purrr for biostatisticians

⚔
with examples

Daniel Sjoberg

February, 14 2019

purrr
www.rstudio.com

:map functions:

let's get started

purrr package

purrr enhances R's functional programming toolkit (read: the apply family of functions) by providing a complete and consistent set of tools for working with functions and vectors

best place to start is the family of `map()` functions which allow you to replace many for loops* with code that is more succinct and easier to read

`map()` functions transform their input by applying a function to each element and returning a vector the same length as the input

[*] And much much more

purrr::map vs base::apply

base apply is to *Sister Act* as purrr map is to __________ ?

purrr::map vs base::apply

base apply is to *Sister Act* as purrr map is to ***Sister Act 2: Back in the Habit*****, the sequal to a great film that is even better than the first!**

www.impawards.com

base::apply

- first argument to `lapply()` is the data; the first argument to `mapply()` is the function
- no consistent way to pass additional arguments; most use `...`, `mapply()` uses `MoreArgs`, and some require you to create a new anonymous function
- output from `*apply()` is not consistent; output from the `map*()` functions are entirely predictable

base::apply

- first argument to `lapply()` is the data; the first argument to `mapply()` is the function
- no consistent way to pass additional arguments; most use `...`, `mapply()` uses `MoreArgs`, and some require you to create a new anonymous function
- output from `*apply()` is not consistent; output from the `map*()` functions are entirely predictable

- `v-`, `s-`, and `mapply()` uses `USE.NAMES = FALSE` to suppress names in output; `lapply()` does not have this argument

purrr::map

- the `map*()` family has greater consistency among functions
- `map()`, `map2()`, and `pmap()` inputs are the same and allow for flexible input
- consistent methods for passing additional arguments

purrr::map

- the `map*()` family has greater consistency among functions
- `map()`, `map2()`, and `pmap()` inputs are the same and allow for flexible input
- consistent methods for passing additional arguments

- the output from the map family of functions is predictable and easily modifiable

use cases

1. subgroup analyses
2. sensitivity analyses
3. reading all files in a folder
4. bootstrap analyses
5. other purrr package functions

usage

```
map(.x, .f, ...)
```

```
.x      A list or atomic vector
.f      A function or formula
          If a function, it is used as is,
          If a formula (e.g. ~ .x +2), if is converted to a function.
            - For a single argument function, use `.`, `.x`, or `..1`
...     Additional arguments passed on to the mapped function
```

usage

```
map(.x, .f, ...)
```

```
.x      A list or atomic vector
.f      A function or formula
          If a function, it is used as is,
          If a formula (e.g. ~ .x +2), if is converted to a function.
            - For a single argument function, use `.`, `.x`, or `..1`
...     Additional arguments passed on to the mapped function
```

```
a = list(1:3, 4:6, 7:9)
map(a, sum)
map(a, sum, na.rm = FALSE)
```

usage

```
map(.x, .f, ...)
```

```
.x      A list or atomic vector
.f      A function or formula
          If a function, it is used as is,
          If a formula (e.g. ~ .x +2), if is converted to a function.
            - For a single argument function, use `.`, `.x`, or `..1`
...     Additional arguments passed on to the mapped function
```

```
a = list(1:3, 4:6, 7:9)
map(a, sum)
map(a, sum, na.rm = FALSE)
```

```
map(a, function(x) sum(x))
```

usage

```r
map(.x, .f, ...)
```

```
.x      A list or atomic vector
.f      A function or formula
          If a function, it is used as is,
          If a formula (e.g. ~ .x +2), if is converted to a function.
            - For a single argument function, use `.`, `.x`, or `..1`
...     Additional arguments passed on to the mapped function
```

```r
a = list(1:3, 4:6, 7:9)
map(a, sum)
map(a, sum, na.rm = FALSE)
```

```r
map(a, function(x) sum(x))
```

```r
map(a, ~sum(.))
map(a, ~sum(.x))
map(a, ~sum(..1))
```

more usage

```
map2(.x, .y, .f, ...)
pmap(.l, .f, ...)
```

```
.x, .y A list or atomic vector
.l     A list of vectors
.f     A function or formula
         If a function, it is used as is,
         If a formula (e.g. ~ .x +2), if is converted to a function.
           - For two arguments use `.x` and `.y`, or `..1` and `..2`
           - For more arguments, use `..1`, `..2`, `..3`, etc.
...    Additional arguments passed on to the mapped function
```

more usage

```
map2(.x, .y, .f, ...)
pmap(.l, .f, ...)
```

```
.x, .y A list or atomic vector
.l     A list of vectors
.f     A function or formula
         If a function, it is used as is,
         If a formula (e.g. ~ .x +2), if is converted to a function.
           - For two arguments use `.x` and `.y`, or `..1` and `..2`
           - For more arguments, use `..1`, `..2`, `..3`, etc.
...    Additional arguments passed on to the mapped function
```

```
a = list(1:3, 4:6, 7:9)
b = list(9:7, 6:4, 3:1)
map2(a, b, ~sum(c(.x, .y)))
pmap(list(a, b), ~sum(c(.x, .y)))
```

more usage

```
map2(.x, .y, .f, ...)
pmap(.l, .f, ...)
```

```
.x, .y A list or atomic vector
.l     A list of vectors
.f     A function or formula
         If a function, it is used as is,
         If a formula (e.g. ~ .x +2), if is converted to a function.
           - For two arguments use `.x` and `.y`, or `..1` and `..2`
           - For more arguments, use `..1`, `..2`, `..3`, etc.
...    Additional arguments passed on to the mapped function
```

```
a = list(1:3, 4:6, 7:9)
b = list(9:7, 6:4, 3:1)
map2(a, b, ~sum(c(.x, .y)))
pmap(list(a, b), ~sum(c(.x, .y)))
```

```
map2(a, b, ~sum(c(..1, ..2)))
pmap(list(a, b), ~sum(c(..1, ..2)))
```

trial dataset

```
library(biostatR)
trial %>%
  fmt_table1(by ="trt", missing = "no") %>%
  add_n()
```

Variable	N	Drug	Placebo
		N = 107	N = 93
Age, yrs	192	47 (39, 58)	46 (36, 54)
Marker Level, ng/mL	192	0.61 (0.22, 1.20)	0.72 (0.22, 1.63)
T Stage	200		
T1		25 (23%)	26 (28%)
T2		26 (24%)	23 (25%)
T3		29 (27%)	13 (14%)
T4		27 (25%)	31 (33%)
Grade	200		
I		38 (36%)	29 (31%)
II		34 (32%)	24 (26%)
III		35 (33%)	40 (43%)
Tumor Response	191	52 (51%)	30 (33%)

use cases

1. ***subgroup analyses***
2. sensitivity analyses
3. read all files in a folder
4. bootstrap analyses
5. other purrr package functions

subgroup analysis

tidyr::nest + purrr::map

```
trial %>%
  group_by(grade) %>%
  nest()
```

```
## # A tibble: 3 x 2
##   grade data
##   <fct> <list>
## 1 I     <tibble [67 x 5]>
## 2 III   <tibble [75 x 5]>
## 3 II    <tibble [58 x 5]>
```

subgroup analysis

tidyr::nest + purrr::map

```
trial %>%
  group_by(grade) %>%
  nest()
```

```
## # A tibble: 3 x 2
##   grade data
##   <fct> <list>
## 1 I     <tibble [67 x 5]>
## 2 III   <tibble [75 x 5]>
## 3 II    <tibble [58 x 5]>
```

- tibbles share the same structure as data frames
- tibbles are a list of vectors, and it is possible for to have a list column
- very useful because a list can contain any other object: this means you can put any object in a data frame!
- allows you to keep related objects together in a row, no matter how complex the individual objects are

subgroup analysis

tidyr::nest + dplyr::mutate + purrr::map

```
trial %>%
  group_by(grade) %>%
  nest() %>%
  mutate(
    cross_tab = map(data, ~ table(.x[["response"]], .x[["trt"]])),
    chi_sq = map(cross_tab, ~ chisq.test(.)),
    p_value = map(chi_sq, ~ ..1[["p.value"]])
  )
```

```
## # A tibble: 3 x 5
##   grade data               cross_tab    chi_sq       p_value
##   <fct> <list>             <list>       <list>       <list>
## 1 I     <tibble [67 x 5]> <S3: table> <S3: htest> <dbl [1]>
## 2 III   <tibble [75 x 5]> <S3: table> <S3: htest> <dbl [1]>
## 3 II    <tibble [58 x 5]> <S3: table> <S3: htest> <dbl [1]>
```

subgroup analysis

tidyr::nest + dplyr::mutate + purrr::map

```
trial %>%
  group_by(grade) %>%
  nest() %>%
  mutate(
    cross_tab = map(data, ~ table(.x[["response"]], .x[["trt"]])),
    chi_sq = map(cross_tab, ~ chisq.test(.)),
    p_value = map(chi_sq, ~ ..1[["p.value"]])
  )
```

```
## # A tibble: 3 x 5
##   grade data               cross_tab   chi_sq       p_value
##   <fct> <list>             <list>      <list>       <list>
## 1 I     <tibble [67 x 5]> <S3: table> <S3: htest> <dbl [1]>
## 2 III   <tibble [75 x 5]> <S3: table> <S3: htest> <dbl [1]>
## 3 II    <tibble [58 x 5]> <S3: table> <S3: htest> <dbl [1]>
```

we want the p-values, not a list of p-values

subgroup analysis

tidyr::nest + dplyr::mutate + purrr::map_dbl

```
trial %>%
  group_by(grade) %>%
  nest() %>%
  mutate(
    cross_tab = map(data, ~ table(.x[["response"]], .x[["trt"]])),
    chi_sq = map(cross_tab, ~ chisq.test(.)),
    p_value = map_dbl(chi_sq, ~ ..1[["p.value"]])
  )
```

```
## # A tibble: 3 x 5
##   grade data               cross_tab   chi_sq      p_value
##   <fct> <list>             <list>      <list>        <dbl>
## 1 I     <tibble [67 x 5]> <S3: table> <S3: htest>   0.152
## 2 III   <tibble [75 x 5]> <S3: table> <S3: htest>   0.328
## 3 II    <tibble [58 x 5]> <S3: table> <S3: htest>   0.193
```

output types

the default output of `map()` is a list

we can coerce the output type with `map_*()`

function	output type
map()	list
map_dbl()	double
map_int()	integer
map_lgl()	logical
map_dfr()	tibble (bind_rows)
map_dfc()	tibble (bind_cols)

when using the `map_*()` functions, `map()` runs as it typically would with the added step of coercing the output at the end

tip: make sure your code works with `map()` before adding `map_*()`.

use cases

1. subgroup analyses
2. ***sensitivity analyses***
3. read all files in a folder
4. bootstrap analyses
5. other purrr package functions

sensitivity analyses

run your analysis among all patients, and again excluding low grade patients

```
tibble(
  cohort = c("TRUE", "grade != 'I'")
) %>%
kable(format = "html")
```

cohort
TRUE
grade != 'I'

sensitivity analyses

run your analysis among

- all patients
- excluding low grade patients

```
tibble(
  cohort = c("TRUE", "grade != 'I'")
) %>%
mutate(
  data = map(cohort, ~ trial %>% filter_(.x)),
  p_value = map_dbl(
    data,
    ~ table(.x[["response"]], .x[["trt"]]) %>%
      chisq.test() %>%
      pluck("p.value")
  )
)
```

sensitivity analyses

```
tibble(
  cohort = c("TRUE", "grade != 'I'")
) %>%
mutate(
  data = map(cohort, ~ trial %>% filter_(.x)),
  p_value = map_dbl(
    data,
    ~ table(.x[["response"]], .x[["trt"]]) %>%
      chisq.test() %>%
      pluck("p.value")
  )
)
```

```
## # A tibble: 2 x 3
##   cohort       data               p_value
##   <chr>        <list>               <dbl>
## 1 TRUE         <tibble [200 x 6]>  0.0172
## 2 grade != 'I' <tibble [133 x 6]>  0.0922
```

use cases

1. subgroup analyses
2. sensitivity analyses
3. ***read all files in a folder***
4. bootstrap analyses
5. other purrr package functions

read files

- store vector of the files you want to import

```
file_list = list.files(pattern = "*.csv")
```

read files

- store vector of the files you want to import

```
file_list = list.files(pattern = "*.csv")
```

- use `map()` to read the files. returns a list where each element is a tibble

```
map(file_list, read_csv)
```

read files

- store vector of the files you want to import

```r
file_list = list.files(pattern = "*.csv")
```

- use `map()` to read the files. returns a list where each element is a tibble

```r
map(file_list, read_csv)
```

- append each of the datasets with the `map_dfr()` function. after files have been imported, `bind_rows()` will create one final tibble

```r
map_dfr(file_list, read_csv)
```

read files

- store vector of the files you want to import

```
file_list = list.files(pattern = "*.csv")
```

- use `map()` to read the files. returns a list where each element is a tibble

```
map(file_list, read_csv)
```

- append each of the datasets with the `map_dfr()` function. after files have been imported, `bind_rows()` will create one final tibble

```
map_dfr(file_list, read_csv)
```

- include an identifier with an added `mutate()`

```
map_dfr(file_list, ~read_csv(.x) %>% mutate(file = .x))
map2_dfr(file_list, 1:length(file_list),
         ~read_csv(.x) %>% mutate(file_id = .y))
```

use cases

1. subgroup analyses
2. sensitivity analyses
3. read all files in a folder
4. ***bootstrap analyses***
5. other purrr package functions

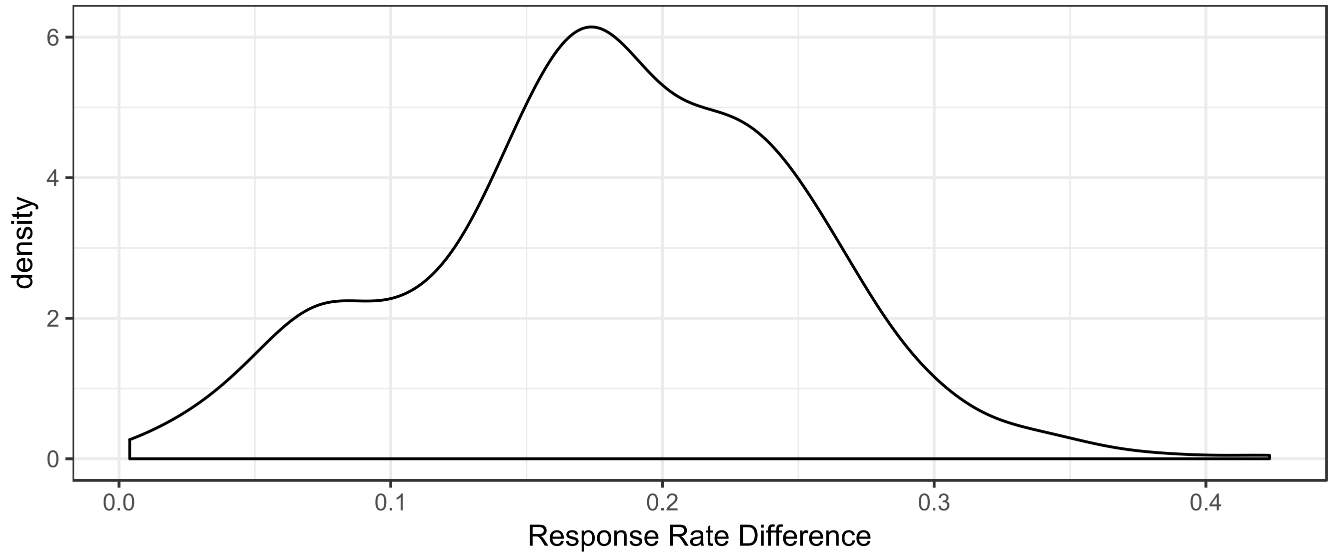
bootstrap analyses

- use bootstrap resampling to estimate the difference in response rate by treatment
- we'll use 500 resampled datasets to estimate the standard deviation of the response rate difference
- assuming normality of the response rate difference, construct a 95% confidence interval for the difference

```r
# create function to calculate risk difference
risk_diff <- function(data) {
  mean(data$response[data$trt == "Drug"], na.rm = TRUE) -
    mean(data$response[data$trt == "Placebo"], na.rm = TRUE)
}
```

bootstrap analyses

```r
# resample dataset 500 times and calculate risk difference
risk_diff_boot <- map_dbl(
  1:500,
  ~ trial %>%
    slice(sample.int(n(), replace = TRUE)) %>%
    risk_diff()
  )
```



bootstrap analyses

the result

- 18% (95% CI 4.6%, 32%)

bootstrap analyses

the result

- 18% (95% CI 4.6%, 32%)

how do the results compare to the the Wald CI?

bootstrap analyses

the result

- 18% (95% CI 4.6%, 32%)

how do the results compare to the the Wald CI?

- 18% (95% CI 4.1%, 31%)

Success!

use cases

1. subgroup analyses
2. sensitivity analyses
3. read all files in a folder
4. bootstrap analyses
5. ***other purrr package functions***

other purrr functions

- `modify()`
- `imap()`
- `keep()` and `discard()`
- `pluck()` and `chuck()`
- `cross_df()`
- `possibly()`, `safely()`, and `quietly()`

Yes, Lauren Hill was in *Sister Act 2*!

modify()

unlike `map()` and its variants which always return a fixed object type (list for map(), integer vector for `map_int()`, etc), the `modify()` family always returns the same type as the input object.

```
modify(
  c("hello", "world"),
  ~ .x
)
```

```
## [1] "hello" "world"
```

```
modify(
  1:3,
  ~ .x
)
```

```
## [1] 1 2 3
```

`imap()`

`imap(x, ...)` is short hand for `map2(x, names(x), ...)` if x has names, or `map2(x, seq_along(x), ...)` if it does not

```
imap(mtcars[, 1:2], ~ glue::glue("{.y}: {mean(.x, na.rm = TRUE)}"))
```

```
## $mpg
## mpg: 20.090625
##
## $cyl
## cyl: 6.1875
```

```
map2(
  mtcars[, 1:2], names(mtcars[, 1:2]),
  ~ glue::glue("{.y}: {mean(.x, na.rm = TRUE)}")
)
```

```
## $mpg
## mpg: 20.090625
##
## $cyl
## cyl: 6.1875
```

`keep()` and `discard()`

keep or discard elements of a list of vector

```
1:10 %>%
  keep(~. < 5)
```

```
## [1] 1 2 3 4
```

```
1:10 %>%
  discard(~.x >= 5)
```

```
## [1] 1 2 3 4
```

pluck()

selects a single element from a vector or list (perfect for use with `%>%`)

```
lm(mpg ~ vs, mtcars) %>%
  pluck("coefficients")
```

```
## (Intercept)          vs
##   16.616667    7.940476
```

cross_df()

it's like `expand.grid()` without the factors...finally!

```
list(
  outcome = c("mets", "death"),
  cohort = c("tpsa > 0", "tpsa > 1")
) %>%
  cross_df()
```

```
## # A tibble: 4 x 2
##   outcome cohort
##   <chr>   <chr>
## 1 mets    tpsa > 0
## 2 death   tpsa > 0
## 3 mets    tpsa > 1
## 4 death   tpsa > 1
```

`possibly()`, `safely()`, and `quietly()`

- these functions wrap functions so that instead of generating side effects through printed output, messages, warnings, and errors, they return enhanced output
- similar to `try()` and `tryCatch()`

done!

questions?