Package ‘ReGenesees’

December 6, 2011

Type Package

Title R evolved Generalised software for sampling estimates and errors in surveys.

Description Design-Based and Model-Assisted analysis of complex sampling surveys.
Multistage, stratified, clustered, unequally weighted survey designs.
Horvitz-Thompson and Calibration Estimators. Variance Estimation for nonlinear
smooth estimators by Taylor-series linearization. Estimates, standard errors,confidence intervals and design effects for: Totals, Means, absolute and relative
Frequency Distributions (marginal or joint), Ratios and Quantiles. Automated
Linearization of Complex Analytic Estimators. Estimates, standard errors,confidence intervals and design effects for user-defined analytic estimators.
Estimates and sampling errors for subpopulations.

Version 1.0

Author Diego Zardetto

Maintainer Diego Zardetto <zardetto@istat.it>

License file LICENCE

Imports stats

Depends R (>= 2.14.0)

Suggests MASS

R topics documented:

`aux.estimates` *Quick Estimates of Auxiliary Variables Totals*

Description

Quickly estimates the totals of the auxiliary variables of a calibration model.

Usage

```
aux.estimates(design,
              calmodel = if (inherits(template, "pop.totals"))
                         attr(template, "calmodel"),
              partition = if (inherits(template, "pop.totals"))
                          attr(template, "partition") else FALSE,
              template = NULL)
```

Arguments

`design`	Object of class `analytic` (or inheriting from it) containing survey data and sampling design metadata.
`calmodel`	Formula defining the linear structure of the calibration model.
`partition`	Formula specifying the variables that define the "calibration domains" for the model (see 'Details'); `FALSE` (the default) implies no calibration domains.
`template`	An object of class `pop.totals`, be it a template or the actual known totals data frame for the calibration task.

Details

The main purpose of function `aux.estimates` is to make easy the task of estimating the totals of *all* the auxiliary variables involved in a calibration model (separately inside distinct calibration domains, if specified). Even if such totals can be estimated also by repeatedly invoking function `svystatTM`, this may reveal very tricky in practice, because real-world calibration tasks (e.g. in the field of Official Statistics) can simultaneously involve several hundreds of auxiliary variables. Moreover, total estimates provided by function `svystatTM` are always complemented by sampling errors, whose estimation is very computationally demanding.

Function aux.estimates, on the contrary, *only* provides estimates of totals (i.e. without associated sampling errors), thus being very quick to be executed. Moreover, aux.estimates is able to compute, *in just a single shot*, all the totals of the auxiliary variables of a calibration model, no matter how complex the model is. Lastly, as a third strong point, the totals estimated by aux.estimates will be returned exactly in the same *standard format* in which the known population totals for the related calibration task need to be represented (see pop.template, population.check, fill.template).

It may be useful to point out that, besides having been designed to handle auxiliary variables involved in calibration models, function aux.estimates could be also used for computing *general* estimates of totals inside subpopulations in a very effective way (see 'Examples').

Value

An object of class pop.totals, thus inheriting from class data.frame storing the estimated totals in a standard format.

Author(s)

Diego Zardetto

See Also

e.svydesign to bind survey data and sampling design metadata, svystatTM for calculating estimates and standard errors of totals, e.calibrate for calibrating weights, pop.template for constructing known totals data frames in compliance with the standard required by e.calibrate, population.check to check that the known totals data frame satisfies that standard, fill.template to automatically fill the template when a sampling frame is available.

Examples

```
# Load sbs data:
data(sbs)

# Build a design object:
sbsdes<-e.svydesign(data=sbs,ids=~id,strata=~strata,weights=~weight,fpc=~fpc)

# Now suppose you have to perform a calibration process which
# exploits as auxiliary information:
#  i) the total number of employees (emp.num)
#     by class of number of employees (emp.cl) crossed with nace.macro;
# ii) the total number of enterprises (ent)
#     by region crossed with nace.macro;

# Build a template for the known totals:
pop<-pop.template(sbsdes,
     calmodel=~emp.num:emp.cl + region -1,
     partition=~nace.macro)

# Use the fill.template function to automatically compute
# the totals from the universe (sbs.frame) and safely fill
# the template:
pop<-fill.template(sbs.frame,template=pop)
pop

# You can now use aux.estimates to verify how much difference
# exists between the target totals and the initial HT estimates:
```

```
aux.HT<-aux.estimates(sbsdes,template=pop)
aux.HT

# If you calibrate, ...
sbscal<-e.calibrate(sbsdes,pop)

# ... you can verify that CAL estimates exactly match the known totals:
aux.CAL<-aux.estimates(sbscal,template=pop)
aux.CAL

# Recall that you can also use aux.estimates for computing
# general estimates of totals inside subpopulations (even
# not related to any calibration task).
# E.g. estimate the total of value added inside areas:
aux.estimates(sbsdes,~va.imp2-1,~area)

# ...and compare to svystatTM (notice also
# the increased execution time):
svystatTM(sbsdes,~va.imp2,~area)
```

---

bounds.hint *A hint for range restricted calibration*

Description

Suggests a sound bounds value for which `e.calibrate` is likely to converge.

Usage

```
bounds.hint(design, df.population,
    calmodel = if (inherits(df.population, "pop.totals"))
                    attr(df.population, "calmodel"),
    partition = if (inherits(df.population, "pop.totals"))
                     attr(df.population, "partition") else FALSE,
    msg = TRUE)
```

Arguments

`design`	Object of class `analytic` (or inheriting from it) containing survey data and sampling design metadata.
`df.population`	Data frame containing the known population totals for the auxiliary variables.
`calmodel`	Formula defining the linear structure of the calibration model.
`partition`	Formula specifying the variables that define the "calibration domains" for the model; `FALSE` (the default) implies no calibration domains.
`msg`	Enables printing of a summary description of the result (default is `TRUE`).

Details

The function `bounds.hint` returns a bounds value for which `e.calibtrate` is *likely* to converge. This interval is just a sound hint, *not* an exact result (see 'Note').

The mandatory argument `design` identifies the `analytic` object on which the calibration problem is defined.

The mandatory argument df.population identifies the known totals data frame.

The argument calmodel symbolically defines the calibration model you want to use: it identifies the auxiliary variables and the constraints for the calibration problem. The design variables referenced by calmodel must be numeric or factor and must not contain any missing value (NA). The argument can be omitted provided df.population is an object of class pop.totals (see population.check).

The optional argument partition specifies the variables that define the calibration domains for the model. The default value (FALSE) means either that there are not calibration domains or that you want to solve the problem globally (even though it could be factorised). The design variables referenced by partition (if any) must be factor and must not contain any missing value (NA). The argument can be omitted provided df.population is an object of class pop.totals (see population.check).

The optional argument msg enables/disables printing of a summary description of the achieved result.

Value

A numeric vector of length 2, representing the *suggested* value for the bounds argument of e.calibrate. The attributes of that vector store additional information, which can lead to better understand why a given calibration problem is (un)feasible (see 'Examples').

Note

Assessing the feasibility of an arbitrary calibration problem is not an easy task. The problem is even more difficult whenever additional *"range restrictions"* are imposed. Indeed, even if one assumes that the calibration constraints define a consistent system, one also has to choose the bounds such that the feasible region is non-empty.

One can argue that there must exist a minimun-length interval $I = [L, U]$ such that, if it is covered by bounds, the specified calibration problem is feasible. Unfortunately in order to compute exactly that minimun-length interval I one should solve a big linear programming problem [Vanderhoeft 01]. As an alternative, a trial and error procedure has been frequently proposed [Deville et al 1993; Sautory 1993]: (i) start with a very large interval bounds.0; (ii) if convergence is achieved, shrink it so as to obtain a new inteval bounds.1; (iii) repeat until you get a sufficiently tight feasible interval bounds.n. The drawback is that this procedure can cost a lot of computer time since, for each choice of the bounds, the full calibration problem has to be solved.

A rather easy task is, on the contrary, the one of finding at least a given specific interval $I^* = [L^*, U^*]$ such that, if it is *not* covered by bounds, the current calibration problem is *surely unfeasible*. This means that any feasible bounds value must necessarily contain the I^* interval. The function bounds.hint: (i) first identifies such an I^* interval (by computing the range of the ratios between known population totals and corresponding direct Horvitz-Thompson estimates), (ii) then builds a new interval I^{sugg} with same midpoint and double length. The latter is the *suggested* value for the bounds argument of e.calibrate. The return value of bounds.hint should be understood as a useful starting guess for bounds, even though there is definitely no warranty that the calibration algorithm will actually converge.

Author(s)

Diego Zardetto

References

Vanderhoeft, C. (2001) *"Generalized Calibration at Statistic Belgium"*, Statistics Belgium Working Paper n. 3, http://www.statbel.fgov.be/studies/paper03_en.asp.

Deville, J.C., Sarndal, C.E. and Sautory, O. (1993) *"Generalized Raking Procedures in Survey Sampling"*, Journal of the American Statistical Association, Vol. 88, No. 423, pp.1013-1020.

Sautory, O. (1993) *"La macro CALMAR: Redressement d'un Echantillon par Calage sur Marges"*, Document de travail de la Direction des Statistiques Demographiques et Sociales, no. F9310.

See Also

e.calibrate for calibrating weights, pop.template for constructing known totals data frames in compliance with the standard required by e.calibrate, population.check to check that the known totals data frame satisfies that standard and g.range to compute the range of the obtained g-weights.

Examples

```
# Creation of the object to be calibrated:
data(data.examples)
des<-e.svydesign(data=example,ids=~towcod+famcod,strata=~SUPERSTRATUM,
     weights=~weight)

# Calibration (iterative solution) on the marginal distribution
# of age in 5 classes (age5c) inside provinces (procod)
# (totals in pop06p). Get a hint for feasible bounds:
hint<-bounds.hint(des,pop06p,~age5c-1,~procod)

# Let's verify if calibration converges with the suggested
# value for the bounds argument (i.e. c(0.219, 1.786) ):
descal06p<-e.calibrate(design=des,df.population=pop06p,
           calmodel=~age5c-1,partition=~procod,calfun="logit",
           bounds=hint,aggregate.stage=2)

# Now let's verify that calibration fails, if bounds don't cover
# the interval [0.611, 1.394]:
## Not run:
descal06p<-e.calibrate(design=des,df.population=pop06p,
           calmodel=~age5c-1,partition=~procod,calfun="logit",
           bounds=c(0.62,1.50),aggregate.stage=2,force=FALSE)

## End(Not run)
# The warning message raised by e.calibrate tells that
# the population total of variable age5c5 (i.e. the fifth
# age class frequency) was not matched.

# By analysing ecal.status one understands that calibration
# failed due to the sub-task identified by procod 30:
ecal.status

# this is easily explained by inspectioning the "bounds"
# attribute of the bounds.hint output object:
attr(hint,"bounds")

# indeed the specified lower bound (0.62) was too high
# for procod 30, where instead a value ~0.61 was required.
```

```
# Recall that you can always "force" a calibration task that
# would not converge:
descal06p.forced<-e.calibrate(design=des,df.population=pop06p,
                  calmodel=~age5c-1,partition=~procod,calfun="logit",
                  bounds=c(0.62,1.50),aggregate.stage=2,force=TRUE)

# Notice, also, that forced sub-tasks can be tracked down by
# looking at ecal.status:
ecal.status
```

`check.cal` *Calibration Convergence Check*

Description

Checks whether Calibration Constraints are fulfilled; if not, assesses constraints violation degree.

Usage

```
check.cal(cal.design)
```

Arguments

`cal.design`	Object of class `cal.analytic`.

Details

The function verifies if all the imposed Calibration Constraints are actually fulfilled by object `cal.design`. If it is not the case, the function evaluates the degree of violation of the constraints and prints a summary of the mismatches between population totals and achieved estimates (see also Section 'Calibration process diagnostics' in the help page of `e.calibrate`).

Value

The main purpose of the function is to print on screen; anyway a list is invisibly returned, which summarizes the results of the check.

Author(s)

Diego Zardetto

See Also

`e.calibrate` for calibrating weights.

Examples

```
# Load sbs data:
data(sbs)

# Build a design object:
sbsdes<-e.svydesign(data=sbs,ids=~id,strata=~strata,weights=~weight,fpc=~fpc)

## Example 1
 # Build template...
   pop<-pop.template(sbsdes,~emp.num:emp.cl+ent-1,~region)
 # Fill template...
   pop<-fill.template(sbs.frame,pop)
 # Calibrate...
   sbscal<-e.calibrate(sbsdes,pop,sigma2=~emp.num)
 # Check calibration...
   check.cal(sbscal)

## Example 2
 # Build template...
   pop<-pop.template(sbsdes,~emp.num+ent-1,~area)
 # Fill template...
   pop<-fill.template(sbs.frame,pop)
 # Calibrate with tight bounds...
   sbscal<-e.calibrate(sbsdes,pop,sigma2=~emp.num,bounds=c(0.8,1.2))
 # Check calibration...
   check.cal(sbscal)

 # Now try to calibrate with suggested bounds...
   hint <- bounds.hint(sbsdes,pop)
   sbscal<-e.calibrate(sbsdes,pop,sigma2=~emp.num,bounds=hint)
 # Check calibration...
   check.cal(sbscal)
```

---

collapse.strata *Collapse strata technique for eliminating lonely PSUs*

Description

Modifies a stratified design containing lonely PSUs by collapsing its design strata into superstrata.

Usage

```
collapse.strata(design, block.vars = NULL, sim.score = NULL)
```

Arguments

`design`	Object of class `analytic` (or inheriting from it) containing survey data and sampling design metadata.
`block.vars`	Formula specifying blocking variables: only strata belonging to the same block will be aggregated (see 'Details'). If `NULL` (the default option) no constraints will be imposed.
`sim.score`	Formula specifying a similarity score for strata: lonely strata will be paired with the most similar stratum in each block (see 'Details'). If `NULL` (the default option) random pairs will be formed.

Details

Lonely PSUs (i.e. PSUs which are alone inside a not self-representing stratum) are a concern from the viewpoint of variance estimation. As a general solution, the **ReGenesees** package can handle the lonely PSUs problem by setting proper variance estimation options (see `ReGenesees.options`). The `collapse.strata` function implements a widely used alternative: the so called collapsed strata technique. The basic idea is to build artificial *"superstrata"* by aggregating strata containing lonely PSUs to other strata, and then to use such superstrata for variance estimation (see e.g. [Wolter 85] and [Rust, Kalton 87]).

The optional argument `block.vars` identifies *"blocking variables"* that can be used to constrain the way lonely strata are collapsed to form superstrata. More specifically: first, blocking variables are used to partition sample data in *"blocks"* via factor crossing, then, only lonely strata belonging to the same block are aggregated. If `block.vars=NULL` (the default option), no constraint will act on collapsing. The `design` variables referenced by `block.vars` (if any) should be of type `factor`. Errors will be raised if (i) blocks cut across strata, or (ii) `block.vars` generate any non-aggregable strata (i.e. lonely strata which are a singleton inside a block).

The optional argument `sim.score` can be used to specify a similarity score for strata aggregation. This means that each lonely stratum will be collapsed with the stratum that has the most similar value of variable `sim.score` inside the block. Thus the similarity of two strata is actually measured by the (absolute value of the) difference among the corresponding `sim.score` values. Only one `design` variable can be referenced by the `sim.score` formula: (i) it must be of type `numeric`, (ii) it must be constant inside each stratum, and (iii) it should be positive (otherwise its `abs()` will be silently used). Note that if no similarity score is specified (i.e. `sim.score=NULL`), the achieved strata aggregation will depend on the ordering of input sample data in `design`.

The collapsing algorithm will, whenever possible, build superstrata by pairing a lonely stratum to another not-yet-aggregated stratum. Therefore, in general, superstrata will contain only two design strata. Rare exceptions can arise, e.g. due to constraints, with at most three design strata inside a superstratum. The choice to collapse strata in pairs has been taken because it is known to be appropriate for large-scale surveys with many strata (at least for national level estimates, see e.g. [Rust, Kalton 87]).

The `collapse.strata` function handles correctly finite population corrections. If `design` has been built by passing strata sampling fractions via the `fpc` argument, the function re-computes sampling fractions inside superstrata by exploiting the achieved mapping of strata to superstrata and the `fpc` slot of `design`.

Value

An object of the same class as `design`, without strata containing lonely PSUs.

Strata collapse process diagnostics

As already observed in the 'Details' Section, there are three non trivial reasons why function `collapse.strata` can run into errors: (1) the blocks cut across strata, (2) some blocks contain a stratum needing to be aggregated while this stratum happens to be the only one inside the block, (3) the similarity score for strata aggregation varies inside strata. In order to help the user to identify such data anomalies, hence taking a step forward to eliminate them, every call to `collapse.strata` generates, by side effect, a diagnostics data structure named `clps.strata.status` into the `.GlobalEnv` (see 'Examples').
The `clps.strata.status` list has three components: the first reports the error message, the second stores a vector identifying the data subsets that have been hit by the anomaly, the third reports the call to `collapse.strata` that generated the list. For instance, when error condition (1) holds, the second element of `clps.strata.status` identifies the strata that are cut by blocks; if, instead, error

condition (2) holds, the second element of the list identifies the blocks containing non-aggregable strata.
It must be stressed that *every call* to `collapse.strata` generates the `clps.strata.status` list, *even* when the strata collapsing process ends *succesfully*. In such cases, the first element of the list reports the number of lonely strata that have undergone aggregation, whereas the second is a useful dataframe (named `clps.table`) mapping collapsed strata to superstrata. To be more specific: each row of `clps.table` identifies a stratum that has been mapped to a superstratum, while the columns of `clps.table` give: (i) the block to which the stratum belongs, (ii) the stratum name, (iii) a flag indicating if the stratum was lonely or not, (iv) the name of the superstratum to which it has been mapped.

Methodological warning

A warning must be emphasized: strata similarity score `sim.score` should be based on prior knowledge and/or on expectations on *true* values of stratum means for the variable(s) to be estimated, not on current sample data. Indeed, building `sim.score` by estimating stratum means with the current sample can lead to severe *underestimation* of sampling variance, i.e. to too tight confidence intervals.

Author(s)

Diego Zardetto

References

Wolter, K.M. (1985) *"Introduction to Variance Estimation"*, Springer-Verlag, New York.

Rust, K., Kalton, G. (1987) *"Strategies for Collapsing Strata for Variance Estimation"*, Journal of Official Statistics, Vol. 3, No. 1, pp. 69-81.

See Also

`ReGenesees.options` for a different way to handle the lonely PSUs problem (namely by setting variance estimation options).

Examples

```
# Build a survey design with lonely PSU strata:
data(data.examples)
exdes <- e.svydesign(data= example, ids= ~ towcod+famcod,
         strata= ~ stratum, weights= ~ weight)
exdes

# Explore 3 possible collapsing strategies:
  # 1) Aggregate lonely strata by forming random pairs
exdes.clps1 <- collapse.strata(exdes)
exdes.clps1

  # 2) Aggregate lonely strata in pairs under constraints:
  #    i.  aggregated strata must be both not self-representing
  #    ii. aggregated strata must belong to the same province (which
  #        is appropriate if e.g. provinces are planned estimation domains)
exdes.clps2 <- collapse.strata(exdes,~sr:procod)
exdes.clps2

  # 3) A WRONG strategy: compute strata similarity score by using
```

```
  #    sample estimates of the interest variable (y1) inside strata:
old.op <- options("RG.lonely.psu"="remove")
stat.score <- svystatTM(design= exdes, ~y1, by= ~ stratum)
options(old.op)
exdes2 <- des.addvars(exdes,
          sim.score= sapply(stratum, function(str)
                                     stat.score[stat.score[,"stratum"]==str,2]))
exdes.clps3 <- collapse.strata(exdes2,~sr:procod,~sim.score)
exdes.clps3

# Compute total estimates of y1 at the province level
# for all 3 designs with collapsed strata:
stat.clps1 <- svystatTM(design= exdes.clps1, y= ~ y1, by= ~ procod,
              estimator= "Total", vartype= "cvpct")
stat.clps2 <- svystatTM(design= exdes.clps2, y= ~ y1, by= ~ procod,
              estimator= "Total", vartype= "cvpct")
stat.clps3 <- svystatTM(design= exdes.clps3, y= ~ y1, by= ~ procod,
              estimator= "Total", vartype= "cvpct")

# Compute the same estimates by using two alternatives
# to handle lonely PSUs:
  # "adjust" option
old.op <- options("RG.lonely.psu"="adjust")
stat.adj <- svystatTM(design= exdes, y= ~ y1, by= ~ procod,
            estimator= "Total", vartype= "cvpct")
options(old.op)
 # "everage" option
old.op <- options("RG.lonely.psu"="average")
stat.ave <- svystatTM(design= exdes, y= ~ y1, by= ~ procod,
            estimator= "Total", vartype= "cvpct")
options(old.op)

# Lastly, compare achieved estimates for CV percentages:
stat.clps1
stat.clps2
stat.clps3
stat.adj
stat.ave

# Thus the qualitative features are as expected: the "adjust" option
# tends to give conservative sampling variance estimates, the WRONG collapsing
# strategy 3) tends to underestimate sampling variance, while other methods
# give results in-between those extrema.


# Few examples to inspect the clps.strata.status list generated
# for diagnostics purposes:
  # 1) Ill defined blocks: cutting across strata:
## Not run:
clps.err1 <- collapse.strata(exdes,~sex)

## End(Not run)
clps.strata.status

  # 2) Ill defined blocks: generating non-aggregable strata
## Not run:
clps.err2 <- collapse.strata(exdes,~regcod:stratum)
```

```
## End(Not run)
clps.strata.status

  # 3) Successful collapsing: explore strata to superstrata mapping
exdes.ok <- collapse.strata(exdes,~sr:regcod:procod)
clps.strata.status
```

`data.examples` *Example data for the ReGenesees package*

Description

Example data frames and functions. Allow to run R code contained in the 'Examples' section of the ReGenesees package help pages.

Usage

```
data(data.examples)
```

Format

The main data frame, named `example`, contains (artificial) data from a two stage stratified cluster sampling design. The sample is made up of 3000 final units, for which the following 21 variables were observed:

- `towcod` Code identifying "variance PSUs": towns (PSUs) in not-self-representing (NSR) strata, families (SSUs) in self-representing (SR) strata, `numeric`
- `famcod` Code identifying families (SSUs), `numeric`
- `key` Key identifying final units (individuals), `numeric`
- `weight` Initial weights, `numeric`
- `stratum` Stratification variable, `factor` with levels 801 802 803 901 902 903 904 905 906 907 908 1001 1002 1003 1004 1005 1006 1007 1008 1009 1101 1102 1103 1104 3001 3002 3003 3004 3005 3006 3007 3008 3009 3010 3011 3012 3101 3102 3103 3104 3105 3106 3107 3108 3201 3202 3203 3204 5401 5402 5403 5404 5405 5406 5407 5408 5409 5410 5411 5412 5413 5414 5415 5416 5501 5502 5503 5504 9301 9302 9303 9304 9305 9306 9307 9308 9309 9310 9311 9312
- `SUPERSTRATUM` Collapsed strata variable (eliminates lonely PSUs), `factor` with levels 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55
- `sr` Strata type, `integer` with values 0 (NSR strata) and 1 (SR strata)
- `regcod` Code identifying regions, `factor` with levels 6 7 10
- `procod` Code identifying provinces, `factor` with levels 8 9 10 11 30 31 32 54 55 93
- `x1` Indicator variable (integer), `numeric`
- `x2` Indicator variable (integer), `numeric`
- `x3` Indicator variable (integer), `numeric`
- `y1` Indicator variable (integer), `numeric`
- `y2` Indicator variable (integer), `numeric`

y3 Indicator variable (integer), numeric

age5c Age variable with 5 classes, factor with levels 1 2 3 4 5

age10c Age variable with 10 classes, factor with levels 1 2 3 4 5 6 7 8 9 10

sex Sex variable, factor with levels f m

marstat Marital status variable, factor with levels married unmarried widowed

z A continuous quantitative variable, numeric

income Income variable, numeric

Details

Objects pop01, ..., pop07pp contain known population totals for various calibration models. Object pairs with names differing in the 'p' suffix (such as pop03 and pop03p) refer to the *same* calibration problem but pertain to *different* solution methods (global and iterative respectively, see e.calibrate). The two-component numeric vector bounds expresses a possible choice for the allowed range for the ratios between calibrated weights and direct weights in the aforementioned calibration problems.

Examples

```
data(data.examples)
str(example)
```

---

des.addvars *Add variables to design objects*

Description

Modifies an analytic object by adding new variables to it.

Usage

```
des.addvars(design, ...)
```

Arguments

design	Object of class analytic (or inheriting from it) containing survey data and sampling design metadata.
...	tag = expr arguments defining columns to be added to design.

Details

This function adds to the data frame contained in design the *new* variables defined by the tag = expr arguments. A tag can be specified either by means of an identifier or by a character string; expr can be any expression that it makes sense to evaluate in the design environment.

For each argument tag = expr bound to the formal argument ... the added column will have *name* given by the tag value and *values* obtained by evaluating the expr expression on design. Any input expression not supplied with a tag will be ignored and will therefore have no effect on the des.addvars return value.

Variables to be added to the input object have to be *new*: namely it is not possible to use des.addvars to modify the values in a pre-existing design column.

Value

An object of the same class of `design`, containing new variables but supplied with exactly the same metadata.

Author(s)

Diego Zardetto

See Also

`e.svydesign` to bind survey data and sampling design metadata, `e.calibrate` for calibrating weights.

Examples

```
data(data.examples)

# Creation of an analytic object:
des<-e.svydesign(data=example,ids=~towcod+famcod,strata=~SUPERSTRATUM,
     weights=~weight)

# Adding the new 'ones' variable to estimate the number
# of final units in the population:
des<-des.addvars(des,ones=1)
svystatTM(des,~ones)

# Recoding a qualitative variable:
des<-des.addvars(des,agerange=factor(ifelse(age5c==1,
                                     "young","not-young")))
svystatTM(des,~agerange,estimator="Mean")
svystatTM(des,~income,~agerange,estimator="Mean",conf.int=TRUE)

# Algebraic operations on numeric variables:
des<-des.addvars(des,z2=z^2)
svystatTM(des,~z2,estimator="Mean")

# A more interesting example: estimating the
# percentage of population with income below
# the poverty threshold (defined as 0.6 times
# the average income for the whole population):
Mean.Income <- coef(svystatTM(des, ~income,estimator="Mean"))
des <- des.addvars(des,
                   status = factor(
                          ifelse(income < (0.6 * Mean.Income),
                          "poor","not-poor")
                                  )
                   )
svystatTM(des,~status,estimator="Mean")
# Mean income for poors and not-poors:
svystatTM(des,~income,~status,estimator="Mean")
```

e.calibrate *Calibration of survey weights*

Description

Adds to an analytic object the calibrated weights column.

Usage

```
e.calibrate(design, df.population,
            calmodel = if (inherits(df.population, "pop.totals"))
                       attr(df.population, "calmodel"),
            partition = if (inherits(df.population, "pop.totals"))
                        attr(df.population, "partition") else FALSE,
            calfun = c("linear", "raking", "logit"),
            bounds = c(-Inf, Inf), aggregate.stage = NULL,
            sigma2 = NULL, maxit = 50, epsilon = 1e-07, force = TRUE)
```

Arguments

design	Object of class analytic (or inheriting from it) containing survey data and sampling design metadata.
df.population	Data frame containing the known population totals for the auxiliary variables.
calmodel	Formula defining the linear structure of the calibration model.
partition	Formula specifying the variables that define the "calibration domains" for the model (see 'Details'); FALSE (the default) implies no calibration domains.
calfun	character specifying the distance function for the calibration process; the default is 'linear'.
bounds	Allowed range for the ratios between calibrated and initial weights; the default is c(-Inf,Inf).
aggregate.stage	An integer: if specified, causes the calibrated weights to be constant within sampling units at this stage.
sigma2	Formula specifying a possible heteroskedasticity effect in the calibration model; NULL (the default) implies homoskedasticity.
maxit	Maximum number of iterations for the Newton-Raphson algorithm; the default is 50.
epsilon	Tolerance for the absolute relative differences between the population totals and the corresponding estimates based on the calibrated weights; the default is 10^-7.
force	If TRUE, whenever the calibration algorithm does not converge, forces the function to return a value (see 'Details' and 'Calibration process diagnostics'); the default is TRUE.

Details

This function creates an object of class cal.analytic. A cal.analytic object makes it possible to compute estimates and standard errors of calibration estimators [Deville, Sarndal 92] [Deville, Sarndal, Sautory 93].

The mandatory argument calmodel symbolically defines the calibration model you intend using, that is - in the language of the Generalised Regression Estimator - the assisting linear regression model underlying the calibration problem [Wilkinson, Rogers 73]. More specifically, the calmodel formula identifies the auxiliary variables and the constraints for the calibration problem. For example, calmodel=~(X+Z):C+(A+B):D defines the calibration problem in which constraints are imposed: (i) on the auxiliary (quantitative) variables X and Z within the subpopulations identified by the (qualitative) classification variable C and, at the same time, (ii) on the absolute frequency of the (qualitative) variables A and B within the subpopulations identified by the (qualitative) classification variable D.
The design variables referenced by calmodel must be numeric or factor and must not contain any missing value (NA).

Problems for which one or more qualitative variables can be *"factorised"* in the formula that specifies the calibration model, are particularly interesting. These variables split the population into non-overlapping subpopulations known as *"calibration domains"* for the model. An example is provided by the statement calmodel=~(A+B+X+Z):D in which the variable that identifies the calibration domains is D; similarly, the formula calmodel=~(A+B+X+Z):D1:D2 identifies as calibration domains the subpopulations determined by crossing the modalities of D1 and D2. The interest in models of this kind lies in the fact that the *global* calibration problem they describe can, actually, be broken down into *local* subproblems, one per calibration domain, which can be solved separately [Vanderhoeft 01]. Thus, for example, the global problem defined by calmodel=~(A+B+X+Z):D is equivalent to the sequence of problems defined by the *"reduced model"* calmodel=~A+B+X+Z in each of the domains identified by the modalities of D. The opportunity to separately solve the subproblems related to different calibration domains achieves a significant reduction in computation complexity: the gain increases with increasing survey data size and (most importantly) with increasing auxiliary variables number.

The optional argument partition makes it possible to choose, in cases in which the calibration problem can be factorised, whether to solve the problem globally or iteratively (that is, separately for each calibration domain). The global solution (which is the default option) can be selected invoking the e.calibrate function with partition=FALSE. To request the iterative solution - a strongly recommended option when dealing with a lot of auxiliary variables and big data sizes - it is necessary to specify via partition the variables defining the calibration domains for the model. If a formula is passed through the partition argument (for example: partition=~D1:D2), the program checks that calmodel actually describes a "reduced model" (for example: calmodel=~X+Z+A+B), that is it does not reference any of the partition variables; if this is not the case, the program stops and prints an error message.
The design variables referenced by partition (if any) must be factor and must not contain any missing value (NA).

The mandatory argument df.population is used to specify the known totals of the auxiliary variables referenced by calmodel within the subpopulations (if any) identified by partition. These known totals must be stored in a data frame whose structure (i) depends on the values of calmodel and partition and (ii) must conform to a standard. In order to facilitate understanding of and compliance with this standard, the **ReGenesees** package provides the user with three functions: pop.template, population.check, and fill.template. The pop.template function is able to guide the user in constructing the known totals data frame for a specific calibration problem, the fill.template function can be exploited to automatically fill the template when a sampling frame is available, while the population.check function allows to check whether a known totals data frame conforms to the standard required by e.calibrate. In any case, if the df.population data frame does not comply with the standard, the e.calibrate function stops and prints an error message: the meaning of the message should help the user diagnose the cause of the problem.

The calfun argument identifies the distance function to be used in the calibration process. Three built-in functions are provided: "linear", "raking", and "logit" (see [Deville, Sarndal, Sautory 93]). The default is "linear", which corresponds to the euclidean metric and yields the Generalised Regression Estimator (provided that no range restrictions are imposed on the g-weights). The "raking" distance corresponds to the *"multiplicative method"* of [Deville, Sarndal, Sautory 93].

The bounds argument allows to add *"range constraints"* to the calibration problem. To be precise, the interval defined by bounds will contain the values of the ratios between final (calibrated) and initial (direct) weights. The default value is c(-Inf,Inf), i.e. no range constraints are imposed. These constraints are optional unless the "logit" function is selected: in the latter case the range defined by bounds has to be finite (see, again, [Deville, Sarndal, Sautory 93]).

The value passed by the aggregate.stage argument must be an integer between 1 and the number of sampling stages of design. If specified, causes the calibrated weights to be constant within sampling units selected at the aggregate.stage stage (actually this is only allowed if the initial weights had already this property, as it is sometimes the case in multistage cluster sampling). If not specified, the calibrated weights may differ even for sampling units with identical initial weights. The same holds if some final units belonging to the same cluster selected at the stage aggregate.stage fall in distinct calibration domains (i.e. if the domains defined by partition "cut across" the aggregate.stage-stage clusters).

The argument sigma2 can be used to take into account a possible heteroskedasticity effect in the (assisting linear regression model underlying the) calibration problem. In such cases, sigma2 must identify some variable to which the variances of the error terms are believed to be proportional. Notice that sigma2 can also be interpreted from a "purely calibration-based" point of view: it corresponds to the $1/q_k$ unit-weights appearing inside the distance measures of [Deville, Sarndal 92] [Deville, Sarndal, Sautory 93]. The final effect is, on average, that calibrated weights associated to higher values of sigma2 tend to stay closer to their corresponding initial weights. The sigma2 formula can reference just a single design variable: such variable must be numeric, strictly positive and must not contain NAs. If aggregate.stage is specified, sigma2 must obviously be constant inside aggregate.stage-stage clusters (otherwise the function stops and prints an error message).

The maxit argument sets the maximum number of iteration for the Newton-Raphson algorithm that is used to solve the calibration problem (the only exception being *unbounded linear* calibration, i.e. calfun='linear' and bounds=c(-Inf, Inf), which is actually handled by directly solving a linear problem). The default value of maxit is 50.

The epsilon argument determines the convergence criterion for the optimisation algorithm: it fixes the maximum allowed absolute value for the relative differences between the population totals and the corresponding estimates based on the calibrated weights. The default value is 10^-7.

The calibrated weights computed by e.calibrate must ensure that the calibration estimators of the auxiliary variables *exactly* match the corresponding known population totals. It is, however, possible (more likely when range constraints are imposed) that, for a specific calibration problem and for given values of epsilon and maxit, the solving algorithm does not converge. In this case, if force = FALSE, e.calibrate stops and prints an error message. If - on the contrary - force = TRUE, the function is forced to return the best approximation achieved for the calibrated weights, neverthless signaling the calibration failure by a warning (see also Section 'Calibration process diagnostics').

Value

An object of class cal.analytic. The data frame it contains includes (in addition to the data already stored in design) the calibrated weights columns. The name of this column is obtained by pasting the name of the initial weights column with the string ".cal".

Calibration process diagnostics

When, dealing with a factorisable calibration problem, the user selects the iterative solution, the global calibration problem gets split into as many *sub-problems* as the number of subpopulations defined by partition. In turn, each one of these calibration sub-problems can end without convergence on any one of the involved auxiliary variables. A calibration process with such a complex structure needs some ad hoc tool for error diagnostics. For this purpose, every call to e.calibrate creates, by side effect, a dedicated data structure named ecal.status into the .GlobalEnv. ecal.status is a list with up to three components: the first, "call", identifies the call to e.calibrate that generated the list, the second, return.code, is a matrix each element of which identifies the return code of a specific calibration sub-problem. The meaning of the return codes is as follows:

-1 not yet tackled sub-problem;

0 solved sub-problem (convergence achieved);

1 unsolved sub-problem (no convergence): output forced.

Recall that the latter return code may only occur if force = TRUE.
If any return.code equal to 1 exists, the ecal.status list gains a third component named "fail.diagnostics" which is itself a list; its components correspond to sub-problems for which convergence was not achieved, and store useful information about the auxiliary variables for which calibration constraints are violated. Therefore, users can exploit ecal.status to identify sub-problems and variables from which errors stemmed, hence taking a step forward to eliminate them.
Notice, lastly, that the ecal.status list will also be persistently bound to the e.calibrate return object, stored inside a dedicated attribute. For the inspection of such diagnostics information the check.cal function is available.

Note

The cal.analytic class is a specialisation of the analytic class; this means that an object created by e.calibrate inherits from the analytic class and you can use on it every method defined on the latter class. For instance, a calibrated design can be passed again to e.calibrate, thus undergoing further calibration steps.

Author(s)

Diego Zardetto

References

Deville, J.C., Sarndal, C.E. (1992) *"Calibration Estimators in Survey Sampling"*, Journal of the American Statistical Association, Vol. 87, No. 418, pp. 376-382.

Deville, J.C., Sarndal, C.E., Sautory, O. (1993) *"Generalized Raking Procedures in Survey Sampling"*, Journal of the American Statistical Association, Vol. 88, No. 423, pp. 1013-1020.

Wilkinson, G.N., Rogers, C.E. (1973) *"Symbolic Description of Factorial Models for Analysis of Variance"*, Journal of the Royal Statistical Society, series C (Applied Statistics), Vol. 22, pp. 181-191.

Vanderhoeft, C. (2001) *"Generalized Calibration at Statistic Belgium"*, Statistics Belgium Working Paper n. 3, http://www.statbel.fgov.be/studies/paper03_en.asp.

Scannapieco, M., Zardetto, D., Barcaroli, G. (2007) *"La Calibrazione dei Dati con R: una Sperimentazione sull'Indagine Forze di Lavoro ed un Confronto con GENESEES/SAS"*, Contributi Istat n. 4., http://www.istat.it/dati/pubbsci/contributi/Contributi/contr_2007/2007_4.pdf.

See Also

e.svydesign to bind survey data and sampling design metadata, svystatTM, svystatR, svystatQ and svystatL for calculating estimates and standard errors, pop.template for constructing known totals data frames in compliance with the standard required by e.calibrate, population.check to check that the known totals data frame satisfies that standard, fill.template to automatically fill the template when a sampling frame is available, bounds.hint to obtain a hint for range restricted calibration, g.range to asses the variation of weights after calibration and check.cal to check if calibration constraints have been fulfilled.

Examples

```
######################################################################
# Calibration of a design object according to different calibration  #
# models (the known totals data frames pop01, ..., pop05p and the    #
# bounds vector are all contained in the data.examples file).        #
# For the examples relating to calibration models that can be        #
# factorised both a global and an iterative solution are given.      #
######################################################################

data(data.examples)

# Creation of the object to be calibrated:
des<-e.svydesign(data=example,ids=~towcod+famcod,strata=~SUPERSTRATUM,
      weights=~weight)


# 1) Calibration on the total number of units in the population
#   (totals in pop01):
descal01<-e.calibrate(design=des,df.population=pop01,calmodel=~1,
          calfun="logit",bounds=bounds,aggregate.stage=2)

# Checking the result (first add the new 'ones' variable
# to estimate the number of final units in the population):
descal01<-des.addvars(descal01,ones=1)
svystatTM(descal01, ~ones)


# 2) Calibration on the marginal distributions of sex and marstat
#    (totals in pop02):
descal02<-e.calibrate(design=des,df.population=pop02,
          calmodel=~sex+marstat-1,calfun="logit",bounds=bounds,
          aggregate.stage=2)

# Checking the result:
svystatTM(descal02,~sex+marstat)


# 3) Calibration (global solution) on the joint distribution of sex
#    and marstat (totals in pop03):
descal03<-e.calibrate(design=des,df.population=pop03,
          calmodel=~marstat:sex-1,calfun="logit",bounds=bounds)
```

```
# Checking the result:
svystatTM(descal03,~sex,~marstat) # or: svystatTM(descal03,~marstat,~sex)

# which, obviously, is not respected by descal02 (notice the size of SE):
svystatTM(descal02,~sex,~marstat)


# 3.1) Again a calibration on the joint distribution of sex and marstat
#      but, this time, with the iterative solution (partition=~sex,
#      totals in pop03p):
descal03p<-e.calibrate(design=des,df.population=pop03p,
           calmodel=~marstat-1,partition=~sex,calfun="logit",
           bounds=bounds)

# Checking the result:
svystatTM(descal03p,~sex,~marstat)


# 4) Calibration (global solution) on the totals for the quantitative
#    variables x1, x2 and x3 in the subpopulations defined by the
#    regcod variable (totals in pop04):
descal04<-e.calibrate(design=des,df.population=pop04,
          calmodel=~(x1+x2+x3-1):regcod,calfun="logit",
          bounds=bounds,aggregate.stage=2)

# Checking the result:
svystatTM(descal04,~x1+x2+x3,~regcod)


# 4.1) Same problem with the iterative solution (partition=~regcod,
#      totals in pop04p):
descal04p<-e.calibrate(design=des,df.population=pop04p,
           calmodel=~x1+x2+x3-1,partition=~regcod,calfun="logit",
           bounds=bounds,aggregate.stage=2)

# Checking the result:
svystatTM(descal04p,~x1+x2+x3,~regcod)


# 5) Calibration (global solution) on the total for the quantitative
#    variable x1 and on the marginal distribution of the qualitative
#    variable age5c, in the subpopulations defined by crossing sex
#    and marstat (totals in pop05):
descal05<-e.calibrate(design=des,df.population=pop05,
          calmodel=~(age5c+x1-1):sex:marstat,calfun="logit",
          bounds=bounds)

# Checking the result:
svystatTM(descal05,~age5c+x1,~sex:marstat)


# 5.1) Same problem with the iterative solution (partition=~sex:marstat,
#      totals in pop05p):
descal05p<-e.calibrate(design=des,df.population=pop05p,
           calmodel=~age5c+x1-1,partition=~sex:marstat,
           calfun="logit",bounds=bounds)
```

```
# Checking the result:
svystatTM(descal05p,~age5c+x1,~sex:marstat)


# Notice that 3.1 and 5.1) 5.2) do not impose the aggregate.stage=2
# condition. This condition cannot, in fact, be fulfilled because
# in both cases the domains defined by partition "cut across"
# the des second stage clusters (households). To compare the results,
# the same choice was also made for 3) and 5).

###########################################################################
# Example with heteroskedatic assisting linear model: shows how to obtain #
# the ratio estimator of a total by calibration.                          #
###########################################################################

# Load sbs data:
data(sbs)

# Create the design object to be calibrated:
sbsdes<-e.svydesign(data=sbs,ids=~id,strata=~strata,weights=~weight,fpc=~fpc)

# Suppose you have to calibrate on the total amount of employees:
# Prepare the template:
pop<-pop.template(data=sbsdes,calmodel=~emp.num-1)
pop

# Fill it by using the sampling frame (sbs.frame)...
pop<-fill.template(sbs.frame,pop)
pop

# ... thus the total number of employees is 984394.
# Now calibrate assuming that error terms variances are proportional
# to emp.num:
sbscal<-e.calibrate(design=sbsdes,df.population=pop,sigma2=~emp.num)

# Now compute the calibration estimator of the total
# of value added (i.e. variable va.imp2)...
VA.tot.cal<-svystatTM(design=sbscal,y=~va.imp2)
VA.tot.cal

#... and observe that this is identical to the ratio estimator of the total...
VA.ratio<-svystatL(design=sbsdes, expression(984394*va.imp2/emp.num))
VA.ratio

# ...as it must be.

# Recall that, for the calibration problem above, one must expect, by virtue of
# simple theoretical arguments, that the g-weights are constant and equal to the
# ratio between the known total of emp.num (984394) and its HT estimate.
# This propertly is exactly stisfied by our numerical results, see below:
984394/coef(svystatTM(sbsdes, ~emp.num))
g.range(sbscal)

# ...as it must be.

########################################################################
```

```
# A second example of calibration with heteroskedatic assisting linear #
# model. Shows that calibrated weights associated to higher values of  #
# sigma2 tend to stay closer to their corresponding initial weights.   #
########################################################################

# Perform a calibration process which exploits as auxiliary
# information the total number of employees (emp.num)
# and enterprises (ent) inside the domains obtained by:
#  i) crossing nace2 and region;
# ii) crossing emp.cl, region and nace.macro;

# Build the population totals template:
pop<-pop.template(sbsdes,
     calmodel=~(emp.num+ent):(nace2+emp.cl:nace.macro)-1,
     partition=~region)

# Use the fill.template function to (i) automatically compute
# the totals from the universe (sbs.frame) and (ii) safely fill
# the template:
pop<-fill.template(universe=sbs.frame,template=pop)

# Now calibrate:
  # 1) First, without any heteroskedasticy effect
  sbscal1<-e.calibrate(sbsdes,pop,calfun="linear",bounds = c(0.01, 3),
           sigma2=NULL)

  # 2) Then, with heteroskedastic effect proportional to emp.num:
  sbscal2<-e.calibrate(sbsdes,pop,calfun="linear",bounds = c(0.01, 3),
           sigma2=~emp.num)

# Compute the g-weights for both the calibrated objects:
g1<-weights(sbscal1)/weights(sbsdes)
g2<-weights(sbscal2)/weights(sbsdes)

# Now visually compare the absolute deviations from 1 of the g-weights
# as a function of emp.num:
plot(log10(sbs$emp.num),abs(g1-1), col="blue", pch=19, cex=0.5)
points(log10(sbs$emp.num),abs(g2-1), col="red", pch=19, cex=0.5)

#...as emp.num grows red points clearly tend to stay closer to
# the horizontal axis than blue ones, as expected.
```

`e.svydesign` *Specification of a complex survey design*

Description

Binds survey data and sampling design metadata.

Usage

```
e.svydesign(data, ids, strata = NULL, weights,
            fpc = NULL, self.rep.str = NULL, check.data = TRUE)
```

Arguments

`data`	Data frame of survey data.
`ids`	Formula identifying clusters selected at subsequent sampling stages (PSUs, SSUs, ...).
`strata`	Formula identifying the stratification variable; `NULL` (the default) implies no stratification.
`weights`	Formula identifying the initial weights for the sampling units.
`fpc`	Formula identifying finite population corrections at subsequent sampling stages (see 'Details').
`self.rep.str`	Formula identifying self-representing strata (SR), if any; `NULL` (the default) means no SR strata (see 'Details').
`check.data`	Check out the correct nesting of `data` clusters? The default is `TRUE`.

Details

This function has the purpose of binding in an effective and persistent way the survey data to the metadata describing the adopted sampling design. Both kinds of information are stored in a complex object of class `analytic`, which extends the `survey.design2` class from the **survey** package. The sampling design metadata are then used to enable and guide processing and analyses provided by other functions in the **ReGenesees** package (such as `e.calibrate`, `svystatTM`, ...).

The `data`, `ids` and `weights` arguments are mandatory, while `strata`, `fpc`, `self.rep.str` and `check.data` arguments are optional. The `data` variables that are referenced by `ids`, `weights` and, if specified, by `strata`, `fpc`, `self.rep.str`, `check.data` must not contain any missing value (`NA`).

The `ids` argument specifies the cluster identifiers. It is possible to specify a multi-stage sampling design by simply using a formula which involves the identifiers of clusters selected at subsequent sampling stages. For example, `ids=~id.PSU + id.SSU` declares a two-stage sampling in which the first stage units are identified by the `id.PSU` variable and second stage ones by the `id.SSU` variable.

The `strata` argument identifies the stratification variable. The `data` variable referenced by `strata` (if specified) must be a `factor`. By default the sample is assumed to be non-stratified.

The `weights` argument identifies the initial (or direct) weights for the units included in the sample. The `data` variable referenced by `weights` must be `numeric`.

The `fpc` formula specifies the finite population corrections at subsequent sampling stages. If the survey has only one stage, then the `fpc`s can be given either as the total population size in each stratum or as the fraction of the total population that has been sampled. In either case the relevant population size is the sampling units (be they actual units or clusters). That is, sampling 100 units from a population stratum of size 500 can be specified as 500 or as 100/500=0.2.
For multistage sampling the population size for each sampling stage should also be specified in `fpc`. For instance, when `ids=~id.PSU + id.SSU` the `fpc` formula should look like `fpc=~fpc.PSU + fpc.SSU`, with variable `fpc.PSU` giving the sampling fractions in each stratum for the first stage units, while variable `fpc.SSU` gives the sampling fractions for the second stage units in each sampled PSU. If `fpc` is specified but for fewer stages than `ids`, sampling is assumed to be complete for subsequent stages. The function will check that `fpc`s values at each sampling stage do not vary within strata.

When dealing with a multistage, stratified sampling design that includes *self-representing (SR) strata* (i.e. strata containing PSUs selected with probability 1), the main contribution to the variance of the SR strata arises from the second stage units (*"variance PSUs"*).
When `options("RG.ultimate.cluster")` is `FALSE` (which is the default for **ReGenesees**), variance estimation for SR strata is correctly handled provided the survey `fpc`s have been properly

specified.
When, on the contrary, the *"Ultimate Cluster Approximation"* holds (i.e. `options("RG.ultimate.cluster")` has been set to `TRUE`) the SR strata give no contribution at all to the sampling variance.
A compromise solution (adopted by former existing survey softwares) is the one of retaining, for both SR and not-SR strata, only the leading contribution to the sampling variance. This means that only the SSUs are relevant for SR strata, whereby only the PSUs matter in not-SR strata. This compromise solution can be achieved by using the `self.rep.str` argument. If this argument is actually specified (as a formula referencing the `data` variable that identifies the SR strata), a warning is generated in order to remind the user that a compromise solution for variance estimation will be adopted on that design. Notice that, when choosing the `self.rep.str` option, the user must ensure that the variable referenced by `self.rep.str` is `logical` (with value `TRUE` for SR strata and `FALSE` otherwise) or `numeric` (with value `1` for SR strata and `0` otherwise) or `factor` (with levels `"1"` for SR strata and `"0"` otherwise).

The optional argument `check.data` allows to check out the correct nesting of `data` clusters (PSUs, SSUs, . . .). If `check.data=TRUE` the function checks that every unit selected at stage `k+1` is associated to one and only one unit selected at stage `k`. For a stratified design the function checks also the correct nesting of clusters within strata.

Value

An object of class `analytic`. The `print` method for that class gives a concise description of the sampling design.

Note

The `analytic` class is a specialisation of the `survey.design2` class from the **survey** package; this means that an object created by `e.svydesign` inherits from the `survey.design2` class and you can use on it every method defined on the latter class.

Author(s)

Diego Zardetto.

References

Sarndal, C.E., Swensson, B., Wretman, J. (1992) *"Model Assisted Survey Sampling"*, Springer Verlag.

Lumley, T. (2006) *"survey: analysis of complex survey samples"*, http://cran.at.r-project.org/web/packages/survey/index.html.

See Also

`svystatTM`, `svystatR`, `svystatQ`, `svystatL` for calculating estimates and standard errors, `e.calibrate` for calibrating weights, `ReGenesees.options` for setting/changing variance estimation options.

Examples

```
###############################################################
# The following examples illustrate how to create objects    #
# (of class 'analytic') defining different sampling designs. #
# Note: sometimes the same survey data will be used to       #
# define more than one design: this serves only the purpose  #
```

```
# of illustrating e.svydesign syntax.                        #
##############################################################

data(data.examples)
# Two-stage stratified cluster sampling design (notice that
# the design contains lonely PSUs):
des<-e.svydesign(data=example,ids=~towcod+famcod,strata=~stratum,
     weights=~weight)
des

# The same using collapsed strata (SUPERSTRATUM variable) to remove
# lonely PSUs:
des<-e.svydesign(data=example,ids=~towcod+famcod,strata=~SUPERSTRATUM,
     weights=~weight)
des

# Two stage cluster sampling (no stratification):
des<-e.svydesign(data=example,ids=~towcod+famcod,weights=~weight)
des

# Stratified unit sampling design:
des<-e.svydesign(data=example,ids=~key,strata=~SUPERSTRATUM,
     weights=~weight)
des


data(sbs)
# One-stage stratified unit sampling without replacement
# (notice the presence of the fpc argument):
des<-e.svydesign(data=sbs,ids=~id,strata=~strata,weights=~weight,
     fpc=~fpc)
des

# Same design as above but ignoring the finite population corrections:
des<-e.svydesign(data=sbs,ids=~id,strata=~strata,weights=~weight)
des


data(fpcdat)
# Two-stage stratified cluster sampling without replacement
# (notice that the fpcs are specified for both stages):
des<-e.svydesign(data=fpcdat,ids=~psu+ssu,strata=~stratum,weights=~w,
     fpc=~fpc1+fpc2)
des

# Same design as above but assuming complete sampling for the
# second stage units (notice fpcs have been passed only for the
# first stage):
des<-e.svydesign(data=fpcdat,ids=~psu+ssu,strata=~stratum,weights=~w,
     fpc=~fpc1)
des

# Again a two-stage stratified cluster sampling without replacement but
# specified in such a way as to retain, in the estimation phase, only
# the leading contribution to the sampling variance (i.e. the one arising
# from PSUs in SR strata and SSUs in not-SR strata). Notice that the
# self.rep.str argument is used:
```

```
des<-e.svydesign(data=fpcdat,ids=~psu+ssu,strata=~stratum,weights=~w,
     fpc=~fpc1+fpc2, self.rep.str=~sr)
des
```

---

extractors *Extractor functions for variance statistics*

Description

These functions extract standard errors (SE), variances (VAR), coefficients of variation (cv) and design effects (deff) from an object which has been returned by a survey statistic function (e.g. svystatTM, svystatR, svystatQ, svystatL, ...).

Usage

```
SE(object, ...)
VAR(object, ...)
cv(object, ...)
deff(object, ...)
```

Arguments

object	An object containing survey statistics.
...	Arguments for future expansion.

Details

With the exception of deff, all extractor functions can be used on any object returned by a survey statistic function: the correct answer will be obtained wathever the call that generated the object. For getting the design effect, object must have been built with option deff = TRUE.

Value

A vector storing the requested informations.

Note

Package **ReGenesees** provides extensions of methods coef and confint (originally from package **stats**) that can be used to extract estimates and confidence intervals respectively.

Author(s)

Diego Zardetto

See Also

Function coef to extract estimates and function confint to extract confidence intervals. Estimators of Totals and Means svystatTM, Ratios between Totals svystatR, Quantiles svystatQ and Complex Analytic Functions of Totals and/or Means svystatL.

Examples

```
# Creation of a design object:
data(sbs)
des<-e.svydesign(data=sbs,ids=~id,strata=~strata,weights=~weight,
     fpc=~fpc)

# Estimation of the average value added at the
# nation level (by default one gets the SE):
VA.avg <- svystatTM(des,~va.imp2,estimator="Mean")
VA.avg

# Extractions of some variance statistics from the
# object above:
## 1) SE
SE(VA.avg)
## 2) CV
cv(VA.avg)
## 3) VAR
VAR(VA.avg)

# Design effects have to be requested in advance,
# i.e. the following invocation produces an error:
## Not run:
deff(VA.avg)

## End(Not run)
# ...while the following works:
VA.avg <- svystatTM(des,~va.imp2,estimator="Mean",deff=TRUE)
deff(VA.avg)

# Further examples:
## extract the statistic:
coef(VA.avg)
## extract the confidence interval at 90%
## confidence level (the default would be 95%):
confint(VA.avg, conf.lev=0.9)
```

---

fill.template — *Fill the known totals template for a calibration task*

Description

Given a template prepared to store the totals of the auxiliary variables for a specific calibration task, computes the actual values of such totals from a sampling frame.

Usage

```
fill.template(universe, template, mem.frac = 10)
```

Arguments

`universe`	Data frame containing the complete list of the units belonging to the target population, along with the corrisponding values of the auxiliary variables (the sampling frame).

template	The template for the calibration task, an object of class pop.totals.
mem.frac	A numeric and non-negative value (the default is 10). It triggers a memory-efficient algorithm when universe is really huge (see 'Details' and 'Performance').

Details

Recall that a template object returned by function pop.template has a structure that complies with the standard required by e.calibrate, but is *empty*, in the sense that all the known totals it must be able to store are missing (NA). Whenever these totals are available to the user as such, that is in the form of already computed aggregated values (e.g. because they come from an external source, like a Population Census), the **ReGenesees** package cannot help the user to correctly fill the template. Stated more explicitely: the user himself has to bear the responsibility of putting the *right values* in the *right slots* of the prepared template data frame.

A lucky alternative arises when a *"sampling frame"* (that is a data frame containing the complete list of the units belonging to the target population, along with the corrisponding values of the auxiliary variables) is available. In such cases, indeed, the fill.template function is able to: (i) automatically compute the totals of the auxiliary variables from the universe data frame, (ii) safely arrange and format these values according to the template structure.

Notice that fill.template will perform a complete coherence check between universe and template. If this check fails, the program stops and prints an error message: the meaning of the message should help the user diagnose the cause of the problem.

Argument mem.frac (whose value must be numeric and non-negative) triggers a memory-efficient algorithm when universe is *really huge*. The *only* sound reason to ever change the value of this argument from its default (mem.frac=10) is that an invocation of fill.template caused a memory-failure (i.e. a messages beginning cannot allocate vector of size ...) on your machine. In such a case, *increasing* the value of mem.frac (e.g. mem.frac=20) will provide a better chance of succeeding (for more details, see 'Performance' section below).

Value

An object of class pop.totals storing the *actual* values of the population totals for the specified calibration task, ready to be safely passed to e.calibrate.

Performance

Real-world calibration tasks (e.g. in the field of Official Statistics) can simultaneously involve several hundreds of auxiliary variables and refer to target populations of several millions units. In such circumstances, the naive aggregation of the calibration model.matrix of universe may turn out to be too memory-demanding (at least in ordinary PC environments) and determine a memory-failure error.
The alternative implemented in fill.template is to: (i) split universe in chunks, (ii) compute partial sums of auxiliary variables chunk-by-chunk, (iii) update template by adding progressively such partial sums. This alternative is triggered by parameter mem.frac, which also implicitely controls the number of chunks. The function estimates the memory that would be used to store the *full* model.matrix of universe and compares it to the maximum memory allocable on the machine (as returned by memory.limit): if the resulting ratio is bigger than 1/mem.frac, the memory-efficient algorithm starts; the number of chunks in which universe will then be split is determined in such a way that the memory needed to store the model.matrix of *each* chunk does not exceed a fraction 1/mem.frac of the maximum allocable memory.
Whenever fill.template switchs to the memory-efficient "chunking" algorithm, a warning message will signal it and will specify as well the number of chunks that are being processed.

Author(s)

Diego Zardetto

See Also

`e.calibrate` for calibrating weights, `pop.template` for the definition of the class `pop.totals` and to build a "template" data frame for known population totals, and `%into%` for the compression operator for nested factors.

Examples

```
# Load sbs data:
data(sbs)

# Build a design object:
sbsdes<-e.svydesign(data=sbs,ids=~id,strata=~strata,weights=~weight,fpc=~fpc)

# Now suppose you have to perform a calibration process which
# exploits as auxiliary information the total number of employees (emp.num)
# and enterprises (ent) inside the domains obtained by:
#  i) crossing nace2 and region;
# ii) crossing emp.cl, region and nace.macro;

# Due to the fact that nace2 is nested into nace.macro,
# the calibration model can be efficiently factorised as follows:
## 1) Add to the design object and universe the new compressed
 #    factor variable involving nested factors, namely:
sbsdes<-des.addvars(sbsdes,nace2.in.nace.macro=nace2 %into% nace.macro)
sbs.frame$nace2.in.nace.macro<-sbs.frame$nace2 %into% sbs.frame$nace.macro

  # 2) Build the template exploiting the new variable:
pop<-pop.template(sbsdes,
     calmodel=~(emp.num+ent):(nace2.in.nace.macro + emp.cl)-1,
     partition=~nace.macro:region)

 # Note: given the dimension of the obtained template...
dim(pop)

 # ...the number of (independent) known totals to be stored is 792.

 # 3) Use the fill.template function to (i) automatically compute
 #    such 792 totals from the universe (sbs.frame) and (ii) safely fill
 #    the template:
pop<-fill.template(universe=sbs.frame,template=pop)

 # 4) Lastly calibrate, e.g. with the unbounded linear distance and
 #    heteroskedastic effects proportional to emp.num:
sbscal<-e.calibrate(sbsdes,pop,sigma2=~emp.num,bounds=c(-Inf,Inf))

# Note: a global calibration task would have led to identical calibrated
# weights, but in a more memory-hungry and time-consuming way, as you can
# verify:
  # 1) Build template:
pop.g<-pop.template(sbsdes,
       calmodel=~(emp.num+ent):(nace2:region + emp.cl:nace.macro:region)-1)
dim(pop.g)
```

```
  # 2) Fill template:
pop.g <- fill.template(sbs.frame,pop.g)

  # 3) Calibrate globally:
## Not run:
sbscal.g<-e.calibrate(sbsdes,pop.g,sigma2=~emp.num,bounds=c(-1E6,1E6))

  # 4) Compare calibrated weights (factorised vs. global solution):
range(weights(sbscal)/weights(sbscal.g))

  # ... they are equal.

## End(Not run)

# Just a single example of the memory-efficient algorithm triggered
# by mem.frac:
## Not run:
 # First artificially increase the size of the sampling frame (e.g.
 # up to 5 millions rows):
sbs.frame.HUGE<-sbs.frame[sample(1:nrow(sbs.frame),5000000,rep=TRUE),]
dim(sbs.frame.HUGE)

 # Build the template:
pop<-pop.template(sbsdes,
     calmodel=~(emp.num+ent):(nace2.in.nace.macro + emp.cl)-1,
     partition=~nace.macro:region)
dim(pop)

 # Fill the template by using the HUGE universe:
pop<-fill.template(universe=sbs.frame.HUGE,template=pop)

## End(Not run)
```

---

fpcdat *Artificial sample data for the ReGenesees package*

Description

A small dataset mimicking sample data selected with a 2-stage, stratified, cluster sampling without replacement. Allows to run R code contained in the 'Examples' section of the ReGenesees package help pages.

Usage

```
data(fpcdat)
```

Format

A data frame with 28 observations on the following 12 variables.

`psu` Identifier of the primary sampling units, `numeric`

`ssu` Identifier of the second stage sampling units, `numeric`

`stratum` Stratification Variable, a `factor` with 5 levels: `S.1`, `S.2`, `S.3`, `S.4`, `S.5` sr Strata type, integer with values 0 (NSR strata) and 1 (SR strata)

fpc1 First stage finite population corrections, given as population sizes (in terms of psu clusters) inside strata, numeric

fpc2 Second stage finite population corrections, given as population sizes (in terms of ssu clusters) inside the corresponding sampled psu, numeric

x A numeric variable

y A numeric variable

dom1 A variable defining unplanned estimation domains, factor with 3 levels: A, B, C

dom2 A variable defining unplanned estimation domains, factor with 6 levels: a, b, c, d, e, f

w Direct weights, numeric

z A numeric variable

pl.domain A variable defining planned estimation domains, factor with 3 levels: pd.1, pd.2, pd.3

Details

Though very small, the fpcdat dataset concentrates a lot of interesting features. The sampling design is a complex one, with both self-representing (SR) and not-self-representing (NSR) strata. Sampling fractions are deliberately not negligible, in order to stress the effects of finite population corrections on variance estimation. Moreover, being the observations so few, performing computations on the fpcdat dataset allows to check and understand easily all the effects of setting/changing the global variance estimation options of the **ReGenesees** package (see e.g. ReGenesees.options).

See Also

ReGenesees.options for setting/changing variance estimation options.

Examples

```
data(fpcdat)
str(fpcdat)
```

---

g.range *Range of g-weights*

Description

Computes the range of the ratios between calibrated weights and initial weights (*g-weights*).

Usage

```
g.range(cal.design)
```

Arguments

cal.design Object of class cal.analytic.

Details

This function computes the smallest interval which contains the ratios between calibrated weights and initial weights.

Value

A numeric vector of length 2.

Note

If `cal.design` has undergone `k` subsequent calibration steps (with `k > 2`), the function will return the range of the ratios between the output weights of calibration steps `k` and `k - 1`.

Author(s)

Diego Zardetto

See Also

`weights` to extract the weights from a design object, `e.calibrate` for calibrating weights and `bounds.hint` to obtain a hint for calibration problems where range restrictions are imposed on the *g-weights*.

Examples

```
# Creation of the object to be calibrated:
data(data.examples)
des<-e.svydesign(data=example,ids=~towcod+famcod,strata=~SUPERSTRATUM,
     weights=~weight)

# Calibration (iterative solution) on the marginal distribution
# of age in 5 classes (age5c) inside provinces (procod)
# (totals in pop06p) with bounds=c(0.5, 1.5):
descal06p<-e.calibrate(design=des,df.population=pop06p,
           calmodel=~age5c-1,partition=~procod,calfun="logit",
           bounds=c(0.5, 1.5),aggregate.stage=2)

# Now let's verify the actual range of the obtained g-weights:
g.range(descal06p)

# which indeed is covered by c(0.5, 1.5), as required.

# Now calibrate once again, this time on the joint distribution of sex
# and marstat (totals in pop03) with the global solution:
descal2<-e.calibrate(design=descal06p,df.population=pop03,
         calmodel=~marstat:sex-1,calfun="linear",bounds=bounds)

# Notice that the print method correctly takes the calibration chain
# into account:
descal2

# The range of the g-weights for the twice calibrated object is:
g.range(descal2)

#... which is equal to:
range(weights(descal2)/weights(descal06p))
```

```
#... and must not be confused with:
range(weights(descal2)/weights(des))
```

`pop.template` *Template data frame for known population totals*

Description

Constructs a *"template"* data frame to store known population totals for a calibration problem.

Usage

```
pop.template(data, calmodel, partition = FALSE)
```

Arguments

`data`	Data frame of survey data (or an object inheriting from class `analytic`).
`calmodel`	Formula defining the linear structure of the calibration model.
`partition`	Formula specifying the variables that define the "calibration domains" for the model. `FALSE` (the default) implies no calibration domains.

Details

This function creates an object of class `pop.totals`. A `pop.totals` object is made up by the union of a data frame (whose structure conforms to the standard required by `e.calibrate` for the known totals) and the metadata describing the calibration problem.

The mandatory argument `data` must identify the survey data frame on which the calibration problem is defined (or, as an alternative, an `analytic` object built upon that data frame).

The mandatory argument `calmodel` symbolically defines the calibration model you intend using: it identifies the auxiliary variables and the constraints for the calibration problem. The `data` variables referenced by `calmodel` must be `numeric` or `factor` and must not contain any missing value (`NA`).

The optional argument `partition` specifies the variables that define the calibration domains for the model. The default value (`FALSE`) means either that there are not calibration domains or that you want to solve the problem globally (even though it could be factorised). If a formula is passed through the `partition` argument the program checks that `calmodel` actually describes a "reduced model", that is it does not reference any of the partition variables; if this is not the case, the program stops and prints an error message. Notice that a formula like `by=~D1+D2` will be automatically translated into the factor-crossing formula `by=~D1:D2`. The `data` variables referenced by `partition` (if any) must be `factor` and must not contain any missing value (`NA`).

Value

An object of class `pop.totals`. The data frame it contains is a *"template"* in the sense that all the known totals it must be able to store are missing (`NA`). However, this data frame has a structure that complies with the standard required by `e.calibrate` (provided the latter is invoked with the same `calmodel` and `partition` values used to create the template).
The operation of filling the template's `NA`s with the actual values of the corresponding population totals has, obviously, to be done by the user. If the user has access to a *"sampling frame"* (that is a data frame containing the complete list of the units belonging to the target population along with the

corrisponding values of the auxiliary variables), then he can exploit the function `fill.template` to automatically fill the template.

The `pop.totals` class is a specialisation of the `data.frame` class; this means that an object built by `pop.template` inherits from the `data.frame` class and you can use on it every method defined on that class.

Author(s)

Diego Zardetto

See Also

`e.calibrate` for calibrating weights, `population.check` to check that the known totals data frame satisfies the standard required by `e.calibrate`, `fill.template` to automatically fill the template when a sampling frame is available.

Examples

```
# Creation of population totals template data frames for different
# calibration problems (if the calibration models can be factorised
# both a global and an iterative solution are given):

data(data.examples)

# 1) Calibration on the total number of units in the population:
pop.template(data=example,calmodel=~1)


# 2) Calibration on the total number of units in the population
#    and on the marginal distribution of marstat (notice that the
#    total for the first level "married" of the marstat factor
#    variable is missing because it can be deduced from
#    the remaining totals):
pop.template(data=example,calmodel=~marstat)


# 3) Calibration on the marginal distribution of marstat (you
#    must explicitly remove the intercept term in the
#    calibration model adding -1 to the calmodel formula):
pop.template(data=example,calmodel=~marstat-1)


# 4) Calibration (global solution) on the joint distribution of sex
#    and marstat:
pop.template(data=example,calmodel=~sex:marstat-1)

# 4.1) Calibration (iterative solution) on the joint distribution
#      of sex and marstat:
#      4.1.1) Using sex to define calibration domains:
pop.template(data=example,calmodel=~marstat-1,partition=~sex)

#      4.1.2) Using marstat to define calibration domains:
pop.template(data=example,calmodel=~sex-1,partition=~marstat)


# 5) Calibration (global solution) on the total for the quantitative
```

```
#    variable x1 and on the marginal distribution of the qualitative
#    variable age5c, in the subpopulations defined by crossing sex
#    and marstat:
pop.template(data=example,calmodel=~(age5c+x1-1):sex:marstat)

# 5.1) The same problem with iterative solutions:
#      5.1.1) Using sex to define calibration domains:
pop.template(data=example,calmodel=~(age5c+x1-1):marstat,partition=~sex)

#      5.1.2) Using marstat to define calibration domains:
pop.template(data=example,calmodel=~(age5c+x1-1):sex,partition=~marstat)

#      5.1.3) Using sex and marstat to define calibration domains:
pop.template(data=example,calmodel=~age5c+x1-1,partition=~sex:marstat)
```

population.check *Compliance test for known totals data frames*

Description

Checks whether a known population totals data frame conforms to the standard required by `e.calibrate` for a specific calibration problem.

Usage

```
population.check(df.population, data, calmodel, partition = FALSE)
```

Arguments

`df.population`	Data frame of known population totals.
`data`	Data frame of survey data (or an object inheriting from class `analytic`).
`calmodel`	Formula defining the linear structure of the calibration model.
`partition`	Formula specifying the variables that define the "calibration domains" for the model. `FALSE` (the default) implies no calibration domains.

Details

The behaviour of this function depends on the outcome of the test. If `df.population` is found to conform to the standard, the function first converts it into an object of class `pop.totals` and then invisibly returns it. Failing this, the function stops and prints an error message: the meaning of the message should help the user diagnose the cause of the problem.

The mandatory argument `df.population` identifies the known totals data frame for which compliance with the standard is to be checked.

The mandatory argument `data` identifies the survey data frame on which the calibration problem is defined (or, as an alternative, an `analytic` object built upon that data frame).

The mandatory argument `calmodel` symbolically defines the calibration model you intend using: it identifies the auxiliary variables and the constraints for the calibration problem. The `data` variables referenced by `calmodel` must be `numeric` or `factor` and must not contain any missing value (`NA`).

The optional argument `partition` specifies the variables that define the calibration domains for the model. The default value (`FALSE`) means either that there are not calibration domains or that you want to solve the problem globally (even though it could be factorised). If a formula is passed through the `partition` argument the program checks that `calmodel` actually describes a "reduced model", that is it does not reference any of the partition variables; if this is not the case, the program stops and prints an error message. Notice that a formula like `by=~D1+D2` will be automatically translated into the factor-crossing formula `by=~D1:D2`. The `data` variables referenced by `partition` (if any) must be `factor` and must not contain any missing value (`NA`).

Value

An invisibile object of class `pop.totals`. The `pop.totals` class is a specialisation of the `data.frame` class; this means that an object built by `pop.template` inherits from the `data.frame` class and you can use on it every method defined on that class.

Note

The `population.check` function can be used to convert a known totals data frame that conforms to the standard required by `e.calibrate` into an object of class `pop.totals`. The usefulness of this conversion lies in the fact that, once you have known totals with this "certified format", you can invoke `e.calibrate` without specifying the values for the `calmodel` and `partition` arguments (this means that the function is able to extract them directly from the attributes of the `pop.totals` object).

Author(s)

Diego Zardetto

See Also

`e.calibrate` for calibrating weights, `pop.template` for the definition of the class `pop.totals` and to build a "template" data frame for known population totals, `fill.template` to automatically fill the template when a sampling frame is available.

Examples

```
data(data.examples)

# Suppose you have to calibrate the example survey data frame
# on the totals of x1 by sex and you want the iterative solution.
# Start creating a design object:
des<-e.svydesign(data=example,ids=~towcod+famcod,strata=~SUPERSTRATUM,
     weights=~weight)

# Then build a template data frame for the known totals:
pop<-pop.template(data=example,calmodel=~x1-1,partition=~sex)
pop
class(pop)

# Now fill NAs with the actual values for the population
# totals (suppose 123 for sex="f" and 456 for sex="m"):
pop[,"x1"]<-c(123,456)
pop
class(pop)

# Finally check if pop complies with the kottcalibrate standard:
population.check(df.population=pop,data=example,calmodel=~x1-1,
```

```
                 partition=~sex)


# If, despite keeping the content unchanged, we altered the
# structure of the data frame (for example, by changing the
# order of its rows)...
pop.mod<-pop ; pop.mod[1,]<-pop[2,] ; pop.mod[2,]<-pop[1,]
pop
pop.mod

# ...we would obtain an error:
## Not run:
population.check(df.population=pop.mod,data=example,calmodel=~x1-1,
                 partition=~sex)

## End(Not run)


# Remember that, if the known totals have been converted
# into the pop.totals "format" by means of population.check,
# it is possible to invoke kottcalibrate without specifying
# calmodel and partition:

class(pop04p)
pop04p
descal04p<-e.calibrate(design=des,df.population=pop04p,
           calfun="logit",bounds=bounds,aggregate.stage=2)


# ...this option is not allowed if the known totals
# are not of class pop.totals even if they conform to the
# standard:

pop04p.mod=data.frame(pop04p)
class(pop04p.mod)
pop04p.mod
## Not run:
e.calibrate(design=des,df.population=pop04p.mod,calfun="logit",
             bounds=bounds,aggregate.stage=2)

## End(Not run)
```

`ReGenesees.options` *Variance estimation options for the ReGenesees package*

Description

This help page documents the options that control the behaviour of the ReGenesees package with respect to standard error estimation.

Details

The **ReGenesees** package provides three options for variance estimations which can be freely set and modified by the user:

- `RG.ultimate.cluster`
- `RG.lonely.psu`
- `RG.adjust.domain.lonely`

When `options("RG.ultimate.cluster")` is `TRUE`, the **ReGenesees** package adopts the so called *"Ultimate Cluster Approximation"* [Kalton 79]. Under this approximation, the overall sampling variance for a multistage sampling design is estimated by taking into account only the contribution arising from the estimated PSU totals (thus simply ignoring any available information about subsequent sampling stages). This approach is known to underestimate the true multistage variance, while - at the same time - overestimating its true first-stage component. Anyway, the underestimation becomes negligible if the PSUs' sampling fractions across strata are very small.

When `options("RG.ultimate.cluster")` is `FALSE`, each sampling stage contributes and variances get estimated by means of a recursive algorithm [Bellhouse, 85] inherited and adapted from package **survey** [Lumley 06]. Notice that the results obtained by choosing this option can differ from the one that would be obtained under the "Ultimate Cluster Approximation" *only if* first-stage finite population corrections are specified.

Lonely PSUs (i.e. PSUs which are alone inside a not self-representing stratum) are a concern from the viewpoint of variance estimation. The suggested **ReGenesees** facility to handle the lonely PSUs problem is the strata aggregation technique (see e.g. [Wolter 85] and [Rust, Kalton 87]) provided in function `collapse.strata`. As a possible alternative, you can get rid of lonely PSUs also by setting proper variance estimation options via `options("RG.lonely.psu")`. The default setting is `"fail"`, which raises an error if a lonely PSU is met. Option `"remove"` simply causes the software to ignore lonely PSUS for variance computation porposes. Option `"adjust"` means that deviations from the *population mean* will be used in variance estimation formulae, instead of deviations from the stratum mean (a conservative choice). Finally, option `"average"` causes the software to replace the variance contribution of the stratum by the average variance contribution across strata (this can be appropriate e.g. when one believes that lonely PSU strata occur at random due to uniform nonresponse among strata).

The variance formulae for domain estimation give well-defined, positive results when a stratum contains only one PSU with observations in the domain, but are not unbiased.
If `options("RG.adjust.domain.lonely")` is `TRUE` and
`options("RG.lonely.psu")` is `"average"` or `"adjust"` the same adjustment for lonely PSUs will be used within a domain. Note that this adjustment is not available for calibrated designs.

References

Kalton, G. (1979). *"Ultimate cluster sampling"*, Journal of the Royal Statistical Society, Series A, 142, pp. 210-222.

Bellhouse, D. R. (1985). *"Computing Methods for Variance Estimation in Complex Surveys"*. Journal of Official Statistics, Vol. 1, No. 3, pp. 323-329.

Lumley, T. (2006) *"survey: analysis of complex survey samples"*, http://cran.at.r-project.org/web/packages/survey/index.html.

Wolter, K.M. (1985) *"Introduction to Variance Estimation"*, Springer-Verlag, New York.

Rust, K., Kalton, G. (1987) *"Strategies for Collapsing Strata for Variance Estimation"*, Journal of Official Statistics, Vol. 3, No. 1, pp. 69-81.

See Also

`e.svydesign` and its `self.rep.str` argument for a "compromise solution" that can be adopted when the sampling design involves self-representing (SR) strata, `collapse.strata` for the suggested way of handling lonely PSUs, and `fpcdat` for useful data examples.

Examples

```
# Define a two-stage stratified cluster sampling without
# replacement:
data(fpcdat)
des<-e.svydesign(data=fpcdat,ids=~psu+ssu,strata=~stratum,weights=~w,
     fpc=~fpc1+fpc2)

# Now compare SE (or CV%) sizes under different settings:

## 1) Default setting, i.e. Ultimate Cluster Approximation is off
svystatTM(des,~x+y+z,vartype=c("se","cvpct"))

## 2) Turn on the Ultimate Cluster Approximation, thus missing
##    the variance contribution from the second stage
##    (hence SR strata give no contribution at all):
old.op <- options("RG.ultimate.cluster"=TRUE)
svystatTM(des,~x+y+z,vartype=c("se","cvpct"))
options(old.op)

## 3) The "compromise solution" (see ?e.svydesign) i.e. retaining
##    only the leading contribution to the sampling variance (namely
##    the one arising from PSUs in SR strata and SSUs in not-SR strata):
des2<-e.svydesign(data=fpcdat,ids=~psu+ssu,strata=~stratum,weights=~w,
      fpc=~fpc1+fpc2, self.rep.str=~sr)
svystatTM(des2,~x+y+z,vartype=c("se","cvpct"))

# Therefore, sampling variances come out in the expected
# hierarchy: 1) > 3) > 2).


# Under default settings lonely PSUs produce errors in standard
# errors estimation (notice we didn't pass the fpcs):
data(fpcdat)
des.lpsu<-e.svydesign(data=fpcdat,ids=~psu+ssu,strata=~stratum,
          weights=~w)
## Not run:
svystatTM(des.lpsu,~x+y+z,vartype=c("se","cvpct"))

## End(Not run)

# This can be circumvented in different ways, namely:
old.op <- options("RG.lonely.psu"="adjust")
svystatTM(des.lpsu,~x+y+z,vartype=c("se","cvpct"))
options(old.op)

# or:
options("RG.lonely.psu"="average")
svystatTM(des.lpsu,~x+y+z,vartype=c("se","cvpct"))
options(old.op)

# or otherwise by collapsing strata inside planned
# estimation domains:
des.clps<-collapse.strata(design=des.lpsu,block.vars=~pl.domain)
svystatTM(des.clps,~x+y+z,vartype=c("se","cvpct"))
```

sbs *Artificial Structural Business Statistics data for the ReGenesees package*

Description

The sbs data frame stores artificial sbs-like sampling data, while sbs.frame is the artificial sampling frame from which the sbs units have been drawn. They allow to run R code contained in the 'Examples' section of the ReGenesees package help pages.

Usage

```
data(sbs)
```

Format

The sbs data frame mimics data observed in a Structural Business Statistics survey, under a one-stage stratified unit sampling design. The sample is made up of 6909 units, for which the following 20 variables were observed:

id Identifier of the sampling units (enterprises), numeric

public Does the enterprise belong to the Public Sector? factor with levels 0 (No) and 1 (Yes)

emp.num Number of employees, numeric

emp.cl Number of employees classified into 5 categories, factor with levels [6,9] (9,19] (19,49] (49,99] (99,Inf] (notice that small enterprises with less than 6 employees fell outside the scope of the survey)

nace5 Economic Activity code with 5 digits, factor with 596 levels

nace2 Economic Activity code with 2 digits, factor with 57 levels

area Territorial Division, factor with 24 levels

cens Flag identifying statistical units to be censused (hence defining take-all strata), factor with levels 0 (No) and 1 (Yes)

region Macroregion, factor with levels North Center South

va.cl Class of Value Added, factor with 27 levels

va Value Added, numeric (contains NAs)

dom1 A planned estimation domain, factor with 261 levels (dom1 crosses nace2 and emp.cl)

nace.macro Economic Activity Macrosector, factor with levels Agriculture Industry Commerce Services

dom2 A planned estimation domain, factor with 12 levels (dom2 crosses nace.macro and region)

strata Stratification Variable, a factor with 664 levels (obtained by crossing variables region, nace2, emp.cl and cens)

va.imp1 Value Added Imputed1, numeric (NAs were replaced with average values computed inside imputation strata obtained by crossing region, nace.macro, emp.cl)

va.imp2 Value Added Imputed2, numeric (NAs were replaced with median values computed inside imputation strata obtained by crossing region, nace.macro, emp.cl)

y A numeric variable correlated with va

weight Direct weights, numeric

- `fpc` Finite Population Corrections (given as sampling fractions inside strata), `numeric`
- `ent` Convenience `numeric` variable identically equal to 1 (sometimes useful, e.g. to estimate the total number of enterprises)

The `sbs.frame` sampling frame (from which `sbs` units have been drawn) contains 17318 units.

Examples

```
data(sbs)
str(sbs)
str(sbs.frame)
```

`svystatL` *Estimation of Complex Estimators in subpopulations*

Description

Computes estimates, standard errors and confidence intervals for Complex Estimators in subpopulations. A Complex Estimator can be any analytic function of (Horvitz-Thompson or Calibration) estimators of Totals and Means.

Usage

```
svystatL(design, expr, by = NULL,
         vartype = c("se", "cv", "cvpct", "var"),
         conf.int = FALSE, conf.lev = 0.95, deff = FALSE,
         na.rm = FALSE)

## S3 method for class 'svystatL'
coef(object,...)
## S3 method for class 'svystatL'
SE(object,...)
## S3 method for class 'svystatL'
VAR(object,...)
## S3 method for class 'svystatL'
cv(object,...)
## S3 method for class 'svystatL'
deff(object,...)
## S3 method for class 'svystatL'
confint(object,...)
```

Arguments

`design`	Object of class `analytic` (or inheriting from it) containing survey data and sampling design metadata.
`expr`	R expression defining the Complex Estimator (see 'Details').
`by`	Formula specifying the variables that define the "estimation domains". If `NULL` (the default option) estimates refer to the whole population.
`vartype`	`character` vector specifying the desired variability estimators. It is possible to choose one or more of: standard error (`'se'`, the default), coefficient of variation (`'cv'`), percent coefficient of variation (`'cvpct'`), or variance (`'var'`).

conf.int	Compute confidence intervals for the estimates? The default is FALSE.
conf.lev	Probability specifying the desired confidence level: the default value is 0.95.
deff	Should the design effect be computed? The default is FALSE (see 'Details').
na.rm	Should missing values (if any) be removed from the variables of interest? (the default is FALSE).
object	An object of class svystatL.
...	Additional arguments to coef, ..., confint methods (if any).

Details

This function computes weighted estimates for Complex Estimators using suitable weights depending on the class of design: calibrated weights for class cal.analytic and direct weights otherwise. Standard errors are calculated using the Taylor linearization technique.

The mandatory argument expr, which identifies the Complex Estimator, must be an object of class expression. It can be specified just a single Complex Estimator at a time, i.e. length(expr) must be equal to 1. Any analytic function of estimators of Totals and Means is allowed.
Inside expr the estimator of the Total of a variable is simply represented by the *name* of the variable itself. To represent the estimator of the Mean of a variable y, the expression y/ones has to be used (ones being the convenience name of an artificial variable whose value is 1 for each sampling unit, so that its Total estimator actually estimates the population total). Variables referenced inside expr have obviously to belong to design and must be numeric.
At a minimal level, svystatL can be used to estimate Totals, Means and Ratios, thus reproducing the same results achieved by using the corresponding dedicated functions svystatTM and svystatR. For instance, calling svystatL(design, expression(y/x)) is equivalent to invoking svystatR(design, ~y, ~x), while using svystatL(design, expression(y/ones)) or svystatTM(design, ~y, estimator = "Mean") achieves an identical result.

The optional argument by specifies the variables that define the "estimation domains", that is the subpopulations for which the estimates are to be calculated. If by=NULL (the default option), the estimates produced by svystatL refer to the whole population. Estimation domains must be defined by a formula: for example the statement by=~B1:B2 selects as estimation domains the subpopulations determined by crossing the modalities of variables B1 and B2. Notice that a formula like by=~B1+B2 will be automatically translated into the factor-crossing formula by=~B1:B2: if you need to compute estimates for domains B1 and B2 *separately*, you have to call svystatL twice. The design variables referenced by by (if any) should be of type factor, otherwise they will be coerced.

The conf.int argument allows to request the confidence intervals for the estimates. By default conf.int=FALSE, that is the confidence intervals are not provided.

Whenever confidence intervals are requested (i.e. conf.int=TRUE), the desired confidence level can be specified by means of the conf.lev argument. The conf.lev value must represent a probability (0<=conf.lev<=1) and its default is chosen to be 0.95.

The optional argument deff allows to request the design effect [Kish 1995] for the estimates. By default deff=FALSE, that is the design effect is not provided. The design effect of an estimator is defined as the ratio between the variance of the estimator under the actual sampling design and the variance that would be obtained for an 'equivalent' estimator under a hypothetical simple random sampling without replacement of the same size. To obtain an estimate of the design effect comparing to simple random sampling *"with replacement"*, one must use deff="replace".
For nonlinear estimators, the design effect is estimated on the linearized version of the estimator (that is for the estimator of the total of the linearized variable, aka "Woodruff transform").
When dealing with domain estimation, the design effects referring to a given subpopulation are currently computed by taking the ratios between the actual variance estimates and those that would

have been obtained if a simple random sampling were carried out *within* that subpopulation. This is the same as the `srssubpop` option for Stata's function `estat`.

Value

An object inheriting from the `data.frame` class, whose detailed structure depends on input parameters' values.

Author(s)

Diego Zardetto

References

Sarndal, C.E., Swensson, B., Wretman, J. (1992) *"Model Assisted Survey Sampling"*, Springer Verlag.

Kish, L. (1995). *"Methods for design effects"*. Journal of Official Statistics, Vol. 11, pp. 55-77.

See Also

Estimators of Totals and Means `svystatTM`, Ratios between Totals `svystatR` and Quantiles `svystatQ`.

Examples

```
# Creation of a design object:
data(data.examples)
des<-e.svydesign(data=example,ids=~towcod+famcod,strata=~SUPERSTRATUM,
     weights=~weight)

# A first example: the ratio estimator of a Total,
# which relies on auxiliary information.
# Suppose you want to estimate the total of income
# and you know from an external source that the
# population size is, say, 1E6:
svystatL(des,expression(1E6*(income/ones)),vartype="cvpct")

# By comparing the latter result with the ordinary
# estimator of the mean one can see the variance
# reduction stemming from the correlation between
# numerator and denominator:
svystatTM(des,~income,vartype="cvpct")


# Creation of another design object:
data(sbs)
des<-e.svydesign(data=sbs,ids=~id,strata=~strata,weights=~weight,
     fpc=~fpc)

# A complex example: estimation of the Population Standard
# Deviation of a variable.
# Suppose you want to estimate the standard deviation of the
# population distribution of value added (va.imp2):
des<-des.addvars(des,va.imp2.sq=va.imp2^2)
svystatL(des,expression( sqrt( (ones/(ones-1))*
                               ( (va.imp2.sq/ones)-(va.imp2/ones)^2 )
                             )
```

```
                        ), conf.int=TRUE)

# The estimate above and the associated confidence interval (which
# involves the estimate of the sampling variance of the complex
# estimator) turn out to be very sound: indeed the TRUE value of the
# parameter is:
sd(sbs.frame$va.imp2)
```

---

svystatQ *Estimation of Quantiles in subpopulations*

Description

Calculates estimates, standard errors and confidence intervals for quantiles of numeric variables in subpopulations.

Usage

```
svystatQ(design, y, probs = c(0.25, 0.5, 0.75), by = NULL,
         vartype = c("se", "cv", "cvpct", "var"),
         conf.lev = 0.95, na.rm = FALSE,
         ties=c("discrete", "rounded"))

## S3 method for class 'svystatQ'
coef(object,...)
## S3 method for class 'svystatQ'
SE(object,...)
## S3 method for class 'svystatQ'
VAR(object,...)
## S3 method for class 'svystatQ'
cv(object,...)
## S3 method for class 'svystatQ'
confint(object,...)
```

Arguments

design	Object of class `analytic` (or inheriting from it) containing survey data and sampling design metadata.
y	Formula defining the interest variable.
probs	Vector of probability values to be used to calculate the quantiles estimates. The default value selects estimates of quartiles.
by	Formula specifying the variables that define the "estimation domains". If `NULL` (the default option) estimates refer to the whole population.
vartype	`character` vector specifying the desired variability estimators. It is possible to choose one or more of: standard error (`'se'`, the default), coefficient of variation (`'cv'`), percent coefficient of variation (`'cvpct'`), or variance (`'var'`).
conf.lev	Probability specifying the desired confidence level: the default value is `0.95`.
na.rm	Should missing values (if any) be removed from the variables of interest? (the default is `FALSE`).

`ties`	How should duplicated observed values be treated? Select `'discrete'` for a genuinely discrete interest variable and `'rounded'` for a continuous one.
`object`	An object of class `svystatQ`.
`...`	Additional arguments to `coef`, `...`, `confint` methods (if any).

Details

This function calculates weighted estimates for the quantiles of a quantitative variable using suitable weights depending on the class of `design`: calibrated weights for class `cal.analytic` and direct weights otherwise.
Standard errors are calculated using the so-called "Woodruff method" [Woodruff 52][Sarndal, Swensson, Wretman 92]: (i) first a confidence interval (at a given confidence level 1-a) is constructed for the relative frequency of units with values below the estimated quantile, (ii) then the inverse of the estimated cumulative relative frequency distribution (ECDF) is used to map this interval to a confidence interval for the quantile, (iii) lastly the desired standard error is estimated by dividing the length of the obtained confidence interval by the value 2*qnorm(1-a/2). Notice that the procedure above builds, in general, asymmetric confidence intervals around the estimated quantiles.

The mandatory argument `y` identifies the variable of interest, that is the variable for which estimates of quantiles have to be calculated. The `design` variable referenced by `y` must be `numeric`.

The optional argument `probs` specifies the probability values (`0.001<=probs[i]<=0.999`) corresponding to the quantiles one wants to estimate; the default option selects quartiles.

The optional argument `by` specifies the variables that define the "estimation domains", that is the subpopulations for which the estimates are to be calculated. If `by=NULL` (the default option), the estimates produced by `svystatTM` refer to the whole population. Estimation domains must be defined by a formula: for example the statement `by=~B1:B2` selects as estimation domains the subpopulations determined by crossing the modalities of variables `B1` and `B2`. Notice that a formula like `by=~B1+B2` will be automatically translated into the factor-crossing formula `by=~B1:B2`: if you need to compute estimates for domains `B1` and `B2` *separately*, you have to call `svystatQ` twice. The `design` variables referenced by `by` (if any) should be of type `factor`, otherwise they will be coerced.

The `conf.int` argument allows to request the confidence intervals for the estimates. By default `conf.int=FALSE`, that is the confidence intervals are not provided.

Whenever confidence intervals are requested (i.e. `conf.int=TRUE`), the desired confidence level can be specified by means of the `conf.lev` argument. The `conf.lev` value must represent a probability (`0<=conf.lev<=1`) and its default is chosen to be `0.95`.

Argument `ties` addresses the problem of how to treat duplicated observed values (if any) when computing the ECDF. Option `'discrete'` (the default) is appropriate when the variable of interest is genuinely discrete, while `'rounded'` is a better choice for a continuous variable, i.e. when duplicates stem from rounding. In the first case the ECDF will show a vertical step corresponding to a duplicated value, in the second a smoother shape will be achieved by linear interpolation.

Value

An object inheriting from the `data.frame` class, whose detailed structure depends on input parameters' values.

Author(s)

Diego Zardetto

References

Woodruff, R.S. (1952) *"Confidence Intervals for Medians and Other Position Measures"*, Journal of the American Statistical Association, Vol. 47, No. 260, pp. 635-646.

Sarndal, C.E., Swensson, B., Wretman, J. (1992) *"Model Assisted Survey Sampling"*, Springer Verlag.

See Also

Estimators of Totals and Means `svystatTM`, Ratios between Totals `svystatR` and Complex Analytic Functions of Totals and/or Means `svystatL`.

Examples

```
# Creation of a design object:
data(data.examples)
des<-e.svydesign(data=example,ids=~towcod+famcod,strata=~SUPERSTRATUM,
     weights=~weight)

# Estimate of the deciles of the income variable for
# the whole population:
svystatQ(des,~income,probs=seq(0.1,0.9,0.1),ties="rounded")


# Another design object:
data(sbs)
des<-e.svydesign(data=sbs,ids=~id,strata=~strata,weights=~weight,
     fpc=~fpc)

# Estimation of the median value added
# for economic activity macro-sectors:
svystatQ(des,~va.imp2,probs=0.5,by=~nace.macro,
         ties="rounded",vartype="cvpct")

# Estimation of the Interquartile Range (IQR) of the number
# of employees for economic activity macro-sectors:
apply(svystatQ(des,~emp.num,probs=c(0.25,0.75),by=~nace.macro)[,2:3],1,diff)
```

`svystatR` *Estimation of Ratios in subpopulations*

Description

Calculates estimates, standard errors and confidence intervals for ratios between totals in subpopulations.

Usage

```
svystatR(design, num, den, by = NULL, cross = FALSE,
         vartype = c("se", "cv", "cvpct", "var"),
         conf.int = FALSE, conf.lev = 0.95, deff = FALSE,
         na.rm = FALSE)
```

```
## S3 method for class 'svystatR'
coef(object,...)
## S3 method for class 'svystatR'
SE(object,...)
## S3 method for class 'svystatR'
VAR(object,...)
## S3 method for class 'svystatR'
cv(object,...)
## S3 method for class 'svystatR'
deff(object,...)
## S3 method for class 'svystatR'
confint(object,...)
```

Arguments

`design`	Object of class `analytic` (or inheriting from it) containing survey data and sampling design metadata.
`num`	Formula defining the numerator variables for the ratio estimators.
`den`	Formula defining the denominator variables for the ratio estimators.
`by`	Formula specifying the variables that define the "estimation domains". If `NULL` (the default option) estimates refer to the whole population.
`cross`	Should ratios be estimated for all the pairs of variables in `'num'` and `'den'`? The default is `FALSE`, meaning that ratios get estimated parallel-wise (see 'Details').
`vartype`	`character` vector specifying the desired variability estimators. It is possible to choose one or more of: standard error (`'se'`, the default), coefficient of variation (`'cv'`), percent coefficient of variation (`'cvpct'`), or variance (`'var'`).
`conf.int`	Compute confidence intervals for the estimates? The default is `FALSE`.
`conf.lev`	Probability specifying the desired confidence level: the default value is `0.95`.
`deff`	Should the design effect be computed? The default is `FALSE` (see 'Details').
`na.rm`	Should missing values (if any) be removed from the variables of interest? (the default is `FALSE`).
`object`	An object of class `svystatR`.
`...`	Additional arguments to `coef`, ..., `confint` methods (if any).

Details

This function computes weighted estimates for Ratios between Totals using suitable weights depending on the class of `design`: calibrated weights for class `cal.analytic` and direct weights otherwise. Standard errors are calculated using the Taylor linearization technique.

The mandatory argument `num` (`den`) identifies the variables whose totals appear as numerators (denominators) in the Ratio estimators: the corresponding formula must be of the type `num = ~num.1 + ... + num.k` (`den = ~den.1 + ... + den.l`). The `design` variables referenced by `num` (`den`) must be `numeric`.

If `cross=TRUE`, the function computes estimates for *all* the Ratios between pairs of variables coming from `num` and `den` (that is `k*l` estimates for the formulae above). If, on the contrary, `cross=FALSE` (the default), Ratios get estimated parallel-wise and R recycling rule is applied whenever `k!=l`: for the formulae above, this generates `r` Ratios, where `r=max(k,l)`.

The optional argument by specifies the variables that define the "estimation domains", that is the subpopulations for which the estimates are to be calculated. If by=NULL (the default option), the estimates produced by svystatR refer to the whole population. Estimation domains must be defined by a formula: for example the statement by=~B1:B2 selects as estimation domains the subpopulations determined by crossing the modalities of variables B1 and B2. Notice that a formula like by=~B1+B2 will be automatically translated into the factor-crossing formula by=~B1:B2: if you need to compute estimates for domains B1 and B2 *separately*, you have to call svystatR twice. The design variables referenced by by (if any) should be of type factor, otherwise they will be coerced.

The conf.int argument allows to request the confidence intervals for the estimates. By default conf.int=FALSE, that is the confidence intervals are not provided.

Whenever confidence intervals are requested (i.e. conf.int=TRUE), the desired confidence level can be specified by means of the conf.lev argument. The conf.lev value must represent a probability (0<=conf.lev<=1) and its default is chosen to be 0.95.

The optional argument deff allows to request the design effect [Kish 1995] for the estimates. By default deff=FALSE, that is the design effect is not provided. The design effect of an estimator is defined as the ratio between the variance of the estimator under the actual sampling design and the variance that would be obtained for an 'equivalent' estimator under a hypothetical simple random sampling without replacement of the same size. To obtain an estimate of the design effect comparing to simple random sampling *"with replacement"*, one must use deff="replace".
Being Ratios nonlinear estimators, the design effect is estimated on the linearized version of the estimator (that is: for the estimator of the total of the linearized variable, aka "Woodruff transform"). When dealing with domain estimation, the design effects referring to a given subpopulation are currently computed by taking the ratios between the actual variance estimates and those that would have been obtained if a simple random sampling were carried out *within* that subpopulation. This is the same as the srssubpop option for Stata's function estat.

Value

An object inheriting from the data.frame class, whose detailed structure depends on input parameters' values.

Warning

It can happen that, in some subpopulations, the estimate of the Total of some den variables turns out to be zero. In such cases svystatR returns NaN for the corresponding estimates.

Author(s)

Diego Zardetto

References

Sarndal, C.E., Swensson, B., Wretman, J. (1992) *"Model Assisted Survey Sampling"*, Springer Verlag.

Kish, L. (1995). *"Methods for design effects"*. Journal of Official Statistics, Vol. 11, pp. 55-77.

See Also

Estimators of Totals and Means svystatTM, Quantiles svystatQ and Complex Analytic Functions of Totals and/or Means svystatL.

Examples

```
# Creation of a design object:
data(sbs)
des<-e.svydesign(data=sbs,ids=~id,strata=~strata,weights=~weight,
     fpc=~fpc)

# Estimation of the average value added per employee
# at the nation level:
svystatR(des,~va.imp2,~emp.num)

# The same as above by economic activity macro-sector:
svystatR(des,~va.imp2,~emp.num,~nace.macro,vartype="cvpct")


# Another design object:
data(data.examples)
des<-e.svydesign(data=example,ids=~towcod+famcod,strata=~SUPERSTRATUM,
     weights=~weight)

# Estimation of the ratios y1/x1, y1/x2, y2/x1 and y2/x2 by region,
# notice the use of argument cross:
svystatR(des,~y1+y2,~x1+x2,by=~regcod,cross=TRUE)

# ... compare the latter with the default (i.e. cross=FALSE)
svystatR(des,~y1+y2,~x1+x2,by=~regcod)


# Estimation of the ratios z/x1, z/x2 e z/x3
# for the whole population (notice the recycling rule):
svystatR(des,~z,~x1+x2+x3,conf.int=TRUE)

# Estimators of means can be thought as
# estimators of ratios:
svystatTM(des,~income,estimator="Mean")
svystatR(des.addvars(des,ones=1),num=~income,den=~ones)
```

---

svystatTM — *Estimation of Totals and Means in subpopulations*

Description

Computes estimates, standard errors and confidence intervals for Totals and Means in subpopulations.

Usage

```
svystatTM(design, y, by = NULL, estimator = c("Total", "Mean"),
          vartype = c("se", "cv", "cvpct", "var"),
          conf.int = FALSE, conf.lev = 0.95, deff = FALSE,
          na.rm = FALSE)

## S3 method for class 'svystatTM'
coef(object,...)
```

```
## S3 method for class 'svystatTM'
SE(object,...)
## S3 method for class 'svystatTM'
VAR(object,...)
## S3 method for class 'svystatTM'
cv(object,...)
## S3 method for class 'svystatTM'
deff(object,...)
## S3 method for class 'svystatTM'
confint(object,...)
```

Arguments

`design`	Object of class `analytic` (or inheriting from it) containing survey data and sampling design metadata.
`y`	Formula defining the variables of interest.
`by`	Formula specifying the variables that define the "estimation domains". If `NULL` (the default option) estimates refer to the whole population.
`estimator`	`character` specifying the desired estimator: it may be `'Total'` (the default) or `'Mean'`.
`vartype`	`character` vector specifying the desired variability estimators. It is possible to choose one or more of: standard error (`'se'`, the default), coefficient of variation (`'cv'`), percent coefficient of variation (`'cvpct'`), or variance (`'var'`).
`conf.int`	Compute confidence intervals for the estimates? The default is `FALSE`.
`conf.lev`	Probability specifying the desired confidence level: the default value is `0.95`.
`deff`	Should the design effect be computed? The default is `FALSE` (see 'Details').
`na.rm`	Should missing values (if any) be removed from the variables of interest? (the default is `FALSE`).
`object`	An object of class `svystatTM`.
`...`	Additional arguments to `coef`, ..., `confint` methods (if any).

Details

This function computes weighted estimates for Totals and Means using suitable weights depending on the class of `design`: calibrated weights for class `cal.analytic` and direct weights otherwise. Standard errors for nonlinear estimators (e.g. calibration estimators) are calculated using the Taylor linearization technique.

The mandatory argument `y` identifies the variables of interest, that is the variables for which estimates are to be calculated. The corresponding formula should be of the type `y=~var1+...+varn`. The `design` variables referenced by `y` should be `numeric` or `factor` (variables of other types - e.g. `character` - will be coerced). It is admissible to specify for `y` "mixed" formulae that simultaneously contain quantitative (`numeric`) variables and qualitative (`factor`) variables.

To override the restriction to formulae of the type `y=~var1+...+varn`, the AsIs operator `I()` can be used (see 'Examples'). Though the latter opportunity could appear quite useful in some occasion, actually it should be almost always possible to find a work-around by using other functions of the **ReGenesees** package.

The optional argument `by` specifies the variables that define the "estimation domains", that is the subpopulations for which the estimates are to be calculated. If `by=NULL` (the default option), the estimates produced by `svystatTM` refer to the whole population. Estimation domains must be defined by a formula: for example the statement by=~B1:B2 selects as estimation domains the subpopulations determined by crossing the modalities of variables B1 and B2. Notice that a formula like by=~B1+B2 will be automatically translated into the factor-crossing formula by=~B1:B2: if you need to compute estimates for domains B1 and B2 *separately*, you have to call svystatTM twice. The design variables referenced by by (if any) should be of type factor, otherwise they will be coerced.

The optional argument estimator makes it possible to select the desired estimator. If estimator="Total" (the default option), svystatTM calculates, for a given variable of interest vark, the estimate of the total (when vark is numeric) or the estimate of the absolute frequency distribution (when vark is factor). Similarly, if estimator="Mean", the function calculates the estimate of the mean (when vark is numeric) or the the estimate of the relative frequency distribution (when vark is factor).

The conf.int argument allows to request the confidence intervals for the estimates. By default conf.int=FALSE, that is the confidence intervals are not provided.

Whenever confidence intervals are requested (i.e. conf.int=TRUE), the desired confidence level can be specified by means of the conf.lev argument. The conf.lev value must represent a probability (0<=conf.lev<=1) and its default is chosen to be 0.95.

The optional argument deff allows to request the design effect [Kish 1995] for the estimates. By default deff=FALSE, that is the design effect is not provided. The design effect of an estimator is defined as the ratio between the variance of the estimator under the actual sampling design and the variance that would be obtained for an 'equivalent' estimator under a hypothetical simple random sampling without replacement of the same size. To obtain an estimate of the design effect comparing to simple random sampling *"with replacement"*, one must use deff="replace".
Understanding what 'equivalent' estimator actually means is straightforward when dealing with Horvitz-Thompson estimators of Totals and Means. This is not the case when, on the contrary, the estimator to which the deff refers is a nonlinear estimator (e.g. for Calibration estimators of Totals and Means). In such cases, the standard approach is to use as 'equivalent' estimator the linearized version of the original estimator (that is: the estimator of the total of the linearized variable, aka "Woodruff transform").
When dealing with domain estimation, the design effects referring to a given subpopulation are currently computed by taking the ratios between the actual variance estimates and those that would have been obtained if a simple random sampling were carried out *within* that subpopulation. This is the same as the srssubpop option for Stata's function estat.

Value

An object inheriting from the data.frame class, whose detailed structure depends on input parameters' values.

Author(s)

Diego Zardetto

References

Sarndal, C.E., Swensson, B., Wretman, J. (1992) *"Model Assisted Survey Sampling"*, Springer Verlag.

Kish, L. (1995). *"Methods for design effects"*. Journal of Official Statistics, Vol. 11, pp. 55-77.

See Also

Estimators of Ratios between Totals svystatR, Quantiles svystatQ, and Complex Analytic Functions of Totals and/or Means svystatL.

Examples

```
data(data.examples)

# Creation of a design object:
des<-e.svydesign(data=example,ids=~towcod+famcod,strata=~SUPERSTRATUM,
     weights=~weight)


# Estimation of the total of 3 quantitative variables for the whole
# population:
svystatTM(des,~y1+y2+y3)


# Estimation of the total of the same 3 variables by region, with SE
# and CV%:
svystatTM(des,~y1+y2+y3,~regcod,vartype=c("se","cvpct"))


# Estimation of the mean of the same 3 variables by marstat and sex:
svystatTM(des,~y1+y2+y3,~marstat:sex,estimator="Mean")


# Estimation of the absolute frequency distribution of the qualitative
# variable age5c for the whole population, with the design effect:
svystatTM(des,~age5c,deff=TRUE)


# Estimation of the relative frequency distribution of the qualitative
# variable marstat by sex:
svystatTM(des,~marstat,~sex,estimator="Mean")

# Estimation of the relative frequency of the joint distribution of sex
# and marstat:
  # First Solution (using the AsIs operator I()):
svystatTM(des,~I(sex:marstat),estimator="Mean")
  # Second Solution (adding a new variable to des):
des2 <- des.addvars(des, sex.marstat=sex:marstat)
svystatTM(des2,~sex.marstat,estimator="Mean")

# Estimation of the mean income inside provinces, with confidence intervals
# at a confidence level of 0.9:
svystatTM(des,~income,~procod,estimator="Mean",conf.int=TRUE,conf.lev=0.9)


# Quantitative and qualitative variables together: estimation of the
# total of income and of the absolute frequency distribution of sex,
# by marstat:
svystatTM(des,~income+sex,~marstat)


# Under default settings lonely PSUs produce errors in standard
```

```
# errors estimation (notice we didn't pass the fpcs):
data(fpcdat)
des.lpsu<-e.svydesign(data=fpcdat,ids=~psu+ssu,strata=~stratum,
          weights=~w)
## Not run:
svystatTM(des.lpsu,~x+y+z,vartype=c("se","cvpct"))

## End(Not run)

# This can be circumvented in different ways, namely:
old.op <- options("RG.lonely.psu"="adjust")
svystatTM(des.lpsu,~x+y+z,vartype=c("se","cvpct"))

# or otherwise:
options("RG.lonely.psu"="average")
svystatTM(des.lpsu,~x+y+z,vartype=c("se","cvpct"))
options(old.op)
```

`weights` *Retrieve sampling units weights*

Description

Extracts the *current* weights of the units belonging to a survey design object.

Usage

```
weights(object, ...)
```

Arguments

`object`	Object of class `analytic` (or inheriting from it) containing survey data and sampling design metadata.
`...`	Arguments for future expansion.

Details

The *current* weights of `object` are, by definition, those weights that would be used for estimation purposes on that object (e.g. by functions `svystatTM`, `svystatR`, `svystatQ`, `svystatL`, ...). The nature of such weights depends on the class of `object`: calibrated weights for class `cal.analytic` and direct weights otherwise.

Value

A vector of weights, whose components are positionally tied to the sampling units belonging to `object`.

Note

If `object` has undergone multiple, subsequent calibration steps, the function will return the output weights generated by the *last* calibration step.

Author(s)

Diego Zardetto

See Also

Function `g.range` to asses the range of the g-weights of a calibrated design object.

Examples

```
# Creation of the object to be calibrated:
data(data.examples)
exdes<-e.svydesign(data=example,ids=~towcod+famcod,strata=~SUPERSTRATUM,
     weights=~weight)

# Retrieve the weights and summarize their distribution:
summary(weights(exdes))

# Now calibrate (global solution) on the joint distribution of sex
# and marstat (totals in pop03):
excal.1st<-e.calibrate(design=exdes,df.population=pop03,
           calmodel=~marstat:sex-1,calfun="linear",bounds=bounds)

# Retrieve the current weights (i.e. the calibrated ones) and
# summarize their distribution:
summary(weights(excal.1st))

# Now calibrate once again, this time on the marginal distribution
# of age in 5 classes (age5c) inside provinces (procod) (totals in pop06p)
# with the iterative solution, the logit distance and bounds=c(0.5, 1.5):
excal.2nd<-e.calibrate(design=excal.1st,df.population=pop06p,
           calmodel=~age5c-1,partition=~procod,calfun="logit",
           bounds=c(0.5, 1.5))

# Notice that the print method correctly takes the calibration chain
# into account:
excal.2nd

# Now retrieve the current weights (i.e. the ones generated by the second
# calibration step) and summarize their distribution:
summary(weights(excal.2nd))
```

`write.svystat` *Export Survey Statistics*

Description

Prints Survey Statistics to a file or connection.

Usage

```
write.svystat(x, ...)
```

Arguments

x	An object containing survey statistics.
...	Arguments to write.table

Details

This function is just a convenience wrapper to write.table, designed to export objects which have been returned by survey statistics functions (e.g. svystatTM, svystatR, svystatQ, svystatL).

Author(s)

Diego Zardetto

See Also

write.table and the 'R Data Import/Export' manual.

Examples

```
# Creation of a design object:
data(sbs)
des<-e.svydesign(data=sbs,ids=~id,strata=~strata,weights=~weight,
     fpc=~fpc)

# Estimation of the average value added per employee
# for economic activity region and macro-sectors,
# with SE, CV% and standard confidce intervals:
stat <- svystatR(des,~va.imp2,~emp.num,by=~region:nace.macro,
        vartype=c("se","cvpct"),conf.int=TRUE)
stat

# In order to export the summary statistics above
# into a CSV file for input to Excel one can use:
## Not run:
write.svystat(stat,file="stat.csv",sep=";")

## End(Not run)

# ...and to read this file back into R one needs
## Not run:
stat.back <- read.table("stat.csv",header=TRUE,sep=";",
             check.names=FALSE)
stat.back

## End(Not run)

# Notice, however, that the latter object has
# lost a lot of meta-data as compared to the
# original one, so that e.g.:
## Not run:
confint(stat.back)

## End(Not run)

# ...while, on the contrary:
confint(stat)
```

`%into%` *Compress nested factors*

Description

The special binary operator `%into%` transforms nested factors in such a way as to reduce the dimension and/or the sparseness of the model matrix of a calibration problem.

Usage

```
inner %into% outer
"%into%"(inner, outer)
```

Arguments

`inner`	Factor with levels nested into `outer` (see 'Details').
`outer`	Factor whose levels are an aggregation of those in `inner` (see 'Details').

Details

Arguments `inner` and `outer` must be both `factors` and must have the same `length`. Moreover, `inner` has to be *strictly nested* into `outer`. Nesting is defined by treating elements in `inner` and `outer` as if they were positionally tied (i.e. as if they belonged to columns of a given dataframe). The definition is as follows:

`inner` and `outer` are strictly nested if, and only if, 1) every set of equal elements in `inner` correspond to a set of equal elements in `outer`, and 2) `inner` has *more* non-empty levels than `outer`.

If `inner` and `outer` do not fulfill the conditions above, evaluating `inner %into% outer` gives an error.

Suppose `inner` is actually nested into `outer` and define `inner.in.outer <- inner %into% outer`. The output factor `inner.in.outer` is built by recoding `inner` levels in such a way that each of them is mapped into the integer which represents its order inside the corresponding level of `outer` (see 'Examples'). As a consequence, the levels of `inner.in.outer` will be `1:n.max`, being `n.max` the *maximum* number of levels of `inner` tied to a level of `outer`. Since this number is generally considerably smaller than the number of levels of `inner`, `inner.in.outer` can be seen as a *compressed* representation of `inner`. Obviously, compression comes at a price: indeed `inner.in.outer` can now be used to identify a level of `inner` only *inside* a given level of `outer` (see 'Examples').

The usefulness of the `%into%` operator emerges in the calibration context. As we already documented in `e.calibrate`, factorizing a calibration problem (i.e. exploiting the `partition` argument of `e.calibrate`) determines a significant reduction in computation complexity, especially for big surveys. Now, it is sometimes the case that a calibration model is actually factorizable, even if this property is not self-apparent, due to factor nesting. In such cases, anyway, trying naively to factorize the `outer` variable(s) typically leads to very big and sparse model matrices (as well as population totals dataframes), with the net result of vashing-out the expected efficiency gain. A better alternative is to exploit the `%into%` operator in order to *compress* the `inner` variable in such a way that the `outer` variable can be actually factorized *without* giving rise to huge and sparse matrices. Section 'Examples' reports some practical illustration of the above line of reasoning.

Value

A factor with levels `1:n.max`, being `n.max` the *maximum* number of levels of `inner` tied to a level of `outer`.

Author(s)

Diego Zardetto

See Also

Further examples can be found in the `fill.template` help page.

Examples

```
#################################################
## General properties of the %into% operator   #
#################################################
 # First build a small dataframe with 2 nested factors representing
 # regions and provinces:
dd <- data.frame(
                 reg  = factor( rep(LETTERS[1:3], c(6, 3, 1)) ),
                 prov = factor( rep(letters[1:6], c(3, 2, 1, 2, 1, 1)) )
                )
dd

 # Since prov is strictly nested into reg we can compute:
prov.in.reg <- dd$prov %into% dd$reg
prov.in.reg

 # Note that prov.in.reg has 3 levels because, as can be seen from dd,
 # the maximum number of provinces inside regions is 3. Thus prov.in.reg
 # is actually a compressed version of dd$prov (whose levels were 6)
 # but, obviously, it can now be used to identify a province only inside
 # a given region. This means that the the two factors below are identical (up
 # to levels' labels):
dd$prov
interaction(prov.in.reg,dd$reg,drop=TRUE)

 # Note that all the statements below generate errors:
## Not run:
dd$reg  %into% dd$prov
dd$reg  %into% dd$reg
dd$prov %into% dd$prov

## End(Not run)

#####################################################################
## A more useful (and complex) example from the calibration context #
#####################################################################
 # First define a design object:
data(data.examples)
exdes <- e.svydesign(data=example,ids=~towcod+famcod,strata=~SUPERSTRATUM,
weights=~weight)

 # Now suppose you have to perform a calibration process which
 # exploits the following known population totals:
```

```
# 1) Joint distribution of sex and age10c (age in 10 classes)
#    at the region level;
# 2) Joint distribution of sex and age5c (age in 5 classes)
#    at the province level;
#
# The auxiliary variables corresponding to the population totals above
# can be symbolically represented by a calibration model like the following:
# ~(procod:age5c + regcod:age10c - 1):sex
#
# At first sight it seems that only the sex variable can be factorized
# in the model above. However if one observe that regions are an aggregation
# of provinces, one realizes that also the regcod variable can be factorized.
# Similarly, since categories of age5c are an aggregation of categories of
# age10c, age10c can be factorized too. In both cases, using the %into%
# operator will save computation time and memory usage.
# Let us see it in practice:
#
## 1) Global calibration (i.e. calmodel=~(procod:age5c + regcod:age10c - 1):sex,
 # no partition variable, known totals stored in pop07):
t<-system.time(
               cal07<-e.calibrate(design=exdes,df.population=pop07,
                      calmodel=~(procod:age5c + regcod:age10c - 1):sex,
                      calfun="logit",bounds=c(0.2,1.8),aggregate.stage=2)
               )

## 2) Partitioned calibration on the self evident variable sex only
 # (i.e. calmodel=~procod:age5c + regcod:age10c - 1, partition=~sex,
 # known totals stored in pop07p):
tp<-system.time(
                cal07p<-e.calibrate(design=exdes,df.population=pop07p,
                        calmodel=~procod:age5c + regcod:age10c - 1,partition=~sex,
                        calfun="logit",bounds=c(0.2,1.8),aggregate.stage=2)
                )

## 3) Full partitioned calibration on variables sex, regcod and age5c
 # by exploiting %into%.
 # First add to the design object the new compressed factor variables
 # involving nested factors, namely provinces inside regions...
exdes<-des.addvars(exdes,procod.in.regcod=procod %into% regcod)

 # ...and age10c inside age5c:
exdes<-des.addvars(exdes,age10c.in.age5c=age10c %into% age5c)

 # Now calibrate exploiting the new variables
 # (i.e. calmodel=~procod.in.regcod + age10c.in.age5c - 1,
 # partition=~sex:regcod:age5c, known totals stored inside cal07pp)
tpp<-system.time(
                 cal07pp<-e.calibrate(design=exdes,df.population=pop07pp,
                          calmodel=~procod.in.regcod + age10c.in.age5c - 1,
                          partition=~sex:regcod:age5c,
                          calfun="logit",bounds=c(0.2,1.8),aggregate.stage=2)
                )

# Now compare execution times:
t
tp
tpp
```

```
# thus, tpp < tp < t, as expected.
```

Index