Managing and Manipulating Data Using R

Lecture 3

Ozan Jaquette

1. Introduction/logistics

2. Factors

3. FACTORS

4. Labeling variables

5. Pipes

6. Creating variables using mutate

Introduction/logistics

Libraries we will use today

```
library(tidyverse)
#> -- Attaching packages -------------------------------------------------------
#> v ggplot2 3.0.0     v purrr   0.2.5
#> v tibble  1.4.2     v dplyr   0.7.6
#> v tidyr   0.8.1     v stringr 1.3.1
#> v readr   1.1.1     v forcats 0.3.0
#> -- Conflicts ----------------------------------------------------------------
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()
```

Data we will use today

Data on off-campus recruiting events by public universities

```r
rm(list = ls()) # remove all objects

#load dataset with one obs per recruiting event
load("../../data/recruiting/recruit_event_somevars.Rdata")

#load dataset with one obs per high school
load("../../data/recruiting/recruit_school_somevars.Rdata")

load("../../data/prospect_list/western_washington_college_board_list.RData")
```

Object \hlgc{df_event}

- One observation per university, recruiting event

Object \hlgc{df_event}

- One observation per high school (visited and non-visited)

Factors

Review data types and structures

Review data types

Primary **data types** in R:

- numeric (integer & double)
- character
- logical

R CODE CHUNK WITH EXAMPLES

Review data structures: vectors

Primary **data structures** in R are **vectors** and **lists**

A **vector** is a collection of values

- each value in a vector is an **element**
- all elements within vector must have same **data type**

```
a <- c(1,2,3)
a
#> [1] 1 2 3
str(a)
#>  num [1:3] 1 2 3
```

You can assign **names** to elements of a vector, thereby creating a **named vector**

```
b <- c(v1=1,v2=2,v3=3)
b
#> v1 v2 v3
#>  1  2  3
str(b)
#>  Named num [1:3] 1 2 3
#>  - attr(*, "names")= chr [1:3] "v1" "v2" "v3"
```

Review data structures: lists

Like vectors, **lists** are objects that contain **elements**; However, **data type** can differ across elements within a list; an element of a list can be another list

Examples of lists:

```
list_a <- list(1,2,"apple")
str(list_a)
#> List of 3
#>  $ : num 1
#>  $ : num 2
#>  $ : chr "apple"
list_b <- list(1, c("apple", "orange"), list(1, 2, 3))
str(list_b)
#> List of 3
#>  $ : num 1
#>  $ : chr [1:2] "apple" "orange"
#>  $ :List of 3
#>   ..$ : num 1
#>   ..$ : num 2
#>   ..$ : num 3
```

Review data structures: lists

Like vectors, elements within a list can be named, thereby creating a **named list**

```
str(list_b) # not named
#> List of 3
#>  $ : num 1
#>  $ : chr [1:2] "apple" "orange"
#>  $ :List of 3
#>   ..$ : num 1
#>   ..$ : num 2
#>   ..$ : num 3

list_c <- list(v1=1, v2=c("apple", "orange"), v3=list(1, 2, 3))
str(list_c) # named
#> List of 3
#>  $ v1: num 1
#>  $ v2: chr [1:2] "apple" "orange"
#>  $ v3:List of 3
#>   ..$ : num 1
#>   ..$ : num 2
#>   ..$ : num 3
```

Review data structures: a data frame is a list

A **data frame** is a list with the following characteristics:

- All the elements must be **vectors** with the same **length**
- Data frames are **augmented lists** because they have additional **attributes** [described later]

```
list_d <- list(col_a = c(1,2,3), col_b = c(4,5,6), col_c = c(7,8,9))
typeof(list_d)
#> [1] "list"
str(list_d)
#> List of 3
#>  $ col_a: num [1:3] 1 2 3
#>  $ col_b: num [1:3] 4 5 6
#>  $ col_c: num [1:3] 7 8 9

df_a <- data.frame(col_a = c(1,2,3), col_b = c(4,5,6), col_c = c(7,8,9))
typeof(df_a)
#> [1] "list"
str(df_a)
#> 'data.frame':    3 obs. of  3 variables:
#>  $ col_a: num  1 2 3
#>  $ col_b: num  4 5 6
#>  $ col_c: num  7 8 9
```

Attributes and augmented vectors

Atomic vectors versus augmented vectors

Atomic vectors [our focus so far] - (See figure) - I think of atomic vectors as "just the data" - Atomic vectors are the building blocks for augmented vectors

Augmented vectors

- **Augmented vectors** are atomic vectors with additional **atributes** attached

Attributes

- **Attributes** are additional "metadata" that can be attached to any object (e.g., vector or list)
- Important attributes in R:
  - **Names**: name the elements of a vector (e.g., variable names)
  - **Class**: How object should be treated by object oriented programming language [discussed below]

Main takaway:

- Augmented vectors are atomic vectors (just the data) with additional attributes attached

Attributes in vectors

```
vector1 <- c(1,2,3,4)
vector1
#> [1] 1 2 3 4
attributes(vector1)
#> NULL

vector2 <- c(a = 1, b= 2, c= 3, d = 4)
vector2
#> a b c d
#> 1 2 3 4
attributes(vector2)
#> $names
#> [1] "a" "b" "c" "d"
```

Attributes in lists

```
list1 <- list(c(1,2,3), c(4,5,6))
str(list1)
#> List of 2
#>  $ : num [1:3] 1 2 3
#>  $ : num [1:3] 4 5 6
attributes(list1)
#> NULL

list2 <- list(col_a = c(1,2,3), col_b = c(4,5,6))
str(list2)
#> List of 2
#>  $ col_a: num [1:3] 1 2 3
#>  $ col_b: num [1:3] 4 5 6
attributes(list2)
#> $names
#> [1] "col_a" "col_b"

list3 <- data.frame(col_a = c(1,2,3), col_b = c(4,5,6))
str(list3)
#> 'data.frame':    3 obs. of  2 variables:
#>  $ col_a: num  1 2 3
#>  $ col_b: num  4 5 6
attributes(list3)
#> $names
#> [1] "col_a" "col_b"
#>
#> $class
```

Object class

```
vector1 <- c(1,2,3,4)
vector1
#> [1] 1 2 3 4
typeof(vector1)
#> [1] "double"
class(vector1)
#> [1] "numeric"
attributes(vector1)
#> NULL

vector2 <- c(a = 1, b= 2, c= 3, d = 4)
vector2
#> a b c d
#> 1 2 3 4
attributes(vector2)
#> $names
#> [1] "a" "b" "c" "d"
typeof(vector2)
#> [1] "double"
class(vector2)
#> [1] "numeric"
```

FACTORS

Factors

Factors are used to display categorical data (e.g., marital status)

- A factor is an **augmented vector** built by attaching a “levels” attribute to an (atomic) integer vectors

The `str()` function is useful for identifying which variables are factors. Let’s examine the factor variable `ethn_code`

```
typeof(wwlist$ethn_code)
#> [1] "integer"
class(wwlist$ethn_code)
#> [1] "factor"
str(wwlist$ethn_code)
#>  Factor w/ 11 levels "American Indian or Alaska Native",..: 8 11 11 8 11 8 8
```

Note that `ethn_code` has `type=integer` and `class=factor` because the variable has a “levels” attribute

```
attributes(wwlist$ethn_code)
```

Main takeaway:

- The underlying data are integers but the levels attribute is used to display the data.

Working with factor variables

```
attributes(wwlist$ethn_code)
```

Refer to categories of a factor by the values of the level attribute rather than the underlying values of the variable

```
count(filter(wwlist,ethn_code==11))
#> # A tibble: 1 x 1
#>       n
#>   <int>
#> 1     0

count(filter(wwlist,ethn_code=="White"))
#> # A tibble: 1 x 1
#>        n
#>    <int>
#> 1 159680
```

If you want to refer to underlying values, then apply `as.integer()` function to the factor variable

```
count(filter(wwlist,as.integer(ethn_code)==11))
#> # A tibble: 1 x 1
#>        n
#>    <int>
#> 1 159680
```

How to identify the variable values associated with factor levels

MAYBE CUT THIS SLIDE IF YOU CAN'T DO THIS WITHOUT PIPES

```
wwlist %>% count(psat_range) %>% as_factor()
#> # A tibble: 8 x 2
#>   psat_range     n
#>   <fct>      <int>
#> 1 1030-1160  45708
#> 2 1030-1520  67192
#> 3 1170-1520  48982
#> 4 1270-1520   8348
#> 5 930-1160   17387
#> 6 930-1260   15660
#> 7 990-1260   27628
#> 8 <NA>       37491
```

```
count(filter(wwlist,as.integer(psat_range)==4))
#> # A tibble: 1 x 1
#>       n
#>   <int>
#> 1  8348
count(filter(wwlist,psat_range=="1270-1520"))
#> # A tibble: 1 x 1
#>       n
#>   <int>
#> 1  8348
```

Some in-class exercise involving factors

```
str(wwlist)
#> Classes 'tbl_df', 'tbl' and 'data.frame':    268396 obs. of  19 variables:
#>  $ receive_date    : Date, format: "2016-05-31" "2016-05-31" ...
#>  $ psat_range      : Factor w/ 7 levels "1030-1160","1030-1520",..: 5 4 7 3 7
#>  $ sat_range       : Factor w/ 3 levels "1030-1600","930-1600",..: NA NA NA N
#>  $ ap_range        : Factor w/ 2 levels "1 or higher",..: NA NA NA NA NA NA N
#>  $ gpa_b_aplus     : Factor w/ 1 level "x": 1 1 1 1 1 1 1 1 1 NA ...
#>  $ gpa_b_aplus_null: Factor w/ 1 level "x": NA NA NA NA NA NA NA NA NA NA ...
#>  $ gpa_bplus_aplus : Factor w/ 1 level "x": NA NA NA NA NA NA NA NA NA 1 ...
#>  $ state           : chr  "WA" "WA" "WA" "WA" ...
#>  $ zip             : chr  "98103-3528" "98030-7964" "98290-8659" "98105-0002"
#>  $ for_country     : chr  NA NA NA NA ...
#>  $ sex             : Factor w/ 3 levels "F","M","U": 2 1 2 1 1 2 2 1 2 2 ...
#>  $ hs_ceeb_code    : int  481112 480539 480391 481115 480585 481080 480118 48
#>  $ hs_name         : chr  "Ingraham High School" "Kentwood Senior High School
#>  $ hs_city         : chr  "Seattle" "Covington" "Everett" "Seattle" ...
#>  $ hs_state        : chr  "WA" "WA" "WA" "WA" ...
#>  $ hs_grad_date    : Date, format: "2018-06-01" "2017-06-01" ...
#>  $ ethn_code       : Factor w/ 11 levels "American Indian or Alaska Native",.
#>  $ homeschool      : Factor w/ 2 levels "N","Y": 1 1 1 1 1 1 1 1 1 1 ...
#>  $ firstgen        : Factor w/ 2 levels "N","Y": NA 1 1 1 NA 1 1 2 2 1 ...
```

Creating factors [from integer vectors]

Factors are just integer vectors with level attributes attached to them. So, to create a factor:

1. create a vector for the underlying data
2. create a vector that has level attributes
3. Attach levels to the data using the `factor()` function

```r
a1 <- c(1,1,1,0,1,1,0) #a vector of data
a2 <- c("zero","one") #a vector of labels

#attach labels to values
a3 <- factor(a1, labels = a2)
a3
#> [1] one  one  one  zero one  one  zero
#> Levels: zero one
str(a3)
#>  Factor w/ 2 levels "zero","one": 2 2 2 1 2 2 1
```

Note: By default, `factor()` function attached “zero” to the lowest value of vector `a1` because “zero” was the first element of vector `a2`

Creating factors [from integer vectors]

Let’s turn an integer variable into a factor variable in the `wwlist` data frame

Create integer version of `sex`

```
wwlist$sex_int <- as.integer(wwlist$sex)
str(wwlist$sex_int)
#>  int [1:268396] 2 1 2 1 1 2 2 1 2 2 ...
#wwlist %>% count(sex) %>% as_factor()
```

Assume we know that 1=female, 2=male, 3=unknown

Assign levels to values of integer variable

```
wwlist$sex_int <- factor(wwlist$sex_int, labels=c("female","male","unknown"))
str(wwlist$sex_int)
#>  Factor w/ 3 levels "female","male",..: 2 1 2 1 1 2 2 1 2 2 ...
str(wwlist$sex)
#>  Factor w/ 3 levels "F","M","U": 2 1 2 1 1 2 2 1 2 2 ...
```

Create factors [from string variables]

To create a factor variable from string variable

1. create a character vector containing underlying data
2. create a vector containing valid levels
3. Attach levels to the data using the `factor()` function

```r
#underlying data: months my fam is born
x1 <- c("Jan", "Aug", "Apr", "Mar")
#create vector with valid levels
month_levels <- c("Jan", "Feb", "Mar", "Apr", "May", "Jun",
  "Jul", "Aug", "Sep", "Oct", "Nov", "Dec")
#attach levels to data
x2 <- factor(x1, levels = month_levels)
```

Note how attributes differ

```r
str(x1)
#>  chr [1:4] "Jan" "Aug" "Apr" "Mar"
str(x2)
#>  Factor w/ 12 levels "Jan","Feb","Mar",..: 1 8 4 3
```

Sorting differs

```r
sort(x1)
#> [1] "Apr" "Aug" "Jan" "Mar"
sort(x2)
#> [1] Jan Mar Apr Aug
#> Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

Create factors [from string variables]

Let's create a character version of variable `sex` and then turn it into a factor

```
#Create character version of sex
wwlist$sex_char <- as.character(wwlist$sex)

#investigate character variable
str(wwlist$sex_char)
#>  chr [1:268396] "M" "F" "M" "F" "F" "M" "M" "F" "M" "M" "M" "F" "M" ...
table(wwlist$sex_char)
#> 
#>      F      M      U 
#> 147434 120470    492

#create new variable that assigns levels
sex_fac <- factor(wwlist$sex_char, levels = c("F","M","U"))
str(wwlist$sex_char)
#>  chr [1:268396] "M" "F" "M" "F" "F" "M" "M" "F" "M" "M" "M" "F" "M" ...
```

How the `levels` argument works when underlying data is character

- Matches value of underlying data to value of the level attribute
- Converts underlying data to integer, with level attribute attached

See chapter 15 of Wickham for more on factors (e.g., modifying factor order, modifying factor levels)

Substantial exercise on using/creating factors, using either df_school or df_event datasets

Labeling variables

Pipes

What are “pipes”, %>%

Pipes are a means of perfoming multiple steps in a single line of code

- Pipes are part of **tidyverse** suite of packages, not **base R**
- When writing code, the pipe symbol is `%>%`
- Basic flow of using pipes in code:
  - `object %>% some_function %>% some_function, ...`
- Pipes work from left to right:
  - The object/result from left of `%>%` pipe symbol is the input of function to the right of the `%>%` pipe symbol
  - In turn, the resulting output becomes the input of the function to the right of the next `%>%` pipe symbol

Do some tasks with and without pipes

Print data for “first-generation” prospects

```
filter(wwlist, firstgen == "Y")
wwlist %>% filter(firstgen == "Y")
```

Comparing the two approaches:

- In the “without pipes” approach, the object is the first argument `filter()` function
- In the “pipes” approach, you don’t specify the object as the first argument of `filter()`
  - Why? Because `%>%` “pipes” the object to the left of the `%>%` operator into the function to the right of the `%>%` operator

Main takeaway:

- Whenever you write code using pipes, functions to the right of a `%>%` pipe operator should not explicitly name the object that is the input to that function. Rather, the object to the left of the `%>%` pipe operator is automatically the input.

Do some tasks with and without pipes

Print data for “first-generation” prospects for selected variables [output omitted]

```
select(filter(wwlist, firstgen == "Y"), state, hs_city, ethn_code)

wwlist %>% filter(firstgen == "Y") %>% select(state, hs_city, ethn_code)
```

Comparing the two approaches:

- In the “without pipes” approach, code is written “inside out”
  - The first step in the task – identifying the object – is the innermost part of code
  - The last step in task – selecting variables to print – is the outermost part of code
- In “pipes” approach the left-to-right order of code matches how we think about the task
  - First, we start with an object ***and then*** (`%>%`) we use `filter()` to isolate first-gen students ***and then*** (`%>%`) we select which variables to print

Do some tasks with and without pipes

Count the number “first-generation” prospects from the state of Washington

```
count(filter(wwlist, firstgen == "Y", state == "WA"))
#> # A tibble: 1 x 1
#>       n
#>   <int>
#> 1 32428

wwlist %>% filter(firstgen == "Y", state == "WA") %>% count()
#> # A tibble: 1 x 1
#>       n
#>   <int>
#> 1 32428
```

Do some tasks with and without pipes [last example]

Create frequency table of `sex` for “first-generation” prospects from WA

```
wwlist_temp <- filter(wwlist, firstgen == "Y", state == "WA")
table(wwlist_temp$sex, useNA = "ifany")
#> 
#>     F     M     U 
#> 19080 13282    66 

wwlist %>% filter(firstgen == "Y", state == "WA") %>% count(sex)
#> # A tibble: 3 x 2
#>   sex       n
#>   <fct> <int>
#> 1 F     19080
#> 2 M     13282
#> 3 U        66
```

Comparison of two approaches

- without pipes, task requires multiple lines of code; this is quite common
  - first line creates object; second line analyzes object
- with pipes, task can be completed in one line of code and you aren’t left with objects you don’t care about

Note: the pipes approach above is a useful way to show the **values** associated with each **factor level** for factor variables

Student exercises with pipes

CREATE STUDENT EXERCISES

Creating variables using mutate

Our plan for learning how to create new variables

Recall that the `dplyr` package within the `tidyverse` provide a set of functions that can be described as “verbs”:

- **subsetting**, **sorting**, and **transforming**

What we’ve done	Where we’re going
Subsetting data	**Transforming data**
- `select()` variables	- `mutate()` creates new variables
- `filter()` observations	- `summarize()` calculates across rows
Sorting data	- `group_by()` to calculate across rows within groups
- `arrange()`	

Today

- we’ll use `mutate()` to create new variables based on calculations across columns within a row

Next week

- we’ll combine `mutate()` with `summarize()` and `group_by()` to create variables based on calculations across rows

Introduce `mutate()` function

`mutate()` creates new columns (variblaes) that are functions of existing columns

- \hlgc{mutate() is the **tidyverse** approach to creating variables, not the **Base R** approach
- \hlgc{mutate() works best with pipes `%>%`

We'll create variables from data frame `df_school` which has one observation for each high school

- Task: create pct of students on free/reduced lunch (output omitted)

```r
school_sml <- df_school %>% filter(school_type == "public") %>%
    select(ncessch, num_fr_lunch, total_students)

school_sml %>% mutate(pct_fr_lunch = num_fr_lunch/total_students)
```

Can combine `(select())` with pipes `%>%` to control which columns printed, so no need to create dataset with fewer columns. But let's create data frame with public high schools only

```r
school_pub <- df_school %>% filter(school_type == "public")
```

Introduce `mutate()` function

New variable not retained unless we **assign** `<-` it to an object (existing or new)

```
school_pub %>% mutate(pct_fr_lunch = num_fr_lunch/total_students)
names(school_pub)

school_pub_temp <- school_pub %>%
  mutate(pct_fr_lunch = num_fr_lunch/total_students)
names(school_pub_temp)
```

How to create percent free/reduced lunch in **Base R**

```
school_pub$pct_fr_lunch <- school_pub$num_fr_lunch/school_pub$total_students
```

`mutate()` can create multiple variables at once

```
school_pub %>%
  mutate(pct_fr_lunch = num_fr_lunch/total_students,
         pct_prof_math= num_prof_math/num_took_math) %>%
  select(num_fr_lunch, total_students, pct_fr_lunch,
         num_prof_math, num_took_math, pct_prof_math)
```

Student exercise using mutate()

Mutate to create indicator variables

We often create dichotomous (0/1) indicator variables of whether something happened (or whether something is TRUE)

- Variables that are of substantive interest to project
  - e.g., did student graduate from college
- Variables that help you investigate data, check quality
  - e.g., indicator of whether an observation is missing/non-missing for a particular variable

Let's conduct some investigations of `df_school`, which has one observation for each high school

Rename some variables (output omitted)

```r
str(df_school)
df_schoolv2 <- df_school %>%
  rename(
    visits_berkeley = visits_by_110635,
    visits_boulder = visits_by_126614,
    visits_bama = visits_by_100751,
    state_berkeley = inst_110635,
    state_boulder = inst_126614,
    state_bama = inst_100751)

names(df_schoolv2)
```

Creating indicators for `df_schoolv2` data frame

Create TRUE/FALSE indicator that median household income greater than $50,000

```
df_schoolv2_temp <- df_schoolv2 %>% mutate(incgt50k = avgmedian_inc_2564>50000)

df_schoolv2_temp %>% select(avgmedian_inc_2564, incgt50k) %>% head(n=3)
#> # A tibble: 3 x 2
#>   avgmedian_inc_2564 incgt50k
#>                <dbl> <lgl>
#> 1              76160 TRUE
#> 2              76160 TRUE
#> 3                 NA NA

df_schoolv2_temp %>% filter(is.na(avgmedian_inc_2564)) %>% count(incgt50k)
#> # A tibble: 1 x 2
#>   incgt50k     n
#>   <lgl>    <int>
#> 1 NA         624
```

Important takeaway:

- Variable created by `mutate()` equals `NA` for obs if input variable to `mutate()` is missing for that obs. This is a good thing!

Creating indicators for `df_schoolv2` data frame

Create TRUE/FALSE indicator that school is less than 50 percent white

```
df_schoolv2_temp <- df_schoolv2 %>% mutate(lt50pctwhite = pct_white<50)
df_schoolv2_temp %>% select(pct_white,lt50pctwhite) %>% head(n=3)
#> # A tibble: 3 x 2
#>   pct_white lt50pctwhite
#>       <dbl> <lgl>
#> 1      11.8 TRUE
#> 2       0   TRUE
#> 3       0   TRUE
str(df_schoolv2_temp$lt50pctwhite)
#>  logi [1:21301] TRUE TRUE TRUE TRUE TRUE TRUE ...
```

Create 0/1 integer indicator rather than logical indicator

```
df_schoolv2_temp <- df_schoolv2 %>% mutate(lt50pctwhite = as.integer(pct_white<5
df_schoolv2_temp %>% select(pct_white,lt50pctwhite) %>% head(n=3)
#> # A tibble: 3 x 2
#>   pct_white lt50pctwhite
#>       <dbl>        <int>
#> 1      11.8            1
#> 2       0              1
#> 3       0              1
str(df_schoolv2_temp$lt50pctwhite)
#>  int [1:21301] 1 1 1 1 1 1 1 1 1 1 ...
```

Student exercises

0/1 indicators of whether school received visit from each university

```
df_schoolv2 %>% count(visits_berkeley)
#> # A tibble: 4 x 2
#>   visits_berkeley     n
#>             <int> <int>
#> 1               0 20732
#> 2               1   528
#> 3               2    36
#> 4               3     5
df_schoolv2_temp <- df_schoolv2 %>% mutate(yesvis_berkeley = visits_berkeley>0)

df_schoolv2_temp %>% filter(visits_berkeley>0) %>% select(visits_berkeley,yesvis
#> # A tibble: 569 x 2
#>    visits_berkeley yesvis_berkeley
#>              <int> <lgl>
#>  1               2 TRUE
#>  2               2 TRUE
#>  3               2 TRUE
#>  4               1 TRUE
#>  5               1 TRUE
#>  6               1 TRUE
#>  7               1 TRUE
#>  8               1 TRUE
#>  9               1 TRUE
#> 10               1 TRUE
#> # ... with 559 more rows
df_schoolv2_temp %>% filter(visits_berkeley==0) %>% select(visits_berkeley,yesvis
```

Investigating `wwlist` data frame

?MAYBE CUT UNTIL NEXT LECTURE AND HAVE SECTION ON USING DESCRIPTIVE STATS TO INVESTIGATE DATA?

Let's conduct some investigations of `wwlist`, which is frankly a pretty weird dataset!

- When conducting investigations, really important to be careful about missing values

```
str(wwlist)
```

Variable `receive_date` indicates date prospect list data received from College Board

```
wwlist %>% count(receive_date)
#> # A tibble: 15 x 2
#>    receive_date     n
#>    <date>       <int>
#>  1 2016-05-31   50975
#>  2 2016-06-01   23195
#>  3 2016-06-02    4710
#>  4 2016-08-02    3929
#>  5 2016-08-03    5650
#>  6 2016-11-09    1178
#>  7 2017-01-23    1105
#>  8 2017-05-05   73430
#>  9 2017-06-05    3058
#> 10 2017-08-10   11396
#> 11 2017-10-11     827
```

other

```r
#num_took_math, num_prof_math
ww_narrow <- wwlist %>% select(psat_range, contains("gpa"), state, zip, for_coun
load("../../data/recruiting/recruit_event_somevars.Rdata")
```