Here’s the problem: have some data with nested time series. Lots of them. It’s  
like there’s many, many little datasets inside my data. There are too many  
groups to plot all of the time series at once, so just want to preview a  
handful of them.

For a working example, suppose we want to visualize the top 50 American female  
baby names over time. start by adding up the total number of births for each  
name, finding the overall top 50 most populous names, and then keeping just the  
time series from those top names.

library(ggplot2)

library(dplyr, warn.conflicts = FALSE)

babynames <- babynames::babynames %>%

filter(sex == "F")

top50 <- babynames %>%

group\_by(name) %>%

summarise(total = sum(n)) %>%

top\_n(50, total)

# keep just rows in babynames that match a row in top50

top\_names <- babynames %>%

semi\_join(top50, by = "name")

Hmm, so what does this look like?

ggplot(top\_names) +

aes(x = year, y = n) +

geom\_line() +

facet\_wrap("name")

An illegible plot because too many facets are plotted

Aaack, I can’t read anything! Can’t I just see a few of them?

This is a problem I face frequently, so frequently that I wrote a helper  
function to handle this problem: sample\_n\_of(). This is not a very clever  
name, but it works. Below I call the function from my personal R package  
and plot just the data from four names.

# For reproducible blogging

set.seed(20180524)

top\_names %>%

tjmisc::sample\_n\_of(4, name) %>%

ggplot() +

aes(x = year, y = n) +

geom\_line() +

facet\_wrap("name")

A plot with four faceted timeseries

In this post, I walk through how this function works. It’s not very  
complicated: It relies on some light tidy evaluation plus one obscure dplyr  
function.

**Working through the function**

As usual, let’s start by sketching out the function we want to write:

sample\_n\_of <- function(data, size, ...) {

# quote the dots

dots <- quos(...)

# ...now make things happen...

}

where size are the number of groups to sample and ... are the columns names  
that define the groups. We use quos(...) to capture and quote those column  
names.

Code Chunks - set-na-where-nonstandard-evaluation-use-case

I had to deal with a file with some eyetracking data from a sequence-learning experiment. The eyetracker records the participant’s gaze location at a rate of 60 frames per second—except for this weird file which wrote out ~80 frames each second. In this kind of data, we have one row per eyetracking sample, and each sample records a timestamp and the gaze location :eyes: on the computer screen at each timestamp. In this particular dataset, we have x and y gaze coordinates in pixels (both eyes averaged together, GazeX and GazeY) or in screen proportions (for each eye in the EyeCoord columns.)

library(dplyr)

library(ggplot2)

library(rlang)

df <- system.file("test-gaze.csv", package = "fillgaze") %>%

readr::read\_csv() %>%

mutate(Time = Time - min(Time)) %>%

select(Time:REyeCoordY) %>%

round(3) %>%

mutate\_at(vars(Time), round, 1) %>%

mutate\_at(vars(GazeX, GazeY), round, 0)

df

#> # A tibble: 14,823 x 8

#> Time Trial GazeX GazeY LEyeCoordX LEyeCoordY REyeCoordX REyeCoordY

#> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>

#> 1 0 1 1176 643 0.659 0.589 0.566 0.602

#> 2 3.5 1 -1920 -1080 -1 -1 -1 -1

#> 3 20.2 1 -1920 -1080 -1 -1 -1 -1

#> 4 36.8 1 1184 648 0.664 0.593 0.57 0.606

#> 5 40 1 1225 617 0.685 0.564 0.591 0.579

#> 6 56.7 1 -1920 -1080 -1 -1 -1 -1

#> 7 73.4 1 1188 641 0.665 0.587 0.572 0.6

#> 8 76.6 1 1204 621 0.674 0.568 0.58 0.582

#> 9 93.3 1 -1920 -1080 -1 -1 -1 -1

#> 10 110. 1 1189 665 0.666 0.609 0.572 0.622

#> # ... with 14,813 more rows

In this particular eyetracking setup, offscreen looks are coded as negative gaze coordinates, and what’s extra weird here is that every second or third point is incorrectly placed offscreen. We see that in the frequent -1920 values in GazeX. Plotting the first few x and y pixel locations shows the pattern as well.

p <- ggplot(head(df, 40)) +

aes(x = Time) +

geom\_hline(yintercept = 0, size = 2, color = "white") +

geom\_point(aes(y = GazeX, color = "GazeX")) +

geom\_point(aes(y = GazeY, color = "GazeY")) +

labs(

x = "Time (ms)",

y = "Screen location (pixels)",

color = "Variable"

)

p +

annotate(

"text", x = 50, y = -200,

label = "offscreen", color = "grey20"

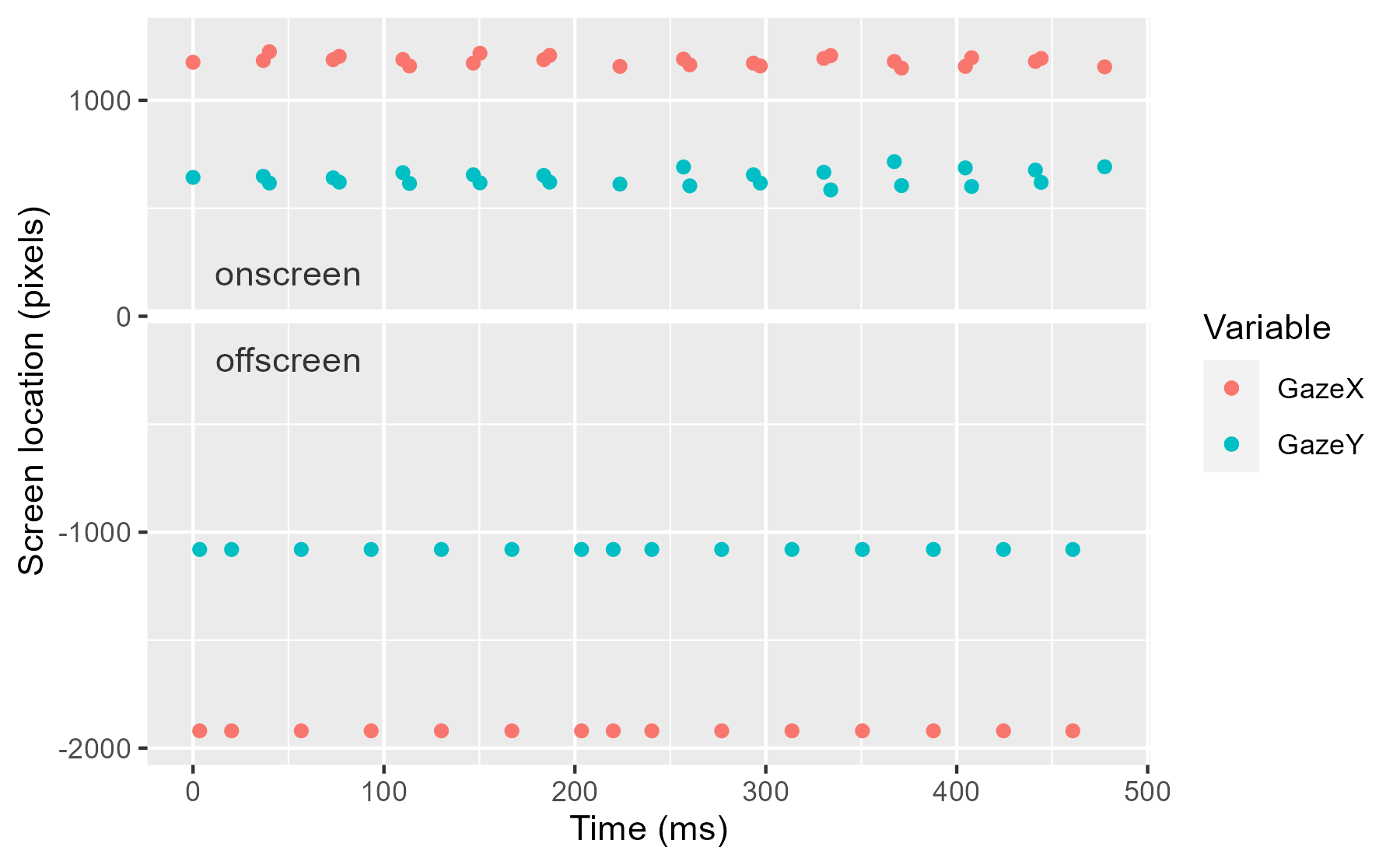
) +

annotate(

"text", x = 50, y = 200,

label = "onscreen", color = "grey20"

)



It is physiologically impossible for a person’s gaze to oscillate so quickly and with such magnitude (the gaze is tracked on a large screen display), so obviously something weird was going on with the experiment software.

This file motivated me to develop a general purpose package for interpolating missing data in eyetracking experiments. This package was always something I wanted to do, and this file moved it from the someday list to the today list.

## A function to recode values in many columns as NAPermalink

The first step in handling this problematic dataset is to convert the offscreen values into actual missing (NA) values). Because we have several columns of data, I wanted a succinct way to recode values in multiple columns into NA values.

First, we sketch out the code we want to write when we’re done.

set\_na\_where <- function(data, ...) {

# do things

}

set\_na\_where(

data = df,

GazeX = GazeX < -500 | 2200 < GazeX,

GazeY = GazeY < -200 | 1200 < GazeY

)

That is, after specifying the data, we list off an arbitrary number of column names, and with each name, we provide a rule to determine whether a value in that column is offscreen and should be set to NA. For example, we want every value in GazeX where GazeX < -500 or 2299 < GazeX is TRUE to be replaced with NA.

Lines of computer code are magic spells: We say the incantations and things happen around us. Put more formally, the code contains expressions that are evaluated in an environment.

hey <- "Hello!"

message(hey)

#> Hello!

exists("x")

#> [1] FALSE

x <- pi ^ 2

exists("x")

#> [1] TRUE

print(x)

#> [1] 9.869604

stop("what are you doing?")

#> Error in eval(expr, envir, enclos): what are you doing?

In our function signature, function(data, ...), the expressions are collected in the special “dots” argument (...). In normal circumstances, we can view the contents of the dots by storing them in a list. Consider:

hello\_dots <- function(...) {

str(list(...))

}

hello\_dots(x = pi, y = 1:10, z = NA)

#> List of 3

#> $ x: num 3.14

#> $ y: int [1:10] 1 2 3 4 5 6 7 8 9 10

#> $ z: logi NA

But we not passing in regular data, but expressions that need to be evaluated in a particular location. Below the magic words are uttered and we get an error because they mention things that do not exist in the current environment.

hello\_dots(GazeX = GazeX < -500 | 2200 < GazeX)

#> Error in str(list(...)): object 'GazeX' not found

What we need to do is prevent these words from being uttered until the time and place are right. **Nonstandard evaluation is a way of bottling up magic spells and changing how or where they are cast**—sometimes we even change the magic words themselves. We bottle up or capture the expressions given by the user by quoting them. quo() quotes a single expression, and quos() (plural) will quote a list of expressions. Below, we capture the expressions stored in the dots :speech\_balloon: and then make sure that their names match column names in the dataframe.

set\_na\_where <- function(data, ...) {

dots <- quos(...)

stopifnot(names(dots) %in% names(data), !anyDuplicated(names(dots)))

dots

# more to come

}

spells <- set\_na\_where(

data = df,

GazeX = GazeX < -500 | 2200 < GazeX,

GazeY = GazeY < -200 | 1200 < GazeY

)

spells

#> <list\_of<quosure>>

#>

#> $GazeX

#> <quosure>

#> expr: ^GazeX < -500 | 2200 < GazeX

#> env: 0000000017704B48

#>

#> $GazeY

#> <quosure>

#> expr: ^GazeY < -200 | 1200 < GazeY

#> env: 0000000017704B48

I call these results spells because it just contains the expressions stored as data. We can interrogate these results like data. We can query the names of the stored data, and we can extract values (the quoted expressions).

names(spells)

#> [1] "GazeX" "GazeY"

spells[[1]]

#> <quosure>

#> expr: ^GazeX < -500 | 2200 < GazeX

#> env: 0000000017704B48

### Casting spellsPermalink

We can cast a spell by evaluating an expression. To keep the incantation from fizzling out, we specify that we want to evaluate the expression inside of the dataframe. The function eval\_tidy(expr, data) lets us do just that: evaluate an expression expr inside of some data.

# Evaluate the first expression inside of the data

xs\_to\_set\_na <- eval\_tidy(spells[[1]], data = df)

# Just the first few bc there are 10000+ values

xs\_to\_set\_na[1:20]

#> [1] FALSE TRUE TRUE FALSE FALSE TRUE FALSE FALSE TRUE FALSE FALSE TRUE

#> [13] FALSE FALSE TRUE FALSE FALSE TRUE TRUE FALSE

In fact, we can evaluate them all at once with by applying eval\_tidy() on each listed expression.

to\_set\_na <- lapply(spells, eval\_tidy, data = df)

str(to\_set\_na)

#> List of 2

#> $ GazeX: logi [1:14823] FALSE TRUE TRUE FALSE FALSE TRUE ...

#> $ GazeY: logi [1:14823] FALSE TRUE TRUE FALSE FALSE TRUE ...

### Finishing touchesPermalink

Now, the rest of the function is straightforward. Evaluate each NA-rule on the named columns, and then set each row where the rule is TRUE to NA.

set\_na\_where <- function(data, ...) {

dots <- quos(...)

stopifnot(names(dots) %in% names(data), !anyDuplicated(names(dots)))

set\_to\_na <- lapply(dots, eval\_tidy, data = data)

for (col in names(set\_to\_na)) {

data[set\_to\_na[[col]], col] <- NA

}

data

}

results <- set\_na\_where(

data = df,

GazeX = GazeX < -500 | 2200 < GazeX,

GazeY = GazeY < -200 | 1200 < GazeY

)

results

#> # A tibble: 14,823 x 8

#> Time Trial GazeX GazeY LEyeCoordX LEyeCoordY REyeCoordX REyeCoordY

#> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>

#> 1 0 1 1176 643 0.659 0.589 0.566 0.602

#> 2 3.5 1 NA NA -1 -1 -1 -1

#> 3 20.2 1 NA NA -1 -1 -1 -1

#> 4 36.8 1 1184 648 0.664 0.593 0.57 0.606

#> 5 40 1 1225 617 0.685 0.564 0.591 0.579

#> 6 56.7 1 NA NA -1 -1 -1 -1

#> 7 73.4 1 1188 641 0.665 0.587 0.572 0.6

#> 8 76.6 1 1204 621 0.674 0.568 0.58 0.582

#> 9 93.3 1 NA NA -1 -1 -1 -1

#> 10 110. 1 1189 665 0.666 0.609 0.572 0.622

#> # ... with 14,813 more rows

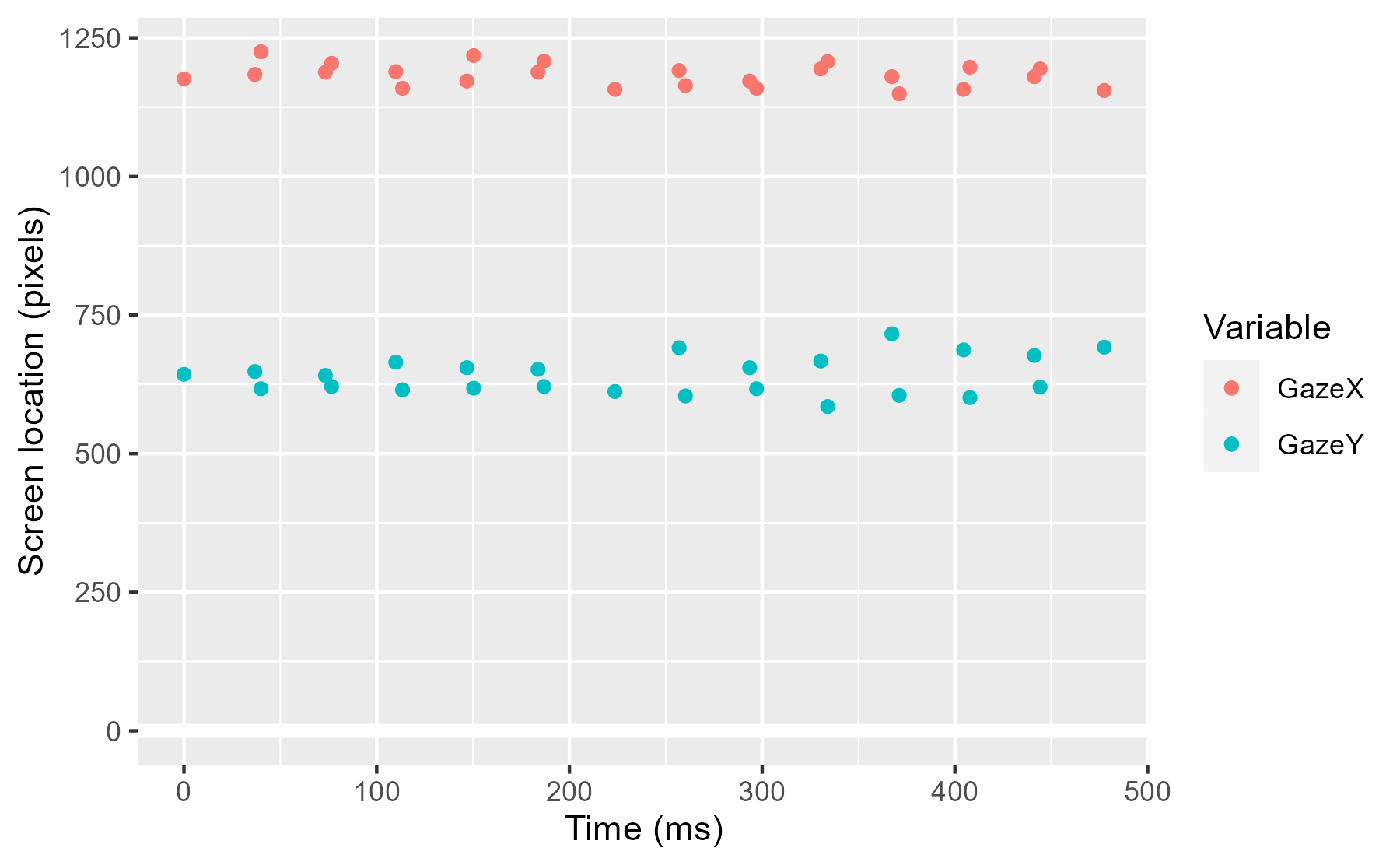
Visually, we can see that the offscreen values are no longer plotted. Plus, we are told that our data now has missing values.

# `plot %+% data`: replace the data in `plot` with `data`

p %+% head(results, 40)

#> Warning: Removed 15 rows containing missing values (geom\_point).

#> Warning: Removed 15 rows containing missing values (geom\_point).



One of the quirks about some eyetracking data is that during a blink, sometimes the device will record the x location but not the y location. (I think this happens because blinks move vertically so the horizontal detail can still be inferred in a half-closed eye.) This effect shows up in the data when there are more NA values for the y values than for the x values:

count\_na <- function(data, ...) {

subset <- select(data, ...)

lapply(subset, function(xs) sum(is.na(xs)))

}

count\_na(results, GazeX, GazeY)

#> $GazeX

#> [1] 2808

#>

#> $GazeY

#> [1] 3064

We can equalize these counts by running the function a second time with new rules.

df %>%

set\_na\_where(

GazeX = GazeX < -500 | 2200 < GazeX,

GazeY = GazeY < -200 | 1200 < GazeY

) %>%

set\_na\_where(

GazeX = is.na(GazeY),

GazeY = is.na(GazeX)

) %>%

count\_na(GazeX, GazeY)

#> $GazeX

#> [1] 3069

#>

#> $GazeY

#> [1] 3069

Alternatively, we can do this all at once by using the same NA-filtering rule on GazeX and GazeY.

df %>%

set\_na\_where(

GazeX = GazeX < -500 | 2200 < GazeX | GazeY < -200 | 1200 < GazeY,

GazeY = GazeX < -500 | 2200 < GazeX | GazeY < -200 | 1200 < GazeY

) %>%

count\_na(GazeX, GazeY)

#> $GazeX

#> [1] 3069

#>

#> $GazeY

#> [1] 3069

These last examples, where we compare different rules, showcases how nonstandard evaluation lets us write in a very succinct and convenient manner and quickly iterate over possible rules. Works like magic, indeed.

For interactive testing, suppose our dataset are the time series from the top 50  
names and we want data from a sample of 5 names. In this case, the values for  
the arguments would be:

data <- top\_names

size <- 5

dots <- quos(name)

A natural way to think about this problem is that we want to sample subgroups of  
the dataframe. First, we create a grouped version of the dataframe using  
group\_by(). The function group\_by() also takes a ... argument where the  
dots are typically names of columns in the dataframe. We want to take the  
names inside of our dots, unquote them and plug them in to where the ...  
goes in group\_by(). This is what the tidy evaluation world calls  
*splicing*.

Think of splicing as doing this:

Library(dplyr)

# Demo function that counts the number of arguments in the dots

count\_args <- function(...) length(quos(...))

example\_dots <- quos(var1, var2, var2)

# Splicing turns the first form into the second one

count\_args(!!! example\_dots)

#> [1] 3

count\_args(var1, var2, var2)

#> [1] 3

So, we create a grouped dataframe by splicing our dots into the group\_by()  
function.

grouped <- data %>%

group\_by(!!! dots)

There is a helper function buried in dplyr called group\_indices() which  
returns the grouping index for each row in a grouped dataframe.

grouped %>%

tibble::add\_column(group\_index = group\_indices(grouped))

#> # A tibble: 6,407 x 6

#> # Groups: name [50]

#> year sex name n prop group\_index

#>

#> 1 1880 F Mary 7065 0.0724 33

#> 2 1880 F Anna 2604 0.0267 4

#> 3 1880 F Emma 2003 0.0205 19

#> 4 1880 F Elizabeth 1939 0.0199 17

#> 5 1880 F Margaret 1578 0.0162 32

#> 6 1880 F Sarah 1288 0.0132 45

#> 7 1880 F Laura 1012 0.0104 29

#> 8 1880 F Catherine 688 0.00705 11

#> 9 1880 F Helen 636 0.00652 21

#> 10 1880 F Frances 605 0.00620 20

#> # ... with 6,397 more rows

We can randomly sample five of the group indices and keep the rows for just  
those groups.

unique\_groups <- unique(group\_indices(grouped))

sampled\_groups <- sample(unique\_groups, size)

sampled\_groups

#> [1] 4 25 43 20 21

subset\_of\_the\_data <- data %>%

filter(group\_indices(grouped) %in% sampled\_groups)

subset\_of\_the\_data

#> # A tibble: 674 x 5

#> year sex name n prop

#>

#> 1 1880 F Anna 2604 0.0267

#> 2 1880 F Helen 636 0.00652

#> 3 1880 F Frances 605 0.00620

#> 4 1880 F Samantha 21 0.000215

#> 5 1881 F Anna 2698 0.0273

#> 6 1881 F Helen 612 0.00619

#> 7 1881 F Frances 586 0.00593

#> 8 1881 F Samantha 12 0.000121

#> 9 1881 F Karen 6 0.0000607

#> 10 1882 F Anna 3143 0.0272

#> # ... with 664 more rows

# Confirm that only five names are in the dataset

subset\_of\_the\_data %>%

distinct(name)

#> # A tibble: 5 x 1

#> name

#>

#> 1 Anna

#> 2 Helen

#> 3 Frances

#> 4 Samantha

#> 5 Karen

Putting these steps together, we get:

sample\_n\_of <- function(data, size, ...) {

dots <- quos(...)

group\_ids <- data %>%

group\_by(!!! dots) %>%

group\_indices()

sampled\_groups <- sample(unique(group\_ids), size)

data %>%

filter(group\_ids %in% sampled\_groups)

}

We can test that the function works as we might expect. Sampling 10 names  
returns the data for 10 names.

ten\_names <- top\_names %>%

sample\_n\_of(10, name) %>%

print()

#> # A tibble: 1,326 x 5

#> year sex name n prop

#>

#> 1 1880 F Sarah 1288 0.0132

#> 2 1880 F Frances 605 0.00620

#> 3 1880 F Rachel 166 0.00170

#> 4 1880 F Samantha 21 0.000215

#> 5 1880 F Deborah 12 0.000123

#> 6 1880 F Shirley 8 0.0000820

#> 7 1880 F Carol 7 0.0000717

#> 8 1880 F Jessica 7 0.0000717

#> 9 1881 F Sarah 1226 0.0124

#> 10 1881 F Frances 586 0.00593

#> # ... with 1,316 more rows

ten\_names %>%

distinct(name)

#> # A tibble: 10 x 1

#> name

#>

#> 1 Sarah

#> 2 Frances

#> 3 Rachel

#> 4 Samantha

#> 5 Deborah

#> 6 Shirley

#> 7 Carol

#> 8 Jessica

#> 9 Patricia

#> 10 Sharon

We can sample based on multiple columns too. Ten combinations of names and years  
should return just ten rows.

top\_names %>%

sample\_n\_of(10, name, year)

#> # A tibble: 10 x 5

#> year sex name n prop

#>

#> 1 1907 F Jessica 17 0.0000504

#> 2 1932 F Catherine 5446 0.00492

#> 3 1951 F Nicole 94 0.0000509

#> 4 1953 F Janet 17761 0.00921

#> 5 1970 F Sharon 9174 0.00501

#> 6 1983 F Melissa 23473 0.0131

#> 7 1989 F Brenda 2270 0.00114

#> 8 1989 F Pamela 1334 0.000670

#> 9 1994 F Samantha 22817 0.0117

#> 10 2014 F Kimberly 2891 0.00148

**Next steps**

There are a few tweaks we could make to this function. For example, in my  
package’s version, I warn the user when the number of groups is too large.

too\_many <- top\_names %>%

tjmisc::sample\_n\_of(100, name)

#> Warning: Sample size (100) is larger than number of groups (50). Using size

#> = 50.

My version also randomly samples *n* of the rows when there are no grouping  
variables provided.

top\_names %>%

tjmisc::sample\_n\_of(2)

#> # A tibble: 2 x 5

#> year sex name n prop

#>

#> 1 1934 F Stephanie 128 0.000118

#> 2 2007 F Mary 3674 0.00174

One open question is how to handle data that’s already grouped. The function we  
wrote above fails.

top\_names %>%

group\_by(name) %>%

sample\_n\_of(2, year)

#> Error in filter\_impl(.data, quo): Result must have length 136, not 6407

Is this a problem?

Here I think failure is okay because what do we think should happen? It’s not  
obvious. It should randomly choose 2 of the years for each name.  
Should it be the same two years? Then this should be fine.

top\_names %>%

sample\_n\_of(2, year)

#> # A tibble: 100 x 5

#> year sex name n prop

#>

#> 1 1970 F Jennifer 46160 0.0252

#> 2 1970 F Lisa 38965 0.0213

#> 3 1970 F Kimberly 34141 0.0186

#> 4 1970 F Michelle 34053 0.0186

#> 5 1970 F Amy 25212 0.0138

#> 6 1970 F Angela 24926 0.0136

#> 7 1970 F Melissa 23742 0.0130

#> 8 1970 F Mary 19204 0.0105

#> 9 1970 F Karen 16701 0.00912

#> 10 1970 F Laura 16497 0.00901

#> # ... with 90 more rows

Or, should those two years be randomly selected for each name? Then, we should  
let do() handle that. do() takes some code that returns a dataframe, applies  
it to each group, and returns the combined result.

top\_names %>%

group\_by(name) %>%

do(sample\_n\_of(., 2, year))

#> # A tibble: 100 x 5

#> # Groups: name [50]

#> year sex name n prop

#>

#> 1 1913 F Amanda 346 0.000528

#> 2 1953 F Amanda 428 0.000222

#> 3 1899 F Amy 281 0.00114

#> 4 1964 F Amy 9579 0.00489

#> 5 1916 F Angela 715 0.000659

#> 6 2005 F Angela 2893 0.00143

#> 7 1999 F Anna 9092 0.00467

#> 8 2011 F Anna 5649 0.00292

#> 9 1952 F Ashley 24 0.0000126

#> 10 2006 F Ashley 12340 0.00591

#> # ... with 90 more rows

I think raising an error and forcing the user to clarify their code is a better  
than choosing one of these options and not doing what the user expects.