An Intro to purrr

purrr: Functional programming in R

This presentation was created by Jennifer Thompson in 2017 for R-Ladies Nashville, I've modified it for us today

Setup

First we'll load the packages we need today

```
## -- Load R libraries ----------------------------------------------------
suppressPackageStartupMessages(library(purrr)) ## obvs
suppressPackageStartupMessages(library(dplyr)) ## for data management
suppressPackageStartupMessages(library(tidyr)) ## for data management
  ## Note: If you have the tidyverse package installed, library(tidyverse) will
  ## load purrr along with several other core packages, including dplyr & tidyr
suppressPackageStartupMessages(library(stringr)) ## for string manipulation
suppressPackageStartupMessages(library(ggplot2)) ## for plotting
suppressPackageStartupMessages(library(viridis)) ## for lovely color scales
```

Iteration: A Definition

Doing the same* thing to a bunch of things.

*ish

But We Have Ways to Do That Already, Right?

Let's try for a toy data set with 100 samples, each with 6 recorded variables.

```
load('toy_purrr.Rdata')
head(toy.data)
```

```
##            X1        X2         X3 X4       X5       X6
## 1  12.427794 100.05021 0.81887553  0 4.875051 4.624577
## 2   1.395516  99.60549 0.09166962  0 5.018907 4.691991
## 3  -3.752113  99.50573 0.49065231  0 5.143674 5.016823
## 4 -10.075738 102.02918 0.57743401  0 5.006670 2.895530
## 5  -5.570775  99.58698 0.08013755  0 4.992604 5.758431
## 6   6.230151 100.30338 0.09991487  1 5.040994 4.080104
```

How could we . . .

- find the mean of each variable

Try it now!

Try solutions in Rstudio

Iteration methods

- Copying and pasting
- `for` loops
- `lapply()`
- `apply()`, `mapply()`, `sapply()`, `tapply()`, `vapply()`

Nothing wrong with any of them if they work for you and your use case! But `purrr` can have some advantages.

Why You Might Use `purrr` vs copy and paste / for loops / `apply()`s

1. Consistent, readable syntax (compare to the `_apply()`s)
2. Efficient (compare to `for` loops)
3. Plays nicely with pipes `%>%`
4. Returns the output you expect (type-stable)
5. Reproducibility/ease of making changes
6. Uses either built-in functions (eg, `mean()`) OR build your own, either inline (anonymous) or separately (user-defined)
7. Particularly useful if you're working with list-columns, JSON data, other non-strictly-rectangular data formats

Preamble: Stop Worrying and Learn to Love the List

You're probably already using lists even if you don't know it (for example, a data.frame is a special kind of list!). Generically, lists in R can have as many elements as you want, and each element can be of whatever type you want (including another list… it's lists all the way down). For example, a totally valid list:

```
list("a" = 1:10,          ## numeric vector of length 10
     "b" = list(1:10),    ## list of length 1; element 1 = vector of length 10
     "c" = LETTERS[1:10]) ## character vector of length 10
```

```
## $a
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $b
## $b[[1]]
##  [1]  1  2  3  4  5  6  7  8  9 10
##
##
## $c
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"
```

Other examples of lists include model fit objects (we'll see this with `lm` later), `ggplot2` objects - lots of functions return lists. R for Data Science has a great intro to lists for more information.

Lists' flexibility can allow you lots of freedom once you get comfortable with them; that flexibility can also introduce some complexity. `purrr` is built in part to let you take advantage of lists' benefits as well as some help dealing with the potential pitfalls.

maps Are Where It's At

map() and its variants are the workhorses of purrr. They let us do the same or similar things to a bunch of things, get the output we expect, and sometimes get the final result we want in one step.

How map Works

There are several variants of map, but they all work in the same general way:

1. Over a set of arguments (called .x in map() classic),
2. Do a function (.f)

map can work with three kinds of functions:

1. Built-in functions (mean, subset...)

An example:

Try finding the mean of each of the variables with map

Solution

```
toy.data %>% map(mean)
```

```
## $X1
## [1] -0.2587203
##
## $X2
## [1] 99.85299
##
## $X3
## [1] 0.4654929
##
## $X4
## [1] 0.48
##
## $X5
## [1] 5.00043
##
## $X6
## [1] 5.137586
```

map can work with three kinds of functions:

1. Built-in functions (mean, subset...)
2. User-defined functions

An example:

Try writing your own function to find the two integers on either side of the mean

Then find these bounds for each of the variables with `map`

Solution

```
my.mean.bounds <- function(x){
  c(floor(mean(x)),ceiling(mean(x)))
}

toy.data %>% map(my.mean.bounds)
```

```
## $X1
## [1] -1  0
##
## $X2
## [1]  99 100
##
## $X3
## [1] 0 1
##
## $X4
## [1] 0 1
##
## $X5
## [1] 5 6
##
## $X6
## [1] 5 6
```

`map` can work with three kinds of functions:

1. Built-in functions (`mean`, `subset`…)
2. User-defined functions
3. Anonymous in-line functions

Anonymous functions

Also called lambda functions

- ~ lets R know the following stuff will be an anonymous function
- . is each item in the list

map ([list I am iterating over], ~ .) This would do nothing!

map ([list I am iterating over], ~ ./2) This would just divide each thing in the list in half

An example:

Try to find the two integers on either side of the mean for each of the variables with `map`

Using only map, only one line of code!

Solution

```
toy.data %>% map(~ c(floor(mean(.)),ceiling(mean(.))) )
```

```
## $X1
## [1] -1  0
##
## $X2
## [1]  99 100
##
## $X3
## [1] 0 1
##
## $X4
## [1] 0 1
##
## $X5
## [1] 5 6
##
## $X6
## [1] 5 6
```

Types of `map`

`map` in its purest form will always give you a list. But if you've ever written `do.call(rbind, lapply(...))`, you know that sometimes you don't actually *want* a list. `purrr` is HERE FOR YOU. `map` has several type-specific variants:

1. `_df`: turns your result into a data.frame/tibble! Can do this via rows (default; also `map_dfr`) or columns (`map_dfc`)
2. `_chr`: results in a character vector
3. `_lgl`: results in a logical vector
4. `_int`: results in an integer vector
5. `_dbl`: results in a double vector

Review

Let's take two vectors, both `1:10`, and see what happens if we map over both using `map` variants. This will also be a basic introduction to using anonymous functions.

```
v1 <- 1:10

v1 %>% map(~ . * 3)
```

```
## [[1]]
## [1] 3
##
## [[2]]
## [1] 6
##
## [[3]]
## [1] 9
##
## [[4]]
## [1] 12
##
## [[5]]
## [1] 15
##
## [[6]]
## [1] 18
##
## [[7]]
## [1] 21
##
## [[8]]
## [1] 24
##
## [[9]]
## [1] 27
##
## [[10]]
## [1] 30
```

```
## Returns a list, because we used map()

v1 %>% map_dbl(~ . * 3)
```

```
##  [1]  3  6  9 12 15 18 21 24 27 30
```

```
## Same values, but returns a vector of doubles

v1 %>% map_chr(~ LETTERS[.])
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"
```

```
## Character vector of LETTERS[1:10]

v1 %>% map_lgl(~ . < 5)
```

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

```
## Logical vector that indicates whether the number is less than 5
```

"Amounts" of `map`

You might use a slightly different version of `map` depending on how many things you want to change for each iteration.

1. `map`: Do the exact same thing to a bunch of things (specifies one argument to a function)
2. `map2`: Do the exact same thing to a bunch of things, except for one thing (specifies two arguments to a function)
3. `pmap`: Do similar things to a bunch of things (specifies many arguments to a function)

Each of these has a match in the `walk` functions. While `map` returns an object, `walk` is called for "side effects" (eg, plots, printed text, etc) and returns nothing. We'll see examples of both later.

Example Time!

We're going to try out some `map` uses, and some other fun surprises of `purrr`, by looking at some US National Parks Service data. Happy 101st birthday, National Parks! Specifically, we'll use iteration to:

1. Fit the same model to three different outcomes
2. Check assumptions for those models
3. If needed, update the model
4. Visualize our model results

(Our statistical example is purposely kept very simple so the focus can be on iteration)

Data

We'll be using a few datasets:

1. Annual Recreation Visits, 2007-2016
2. Annual Backcountry Campers, 2007-2016
3. Annual Tent Campers, 2007-2016
4. NPS Data Glossary

```
load('purrr_data.Rdata')

length(datalist)
```

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

```
# total recreational visitors
head(datalist[[1]])
```

```
## # A tibble: 6 x 7
##   parkname             value  year name             type  location region
##   <chr>                <int> <int> <chr>            <chr> <chr>    <fct>
## 1 Kings Canyon NP     580129  2007 Kings Canyon     NP    CA       Pacifi~
## 2 Virgin Islands NP   571382  2007 Virgin Islands   NP    VI       Easter~
## 3 Petrified Forest NP 563590  2007 Petrified Forest NP    AZ       Interm~
## 4 Capitol Reef NP     554907  2007 Capitol Reef     NP    UT       Interm~
## 5 Mesa Verde NP       541102  2007 Mesa Verde       NP    CO       Interm~
## 6 Biscayne NP         517442  2007 Biscayne         NP    FL       Easter~
```

```
# tent campers
head(datalist[[2]])
```

```
## # A tibble: 6 x 7
##   parkname                   value  year name       type  location  region
##   <chr>                      <int> <int> <chr>      <chr> <chr>     <fct>
## 1 Yosemite NP               437429  2007 Yosemite   NP    CA        Pacifi~
```

```
## 2 Great Smoky Mountains NP 163489  2007 Great Smo~ NP    NC, TN    Easter~
## 3 Shenandoah NP              91875  2007 Shenandoah NP    VA        Easter~
## 4 Glacier NP                 80372  2007 Glacier    NP    MT        Interm~
## 5 Yellowstone NP             77754  2007 Yellowsto~ NP    ID, MT, ~ Midwest
## 6 Kings Canyon NP            69607  2007 Kings Can~ NP    CA        Pacifi~
```

```r
# backcountry visitors
head(datalist[[3]])
```

```
## # A tibble: 6 x 7
##   parkname           value  year name           type  location region
##   <chr>              <int> <int> <chr>          <chr> <chr>    <fct>
## 1 Grand Canyon NP   282663  2007 Grand Canyon   NP    AZ       Intermount~
## 2 Rocky Mountain NP  36078  2007 Rocky Mountain NP    CO       Intermount~
## 3 Isle Royale NP     35400  2007 Isle Royale    NP    MI       Midwest
## 4 Shenandoah NP      33668  2007 Shenandoah     NP    VA       Eastern US
## 5 Grand Teton NP     29906  2007 Grand Teton    NP    WY       Intermount~
## 6 Glacier NP         27993  2007 Glacier        NP    MT       Intermount~
```

Run Models

Let's say we want to predict the number of a) total recreational, b) tent campers, and c) backcountry visitors per year using the year, the region, and an interaction between the two. You guessed it: We can use `map`! This seems like a good time for an anonymous function.

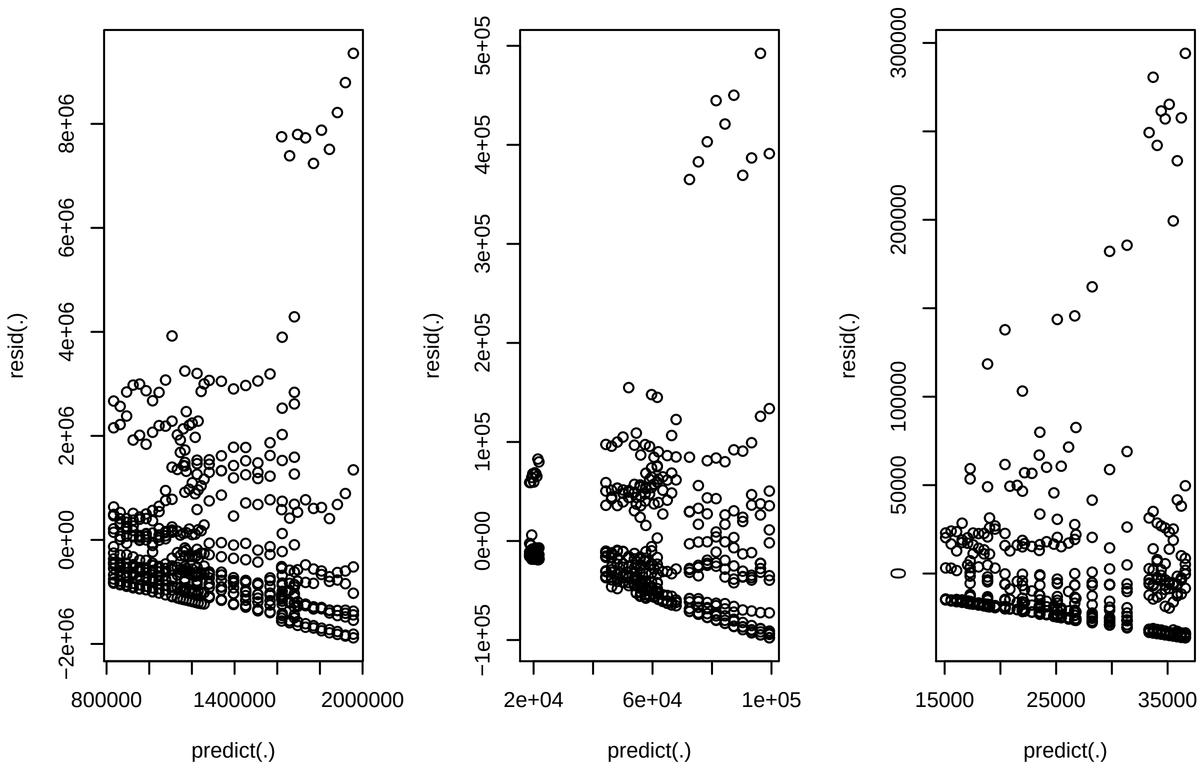
```r
## Fit the same model to each dataset
orgmod_list <- map(
  .x = datalist,
  .f = ~ lm(value ~ year * region, data = .)
)

orgmod_sum <- orgmod_list %>% map(summary)

orgmod_sum
```

```
## [[1]]
##
## Call:
## lm(formula = value ~ year * region, data = .)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1883035  -921339  -566145   407709  9356090
##
## Coefficients:
##                             Estimate Std. Error t value Pr(>|t|)
## (Intercept)               -73310800  120938773  -0.606    0.545
## year                          37335      60124   0.621    0.535
## regionIntermountain       -39958726  151173466  -0.264    0.792
## regionMidwest              46483370  176297042   0.264    0.792
## regionPacific NW           13119199  148432842   0.088    0.930
## year:regionIntermountain      19684      75154   0.262    0.793
## year:regionMidwest           -23404      87644  -0.267    0.790
## year:regionPacific NW         -6929      73792  -0.094    0.925
```

```
##
## Residual standard error: 1638000 on 500 degrees of freedom
## Multiple R-squared:  0.03669,    Adjusted R-squared:  0.0232
## F-statistic:  2.72 on 7 and 500 DF,  p-value: 0.008899
##
##
## [[2]]
##
## Call:
## lm(formula = value ~ year * region, data = .)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -97774 -46674 -18059  36152 492398
##
## Coefficients:
##                            Estimate Std. Error t value Pr(>|t|)
## (Intercept)              -2929260.0  7137670.8  -0.410    0.682
## year                         1486.7     3548.4   0.419    0.675
## regionIntermountain       -880746.6  8937365.8  -0.099    0.922
## regionMidwest             2253165.1 10506367.0   0.214    0.830
## regionPacific NW         -2987519.9  8705236.3  -0.343    0.732
## year:regionIntermountain      433.7     4443.1   0.098    0.922
## year:regionMidwest          -1140.5     5223.1  -0.218    0.827
## year:regionPacific NW        1497.5     4327.8   0.346    0.730
##
## Residual standard error: 85270 on 387 degrees of freedom
## Multiple R-squared:  0.06919,    Adjusted R-squared:  0.05235
## F-statistic:  4.11 on 7 and 387 DF,  p-value: 0.0002268
##
##
## [[3]]
##
## Call:
## lm(formula = value ~ year * region, data = .)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -36339 -24334 -15379   4838 294113
##
## Coefficients:
##                            Estimate Std. Error t value Pr(>|t|)
## (Intercept)              -1300289.7  4607966.3  -0.282    0.778
## year                          658.3     2290.8   0.287    0.774
## regionIntermountain        609079.4  5452219.3   0.112    0.911
## regionMidwest              327719.1  6516648.5   0.050    0.960
## regionPacific NW         -1821274.5  5413885.1  -0.336    0.737
## year:regionIntermountain     -297.3     2710.5  -0.110    0.913
## year:regionMidwest           -166.2     3239.7  -0.051    0.959
## year:regionPacific NW         905.7     2691.5   0.337    0.737
##
## Residual standard error: 50970 on 420 degrees of freedom
## Multiple R-squared:  0.01856,    Adjusted R-squared:  0.002199
## F-statistic: 1.134 on 7 and 420 DF,  p-value: 0.3403
```

Looks like everything went well, but lots of us are statisticians, after all. Do these models fit the usual assumptions? Let's quickly look at some residuals vs fitted plots using `purrr`'s `walk()` function, which you can call when you want the *side effects* of a function instead of returning an object.

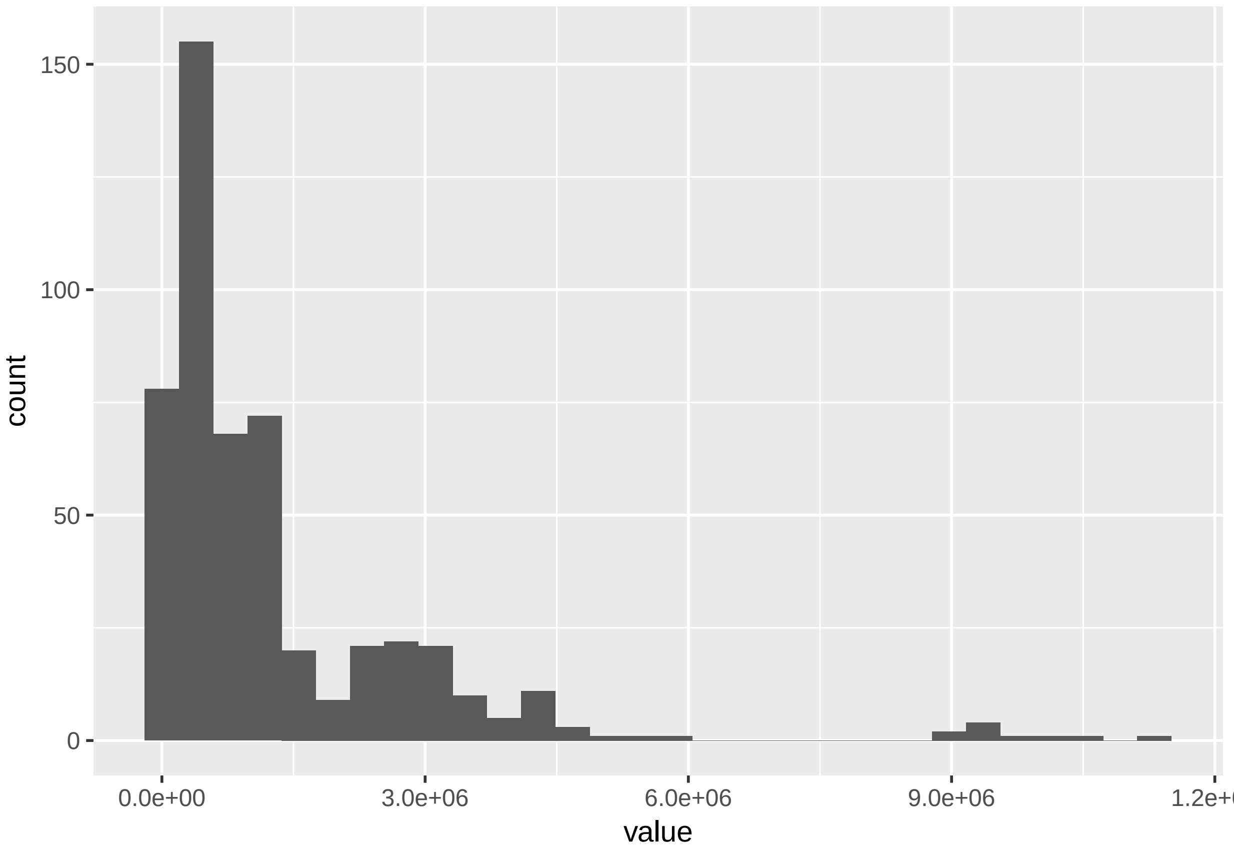
```
par(mfrow = c(1, 3))
walk(orgmod_list, ~ plot(resid(.) ~ predict(.)))
```



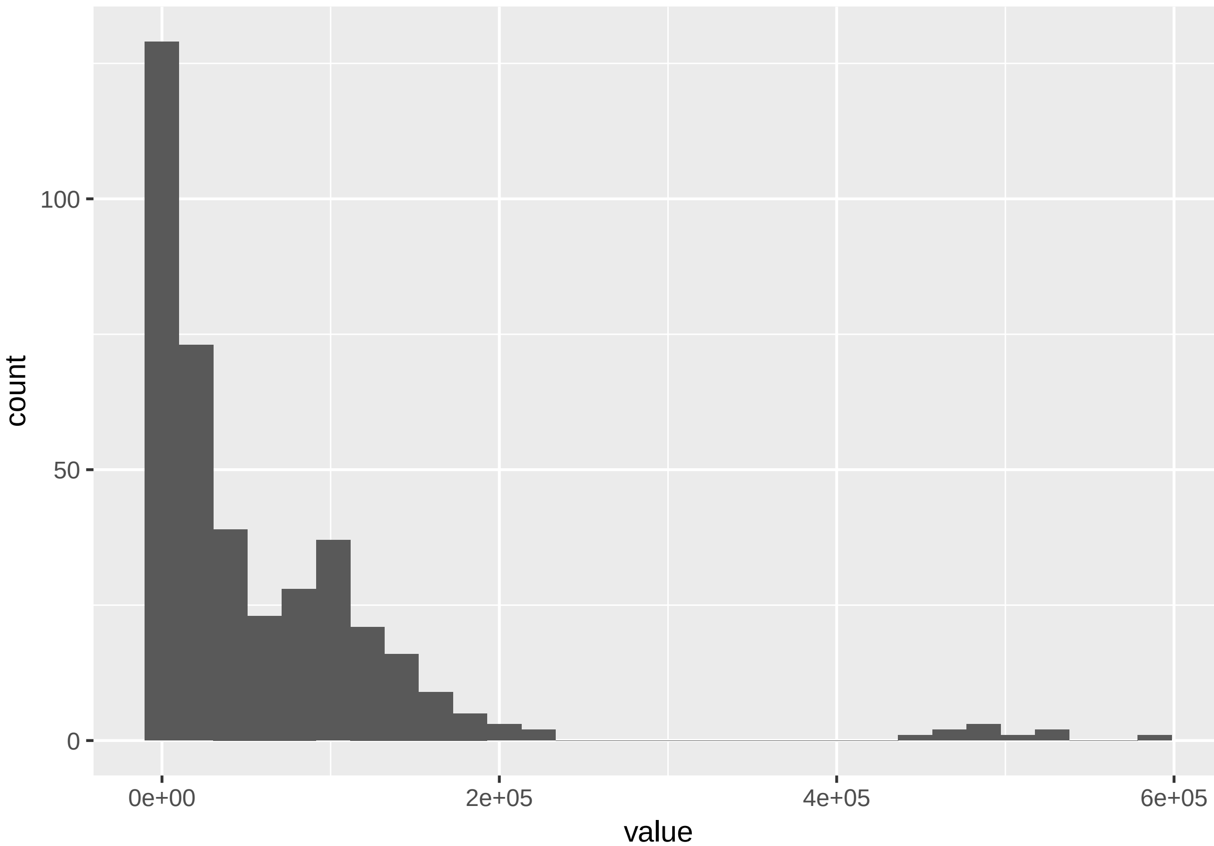
Hmm… some weirdness. What's the distribution of our outcome?

```
walk(datalist, ~ print(ggplot(data = ., aes(x = value)) + geom_histogram()))
```

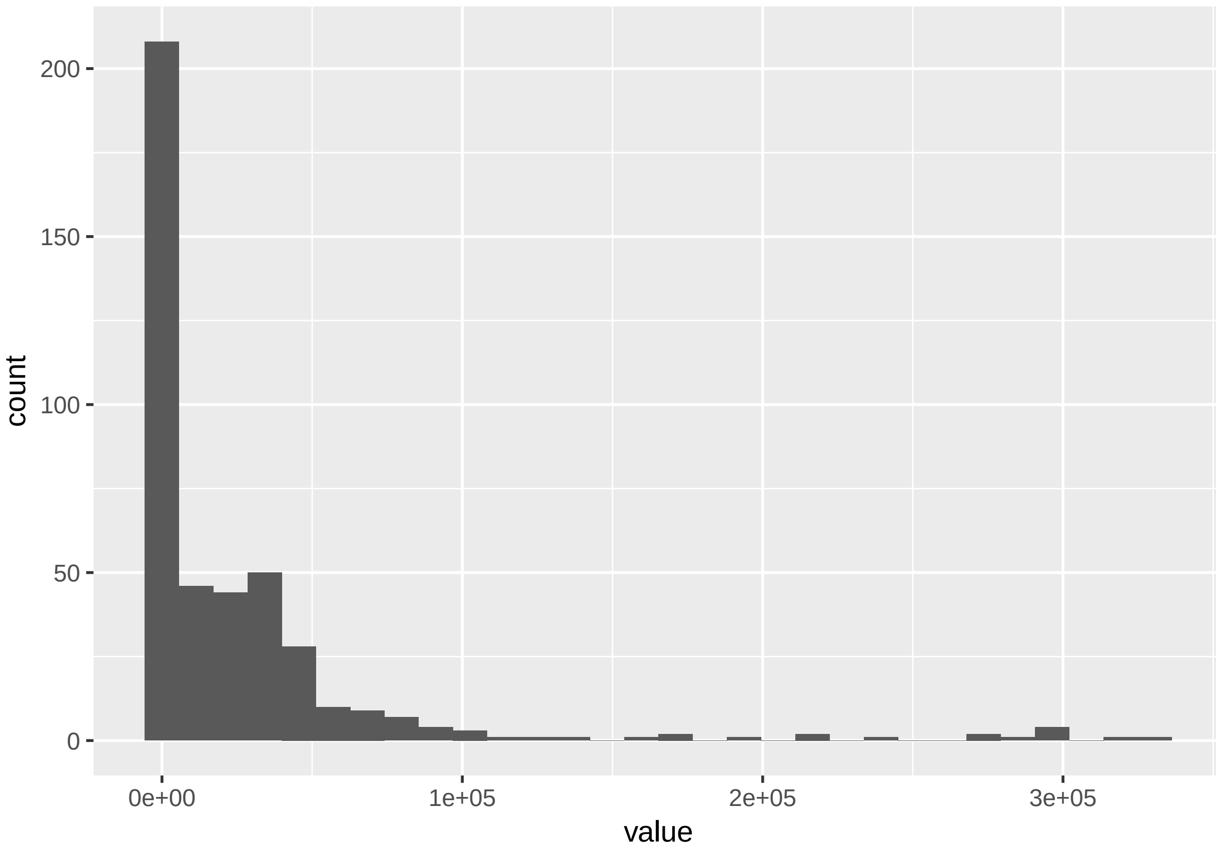
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```




```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

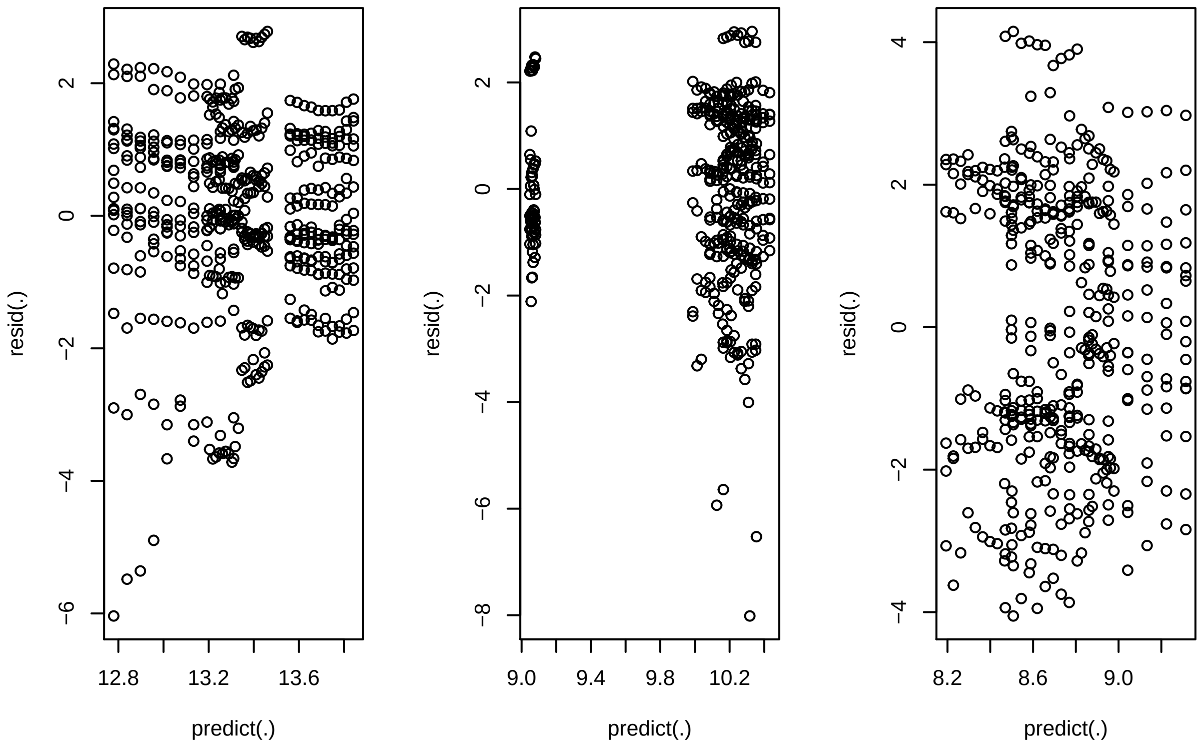



```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Some skewness there! Let's log transform our outcome and refit the models.

```
## Add log transformed value to each dataset
## One base way
# for(i in 1:length(datalist)){
#   datalist[[i]]$logvalue <- log(datalist[[i]]$value)
# }

## purrr + dplyr way: apply the log function to the value column in each dataset
datalist <- datalist %>%
  map(~ mutate_at(.x, "value", log))

## Refit linear model to each dataset, recheck RP plots
logmod_list <- map(datalist, ~ lm(value ~ year * region, data = .))

par(mfrow = c(1, 3))
walk(logmod_list, ~ plot(resid(.) ~ predict(.)))
```



Looking better. Just out of curiosity, what's our R^2 on those models? `summary()` of an `lm` object returns a list, of which one element is the adjusted R^2. We can extract that value for each of our models really quickly using `map_dbl`.

```
## You can do this two ways, whichever you find more readable:
## All in one line:
round(map_dbl(logmod_list, ~ summary(.)$adj.r.squared), 2)
```

```
## [1] 0.03 0.05 0.00
```

```
## In a pipe:
logmod_list %>%
  map(summary) %>%
  map_dbl(.f = "adj.r.squared") %>%
  ## Passing .f a quoted string means "get this element out of the object in .x"
  round(2)
```

```
## [1] 0.03 0.05 0.00
```

Well, that's not great, but that's not really the point now is it. Moving on!

Plot Results

Now let's say we want to generate separate plots for the predicted visitors over time by region for each dataset, and save each plot as a PDF. We're going to

1. Create a list of data.frames with predicted values for each region and year
2. Plot each
3. Save those plots

In this chunk of code, we use:

- `purrr::cross_df` to get all possible combinations of two vectors and put them in a data.frame *(this does essentially the same thing as `expand.grid`, but `cross` can also create lists, which can be really helpful for simulations, for example)*
- `purrr::pluck` to extract elements of a list - this can be helpful, since list notation can get confusing in its natural habitat, mixing `[[double brackets]][singlebrackets]$dollarsigns`
- `purrr::map` in a pipeline, starting with one list of elements and putting it through a process with multiple steps

```
## -- Create base data set with records for which we want predicted values -----
preddata <- cross_df(
  ## You can access the columns of one of our datasets using purrr::pluck() or
  ## base R; both ways shown here
  .l = list("year" = unique(pluck(datalist, 1, "year")),
            "region" = levels(datalist[[1]]$region))
)

## -- Get actual predicted values for each year, region ------------------------
pred_list <- logmod_list %>%
  ## Apply the predict function to each model
  map(predict, newdata = preddata, se.fit = TRUE) %>%
  ## predict() returns a list; extract the fit and se.fit elements
  ## Again, elements of our list are extracted two ways to compare
  map(~ data.frame(fit = pluck(., "fit"), se = .$se.fit) %>%
        ## Calculate confidence limits
        mutate(lcl = fit - qnorm(0.975) * se,
               ucl = fit + qnorm(0.975) * se)) %>%
  ## Add year and region onto each
  map(dplyr::bind_cols, preddata)
```

```
## -- Write a function to plot values for a given dataset ----------------------
plot_predicted <- function(df, vscale, maintitle){
  ## Make sure df has all the columns we need
  if(!all(c("fit", "se", "lcl", "ucl", "year", "region") %in% names(df))){
    stop("df should have columns fit, se, lcl, ucl, year, region")
  }

  ## Create a plot faceted by region
  p <- ggplot(data = df, aes(x = year, y = fit)) +
    facet_wrap(~ region, nrow = 2) +
    geom_ribbon(aes(ymin = lcl, ymax = ucl, fill = region), alpha = 0.4) +
    geom_line(aes(color = region), size = 2) +
    scale_fill_viridis(option = vscale, discrete = TRUE, end = 0.75) +
    scale_colour_viridis(option = vscale, discrete = TRUE, end = 0.75) +
    labs(title = maintitle,
         x = NULL, y = "Log(Visitors)") +
    theme(legend.position = "none")

  return(p)

}
```

Notice our function has three arguments, which means we can't use `map`. We need the big guns: `pmap`. The `p` stands for `parallel`, and we're going to iterate over a **list** of arguments in *parallel* to get the plots we want. First, we'll set up our named list of arguments.

```
plot_args <- list(
  "df" = pred_list, ## list with three elements
  "vscale" = c("D", "A", "C"),
  "maintitle" = c("Total Recreational Visits",
                  "Tent Campers",
                  "Backcountry Campers")
)
```

Because we wrote our function already, once that list is done, it's one simple line to generate all of our plots:

```
nps_plots <- pmap(plot_args, plot_predicted)
```

Notice that nothing printed; `pmap` saved these three plots to a list, but now we need to do something with them. We could print them to our screen with `walk(nps_plots, print)`, OR we could save them to PDFs using `walk2`. Remember, `map2` and `walk2` iterate over *exactly two* arguments - here, it'll be our list of plots, and a list of file names.

```
walk2(.x = c("rec.pdf", "tent.pdf", "backcountry.pdf"),
      .y = nps_plots,
      ggsave,
      width = 8, height = 6)
```

Thus ends our example!

BUT WAIT! THERE'S MORE!

A few purrr features we haven't mentioned yet:

- `partial`, for when you want to create a partially specified version of a function (eg, `q25 <- partial(quantile, probs = 0.25, na.rm = TRUE)`)
- `flatten`, for removing hierarchies from a list
- `safely`, `quietly`, `possibly` - can be helpful especially when writing functions or packages
- `invoke`, `modify` - I haven't used these a ton yet
- List-columns can be your friend if you want to store complex data, results, etc in a tidy way; this is likely a whole other meetup, but `purrr` functions can be really helpful when working with these. Jenny Bryan's tutorial linked below is a great resource here.

purrr resources

- Official page on tidyverse.org
- RStudio cheatsheet (under "Apply Functions")
- DataCamp: Writing Functions in R
- Charlotte Wickham's purrr tutorial
- Jenny Bryan's purrr tutorial: particularly great if you love the idea of list-columns
- Hadley Wickham on purrr vs *apply
- Fun use cases:
  - A roundup of blog posts curated by Mara Averick
  - Peter Kamerman on bootstrap CIs
  - Ken Butler on handling errors with safely/possibly