_eighty\_twenty\_tbl %>%

count(high\_usage) %>%

count\_to\_pct(col = nn) %>%

knitr::kable()

| **high\_usage** | **nn** | **pct** |
| --- | --- | --- |
| No | 224 | 0.7179487 |
| Yes | 88 | 0.2820513 |

Finally, here are the functions by package that we should focus on if we are just starting out. Keep in mind this is just DRob and we may want to expand to other masters of data science to get an even better picture of the high usage functions.

ve\_eighty\_twenty\_tbl %>%

filter(high\_usage == "Yes") %>%

split(.$package)

## $base

## # A tibble: 23 x 6

## package function\_name n pct pct\_cum high\_usage

##

## 1 base library 171 0.0739 0.0739 Yes

## 2 base c 61 0.0264 0.322 Yes

## 3 base sum 59 0.0255 0.347 Yes

## 4 base mean 38 0.0164 0.382 Yes

## 5 base function 23 0.00994 0.519 Yes

## 6 base list 18 0.00778 0.537 Yes

## 7 base seq 17 0.00735 0.552 Yes

## 8 base set.seed 14 0.00605 0.585 Yes

## 9 base seq\_len 12 0.00519 0.619 Yes

## 10 base log10 11 0.00475 0.634 Yes

## 11 base sample 11 0.00475 0.639 Yes

## 12 base cbind 10 0.00432 0.653 Yes

## 13 base log 10 0.00432 0.657 Yes

## 14 base is.na 9 0.00389 0.674 Yes

## 15 base min 9 0.00389 0.678 Yes

## 16 base cumsum 8 0.00346 0.720 Yes

## 17 base paste0 8 0.00346 0.724 Yes

## 18 base matrix 7 0.00303 0.741 Yes

## 19 base colSums 6 0.00259 0.768 Yes

## 20 base max 6 0.00259 0.770 Yes

## 21 base as.Date 5 0.00216 0.788 Yes

## 22 base replicate 5 0.00216 0.790 Yes

## 23 base t 5 0.00216 0.792 Yes

##

## $broom

## # A tibble: 2 x 6

## package function\_name n pct pct\_cum high\_usage

##

## 1 broom tidy 24 0.0104 0.509 Yes

## 2 broom augment 7 0.00303 0.744 Yes

##

## $dplyr

## # A tibble: 20 x 6

## package function\_name n pct pct\_cum high\_usage

##

## 1 dplyr filter 123 0.0532 0.127 Yes

## 2 dplyr mutate 122 0.0527 0.233 Yes

## 3 dplyr group\_by 62 0.0268 0.296 Yes

## 4 dplyr summarize 41 0.0177 0.365 Yes

## 5 dplyr select 35 0.0151 0.397 Yes

## 6 dplyr count 31 0.0134 0.439 Yes

## 7 dplyr inner\_join 28 0.0121 0.477 Yes

## 8 dplyr arrange 25 0.0108 0.488 Yes

## 9 dplyr ungroup 25 0.0108 0.499 Yes

## 10 dplyr n 23 0.00994 0.529 Yes

## 11 dplyr desc 15 0.00648 0.573 Yes

## 12 dplyr tbl\_df 14 0.00605 0.591 Yes

## 13 dplyr funs 11 0.00475 0.644 Yes

## 14 dplyr anti\_join 9 0.00389 0.682 Yes

## 15 dplyr mutate\_each 8 0.00346 0.727 Yes

## 16 dplyr rename 8 0.00346 0.731 Yes

## 17 dplyr top\_n 8 0.00346 0.734 Yes

## 18 dplyr data\_frame 7 0.00303 0.747 Yes

## 19 dplyr distinct 7 0.00303 0.750 Yes

## 20 dplyr do 6 0.00259 0.773 Yes

##

## $ggplot2

## # A tibble: 23 x 6

## package function\_name n pct pct\_cum high\_usage

##

## 1 ggplot2 aes 123 0.0532 0.180 Yes

## 2 ggplot2 ggplot 83 0.0359 0.269 Yes

## 3 ggplot2 labs 35 0.0151 0.412 Yes

## 4 ggplot2 geom\_line 32 0.0138 0.426 Yes

## 5 ggplot2 facet\_wrap 30 0.0130 0.452 Yes

## 6 ggplot2 geom\_point 30 0.0130 0.465 Yes

## 7 ggplot2 geom\_histogram 14 0.00605 0.597 Yes

## 8 ggplot2 geom\_smooth 13 0.00562 0.603 Yes

## 9 ggplot2 scale\_y\_continuous 13 0.00562 0.608 Yes

## 10 ggplot2 theme 12 0.00519 0.624 Yes

## 11 ggplot2 theme\_set 10 0.00432 0.662 Yes

## 12 ggplot2 ylab 10 0.00432 0.666 Yes

## 13 ggplot2 element\_text 9 0.00389 0.686 Yes

## 14 ggplot2 geom\_bar 9 0.00389 0.690 Yes

## 15 ggplot2 geom\_text 9 0.00389 0.694 Yes

## 16 ggplot2 scale\_x\_log10 9 0.00389 0.697 Yes

## 17 ggplot2 theme\_bw 9 0.00389 0.701 Yes

## 18 ggplot2 geom\_tile 7 0.00303 0.753 Yes

## 19 ggplot2 geom\_abline 6 0.00259 0.775 Yes

## 20 ggplot2 geom\_vline 6 0.00259 0.778 Yes

## 21 ggplot2 geom\_boxplot 5 0.00216 0.794 Yes

## 22 ggplot2 geom\_hline 5 0.00216 0.796 Yes

## 23 ggplot2 theme\_void 5 0.00216 0.799 Yes

##

## $lubridate

## # A tibble: 1 x 6

## package function\_name n pct pct\_cum high\_usage

##

## 1 lubridate round\_date 7 0.00303 0.756 Yes

##

## $purrr

## # A tibble: 1 x 6

## package function\_name n pct pct\_cum high\_usage

##

## 1 purrr map 9 0.00389 0.705 Yes

##

## $stats

## # A tibble: 5 x 6

## package function\_name n pct pct\_cum high\_usage

##

## 1 stats reorder 16 0.00691 0.566 Yes

## 2 stats qbeta 12 0.00519 0.630 Yes

## 3 stats lm 10 0.00432 0.670 Yes

## 4 stats dbeta 7 0.00303 0.759 Yes

## 5 stats rbeta 6 0.00259 0.780 Yes

##

## $stringr

## # A tibble: 1 x 6

## package function\_name n pct pct\_cum high\_usage

##

## 1 stringr str\_detect 17 0.00735 0.559 Yes

##

## $tibble

## # A tibble: 1 x 6

## package function\_name n pct pct\_cum high\_usage

##

## 1 tibble data\_frame 7 0.00303 0.762 Yes

##

## $tidyr

## # A tibble: 5 x 6

## package function\_name n pct pct\_cum high\_usage

##

## 1 tidyr separate 18 0.00778 0.545 Yes

## 2 tidyr gather 15 0.00648 0.579 Yes

## 3 tidyr crossing 13 0.00562 0.614 Yes

## 4 tidyr unite 9 0.00389 0.709 Yes

## 5 tidyr unnest 9 0.00389 0.713 Yes

##

## $Unknown

## # A tibble: 5 x 6

## package function\_name n pct pct\_cum high\_usage

##

## 1 Unknown percent\_format 9 0.00389 0.717 Yes

## 2 Unknown ebb\_fit\_prior 8 0.00346 0.738 Yes

## 3 Unknown unnest\_tokens 7 0.00303 0.765 Yes

## 4 Unknown dbetabinom.ab 6 0.00259 0.783 Yes

## 5 Unknown mcbind 6 0.00259 0.786 Yes

##

## $utils

## # A tibble: 1 x 6

## package function\_name n pct pct\_cum high\_usage

##

## 1 utils head 11 0.00475 0.649 Yes

**Takeaways From DRob’s Code**

1. DRob is using quite a bit of base, dplyr, and ggplot2 code. **In fact, these three libraries account for 77.6% of his code on Variance Explained.**
2. DRob’s code is getting… tidier! **DRob is using approximately 80% tidyverse code in the most recent half-year of blogging.** This trend is increasing, although it will eventually top out. This compares to around 37% tidy code when he began blogging in 2015.
3. If DRob is getting tidier, which area is getting impacted the most? It’s the packages I’ve categorized as “Unknown”. These are non-tidyverse or pre-loaded packages. In other words, these are uncommonly used packages that may serve a specialized need. I do not currently have these loaded, which is why they are considered “Unknown”. It’s worth mentioning that base and stats libraries are declining slightly, but not to the extent that specialized packages are declining. **The bottom line – DRob is using less specialized packages and more tidyverse.**

**Analysis Risks**

One point I have not discussed is that DRob is just one *really good* data scientist. His code is clearly representative of the tidyverse-style, which resonates with many future data scientists coming into the industry. If one wishes to emulate DRob, this is probably a good analysis to take and run with. However, it may make sense to also view other “masters” that exist as part of a future endeavor.

Another point is that we got 2,314 functions out of 58 posts. While this is by no means a small sample, we certainly may wish to increase the sample size to get more confidence in the most high usage functions. Personally, I’d like to see a 100X ratio between top functions and total observations, meaning the top 100 functions would be from at a minimum 10,000 functions. With that said, the analysis was performed accross a large sample of projects (58 posts less those that do not contain code) and multiple years which is another factor that improves confidence.

**Bonus: Analyze Your Code**

We spoke a lot about analyzing DRob’s code, but with a few modifications you can apply this analysis to your own code stored in .R or .Rmd files! Here’s how with the fs package.

We’ll begin with a relatively large code base from a project I’m working on, which is a new course called **HR 201: Predicting Employee Attrition**.

**Part 1: Extracting Your Functions From Your Code Base**

First, load the fs package. This is a great package for working with the file system on you computer.

library(fs)

Next, collect the path for YOUR code base directory. I will use my R Project directory for the HR 201 Course.

dir\_path <- "../../../Business Science/Courses/Teachable/HR201\_Employee\_Turnover\_H2O/HR201\_Employee\_Turnover\_Project/"

Use a function called dir\_info() to retrieve the contents of the directory. Add the argument, recursive = TRUE, to collect all the files from the sub-directories. Use head() to return the first six rows only.

dir\_info(dir\_path, recursive = TRUE) %>%

head() %>%

knitr::kable()

| **path** | **type** | **size** | **permissions** | **modification\_time** | **user** | **group** | **device\_id** | **hard\_links** | **special\_device\_id** | **inode** | **block\_size** | **blocks** | **flags** | **generation** | **access\_time** | **change\_time** | **birth\_time** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ../../../Business Science/Courses/Teachable/HR201\_Employee\_Turnover\_H2O/HR201\_Employee\_Turnover\_Project/00\_Data | directory | 0 | r– | 2018-02-18 08:25:04 | NA | NA | 2586258886 | 1 | 0 | 1.322932e+16 | 4096 | 8 | 0 | 0 | 2018-02-18 08:25:04 | 2018-02-18 08:25:04 | 2018-02-02 16:43:35 |
| ../../../Business Science/Courses/Teachable/HR201\_Employee\_Turnover\_H2O/HR201\_Employee\_Turnover\_Project/00\_Data/desktop.ini | file | 142 | rw- | 2018-02-02 16:43:49 | NA | NA | 2586258886 | 1 | 0 | 2.589570e+16 | 4096 | 0 | 0 | 0 | 2018-02-02 16:43:49 | 2018-02-02 16:43:49 | 2018-02-02 16:43:49 |
| ../../../Business Science/Courses/Teachable/HR201*Employee\_Turnover\_H2O/HR201\_Employee\_Turnover\_Project/00\_Data/WA\_Fn-UseC*-HR-Employee-Attrition.xlsx | file | 255.7K | rw- | 2017-11-26 06:37:44 | NA | NA | 2586258886 | 1 | 0 | 1.857735e+16 | 4096 | 512 | 0 | 0 | 2018-02-02 16:43:35 | 2018-02-08 15:43:31 | 2018-02-02 16:43:35 |
| ../../../Business Science/Courses/Teachable/HR201\_Employee\_Turnover\_H2O/HR201\_Employee\_Turnover\_Project/00\_Scripts | directory | 0 | r– | 2018-02-26 09:28:32 | NA | NA | 2586258886 | 1 | 0 | 4.053240e+16 | 4096 | 8 | 0 | 0 | 2018-02-26 09:28:32 | 2018-02-26 09:28:32 | 2018-02-02 16:43:35 |
| ../../../Business Science/Courses/Teachable/HR201\_Employee\_Turnover\_H2O/HR201\_Employee\_Turnover\_Project/00\_Scripts/assess\_attrition.R | file | 4.01K | rw- | 2018-02-27 21:22:11 | NA | NA | 2586258886 | 1 | 0 | 9.007199e+15 | 4096 | 16 | 0 | 0 | 2018-02-27 21:22:11 | 2018-02-27 21:22:24 | 2018-02-05 16:36:02 |
| ../../../Business Science/Courses/Teachable/HR201\_Employee\_Turnover\_H2O/HR201\_Employee\_Turnover\_Project/00\_Scripts/desktop.ini | file | 142 | rw- | 2018-02-02 16:43:49 | NA | NA | 2586258886 | 1 | 0 | 4.503600e+15 | 4096 | 0 | 0 | 0 | 2018-02-02 16:43:49 | 2018-02-02 16:43:49 | 2018-02-02 16:43:49 |

Now that we see how dir\_info() works, we can use one more function called path\_file() to retrieve just the file portion of the path. We can then use the file name with str\_detect() to detect only files with “.R” or “.Rmd” at the end. We’ll create a tibble of the file names and paths.

rmd\_or\_r\_file\_paths\_tbl <- dir\_info(dir\_path, recursive = T) %>%

mutate(file\_name = path\_file(path)) %>%

select(file\_name, path) %>%

filter(str\_detect(file\_name, "(\\.R|\\.Rmd)$"))

rmd\_or\_r\_file\_paths\_tbl %>% knitr::kable()

| **file\_name** | **path** |
| --- | --- |
| assess\_attrition.R | ../../../Business Science/Courses/Teachable/HR201\_Employee\_Turnover\_H2O/HR201\_Employee\_Turnover\_Project/00\_Scripts/assess\_attrition.R |
| make\_directory.R | ../../../Business Science/Courses/Teachable/HR201\_Employee\_Turnover\_H2O/HR201\_Employee\_Turnover\_Project/00\_Scripts/make\_directory.R |
| mlhelpers.R | ../../../Business Science/Courses/Teachable/HR201\_Employee\_Turnover\_H2O/HR201\_Employee\_Turnover\_Project/00\_Scripts/mlhelpers.R |
| pipeline\_for\_modeling\_attrition.R | ../../../Business Science/Courses/Teachable/HR201\_Employee\_Turnover\_H2O/HR201\_Employee\_Turnover\_Project/00\_Scripts/pipeline\_for\_modeling\_attrition.R |
| business\_understanding.R | ../../../Business Science/Courses/Teachable/HR201\_Employee\_Turnover\_H2O/HR201\_Employee\_Turnover\_Project/01\_Business\_Understanding/business\_understanding.R |
| data\_understanding.R | ../../../Business Science/Courses/Teachable/HR201\_Employee\_Turnover\_H2O/HR201\_Employee\_Turnover\_Project/02\_Data\_Understanding/data\_understanding.R |
| data\_preparation\_part\_1\_readable.R | ../../../Business Science/Courses/Teachable/HR201\_Employee\_Turnover\_H2O/HR201\_Employee\_Turnover\_Project/03\_Data\_Preparation/data\_preparation\_part\_1\_readable.R |
| data\_preparation\_part\_2\_machine\_learning.R | ../../../Business Science/Courses/Teachable/HR201\_Employee\_Turnover\_H2O/HR201\_Employee\_Turnover\_Project/03\_Data\_Preparation/data\_preparation\_part\_2\_machine\_learning.R |
| modeling\_part\_1\_h2o.R | ../../../Business Science/Courses/Teachable/HR201\_Employee\_Turnover\_H2O/HR201\_Employee\_Turnover\_Project/04\_Modeling/modeling\_part\_1\_h2o.R |
| modeling\_part\_2\_lime.R | ../../../Business Science/Courses/Teachable/HR201\_Employee\_Turnover\_H2O/HR201\_Employee\_Turnover\_Project/04\_Modeling/modeling\_part\_2\_lime.R |
| evaluation\_expected\_value.R | ../../../Business Science/Courses/Teachable/HR201\_Employee\_Turnover\_H2O/HR201\_Employee\_Turnover\_Project/05\_Evaluation/evaluation\_expected\_value.R |

Next, we can create a custom function called, build\_function\_names\_tbl\_from\_file\_path(), which is very similar function to the url builder before. The main difference is that the HTML extraction code is replaced with readLines().

build\_function\_names\_tbl\_from\_file\_path <- function(path, loaded\_functions\_tbl) {

builder <- function(path, loaded\_functions\_tbl) {

readLines(path) %>%

parse\_function\_names() %>%

left\_join(loaded\_functions\_tbl) %>%

filter(

!(function\_name == "filter" & !(package == "dplyr"))

) %>%

mutate(package = ifelse(is.na(package), "Unknown", package))

}

safe\_builder <- possibly(builder, otherwise = NA)

safe\_builder(path, loaded\_functions\_tbl)

}

We can test it with one of the file paths.

file\_path\_1 <- rmd\_or\_r\_file\_paths\_tbl$path[[1]]

file\_path\_1

## [1] "../../../Business Science/Courses/Teachable/HR201\_Employee\_Turnover\_H2O/HR201\_Employee\_Turnover\_Project/00\_Scripts/assess\_attrition.R"

Let’s see what it returns.

build\_function\_names\_tbl\_from\_file\_path(file\_path\_1, loaded\_functions\_tbl) %>%

glimpse()

## Observations: 57

## Variables: 2

## $ function\_name "library", "count", "function", "quos", "en...

## $ package "base", "dplyr", "base", "dplyr", "dplyr", ...

Great, it works identically to the web scraping version but with local file paths. We have 57 functions just in the first file.

We can scale it to all code in the code base using the file paths. The process is almost identical to the web scraping process.

local\_function\_names\_tbl <- rmd\_or\_r\_file\_paths\_tbl %>%

mutate(

function\_name = map(path, build\_function\_names\_tbl\_from\_file\_path, loaded\_functions\_tbl),

is\_logical = map\_dbl(function\_name, is.logical)

) %>%

filter(is\_logical != 1) %>%

select(file\_name, function\_name) %>%

unnest() %>%

left\_join(loaded\_functions\_tbl)

local\_function\_names\_tbl %>% glimpse()

## Observations: 1,422

## Variables: 3

## $ file\_name "assess\_attrition.R", "assess\_attritio...

## $ function\_name "library", "count", "function", "quos", "en...

## $ package "base", "dplyr", "base", "dplyr", "dplyr", ...

**Part 2: Analyzing Your Code**

You can run through the same process with your code. Here are my top 20 functions.

local\_functions\_top\_20\_tbl <- local\_function\_names\_tbl %>%

count(package, function\_name) %>%

count\_to\_pct() %>%

arrange(desc(n)) %>%

top\_n(20) %>%

rowid\_to\_column(var = "rank")

local\_functions\_top\_20\_tbl %>%

ggplot(aes(x = n, y = fct\_reorder(function\_name, n), color = package)) +

geom\_segment(aes(xend = 0, yend = function\_name), size = 2) +

geom\_point(size = 4) +

geom\_label(aes(label = paste0(function\_name, "(), ", package, ", ", scales::percent(pct))),

hjust = "inward", size = 3.5) +

expand\_limits(x = 0) +

labs(

title = "Which Functions Are Most Frequently Used by DRob?",

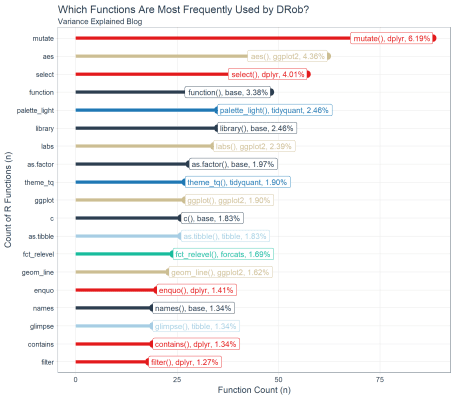
subtitle = "Variance Explained Blog",

x = "Function Count (n)", y = "Count of R Functions (n)") +

scale\_color\_tq() +

theme\_tq() +

theme(legend.position = "none")



**Similarities**

You can assess the similarities and differences between you and DRob. For example, DRob and I both use quite a bit of dplyr for data manipulation and ggplot2 for visualization.

local\_functions\_top\_20\_tbl %>%

filter(function\_name %in% ve\_functions\_top\_20\_tbl$function\_name) %>%

knitr::kable()

| **rank** | **package** | **function\_name** | **n** | **pct** |
| --- | --- | --- | --- | --- |
| 1 | dplyr | mutate | 88 | 0.0618847 |
| 2 | ggplot2 | aes | 62 | 0.0436006 |
| 3 | dplyr | select | 57 | 0.0400844 |
| 5 | base | library | 35 | 0.0246132 |
| 7 | ggplot2 | labs | 34 | 0.0239100 |
| 9 | ggplot2 | ggplot | 27 | 0.0189873 |
| 11 | base | c | 26 | 0.0182841 |
| 14 | ggplot2 | geom\_line | 23 | 0.0161744 |
| 20 | dplyr | filter | 18 | 0.0126582 |

**Differences**

I have a few differences related to my coding techniques. I do quite a bit of programming so base::function() is in fourth place and dplyr::enquo() (part of the new [tidy eval](http://dplyr.tidyverse.org/articles/programming.html) framework) is in 15th place. I also have tidyquant::palette\_light() and tidyquant::theme\_tq() related to my preference for tidyquant ggplot2 themes.

local\_functions\_top\_20\_tbl %>%

filter(!function\_name %in% ve\_functions\_top\_20\_tbl$function\_name) %>%

knitr::kable()

| **rank** | **package** | **function\_name** | **n** | **pct** |
| --- | --- | --- | --- | --- |
| 4 | base | function | 48 | 0.0337553 |
| 6 | tidyquant | palette\_light | 35 | 0.0246132 |
| 8 | base | as.factor | 28 | 0.0196906 |
| 10 | tidyquant | theme\_tq | 27 | 0.0189873 |
| 12 | tibble | as.tibble | 26 | 0.0182841 |
| 13 | forcats | fct\_relevel | 24 | 0.0168776 |
| 15 | dplyr | enquo | 20 | 0.0140647 |
| 16 | base | names | 19 | 0.0133615 |
| 17 | dplyr | contains | 19 | 0.0133615 |
| 18 | dplyr | glimpse | 19 | 0.0133615 |
| 19 | tibble | glimpse | 19 | 0.0133615 |

And, we can also see how DRob’s top 20 differs from mine. Most of these functions are ones I use frequently, just not in my top 20. And, this is likely the case for DRob with the dissimilar functions in the table above.

ve\_functions\_top\_20\_tbl %>%

rowid\_to\_column(var = "rank") %>%

filter(!function\_name %in% local\_functions\_top\_20\_tbl$function\_name) %>%

knitr::kable()

| **rank** | **package** | **function\_name** | **n** | **pct** |
| --- | --- | --- | --- | --- |
| 6 | dplyr | group\_by | 62 | 0.0267934 |
| 8 | base | sum | 59 | 0.0254970 |
| 9 | dplyr | summarize | 41 | 0.0177182 |
| 10 | base | mean | 38 | 0.0164218 |
| 14 | dplyr | count | 31 | 0.0133967 |
| 15 | ggplot2 | geom\_point | 30 | 0.0129646 |
| 16 | ggplot2 | facet\_wrap | 30 | 0.0129646 |
| 17 | dplyr | inner\_join | 28 | 0.0121003 |
| 18 | dplyr | ungroup | 25 | 0.0108038 |
| 19 | dplyr | arrange | 25 | 0.0108038 |
| 20 | broom | tidy | 24 | 0.0103717 |

**Conclusions**

We are half-way on our quest to develop an optimal strategy to learn R. We picked a great candidate in DRob to learn from. He’s a tidyverse afficianado, a master data scientist, and he has a large sample of blog posts spanning multiple years to aggregate and analyze.

We learned a bunch of cool things related to our hypothesis. To recap, we hypothesized that (1) you don’t need to learn everything to become proficient at R, and (2) we can develop a strategic plan by learning from a master data scientist. We have not proven the second point yet, but the first we can confirm with confidence given that 88 functions created 80% of the output on DRob’s blog.

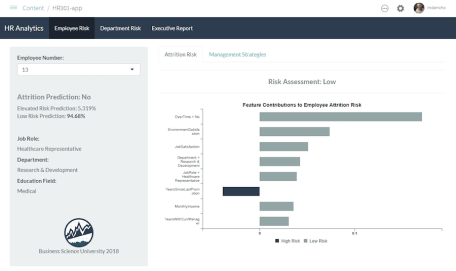
In the next post we’ll dive deeper into the list of top functions generated to see if we can develop a program to go from zero experience in R to intermediate status quickly! **If our 80/20 theory is right, we should be able to go from zero to intermediate in just a couple weeks by focusing on the most critical functions**.

**Business Science University**

**Interested in learning data science for business?** We are developing our Virtual Workshop on analyzing a major problem affecting companies: **employee attrition (turnover)**. Don’t wait… You’ll get an **early-bird discount** on the first Virtual Workshop.

**Virtual Workshop: Predicting Employee Attrition**

Did you know that **an organization that loses 200 high performing employees per year is essentially losing $15M/year in lost productivity**? Many organizations don’t realize this because it’s an indirect cost. It goes unnoticed. What if you could use data science to predict and explain turnover in a way that managers could make better decisions and executives would see results? You will learn the tools to do so in our Virtual Workshop. Here’s an example of a Shiny app you will create.



Shiny App That Predicts Attrition and Recommends Management Strategies, Taught in HR 301

Our first Virtual Workshop teaches you how to solve this employee attrition problem using data science. We are building four courses that are fully integrated into a single virtual workshop:

* HR 201: Predicting Employee Attrition with h2o and lime
* HR 301: Building A Shiny Web Application
* HR 302: Data Story Telling With RMarkdown Reports and Presentations
* HR 303: Building An R Package For Your Organization, tidyattrition

The Virtual Workshop is intended for **intermediate and advanced R users**. It’s code intensive, but also teaches you fundamentals of data science consulting including CRISP-DM and the Business Science Problem Framework. **The content bridges the gap between data science and the business, making you even more effective and improving your organization in the process.**