Package: epikit (via r-universe)

September 1, 2024

Title Miscellaneous helper tools for epidemiologists

Version 0.1.4

Description Contains tools for formatting inline code, renaming redundant columns, aggregating age categories, adding survey weights, finding the earliest date of an event, plotting z-curves, generating population counts and calculating proportions with confidence intervals. This is part of the 'R4Epis' project <https://r4epis.netlify.com>.

License GPL-3

Encoding UTF-8

LazyData true

Roxygen list(markdown = TRUE)

RoxygenNote 7.2.3

Imports binom, scales, dplyr (>= 1.0.2), rlang, forcats, tidyr (>= 1.0.0), tibble (>= 3.0.0), glue, tidyselect, ggplot2, sf

Suggests testthat (>= 2.1.0), outbreaks, epidict, epitabulate, covr, knitr, magrittr, rmarkdown

Additional_repositories https://r4epi.github.io/drat

URL https://github.com/R4EPI/epikit, https://r4epis.netlify.com, https://r4epi.github.io/epikit/

BugReports https://github.com/R4EPI/epikit/issues

VignetteBuilder knitr

Config/Needs/check R4EPI/epidict R4EPI/epitabulate

Repository https://r4epi.r-universe.dev

RemoteUrl https://github.com/r4epi/epikit

RemoteRef HEAD

RemoteSha 6a1fd659966d315d0aadeb126f23b75c7a6b6e62

Contents

`add_weights_cluster` *Add a column of cluster survey weights to a data frame.*

Description

For use in surveys where you took a sample population out of a larger source population, with a cluster survey design.

Usage

```
add_weights_cluster(
  x,
  cl,
  eligible,
  interviewed,
  cluster_x = NULL,
  cluster_cl = NULL,
  household_x = NULL,
  household_cl = NULL,
  ignore_cluster = TRUE,
  ignore_household = TRUE,
  surv_weight = "surv_weight",
  surv_weight_ID = "surv_weight_ID"
)
```

Arguments

`x`	a data frame of survey data
`cl`	a data frame containing a list of clusters and the number of households in each.

eligible	the column in x which specifies the number of people eligible for being interviewed in that household. (e.g. the total number of children)
interviewed	the column in x which specifies the number of people actually interviewed in that household.
cluster_x	the column in x that indicates which cluster rows belong to. Ignored if ignore_cluster is TRUE.
cluster_cl	the column in cl that lists all possible clusters. Ignored if ignore_cluster is TRUE.
household_x	the column in x that indicates a unique household identifier. Ignored if ignore_household is TRUE.
household_cl	the column in cl that lists the number of households per cluster. Ignored if ignore_household is TRUE.
ignore_cluster	If TRUE (default), set the weight for clusters to be 1. This assumes that your sample was taken in a way which is a close approximation of a simple random sample. Ignores inputs from cluster_cl as well as cluster_x.
ignore_household	If TRUE (default), set the weight for households to be 1. This assumes that your sample of households was takenin a way which is a close approximation of a simple random sample. Ignores inputs from household_cl and household_x.
surv_weight	the name of the new column to store the weights. Defaults to "surv_weight".
surv_weight_ID	the name of the new ID column to be created. Defaults to "surv_weight_ID"

Details

Will multiply the inverse chances of a cluster being selected, a household being selected within a cluster, and an individual being selected within a household.

As follows:

```
((clusters available) / (clusters surveyed)) *
((households in each cluster) / (households surveyed in each cluster)) *
((individuals eligible in each household) / (individuals interviewed))
```

In the case where both ignore_cluster and ignore_household are TRUE, this will simply be:

```
1 * 1 * (individuals eligible in each household) / (individuals interviewed)
```

Author(s)

Alex Spina, Zhian N. Kamvar, Lukas Richter

Examples

```
# define a fake dataset of survey data
# including household and individual information
x <- data.frame(stringsAsFactors=FALSE,
```

```
          cluster = c("Village A", "Village A", "Village A", "Village A",
                      "Village A", "Village B", "Village B", "Village B"),
     household_id = c(1, 1, 1, 1, 2, 2, 2, 2),
       eligible_n = c(6, 6, 6, 6, 6, 3, 3, 3),
       surveyed_n = c(4, 4, 4, 4, 4, 3, 3, 3),
    individual_id = c(1, 2, 3, 4, 4, 1, 2, 3),
          age_grp = c("0-10", "20-30", "30-40", "50-60", "50-60", "20-30",
                      "50-60", "30-40"),
              sex = c("Male", "Female", "Male", "Female", "Female", "Male",
                      "Female", "Female"),
          outcome = c("Y", "Y", "N", "N", "N", "N", "N", "Y")
)

# define a fake dataset of cluster listings
# including cluster names and number of households
cl <- tibble::tribble(
     ~cluster, ~n_houses,
  "Village A",        23,
  "Village B",        42,
  "Village C",        56,
  "Village D",        38
)


# add weights to a cluster sample
# include weights for cluster, household and individual levels
add_weights_cluster(x, cl = cl,
                    eligible = eligible_n,
                    interviewed = surveyed_n,
                    cluster_cl = cluster, household_cl = n_houses,
                    cluster_x = cluster,  household_x = household_id,
                    ignore_cluster = FALSE, ignore_household = FALSE)


# add weights to a cluster sample
# ignore weights for cluster and household level (set equal to 1)
# only include weights at individual level
add_weights_cluster(x, cl = cl,
                    eligible = eligible_n,
                    interviewed = surveyed_n,
                    cluster_cl = cluster, household_cl = n_houses,
                    cluster_x = cluster,  household_x = household_id,
                    ignore_cluster = TRUE, ignore_household = TRUE)
```

`add_weights_strata` *Add a column of stratified survey weights to a data frame. For use in surveys where you took a sample population out of a larger source population, with a simple-random or stratified survey design.*

Description

Creates weight based on dividing stratified population counts from the source population by surveyed counts in the sample population.

Usage

```
add_weights_strata(
  x,
  p,
  ...,
  population = population,
  surv_weight = "surv_weight",
  surv_weight_ID = "surv_weight_ID"
)
```

Arguments

x	a data frame of survey data
p	a data frame containing population data for groups in ...
...	shared grouping columns across both x and p. These are used to match the weights to the correct subset of the population.
population	the column in p that defines the population numbers
surv_weight	the name of the new column to store the weights. Defaults to "surv_weight".
surv_weight_ID	the name of the new ID column to be created. Defaults to "surv_weight_ID"

Author(s)

Zhian N. Kamvar Alex Spina Lukas Richter

Examples

```
# define a fake dataset of survey data
# including household and individual information
x <- data.frame(stringsAsFactors=FALSE,
        cluster = c("Village A", "Village A", "Village A", "Village A",
                    "Village A", "Village B", "Village B", "Village B"),
    household_id = c(1, 1, 1, 1, 2, 2, 2, 2),
     eligibile_n = c(6, 6, 6, 6, 6, 3, 3, 3),
      surveyed_n = c(4, 4, 4, 4, 4, 3, 3, 3),
   individual_id = c(1, 2, 3, 4, 4, 1, 2, 3),
        age_grp = c("0-10", "20-30", "30-40", "50-60", "50-60", "20-30",
                    "50-60", "30-40"),
            sex = c("Male", "Female", "Male", "Female", "Female", "Male",
                    "Female", "Female"),
        outcome = c("Y", "Y", "N", "N", "N", "N", "N", "Y")
)

# define a fake population data set
# including age group, sex, counts and proportions
```

```
p <- epikit::gen_population(total = 10000,
  groups = c("0-10", "10-20", "20-30", "30-40", "40-50", "50-60"),
  proportions = c(0.1, 0.2, 0.3, 0.4, 0.2, 0.1))

  # make sure col names match survey dataset
p <- dplyr::rename(p, age_grp = groups, sex = strata, population = n)

# add weights to a stratified simple random sample
# weight based on age group and sex
add_weights_strata(x, p = p, age_grp, sex, population = population)
```

`age_categories` *Create an age group variable*

Description

Create an age group variable

Usage

```
age_categories(
  x,
  breakers = NULL,
  lower = 0,
  upper = NULL,
  by = 10,
  separator = "-",
  ceiling = FALSE,
  above.char = "+"
)

group_age_categories(
  dat,
  years = NULL,
  months = NULL,
  weeks = NULL,
  days = NULL,
  one_column = TRUE,
  drop_empty_overlaps = TRUE
)
```

Arguments

`x`	Your age variable
`breakers`	A string. Age category breaks you can define within c(). Alternatively use "lower", "upper" and "by" to set these breaks based on a sequence.

lower	A number. The lowest age value you want to consider (default is 0)
upper	A number. The highest age value you want to consider
by	A number. The number of years you want between groups
separator	A character that you want to have between ages in group names. The default is "-" producing e.g. 0-10.
ceiling	A TRUE/FALSE variable. Specify whether you would like the highest value in your breakers, or alternatively the upper value specified, to be the endpoint. This would produce the highest group of "70-80" rather than "80+". The default is FALSE (to produce a group of 80+).
above.char	Only considered when ceiling == FALSE. A character that you want to have after your highest age group. The default is "+" producing e.g. 80+
dat	a data frame with at least one column defining an age category
years, months, weeks, days	the bare name of the column defining years, months, weeks, or days (or NULL if the column doesn't exist)
one_column	if TRUE (default), the categories will be joined into a single column called "age_category" that appends the type of age category used. If FALSE, there will be one column with the grouped age categories called "age_category" and a second column indicating age unit called "age_unit".
drop_empty_overlaps	if TRUE, unused levels are dropped if they have been replaced by a more fine-grained definition and are empty. Practically, this means that the first level for years, months, and weeks are in consideration for being removed via forcats::fct_drop()

Value

a factor representing age ranges, open at the upper end of the range.

a data frame

Examples

```
if (interactive() && require("dplyr") && require("epidict")) {
withAutoprint({
set.seed(50)
dat <- epidict::gen_data("Cholera", n = 100, org = "MSF")
ages <- dat %>%
  select(starts_with("age")) %>%
  mutate(age_years = age_categories(age_years, breakers = c(0, 5, 10, 15, 20))) %>%
  mutate(age_months = age_categories(age_months, breakers = c(0, 5, 10, 15, 20))) %>%
  mutate(age_days = age_categories(age_days, breakers = c(0, 5, 15)))

ages %>%
  group_age_categories(years = age_years, months = age_months, days = age_days) %>%
  pull(age_category) %>%
  table()
})
}
```

fac_from_num *create factors from numbers*

Description

If the number of unique numbers is five or fewer, then they will simply be converted to factors in order, otherwise, they will be passed to cut and pretty, preserving the lowest value.

Usage

```
fac_from_num(x)
```

Arguments

x	a vector of integers or numerics

Value

a factor

Examples

```
fac_from_num(1:100)
fac_from_num(sample(100, 5))
```

find_breaks *Automatically calculate breaks for a number*

Description

Automatically calculate breaks for a number

Usage

```
find_breaks(n, breaks = 4, snap = 1, ceiling = FALSE)
```

Arguments

n	a number to calcluate breaks for
breaks	the maximum number of segments you want to have
snap	the number defining where to snap to the nearest factor
ceiling	if TRUE, n is included in the breaks

Value

a vector of integers

Examples

```
# find four breaks from 1 to 100
find_breaks(100)

# find four breaks from 1 to 123, rounding to the nearest 20
find_breaks(123, snap = 20)

# note that there are only three breaks here because of the rounding
find_breaks(123, snap = 25)

# Include the value itself
find_breaks(123, snap = 25, ceiling = TRUE)
```

---

find_date_cause *Find the first date beyond a cutoff in several columns*

Description

This function will find the first date in an orderd series of columns that is either before or after a cutoff date, inclusive.

Usage

```
find_date_cause(
  x,
  ...,
  period_start = NULL,
  period_end = NULL,
  datecol = "start_date",
  datereason = "start_date_reason",
  na_fill = "start"
)

find_start_date(
  x,
  ...,
  period_start = NULL,
  period_end = NULL,
  datecol = "start_date",
  datereason = "start_date_reason"
)

find_end_date(
  x,
  ...,
  period_start = NULL,
  period_end = NULL,
```

```
  datecol = "end_date",
  datereason = "end_date_reason"
)

constrain_dates(i, period_start, period_end, boundary = "both")

assert_positive_timespan(x, date_start, date_end)
```

Arguments

x	a data frame
...	an ordered series of date columns (i.e. the most important date to be considered first).
period_start, period_end	for the find_ functions, this should be the name of a column in x that contains the start/end of the recall period. For constrain_dates, this should be a vector of dates.
datecol	the name of the new column to contain the dates
datereason	the name of the column to contain the name of the column from which the date came.
na_fill	one of either "before" or "after" indicating that the new column should only contain dates before or after the cutoff date.
i	a vector of dates
boundary	one of "both", "start", or "end". Dates outside of the boundary will be set to NA.
date_start, date_end	column name of a date vector

Examples

```
d <- data.frame(
  s1 = c(as.Date("2013-01-01") + 0:10, as.Date(c("2012-01-01", "2014-01-01"))),
  s2 = c(as.Date("2013-02-01") + 0:10, as.Date(c("2012-01-01", "2014-01-01"))),
  s3 = c(as.Date("2013-01-10") - 0:10, as.Date(c("2012-01-01", "2014-01-01"))),
  ps = as.Date("2012-12-31"),
  pe = as.Date("2013-01-09")
)
print(dd <- find_date_cause(d, s1, s2, s3, period_start = ps, period_end = pe))
print(bb <- find_date_cause(d, s1, s2, s3, period_start = ps, period_end = pe,
                            na_fill = "end",
                            datecol = "enddate",
                            datereason = "endcause"))
find_date_cause(d, s3, s2, s1, period_start = ps, period_end = pe)

# works
assert_positive_timespan(dd, start_date, pe)

# returns a warning because the last date isn't later than the start_date
assert_positive_timespan(dd, start_date, s2)
```

```
with(d, constrain_dates(s1, ps, pe))
with(d, constrain_dates(s2, ps, pe))
with(d, constrain_dates(s3, ps, pe))
```

---

fmt_ci *Helper to format confidence interval for text*

Description

This function is mainly used for placing in the text fields of Rmarkdown reports.

Usage

```
fmt_ci(
  e = numeric(),
  l = numeric(),
  u = numeric(),
  digits = 2,
  percent = TRUE,
  separator = "-"
)

fmt_pci(
  e = numeric(),
  l = numeric(),
  u = numeric(),
  digits = 2,
  percent = TRUE,
  separator = "-"
)

fmt_pci_df(
  x,
  e = 3,
  l = e + 1,
  u = e + 2,
  digits = 2,
  percent = TRUE,
  separator = "-"
)

fmt_ci_df(
  x,
  e = 3,
```

```
  l = e + 1,
  u = e + 2,
  digits = 2,
  percent = TRUE,
  separator = "-"
)
```

Arguments

e	the column of the estimate (defaults to the third column). Otherwise, a number
l	the column of the lower bound (defaults to the fourth column). Otherwise, a number
u	the column of the upper bound (defaults to the fifth column), otherwise, a number
digits	the number of digits to show
percent	if TRUE (default), converts the number to percent, otherwise it's treated as a raw value
separator	what to separate lower and upper confidence intervals with, default is "-"
x	a data frame

Value

a text string in the format of "e\

Examples

```
cfr <- data.frame(x = 1, y = 2, est = 0.5, lower = 0.25, upper = 0.75)
fmt_pci_df(cfr)

# If the data starts at a different column, specify a different number
fmt_pci_df(cfr[-1], 2, d = 1)

# It's also possible to provide numbers directly and remove the percent sign.
fmt_ci(pi, pi - runif(1), pi + runif(1), percent = FALSE)
```

---

fmt_count *Counts and proportions inline*

Description

These functions will give proportions for different variables inline.

Usage

```
fmt_count(x, ...)
```

Arguments

x	a data frame
...	an expression or series of expressions to pass to dplyr::filter()

Value

a one-element character vector of the format "n (%)"

Examples

```
fmt_count(mtcars, cyl > 3, hp < 100)
fmt_count(iris, Species == "virginica")
```

---

gen_polygon *Fake spatial data as polygons This function returns a polygon which is split in to regions based on a supplied vector of names*

Description

Fake spatial data as polygons This function returns a polygon which is split in to regions based on a supplied vector of names

Usage

```
gen_polygon(regions)
```

Arguments

regions	A string of names for each region to label the polygon with

References

The coordinates used for the polygon are of Vienna, Austria. based off government data (see metadata)

gen_population *Generate population counts from estimated population age breakdowns.*

Description

This generates based on predefined age groups and proportions, however you could also define these yourself.

Usage

```
gen_population(
  total_pop = 1000,
  groups = c("0-4", "5-14", "15-29", "30-44", "45+"),
  strata = c("Male", "Female"),
  proportions = c(0.079, 0.134, 0.139, 0.082, 0.066),
  counts = NULL,
  tibble = TRUE
)
```

Arguments

total_pop	The overal population count of interest - the default is 1000 people
groups	A character vector of groups - the default is set for age groups: c("0-4","5-14","15-29","30-44","45+")
strata	A character vector for stratifying groups - the default is set for gender: c("Male", "Female")
proportions	A numeric vector specifying the proportions (as decimals) for each group of the total_pop. The default repeats c(0.079, 0.134, 0.139, 0.082, 0.067) for strata. However you can change this manually, make sure to have the length equal to groups times strata (or half thereof). These defaults are based of MSF general emergency intervention standard values.
counts	A numeric vector specifying the counts for each group. The default is NULL - as most often proportions above will be used. If is not NULL then total_pop and proportions will be ignored. Make sure the length of this vector is equal to groups times strata (or if it is half then it will repeat for each strata). For reference, the MSF general emergency intervention standard values are c(7945, 13391, 13861, 8138, 6665) based on above groups for a 100,000 person population.
tibble	Return data as a tidyverse tibble (default is TRUE)

Examples

```
# get population counts based on proportion, unstratified
gen_population(groups = c(1, 2, 3, 4),
               strata = NULL,
```

```
                proportions = c(0.3, 0.2, 0.4, 0.1))

# get population counts based on proportion, stratified
gen_population(groups = c(1, 2, 3, 4),
                strata = c("a", "b"),
                proportions = c(0.3, 0.2, 0.4, 0.1))

# get population counts based on counts, unstratified
gen_population(groups = c(1, 2, 3, 4),
                strata = NULL,
                counts = c(20, 10, 30, 40))

# get population counts based on counts, stratified
gen_population(groups = c(1, 2, 3, 4),
                strata = c("a", "b"),
                counts = c(20, 10, 30, 40))

# get population counts based on counts, stratified - type out counts
# for each group and strata
gen_population(groups = c(1, 2, 3, 4),
                strata = c("a", "b"),
                counts = c(20, 10, 30, 40, 40, 30, 20, 20))
```

`rename_redundant` *Cosmetically relabel all columns that contains a certain pattern*

Description

These function are only to be used cosmetically before kable and will likely return a data frame with duplicate names.

Usage

```
rename_redundant(x, ...)

augment_redundant(x, ...)
```

Arguments

x	a data frame
...	a series of keys and values to replace columns that match specific patterns.

Details

- rename_redundant fully replaces any column names matching the keys
- augment_redundant will take a regular expression and rename columns via `gsub()`.

Value

a data frame.

Author(s)

Zhian N. Kamvar

Examples

```
df <- data.frame(
  x = letters[1:10],
  `a n` = 1:10,
  `a prop` = (1:10) / 10,
  `a deff` = round(pi, 2),
  `b n` = 10:1,
  `b prop` = (10:1) / 10,
  `b deff` = round(pi * 2, 2),
  check.names = FALSE
)
df
print(df <- rename_redundant(df, "%" = "prop", "Design Effect" = "deff"))
print(df <- augment_redundant(df, " (n)" = " n$"))
```

---

unite_ci *Unite estimates and confidence intervals*

Description

create a character column by combining estimate, lower and upper columns. This is similar to `tidyr::unite()`.

Usage

```
unite_ci(
  x,
  col = NULL,
  ...,
  remove = TRUE,
  digits = 2,
  m100 = TRUE,
  percent = FALSE,
  ci = FALSE,
  separator = "-"
)

merge_ci_df(x, e = 3, l = e + 1, u = e + 2, digits = 2, separator = "-")

merge_pci_df(x, e = 3, l = e + 1, u = e + 2, digits = 2, separator = "-")
```

Arguments

x	a data frame with at least three columns defining an estimate, lower bounds, and upper bounds.
col	the quoted name of the replacement column to create
...	three columns to bind together in the order of Estimate, Lower, and Upper.
remove	if TRUE (default), the three columns in ... will be replaced by col
digits	the number of digits to retain for the confidence interval.
m100	TRUE if the result should be multiplied by 100
percent	TRUE if the result should have a percent symbol added.
ci	TRUE if the result should include "CI" within the braces (defaults to FALSE)
separator	what to separate lower and upper confidence intervals with, default is "-"
e	the column of the estimate (defaults to the third column). Otherwise, a number
l	the column of the lower bound (defaults to the fourth column). Otherwise, a number
u	the column of the upper bound (defaults to the fifth column), otherwise, a number

Value

a modified data frame with merged columns or one additional column representing the estimate and confidence interval

Examples

```
fit <- lm(100/mpg ~ disp + hp + wt + am, data = mtcars)
df  <- data.frame(v = names(coef(fit)), e = coef(fit), confint(fit), row.names = NULL)
names(df) <- c("variable", "estimate", "lower", "upper")
print(df)
unite_ci(df, "slope (CI)", estimate, lower, upper, m100 = FALSE, percent = FALSE)
```

---

zcurve *Create a curve comparing observed Z-scores to the WHO standard.*

Description

Create a curve comparing observed Z-scores to the WHO standard.

Usage

```
zcurve(x, zscore)
```

Arguments

x	a data frame
zscore	bare name of a numeric vector containing computed zscores

Value

a ggplot2 object that is customisable via the ggplot2 package.

Examples

```
library("ggplot2")
set.seed(9)
dat <- data.frame(observed = rnorm(204) + runif(1),
                  skewed   = rnorm(204) + runif(1, 0.5)
                 ) # slightly skewed
zcurve(dat, observed) +
  labs(title = "Weight-for-Height Z-scores") +
  theme_classic()

zcurve(dat, skewed) +
  labs(title = "Weight-for-Height Z-scores") +
  theme_classic()

# Including different groups to facet
dat <- data.frame(
  observed = c(rnorm(204) + runif(1), rnorm(204) + runif(1, 0.5)),
  groups   = rep(c("A", "B"), each = 204),
  treat    = sample(c('up', 'down'), 408, replace = TRUE)
                 )
zcurve(dat, observed) +
  facet_grid(treat~groups)
```

Index