Package ‘surveysd’

February 5, 2020

Type Package

Title Survey Standard Error Estimation for Cumulated Estimates and their Differences in Complex Panel Designs

Version 1.2.0

Maintainer Johannes Gussenbauer <Johannes.Gussenbauer@statistik.gv.at>

Description Calculate point estimates and their standard errors in complex household surveys using bootstrap replicates. Bootstrapping considers survey design with a rotating panel. A comprehensive description of the methodology can be found under <https://statistikat.github.io/surveysd/articles/methodology.html>.

Encoding UTF-8

LazyData true

License GPL (>= 2)

Imports Rcpp (>= 0.12.12),data.table,matrixStats, ggplot2, laeken, methods, dplyr

LinkingTo Rcpp

URL https://github.com/statistikat/surveysd

BugReports https://github.com/statistikat/surveysd/issues

RoxygenNote 7.0.2

Suggests testthat, knitr, rmarkdown

VignetteBuilder knitr

NeedsCompilation yes

Author Johannes Gussenbauer [aut, cre],
Alexander Kowarik [aut] (<https://orcid.org/0000-0001-8598-4130>),
Gregor de Cillia [aut],
Matthias Till [ctb]

Repository CRAN

Date/Publication 2020-02-05 14:40:02 UTC

R topics documented:

`calc.stError` *Calcualte point estimates and their standard errors using bootstrap weights.*

Description

Calculate point estimates as well as standard errors of variables in surveys. Standard errors are estimated using bootstrap weights (see draw.bootstrap and recalib). In addition the standard error of an estimate can be calcualted using the survey data for 3 or more consecutive periods, which results in a reduction of the standard error.

Usage

```
calc.stError(
  dat,
  weights = attr(dat, "weights"),
  b.weights = attr(dat, "b.rep"),
  period = attr(dat, "period"),
  var,
  fun = weightedRatio,
  national = FALSE,
  group = NULL,
  fun.adjust.var = NULL,
  adjust.var = NULL,
  period.diff = NULL,
  period.mean = NULL,
  bias = FALSE,
  size.limit = 20,
  cv.limit = 10,
  p = NULL,
  add.arg = NULL
)
```

Arguments

dat — either data.frame or data.table containing the survey data. Surveys can be a panel survey or rotating panel survey, but does not need to be. For rotating panel survey bootstrap weights can be created using draw.bootstrap and recalib.

weights — character specifying the name of the column in dat containing the original sample weights. Used to calculate point estimates.

b.weights — character vector specifying the names of the columns in dat containing bootstrap weights. Used to calculate standard errors.

period — character specifying the name of the column in dat containing the sample periods.

var — character vector containing variable names in dat on which fun shall be applied for each sample period.

fun — function which will be applied on var for each sample period. Predefined functions are weightedRatio, weightedSum, but can also take any other function which returns a double or integer and uses weights as its second argument.

national — boolean, if TRUE point estimates resulting from fun will be divided by the point estimate at the national level.

group — character vectors or list of character vectors containig variables in dat. For each list entry dat will be split in subgroups according to the containing variables as well as period. The pointestimates are then estimated for each subgroup seperately. If group=NULL the data will split into sample periods by default.

fun.adjust.var — can be either NULL or a function. This argument can be used to apply a function for each period and bootstrap weight to the data. The resulting estimates will be passed down to fun. See details for more explanations.

adjust.var — can be either NULL or a character specifying the first argument in fun.adjust.var.

period.diff — character vectors, defining periods for which the differences in the point estimate as well it's standard error is calculated. Each entry must have the form of "period1 -period2". Can be NULL

period.mean — odd integer, defining the range of periods over which the sample mean of point estimates is additionally calcualted.

bias — boolean, if TRUE the sample mean over the point estimates of the bootstrap weights is returned.

size.limit — integer defining a lower bound on the number of observations on dat in each group defined by period and the entries in group. Warnings are returned if the number of observations in a subgroup falls below size.limit. In addition the concerned groups are available in the function output.

cv.limit — non-negativ value defining a upper bound for the standard error in relation to the point estimate. If this relation exceed cv.limit, for a point estimate, they are flagged and available in the function output.

`p` numeric vector containing values between 0 and 1. Defines which quantiles for the distribution of `var` are additionally estimated.

`add.arg` additional arguments which will be passed to fun. Can be either a named list or vector. The names of the object correspond to the function arguments and the values to column names in dat, see also examples.

Details

`calc.stError` takes survey data (`dat`) and returns point estimates as well as their standard Errors defined by `fun` and `var` for each sample period in `dat`. `dat` must be household data where household members correspond to multiple rows with the same household identifier. The data should at least contain the following columns:

- Column indicating the sample period;
- Column indicating the household ID;
- Column containing the household sample weights;
- Columns which contain the bootstrap weights (see output of recalib);
- Columns listed in `var` as well as in `group`

For each variable in `var` as well as sample period the function `fun` is applied using the original as well as the bootstrap sample weights.
The point estimate is then selected as the result of `fun` when using the original sample weights and it's standard error is estimated with the result of `fun` using the bootstrap sample weights.

`fun` can be any function which returns a double or integer and uses sample weights as it's second argument. The predifined options are `weightedRatio` and `weightedSum`.

For the option `weightedRatio` a weighted ratio (in \ calculated for `var` equal to 1, e.g `sum(weight[var==1])/sum(weight[!`
Additionally using the option `national=TRUE` the weighted ratio (in \ divided by the weighted ratio at the national level for each `period`.
If `group` is not `NULL` but a vector of variables from `dat` then `fun` is applied on each subset of `dat` defined by all combinations of values in `group`.
For instance if `group = "sex"` with "sex" having the values "Male" and "Female" in `dat` the point estimate and standard error is calculated on the subsets of `dat` with only "Male" or "Female" value for "sex". This is done for each value of `period`. For variables in `group` which have `NA`s in `dat` the rows containing the missings will be discarded.
When `group` is a list of character vectors, subsets of `dat` and the following estimation of the point estimate, including the estimate for the standard error, are calculated for each list entry.

The optional parameters `fun.adjust.var` and `adjust.var` can be used if the values in `var` are dependent on the `weights`. As is for instance the case for the poverty thershhold calculated from EU-SILC. In such a case an additional function can be supplied using `fun.adjust.var` as well as its first argument `adjust.var`, which needs to be part of the data set `dat`. Then, before applying `fun` on variable `var` for all `period` and groups, the function `fun.adjust.var` is applied to `adjust.var` using each of the bootstrap weights seperately (NOTE: weight is used as the second argument of `fun.adjust.var`). Thus creating i=1,...,`length(b.weights)` additional variables. For applying `fun` on `var` the estimates for the bootstrap replicate will now use each of the corresponding new additional variables. So instead of

$$fun(var, weights, ...), fun(var, b.weights[1], ...), fun(var, b.weights[2], ...), ...$$

the function `fun` will be applied in the way

$$fun(var, weights, ...), fun(var.1, b.weights[1], ...), fun(var.2, b.weights[2], ...), ...$$

where `var.1`, `var.2`, `...` correspond to the estimates resulting from `fun.adjust.var` and `adjust.var`. NOTE: This procedure is especially usefull if the `var` is dependent on `weights` and `fun` is applied on subgroups of the data set. Then it is not possible to capture this procedure with `fun` and `var`, see examples for a more hands on explanation.
When defining `period.diff` the difference of point estimates between periods as well their standard errors are calculated.
The entries in `period.diff` must have the form of `"period1 -period2"` which means that the results of the point estimates for `period2` will be substracted from the results of the point estimates for `period1`.

Specifying `period.mean` leads to an improvement in standard error by averaging the results for the point estimates, using the bootstrap weights, over `period.mean` periods. Setting, for instance, `period.mean = 3` the results in averaging these results over each consecutive set of 3 periods.
Estimating the standard error over these averages gives an improved estimate of the standard error for the central period, which was used for averaging.
The averaging of the results is also applied in differences of point estimates. For instance defining `period.diff = "2015-2009"` and `period.mean = 3` the differences in point estimates of 2015 and 2009, 2016 and 2010 as well as 2014 and 2008 are calcualated and finally the average over these 3 differences is calculated. The periods set in `period.diff` are always used as the middle periods around which the mean over `period.mean` years is build.
Setting `bias` to `TRUE` returns the calculation of a mean over the results from the bootstrap replicates. In the output the corresponding columns is labeled *_mean* at the end.

If `fun` needs more arguments they can be supplied in `add.arg`. This can either be a named list or vector.

The parameter `size.limit` indicates a lower bound of the sample size for subsets in `dat` created by `group`. If the sample size of a subset falls below `size.limit` a warning will be displayed.
In addition all subsets for which this is the case can be selected from the output of `calc.stError` with $smallGroups.
With the parameter `cv.limit` one can set an upper bound on the coefficient of variantion. Estimates which exceed this bound are flagged with `TRUE` and are available in the function output with $cvHigh. `cv.limit` must be a positive integer and is treated internally as \ for `cv.limit=1` the estimate will be flagged if the coefficient of variantion exceeds 1\
When specifying `period.mean`, the decrease in standard error for choosing this method is internally calcualted and a rough estimate for an implied increase in sample size is available in the output with $stEDecrease. The rough estimate for the increase in sample size uses the fact that for a sample of size n the sample estimate for the standard error of most point estimates converges with a factor $1/\sqrt{n}$ against the true standard error σ.

Value

Returns a list containing:

- Estimates: data.table containing period differences and/or k period averages for estimates of fun applied to var as well as the corresponding standard errors, which are calculated using the bootstrap weights. In addition the sample size, n, and poplutaion size for each group is added to the output.
- smallGroups: data.table containing groups for which the number of observation falls below size.limit.
- cvHigh: data.table containing a boolean variable which indicates for each estimate if the estimated standard error exceeds cv.limit.
- stEDecrease: data.table indicating for each estimate the theoretical increase in sample size which is gained when averaging over k periods. Only returned if period.mean is not NULL.

Author(s)

Johannes Gussenbauer, Alexander Kowarik, Statistics Austria

See Also

draw.bootstrap
recalib

Examples

```
# Import data and calibrate

set.seed(1234)
eusilc <- demo.eusilc(n = 4,prettyNames = TRUE)
dat_boot <- draw.bootstrap(eusilc, REP = 3, hid = "hid", weights = "pWeight",
                           strata = "region", period = "year")
dat_boot_calib <- recalib(dat_boot, conP.var = "gender", conH.var = "region")

# estimate weightedRatio for povertyRisk per period

err.est <- calc.stError(dat_boot_calib, var = "povertyRisk",
                        fun = weightedRatio)
err.est$Estimates

# calculate weightedRatio for povertyRisk and fraction of one-person
# households per period

dat_boot_calib[, onePerson := .N == 1, by = .(year, hid)]
err.est <- calc.stError(dat_boot_calib, var = c("povertyRisk", "onePerson"),
                        fun = weightedRatio)
err.est$Estimates

# estimate weightedRatio for povertyRisk per period and gender

group <- "gender"
err.est <- calc.stError(dat_boot_calib, var = "povertyRisk",
                        fun = weightedRatio, group = group)
err.est$Estimates
```

```
# estimate weightedRatio for povertyRisk per period and gender, region and
# combination of both

group <- list(”gender”, ”region”, c(”gender”, ”region”))
err.est <- calc.stError(dat_boot_calib, var = ”povertyRisk”,
                        fun = weightedRatio, group = group)
err.est$Estimates

# use average over 3 periods for standard error estimation

err.est <- calc.stError(dat_boot_calib, var = ”povertyRisk”,
                        fun = weightedRatio, period.mean = 3)
err.est$Estimates

# get estimate for difference of period 2011 and 2012

period.diff <- c(”2012-2011”)
err.est <- calc.stError(
  dat_boot_calib, var = ”povertyRisk”, fun = weightedRatio,
  period.diff = period.diff, period.mean = 3)
err.est$Estimates

# use add.arg-argument
fun <- function(x, w, b) {
  sum(x*w*b)
}
add.arg = list(b=”onePerson”)

err.est <- calc.stError(dat_boot_calib, var = ”povertyRisk”, fun = fun,
                        period.mean = 0, add.arg=add.arg)
err.est$Estimates
# compare with direkt computation
compare.value <- dat_boot_calib[,fun(povertyRisk,pWeight,b=onePerson),
                                 by=c(”year”)]
all((compare.value$V1-err.est$Estimates$val_povertyRisk)==0)

# use a function from an other package that has sampling weights as its
# second argument
# for example gini() from laeken

library(laeken)

## set up help function that returns only the gini index
help_gini <- function(x, w) {
 return(gini(x, w)$value)
}

## make sure povertyRisk get coerced to a numeric in order to work with the
## external functions
invisible(dat_boot_calib[, povertyRisk := as.numeric(povertyRisk)])

err.est <- calc.stError(
  dat_boot_calib, var = ”povertyRisk”, fun = help_gini, group = group,
  period.diff = period.diff, period.mean = 3)
err.est$Estimates

# using fun.adjust.var and adjust.var to estimate povmd60 indicator
# for each period and bootstrap weight before applying the weightedRatio
# point estimate

# this function estimates the povmd60 indicator with x as income vector
# and w as weight vector
povmd <- function(x, w) {
 md <- laeken::weightedMedian(x, w)*0.6
 pmd60 <- x < md
 return(as.integer(pmd60))
}

# set adjust.var="eqIncome" so the income vector ist used to estimate
# the povmd60 indicator for each bootstrap weight
# and the resultung indicators are passed to function weightedRatio

err.est <- calc.stError(
  dat_boot_calib, var = "povertyRisk", fun = weightedRatio, group = group,
  fun.adjust.var = povmd, adjust.var = "eqIncome", period.mean = 3)
err.est$Estimates

# why fun.adjust.var and adjust.var are needed (!!!):
# one could also use the following function
# and set fun.adjust.var=NULL,adjust.var=NULL
# and set fun = povmd, var = "eqIncome"

povmd2 <- function(x, w) {
 md <- laeken::weightedMedian(x, w)*0.6
 pmd60 <- x < md
 # weighted ratio is directly estimated inside my function
 return(sum(w[pmd60])/sum(w)*100)
}

# but this results in different results in subgroups
# compared to using fun.adjust.var and adjust.var

err.est.different <- calc.stError(
  dat_boot_calib, var = "eqIncome", fun = povmd2, group = group,
  fun.adjust.var = NULL, adjust.var = NULL, period.mean = 3)
err.est.different$Estimates

## results are equal for yearly estimates
all.equal(err.est.different$Estimates[is.na(gender) & is.na(region)],
          err.est$Estimates[is.na(gender)&is.na(region)],
          check.attributes = FALSE)

## but for subgroups (gender, region) results vary
all.equal(err.est.different$Estimates[!(is.na(gender) & is.na(region))],
          err.est$Estimates[!(is.na(gender) & is.na(region))],
          check.attributes = FALSE)
```

`computeLinear` *Numerical weighting functions*

Description

Customize weight-updating within factor levels in case of numerical calibration. The functions described here serve as inputs for ipf.

Usage

```
computeLinear(curValue, target, x, w, boundLinear = 10)

computeLinearG1(curValue, target, x, w, boundLinear = 10)

computeFrac(curValue, target, x, w)
```

Arguments

`curValue`	Current summed up value. Same as `sum(x*w)`
`target`	Target value. An element of `conP` in ipf
`x`	Vector of numeric values to be calibrated against
`w`	Vector of weights
`boundLinear`	The output `f` will satisfy 1/boundLinear <= f <= boundLinear. See bound in ipf

Details

`computeFrac` provides the "standard" IPU updating scheme given as

$$f = target/curValue$$

which means that each weight inside the level will be multtiplied by the same factor when doing the actual update step (`w := f*w`). `computeLinear` on the other hand calculates `f` as

$$f_i = ax_i + b$$

where `a` and `b` are chosen, so f satisfies the following two equations.

$$\sum f_i * w_i * x_i = target$$

$$\sum f_i * w_i = \sum w_i$$

`computeLinearG1` calculates `f` in the same way as `computeLinear`, but if `f_i*w_i<1` `f_i` will be set to `1/w_i`.

Value

A weight multiplier `f`

`cpp_mean` *Calculate mean by factors*

Description

These functions calculate the arithmetic and geometric mean of the weight for each class. `geometric_mean` and `arithmetic_mean` return a `numeric` vector of the same length as `w` which stores the averaged weight for each observation. `geometric_mean_reference` returns the same value by reference, i.e. the input value `w` gets overwritten by the updated weights. See examples.

Usage

```
geometric_mean_reference(w, classes)
```

Arguments

`w` An numeric vector. All entries should be positive.

`classes` A factor variable. Must have the same length as `w`.

Examples

```
## Not run:

## create random data
nobs <- 10
classLabels <- letters[1:3]
dat = data.frame(
  weight = exp(rnorm(nobs)),
  household = factor(sample(classLabels, nobs, replace = TRUE))
)
dat

## calculate weights with geometric_mean
geom_weight <- geometric_mean(dat$weight, dat$household)
cbind(dat, geom_weight)

## calculate weights with arithmetic_mean
arith_weight <- arithmetic_mean(dat$weight, dat$household)
cbind(dat, arith_weight)

## calculate weights "by reference"
geometric_mean_reference(dat$weight, dat$household)
dat

## End(Not run)
```

demo.eusilc *Generate multiple years of EU-SILC data*

Description

Create a dummy dataset to be used for demonstrating the functionalities of the surveysd package based on laeken::eusilc. Please refer to the documentation page of the original data for details about the variables.

Usage

```
demo.eusilc(n = 8, prettyNames = FALSE)
```

Arguments

n	Number of years to generate. Should be at least 1
prettyNames	Create easy-to-read names for certain variables. Recommended for demonstration purposes. Otherwise, use the original codes documented in laeken::eusilc.

Details

If prettyNames is TRUE, the following variables will be available in an easy-to-read manner.

- hid Household id. Consistent with respect to the reference period (year)
- hsize Size of the household. derived from hid and period
- region Federal state of austria where the household is located
- pid Personal id. Consistent with respect to the reference period (year)
- age Age-class of the respondent
- gender A persons gender ("male", "Female")
- ecoStat Ecnomic status ("part time", "full time", "unemployed", ...)
- citizenship Citizenship ("AT", "EU", "other")
- pWeight Personal sample weight inside the reference period
- year. Simulated reference period
- povertyRisk. Logical variable determining whether a respondent is at risk of poverty

Examples

```
demo.eusilc(n = 1, prettyNames = TRUE)[, c(1:8, 26, 28:30)]
```

draw.bootstrap *Draw bootstrap replicates*

Description

Draw bootstrap replicates from survey data with rotating panel design. Survey information, like ID, sample weights, strata and population totals per strata, should be specified to ensure meaningfull survey bootstraping.

Usage

```
draw.bootstrap(
  dat,
  REP = 1000,
  hid = NULL,
  weights,
  period = NULL,
  strata = NULL,
  cluster = NULL,
  totals = NULL,
  single.PSU = c("merge", "mean"),
  boot.names = NULL,
  split = FALSE,
  pid = NULL,
  new.method = FALSE
)
```

Arguments

`dat` either data.frame or data.table containing the survey data with rotating panel design.

`REP` integer indicating the number of bootstrap replicates.

`hid` character specifying the name of the column in `dat` containing the household id. If `NULL` (the default), the household structure is not regarded.

`weights` character specifying the name of the column in `dat` containing the sample weights.

`period` character specifying the name of the column in `dat` containing the sample periods. If `NULL` (the default), it is assumed that all observations belong to the same period.

`strata` character vector specifying the name(s) of the column in `dat` by which the population was stratified. If `strata` is a vector stratification will be assumed as the combination of column names contained in `strata`. Setting in addition `cluster` not NULL stratification will be assumed on multiple stages, where each additional entry in `strata` specifies the stratification variable for the next lower stage. see Details for more information.

`cluster` character vector specifying cluster in the data. If not already specified in `cluster` household ID is taken es the lowest level cluster.

`totals` character specifying the name of the column in `dat` containing the the totals per strata and/or cluster. Is ONLY optional if `cluster` is `NULL` or equal `hid` and `strata` contains one columnname! Then the households per strata will be calcualted using the `weights` argument. If clusters and strata for multiple stages are specified `totals` needs to be a vector of `length(strata)` specifying the column on `dat` that contain the total number of PSUs at each stage. `totals` is interpreted from left the right, meaning that the first argument corresponds to the number of PSUs at the first and the last argument to the number of PSUs at the last stage.

`single.PSU` either "merge" or "mean" defining how single PSUs need to be dealt with. For `single.PSU="merge"` single PSUs at each stage are merged with the strata or cluster with the next least number of PSUs. If multiple of those exist one will be select via random draw. For `single.PSU="mean"` single PSUs will get the mean over all bootstrap replicates at the stage which did not contain single PSUs.

`boot.names` character indicating the leading string of the column names for each bootstrap replica. If NULL defaults to "w".

`split` logical, if TRUE split households are considered using `pid`, for more information see Details.

`pid` column in `dat` specifying the personal identifier. This identifier needs to be unique for each person throught the whole data set.

`new.method` logical, if TRUE bootstrap replicates will never be negative even if in some strata the whole population is in the sample. WARNING: This is still experimental and resulting standard errors might be underestimated! Use this if for some strata the whole population is in the sample!

Details

`draw.bootstrap` takes `dat` and draws `REP` bootstrap replicates from it. `dat` must be household data where household members correspond to multiple rows with the same household identifier. For most practical applications, the following columns should be available in the dataset and passed via the corresponding parameters:

- Column indicating the sample period (parameter `period`).
- Column indicating the household ID (parameter `hid`)
- Column containing the household sample weights (parameter `weights`);
- Columns by which population was stratified during the sampling process (parameter: `strata`).

For single stage sampling design a column the argument `totals` is optional, meaning that a column of the number of PSUs at the first stage does not need to be supplied. For this case the number of PSUs is calculated and added to `dat` using `strata` and `weights`. By setting `cluster` to NULL single stage sampling design is always assumed and if `strata` contains of multiple column names the combination of all those column names will be used for stratification.

In the case of multi stage sampling design the argument `totals` needs to be specified and needs to have the same number of arguments as `strata`.

If `cluster` is `NULL` or does not contain `hid` at the last stage, `hid` will automatically be used as the final cluster. If, besides `hid`, clustering in additional stages is specified the number of column names in strata and cluster (including hid) must be the same. If for any stage there was no clustering or stratification one can set "1" or "I" for this stage.

For example strata=c("REGION","I"),cluster=c("MUNICIPALITY","HID") would speficy a 2 stage sampling design where at the first stage the municipalities where drawn stratified by regions and at the 2nd stage housholds are drawn in each municipality without stratification.

Bootstrap replicates are drawn for each survey period (period) using the function rescaled.bootstrap. Afterwards the bootstrap replicates for each household are carried forward from the first period the household enters the survey to all the censecutive periods it stays in the survey.

This ensures that the bootstrap replicates follow the same logic as the sampled households, making the bootstrap replicates more comparable to the actual sample units.

If split ist set to TRUE and pid is specified, the bootstrap replicates are carried forward using the personal identifiers instead of the houshold identifier. This takes into account the issue of a houshold splitting up. Any person in this new split household will get the same bootstrap replicate as the person that has come from an other household in the survey. People who enter already existing households will also get the same bootstrap replicate as the other households members had in the previous periods.

Value

the survey data with the number of REP bootstrap replicates added as columns.

Returns a data.table containing the original data as well as the number of REP columns containing the bootstrap replicates for each repetition.
The columns of the bootstrap replicates are by default labeled "w*Number*" where *Number* goes from 1 to REP. If the column names of the bootstrap replicates should start with a different character or string the parameter boot.names can be used.

Author(s)

Johannes Gussenbauer, Alexander Kowarik, Statistics Austria

See Also

data.table for more information on data.table objects.

Examples

```
## Not run:
eusilc <- demo.eusilc(prettyNames = TRUE)

## draw sample without stratification or clustering
dat_boot <- draw.bootstrap(eusilc, REP = 10, weights = "pWeight",
                           period = "year")

## use stratification w.r.t. region and clustering w.r.t. households
dat_boot <- draw.bootstrap(
  eusilc, REP = 10, hid = "hid", weights = "pWeight",
  strata = "region", period = "year")

## use multi-level clustering
```

```
dat_boot <- draw.bootstrap(
  eusilc, REP = 10, hid = "hid", weights = "pWeight",
  strata = c("region", "age"), period = "year")


# create spit households
eusilc[, pidsplit := pid]
year <- eusilc[, unique(year)]
year <- year[-1]
leaf_out <- c()
for(y in year) {
  split.person <- eusilc[
    year == (y-1) & !duplicated(hid) & !(hid %in% leaf_out),
    sample(pid, 20)
  ]
  overwrite.person <- eusilc[
    (year == (y)) & !duplicated(hid) & !(hid %in% leaf_out),
    .(pid = sample(pid, 20))
  ]
  overwrite.person[, c("pidsplit", "year_curr") := .(split.person, y)]

  eusilc[overwrite.person, pidsplit := i.pidsplit,
         on = .(pid, year >= year_curr)]
  leaf_out <- c(leaf_out,
                eusilc[pid %in% c(overwrite.person$pid,
                                  overwrite.person$pidsplit),
                unique(hid)])
}

dat_boot <- draw.bootstrap(
  eusilc, REP = 10, hid = "hid", weights = "pWeight",
  strata = c("region", "age"), period = "year", split = TRUE,
  pid = "pidsplit")
# split households were considered e.g. household and
# split household were both selected or not selected
dat_boot[, data.table::uniqueN(w1), by = pidsplit][V1 > 1]

## End(Not run)
```

`generate.HHID` *Generate new houshold ID for survey data with rotating panel design taking into account split households*

Description

Generating a new houshold ID for survey data using a houshold ID and a personal ID. For surveys with rotating panel design containing housholds, houshold members can move from an existing household to a new one, that was not originally in the sample. This leads to the creation of so called split households. Using a peronal ID (that stays fixed over the whole survey), an indicator for different time steps and a houshold ID, a new houshold ID is assigned to the original and the split household.

Usage

```
generate.HHID(dat, period = "RB010", pid = "RB030", hid = "DB030")
```

Arguments

`dat`	data table of data frame containing the survey data
`period`	column name of `dat` containing an indicator for the rotations, e.g years, quarters, months, ect...
`pid`	column name of `dat` containing the personal identifier. This needs to be fixed for an indiviual throught the whole survey
`hid`	column name of `dat` containing the household id. This needs to for a household throught the whole survey

Value

the survey data `dat` as data.table object containing a new and an old household ID. The new household ID which considers the split households is now named `hid` and the original household ID has a trailing "_orig".

Examples

```
## Not run:
library(surveysd)
library(laeken)
library(data.table)

eusilc <- surveysd:::demo.eusilc(n=4)

# create spit households
eusilc[,rb030split:=rb030]
year <- eusilc[,unique(year)]
year <- year[-1]
leaf_out <- c()
for(y in year) {
  split.person <- eusilc[year==(y-1)&!duplicated(db030)&!db030%in%leaf_out,
                         sample(rb030,20)]
  overwrite.person <- eusilc[year==(y)&!duplicated(db030)&!db030%in%leaf_out,
                             .(rb030=sample(rb030,20))]
  overwrite.person[,c("rb030split","year_curr"):=.(split.person,y)]

  eusilc[overwrite.person,
         rb030split:=i.rb030split,on=.(rb030,year>=year_curr)]
  leaf_out <- c(
    leaf_out,
    eusilc[rb030%in%c(overwrite.person$rb030,overwrite.person$rb030split),
    unique(db030)])
}
```

```
# pid which are in split households
eusilc[,.(uniqueN(db030)),by=list(rb030split)][V1>1]

eusilc.new <- generate.HHID(eusilc, period = "year", pid = "rb030split",
                            hid = "db030")

# no longer any split households in the data
eusilc.new[,.(uniqueN(db030)),by=list(rb030split)][V1>1]

## End(Not run)
```

---

ipf *Iterative Proportional Fitting*

Description

Adjust sampling weights to given totals based on household-level and/or individual level constraints.

Usage

```
ipf(
  dat,
  hid = NULL,
  conP = NULL,
  conH = NULL,
  epsP = 1e-06,
  epsH = 0.01,
  verbose = FALSE,
  w = NULL,
  bound = 4,
  maxIter = 200,
  meanHH = TRUE,
  allPthenH = TRUE,
  returnNA = TRUE,
  looseH = FALSE,
  numericalWeighting = computeLinear,
  check_hh_vars = TRUE,
  conversion_messages = FALSE,
  nameCalibWeight = "calibWeight"
)
```

Arguments

`dat` a `data.table` containing household ids (optionally), base weights (optionally), household and/or personal level variables (numerical or categorical) that should be fitted.

`hid` name of the column containing the household-ids within `dat` or NULL if such a variable does not exist.

`conP` list or (partly) named list defining the constraints on person level. The list elements are contingency tables in array representation with dimnames corresponding to the names of the relevant calibration variables in `dat`. If a numerical variable is to be calibrated, the respective list element has to be named with the name of that numerical variable. Otherwise the list element shoud NOT be named.

`conH` list or (partly) named list defining the constraints on household level. The list elements are contingency tables in array representation with dimnames corresponding to the names of the relevant calibration variables in `dat`. If a numerical variable is to be calibrated, the respective list element has to be named with the name of that numerical variable. Otherwise the list element shoud NOT be named.

`epsP` numeric value or list (of numeric values and/or arrays) specifying the convergence limit(s) for `conP`. The list can contain numeric values and/or arrays which must appear in the same order as the corresponding constraints in `conP`. Also, an array must have the same dimensions and dimnames as the corresponding constraint in `conP`.

`epsH` numeric value or list (of numeric values and/or arrays) specifying the convergence limit(s) for `conH`. The list can contain numeric values and/or arrays which must appear in the same order as the corresponding constraints in `conH`. Also, an array must have the same dimensions and dimnames as the corresponding constraint in `conH`.

`verbose` if TRUE, some progress information will be printed.

`w` name if the column containing the base weights within `dat` or NULL if such a variable does not exist. In the latter case, every observation in `dat` is assigned a starting weight of 1.

`bound` numeric value specifying the multiplier for determining the weight trimming boundary if the change of the base weights should be restricted, i.e. if the weights should stay between `1/bound*w` and `bound*w`.

`maxIter` numeric value specifying the maximum number of iterations that should be performed.

`meanHH` if TRUE, every person in a household is assigned the mean of the person weights corresponding to the household. If `"geometric"`, the geometric mean is used rather than the arithmetic mean.

`allPthenH` if TRUE, all the person level calibration steps are performed before the houshold level calibration steps (and `meanHH`, if specified). If FALSE, the houshold level calibration steps (and `meanHH`, if specified) are performed after everey person level calibration step. This can lead to better convergence properties in certain cases but also means that the total number of calibration steps is increased.

`returnNA` if TRUE, the calibrated weight will be set to NA in case of no convergence.

`looseH` if FALSE, the actual constraints `conH` are used for calibrating all the hh weights. If TRUE, only the weights for which the lower and upper thresholds defined by `conH` and `epsH` are exceeded are calibrated. They are however not calibrated against the actual constraints `conH` but against these lower and upper thresholds, i.e. `conH-conH*epsH` and `conH+conH*epsH`.

numericalWeighting
: See numericalWeighting

check_hh_vars
: If TRUE check for non-unique values inside of a household for variables in household constraints

conversion_messages
: show a message, if inputs need to be reformatted. This can be useful for speed optimizations if ipf is called several times with similar inputs (for example bootstrapping)

nameCalibWeight
: character defining the name of the variable for the newly generated calibrated weight.

Details

This function implements the weighting procedure described here. Usage examples can be found in the corresponding vignette (vignette("ipf")).

conP and conH are contingency tables, which can be created with xtabs. The dimnames of those tables should match the names and levels of the corresponding columns in dat.

maxIter, epsP and epsH are the stopping criteria. epsP and epsH describe relative tolerances in the sense that

$$1 - epsP < \frac{w_{i+1}}{w_i} < 1 + epsP$$

will be used as convergence criterium. Here i is the iteration step and wi is the weight of a specific person at step i.

The algorithm performs best if all varables occuring in the constraints (conP and conH) as well as the household variable are coded as factor-columns in dat. Otherwise, conversions will be necessary which can be monitored with the conversion_messages argument. Setting check_hh_vars to FALSE can also incease the performance of the scheme.

Value

The function will return the input data dat with the calibrated weights calibWeight as an additional column as well as attributes. If no convergence has been reached in maxIter steps, and returnNA is TRUE (the default), the column calibWeights will only consist of NAs. The attributes of the table are attributes derived from the data.table class as well as the following.

converged	Did the algorithm converge in maxIter steps?
iterations	The number of iterations performed.
conP, conH, epsP, epsH	See Arguments.
conP_adj, conH_adj	Adjusted versions of conP and conH
formP, formH	Formulas that were used to calculate conP_adj and conH_adj based on the output table.

Author(s)

Alexander Kowarik, Gregor de Cillia

Examples

```
## Not run:

# load data
eusilc <- demo.eusilc(n = 1, prettyNames = TRUE)

# personal constraints
conP1 <- xtabs(pWeight ~ age, data = eusilc)
conP2 <- xtabs(pWeight ~ gender + region, data = eusilc)
conP3 <- xtabs(pWeight*eqIncome ~ gender, data = eusilc)

# household constraints
conH1 <- xtabs(pWeight ~ hsize + region, data = eusilc)

# simple usage ------------------------------------------

calibweights1 <- ipf(
  eusilc,
  conP = list(conP1, conP2, eqIncome = conP3),
  bound = NULL,
  verbose = TRUE
)

# compare personal weight with the calibweigth
calibweights1[, .(hid, pWeight, calibWeight)]

# advanced usage ----------------------------------------

# use an array of tolerances
epsH1 <- conH1
epsH1[1:4, ] <- 0.005
epsH1[5, ] <- 0.2

# create an initial weight for the calibration
eusilc[, regSamp := .N, by = region]
eusilc[, regPop := sum(pWeight), by = region]
eusilc[, baseWeight := regPop/regSamp]

calibweights2 <- ipf(
  eusilc,
  conP = list(conP1, conP2),
  conH = list(conH1),
  epsP = 1e-6,
  epsH = list(epsH1),
  bound = 4,
  w = "baseWeight",
  verbose = TRUE
)

# show an adjusted version of conP and the original
attr(calibweights2, "conP_adj")
attr(calibweights2, "conP")
```

```
## End(Not run)
```

ipf_step *Perform one step of iterative proportional updating*

Description

C++ routines to invoke a single iteration of the Iterative proportional updating (IPU) scheme. Targets and classes are assumed to be one dimensional in the `ipf_step` functions. `combine_factors` aggregates several vectors of type factor into a single one to allow multidimensional ipu-steps. See examples.

Usage

```
ipf_step_ref(w, classes, targets)

ipf_step(w, classes, targets)

ipf_step_f(w, classes, targets)

combine_factors(dat, targets)
```

Arguments

`w`	a numeric vector of weights. All entries should be positive.
`classes`	a factor variable. Must have the same length as `w`.
`targets`	key figure to target with the ipu scheme. A numeric verctor of the same length as `levels(classes)`. This can also be a `table` produced by `xtabs`. See examples.
`dat`	a `data.frame` containing the factor variables to be combined.

Details

`ipf_step` returns the adjusted weights. `ipf_step_ref` does the same, but updates `w` by reference rather than returning. `ipf_step_f` returns a multiplicator: adjusted weights divided by unadjusted weights. `combine_factors` is designed to make `ipf_step` work with contingency tables produced by xtabs.

Examples

```
############# one-dimensional ipu ##############

## create random data
nobs <- 10
classLabels <- letters[1:3]
dat = data.frame(
  weight = exp(rnorm(nobs)),
  household = factor(sample(classLabels, nobs, replace = TRUE))
)
dat

## create targets (same lenght as classLabels!)
targets <- 3:5

## calculate weights
new_weight <- ipf_step(dat$weight, dat$household, targets)
cbind(dat, new_weight)

## check solution
xtabs(new_weight ~ dat$household)

## calculate weights "by reference"
ipf_step_ref(dat$weight, dat$household, targets)
dat

############# multidimensional ipu ##############

## load data
factors <- c("time", "sex", "smoker", "day")
tips <- data.frame(sex=c("Female","Male","Male"), day=c("Sun","Mon","Tue"),
time=c("Dinner","Lunch","Lunch"), smoker=c("No","Yes","No"))
tips <- tips[factors]

## combine factors
con <- xtabs(~., tips)
cf <- combine_factors(tips, con)
cbind(tips, cf)[sample(nrow(tips), 10, replace = TRUE),]

## adjust weights
weight <- rnorm(nrow(tips)) + 5
adjusted_weight <- ipf_step(weight, cf, con)

## check outputs
con2 <- xtabs(adjusted_weight ~ ., data = tips)
sum((con - con2)^2)
```

kishFactor *Kish Factor*

Description

Compute the kish factor for a specific weight vector

Usage

```
kishFactor(w)
```

Arguments

`w`	a numeric vector with weights

Value

The function will return the the kish factor

Author(s)

Alexander Kowarik

Examples

```
kishFactor(rep(1,10))
kishFactor(rlnorm(10))
```

`plot.surveysd` *Plot surveysd-Objects*

Description

Plot results of `calc.stError()`

Usage

```
## S3 method for class 'surveysd'
plot(
  x,
  variable = x$param$var[1],
  type = c("summary", "grouping"),
  groups = NULL,
  sd.type = c("dot", "ribbon"),
  ...
)
```

Arguments

`x`	object of class 'surveysd' output of function calc.stError
`variable`	Name of the variable for which standard errors have been calcualated in `dat`
`type`	can bei either `"summary"` or `"grouping"`, default value is `"summary"`. For `"summary"` a barplot is created giving an overview of the number of estimates having the flag `smallGroup`, `cvHigh`, both or none of them. For 'grouping' results for point estimate and standard error are plotted for pre defined groups.
`groups`	If `type='grouping'` variables must be defined by which the data is grouped. Only 2 levels are supported as of right now. If only one group is defined the higher group will be the estimate over the whole period. Results are plotted for the first argument in `groups` as well as for the combination of `groups[1]` and `groups[2]`.

`sd.type` can bei either `'ribbon'` or `'dot'` and is only used if `type='grouping'`. Default is `"dot"` For `sd.type='dot'` point estimates are plotted and flagged if the corresponding standard error and/or the standard error using the mean over k-periods exceeded the value `cv.limit` (see calc.stError). For `sd.type='ribbon'` the point estimates including ribbons, defined by point estimate +- estimated standard error are plotted. The calculated standard errors using the mean over k periods are plotted using less transparency. Results for the higher level (~`groups[1]`) are coloured grey.

`...` additional arguments supplied to plot.

Examples

```
library(surveysd)
library(laeken)
library(data.table)

eusilc <- demo.eusilc(n = 4, prettyNames = TRUE)

dat_boot <- draw.bootstrap(eusilc, REP = 3, hid = "hid", weights = "pWeight",
                           strata = "region", period = "year")

# calibrate weight for bootstrap replicates
dat_boot_calib <- recalib(dat_boot, conP.var = "gender", conH.var = "region")

# estimate weightedRatio for povmd60 per period
group <- list("gender", "region", c("gender", "region"))
err.est <- calc.stError(dat_boot_calib, var = "povertyRisk",
                        fun = weightedRatio,
                        group = group , period.mean = NULL)


plot(err.est)

# plot results for gender
# dotted line is the result on the national level
plot(err.est, type = "grouping", groups = "gender")

# plot results for gender
# with standard errors as ribbons
plot(err.est, type = "grouping", groups = "gender", sd.type = "ribbon")

# plot results for rb090 in each db040
plot(err.est, type = "grouping", groups = c("gender", "region"))

# plot results for db040 in each rb090 with standard errors as ribbons
plot(err.est,type = "grouping", groups = c("gender", "region"))
```

PointEstimates *Weighted Point Estimates*

Description

Predefined functions for weighted point estimates in package surveysd.

Usage

```
weightedRatio(x, w)

weightedSum(x, w)
```

Arguments

x	numeric vector
w	weight vector

Details

Predefined functions are weighted ratio and weighted sum.

Value

Each of the functions return a single numeric value

Examples

```
x <- 1:10
w <- 10:1
weightedRatio(x,w)
x <- 1:10
w <- 10:1
weightedSum(x,w)
```

print.surveysd *Print function for surveysd objects*

Description

Prints the results of a call to calc.stError. Shows used variables and function, number of point estiamtes as well as properties of the results.

Usage

```
## S3 method for class 'surveysd'
print(x, ...)
```

Arguments

`x` an object of class `'surveysd'`

`...` additonal parameters

`recalib` *Calibrate weights*

Description

Calibrate weights for bootstrap replicates by using iterative proportional updating to match population totals on various household and personal levels.

Usage

```
recalib(
  dat,
  hid = attr(dat, "hid"),
  weights = attr(dat, "weights"),
  b.rep = attr(dat, "b.rep"),
  period = attr(dat, "period"),
  conP.var = NULL,
  conH.var = NULL,
  epsP = 0.01,
  epsH = 0.02,
  ...
)
```

Arguments

`dat` either data.frame or data.table containing the sample survey for various periods.

`hid` character specifying the name of the column in `dat` containing the household ID.

`weights` character specifying the name of the column in `dat` containing the sample weights.

`b.rep` character specifying the names of the columns in `dat` containing bootstrap weights which should be recalibratet

`period` character specifying the name of the column in `dat` containing the sample period.

`conP.var` character vector containig person-specific variables to which weights should be calibrated or a list of such character vectors. Contingency tables for the population are calculated per `period` using `weights`.

`conH.var` character vector containig household-specific variables to which weights should be calibrated or a list of such character vectors. Contingency tables for the population are calculated per `period` using `weights`.

`epsP` numeric value specifying the convergence limit for `conP.var` or `conP`, see `ipf()`.

`epsH` numeric value specifying the convergence limit for `conH.var` or `conH`, see `ipf()`.

`...` additional arguments passed on to function `ipf()` from this package.

Details

`recalib` takes survey data (`dat`) containing the bootstrap replicates generated by draw.bootstrap and calibrates weights for each bootstrap replication according to population totals for person- or household-specific variables.
`dat` must be household data where household members correspond to multiple rows with the same household identifier. The data should at least containt the following columns:

- Column indicating the sample period;
- Column indicating the household ID;
- Column containing the household sample weights;
- Columns which contain the bootstrap replicates (see output of draw.bootstrap);
- Columns indicating person- or household-specific variables for which sample weight should be adjusted.

For each period and each variable in `conP.var` and/or `conH.var` contingency tables are estimated to get margin totals on personal- and/or household-specific variables in the population.
Afterwards the bootstrap replicates are multiplied with the original sample weight and the resulting product ist then adjusted using `ipf()` to match the previously calcualted contingency tables. In this process the columns of the bootstrap replicates are overwritten by the calibrated weights.

Value

Returns a data.table containing the survey data as well as the calibrated weights for the bootstrap replicates. The original bootstrap replicates are overwritten by the calibrated weights. If calibration of a bootstrap replicate does not converge the bootsrap weight is not returned and numeration of the returned bootstrap weights is reduced by one.

Author(s)

Johannes Gussenbauer, Alexander Kowarik, Statistics Austria

See Also

`ipf()` for more information on iterative proportional fitting.

Examples

```
## Not run:

eusilc <- demo.eusilc(prettyNames = TRUE)

dat_boot <- draw.bootstrap(eusilc, REP = 10, hid = "hid",
                           weights = "pWeight",
                           strata = "region", period = "year")

# calibrate weight for bootstrap replicates
dat_boot_calib <- recalib(dat_boot, conP.var = "gender", conH.var = "region",
                          verbose = TRUE)
```

```
# calibrate on other variables
dat_boot_calib <- recalib(dat_boot, conP.var = c("gender", "age"),
                          conH.var = c("region", "hsize"), verbose = TRUE)

# supply contingency tables directly
conP <- xtabs(pWeight ~ age + gender + year, data = eusilc)
conH <- xtabs(pWeight ~ hsize + region + year,
              data = eusilc[!duplicated(paste(db030,year))])
dat_boot_calib <- recalib(dat_boot, conP.var = NULL,
                          conH.var = NULL, conP = list(conP),
                          conH = list(conH), verbose = TRUE)

## End(Not run)
```

`rescaled.bootstrap` *Draw bootstrap replicates*

Description

Draw bootstrap replicates from survey data using the rescaled bootstrap for stratified multistage sampling, presented by Preston, J. (2009).

Usage

```
rescaled.bootstrap(
  dat,
  REP = 1000,
  strata = "DB050>1",
  cluster = "DB060>DB030",
  fpc = "N.cluster>N.households",
  single.PSU = c("merge", "mean"),
  return.value = c("data", "replicates"),
  check.input = TRUE,
  new.method = FALSE
)
```

Arguments

`dat`	either data frame or data table containing the survey sample
`REP`	integer indicating the number of bootstraps to be drawn
`strata`	string specifying the column name in `dat` that is used for stratification. For multistage sampling multiple column names can be specified by `strata=c("strata1>strata2>strata3")`. See Details for more information.

`cluster` string specifying the column name in `dat` that is used for clustering. For instance given a household sample the column containing the household ID should be supplied. For multistage sampling multiple column names can be specified by `cluster=c("cluster1>cluster2>cluster3")`. See Details for more information.

`fpc` string specifying the column name in `dat` that contains the number of PSUs at the first stage. For multistage sampling the number of PSUs at each stage must be specified by `strata=c("fpc1>fpc2>fpc3")`.

`single.PSU` either "merge" or "mean" defining how single PSUs need to be dealt with. For `single.PSU="merge"` single PSUs at each stage are merged with the strata or cluster with the next least number of PSUs. If multiple of those exist one will be select via random draw. For `single.PSU="mean"` single PSUs will get the mean over all bootstrap replicates at the stage which did not contain single PSUs.

`return.value` either "data" or "replicates" specifying the return value of the function. For "data" the survey data is returned as class `data.table`, for "replicates" only the bootstrap replicates are returned as `data.table`.

`check.input` logical, if TRUE the input will be checked before applying the bootstrap procedure

`new.method` logical, if TRUE bootstrap replicates will never be negative even if in some strata the whole population is in the sample. WARNING: This is still experimental and resulting standard errors might be underestimated! Use this if for some strata the whole population is in the sample!

Details

For specifying multistage sampling designs the column names in `strata`,`cluster` and `fpc` need to seperated by ">".
For multistage sampling the strings are read from left to right meaning that the column name before the first ">" is taken as the column for stratification/clustering/number of PSUs at the first and the column after the last ">" is taken as the column for stratification/clustering/number of PSUs at the last stage. If for some stages the sample was not stratified or clustered one must specify this by "1" or "I", e.g. `strata=c("strata1>I>strata3")` if there was no stratification at the second stage or `cluster=c("cluster1>cluster2>I")` if there were no clusters at the last stage.
The number of PSUs at each stage is not calculated internally and must be specified for any sampling design. For single stage sampling using stratification this can usually be done by adding over all sample weights of each PSU by each strata-code.
Spaces in each of the strings will be removed, so if column names contain spaces they should be renamed before calling this procedure!

Value

returns the complete data set including the bootstrap replicates or just the bootstrap replicates, depending on `return.value="data"` or `return.value="replicates"` respectively.

Author(s)

Johannes Gussenbauer, Statistics Austria

References

Preston, J. (2009). Rescaled bootstrap for stratified multistage sampling. Survey Methodology. 35. 227-234.

Examples

```
data(eusilc, package = "laeken")
data.table::setDT(eusilc)

eusilc[,N.households:=sum(db090[!duplicated(db030)]),by=db040]
eusilc.bootstrap <- rescaled.bootstrap(eusilc,REP=100,strata="db040",
                                       cluster="db030",fpc="N.households")

eusilc[,new_strata:=paste(db040,rb090,sep="_")]
eusilc[,N.housholds:=sum(db090[!duplicated(db030)]),by=new_strata]
eusilc.bootstrap <- rescaled.bootstrap(eusilc,REP=100,strata=c("new_strata"),
                                       cluster="db030",fpc="N.households")

eusilc.bootstrap <- rescaled.bootstrap(eusilc,REP=100,strata=c("new_strata"),
                                       cluster="db030",fpc="N.households")
```

Index