Package: lay (via r-universe)

October 23, 2026

Title Simple but Efficient Rowwise Jobs

Version 0.1.3

Description Creating efficiently new column(s) in a data frame (including tibble) by applying a function one row at a time.

License MIT + file LICENSE

URL https://courtiol.github.io/lay/, https://github.com/courtiol/lay/

BugReports https://github.com/courtiol/lay/issues/

Encoding UTF-8

LazyData true

Depends R (>= 2.10)

Imports rlang, purrr, vctrs, tibble

Suggests bench, covr, data.table, dplyr (>= 1.0), forcats, ggplot2, ggbeeswarm, knitr, rmarkdown, slider, testthat (>= 2.1.0), tidyr, spelling

Roxygen list(markdown = TRUE)

RoxygenNote 7.2.3

Language en-US

Repository https://courtiol.r-universe.dev

Date/Publication 2023-10-31 13:00:50 UTC

RemoteUrl https://github.com/courtiol/lay

RemoteRef HEAD

RemoteSha 1e243d5dda86c090439836e6de254da9f4f49836

Contents

drugs 2
lay 3

Index **6**

`drugs` *Pain relievers misuse in the US*

Description

Datasets containing information about the use of pain relievers for non medical purpose.

Format

A tibble with either 100 or 55271 rows, and 8 variables:

caseid The identifier code of the respondent

hydrocd Ever use hydrocodone nonmedically?

oxycodp Ever use ever percocet, percodan, tylox, oxycontin... nonmedically?

codeine Ever used codeine nonmedically?

tramadl Ever used tramadol nonmedically?

morphin Ever used morphine nonmedically?

methdon Ever used methadone nonmedically?

vicolor Ever used vicodin, lortab or lorcert nonmedically?

Details

These datasets are a small subset from the "National Survey on Drug Use and Health, 2014". All variables related to drug use have been recoded into vectors of integers talking value 0 for "No/Unknown" and value 1 for "Yes". The original variable names were the same as those defined here but in upper case and ending with the number 2. The dataset called `drugs` contain the first 100 rows of the one called `drugs_full`.

Source

https://www.icpsr.umich.edu/web/NAHDAP/studies/36361

References

United States Department of Health and Human Services. Substance Abuse and Mental Health Services Administration. Center for Behavioral Health Statistics and Quality. National Survey on Drug Use and Health, 2014. Ann Arbor, MI: Inter-university Consortium for Political and Social Research (distributor), 2016-03-22. doi:10.3886/ICPSR36361.v1

Examples

```
drugs
drugs_full
```

`lay` *Apply a function within each row*

Description

Create efficiently new column(s) in data frame (including tibble) by applying a function one row at a time.

Usage

```
lay(.data, .fn, ..., .method = c("apply", "tidy"))
```

Arguments

`.data`	A data frame or tibble (or other data frame extensions).
`.fn`	A function to apply to each row of `.data`. Possible values are: • A function, e.g. `mean` • An anonymous function, .e.g. `function(x) mean(x, na.rm = TRUE)` • An anonymous function with shorthand, .e.g. `\(x) mean(x, na.rm = TRUE)` • A purrr-style lambda, e.g. `~ mean(.x, na.rm = TRUE)` (wrap the output in a data frame to apply several functions at once, e.g. `~ tibble(min = min(.x), max = max(.x))`)
`...`	Additional arguments for the function calls in `.fn` (must be named!).
`.method`	This is an experimental argument that allows you to control which internal method is used to apply the rowwise job: • "apply", the default internally uses the function `apply()`. • "tidy", internally uses `purrr::pmap()` and is stricter with respect to class coercion across columns. The default has been chosen based on these **benchmarks**.

Details

`lay()` create a vector or a data frame (or tibble), by considering in turns each row of a data frame (`.data`) as the vector input of some function(s) `.fn`.

This makes the creation of new columns based on a rowwise operation both simple (see **Examples**; below) and efficient (see the Article **benchmarks**).

The function should be fully compatible with `{dplyr}`-based workflows and follows a syntax close to `dplyr::across()`.

Yet, it takes `.data` instead of `.cols` as a main argument, which makes it possible to also use `lay()` outside `dplyr` verbs (see **Examples**).

The function `lay()` should work in a wide range of situations, provided that:

- The input `.data` should be a data frame (including tibble) with columns of same class, or of classes similar enough to be easily coerced into a single class. Note that `.method = "apply"` also allows for the input to be a matrix and is more permissive in terms of data coercion.

- The output of `.fn` should be a scalar (i.e., vector of length 1) or a 1 row data frame (or tibble).

If you use `lay()` within `dplyr::mutate()`, make sure that the data used by `dplyr::mutate()` contain no row-grouping, i.e., what is passed to `.data` in `dplyr::mutate()` should not be of class `grouped_df` or `rowwise_df`. If it is, `lay()` will be called multiple times, which will slow down the computation despite not influencing the output.

Value

A vector with one element per row of `.data`, or a data frame (or tibble) with one row per row of `.data`. The class of the output is determined by `.fn`.

Examples

```
# usage without dplyr ----------------------------------------------------------------------

# lay can return a vector
lay(drugs[1:10, -1], any)

# lay can return a data frame
## using the shorthand function syntax \(x) .fn(x)
lay(drugs[1:10, -1],
   \(x) data.frame(drugs_taken = sum(x), drugs_not_taken = sum(x == 0)))

## using the rlang lambda syntax ~ fn(.x)
lay(drugs[1:10, -1],
   ~ data.frame(drugs_taken = sum(.x), drugs_not_taken = sum(.x == 0)))

# lay can be used to augment a data frame
cbind(drugs[1:10, ],
      lay(drugs[1:10, -1],
         ~ data.frame(drugs_taken = sum(.x), drugs_not_taken = sum(.x == 0))))


# usage with dplyr -------------------------------------------------------------------------

if (require("dplyr")) {

  # apply any() to each row
  drugs |>
    mutate(everused = lay(pick(-caseid), any))

  # apply any() to each row using all columns
  drugs |>
    select(-caseid) |>
    mutate(everused = lay(pick(everything()), any))

  # a workaround would be to use `rowSums`
  drugs |>
    mutate(everused = rowSums(pick(-caseid)) > 0)

  # but we can lay any function taking a vector as input, e.g. median
  drugs |>
```

```
  mutate(used_median = lay(pick(-caseid), median))

# you can pass arguments to the function
drugs_with_NA <- drugs
drugs_with_NA[1, 2] <- NA

drugs_with_NA |>
  mutate(everused = lay(pick(-caseid), any))
drugs_with_NA |>
  mutate(everused = lay(pick(-caseid), any, na.rm = TRUE))

# you can lay the output into a 1-row tibble (or data.frame)
# if you want to apply multiple functions
drugs |>
  mutate(lay(pick(-caseid),
             ~ tibble(drugs_taken = sum(.x), drugs_not_taken = sum(.x == 0))))

# note that naming the output prevent the automatic splicing and you obtain a df-column
drugs |>
  mutate(usage = lay(pick(-caseid),
              ~ tibble(drugs_taken = sum(.x), drugs_not_taken = sum(.x == 0))))

# if your function returns a vector longer than a scalar, you should turn the output
# into a tibble, which is the job of as_tibble_row()
drugs |>
  mutate(lay(pick(-caseid), ~ as_tibble_row(quantile(.x))))

# note that you could also wrap the output in a list and name it to obtain a list-column
drugs |>
  mutate(usage_quantiles = lay(pick(-caseid), ~ list(quantile(.x))))
}
```

Index

apply(), *3*

dplyr::across(), *3*
dplyr::mutate(), *4*
drugs, 2
drugs_full *(*drugs*)*, 2

lay, 3

purrr::pmap(), *3*