Package 'ReGenesees'

December 29, 2021

Type Package

Title R Evolved Generalized 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, conditional and joint), Ratios, Shares and Ratios of Shares, Multiple Regression Coefficients and Quantiles. Automated Linearization of Complex Analytic Estimators. Design Covariance and Correlation. Estimates, standard errors, confidence intervals and design effects for user-defined analytic estimators. Estimates and sampling errors for subpopulations. Consistent trimming of calibration weights. Calibration on complex population parameters, e.g. multiple regression coefficients. Generalized Variance Functions (GVF) method for predicting variance estimates.

Version 2.2

Author Diego Zardetto [aut, cre]

Maintainer Diego Zardetto <zardetto@istat.it>

License EUPL

URL https://diegozardetto.github.io/ReGenesees/, https://github.com/DiegoZardetto/ReGenesees/

BugReports https://github.com/DiegoZardetto/ReGenesees/issues/

Imports stats, MASS

Depends R (>= 2.14.0)

ByteCompile TRUE

R topics documented:

ReGenesees-package *ReGenesees: a Package for Design-Based and Model-Assisted Analysis of Complex Sample Surveys*

Description

ReGenesees is an R package for design-based and model-assisted analysis of complex sample surveys. It handles multistage, stratified, clustered, unequally weighted survey designs. Sampling variance estimation for nonlinear (smooth) estimators is obtained by Taylor-series linearization. Sampling variance estimation for multistage designs can be obtained both under the Ultimate Cluster approximation or by means of an actual multistage computation. Estimates, standard errors, confidence intervals and design effects are provided for: Totals, Means, absolute and relative Frequency Distributions (marginal, conditional and joint), Ratios, Shares and Ratios of Shares, Multiple Regression Coefficients and Quantiles (variance via the Woodruff method). **ReGenesees** also handles Complex Estimators, i.e. any user-defined estimator that can be expressed as an analytic function of Horvitz-Thompson or Calibration estimators of Totals or Means, by automatically linearizing them. The Design Covariance and Correlation between Complex Estimators is also provided. All analyses above can be carried out for arbitrary subpopulations. In addition, **ReGenesees** can trim calibration weights while preserving all the calibration constraints, and perform 'special purpose calibration' tasks, i.e. can calibrate on complex population parameters like Multiple Regression Coefficients. Lastly, **ReGenesees** offers a Generalized Variance Functions (GVF) infrastructure, i.e. facilities for defining, fitting, testing and plotting GVF models, and to exploit them to predict variance estimates.

The **ReGenesees** *package* is the fundamental building block of a full-fledged R-based software *system*: the **ReGenesees System**. The latter has a clear-cut two-layer architecture. The application layer of the system is embedded into package **ReGenesees**. A second R package, called **ReGenesees.GUI**, implements the presentation layer of the system, namely a user-friendly Tcl/Tk GUI.

A Quick Reading Guide to the Reference Manual

This reference manual reports a documentation entry for each (user visible) function of package **ReGenesees**. As you may have noticed by reading section 'R topics documented' (page 1 of the pdf manual), these documentation entries are automatically sorted according to the alphabetic ordering of the names of the functions. Such an ordering doesn't provide any clue about where should a user start reading, nor on the best way to proceed further.

In section 'Table of Contents', I tried to cluster the most important topics documented in the reference manual into few broad groups, based on both the statistical goals and on the software design of the underlying functions.

Moreover, I provided a *relevance code* for each documented topic/function. The meaning of such codes, along with the corresponding *reading suggestions*, are reported in the following table:

Relevance Codes Legend

```
CODE    RELEVANCE           READING SUGGESTION
 ***    Very Important......Read these topics as soon as possible. A clear
                            understanding of these functions is mandatory
                            in order to start using profitably the package.

  **    Important...........Read these topics once you have been experiencing
                            for a while with (at least some of) the 'Very
```

```
                        Important' functions.

 *    Useful.............These functions are ancillary (albeit in
                        different ways) to the 'Very Important' and
                        'Important' ones (and their usage is generally
                        simpler).

 .    Advanced...........These topics are very relevant but, unfortunately
                        quite difficult. As they involve technical
                        details, you should postpone their reading until
                        you become familiar with the package.
```

Important Notice
It goes without saying that the '**Examples**' sections at the end of each documented topic **represent a crucial part of this reference manual**.

TABLE OF CONTENTS

```
  .  prep.calBeta.........Prepare a Survey Design to Calibration on
                          Multiple Regression Coefficients
  .  pop.calBeta.........Prepare Control Totals for Calibration on
                          Multiple Regression Coefficients
  .  pop.fuse............Fuse Control Totals Data Frames for Special
                          Purpose and Ordinary Calibration Tasks
```

Estimates and Sampling Errors:

```
***  svystatTM............Estimation of Totals and Means in
                          Subpopulations
***  svystatR.............Estimation of Ratios in Subpopulations
***  svystatS.............Estimation of Shares in Subpopulations
***  svystatSR............Estimation of Share Ratios in Subpopulations
***  svystatB.............Estimation of Population Regression Coefficients in
                          Subpopulations
***  svystatQ.............Estimation of Quantiles in Subpopulations
***  svystatL.............Estimation of Complex Estimators in
                          Subpopulations
 **  aux.estimates........Quick Estimates of Auxiliary Variables Totals
 **  CoV, Corr............Design Covariance and Correlation of Complex
                          Estimators in Subpopulations
  *  write.svystat........Export Survey Statistics
  *  extractors...........Extractor Functions for Variability Statistics
  .  ReGenesees.options...Variance Estimation Options for the ReGenesees
                          Package
```

Generalized Variance Functions Method:

```
***  GVF.db...............Archive of Registered GVF Models
***  gvf.input............Prepare Input Data to Fit GVF Models
***  svystat..............Compute Many Estimates and Errors in Just a
                          Single Shot
***  fit.gvf..............Fit GVF Models
 **  plot.gvf.fit.........Diagnostic Plots for Fitted GVF Models
 **  drop.gvf.points......Drop Outliers and Refit a GVF Model
  *  getR2, AIC, BIC......Quality Measures on Fitted GVF Models
  *  getBest..............Identify the Best Fit GVF Model
***  predictCV............Predict CV Values via Fitted GVF Models
  *  gvf.misc.............Miscellanea: Methods for Fitted GVF Models
  *  estimator.kind.......Which Estimator Did Generate these
                          Survey Statistics?
```

Diagnostics and Utilities:

```
  *  UWE..................Unequal Weighting Effect
  *  Zapsmall.............Zapsmall Data Frame Columns and Numeric Vectors
```

Data Sets:

```
 **  data.examples........Artificial Household Survey Data
 **  fpcdat...............A Small But Not Trivial Artificial Sample
                          Data Set
 **  sbs..................Artificial Structural Business Statistics Data
 **  AF.gvf...............Example Data for GVF Model Fitting
```

The ordering of the above 'Table of Contents' reflects only loosely the procedural sequence in which functions could be used. For instance, while you cannot apply function `e.calibrate` unless you have previously built a design object by using `e.svydesign`, you can exploit, e.g., function `collapse.strata` also after calibration. As a further example, all functions in group 'Estimates and Sampling Errors' can be used on objects created by `e.svydesign` (yielding estimates and sampling errors for functions of Horvitz-Thompson estimators), as well as on objects created by `e.calibrate` (yielding estimates and sampling errors for functions of Calibration estimators).

`AF.gvf` *Example Data for GVF Model Fitting*

Description

File `AF.gvf` contains a set of summary statistics that can be used to illustrate **ReGenesees** facilities for fitting Generalized Variance Functions models. These summary statistics have kind 'Absolute Frequency' (see function `estimator.kind`), i.e. involve estimates and errors of counts.

Usage

```
data(AF.gvf)
```

Format

Each row of the `ee.AF` data frame represents an estimated absolute frequency along with its estimated sampling error (expressed in terms of standard error, coefficient of variation and variance). The data frame has `349` rows, and the following `5` columns:

`name` The name of the original estimate, `factor` with `349` levels.

`Y` The value of the original estimate (an absolute frequency), `numeric`.

`SE` The standard error of the original estimate, `numeric`.

`CV` The coefficient of variation of the original estimate, `numeric`.

`VAR` The variance of the original estimate, `numeric`.

Details

Object `AF` is a list storing estimates and errors of counts (namely, summary statistics of kind 'Absolute Frequency') computed on survey design object `exdes`. The names of the slots of list `AF` indicate the nature of the corresponding estimates, e.g. element `AF[["sex.marstat"]]` stores estimates and errors of the joint absolute frequency distribution of variables `sex` and `marstat` (see 'Examples').

Object `ee.AF` is the `gvf.input` object built upon all such summary statistics, via function `gvf.input` (see 'Examples').

See Also

`estimator.kind` to assess what kind of estimates are stored inside a survey statistic object, `GVF.db` to manage **ReGenesees** archive of registered GVF models, `gvf.input` and `svystat` to prepare the input for GVF model fitting, `fit.gvf` to fit GVF models, `plot.gvf.fit` to get diagnostic plots for fitted GVF models, `drop.gvf.points` to drop alleged outliers from a fitted GVF model and simultaneously refit it, and `predictCV` to predict CV values via fitted GVF models.

Examples

```
data(AF.gvf)

# Inspect object AF
class(AF)
length(AF)
names(AF)
AF$sex.marstat
class(AF$sex.marstat)

# Inspect gvf.input object ee.AF
head(ee.AF)
str(ee.AF)
plot(ee.AF)

# The design object used to compute ee.AF is the following:
exdes

# How has object ee.AF been built?
foo <- gvf.input(exdes, stats = AF)
identical(ee.AF, foo)
```

`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 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 (the default is `TRUE`).

Details

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 factorized). 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 minimum-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 minimum-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 interval 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.

However, when both the benchmark population totals and the corresponding Horvitz-Thompson estimates are *all non-negative*, it is easy to find 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. 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.

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, `g.range` to compute the range of the obtained g-weights, and `check.cal` to check if calibration constraints have been fulfilled.

Examples

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

# Calibration (partitioned 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 inspecting 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
# directly looking at ecal.status...
ecal.status

# ...or by using function check.cal:
check.cal(descal06p.forced)
```

`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 (in particular, Section 'Calibration process diagnostics').

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 *successfully*. 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 data frame (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. (2007) *"Introduction to Variance Estimation"*, Second Edition, 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

```
#############################################
# Explore alternative collapsing strategies. #
#############################################

# 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=stat.score[match(stratum,stat.score$stratum),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)
 # "average" 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.


############################################################
# A simple way for defining the strata similarity scores. #
############################################################
# Suppose that strata have been clustered in groups of similar
# strata. You can, then, use the integer codes of the factor
# variable identifying the clusters as a similarity score.
# You can do as follows:
```

```
  # Load some data:
    data(fpcdat)

  # Build a design object:
    fpcdes<-e.svydesign(data=fpcdat,ids=~psu+ssu,strata=~stratum,weights=~w)
    fpcdes

  # As we deliberately omitted to specify fpcs, this design
  # has 2 lonely strata out of 5:
    find.lon.strata(fpcdes)

  # Now, suppose that factor variable pl.domain identifies clusters of
  # similar strata...
    table(fpcdat$stratum,fpcdat$pl.domain)

  # ...hence, the similarity score can be obtained simply...
    fpcdes<-des.addvars(fpcdes,score=unclass(pl.domain))

  # ...and readily be used to drive the strata collapsing:
    fpcdes.clps<-collapse.strata(fpcdes,sim.score=~score)
    fpcdes.clps
    clps.strata.status

  # As we expected from the groups defined by pl.domain, lonely stratum S.2
  # has been paired to S.3, and lonely stratum S.5 to S.4.

  # Should we have omitted to specify a similarity score, we would have
  # obtained different superstrata:
    fpcdes.clps2<-collapse.strata(fpcdes)
    fpcdes.clps2
    clps.strata.status


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

`contrasts.RG` *Set, Reset or Switch Off Contrasts for Calibration Models*

Description

These functions control the way **ReGenesees** translates a symbolic calibration model (as specified by the `calmodel` formula in `e.calibrate`, `pop.template`, `fill.template`, `aux.estimates`, ...) to its numeric encoding (i.e. the model-matrix used by the internal algorithms to perform actual computations).

Usage

```
contrasts.RG()
contrasts.off()
contrasts.reset()
contr.off(n, base = 1, contrasts = TRUE, sparse = FALSE)
```

Arguments

`n`	Formally as in function `contr.treatment` (see 'Details').
`base`	Formally as in function `contr.treatment` (see 'Details').
`contrasts`	Fictitious, but formally as in function `contr.treatment`. (see 'Details')
`sparse`	Formally as in function `contr.treatment`. (see 'Details')

Details

All the calibration facilities in package **ReGenesees** transform symbolic calibration models (as specified by the user via `calmodel`) into numeric model-matrices. Factor variables occurring in `calmodel` play a special role in such transformations, as the encoding of a factor can (and, by default, do) *depend* on the *structure* of the formula in which it occurs. The **ReGenesees** functions documented below control the way factor levels are translated into auxiliary variables and mapped to columns of population totals data frames. The underlying technical tools are `contrasts` handling functions (see Section 'Technical Remarks and Warnings' for further details).

Under the calibration perspective, ordered and unordered factors appearing in `calmodel` must be treated the same way. This obvious constraint defines the **ReGenesees** default for contrasts handling. Such a default is silently set when loading the package. Moreover, you can set it also by calling `contrasts.RG()`. As can be understood by reading Section 'Technical Remarks and Warnings' below, the default setup can be seen as **"efficient-but-slightly-risky"**.

A call to `contrasts.off()` simply disables all contrasts and imposes a complete dummy coding of factors. Under this setup, all levels of factors occurring in `calmodel` generate a distinct model-matrix column, *even if some of these columns can be linearly dependent*. To be very concise, the `contrasts.off()` setup can be seen as **"safe-but-less-efficient"** as compared to the default one (read Section 'Technical Remarks and Warnings' for more details).

Function `contr.off` is not meant to be called directly by users: it serves only the purpose of enabling the `contrasts.off()` setup.

A call to `contrasts.reset()` restores R factory-fresh defaults for contrasts (which do distinguish ordered and unordered factors). Users may want to use this function after having completed a **ReGenesees** session, e.g. before switching to other R functions relying on contrasts (such as `lm`, `glm`, ...).

Technical Remarks and Warnings

"*[...] the corner cases of model.matrix and friends is some of the more impenetrable code in the R sources.*"
Peter Dalgaard

Contrasts handling functions tell R how to encode the model-matrix associated to a given model-formula on specific data (see, e.g., `contr.treatment`, `contrasts`, `model.matrix`, `formula`, and references therein). More specifically, contrasts control the way factor-terms and interaction-terms occurring in formulae get actually represented in the model matrix. For instance, R (by default) avoids the complete dummy coding of a factor whenever it is able to understand, on the basis of the structure of the model-formula, that some of the factor levels would generate linearly dependent (i.e. redundant) columns in the model-matrix (see Section 'Examples').

The usage of contrasts to build smaller, full-rank calibration model-matrices would be a good opportunity for **ReGenesees**, provided it comes *without any information loss*. Indeed, smaller model-matrices mean less population totals to be provided by users, and higher efficiency in computations.

Unfortunately, few controversial cases have been signalled in which R ability to "simplify" a model-matrix on the basis of the structure of the related model-formula seems to lead to strange, unexpected results (see, e.g., this R-help thread). No matter whether such R behaviour is or not an actual bug with respect to its impact on R linear model fitting or ANOVA facilities, it surely represents a concern for **ReGenesees** with respect to calibration (see Section 'Examples'). The risk is the following: there could be rare cases in which exploiting R contrasts handling functions inside **ReGenesees** ends up with a *wrong* (i.e. incomplete) population totals template, and (eventually) with *wrong* calibration results.

Though one could adopt several ad-hoc countermeasures to sterilize the risk described above while still taking advantage of contrasts (see Section 'Examples'), the choice of completely disabling contrasts via `contrasts.off()` would result in a **100% safety guarantee**. If computational efficiency is not a serious concern for you, switching off contrasts may determine the best **ReGenesees** setup for your analyses.

Author(s)

Diego Zardetto

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical models*. Chapter 2 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

"Why does the order of terms in a formula translate into different models/model matrices?", R-help thread.

See Also

`e.calibrate`, `pop.template`, `fill.template`, and `aux.estimates` for the meaning and the usage of `calmodel` in **ReGenesees**. `formula`, `model.matrix`, `contrasts`, and `contr.treatment` to understand the role of contrasts in R.

Examples

```
######################
# Easy things first: #
######################
```

```
  # 1) When ReGenesees is loaded, its standard way of handling contrasts
  #    (i.e. no ordered-unordered factor distinction) is silently set:
options("contrasts")

  # 2) To switch off contrasts (i.e. apply always dummy coding to factors),
  #    simply type:
contrasts.off()

  # 3) To restore R factory-fresh defaults for contrasts, simply type:
contrasts.reset()

  # 4) To switch on again standard ReGenesees contrasts, simply type:
contrasts.RG()


############################################################
# A simple calibration example to understand the effects of #
# switching off contrasts.                                  #
############################################################

# Load sbs data:
data(sbs)

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

# Suppose you want to calibrate on the marginals of 'region' (a factor
# with 3 levels: "North", "Center", and "South") and 'dom3' (a factor
# with 4 levels: "A", "B", "C", and "D").
# Let's see how things go under the 'contrast on' (default) and 'contrasts off'
# setups:

  ########################################
  # 1) ReGenesees default: contrasts ON. #
  ########################################
  # As you see contrasts are ON:
  options("contrasts")

  # Build and fill the population totals template:
  temp1<-pop.template(data=sbsdes,calmodel=~region+dom3-1)
  pop1<-fill.template(universe=sbs.frame,template=temp1)

  # Now inspect the obtained known totals data.frame:
  pop1

  # As you see: (i) it has only 6 columns, and (ii) the "A" level of
  # factor 'dom3' is missing. This is because contrasts are ON, so that
  # R is able to understand that only 6 out of the 3 + 4 marginal counts
  # are actually independent. Indeed, the "A" counts...
  sum(sbs.frame$dom3=="A")

  # ...are actually redundant, since they can be deduced by pop1:
  sum(pop1[,1:3])-sum(pop1[,4:6])

  # Now calibrate:
  cal1<-e.calibrate(sbsdes,pop1)
```

```
  ################################################
  # 2) Switch OFF contrasts: dummy coding for all! #
  ################################################
  # To switch off contrasts simply call:
  contrasts.off()

  # Build and fill the population totals template:
  temp2<-pop.template(data=sbsdes,calmodel=~region+dom3-1)
  pop2<-fill.template(universe=sbs.frame,template=temp2)

  # Now inspect the obtained known totals data.frame:
  pop2

  # As you see: (1) it has now 7 columns, and (2) the "A" level of factor
  # 'dom3' has been resurrected. This is because contrasts are OFF,
  # so that each level of factors in calmodel are coded to dummies.

  # Now calibrate. Since only 6 out of 7 dummy auxiliary variables are
  # actually independent, the model.matrix computed by e.calibrate will not be
  # full-rank. As a consequence, e.calibrate would use the Moore-Penrose
  # generalized inverse (in practice, this could depend on the machine R
  # is running on):
  cal2<-e.calibrate(sbsdes,pop2)

  # Compare the calibration weights generated under setups 1) and 2):
  all.equal(weights(cal2),weights(cal1))

  # Lastly set back contrasts to ReGenesees default:
  contrasts.RG()


##############################################
# Weird results, risks and countermeasures. #
# ("When the going gets tough...")          #
##############################################

# Suppose you want to calibrate on: (A) the joint distribution of 'region' (a
# factor with 3 levels: "North", "Center", and "South") and 'nace.macro' (a
# factor with 4 levels: "Agriculture", "Industry", "Commerce", and "Services")
# and, at the same time, on (B) the total number of employees ('emp.num', a
# numeric variable) by 'nace.macro'.
#
# You rightly expect that 3*4 + 4 = 16 population totals are needed for such a
# calibration task. Indeed, knowing the enterprise counts for the 3*4 cells of
# the joint distribution (A) doesn't tell anything on the number of employees
# working in the 4 nace macrosectors (B), and vice-versa.
#
# Moreover, you might expect that calibration models:
  # (i)  calmodel = ~region:nace.macro  + emp.num:nace.macro - 1
  # (ii) calmodel = ~emp.num:nace.macro + region:nace.macro  - 1
#
# should produce the same results.
# Unfortunately, WHEN CONTRASTS ARE ON, this is not the case: only model (i)
# leads to the expected, right results. Let's see.

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

```
# A strange result when contrasts are ON: #
# the order of terms in calmodel matters! #
##########################################
# As you see contrasts are ON:
options("contrasts")

# Start with (i) calmodel = ~region:nace.macro + emp.num:nace.macro - 1
# Build and fill the population totals template:
temp1<-pop.template(data=sbsdes,~region:nace.macro+emp.num:nace.macro-1)
pop1<-fill.template(universe=sbs.frame,template=temp1)

# Now inspect the obtained known totals data.frame:
pop1

# and verify it stores the right, expected number of totals (i.e. 16):
dim(pop1)

# Now calibrate:
cal1<-e.calibrate(sbsdes,pop1)


# Now compare with (ii) calmodel = ~emp.num:nace.macro + region:nace.macro - 1
# Build and fill the population totals template:
temp2<-pop.template(data=sbsdes,~emp.num:nace.macro+region:nace.macro-1)
pop2<-fill.template(universe=sbs.frame,template=temp2)

# First check if it stores the right, expected number of totals (i.e. 16):
dim(pop2)

# Apparently 4 totals are missing; let's inspect the known totals data.frame
# to understand which ones:
pop2

# Thus we are missing the 4 'nace.macro' totals for 'region' level "North".
# Everything goes as if R contrasts functions mistakenly treated the term
# emp.num:nace.macro as a factor-factor interaction (i.e. a 2 way joint
# distribution), which would have justified to eliminate the 4 missing totals
# as redundant.

# Notice that calibrating on pop2 would generate wrong results...
cal2<-e.calibrate(sbsdes,pop2)

# ...indeed the 4 estimates of 'nace.macro' for 'region' level "North" are not
# actually calibrated (look at the magnitude of SE estimates):
svystatTM(cal2,~region,~nace.macro)


#################################################################
# A possible countermeasure (still working with contrasts ON). #
#################################################################
# Empirical evidence tells that the weird case above is extremely rare
# and that it manifests whenever a numeric (say X) and a factor (say F) both
# interact with the same factor (say D), i.e. calmodel=~(X+F):D-1.
#
# The risky order-dependent nature of such models can be sterilized (while
# still taking advantage of contrasts-driven simplifications for large,
# complex calibrations) by using a numeric variable with values 1 for
```

```
# all sample units.
#
# For instance, one could use variable 'ent' in the sbs data.frame, to
# handle the (A) part of the calibration constraints. Indeed you may easily
# verify that both the calmodel formulae below:
# (i)  calmodel = ~ent:region:nace.macro  + emp.num:nace.macro - 1
# (ii) calmodel = ~emp.num:nace.macro + ent:region:nace.macro  - 1
#
# produce exactly the same, right results.


####################################################################
# THE ULTIMATE, 100% SAFE, COUNTERMEASURE: switch contrasts OFF! #
####################################################################
# No contrasts means no model-matrix simplifications at all, hence
# also no unwanted, wrong simplifications. Let's see:

# To switch off contrasts simply call:
contrasts.off()

# Compare again, with contrasts OFF, the calibration models:
# (i)  calmodel = ~region:nace.macro  + emp.num:nace.macro - 1
# (ii) calmodel = ~emp.num:nace.macro + region:nace.macro  - 1

# Build and fill the population totals templates:
temp1<-pop.template(data=sbsdes,~region:nace.macro+emp.num:nace.macro-1)
pop1<-fill.template(universe=sbs.frame,template=temp1)
pop1

temp2<-pop.template(data=sbsdes,~emp.num:nace.macro+region:nace.macro-1)
pop2<-fill.template(universe=sbs.frame,template=temp2)
pop2

# Verify they store the same, right number of totals (i.e. 16):
dim(pop1)
dim(pop2)

# Verify they lead to right calibrated objects...
cal1<-e.calibrate(sbsdes,pop1)
cal2<-e.calibrate(sbsdes,pop2)

# ...with the same calibrated weights:
all.equal(weights(cal2),weights(cal1))

# Lastly set back contrasts to ReGenesees default:
contrasts.RG()
```

`Corr` *Design Covariance and Correlation of Complex Estimators in Subpopulations*

Description

Estimates the covariance and the correlation of Complex Estimators in subpopulations. A Complex Estimator can be any analytic function of (Horvitz-Thompson or Calibration) estimators of Totals and Means.

Usage

```
CoV(design, expr1, expr2,
     by = NULL, na.rm = FALSE)
Corr(design, expr1, expr2,
     by = NULL, na.rm = FALSE)
```

Arguments

design	Object of class analytic (or inheriting from it) containing survey data and sampling design metadata.
expr1	R expression defining the first Complex Estimator (see 'Details').
expr2	R expression defining the second 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.
na.rm	Should missing values (if any) be removed from the variables of interest? The default is FALSE (see 'Details').

Details

This function allows to estimate the covariance and the correlation of two arbitrary Complex Estimators. Estimates are calculated using the Taylor linearization technique.

The mandatory arguments expr1 and expr2 identify the Complex Estimators: both must be of class expression. For further details on the syntax and the semantics of such expressions, see svystatL.

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 CoV (Corr) 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 CoV (Corr) twice. The design variables referenced by by (if any) should be of type factor, otherwise they will be coerced.

Missing values (NA) in interest variables should be avoided. If na.rm=FALSE (the default) they generate NAs in estimates (or even an error, if design is calibrated). If na.rm=TRUE, observations containing NAs are dropped, and estimates gets computed on non missing values only. This implicitly assumes that missing values hit interest variables *completely at random*: should this not be the case, computed estimates would be *biased*.

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.

See Also

Estimators of Totals and Means svystatTM, Ratios between Totals svystatR, Shares svystatS, Ratios between Shares svystatSR, Multiple Regression Coefficients svystatB, Quantiles svystatQ, and Complex Analytic Functions of Totals and/or Means svystatL.

Examples

```
########################################
# Some checks and some simple examples #
# to illustrate the syntax.            #
########################################
# Load survey data:
data(data.examples)

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

# Let's start with some natural checks:
  ## The covariance of any estimator with itself is its variance
  ## (use mean income as an example):
     CoV(des,expression(income/ones),expression(income/ones))
     VAR(svystatL(des,expression(income/ones)))
     VAR(svystatTM(des,~income,estimator="Mean"))

  ## The correlation of any estimator with itself is 1
  ## (use mean income as an example):
     Corr(des,expression(income/ones),expression(income/ones))

# Switch to non trivial examples:
  ## Correlation of mean income with population size:
     Corr(des,expression(income/ones),expression(ones))

  ## Correlation of mean income with total income:
     # at population level:
     Corr(des,expression(income/ones),expression(income))
     # for regions:
     Corr(des,expression(income/ones),expression(income),by=~regcod)


  ## Correlation of a product of two totals and a ratio of two totals:
     # at population level:
     Corr(des,expression(y1*y2),expression(x1/x2))
     # for provinces:
     Corr(des,expression(income/ones),expression(income),by=~procod)


#####################################################
# A more meaningful and complex example: correlation #
# between Geometric, Harmonic and Arithmetic Means.  #
#####################################################
# Creation of another design object:
data(sbs)
sbsdes<-e.svydesign(data=sbs,ids=~id,strata=~strata,weights=~weight,
        fpc=~fpc)
```

```
# Let's use variable emp.num, which is ok as it is always strictly positive:
  ## Add a convenience variable for estimating the harmonic mean (see ?svystatL
  ## for details) and prepare the formal estimator expression:
     sbsdes<-des.addvars(sbsdes,emp.num.m1=1/emp.num)
     h<-expression(ones/emp.num.m1)

  ## Add a convenience variable for estimating the geometric mean (see ?svystatL
  ## for details) and prepare the formal estimator expression:
     sbsdes<-des.addvars(sbsdes,log.emp.num=log(emp.num))
     g<-expression(exp(log.emp.num/ones))

  ## prepare the formal estimator expression for the arithmetic mean:
     m<-expression(emp.num/ones)

# Now compute correlations:
  ## Harmonic with Arithmetic:
     Corr(sbsdes,h,m)

  ## Geometric with Arithmetic:
     Corr(sbsdes,g,m)

  ## Harmonic with Geometric:
     Corr(sbsdes,h,g)

  ## Hence, while correlations g-m and g-h are high, correlation h-m is low.


####################################################
# Another example: is a ratio estimator of a total #
# expected to be more efficient than an HT one?    #
####################################################

# Let's recall that the ratio estimator of a total is
# expected to be more efficient than HT, if the
# correlation of numerator and denominator exceeds
# half of the ratio between the CVs of denominator
# and numerator.

# Compute the HT estimate of the total of value added (variable va.imp2):
VA<-svystatTM(sbsdes,~va.imp2)
VA

# Compute the HT estimate of the total of emp.num:
EMP<-svystatTM(sbsdes,~emp.num)
EMP

# Now estimate the correlation of the numerator
# and denominator totals:
corr <- Corr(sbsdes,expression(va.imp2),expression(emp.num))
corr

# and compare it with (1/2)*( CV(den)/CV(num) )
stopifnot( corr > 0.5*cv(EMP)/cv(VA) )

# As the comparison holds TRUE, we expect an efficiency gain
# of the ratio estimator of the total compared to HT.
# Let's check...:
```

```
  # Compute the ratio estimate of the total of value added using
  # as auxiliary variable the number of emloyees, whose total
  # is 984394:
  TOT.emp.num <- sum(sbs.frame$emp.num)
  TOT.emp.num

  VA.ratio<-svystatL(sbsdes, expression(TOT.emp.num * (va.imp2/emp.num)))
  VA.ratio

  # Compare standard errors sizes:
  SE(VA.ratio)
  SE(VA)
  stopifnot( SE(VA.ratio) < SE(VA) )

# ...as expected.
```

`data.examples` *Artificial Household Survey Data*

Description

Example data frames. 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 partitioned 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.

Warning

Data in the `example` data frame are artificial. The *structure* of `example` intentionally resembles the one of typical household survey data, but the *values* it stores are unreliable. The only purpose of such data is that they can be fruitfully exploited to illustrate the syntax and the working mechanism of the functions provided by the **ReGenesees** package.

Examples

```
data(data.examples)
head(example)
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. This an intentional feature meant to safeguard the integrity of the relations between survey data and sampling design metadata stored in design.

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 median income for the whole population):
Median.Income <- coef(svystatQ(des, ~income,probs=0.5))
Median.Income
des <- des.addvars(des,
                   status = factor(
                          ifelse(income < (0.6 * Median.Income),
                          "poor", "non-poor")
                                  )
                   )
svystatTM(des,~status,estimator="Mean")
# Mean income for poor and non-poor:
svystatTM(des,~income,~status,estimator="Mean")

### NOTE: Procedure above yields *correct point estimates* of the share of poor
###       population and their average income, while *variance estimation is
###       approximated* since we neglected the sampling variability of the
###       estimated poverty threshold.
```

`des.merge` *Merge New Survey Data into Design Objects*

Description

Modifies an analytic object by joining the original survey data with a new data frame via a common key.

Usage

```
des.merge(design, data, key)
```

Arguments

`design`	Object of class `analytic` (or inheriting from it) containing survey data and sampling design metadata.
`data`	Data frame containing a key variable, plus new variables to be merged to `design` data.
`key`	Formula identifying the common key variable to be used for merging.

Details

This function updates the survey variables contained into `design` (i.e. `design$variables`), by merging the original data with those contained into the `data` data frame. The merge operation exploits a *single* variable `key`, which must be common to both `design` and `data`.

The function preserves both the original *ordering* of the survey data stored into `design`, as well as all the original sampling design *metadata*.

The variable referenced by `key` must be a *valid* unique key for both `design` and `data`: it must not contain duplicated values, nor `NA`s. Moreover, the values of `key` in `design` and `data` must be in

1:1 correspondence. These requirements are meant to ensure that the *new* survey data (that is the merged ones) will have exactly the same number of rows as the *old* survey data stored into `design`.

Should `design` and `data` contain further common variables besides the `key`, only their original `design` version will be retained. Thus, `des.merge` cannot modify any pre-existing `design` columns. This an intentional feature intended to safeguard the integrity of the relations between survey data and sampling design metadata stored in `design`.

Value

An object of the same class of `design`, containing additional survey data but supplied with exactly the same metadata.

Practical Purpose

In the field of Official Statistics, it is not infrequent that calibration weights must be computed even several months before the target variables of the survey are made available for estimation. Such a time lag follows from the fact that target variables typically undergo much more thorough editing and imputation procedures than auxiliary variables.

In such production scenarios, function `des.merge` allows to tackle the task of computing estimates and errors for the fresh-released target variables *without* any need of *repeating* the calibration step. Indeed, by using the function, one can join the data contained into an already calibrated `design` object with new `data` made available only after the calibration step. The merge operation is made easy and safe, and preserves all the original calibration metadata (e.g. those needed for variance estimation).

Author(s)

Diego Zardetto

See Also

`e.svydesign` to bind survey data and sampling design metadata, `e.calibrate` for calibrating weights, `des.addvars` to add new variables to design objects.

Examples

```
data(data.examples)

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

# Create a calibrated design object as well (e.g. using population totals
# stored inside pop03p):
cal<-e.calibrate(design=des,df.population=pop03p,
                 calmodel=~marstat-1,partition=~sex,calfun="logit",
                 bounds=bounds)

# Lastly create a new data frame to be merged into des and cal:
set.seed(12345)    # RNG seed fixed for reproducibility
new.data<-example[,c("income","key")]
new.data$income <- 1000 + new.data$income    # altered income values
new.data$NEW.f<-factor(sample(c("A","B"),nrow(new.data),rep=TRUE))
new.data$NEW.n<-rnorm(nrow(new.data),10,2)
```

```
new.data <- new.data[sample(1:nrow(new.data)), ]    # rows ordering changed
head(new.data)

############################################################
# Example 1: merge new data into a non calibrated design. #
############################################################

# Merge new data inside des (note the warning on income):
des2<-des.merge(design=des,data=new.data,key=~key)

# Compare visually:
## before:
head(des$variables)
## after:
head(des2$variables)

# New data can be used as usual:
svystatTM(des2,~NEW.n,~NEW.f,vartype="cvpct")

# Old data are unaffected, as it must be:
svystatTM(des,~income,estimator="Mean",vartype="cvpct")
svystatTM(des2,~income,estimator="Mean",vartype="cvpct")

#########################################################
# Example 2: merge new data into a calibrated design. #
#########################################################

# Merge new data inside cal (note the warning on income):
cal2<-des.merge(design=cal,data=new.data,key=~key)

# Compare visually:
## before:
head(cal$variables)
## after:
head(cal2$variables)

# New data can be used as usual:
svystatTM(cal2,~NEW.n,~NEW.f,vartype="cvpct")

# Old data are unaffected, as it must be:
svystatTM(cal,~income,estimator="Mean",vartype="cvpct")
svystatTM(cal2,~income,estimator="Mean",vartype="cvpct")
```

`drop.gvf.points` *Drop Outliers and Refit a GVF Model*

Description

This function drops observations (alleged outliers) from a fitted GVF model and simultaneously re-fits the model.

Usage

```
drop.gvf.points(x, method = c("pick", "cut"), which.plot = 1:2,
```

```
res.type = c("standard", "student"), res.cut = 3,
id.n = 3, labels.id = NULL,
cex.id = 0.75, label.pos = c(4, 2),
cex.caption = 1, col = NULL, drop.col = "red",
...)
```

Arguments

x	An object containing a *single* fitted GVF model (i.e. of class gvf.fit or gvf.fit.gr).
method	character specifying the method for identifying observations to be dropped (see 'Details'); it may be either 'pick' (the default) or 'cut'.
which.plot	integer controlling the nature of the plot(s) that are used to identify and/or visualize the observations to be dropped: 1 means 'Observed vs Fitted', 2 means 'Residuals vs Fitted' (see 'Details').
res.type	character specifying what kind of residuals must be used.
res.cut	A positive value: observations to be dropped will be those with residuals whose absolute value exceeds 'res.cut'. Only meaningful if method is 'cut'.
id.n	Number of points to be initially labelled in each plot, starting with the most extreme. Only meaningful if method is 'pick'.
labels.id	Vector of labels, from which the labels for extreme points will be chosen. NULL uses observation numbers.
cex.id	Magnification of point labels.
label.pos	Positioning of labels, for the left half and right half of the graph(s) respectively.
cex.caption	Controls the size of caption.
col	Color to be used for the points in the plot(s).
drop.col	Color to be used to visualize and annotate the points to be dropped in the plot(s).
...	Other parameters to be passed through to plotting functions.

Details

This function drops observations (alleged outliers) from a *single* fitted GVF model and simultaneously re-fits the model. As a side effect, the function prints on screen the induced change for selected quality measures (see, e.g., getR2).

If method = "pick", observations to be dropped are identified interactively by clicking on points of a plot (see 'Note'). Argument which.plot determines the nature of the plot: value 1 is for *'Observed vs Fitted'*, value 2 is for *'Residuals vs Fitted'*. In the latter case, argument res.type specifies what kind of residuals have to be plotted. Argument id.n specifies how many points have to be labelled initially, starting with the most extreme in terms of the selected residuals: this applies to both kinds of plots.

If method = "cut", observations to be dropped are those with residuals whose absolute value exceeds the value of argument res.cut. Again, argument res.type specifies what kind of residuals have to be used (and plotted). The points which have been cut will be highlighted on a plot, whose nature is again specified by argument which.plot. If which.plot = 1:2, dropped points will be visualized on both the *'Observed vs Fitted'* and the *'Residuals vs Fitted'* graphs simultaneously.

Argument drop.col controls the color to be used to visualize and annotate in the plot(s) the points to be dropped. All the other arguments have the same meaning as in function plot.lm.

Value

An object of the same class as x (i.e. either gvf.fit or gvf.fit.gr), containing the original GVF model re-fitted after dropping (alleged) outliers.

Note

For method = "pick", function drop.gvf.points is only supported on those screen devices for which function identify is supported. The identification process can be terminated either by right-clicking the mouse and selecting 'Stop' from the menu, or from the 'Stop' menu on the graphics window.

Author(s)

Diego Zardetto

See Also

GVF.db to manage **ReGenesees** archive of registered GVF models, gvf.input and svystat to prepare the input for GVF model fitting, fit.gvf to fit GVF models, plot.gvf.fit to get diagnostic plots for fitted GVF models, and predictCV to predict CV values via fitted GVF models.

Examples

```
# Load example data:
data(AF.gvf)

# Inspect available estimates and errors of counts:
str(ee.AF)

# List available registered GVF models:
GVF.db

# Fit example data to registered GVF model number one:
m <- fit.gvf(ee.AF, model=1)
m
summary(m)

################################################################
# Method 'pick': identify outlier observations to be dropped #
# interactively by clicking on points of a plot.             #
################################################################
  # Using the 'Observed vs Fitted' plot (the default):
## Not run:
m1 <- drop.gvf.points(m)
m1
summary(m1)

## End(Not run)

 # Using the 'Residuals vs Fitted' plot with standardized
 # residuals (the default) and increasing id.n to get more
 # labelled points to guide your choices:
## Not run:
m1 <- drop.gvf.points(m, which.plot = 2, id.n = 10)
m1
summary(m1)
```

```
## End(Not run)

 # The same as above, but with studentized residuals and
 # playing with colors:
## Not run:
m1 <- drop.gvf.points(m, which.plot = 2, id.n = 10, res.type = "student",
                      col = "blue", drop.col = "green", pch = 20)
m1
summary(m1)

## End(Not run)


############################################################
# Method 'cut': identify outlier observations to be dropped #
# by specifying a threshold for the absolute values of the  #
# residuals.                                                #
############################################################
 # Using default threshold on standardized residuals and visualizing
 # dropped observations on both 'Observed vs Fitted' and 'Residuals
 # vs Fitted' plots:
m1 <- drop.gvf.points(m, method ="cut")
m1
summary(m1)

 # Using a custom threshold on studentized residuals and visualizing
 # dropped observations on the 'Observed vs Fitted' plot:
m1 <- drop.gvf.points(m, method ="cut",  res.type = "student",
                      res.cut = 2.5, which.plot = 1)
m1
summary(m1)

 # The same as above, but visualizing dropped observations on the
 # 'Residuals vs Fitted' plot:
m1 <- drop.gvf.points(m, method ="cut",  res.type = "student",
                      res.cut = 2.5, which.plot = 2)
m1
summary(m1)

 # You can obviously "cut"/"pick" alleged outliers again from an already
 # "cut"/"picked" fitted GVF model:
m2 <- drop.gvf.points(m1, method ="cut",  res.type = "student",
                      res.cut = 2.5, col = "blue", pch = 20)
m2
summary(m2)


#####################################################################
# Identifying outlier observations to be dropped from "grouped" #
# GVF fitted models (i.e. x has class 'gvf.fit.gr').            #
#####################################################################
 # Recall we have at our disposal the following survey design object
 # defined on household data:
exdes

 # Now use function svystat to prepare "grouped" estimates and errors
```

```
 # of counts to be fitted separately (here groups are regions):
ee <- svystat(exdes, y=~ind, by=~age5c:marstat:sex, combo=3, group=~regcod)
ee
plot(ee)

 # Fit registered GVF model number one separately inside groups:
m <- fit.gvf(ee, model=1)
m
summary(m)

 # Now drop alleged outliers separately inside groups:

   ######################################################
   # Method 'pick': work interactively group by group. #
   ######################################################
## Not run:
   m1 <- drop.gvf.points(m, which.plot = 2, res.type = "student", col = "blue",
                         pch = 20)
   m1
   summary(m1)

## End(Not run)

   ########################################################
   # Method 'cut': apply the same threshold to all groups. #
   ########################################################
   m1 <- drop.gvf.points(m, method ="cut",  res.type = "student", res.cut = 2)
   m1
   summary(m1)
```

`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 to use, that is - in the language of the Generalized Regression Estimator - the assisting linear regression model underlying the calibration problem. More specifically, the calmodel formula identifies the auxiliary variables and the constraints for the calibration problem, with a notation inspired by [Wilkinson, Rogers 73]. For example, calmodel=~(X+Z):C+(A+B):D-1 defines the calibration problem in which constraints are imposed: (i) on the totals of 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 *"factorized"* 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-1 in which the variable that identifies the calibration domains is D; similarly, the formula calmodel=~(A+B+X+Z):D1:D2-1 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-1 is equivalent to the sequence of problems defined by the *"reduced model"* calmodel=~A+B+X+Z-1 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 factorized, whether to solve the problem globally or in a partitioned way (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 partitioned 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=~A+B+X+Z-1`), 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 `partition=~D1+D2` will be automatically translated into the factor-crossing formula `partition=~D1:D2`.
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 four functions: `pop.template`, `population.check`, `pop.desc` 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 `pop.desc` function provides a natural language description of the template structure, 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 Generalized 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*.
Note that it is technically possible to exploit this behaviour in order to *prevent some subset of the initial weights from being altered by calibration*. The trick is simple: just build a convenience sigma2 variable whose values are set to some very high value (e.g. 1E12) for those units whose initial weight must be preserved, and to 1 otherwise (see 'Examples'). Nevertheless, this trick should be used sparingly and very carefully, as otherwise it may: (i) cause the calibration algorithm to not converge, (ii) result in introducing bias in calibration estimates. In particular, with respect to bias, one should not select the units whose weight must be preserved on the basis of the current sample.
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 optimization 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, nevertheless 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 column. 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 factorizable calibration problem, the user selects the partitioned 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:

```
CODE        MEANING
  -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 (1) 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 specialization of the `analytic` class; this means that an object created by `e.calibrate` inherits from the `analytic` class and you can use on it all methods defined on the latter class, e.g. `print`, `summary`, `weights`. Moreover, 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.

Sarndal, C.E., Lundstrom, S. (2005) Estimation in surveys with nonresponse. John Wiley & Sons.

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., https://www.istat.it/it/files/2018/07/2007_4.pdf.

See Also

`e.svydesign` to bind survey data and sampling design metadata, `svystatTM`, `svystatR`, `svystatS`, `svystatSR`, `svystatB`, `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, `pop.desc` to provide a natural language description of the template structure, `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, \ldots, pop05p and the    #
# bounds vector are all contained in the data.examples file).        #
# For the examples relating to calibration models that can be        #
# factorized both a global and a partitioned solution are given.     #
#########################################################################

# Load household data:
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)

# Printing descal01 immediately recalls that it is a
# "calibrated" object:
descal01

# Use the summary() function if you need some additional details, e.g.:
summary(descal01)

# Use the 'variables' slot to extract survey data, e.g.:
head(descal01$variables)

# Use the weights() function to extract weights, e.g.:
summary(weights(descal01))

# 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 partitioned 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):regcod-1,calfun="logit",
          bounds=bounds,aggregate.stage=2)

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


# 4.1) Same problem with the partitioned 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):sex:marstat-1,calfun="logit",
          bounds=bounds)

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


# 5.1) Same problem with the partitioned 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).


# 5.2) Just a single example to inspect the ecal.status list generated
#      for diagnostics purposes.
#      Let's shrink the bounds in order to prevent perfect convergence
#      (recall that force=TRUE by default):
approx.cal<-e.calibrate(design=des,df.population=pop05p,
            calmodel=~age5c+x1-1,partition=~sex:marstat,
            calfun="logit",bounds=c(0.95,1.05))

# ...now use check.cal function to assess the amount of calibration
# constraints violation:
check.cal(approx.cal)

# ...or (equivalently) inspect directly ecal.status:
ecal.status


################################################
# Some examples illustrating how calibration   #
# can be exploited to reduce nonresponse bias  #
# (see, e.g. [Sarndal, Lundstrom 05]).         #
################################################

# Load sbs data:
data(sbs)

  ######################
  # Full-response case. #
  ######################

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

  # Now estimate the average value added and its 95% confidence interval:
  mean.VA<-svystatTM(design=sbsdes,y=~va.imp2,estimator="Mean",vartype= "cvpct",
           conf.int=TRUE,conf.lev=0.95)
  mean.VA

  # Compare the obtained estimate with the true population parameter:
  MEAN.VA<-mean(sbs.frame$va.imp2)
  MEAN.VA

  # We get a small overestimation of about 4%...
  100*(coef(mean.VA)-MEAN.VA)/MEAN.VA

  # which, anyway, doesn't indicate a significant bias for the
  # full-response sample, because the 95% confidence interval
  # covers the true value.
```

```
################################################
# Nonresponse case: assume a response propensity #
# which increases with enterprise size.          #
################################################

# Set bigger response probabilities for bigger firms,
# e.g. exploiting available information about the
# number of employees (emp.cl):
levels(sbs$emp.cl)
p.resp <- c(.4,.6,.8,.95,.99)

# Tie response probabilities to sample observations:
pr<-p.resp[unclass(sbs$emp.cl)]

# Now, randomly select a subsample of responding units from sbs:
set.seed(12345)          # (fix the RNG seed for reproducibility)
rand<-runif(1:nrow(sbs))
sbs.nr<-sbs[rand<pr,]

# This implies an overall response rate of about 73%:
nrow(sbs.nr)/nrow(sbs)

# Treat the non-response sample as it was complete: this should
# lead to biased estimates of value added, as the latter is
# positively correlated with firms size...
sbsdes.nr<-e.svydesign(data=sbs.nr,ids=~id,strata=~strata,weights=~weight)

#...indeed:
old.op <- options("RG.lonely.psu"="adjust")   # (prevent lonely-PSUs troubles)
mean.VA.nr<-svystatTM(design=sbsdes.nr,y=~va.imp2,estimator="Mean",
            vartype= "cvpct",conf.int=TRUE,conf.lev=0.95)
mean.VA.nr

# and, comparing with the true population average, we see a
# significant overestimation effect, with the 95% confidence
# interval not even covering the parameter:
MEAN.VA

# Nonresponse bias can be effectively reduced by calibrating
# on variables explaining the response propensity: e.g., in
# the present example, on the population distribution of emp.cl:
  # Prepare the known totals template...
  N.emp.cl<-pop.template(data=sbs.nr,calmodel=~emp.cl-1)
  N.emp.cl

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

  # Lastly calibrate:
    # Get a hint on the calibration bounds:
    hint<-bounds.hint(sbsdes.nr,N.emp.cl)
    sbscal.nr<-e.calibrate(design=sbsdes.nr,df.population=N.emp.cl,
               bounds=hint)
    sbscal.nr
```

```
# Now estimate the average value added on the calibrated design:
mean.VA.cal.nr<-svystatTM(design=sbscal.nr,y=~va.imp2,estimator="Mean",
            vartype= "cvpct",conf.int=TRUE,conf.lev=0.95)

# options(old.op)   # (reset variance estimation options)

# As expected, we see a significant bias reduction:
MEAN.VA
mean.VA.nr
mean.VA.cal.nr

# Even if the 95% confidence interval still doesn't cover the
# true value, by calibration we passed from an initial overestimation
# of about 33% to a 7% one:
100*(coef(mean.VA.nr)-MEAN.VA)/MEAN.VA
100*(coef(mean.VA.cal.nr)-MEAN.VA)/MEAN.VA


################################################
# A multi-step calibration example showing that #
# a calibrated object can be calibrated again   #
# (this can be sometimes useful in practice):   #
# Step 1: calibrate to reduce nonresponse bias; #
# Step 2: calibrate again to gain efficiency.   #
################################################

# Suppose you already performed a first calibration step,
# as shown in the example above, with the aim of softening
# nonresponse bias:
sbscal.nr

# Now you may want to calibrate again in order to reduce
# estimators variance, by using further available auxiliary
# information, e.g. the total number of employees (emp.num)
# and enterprises (ent) inside the domains obtained
# by crossing nace.macro and region:

  # Build the second step population totals template:
  pop2<-pop.template(sbscal.nr,
        calmodel=~emp.num+ent-1,
        partition=~nace.macro:region)

  # Use the fill.template function to (i) automatically compute
  # the totals from the universe (sbs.frame) and (ii) safely fill
  # the template:
  pop2<-fill.template(universe=sbs.frame,template=pop2)

  # Now perform the second calibration step:
      # Get a hint on the calibration bounds:
      hint2<-bounds.hint(sbscal.nr,pop2)
      sbscal.nr2<-e.calibrate(design=sbscal.nr,df.population=pop2,
                  bounds=hint2)

# Notice that printing sbscal.nr2 you immediately understand
# that it is a "twice-calibrated" object:
sbscal.nr2
```

```
# Notice also that, even if the second calibration step causes
# sbscal.nr2 to be no more exactly calibrated with respect to
# emp.cl (look at the cvpct values)...
old.op <- options("RG.lonely.psu"="adjust")   # (prevent lonely-PSUs troubles)
svystatTM(design=sbscal.nr2,y=~emp.cl,vartype="cvpct")

# ...the nonresponse bias has not been resurrected (i.e. it gets stuck
# to its previous 7%):
mean.VA.cal.nr2<-svystatTM(design=sbscal.nr2,y=~va.imp2,estimator="Mean",
                 vartype= "cvpct",conf.int=TRUE,conf.lev=0.95)

options(old.op)   # (reset variance estimation options)

mean.VA.cal.nr2
100*(coef(mean.VA.cal.nr2)-MEAN.VA)/MEAN.VA


############################################################
# Provided the auxiliary variables are chosen in a smart way #
# a single calibration step can simultaneously succeed in:   #
# (i)  softening nonresponse bias;                           #
# (ii) reducing estimators variance.                         #
############################################################

# Let's come back to the original design with nonresponse:
sbsdes.nr

# Now, let's try to calibrate simultaneously on (see examples above):
# (i)  the population distribution of emp.cl;
# (ii) the total number of employees (emp.num) and enterprises (ent)
#      inside the domains obtained by crossing nace.macro and region:

  # Build the population totals template (notice that we are now forced
  # to a global calibration, as we are assuming to ignore emp.cl counts
  # inside domains obtained by crossing nace.macro and region):
  pop1<-pop.template(sbs.nr,
        calmodel=~emp.cl+(emp.num+ent):nace.macro:region-1)

  # Use the fill.template function to (i) automatically compute
  # the totals from the universe (sbs.frame) and (ii) safely fill
  # the template:
  pop1<-fill.template(universe=sbs.frame,template=pop1)

  # Now perform the single calibration step:
      # Get a hint on the calibration bounds:
      hint1<-bounds.hint(sbsdes.nr,pop1)
      sbscal.nr1<-e.calibrate(design=sbsdes.nr,df.population=pop1,
                  bounds=hint1)

  sbscal.nr1

# Now:
# (i) verify the nonresponse bias reduction effect:
  old.op <- options("RG.lonely.psu"="adjust")  #(prevent lonely-PSUs troubles)
  mean.VA.cal.nr1<-svystatTM(design=sbscal.nr1,y=~va.imp2,estimator="Mean",
                   vartype= "cvpct",conf.int=TRUE,conf.lev=0.95)
```

```
  options(old.op)

  mean.VA.cal.nr1
  100*(coef(mean.VA.cal.nr1)-MEAN.VA)/MEAN.VA

# thus we are back to ~7%, as for the previous 2-step calibration example.

# (ii) compare cvpct with the previous 2-step calibration example:
mean.VA.cal.nr1
mean.VA.cal.nr2

# hence, both bias reduction and efficiency are almost the same in 2-step and
# single step calibration (auxiliary information being equal): the choice
# will often depend on practical considerations (e.g. convergence, computation
# time).


##############################################################################
# Example with heteroskedastic 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...
TOT.emp.num <- pop[1, 1]
VA.ratio<-svystatL(design=sbsdes, expression(TOT.emp.num * (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 property is exactly satisfied by our numerical results, see below:
pop[1, 1]/coef(svystatTM(sbsdes, ~emp.num))
g.range(sbscal)

# ...as it must be.


#############################################################################
# A second example of calibration with heteroskedastic 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.


#############################################################################
# A third example. Shows how to exploit the sigma2 argument to prevent  #
# some initial weights from being altered by calibration.               #
#############################################################################

# Let's refer again to object sbsdes:
```

```
sbsdes

# Let's assume that for some reason we want to prevent the *highest* initial
# weights from being altered by calibration:
dmax <- max(weights(sbsdes))
dmax

# The relevent units are the following 4:
to.keep <- which(weights(sbsdes) == dmax)
to.keep

# Now, let's prepare a convenience variable (to be later bound to the 'sigma2'
# argument of e.calibrate) whose values are set to a very high value (say 1E12)
# for those units whose initial weight must be preserved, and to 1 otherwise.
# For definiteness, let's call such variable 'fixed':
sbsdes <- des.addvars(sbsdes, fixed = ifelse(weights(sbsdes) == dmax, 1E12, 1))

# Now, let's 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 region and emp.cl;
# ii) crossing region and nace.macro;

# Build the population totals template:
pop<-pop.template(sbsdes,
     calmodel = ~(emp.num + ent):(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", sigma2 = NULL)
       g.range(sbscal1)

       ## As expected, calibration weights of the 4 units to.keep *differ* from
       ## the corresponding initial weights:
       weights(sbsdes)[to.keep]
       weights(sbscal1)[to.keep]

  # 2) Then, *with* heteroskedasticy effect given by our convenience variable
  #    'fixed':
       sbscal2 <- e.calibrate(sbsdes, pop, calfun = "linear", sigma2 = ~fixed)
       g.range(sbscal2)

       ## Let's verify that calibration weights of the 4 units to.keep are now
       ## *equal* to the corresponding initial weights:
       weights(sbsdes)[to.keep]
       weights(sbscal2)[to.keep]

       ## ...as it must be.

# NOTE: It should be clear that the additional request to held some weights
#       fixed while calibrating will - all other things being equal - increase
```

```
#       the probability of non-convergence of the calibration algorithm.


############################################################
# Calibrating simultaneously on unit-level and cluster-level #
# auxiliary information: an household survey example.        #
############################################################

# Load household data:
data(data.examples)

# Define the survey design:
exdes<-e.svydesign(data=example,ids=~towcod+famcod,strata=~stratum,
       weights=~weight)

# Collapse strata to eliminate lonely PSUs:
exdes<-collapse.strata(design=exdes,block.vars=~sr:procod)

# Now add new convenience variables to the design object:
  ## 'houdensity': to estimate households counts
  ## 'ones':       to estimate individuals counts
exdes<-des.addvars(exdes,
                   houdensity=ave(famcod,famcod,FUN = function(x) 1/length(x)),
                   ones=1)

# Let's see how it's possible to calibrate *simultaneously* on:
  # 1. the number of *individuals* crossclassified by sex, 5 age classes,
  #    and province;
  # 2. the number of *households* by region.

# First, for the purpose of running the example, let's generate some
# artificial population totals. We have only to get HT estimates for
# the auxiliary variables and perturb them randomly:
  # Get HT estimates of auxiliary variables:
  xx<-aux.estimates(design=exdes,calmodel=~houdensity+sex:age5c:procod-1,
                    partition=~regcod)

  # Add a random uniform perturbation to these numbers:
  set.seed(12345)   # Fix the RNG seed for reproducibility
  xx[,-1]<-round(xx[,-1]*runif(prod(dim(xx[,-1])),0.8,1.2))

# Now we have at hand our artificial population totals, and
# we can proceed with the calibration task:
excal<-e.calibrate(design=exdes,df.population=xx,calfun= "linear",
                   bounds=c(0,3),aggregate.stage=2)

# To perceive the effect of calibration, let's e.g. compare the HT and
# calibrated estimates of the average number of individuals per household
# at population level:
svystatR(exdes,~ones,~houdensity,vartype="cvpct")
svystatR(excal,~ones,~houdensity,vartype="cvpct")


########################################
# Calibrating on different patterns of #
# "incomplete" auxiliary information.  #
########################################
```

```
# Usually calibration constraints involve "complete auxiliary information",
# i.e. totals which are known either:
#     (i)  for the target population as a whole (e.g. total number of
#          employees working in italian active enterprises at a given date);
# or:
#     (ii) for each subpopulation belonging to a complete partition of
#          the target population (e.g. number of male and female people
#          residing in Italy at a given date).
#
# Anyway, it may happen sometimes that the available auxiliary information
# is actually "incomplete", i.e. one doesn't know all the totals for all the
# subpopulations in a partition, but rather only for some of them. As an
# example, suppose marital status has categories "married", "unmarried",
# and "widowed" and that one only knows the number of "unmarried" people.
#
# In what follows I show how you can use ReGenesees to handle a calibration
# task on "incomplete" auxiliary information.

  ######################
  # A simple example. #
  ######################

  # Load household data:
  data(data.examples)

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

  # Suppose you only know the number of "unmarried" people (let's say 398240)
  # but you ignore "married" and "widowed" totals, and you want to calibrate
  # on this incomplete information.

  # First, add to the survey design a new numeric variable with value 1
  # for unmarried people and 0 otherwise:
  des<-des.addvars(des,unmarried=as.numeric(marstat=="unmarried"))

  # Second, prepare a template to store the known "unmarried" people count:
  pop<-pop.template(des,calmodel=~unmarried-1)

  # Third, fill the template with the known total:
  pop[1,1]<-398240

  # Fourth, calibrate:
  descal<-e.calibrate(des,pop)

  # Now test that only "unmarried" estimated total has 0 percent CV:
  Zapsmall(svystatTM(descal,~marstat,vartype="cvpct"))

  # ...as it must be.


  ###############################
  # A more complicated example. #
  ###############################
```

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

# Define the survey design:
sbsdes<-e.svydesign(data=sbs,ids=~id,strata=~strata,weights=~weight,fpc=~fpc)

# Suppose you want to calibrate on the following "incomplete" known totals:
  # 1. enterprises counts by nace.macro
  # 2. enterprises counts by dom3 ONLY inside nace.macro 'Industry'
  # 3. total of y by emp.cl ONLY inside nace.macro 'Commerce'

# First, add to the survey design new variables identifying the domains
# where "incomplete" totals 2. and 3. are known:
## 2. -> nace.macro = 'Industry'
sbsdes<-des.addvars(sbsdes,Industry=as.numeric(nace.macro=="Industry"))
## 3. -> nace.macro = 'Commerce'
sbsdes<-des.addvars(sbsdes,Commerce=as.numeric(nace.macro=="Commerce"))

# Do the same for the sampling frame:
## 2. -> nace.macro = 'Industry'
sbs.frame$Industry=as.numeric(sbs.frame$nace.macro=="Industry")
## 3. -> nace.macro = 'Commerce'
sbs.frame$Commerce=as.numeric(sbs.frame$nace.macro=="Commerce")

# Second, prepare a template to store the totals listed in 1., 2. and 3.;
# to this purpose one can e.g. compute HT estimates of the involved auxiliary
# variables:
Xht<-aux.estimates(design=sbsdes,
                   calmodel=~nace.macro+Industry:dom3+Commerce:y:emp.cl-1)
Xht

# Third, use the structure above to compute actual population totals
# from the sampling frame:
pop <- fill.template(universe=sbs.frame,template=Xht)
pop

# Fourth, calibrate:
sbscal <- e.calibrate(design=sbsdes,df.population=pop)

# Test1: nace.macro counts have 0 CVs:
test1<-svystatTM(design=sbscal,y=~nace.macro,vartype="cvpct")
test1

# Test2: only 'Industry' macrosector has 0 CVs for dom3 counts:
test2<-svystatTM(design=sbscal,y=~dom3,by=~nace.macro,vartype="cvpct")
Zapsmall(test2)

# Test3: only 'Commerce' macrosector has 0 CVs for y total by emp.cl:
test3<-svystatTM(design=sbscal,y=~y,by=~emp.cl:nace.macro,vartype="cvpct")
Zapsmall(test3)
```

`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)

## S3 method for class 'analytic'
summary(object, ...)
```

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`	Triggers an approximate variance estimation method for multistage designs (see 'Details'). If not `NULL` (the default), must be a formula identifying self-representing strata (SR), if any.
`check.data`	Check out the correct nesting of `data` clusters? The default is `TRUE`.
`object`	An object of class `analytic`, as returned by `e.svydesign`.
`...`	Arguments for future extensions.

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` must not contain any missing value (`NA`). Should empty levels be present in any factor variable belonging to `data`, they would be dropped.

The `ids` argument specifies the cluster identifiers. It is possible to specify a multistage 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`. Direct weights must be strictly positive.

The fpc formula serves the purpose of specifying the finite population corrections at subsequent sampling stages. By default fpc=NULL, which implies with-replacement sampling.
If the survey has only one stage, then the fpcs 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 must be expressed in terms of sampling units (be they elementary 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. Thus, passing to fpc a column of zeros, means again with-replacement sampling.
For multistage sampling the population size (or the sampling fraction) 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 population sizes (or sampling fractions) in each stratum for the first stage units, while variable fpc.SSU gives population sizes (or sampling fractions) for the second stage units in each sampled PSU. Notice that if you choose to pass to fpc population totals (rather than sampling rates) at a given stage, then you must do the same for all stages (and vice versa).
If fpc is specified but for fewer stages than ids, sampling is assumed to be *complete* for subsequent stages. The function will check that fpcs values at each sampling stage do not vary within strata.

When dealing with a two-stage (multistage) stratified sampling design that includes *self-representing (SR) strata* (i.e. strata containing only PSUs selected with probability 1), the only (leading) contribution to the variance of 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 fpcs have been properly specified. In particular, if fpc=~fpc.PSU + fpc.SSU and one specifies fpcs in terms of sampling fractions, then, inside SR strata, fpc.PSU must be always equal to one. 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 software) 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 an approximate variance estimation method 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. The summary method provides further details. Objects of class analytic persistently store input survey data inside their variables component. Weights can be accessed by using the weights function.

PPS Sampling Designs

Probability proportional to size sampling *with replacement* does not pose any problem: one must simply specify fpc=NULL and pass the right weights. This holds also for multistage designs, where PSUs are selected with replacement with PPS inside strata. Moreover, when the PSUs are sampled

with replacement, the only contribution to the variance arises from the estimated PSU totals, and one can simply ignore any available information about subsequent sampling stages.

For unequal probability sampling *without replacement*, on the contrary, in order to get correct variance estimates, one should know the second-order inclusion probabilities under the sampling design at hand. Unluckily, these probabilities cannot generally be computed, thus one has to resort to some viable approximation. The easier one rests on pretending that PSUs were sampled with replacement, even if this is not actually the case. It is worth stressing that this approach will result in conservative estimates. Moreover, the variance overestimation is expected to be negligible as long as the actual sampling fractions of PSUs are close to zero. Notice that this "with replacement" approximation can be achieved by either not specifying `fpc`, or by passing to the PSUs term of `fpc` a column of zeros.

Note

The `analytic` class is a specialization of the `survey.design2` class from the **survey** package [Lumley 06]; 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"*, https://CRAN.R-project.org/package=survey.

See Also

`svystatTM`, `svystatR`, `svystatS`, `svystatSR`, `svystatB`, `svystatQ`, `svystatL` for calculating estimates and standard errors, `e.calibrate` for calibrating weights, `ReGenesees.options` for setting/changing variance estimation options, `collapse.strata` for the suggested way of handling lonely PSUs, `weights` to extract weights.

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
```

```
# Use the summary() function if you need some additional details, e.g.:
summary(des)

# Use the 'variables' slot to extract survey data, e.g.:
head(des$variables)

# Use the weights() function to extract weights, e.g.:
summary(weights(des))

# Again the same design, but 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 SSUs in SR strata and PSUs 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
```

`estimator.kind` *Which Estimator Did Generate these Survey Statistics?*

Description

Identifies what kind of estimator has been used to compute a (set of) survey static(s).

Usage

```
estimator.kind(stat, design)
```

Arguments

`stat`	An object containing survey statistics.
`design`	Object of class `analytic` (or inheriting from it) containing survey data and sampling design metadata.

Details

Given a survey statistic object `stat` and a survey design object `design` from which `stat` is supposed to have been derived, this function returns the "precise kind" of the corresponding estimator, as a textual description.

Argument `stat` can be any object which have been returned by calling a survey statistics function (e.g. `svystatTM`, `svystatR`, `svystatS`, `svystatSR`, `svystatB`, `svystatQ`, `svystatL`) on survey design object `design`. It can also be a collection of survey statistics as generated by protean function `svystat`, provided that function is invoked with `forGVF = FALSE`.

Should `stat` be a survey statistic derived from a design object *other than* `design`, the function would raise an error.

Note that function `estimator.kind` is smart enough to recognize that estimates of totals/means of *dummy variables* are actually estimates of absolute/relative frequencies, despite such variables are of class `numeric` (see Section 'Examples').

Value

A `character` string describing the estimator kind.
Currently, possible return values (i.e. estimator kinds) are the following:

```
 (1)  'Total'
 (2)  'Absolute Frequency'
 (3)  'Mix of Totals and Absolute Frequencies'
 (4)  'Mean'
 (5)  'Relative Frequency'
 (6)  'Mix of Means and Relative Frequencies'
 (7)  'Ratio'
 (8)  'Share'
 (9)  'Share Ratio'
(10)  'Regression Coefficient'
(11)  'Quantile'
(12)  'Complex Estimator'
```

Author(s)

Diego Zardetto

See Also

gvf.input and svystat to prepare the input for GVF model fitting, fit.gvf to fit GVF models, and GVF.db to manage **ReGenesees** archive of registered GVF models.

Examples

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

# Compute some statistics and ask the corresponding estimator kind:
stat<-svystatTM(des,~emp.num)
stat
estimator.kind(stat,des)

stat<-svystatTM(des,~emp.num,estimator="Mean")
stat
estimator.kind(stat,des)

stat<-svystatTM(des,~emp.num+emp.cl)
stat
estimator.kind(stat,des)

stat<-svystatR(des,num=~va.imp2,den=~emp.num,by=~region)
stat
estimator.kind(stat,des)

stat<-svystatQ(des,y=~va.imp2,ties="rounded")
stat
estimator.kind(stat,des)

# Using protean function svystat to get many statistics in a single shot:
  ## ungrouped summary statistics:
stat<-svystat(des,kind="R",num=~va.imp2,den=~emp.num,by=~emp.cl:nace.macro,
      combo=2,forGVF=FALSE)
stat
estimator.kind(stat,des)

 ## grouped summary statistics:
stat<-svystat(des,kind="R",num=~va.imp2,den=~emp.num,by=~emp.cl:nace.macro,
      group=~region,forGVF=FALSE)
stat
estimator.kind(stat,des)


# Behaviour with dummy variables:
 ## 1. convenience variable 'ent' (whose values are always 1, so that its
  #    estimated total actually estimates haw many enterprises are there in the
  #    target population)
  class(des$variables$ent)
  range(des$variables$ent)
```

```
  # The estimated total is correctly recognized as a count
  stat<-svystatTM(des,~ent)
  stat
  estimator.kind(stat,des)

## 2. an actual dummy variable (built on the fly) which indicates if the
 #    enterprise has more than 29 employess or not:
 des<-des.addvars(des,emp.gt.29=as.numeric(emp.num > 29))
 class(des$variables$emp.gt.29)
 range(des$variables$emp.gt.29)
  # The estimated total is correctly recognized as an absolute frequency
  stat<-svystatTM(des,~emp.gt.29)
  stat
  estimator.kind(stat,des)
  # The estimated mean is correctly recognized as a relative frequency
  stat<-svystatTM(des,~emp.gt.29,estimator="Mean")
  stat
  estimator.kind(stat,des)
```

`ext.calibrated` *Make ReGenesees Digest Externally Calibrated Weights*

Description

Enables **ReGenesees** to provide correct variance estimates of (functions of) calibration estimators, even if the survey weights have not been calibrated by **ReGenesees**.

Usage

```
ext.calibrated(data, ids, strata = NULL, weights,
               fpc = NULL, self.rep.str = NULL, check.data = TRUE,
               weights.cal, calmodel, partition = FALSE, sigma2 = NULL)
```

Arguments

`data`	The same as in function `e.svydesign`.
`ids`	The same as in function `e.svydesign`.
`strata`	The same as in function `e.svydesign`.
`weights`	The same as in function `e.svydesign`.
`fpc`	The same as in function `e.svydesign`.
`self.rep.str`	The same as in function `e.svydesign`.
`check.data`	The same as in function `e.svydesign`.
`weights.cal`	Formula identifying the externally calibrated weights.
`calmodel`	The same as in function `e.calibrate`.
`partition`	The same as in function `e.calibrate`.
`sigma2`	The same as in function `e.calibrate`.

Details

Owing to **ReGenesees**'s ability to provide proper variance estimates for (complex functions of) calibration estimators, some users may be tempted to exploit **ReGenesees** in the estimation phase *even if* they did *not* use **ReGenesees** for calibration.

This result *cannot* be achieved naively, by simply passing to **ReGenesees** function `e.svydesign` the survey data and supplying the externally calibrated weights through its `weights` argument.

Indeed, variance estimation methods of **ReGenesees**'s summary statistics functions (`svystatTM`, `svystatR`, `svystatS`, `svystatSR`, `svystatB`, `svystatQ`, `svystatL` and `svystat`) are dispatched according to the class of the input `design` object:

1. If the design object is un-calibrated (i.e. its class is 'analytic'), variance formulas are appropriate to Horvitz-Thompson estimators (and functions of them).
2. If the design object is calibrated (i.e. its class is 'cal.analytic'), variance formulas are appropriate to Calibration estimators (and functions of them).

Therefore, the naive approach of passing the externally calibrated weights `weights.cal` to `e.svydesign` as if they were initial or design weights cannot succeed, since it would result in HT-like variance estimates, leading generally to *variance overestimation* (with bigger upward bias for variables that are better explained by the calibration model).

Function `ext.calibrated` has been designed exactly to avoid the aforementioned pitfalls and to allow **ReGenesees** provide correct variance estimates of (functions of) calibration estimators, even if the survey weights have been calibrated externally by other software.

Argument `weights.cal` identifies the externally calibrated weights of the units included in the sample. The `data` variable referenced by `weights.cal` must be numeric. Currently, only *positive* externally calibrated weights can be handled (see the dedicated section below).

Other arguments to `ext.calibrated` derive either from function `e.svydesign` or from function `e.calibrate`. The former serve the purpose of passing the survey data and the corresponding sampling design metadata, the latter are meant to tell `ext.calibrated` how the externally calibrated weights have been obtained.

Value

An object of class `cal.analytic`, storing the original survey data *plus* all the sampling design and calibration metadata needed for proper variance estimation.

What if externally calibrated weights happen to be negative?

From a methodological perspective, negative calibration weights are legitimate. However, owing to software implementation details whose modification would not be trivial, function `ext.calibrated` is *not* yet able to cope with this case. Note that the problem is actually due to the *external* origin of the negative calibration weights. In fact, **ReGenesees** calibration and estimation facilities are entirely able to cope with possibly negative calibration weights, provided they were computed *internally*.

Note

Exactly as **ReGenesees**'s base functions `e.svydesign` and `e.calibrate` would do, `ext.calibrated` too will wrap inside its return value a local copy of `data`. As usual, this copy will be stored inside the `variables` slot of the output list. As usual, again, the calibrated weights will be accessible by using the `weights` function.

Author(s)

Diego Zardetto.

See Also

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

Examples

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

#########################################################################
# Simulate an external calibration procedure and compute some benchmark #
# estimates and errors to test function ext.calibrated                  #
#########################################################################
# Define a survey design
sbsdes <- e.svydesign(data= sbs, ids= ~id, strata= ~strata, weights= ~weight,
                      fpc= ~fpc)

# Build a template for population totals
pop <- pop.template(data= sbsdes, calmodel= ~y:nace.macro + emp.cl + emp.num - 1,
                    partition= ~dom3)

# Have a look at the template structure
pop.desc(pop)

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

# Calibrate
sbscal <- e.calibrate(design= sbsdes, df.population= pop, calfun= "logit",
                      bounds= c(0.8, 1.3), sigma2= ~ emp.num)

# Compute benchmark estimates and errors (average value added per employee by
# region) to be later compared with those obtained by using ext.calibrated
benchmark <- svystatR(design= sbscal, num= ~va.imp2, den= ~emp.num, by= ~region)
benchmark

# Extract the 'externally' calibrated weights...
w <- weights(sbscal)

#...and add these 'externally' calibrated weights to the original survey data
sbs.ext <- data.frame(sbs, w.ext = w)

# NOTE: Now sbs.ext is just a data frame, without any knowledge of the
#       calibration metadata formerly stored inside sbscal (i.e. the object
#       calibrated by ReGenesees)


#############################################################
# Let ReGenesees digest the 'externally' calibrated weights, #
# then re-compute benchmark estimates and errors for testing #
#############################################################
# Simply pass survey data along with sampling design and calibration model
```

```
# metadata
sbscal.ext <- ext.calibrated(data= sbs.ext, ids= ~id, strata= ~strata,
                             weights= ~weight, fpc = ~fpc,
                             weights.cal= ~w.ext,
                             calmodel= ~y:nace.macro + emp.cl + emp.num - 1,
                             partition= ~dom3, sigma2= ~emp.num)

# Have a look at the output
sbscal.ext

# Now re-compute benchmark estimates and errors by means of new object
# ext.sbscal
test <- svystatR(design= sbscal.ext, num= ~va.imp2, den= ~emp.num, by= ~region)
test

#####################################################################
# Compare benchmark estimates and errors to those derived from #
# ext.calibrated return object                                      #
#####################################################################
benchmark
test

# ...and they are identical, as it must be.

# NOTE: All utility tools yield exactly the same results, e.g.
identical(weights(sbscal), weights(sbscal.ext))
identical(g.range(sbscal), g.range(sbscal.ext))


#############################################################################
# Show that the naive idea of directly passing the externally calibrated #
# weights to e.svydesign does NOT work properly for variance estimation  #
#############################################################################
naive <- e.svydesign(data= sbs.ext, ids= ~id, strata= ~strata,
                     weights= ~w.ext, fpc = ~fpc)

# Estimated sampling errors derived by this naive design object...
svystatR(design= naive, num= ~va.imp2, den= ~emp.num, by= ~region)

#...do NOT match benchmark values, overestimating them:
benchmark
```

`extractors` *Extractor Functions for Variability 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`, `svystatS`, `svystatSR`, `svystatB`, `svystatQ`, `svystatL`, `svystat`, ...).

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 whatever the call that generated the object. For getting the design effect, object must have been built with option deff = TRUE.

Value

A data structure (typically inheriting from classes matrix or data.frame) storing the requested information.

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, Shares svystatS, Ratios between Shares svystatSR, Multiple Regression Coefficients svystatB, Quantiles svystatQ, Complex Analytic Functions of Totals and/or Means svystatL, and all of the above svystat.

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 corresponding 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 automatically fill the template. Stated more explicitly: the user himself has to bear the responsibility of putting the *right values* in the *right slots* of the prepared `template` data frame. To this end, function `pop.desc` could be very helpful.

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 corresponding 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. Should empty levels be present in any factor variable belonging to `universe`, they would be dropped.

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 hundreds of auxiliary variables and refer to target populations of several million 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 implicitly 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` switches 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, `pop.desc` to provide a natural language description of the template structure, 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)


############################
# A simple example first. #
############################

# Suppose you want to calibrate on the enterprise counts inside areas
  # 1) Build the population totals template:
pop<-pop.template(sbsdes, calmodel=~area-1)

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

 # ...the number of known totals to be stored is 24 (one for each area).

 # 2) Use the fill.template function to (i) automatically compute
 #    such 24 totals from the universe (sbs.frame) and (ii) safely fill
 #    the template:
pop<-fill.template(universe=sbs.frame,template=pop)
pop

 # 3) 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))


########################################
# A more involved (two-sided) example. #
########################################

# 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 factorized 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 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)

 # Note: out of the 792 known totals in pop, only non-zero entries are actually
 # relevant

 # 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 (factorized 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 argument 'mem.frac'.                        #
############################################################
## Not run:
 # First artificially increase the size of the sampling frame (e.g.
 # up to 5 million 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)
```

`find.lon.strata` *Find Strata with Lonely PSUs*

Description

Checks whether a stratified design object contains lonely PSUs: if this is the case, returns the lonely strata levels.

Usage

```
find.lon.strata(design)
```

Arguments

`design`	Object of class `analytic` (or inheriting from it) containing survey data and sampling design metadata.

Details

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 provided in function `collapse.strata` (for further alternatives, see also `ReGenesees.options`).

Function `find.lon.strata` (originally a private function intended to be called only by collapse.strata) is a simple diagnostic tool whose purpose is to identify the levels of the strata containing lonely PSUs (lonely strata for short).

Value

The lonely strata levels, if `design` actually contains lonely PSUs; `invisible(NULL)` otherwise.

Author(s)

Diego Zardetto

See Also

`collapse.strata` for the suggested way of handling lonely PSUs, `ReGenesees.options` for a different way to face the same problem (namely by setting variance estimation options), and `fpcdat` for useful data examples.

Examples

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

# A negative example first:

  # Build a design object:
    fpcdes<-e.svydesign(data=fpcdat,ids=~psu+ssu,strata=~stratum,weights=~w,
            fpc=~fpc1+fpc2)
```

```
    fpcdes

  # Find lonely strata:
    find.lon.strata(fpcdes)


# Recall that the difference between certainty PSUs (those sampled with
# probability 1, contained inside self-representing strata) and lonely PSUs
# rests on the fpc information passed to e.svydesign, e.g.:

  # Build a new design object with the same data, now IGNORING fpcs:
    fpcdes.nofpc<-e.svydesign(data=fpcdat,ids=~psu+ssu,strata=~stratum,
                  weights=~w)
    fpcdes.nofpc

  # Find lonely strata:
    find.lon.strata(fpcdes.nofpc)


# A trivial check: collapsing strata eliminates lonely PSUs

  # Apply the collapse strata technique:
    fpcdes.nofpc.clps<-collapse.strata(fpcdes.nofpc)
    fpcdes.nofpc.clps
    clps.strata.status

  # Find lonely strata:
    find.lon.strata(fpcdes.nofpc.clps)

# ...as it must be.
```

`fit.gvf` *Fit GVF Models*

Description

This function fits one or more GVF models to a set of survey statistics.

Usage

```
fit.gvf(gvf.input, model = NULL, weights = NULL)

## S3 method for class 'gvf.fit'
print(x, digits = max(3L, getOption("digits") - 3L), ...)
## S3 method for class 'gvf.fits'
print(x, digits = max(3L, getOption("digits") - 3L), ...)
## S3 method for class 'gvf.fit.gr'
print(x, digits = max(3L, getOption("digits") - 3L), ...)
## S3 method for class 'gvf.fits.gr'
print(x, digits = max(3L, getOption("digits") - 3L), ...)

## S3 method for class 'gvf.fits'
x[...]
```

```
## S3 method for class 'gvf.fits'
x[[...]]

## S3 method for class 'gvf.fit'
summary(object, correlation = FALSE, symbolic.cor = FALSE, ...)
## S3 method for class 'gvf.fits'
summary(object, correlation = FALSE, symbolic.cor = FALSE, ...)
## S3 method for class 'gvf.fit.gr'
summary(object, correlation = FALSE, symbolic.cor = FALSE, ...)
## S3 method for class 'gvf.fits.gr'
summary(object, correlation = FALSE, symbolic.cor = FALSE, ...)
```

Arguments

gvf.input	An object of class gvf.input (or gvf.input.gr), containing the data to fit.
model	The GVF model(s) to be fitted (see 'Details'). NULL (the default) requires to fit *all* the registered GVF models currently available in GVF.db.
weights	Formula specifying the weights to be used for fitting (via weighted least squares), if any. NULL (the default) means that ordinary least squares will be used. See also 'Details'.
x	An object of class gvf.fits, storing fitted GVF models.
digits	Minimal number of significant digits, see print.default.
object	Any output of fit.gvf, storing one (more than one) fitted GVF model(s).
correlation	Should the correlation matrix of the estimated parameters be returned and printed? Logical, with default FALSE.
symbolic.cor	Should the correlations be printed in symbolic form (see symnum) rather than as numbers. Logical, with default FALSE.
...	Further arguments passed to or from other methods.

Details

Function fit.gvf fits one or more GVF models to a set of survey statistics. The rationale for fitting multiple models to the same data is primarily for comparison purposes: the user is expected to eventually choose his preferred model, in order to obtain sampling errors predictions.

Argument gvf.input specifies the set of (pre computed) estimates and errors to which GVF models are to be fitted, as prepared by functions gvf.input and/or svystat.

One or more GVF models can be fitted *simultaneously* to the same data, depending on the way argument model is passed.

Argument model can be either:

(1) NULL (the default) meaning *all* the registered models currently available in GVF.db;

(2) any sub-vector of GVF.db$Model.id, i.e. an integer vector identifying an arbitrary selection of registered models;

(3) an arbitrary (single) formula, i.e. any custom, user-defined GVF model.

When model is passed via options (1) or (2), function fit.gvf can take advantage of any additional information available inside GVF.db, e.g. to warn the user in case a GVF model is not deemed to be appropriate for the kind of estimates contained into gvf.input (see 'Examples').

Argument weights enables fitting the specified GVF model(s) via weighted least squares. By default weights = NULL and ordinary least squares are used. The weights must be passed by a formula referencing variables belonging to gvf.input. For instance, to weight observations according to reciprocals of squared CVs, one can use weights = ~I(CV^-2).

Value

An object containing one or more fitted GVF models, depending on the way argument model was passed.

Let's first focus on input objects of class gvf.input.
If model specifies a single GVF model, the output object will be of class gvf.fit and inherit from class lm.
If model specifies many GVF models, the output object will be of class gvf.fits and inherit from class list. Hence, it will be possible to subset gvf.fits objects via methods [and [[. Note, moreover, that each component (in the sense of class list) of a gvf.fits object will be of class gvf.fit.

When, instead, the input object has class gvf.input.gr, i.e. it stores *"grouped"* estimates and errors, model fitting is performed *separately* for different groups. Therefore, applying fit.gvf *always* results in *many* fitted GVF models.
If model specifies a single GVF model, the output object will be of class gvf.fit.gr and inherit from class list. Each slot of the list will contain the same GVF model fitted to a specific group.
If model specifies many GVF models, the output object will be of class gvf.fits.gr and again inherit from class list. Each slot of the list will now contain *a second list* storing different GVF models fitted to a specific group.

Author(s)

Diego Zardetto

See Also

estimator.kind to assess what kind of estimates are stored inside a survey statistic object, GVF.db to manage **ReGenesees** archive of registered GVF models, gvf.input and svystat to prepare the input for GVF model fitting, fit.gvf to fit GVF models, plot.gvf.fit to get diagnostic plots for fitted GVF models, drop.gvf.points to drop alleged outliers from a fitted GVF model and simultaneously refit it, and predictCV to predict CV values via fitted GVF models.

Examples

```
# Load example data:
data(AF.gvf)

# Now we have at our disposal a set of estimates and errors
# of Absolute Frequencies:
str(ee.AF)

# And the available registered GVF models are listed below:
GVF.db


##########################################
# How to specify the GVF model(s) to fit? #
##########################################
```

```
## (A) How to specify a *single* GVF model ##

#### (A.1) Select one registered model using its 'Model.id' as reported in
####       the GVF.db archive
# Let's fit, for instance, the GVF model with Model.id = 1:
m <- fit.gvf(ee.AF, model = 1)

# Inspect the result:
class(m)
m
summary(m)

# Now let's fit GVF model with Model.id = 4
m <- fit.gvf(ee.AF, model = 4)
# Beware of the NOTE reported when printing or summarizing this fitted model:
m
summary(m)


#### (A.2) Specify the GVF model to fit by providing its formula directly, e.g.
####       because it is not available in GVF.db (yet):
m <- fit.gvf(ee.AF, model = CV ~ I(1/Y^2) + I(1/Y) + Y + I(Y^2))
m
summary(m)


## (B) How to specify a *many* GVF models simultaneously ##

#### (B.1) Use a subset of column 'Model.id' of GVF.db
# Let's, for instance, fit all the available GVF models which are appropriate
# to Frequencies, as reported in column 'Estimator.kind' of GVF.db
mm <- fit.gvf(ee.AF, model = 1:3)

# Inspect the result:
class(mm)
length(mm)
mm
summary(mm)

# Note that you can subset the output fitted models as a list:
mm.31 <- mm[c(3,1)]
class(mm.31)
mm.31

# and:
mm.2 <- mm[[2]]
class(mm.2)
mm.2


#### (B.2) Not specifying any GVF model, or specifying model = NULL, causes
####       *all* the available models in GVF.db to be fitted simultaneously:
mm <- fit.gvf(ee.AF)

# Inspect the result:
class(mm)
```

```
length(mm)
mm
summary(mm)


#########################################################
# How to fit GVF model(s) via *weighted* least squares? #
#########################################################
# Weights can be specified by a formula. Of course, the 'weights' formula must
# reference variables belonging to gvf.input.

# Let's use the built-in GVF model with Model.id = 1 and weight observations
# according to reciprocals of squared CVs:
mw <- fit.gvf(ee.AF, model = 1, weights = ~I(CV^-2))
mw

# Compute ordinary least squares fit:
m <- fit.gvf(ee.AF, model = 1)
m

# Compare the results:
summary(mw)
summary(m)


#############################################################################
# Fitting GVF model(s) to "grouped" estimates and errors: a quick ride. #
#############################################################################
# Recall we have at our disposal the following survey design object
# defined on household data:
exdes

# Now use function svystat to prepare "grouped" estimates and errors
# of counts to be fitted separately (here groups are regions):
ee.g <- svystat(exdes, y=~ind, by=~age5c:marstat:sex, combo=3, group=~regcod)
class(ee.g)
ee.g

## Fit a *single* registered GVF model separately inside groups ##
m.g <- fit.gvf(ee.g, model = 1)

# Inspect the result:
class(m.g)
length(m.g)
m.g
summary(m.g)

# Can subset the result as a list, e.g. to get the fitted model of region '7':
m.g7 <- m.g[["7"]]
class(m.g7)
summary(m.g7)


## Fit *many* registered GVF models separately inside groups ##
mm.g <- fit.gvf(ee.g, model = 1:3)

# Inspect the result:
```

```
class(mm.g)
length(mm.g)
mm.g
summary(mm.g)

# Still can subset the result as a list, but now each component is a list
# itself. To get the fitted models of region '7', simply:
mm.g7 <- mm.g["7"]]
class(mm.g7)
summary(mm.g7)

# And to isolate GVF fitted model number 2 for region '7', simply:
mm.g7.2 <- mm.g7[[2]]
class(mm.g7.2)
summary(mm.g7.2)
```

`fpcdat` *A Small But Not Trivial Artificial Sample Data Set*

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)
head(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 (partitioned 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))
```

---

get.residuals *Calibration Residuals of Interest Variables*

Description

Computes (scaled) residuals of a set of interest variables w.r.t. the calibration model adopted to build a calibrated object.

Usage

```
get.residuals(cal.design, y, scale = c("no", "w", "d", "g"))
```

Arguments

cal.design	Object of class cal.analytic.
y	Formula defining the variables of interest.
scale	character specifying how to scale the residuals, can be one of: 'no' (the default), 'w', 'd', 'g' (see 'Details').

Details

This function has been designed mainly for programmers willing to build upon **ReGenesees**: typical users are not expected to feel much need of it.

The residuals of an interest variable w.r.t. the linear model defined by the auxiliary variables used for calibration play a central role in estimating the variance of Calibration Estimators. Notice that if object cal.design has been generated by running a *partitioned* calibration task (see e.calibrate), the residuals will be correctly computed using the different estimated regression coefficients pertaining to the different domains belonging to the partition.

The mandatory argument y behaves exactly the same way as it does in function svystatTM.

The scale argument allows to scale the computed residuals by multiplying them by different factors. By default scale="no", that is *unscaled* residuals are returned. Value "w" returns the residuals times the calibrated weights; value "d" returns the residuals times the initial weights; finally, value "g" returns the residuals times the *g-weights* (i.e. the ratios between calibrated and initial weights). Notice that the semantics of argument scale are slightly modified when the input object cal.design has been obtained by a multi-step calibration procedure (see Section 'Note' below).

Value

A matrix of residuals.

Note

If cal.design has undergone k subsequent calibration steps (with k >= 1), the function will return the residuals computed w.r.t. the linear assisting model underlying the *last* (i.e. k-th) calibration step. If k >= 2, the scale parameter will be interpreted as follows:

```
SCALE        MEANING
 "no"........no scale;
  "w"........last calibration weight (i.e. at step k);
  "d"........second to last calibration weight (i.e. at step k - 1);
  "g"........ratio between last and second to last calibration weights.
```

Author(s)

Diego Zardetto

See Also

weights to extract the weights from a design object, e.calibrate for calibrating weights and g.range to get the range of the *g-weights*.

Examples

```
#####################################################################
# Just some checks on the consistency of the numerical results   #
# obtained by ReGenesees with well known theoretical properties. #
#####################################################################

# Load sbs data:
data(sbs)

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

################
## Property 1 ##
################
## If the calibration model has (implicitly or explicitly) an intercept
## the weighted sum of residuals must be zero.

# Suppose you calibrate on enterprise counts inside areas, i.e. a calibration
# model WITH intercept (though implicitly):
# calmodel= ~ area - 1

# First, build and fill the known totals template:
pop<-pop.template(sbsdes, calmodel= ~ area - 1)
pop<-fill.template(pop, universe=sbs.frame)

# Then, calibrate:
sbscal<-e.calibrate(sbsdes, pop)

# Now, get the residuals of any variable (e.g. y and emp.num) scaled with the
# direct weights:
de <- get.residuals(sbscal, ~ y + emp.num, scale="d")

# Lastly, compute the column sums...
colSums(de)

#...which is (numerically) zero, as it must be.


################
## Property 2 ##
################
## If the calibration model does not have (implicitly or explicitly) an
## intercept term the weighted sum of residuals is generally different from zero.

# Suppose you calibrate on employees counts inside areas, i.e. a calibration
# model WITHOUT intercept:
# calmodel= ~ emp.num:area - 1

# First, build and fill the known totals template:
pop<-pop.template(sbsdes, calmodel= ~ emp.num:area - 1)
pop<-fill.template(pop, universe=sbs.frame)

# Then, calibrate:
sbscal<-e.calibrate(sbsdes, pop)
```

```
# Now, get the residuals of any variable (e.g. y and region) scaled with the
# direct weights:
de <- get.residuals(sbscal, ~ y + region, scale="d")

# Lastly, compute the column sums...
colSums(de)

#...which is far from zero, as expected.


################
## Property 3 ##
################
## In the Taylor linearization approach, estimating the variance of a
## Calibration Estimator amounts to estimating the variance of the HT total
## of its linearized variable (i.e. the g-scaled residual), under the sampling
## design at hand.

# Suppose you calibrate on the total number of employees and enterprises
# inside the domains obtained by:
#  i) crossing nace.macro and region;
# ii) crossing emp.cl and region;

# First, build and fill the population totals template:
pop<-pop.template(sbsdes,
     calmodel=~(emp.num+ent):(nace.macro+emp.cl)-1,
     partition=~region)
pop<-fill.template(universe=sbs.frame,template=pop)

# Then, calibrate:
sbscal<-e.calibrate(sbsdes,pop)

# Now, compute the linearized variable of the Calibration Estimator of the
# total of any variable (e.g. va.imp2):
z_va.imp2 <- get.residuals(sbscal, ~ va.imp2, scale="g")

# Now, treat z_va.imp2 as an ordinary variable and compute the standard error
# of its HT total:
sbsdes<-des.addvars(sbsdes, z_va.imp2 = z_va.imp2)
SE(svystatTM(sbsdes, ~z_va.imp2))

# Lastly, compute directly the standard error of the Calibration Estimator...
SE(svystatTM(sbscal, ~va.imp2))

#...and they are identical, as it must be.


# Obviously the same result would hold for domain estimates (e.g. the total
# of va.imp2 for the "Agriculture" nace.macro).

# Compute the linearized variable of the Calibration Estimator of the domain
# total:
z_va.imp2.Agr <- get.residuals(sbscal, ~ I(va.imp2*(nace.macro=="Agriculture")),
                               scale="g")

# Now, treat z_va.imp2.Agr as an ordinary variable and compute the standard
# error of its HT total:
```

```
sbsdes<-des.addvars(sbsdes, z_va.imp2.Agr = z_va.imp2.Agr)
SE(svystatTM(sbsdes, ~z_va.imp2.Agr))

# Lastly, compute directly the standard error of the Calibration Estimator
# by domain...
SE(svystatTM(sbscal, ~va.imp2, ~nace.macro))

#...and the "Agriculture" SEs are identical, as it must be.
```

`getBest` *Identify the Best Fit GVF Model*

Description

Given a set of competing fitted GVF models, this function selects the best model according to a given criterion.

Usage

```
getBest(object,
        criterion = c("R2", "adj.R2", "AIC", "BIC"), ...)
```

Arguments

`object`	Typically, an object containing *many* fitted GVF models (i.e. of class `gvf.fits` or `gvf.fits.gr`).
`criterion`	The quality criterion to be used for model selection. Default is `R^2`.
`...`	Further arguments passed to or from other methods.

Details

Given a set of competing fitted GVF models, this function selects the best model according to a given criterion.

Four goodness-of-fit criteria are available: `R^2`, `adjusted R^2`, `AIC`, and `BIC` (see `getR2`).

If `object` is a *set* of GVF models fitted to *grouped* data (i.e. of class `gvf.fits.gr`), the function will return the fitted GVF model with best *average score* in the given criterion *over the groups*.

Value

A single GVF fitted model.

Methodological Warning

Each one of the available criteria has its own specificities and limitations (e.g. it is senseless to use AIC to compare two GVF models with different response variables). It is up to the user to select the measure which is appropriate to his goals.

Author(s)

Diego Zardetto

See Also

GVF.db to manage **ReGenesees** archive of registered GVF models, gvf.input and svystat to prepare the input for GVF model fitting, fit.gvf to fit GVF models, plot.gvf.fit to get diagnostic plots for fitted GVF models, drop.gvf.points to drop alleged outliers from a fitted GVF model and simultaneously refit it, and predictCV to predict CV values via fitted GVF models.

Examples

```
# Load example data:
data(AF.gvf)

# Inspect available estimates and errors of counts:
str(ee.AF)

# List available registered GVF models:
GVF.db


## (A) A *a set* of GVF models fitted to the same data ##
# Fit example data to all registered GVF models:
mm <- fit.gvf(ee.AF)
summary(mm)

# Get the best model according to adjusted R^2:
mm.best <- getBest(mm, criterion = "adj.R2")
mm.best

# NOTE: The *first* model has been selected. A thorough model comparison
#       by means of diagnostic plots would have led to the same result:
plot(mm, 1:3)


## (B) a *set of* GVF models fitted to *grouped* data ##
# We have at our disposal the following survey design object on household data:
exdes

# Use function svystat to prepare *grouped* estimates and errors of counts
# to be fitted separately (here groups are regions):
ee.g <- svystat(exdes, y=~ind, by=~age5c:marstat:sex, combo=3, group=~regcod)
str(ee.g)

# Fit all registered GVF model number separately inside groups:
mm.g <- fit.gvf(ee.g)
summary(mm.g)

# Get the best model according to R^2:
mm.g.best <- getBest(mm.g)
mm.g.best

# NOTE: Again, the *first* model has been selected. A thorough model comparison
#       by means of diagnostic plots would have led to the same result:
plot(mm.g, 1:3)
```

getR2 *Quality Measures on Fitted GVF Models*

Description

These functions extract goodness-of-fit measures from fitted GVF models.

Usage

```
getR2(object, adjusted = FALSE, ...)

## S3 method for class 'gvf.fits'
AIC(object, ...)

## S3 method for class 'gvf.fits'
BIC(object, ...)
```

Arguments

`object`	An object containing one or more fitted GVF models.
`adjusted`	Should the *adjusted* `R^2` be computed? The default is `FALSE`
`...`	Further arguments passed to or from other methods.

Details

These functions compute three goodness-of-fit measures on fitted GVF models: `R^2`, AIC, and BIC. Such measures can help compare the relative quality of competing GVF models, hence facilitating model selection (see also function `getBest`).

Though `object` can also be a *single* fitted GVF model, these functions are principally meant to compare *different* GVF models fitted to the same data (i.e. the same estimates and errors).

To request the *adjusted* `R^2`, use function `getR2` and specify `adjusted = TRUE`.

Value

If `object` is a single GVF model (class `gvf.fit`), the requested quality measure.

If `object` is a set of GVF models fitted to the same data (class `gvf.fits`), a vector whose elements store the requested quality measure for each GVF model.

If `object` is a single GVF model fitted to "grouped" data (class `gvf.fit.gr`), a list whose components store the requested quality measure for the corresponding groups.

If `object` is a set of GVF models fitted to "grouped" data (class `gvf.fits.gr`), a list whose components store vectors whose elements report the requested quality measure for each GVF model of each group.

Methodological Warning

Each one of the provided quality measures has its own specificities and limitations (e.g. it is senseless to use AIC to compare two GVF models with different response variables). It is up to the user to select the measure which is appropriate to his goals.

Author(s)

Diego Zardetto

See Also

GVF.db to manage **ReGenesees** archive of registered GVF models, gvf.input and svystat to prepare the input for GVF model fitting, fit.gvf to fit GVF models, plot.gvf.fit to get diagnostic plots for fitted GVF models, drop.gvf.points to drop alleged outliers from a fitted GVF model and simultaneously refit it, and predictCV to predict CV values via fitted GVF models.

Examples

```
# Load example data:
data(AF.gvf)

# Inspect available estimates and errors of counts:
str(ee.AF)

# List available registered GVF models:
GVF.db

## (A) A *single* fitted GVF model ##
# Fit example data to registered GVF model number one:
m <- fit.gvf(ee.AF, 1)

# Compute some goodness-of-fit measures:
getR2(m)
AIC(m)


## (B) A *a set* of GVF models fitted to the same data ##
# Fit example data to all registered GVF models:
mm <- fit.gvf(ee.AF)

# Compute some goodness-of-fit measures:
getR2(mm, adjusted = TRUE)
BIC(mm)


## (C) a *single* GVF model fitted to *grouped* data ##
# We have at our disposal the following survey design object on household data:
exdes

# Use function svystat to prepare *grouped* estimates and errors of counts
# to be fitted separately (here groups are regions):
ee.g <- svystat(exdes, y=~ind, by=~age5c:marstat:sex, combo=3, group=~regcod)
str(ee.g)

# Fit registered GVF model number one separately inside groups:
m.g <- fit.gvf(ee.g, 1)

# Compute some goodness-of-fit measures:
getR2(mm)
AIC(mm)
```

```
## (D) a *set of* GVF models fitted to *grouped* data ##
# Fit all registered GVF model number separately inside groups:
mm.g <- fit.gvf(ee.g)

# Compute some goodness-of-fit measures:
getR2(mm.g, adjusted = TRUE)
BIC(mm.g)
```

GVF.db *Archive of Registered GVF Models*

Description

GVF.db is the archive of *registered* (i.e. built-in and/or user-defined) Generalized Variance Functions models supported by **ReGenesees**. Special accessor functions allow to customize, maintain, extend, update, save and reset such archive.

Usage

```
GVF.db

GVF.db$insert(GVF.model, Estimator.kind = NA, Resp.to.CV = NA, verbose = TRUE)

GVF.db$delete(Model.id, verbose = TRUE)

GVF.db$get(verbose = TRUE)

GVF.db$assign(value, verbose = TRUE)

GVF.db$reset(verbose = TRUE)
```

Arguments

GVF.model	A GVF model, expressed as a formula object or as a character string (see 'Details').
Estimator.kind	Character string identifying the kind of estimators for which the GVF model is deemed to be appropriate (see 'Details').
Resp.to.CV	Character string representing the function which maps the response of the GVF model (namely: variable 'resp') to the coefficient of variation (namely: variable 'CV'), see 'Details'.
Model.id	Unique integer key identifying the GVF model.
value	An exported copy of GVF.db, as returned by GVF.db$get().
verbose	Enables printing of a summary description of the result (the default is TRUE).

Format

Each row of the GVF.db data frame represents a registered GVF model, with relevant information on the following 4 variables:

Model.id A unique integer key identifying the GVF model, integer.

`GVF.model` A character string specifying the GVF model formula, `character`. See also 'Details'.

`Estimator.kind` A character string identifying the kind of estimators for which the GVF model is deemed to be appropriate, `character`. See also 'Details'.

`Resp.to.CV` A character string which represents the function mapping the response of the GVF model (namely: variable 'resp') to the coefficient of variation (namely: variable 'CV'), `character`. See also 'Details'.

Details

`GVF.db` stores information about Generalized Variance Functions models supported by **ReGenesees**. When starting a new work session with **ReGenesees**, `GVF.db` contains few built-in GVF models (currently 5, see sections 'Source' and 'Examples'). The content of `GVF.db` can be customized by means of special accessor functions:

```
ACCESSOR FUNCTION       PURPOSE
GVF.db$insert...........Register a new GVF model by adding a new row to the
                        GVF.db archive
GVF.db$delete...........Unregister a GVF model by deleting the corresponding
                        row from GVF.db
GVF.db$get..............Get the current version of GVF.db (e.g. to copy/save a
                        customized archive for later usage)
GVF.db$assign...........Overwrite the current version of GVF.db (e.g. to use a
                        customized archive which was exported in a previous
                        ReGenesees session)
GVF.db$reset............Reset GVF.db to its default version (i.e. the one with
                        built-in GVF models only)
```

Information about registered GVF models stored inside `GVF.db` will be accessed and used by **ReGenesees** Generalized Variance Functions facilities, e.g. functions `fit.gvf` or `predictCV`.

GVF.db$insert()

Function `GVF.db$insert` has just a single mandatory argument: `GVF.model`. This can be either a two-sided formula or a character string which would be transformed into a (well formed) two-sided formula by function `as.formula`.

The `GVF.model` formula to be inserted into `GVF.db` must be *new* (i.e. not already present into the archive) and can involve only variables contained inside `gvf.input` objects, namely:

```
(1)  'Y'
(2)  'SE'
(3)  'CV'
(4)  'VAR'
(5)  'DEFF'
```

Moreover, since GVF models are intended to model variances in terms of estimates, the response term of `GVF.model` must involve some of `'SE'`, `'CV'`, `'VAR'`, and the linear predictor must involve `'Y'`.

Optional argument `Estimator.kind` can be used to specify the kind of estimators for which the `GVF.model` is deemed to be appropriate. There are currently only 11 valid values for `Estimator.kind`, namely:

```
 (1)  'Total'
 (2)  'Mean'
 (3)  'Frequency'
 (4)  'Absolute Frequency'
 (5)  'Relative Frequency'
 (6)  'Ratio'
 (7)  'Share'
 (8)  'Share Ratio'
 (9)  'Regression Coefficient'
(10)  'Quantile'
(11)  'Complex Estimator'
```

Note that category `'Frequency'` has to be understood as an aggregation of categories `'Absolute Frequency'` and `'Relative Frequency'`, thus being appropriate for GVF models which are deemed to work well for estimators of *both* kind of frequencies.

One of the primary motivations for building and fitting a GVF model is to exploit the fitted model to *predict* the sampling error associated to a given estimate, instead of having to *compute* directly an estimate of such sampling error. Optional argument `Resp.to.CV` is relevant to that scope.

Indeed, different GVF models can actually specify as response term (call it `'resp'` for definiteness) different functions of variables `'SE'`, `'CV'`, and `'VAR'`, but **ReGenesees** will always adopt variable `'CV'` as a *pivot*. Thus, when registering a new GVF model, the user can provide via argument `Resp.to.CV` the function which transforms the response of the model, `'resp'`, into the pivot measure of variability, `'CV'`. A look to the default content of `GVF.db` should make the latter statement clear (see 'Examples').

Note that while `Resp.to.CV` is passed as a character string, that string is expected to represent a well-formed mathematical expression (otherwise function `predictCV` would not work). Moreover, only variables `'resp'` and `'Y'` are allowed to appear inside `Resp.to.CV` (which is enough, since `'VAR'` and `'SE'` can be expressed in terms of `'CV'` and `'Y'`).

If the user does not specify `Resp.to.CV` when registering a new GVF model, he will be not able to use function `predictCV` for predicting CV values based on the fitted GVF model.

Lastly, note that the `Model.id` of a newly inserted GVF model will automatically be set, by adding 1 to the previous maximum of `Model.id`.

GVF.db$delete()

Function `GVF.db$delete` has just a single mandatory argument: `Model.id`. It must match the integer key of the (already existing) GVF model you want to drop from `GVF.db`.

Note that, after deleting a GVF model from `GVF.db`, values of column `Model.id` will be automatically renumbered, so as to range always from 1 to `nrow(GVF.db)`.

GVF.db$get()

Function `GVF.db$get` has no mandatory arguments. When invoked, the function returns the *current* content of `GVF.db`, so that it can be assigned and saved/exported for later usage (see 'Examples'). Should the *current* content of `GVF.db` happen to be empty, the function would inform the user and return `NULL`. The return value of `GVF.db$get` has class `"GVF.db_exported"`, and inherits from class `"data.frame"`.

GVF.db$assign()
Function `GVF.db$assign` has just a single mandatory argument: `value`. The object passed to argument `value` can only be a previously exported copy of `GVF.db`, i.e. an object of class `GVF.db_exported`. The function overwrites the *current* version of `GVF.db` with `value`. As a result, after invoking `GVF.db$assign`, the content of `GVF.db` is `value`.

GVF.db$reset()
Function `GVF.db$reset` has no mandatory arguments and simply restores the default version of `GVF.db` (i.e. the one containing built-in GVF models only).

Author(s)

Diego Zardetto

Source

Built-in GVF models for frequencies (i.e. those with `Model.id` 1, 2, and 3) are discussed in Chapter 7 of [Wolter 07], along with their theoretical justification. Built-in GVF models for totals (i.e. those with `Model.id` 4, and 5) lack a rigorous justification, but have sometimes been used successfully on a purely empirical basis. For instance, Istat surveys on structural business statistics adopted models of that kind to summarize standard errors in publications and to allow their approximate evaluation on a custom basis.

References

Wolter, K.M. (2007) *"Introduction to Variance Estimation"*, Second Edition, Springer-Verlag, New York.

See Also

`estimator.kind` to assess what kind of estimates are stored inside a survey statistic object, `gvf.input` and `svystat` to prepare the input for GVF model fitting, `fit.gvf` to fit GVF models, `plot.gvf.fit` to get diagnostic plots for fitted GVF models, `drop.gvf.points` to drop alleged outliers from a fitted GVF model and simultaneously refit it, and `predictCV` to predict CV values via fitted GVF models.

Examples

```
# Print the current content of GVF.db (invoking
# print(GVF.db) would do the same):
GVF.db

# Inspect the structure of the GVF.db data frame:
data.class(GVF.db)
str(GVF.db)
dim(GVF.db)
nrow(GVF.db)


######################
# Accessor functions #
```

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

# Delete the 3rd model:
GVF.db$delete(3)
# Print GVF.db (note that Model.id has been renumbered,
# so as to range always from 1 to nrow(GVF.db))
GVF.db

# Now delete the 1st model:
GVF.db$delete(1)
GVF.db

# Reset GVF.db to its default values:
GVF.db$reset()
GVF.db

# Insert a new tentative GVF model for Totals:
GVF.db$insert(CV ~ I(1/Y^2) + I(1/Y) + Y + I(Y^2), "Total", "resp")
GVF.db
# (notice that invoking GVF.db$insert() with first argument of type character,
# i.e. GVF.model="CV~I(1/Y^2)+I(1/Y)+Y+I(Y^2)", would have obtained exactly the
# same result)

# Now suppose you have somehow validated your newly added model,
# and you want to save your current, enhanced GVF.db in order to
# be able to use it later in a subsequent ReGenesees session.
### This can be achieved as follows:
### START
  # 1. You must first get a copy of it, by using accessor function
  #    GVF.db$get:
myGVF.db <- GVF.db$get()
myGVF.db
data.class(myGVF.db)

  # 2. Then, you must save the copy to a .RData workspace, in order
  #    to be able to load it later when needed, e.g.:
## Not run:
save(myGVF.db, file="custom.GVF.Archive.RData")

## End(Not run)

  # 3. Starting a new ReGenesees session will set the default GVF.db,
  #    which we can simulate in this example as follows:
GVF.db$reset()
GVF.db

  # 4. Now you can load your previously saved customized GVF.db...
## Not run:
load("custom.GVF.Archive.RData")

## End(Not run)
  # ...so that myGVF.db is back into your .GlobalEnv:
myGVF.db

  # 5. Lastly, you must overwrite GVF.db with your custom
  #    GVF archive myGVF.db via function GVF.db$assign:
GVF.db$assign(myGVF.db)
```

```
GVF.db

### Now your custom GVF archive is ready to be used by ReGenesees.
### STOP

# Illustrate some GVF.db$insert checks by trying crazy models
# or ill-specified attributes

# Examples start: reset GVF.db to its default values
GVF.db$reset()
GVF.db

# GVF model must be "syntactically new"...
## Not run:
GVF.db$insert(log(CV^2) ~ log(Y))

## End(Not run)
# ...if this is the case, it can even be "equivalent" to old ones: e.g.
# the following is identical to model number 5 and will produces identical
# estimates and predictions (as you may want to check):
GVF.db$insert(I(sqrt(VAR)/Y) ~ I(1/Y) + Y, "Total", Resp.to.CV = "resp")
GVF.db

# GVF model must have a response term
## Not run:
GVF.db$insert(~ log(Y))

## End(Not run)

# GVF model response must involve some of 'SE', 'CV', 'VAR'
## Not run:
GVF.db$insert(DEFF ~ log(Y))

## End(Not run)

# GVF model predictor must involve 'Y'
## Not run:
GVF.db$insert(VAR ~ SE)

## End(Not run)

# If passed, Resp.to.CV can only involve 'resp' and 'Y'
## Not run:
GVF.db$insert(I(sqrt(VAR)/Y) ~ I(1/Y) + Y + I(Y^2), Resp.to.CV = "sqrt(VAR)/Y")

## End(Not run)

# Examples end: reset GVF.db to its default values:
GVF.db$reset()
```

---

gvf.input *Prepare Input Data to Fit GVF Models*

Description

Transforms a set of computed survey statistics into a suitable (`data.frame`-like) data structure, in order to fit a Generalized Variance Function model.

Usage

```
gvf.input(design, ..., stats = list(...))

## S3 method for class 'gvf.input'
plot(x, ...)
```

Arguments

`design`	The design object (of class `analytic` or inheriting from it) from which the input survey statics are supposed to have been derived.
`...`	For function `gvf.input`, objects containing survey statistics. For `plot`, further arguments passed to or from other methods.
`stats`	A `list` storing survey statistic objects (see 'Details').
`x`	The object of class `gvf.input` to plot.

Details

Given a set of survey statistic objects (via arguments '`...`' or `stats`) and a design object (`design`) from which those statics are supposed to have been derived, function `gvf.input` builds a data structure that can be fed to **ReGenesees** GVF model fitting function `fit.gvf`.

Argument '`...`' can be bound to an arbitrary number of objects. These objects must be output of survey statistics functions, i.e. `svystatTM`, `svystatR`, `svystatS`, `svystatSR`, `svystatB`, `svystatQ`, and `svystatL`.
All input objects passed to '`...`' must derive from estimators of the *same* kind (as returned by function `estimator.kind`). For the same reason, objects of *mixed* kind (see `estimator.kind`) are not allowed. Since function `svystatL` can actually handle estimators of *different* kinds, objects of kind '`Complex Estimator`' are the only exception to the rule.

Argument `stats` can be used as an alternative to argument '`...`': one has only to store the survey statistic objects into a `list` and bind such list to `stats`. Note that, if both are passed, argument `stats` will prevail on '`...`' (see 'Examples').

Should *any* input object be a survey statistic derived from a design object *other than* `design`, the function would raise an error.

The `plot` method for `gvf.input` objects produces a matrix of scatterplots with polynomial smoothers.

Value

An object of class `gvf.input`, inheriting from class `data.frame`: basically a data frame supplied with appropriate attributes.
Each row of the data frame contains an *estimate* along with its estimated sampling error, expressed in terms of *standard error*, *coefficient of variation*, *variance*, and - whenever available - *design effect*.
The data frame has the following structure:

`name` The name of the original estimate, `factor`.

`Y` The value of the original estimate, `numeric`.

`SE` The standard error of the original estimate, `numeric`.

`CV` The coefficient of variation of the original estimate, `numeric`.

`VAR` The variance of the original estimate, `numeric`.

`DEFF` The design effect of the original estimate (if available), `numeric`.

Note that by inspecting the *attributes* of a `gvf.input` object, one can always identify which design object and which kind of estimator generated that object (see 'Examples').

Author(s)

Diego Zardetto

See Also

`estimator.kind` to assess what kind of estimates are stored inside a survey statistic object, `svystat` as a useful alternative to prepare the input for GVF model fitting, `GVF.db` to manage **ReGenesees** archive of registered GVF models, `fit.gvf` to fit GVF models, `plot.gvf.fit` to get diagnostic plots for fitted GVF models, `drop.gvf.points` to drop alleged outliers from a fitted GVF model and simultaneously refit it, and `predictCV` to predict CV values via fitted GVF models.

Examples

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

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

# ...and use it to compute some survey statistics:
va<-svystatTM(sbsdes,~va.imp2)
va.reg<-svystatTM(sbsdes,~va.imp2,~region)
va.area<-svystatTM(sbsdes,~va.imp2,~area)

# Now suppose you want to fit a GVF model on the estimates and errors computed
# above: you must prepare your input as follows:
ee<-gvf.input(sbsdes,va,va.reg,va.area)

# Inspect the obtained data structure:
ee
str(ee)   # Note the "design" and "stats.kind" attributes
plot(ee)

# Note that, instead of argument '...', you could have used argument 'stats'
# as follows:
va.list<-list(va,va.reg,va.area)
ee2<-gvf.input(sbsdes,stats=va.list)

# ...obtaining exactly the same result:
identical(ee,ee2)

# Note also that, if both are passed, argument 'stats' prevails on '...':
  # indeed, while:
  gvf.input(sbsdes,va.reg)
  # we would get again:
  gvf.input(sbsdes,va.reg,stats=va.list)
```

gvf.misc *Miscellanea: Methods for Fitted GVF Models*

Description

These methods extract information from fitted GVF model(s).

Usage

```
## S3 method for class 'gvf.fit'
coef(object, ...)
## S3 method for class 'gvf.fits'
coef(object, ...)
## S3 method for class 'gvf.fit.gr'
coef(object, ...)
## S3 method for class 'gvf.fits.gr'
coef(object, ...)

## S3 method for class 'gvf.fit'
residuals(object, ...)
## S3 method for class 'gvf.fits'
residuals(object, ...)
## S3 method for class 'gvf.fit.gr'
residuals(object, ...)
## S3 method for class 'gvf.fits.gr'
residuals(object, ...)

## S3 method for class 'gvf.fit'
fitted(object, ...)
## S3 method for class 'gvf.fits'
fitted(object, ...)
## S3 method for class 'gvf.fit.gr'
fitted(object, ...)
## S3 method for class 'gvf.fits.gr'
fitted(object, ...)

## S3 method for class 'gvf.fit'
predict(object, ...)
## S3 method for class 'gvf.fits'
predict(object, ...)
## S3 method for class 'gvf.fit.gr'
predict(object, ...)
## S3 method for class 'gvf.fits.gr'
predict(object, ...)

## S3 method for class 'gvf.fit'
effects(object, ...)
## S3 method for class 'gvf.fits'
effects(object, ...)
## S3 method for class 'gvf.fit.gr'
effects(object, ...)
```

```
## S3 method for class 'gvf.fits.gr'
effects(object, ...)

## S3 method for class 'gvf.fit'
rstandard(model, ...)
## S3 method for class 'gvf.fits'
rstandard(model, ...)
## S3 method for class 'gvf.fit.gr'
rstandard(model, ...)
## S3 method for class 'gvf.fits.gr'
rstandard(model, ...)

## S3 method for class 'gvf.fit'
rstudent(model, ...)
## S3 method for class 'gvf.fits'
rstudent(model, ...)
## S3 method for class 'gvf.fit.gr'
rstudent(model, ...)
## S3 method for class 'gvf.fits.gr'
rstudent(model, ...)

## S3 method for class 'gvf.fit'
anova(object, ...)
## S3 method for class 'gvf.fits'
anova(object, ...)
## S3 method for class 'gvf.fit.gr'
anova(object, ...)
## S3 method for class 'gvf.fits.gr'
anova(object, ...)

## S3 method for class 'gvf.fit'
vcov(object, ...)
## S3 method for class 'gvf.fits'
vcov(object, ...)
## S3 method for class 'gvf.fit.gr'
vcov(object, ...)
## S3 method for class 'gvf.fits.gr'
vcov(object, ...)
```

Arguments

`object`	An object containing one or more fitted GVF models (see 'Usage' for the allowed classes).
`model`	An object containing one or more fitted GVF models (see 'Usage' for the allowed classes).
`...`	Further arguments passed to or from other methods (see corresponding .lm methods).

Details

These methods can be used to extract information from fitted GVF model(s).

For more details on their usage, please read the help pages of the methods with same name defined on class .lm by package **stats** (e.g. `coef`, `fitted`, etc.).

Value

The requested information, wrapped into an R object whose structure depends on the class of the input fitted GVF model(s) (i.e. arguments `object` and/or `model`).

Author(s)

Diego Zardetto

See Also

`GVF.db` to manage **ReGenesees** archive of registered GVF models, `gvf.input` and `svystat` to prepare the input for GVF model fitting, `fit.gvf` to fit GVF models, `plot.gvf.fit` to get diagnostic plots for fitted GVF models, `drop.gvf.points` to drop alleged outliers from a fitted GVF model and simultaneously refit it, and `predictCV` to predict CV values via fitted GVF models.

Examples

```
# Load example data:
data(AF.gvf)

# Inspect available estimates and errors of counts:
head(ee.AF)
summary(ee.AF)

# List available registered GVF models:
GVF.db

## (A) A *single* fitted GVF model ##
# Fit example data to registered GVF model number one:
m <- fit.gvf(ee.AF, 1)

# Extract some information:
coef(m)
fitted(m)


## (B) A *a set* of GVF models fitted to the same data ##
# Fit example data to registered GVF models for frequencies (i.e. number 1:3):
mm <- fit.gvf(ee.AF, 1:3)

# Extract some information:
r.mod <- residuals(mm)
lapply(r.mod, head)

r.sta <- rstandard(mm)
lapply(r.sta, head)

r.stu <- rstudent(mm)
lapply(r.stu, head)


## (C) a *single* GVF model fitted to *grouped* data ##
```

```
# We have at our disposal the following survey design object on household data:
exdes

# Use function svystat to prepare *grouped* estimates and errors of counts
# to be fitted separately (here groups are regions):
ee.g <- svystat(exdes, y=~ind, by=~age5c:marstat:sex, combo=3, group=~regcod)
str(ee.g)

# Fit registered GVF model number one separately inside groups:
m.g <- fit.gvf(ee.g, 1)

# Extract some information:
coef(m.g)
fitted(m.g)


## (D) a *set of* GVF models fitted to *grouped* data ##
# Fit all registered GVF models for frequencies (i.e. number 1:3) separately
# inside groups:
mm.g <- fit.gvf(ee.g, 1:3)
# Extract some information:
coef(mm.g)
fitted(mm.g)
```

`plot.gvf.fit` *Diagnostic Plots for Fitted GVF Models*

Description

This function provides basic diagnostic plots for fitted GVF model(s).

Usage

```
## S3 method for class 'gvf.fit'
plot(x, which.more = 1:3, id.n = 3, labels.id = names(residuals(x)),
     cex.id = 0.75, label.pos = c(4, 2), cex.caption = 1, Main = NULL, ...)

## S3 method for class 'gvf.fits'
plot(x, which.more = NULL, id.n = 3, labels.id = names(residuals(x)),
     cex.id = 0.75, label.pos = c(4, 2), cex.caption = 1, Main = NULL, ...)

## S3 method for class 'gvf.fit.gr'
plot(x, which.more = 1:3, id.n = 3, labels.id = NULL,
     cex.id = 0.75, label.pos = c(4, 2), cex.caption = 1, ...)

## S3 method for class 'gvf.fits.gr'
plot(x, which.more = NULL, id.n = 3, labels.id = NULL,
     cex.id = 0.75, label.pos = c(4, 2), cex.caption = 1, ...)
```

Arguments

`x`	An object containing one or more fitted GVF models (see 'Usage' for the allowed classes).

which.more	Select additional plots beyond the default one ('Observed vs Fitted'). Can be any subset of vector 1:6 with up to three elements.
id.n	Number of points to be initially labelled in each plot, starting with the most extreme.
labels.id	Vector of labels, from which the labels for extreme points will be chosen. NULL uses observation numbers.
cex.id	Magnification of point labels.
label.pos	Positioning of labels, for the left half and right half of the graph(s) respectively.
cex.caption	Controls the size of caption.
Main	Optional string to be added to automatic plot titles.
...	Other parameters to be passed through to plotting functions.

Details

Diagnostic plots can be useful both for assessing the goodness of a GVF model fit qualitatively, and for selecting the "best" GVF model among different alternatives.

This function can provide any of the following 7 plots:

```
(0)  'Observed vs Fitted'
(1)  'Residuals vs Fitted'
(2)  'Normal Q-Q'
(3)  'Scale-Location'
(4)  'Cook's distance'
(5)  'Residuals vs Leverage'
(6)  'Cook's distances vs Leverage/(1-Leverage)'
```

The *'Residuals vs Fitted'* plot is special in that it will be always provided: this explains its zero-th order in the list above. The rest of the list, namely plots 1:6, exactly matches the numbering convention of function plot.lm.

Additional plots - beyond *'Residuals vs Fitted'* - can be requested through argument which.more. Any subset of 1:6 is allowed, provided its length does not exceed 3. Therefore, at most 4 plots will be generated simultaneously.

Note that the *default* behaviour of this function do depend on whether input object x stores *one* or *more than one* fitted GVF models. In the first case, plots 0:3 will be returned in a multiple plot with a 2x2 layout. In the second case, only the default plot number 0 will be returned, opening a new graphics frame for each different GVF model.

Argument id.n specifies how many points have to be labelled, starting with the most extreme in terms of residuals: this applies to all plots.

Argument Main is expected to be seldom (if ever) useful: its main purpose is programming consistency at a deeper level.

All the other arguments have the same meaning as in function plot.lm.

Author(s)

Diego Zardetto

References

See plot.lm and references therein.

See Also

GVF.db to manage **ReGenesees** archive of registered GVF models, gvf.input and svystat to prepare the input for GVF model fitting, fit.gvf to fit GVF models, drop.gvf.points to drop alleged outliers from a fitted GVF model and simultaneously refit it, and predictCV to predict CV values via fitted GVF models.

Examples

```
# Load example data:
data(AF.gvf)

# Inspect available estimates and errors of counts:
str(ee.AF)

# List available registered GVF models:
GVF.db


##############################################
# Diagnostic plots for fitted GVF model(s) #
##############################################

## (A) Plots of a *single* fitted GVF model ##
# Fit example data to registered GVF model number one:
m <- fit.gvf(ee.AF, 1)

## Default call yields 4 plots:
plot(m)

# Play with argument 'which.more':
## which.more = NULL yields the "Observed vs Fitted" plot only:
plot(m, which.more = NULL)

## which.more = 1 adds the "Residuals vs Fitted" plot:
plot(m, which.more = 1)

## subsets of 1:6 with length <= 3 are allowed:
plot(m, which.more = c(1:2,4))

# Just for illustration, play with other parameters:
plot(m, id.n = 6, col = "blue", pch = 20)


## (B) Plots of *many* fitted GVF models ##
# Fit example data to registered GVF models for frequencies (i.e. number 1:3):
mm <- fit.gvf(ee.AF, 1:3)

## Default call yields the "Observed vs Fitted" plot reported separately for
## each model in subsequent graphics frames:
plot(mm)

# Play with argument 'which.more':
## which.more = 1:3 yields subsequent 2x2 plots:
plot(mm, which.more = 1:3)

## again, subsets of 1:6 are allowed:
```

```
plot(mm, which.more = 1)


#####################################################
# Diagnostic plots for "grouped" fitted GVF model(s) #
#####################################################
# We have at our disposal the following survey design object on household data:
exdes

# Use function svystat to prepare "grouped" estimates and errors of counts
# to be fitted separately (here groups are regions):
ee <- svystat(exdes, y=~ind, by=~age5c:marstat:sex, combo=3, group=~regcod)
lapply(ee, head)

## (C) Plots of a *single* GVF model fitted to different groups ##
# Fit registered GVF model number one separately inside groups:
m.g <- fit.gvf(ee, 1)

## Default call yields 4 plots reported separately for each group
## in subsequent graphics frames:
plot(m.g)

## Play with argument 'which.more' to select different plots:
plot(m.g, which.more = c(1:2,4))


## (D) Plots of *many* GVF model fitted to different groups ##
# Fit all registered GVF models for frequencies separately inside groups:
mm.g <- fit.gvf(ee, 1:3)

## Default call yields the "Residuals vs Fitted" plot reported separately
## for each group in subsequent graphics frames:
plot(mm.g)

## Play with argument 'which.more' to add more plots:
plot(mm.g, which.more = 1:3, id.n = 6, col = "blue", pch = 20)
```

`pop.desc` *Natural Language Description of Known Totals Templates*

Description

Provides a natural language description of a known totals data frame to be used for a calibration task.

Usage

```
pop.desc(pop.totals, ...)

## S3 method for class 'pop.totals'
pop.desc(pop.totals, ...)

## S3 method for class 'spc.pop'
pop.desc(pop.totals, verbose = FALSE, ...)
```

Arguments

pop.totals	An object of class pop.totals, be it a template or the actual known totals data frame for the calibration task.
verbose	Fully describe the control totals of a special purpose calibration task?
...	Parameters for future extensions (currently unused).

Details

Function pop.template generates a *template* (i.e. *empty*) data frame of class pop.totals, which is appropriate to store the known totals of a given calibration task. Afterwards, the template data frame must be filled with actual figures.

When the sampling frame of the survey is available and the actual population totals can be calculated from this source, function fill.template (i) automatically computes the totals of the auxiliary variables from the sampling frame, (ii) safely arranges and formats these values according to the template structure. Therefore, function fill.template avoids any need for the user to understand, comply with, or even be aware of, the structure of the template that is being filled.

On the contrary, when the population 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), it is up to the user to correctly fill the template, that is to put the *right values* in the *right slots* of the prepared template.

Function pop.desc has been designed for users who cannot take advantage of function fill.template, to help them understand the structure of the known totals template, in order to safely fill it with actual figures.

Invoking pop.desc will print on screen a detailed natural language description of the structure of the input pop.totals object. Such description will clarify how known totals are organized inside the template slots.

Value

The main purpose of the function is to print on screen, anyway it returns invisibly the input pop.totals object (as print would do).

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

```
## First prepare some design objects to work with:

# Load household data:
data(data.examples)
# Build a design object:
exdes<-e.svydesign(data=example,ids=~towcod+famcod,strata=~SUPERSTRATUM,
       weights=~weight)
```

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

## Now build some known totals templates that (after having been filled by
## actual figures) could be used to calibrate the design objects above,
## and explore the corresponding natural language description:

  ####################################
  ## Some simple and small examples ##
  ####################################
expop<-pop.template(exdes,calmodel=~1)
expop
pop.desc(expop)

expop<-pop.template(exdes,calmodel=~sex)
expop
pop.desc(expop)

# equivalent to the one above
expop<-pop.template(exdes,calmodel=~sex-1)
expop
pop.desc(expop)

expop<-pop.template(exdes,calmodel=~x1+x2+x3,partition=~procod)
expop
pop.desc(expop)

expop<-pop.template(exdes,calmodel=~sex:marstat-1)
expop
pop.desc(expop)

# equivalent to the one above
expop<-pop.template(exdes,calmodel=~sex*marstat-1)
expop
pop.desc(expop)

# equivalent to the one above
expop<-pop.template(exdes,calmodel=~1,partition=~sex:marstat)
expop
pop.desc(expop)

expop<-pop.template(exdes,calmodel=~x1+age5c:marstat-1,partition=~regcod:sex)
expop
pop.desc(expop)


  ##########################################
  ## Some more involved and bigger examples ##
  ##########################################
expop<-pop.template(exdes,calmodel=~sex:age10c:regcod + sex:age5c:procod - 1)
expop
pop.desc(expop)

# equivalent to the one above (because procod is nested into regcod and
# age10c is nested into age5c)
```

```
expop<-pop.template(exdes,calmodel=~age10c+procod-1,
       partition=~regcod:sex:age5c)
expop
pop.desc(expop)
# NOTE: Most of the entries of the template above will be structural zeros,
#       as can be seen in what follows:
expop.HT<-aux.estimates(exdes, template=expop)
expop.HT
sum(expop.HT==0)


# Switch to sbs data
sbspop<-pop.template(sbsdes,
        calmodel=~(emp.num + ent):(nace.macro + emp.cl) - 1, partition=~region)

# Can fill the template using the sampling frame...
sbspop<-fill.template(universe=sbs.frame,template=sbspop)
sbspop

# ...and invoke function pop.desc on the filled known totals data frame:
pop.desc(sbspop)

sbspop <- pop.template(data=sbsdes,
                       calmodel=~((emp.num + ent):(nace2 + emp.cl:nace.macro))-1,
                       partition=~region:public)
sbspop
pop.desc(sbspop)
```

`pop.fuse` *Fuse Control Totals Data Frames for Special Purpose and Ordinary Calibration Tasks*

Description

Allows to solve jointly a special purpose calibration task and an ordinary calibration task by fusing their respective control totals.

Usage

```
pop.fuse(spc.pop, pop, design)
```

Arguments

`spc.pop`	A control totals data frame prepared for a *special purpose calibration* task. Must be of class `spc.pop`.
`pop`	A known totals data frame for an *ordinary calibration* task. Must be of class `pop.totals`.
`design`	A design object *prepared* for a special purpose calibration task.

Details

ReGenesees 2.1 introduced support for *'special purpose calibration'* tasks, i.e. facilities to calibrate survey weights so as to match complex, non-linear population parameters, instead of ordinary population totals.

Currently, **ReGenesees**' support for special purpose calibration tasks is limited to Multiple Regression Coefficients (see `prep.calBeta` and `pop.calBeta`), which includes calibration on Means as a notable special case. Further support will likely be provided in future extensions.

Function `pop.fuse` allows you to run a calibration task that *simultaneously* involves as benchmarks:

- Complex population parameters (e.g. multiple regression coefficients).
- Ordinary population totals.

To achieve this goal, `pop.fuse` simply *"fuses"* the corresponding control totals data frames. The resulting fused control totals data frame is indeed enough to automatically instruct function `e.calibrate` to run the joint calibration task.

Argument `spc.pop` must be an object of class `spc.pop`, namely a control totals data frame prepared for a special purpose calibration task (e.g. via function `pop.calBeta` for the case of calibration on multiple regression coefficients).

Argument `pop` must be an object of class `pop.totals`, namely a known totals data frame for an ordinary calibration task (e.g. generated using functions `pop.template` and `fill.template`).

Argument `design` must be the survey design object that:
(i) was already *prepared* for the special purpose calibration task at hand, and
(ii) you want to calibrate simultaneously also on the ordinary population totals at hand.

Note that condition (i) requires that object `design` has actually been used to build object `spc.pop` (see the 'Examples' section). Note, moreover, that condition (ii) requires that you eventually run `e.calibrate` on object `design` (see the 'Examples' section).

Value

A *fused* data frame, with class `spc.pop`, encompassing control totals for both the special purpose calibration task and the ordinary calibration task.

Note that printing this control totals data frame might not be very telling: to better understand its structure you should instead leverage function `pop.desc`, for which a method dedicated to class `spc.pop` is available.

Author(s)

Diego Zardetto

See Also

`e.calibrate` to calibrate weights, functions `prep.calBeta` and `pop.calBeta` to prepare survey data and control totals for calibration on multiple regression coefficients, functions `pop.template` and `fill.template` to generate and fill population totals templates for ordinary calibration tasks, `pop.desc` to obtain a natural language description of control totals data frames.

Examples

```
# Function pop.fuse allows you to run a calibration task that simultaneously
# involves as benchmarks:
# (A) complex population parameters (e.g. multiple regression coefficients)
```

```
# (B) ordinary population totals
# You just have to:
## 1) prepare the survey desing and control totals data frame for (A)
## 2) create and fill the known totals data frame for (B)
## 3) fuse the control totals data frames produced in steps 1) and 2)!

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

# (A) Suppose you know with satisfactory accuracy from some external source
#     the regression coefficients of the following model:
model <- va.imp2 ~ emp.num + emp.cl

# Here, use the sbs sampling frame available in ReGenesees to simulate the
# external source and compute the values of the regression coefficients:
Beta <- coef(lm(model, data = sbs.frame))
Beta

## 1) Prepare the survey design and control totals for calibration (A):
sbsdes.A <- prep.calBeta(sbsdes, model, Beta)
pop.A <- pop.calBeta(sbsdes.A)

# (B) Suppose you know the number of enterprises and employees by economic
#     activity macro sectors
## 2) Create and fill the known totals data frame for calibration (B)
pop.B <- pop.template(sbsdes.A, calmodel = ~(ent + emp.num):nace.macro - 1)
# Note that, to create the template above, you could have used equally well
# the original object sbsdes.
pop.B <- fill.template(pop.B, universe = sbs.frame)

## 3) Lastly, fuse the control totals data frames produced in steps 1) and 2)
pop.AB <- pop.fuse(pop.A, pop.B, sbsdes.A)
# Note that, to create the fused control totals data frame above, you *MUST USE*
# the *PREPARED* design object sbsdes.A

# Have a look:
pop.desc(pop.AB)
# ...and recall you can set verbose = TRUE to see the full structure
# pop.desc(pop.AB, verbose = TRUE)

# Now you are ready to calibrate simultaneously on (A) and (B)
sbscal.AB <- e.calibrate(sbsdes.A, pop.AB)
# Note again that, to run the calibration on the *fused* control totals, you
# *MUST USE* the *PREPARED* design object sbsdes.A

# Now, check that all the benchmarks are indeed matched:
# (A) Multiple regression cofficients:
svystatB(sbscal.AB, model)
Beta

# (B) Ordinary population totals:
svystatTM(sbscal.AB, ~ent + emp.num, ~nace.macro)
pop.B
# OK
```

pop.plot *Plot Calibration Control Totals vs Current Estimates*

Description

Draw a scatter plot of calibration control totals vs current estimates.

Usage

```
pop.plot(pop.totals, ...)

## S3 method for class 'pop.totals'
pop.plot(pop.totals, design,
                              xlab = "Current Estimates",
                              ylab = "Calibration Control Totals",
                              lcol = c("red", "green", "blue"),
                              lwd = c(1, 1, 1),
                              lty = c(2, 1, 2),
                              verbose = TRUE, ...)

## S3 method for class 'spc.pop'
pop.plot(pop.totals, design,
                            xlab = "Current Estimates",
                            ylab = "Calibration Control Totals",
                            lcol = c("red", "green", "blue"),
                            lwd = c(1, 1, 1),
                            lty = c(2, 1, 2),
                            verbose = TRUE, ...)
```

Arguments

pop.totals	A known totals data frame for a calibration task. Must be of class pop.totals (for ordinary calibration tasks) or spc.pop (for special purpose calibration tasks).
design	A design object to compute estimates of the calibration control totals. Must be of class of class analytic (or inheriting from it).
xlab	A suggested label for the x axis. See also plot.
ylab	A suggested label for the y axis. See also plot.
lcol	Colors of reference lines, see 'Details'. See also par.
lwd	Width of reference lines, see 'Details'. See also par.
lty	Type of reference lines, see 'Details'. See also par.
verbose	Print on screen minimum and maximum slopes?
...	Other parameters to be passed through to plotting functions.

Details

Function pop.plot draws a scatter plot of calibration control totals vs current estimates derived from object design. This plot is sometimes very telling and may provide a first-level, rough assessment of how hard the calibration problem at hand will turn out to be in terms of constrained optimization.

A trivial calibration problem would correspond to points lying exactly on the y = x bisector line. A moderate spread around the bisector line without evident and anomalous patterns is usually a sign of a well behaved calibration task. Larger spreads signal increasing calibration complexity. The maximum and minimum slopes of lines connecting the origin to the points in the scatter plot provide often a reasonable clue about the minimum possible range for the calibration bounds [L,U], see bounds.hint. Evident and anomalous patterns emerging from the scatter plot not only generally imply harder calibration tasks, but can even reveal signs of underlying biases in the auxiliary variables. For instance, an extremely imbalanced pattern, with the overwhelming majority of points lying below (or above) the y = x line would be a striking symptom of downward (or upward) bias of the current estimates with respect to the true population totals.

In addition to the scatter plot of calibration control totals vs current estimates, function pop.plot draws three useful reference lines: (1) the maximum slope line connecting the origin to the points, (2) the y = x bisector line, and (3) the minimum slope line connecting the origin to the points.

If verbose = TRUE, the minimum and maximum slopes get printed on screen.

When the calibration control totals in pop.totals span different order of magnitudes (as it often happens in enterprise surveys) it is beneficial to look at the plot in logarithmic scale, which can be achieved by setting log = "xy" via argument ....

If pop.totals is of class spc.pop, i.e. it stores known totals for a special purpose calibration tasks, which are all zero (see prep.calBeta, then just one reference vertical line at x = 0 is added to the scatter plot.

If pop.totals is of class spc.pop but it is a *fused* data frame, encompassing simultaneous control totals for a special purpose calibration task and an ordinary calibration task (see pop.fuse), than two separate scatter plots are drawn.

Note that you can also pass to argument design the calibrated object obtained using pop.totals as known totals data frame: if calibration perfectly converged, points will lie on the y = x bisector line, otherwise the spread around the bisector line will provide a visual assessment of how significantly the calibration task failed.

Value

The main purpose of the function is to draw a plot, anyway it returns invisibly the minimum and maximum slopes whenever it is meaningful to do so (thus excluding non-fused special purpose calibration task).

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, pop.desc to obtain a natural language description of control totals data frames.

Examples

```
## First prepare some design objects to work with:
# Load household data:
data(data.examples)

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

# Now suppose the known population totals for the calibration task
# are stored in object pop07p:
class(pop07p)
dim(pop07p)

# These totals refer to the
# 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;

# For a natural language description, run:
# pop.desc(pop07p)

## Here is the totals vs estimates scatter plot:
pop.plot(pop07p, exdes, pch = 20)

# ...or with a log-log scale:
pop.plot(pop07p, exdes, pch = 20, log = "xy")

# Now calibrate (e.g. with the unbounded linear function):
excal07p <- e.calibrate(exdes, pop07p)

# As calibration converged...
check.cal(excal07p)

# ...the points now lie on the y = x bisector:
pop.plot(pop07p, excal07p, pch = 20)

## You are encouraged to try the function on control totals for special
## purpose calibration tasks (both simple and fused, see ?prep.calBeta,
## ?pop.calBeta, and ?pop.fuse). It might be interesting.
```

`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). Should empty levels be present in any factor variable belonging to data, they would be dropped.

The mandatory argument calmodel symbolically defines the calibration model you intend to use: 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 factorized). 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). Note that, if the partition formula involves two or more factors, their crossed levels will be ordered according to operator : (that is, those from the *rightmost* variable will vary fastest).

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

The pop.totals class is a specialization 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, pop.desc to provide a natural language description

of the template structure, and 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 factorized
# both a global and a partitioned 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 (partitioned 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)

#      4.1.3) Using sex and marstat to define calibration domains:
pop.template(data=example,calmodel=~1,partition=~sex: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 partitioned 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 to use: 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 factorized). 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`). Note that, if the `partition` formula involves two or more factors, their crossed levels will be ordered according to operator `:` (that is, those from the *rightmost* variable will vary fastest).

Value

An invisible object of class pop.totals. The pop.totals class is a specialization 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 partitioned 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 e.calibrate 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 e.calibrate 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)
```

`predictCV` *Predict CV Values via Fitted GVF Models*

Description

This function predicts the CV values associated to given estimates, based on fitted GVF model(s).

Usage

```
predictCV(object, new.Y = NULL, scale = NULL, df = Inf,
          interval = c("none", "confidence", "prediction"), level = 0.95,
          na.action = na.pass, pred.var = NULL, weights = 1)
```

Arguments

`object`	An object containing one or more fitted GVF models.
`new.Y`	A data frame storing new estimates whose CVs have to be predicted. If omitted or `NULL`, CVs arising from the fitted GVF model(s) will be returned.
`scale`	Scale parameter for standard error calculation. See also `predict.lm`.

df	Degrees of freedom for scale. See also predict.lm.
interval	Type of interval calculation. Can be abbreviated. See also predict.lm.
level	Confidence (or tolerance) level for intervals. See also predict.lm.
na.action	Function determining what should be done with missing values in new.Y. The default is to predict NA. See also predict.lm.
pred.var	The variance(s) for future observations to be assumed for prediction intervals. See also predict.lm.
weights	Variance weights for prediction. This can be a numeric vector or a one-sided model formula. In the latter case, it is interpreted as an expression evaluated in new.Y. See also predict.lm.

Details

The main motivation for building and fitting a GVF model is to exploit the fitted model to *predict* the sampling error associated to a given estimate, instead of having to *compute* directly an estimate of such sampling error. Function predictCV is relevant to that scope.

Despite different GVF models can specify as response term (call it 'resp') different functions of variables 'SE', 'CV', and 'VAR' (see e.g. [Wolter 07]), function predictCV adopts variable 'CV' as a universal *pivot*. This means that predictCV can handle *only* fitted GVF models which are *registered* (that is already stored inside GVF.db), and for which variable Resp.to.CV is *not* NA. Indeed, it is variable Resp.to.CV of data frame GVF.db which tells predictCV how to transform the response of an arbitrary GVF model ('resp') into the pivot measure of variability ('CV').

By default new.Y = NULL and CVs (and intervals, if any) obtained by transforming fitted response values will be returned. If passed, argument new.Y must be a data frame storing new estimates for which CVs have to be predicted. Such input estimates have to be stored in column Y of data frame new.Y. Moreover, if object stores GVF model(s) fitted to *grouped* data (namely, it has class gvf.fit.gr or gvf.fits.gr), then new.Y should also have columns identifying the *groups* to which input estimates are referred (see 'Examples'). The *only exception* is the following: if the new.Y data frame has just the Y column, then CVs will be predicted for *all groups* (see 'Examples'). The function will check for consistency between groups available in object and in new.Y (see 'Examples').

If interval = "none" (the default), the function will return predicted CVs only. Otherwise, lower and upper bounds of confidence (or prediction) intervals around predicted CVs will be also provided. Use argument level to specify the desired confidence (or tolerance) level for those intervals.

All the other arguments have the same meaning as in function predict.lm.

Value

If object is a single GVF model (classes gvf.fit and gvf.fit.gr), a data frame.

If object is a set of GVF models fitted to the same data (classes gvf.fits and gvf.fits.gr), a list of data frames, one for each input GVF model.

The output data frame(s) will store input estimates new.Y plus additional columns:

CV.fit Predicted CV value, numeric.

CV.lwr Lower bound of requested interval (if any), numeric.

CV.upr Upper bound of requested interval (if any), numeric.

Of course, lower and upper bounds for CVs will be reported only when interval != "none".

Note

Please read the 'Note' section of predict.lm.

Author(s)

Diego Zardetto

References

Wolter, K.M. (2007) *"Introduction to Variance Estimation"*, Second Edition, Springer-Verlag, New York.

See Also

GVF.db to manage **ReGenesees** archive of registered GVF models, gvf.input and svystat to prepare the input for GVF model fitting, fit.gvf to fit GVF models, plot.gvf.fit to get diagnostic plots for fitted GVF models, and drop.gvf.points to drop alleged outliers from a fitted GVF model and simultaneously refit it.

Examples

```
##########################################
# Simple examples to illustrate the syntax #
##########################################
# Load example data:
data(AF.gvf)

# Inspect available estimates and errors of counts:
head(ee.AF)
summary(ee.AF)

# List available registered GVF models:
GVF.db

## (A) A *single* fitted GVF model ##
# Fit example data to registered GVF model number one:
m <- fit.gvf(ee.AF, 1)

# Not passing 'new.Y' yield CVs from fitted response values:
p <- predictCV(m)

# Take a look:
head(p)
with(p, plot(CV, CV.fit, col = "blue", pch = 20))

# Now let's predict CV values for new estimates of counts
# e.g. Y = c(1000, 5000, 10000, 50000, 100000)
# First, put these values into a data frame:
new.Y <- data.frame(Y = c(1000, 5000, 10000, 50000, 100000))
new.Y

# Then, compute predicted values and confidence intervals:
p <- predictCV(m, new.Y, interval = "confidence")
p

# NOTE: Should we ever need it, we could also use function predict
```

```
#       to predict *response* values instead of CVs:
predict(m, new.Y, interval = "confidence")


## (B) A *a set* of GVF models fitted to the same data ##
# Fit example data to registered GVF models for frequencies (i.e. number 1:3):
mm <- fit.gvf(ee.AF, 1:3)

# Let's predict CV values for the same new estimates of counts used above,
# i.e. Y = c(1000, 5000, 10000, 50000, 100000).

# Separate predictions will be obtained from the three fitted GVF models
pp <- predictCV(mm, new.Y, interval = "confidence")
pp

# NOTE: The WARNING above arises from the third fitted GVF model and explains
#       the appearance of NaN at the lower bound of the CV confidence interval
#       for input Y = 100000. Indeed, the response of the third model is the
#       squared CV (which ought to be *positive*), but the prediction for the
#       lower endpoint of the confidence interval happens to be *negative*:
predict(mm, new.Y, interval = "confidence")


## (C) a *single* GVF model fitted to *grouped* data ##
# We have at our disposal the following survey design object on household data:
exdes

# Use function svystat to prepare *grouped* estimates and errors of counts
# to be fitted separately (here groups are regions):
ee.g <- svystat(exdes, y=~ind, by=~age5c:marstat:sex, combo=3, group=~regcod)

# Inspect these grouped estimates and errors of counts:
lapply(ee.g, head)
lapply(ee.g, summary)

# Fit registered GVF model number one, separately inside regions '6', '7',
# and '10':
m.g <- fit.gvf(ee.g, 1)

# Suppose we want to predict CV values for the same new estimates of counts used
# above, i.e. Y = c(1000, 5000, 10000, 50000, 100000).
# Obviously, we need tell to what groups (i.e. regions) should these Y values
# be referred. Therefore, input data frame new.Y should have columns identifying
# the groups (i.e. regions).

# Case 1: all known regions (i.e. regions '6', '7', and '10')
#         Here we can exploit the exception described in section 'Details' and
#         avoid building group variables explicitly:

 # Predict:
p.g <- predictCV(m.g, new.Y, interval = "confidence")
p.g


# Case 2: a subset of known regions (e.g. region '7' only)
#         Here we must build group variables explicitly:
new.Y.g <- data.frame(Y = c(1000, 5000, 10000, 50000, 100000),
```

```
                      regcod = 7)
new.Y.g

 # Predict:
p.g <- predictCV(m.g, new.Y.g, interval = "confidence")
p.g


# Case 3: a subset of known regions (e.g. region '7') *plus* some *unknown*
#         region (e.g. region '11').
#         Unknown groups will be tacitly *discarded*:
new.Y.g <- data.frame(Y = c(1000, 5000, 10000, 50000, 100000),
                      regcod = rep(c(7, 11), each = 5))
new.Y.g

 # Predict:
p.g <- predictCV(m.g, new.Y.g, interval = "confidence")
p.g


# Case 4: only *unknown* regions (e.g. regions '11' and '12).
#         This will raise an *error*:
new.Y.g <- data.frame(Y = c(1000, 5000, 10000, 50000, 100000),
                      regcod = rep(c(11, 12), each = 5))
new.Y.g

## Not run:
 # Predict:
p.g <- predictCV(m.g, new.Y.g, interval = "confidence")

## End(Not run)


# Case 5: *unknown* group variables (e.g. 'region' instead of 'regcod').
#         This will raise an *error*:
new.Y.g <- data.frame(Y = c(1000, 5000, 10000, 50000, 100000),
                      region = rep(c(11, 12), each = 5))
new.Y.g

## Not run:
 # Predict:
p.g <- predictCV(m.g, new.Y.g, interval = "confidence")

## End(Not run)


## (D) a *set of* GVF models fitted to *grouped* data ##
# Fit all registered GVF models for frequencies (i.e. number 1:3) separately
# inside groups:
mm.g <- fit.gvf(ee.g, 1:3)

# Predict CV values for the same new estimates of counts used above,
# i.e. Y = c(1000, 5000, 10000, 50000, 100000), for all the regions:
# Again, here we can exploit the exception described in section 'Details'
# and avoid building group variables explicitly:

 # Predict:
```

```
pp.g <- predictCV(mm.g, new.Y)
pp.g

# NOTE: The WARNING above explains the appearance of NaN for some predicted
#       CV values stemming from the third GVF model. The reason causing this
#       behaviour is exactly the same as discussed in previous example (B).



#####################################################
# Estimating CVs: Prediction vs Direct Calculation #
#####################################################
# Load example data:
data(AF.gvf)

# Fit available registered GVF models for frequencies:
mm <- fit.gvf(ee.AF, model=1:3)

# Get the best fitted model:
mbest <- getBest(mm, criterion="adj.R2")
mbest
# Note: adjusted R^2 used as a 'quick and dirty' criterion, as a thorough model
#       comparison via diagnostic plots would have given the same result.

# Compute directly the estimates and errors of a set of absolute frequencies
# which did not belong to the previously fitted data ee.AF, e.g. the joint
# distribution of marstat and regcod:
marstat.regcod <- svystatTM(exdes, ~I(marstat:regcod))
marstat.regcod

# Predict CVs of the joint distribution of marstat and regcod
# by means of the selected GVF model:
  # First, prepare data with which to predict:
    newdata <- gvf.input(exdes, marstat.regcod)
  # Then, compute CV predictions:
    p.marstat.regcod <- predictCV(mbest, new.Y = newdata, interval="prediction")

# Inspect the results:
p.marstat.regcod

# Plot of computed and predicted CVs with prediction error bars:
  # plot starts #
plot(p.marstat.regcod$Y, p.marstat.regcod$CV.fit, pch=19, col="red",
     ylim=range(p.marstat.regcod$CV.lwr, p.marstat.regcod$CV.upr, p.marstat.regcod$CV),
     xlab="Absolute Frequency Estimate", ylab="Coefficient of Variation",
     main="Estimated and GVF Predicted CVs\n(joint distribution of marstat and regcod)")
segments(x0=p.marstat.regcod$Y, y0=p.marstat.regcod$CV.lwr, y1=p.marstat.regcod$CV.upr,
         col="red")
points(p.marstat.regcod$Y, p.marstat.regcod$CV, pch=0)
legend("topright", title="CV Estimation Method",
       legend=c("Direct Estimate", "GVF Predicted Value", "GVF Prediction Interval"),
       pch=c(0,19,124), col=c("black", "red", "red"), inset=rep(0.05, 2))
  # plot ends #
```

---

prep.calBeta *Calibration on Multiple Regression Coefficients*

Description

Prepare survey data and control totals to run a calibration task on multiple regression coefficients.

Usage

```
prep.calBeta(design, model, Beta,
             by = NULL, partition = FALSE, drop.z = TRUE)

pop.calBeta(design)
```

Arguments

`design`	Object of class `analytic` (or inheriting from it) containing survey data and sampling design metadata. For function `pop.calBeta`, a *prepared* design object as returned by function `prep.calBeta`.
`model`	Formula (or list of formulas) specifying the linear model(s) whose regression coefficients will be used as calibration benchmarks (see 'Details').
`Beta`	Numeric vector (or list of numeric vectors) of regression coefficients to be used as calibration benchmarks (see 'Details').
`by`	If regression coefficients are known at domain level (i.e. within subpopulations), a formula specifying the domains (see 'Details'). Specify `NULL` (the default option) if the regression coefficients are known at the overall population level.
`partition`	In case domains have been specified through argument `by`, should a partitioned calibration task be performed? The default is `FALSE`, which selects a global calibration task.
`drop.z`	Can the prepared calibration variables (see 'Details') be dropped upon completion of the calibration task? Specify `TRUE` (the default) if you want to save memory space.

Details

***Special Purpose Calibration* tasks**

ReGenesees 2.1 introduced support for *'special purpose calibration'* tasks, i.e. facilities to calibrate survey weights so as to match complex population parameters, instead of ordinary population totals.

When calibration benchmarks come in the form of population parameters that are complex, *non-linear* functions of auxiliary variables (like, in the present case, multiple regression coefficients), calibration constraints have to be linearized first. This generates *synthetic linearized variables* z, which are the *actual calibration variables* the calibration algorithm will eventually work on. Typically, *control totals* for these synthetic linearized variables z will *all* be *zero*. See, e.g. [Lesage, 2011].

Put briefly:

- Function `prep.calBeta` generates and binds to `design` the *synthetic linearized variables* z needed for the calibration task.
- Then, given the *prepared* `design` object returned by `prep.calBeta`, function `pop.calBeta` generates the *control totals* data frame for those z variables.

Of course, once prepared, survey data and control totals returned by the above functions will be fed in input to function `e.calibrate`, which will run the calibration task.

Function `prep.calBeta()`
Function `prep.calBeta` makes it possible to:

1. Calibrate on regression coefficients of *different linear models*, each known at the *overall population level*.
2. Calibrate on regression coefficients of a *single linear model*, known possibly for *different subpopulations*.

Note that, as detailed below, the key argument to switch from case 1. to case 2. is `by`. For extensive illustration of both the above cases 1. and 2., please see the 'Examples' section.

The mandatory argument `model` specifies the linear model(s) whose regression coefficients will be used as calibration benchmarks. Use a `formula` object to specify a *single* linear model, or a `list` of `formula` objects to specify *several* different linear models. For details on model specification, see e.g. `lm`.
The `design` variables referenced by `model` formula(s) must be `numeric` or `factor` and must not contain any missing value (`NA`).

The mandatory argument `Beta` specifies the vector(s) of regression coefficients that will be used as calibration benchmarks. Use a `numeric` vector to specify regression coefficients of a *single* linear model, or a `list` of `numeric` vectors to specify regression coefficients of *several* different linear models. The `Beta` vector(s) must not contain any missing value (`NA`).

If argument `model` is passed as a list, argument `Beta` must be passed as a list too, and the `length` of both must be the same. If this is the case, `Beta` vectors will be *positionally* tied to linear model formulas contained in `model`.

Each vector of regression coefficients specified through `Beta` must be *consistent* with the corresponding `model`, namely match its model matrix columns (as generated using actual `design` data). Function `prep.calBeta` will check for this consistency and raise an informative error message in case of failure.
Note that, in order to ensure consistency with `model`, not only the `length`, but also the *order* of `Beta` elements *matters*. If in doubt, you can easily learn about the *right* ordering of `Beta` coefficients, given `model`, by calling function `svystatB` as follows: `svystatB(design,model)`. This will, of course, return the estimated regression coefficients of `model` *before* calibration.
Note that each `Beta` vector can have `names` or not. If a `Beta` vector has *names* that match the expected ones (given the corresponding `model` formula), but appear in a *different order*, then function `prep.calBeta` will suitably re-order them and inform you with a `warning` message.

As anticipated, argument `by` can be used to enable calibration on regression coefficients known at *domain level* (i.e. within subpopulations).

If passed, `by` must be a formula: for example, the statement `by=~B1:B2` defines as 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`. The `design` variables referenced by `by` (if any) should be of type `factor`, otherwise they will be coerced. Note that, to prevent obvious collinearity issues, the variables referenced by argument `by` must *not* appear in the input `model` formula: otherwise, the program will stop and print an error message.

If you specify domains through argument `by`, you will be allowed to specify *just a single* linear model through argument `model`. Instead, through argument `Beta`, you will have to specify domain-level vectors of regression coefficients. Therefore, `Beta` will be necessarily passed as a *list*. Note that, in this case, the `Beta` list must have as many components as the domains defined through argument `by`, and the two will be matched *positionally* by function `prep.calBeta`. Therefore, the *order* of elements in the `Beta` list *matters*. Specifically, `Beta` vectors must appear in the same order as the domains specified by argument `by`. You can easily learn about the *right* ordering of `by` domains (and hence `Beta` elements) by calling function `svystatB` as follows: `svystatB(design,model,by)`.

This will, of course, return the estimated regression coefficients of `model` within by domains *before* calibration.

Lastly, if you specify domains through argument `by`, you will be allowed to decide whether `design` should be prepared for a *global* or a *partitioned* calibration task (see `e.calibrate`). Recall that partitioned calibration tasks are computationally more efficient, but produce exactly the same results. Note that, if you select `partition = TRUE`, than the *calibration domains* used to split the global calibration task will be the same domains specified via argument `by`. More explicitly: function `e.calibrate` will eventually be called on `prep.calBeta`'s output object silently assuming `partition = by`.

The synthetic linearized variables z prepared by `prep.calBeta` will eventually be used by function `e.calibrate` to solve the calibration task. Argument `drop.z` allows you to instruct `e.calibrate` to drop these variables from its output object - or rather keep them within it - upon completion of the calibration task. The default has been set to `TRUE` to reduce memory usage. In case you want to be able to consistently trim weights after calibration via function `trimcal`, you *must* specify `drop.z = FALSE`.

Function `pop.calBeta()`
Given the *prepared* `design` object returned by `prep.calBeta`, function `pop.calBeta` generates the *control totals* data frame needed to run the calibration task. This dataframe suitably accomodates the control totals of the *synthetic linearized variables* z prepared by `prep.calBeta`.

Note that the data frame object returned by `pop.calBeta` is *already filled*, with control totals that are *all zero*, and it is ready to be directly passed to function `e.calibrate` (see the 'Examples' section).

Note, lastly, that printing this control totals data frame might not be very telling: to better understand its structure you should instead leverage function `pop.desc`, for which a method dedicated to *'special purpose calibration'* tasks is available (see the 'Examples' section).

Value

- For function `pop.calBeta`, an object of the same class as `design`, storing the freshly created *synthetic linearized variables* z as columns of its `$variables` slot (see the 'Examples' section).
- For function `pop.calBeta`, a *control totals* data frame for those z variables, with class `spc.pop` (see the 'Examples' section).

Linear Algebra Remark

Contrary to ordinary calibration tasks, the control totals of a 'special purpose calibration' task are typically all zero. Therefore, calibration constraints of such tasks - unlike ordinary ones - define a system of linear equations that is *homogeneous*. Homogeneous systems always admit the trivial zero solution, which, in calibration terms, would mean output weights that are identically zero (and thus statistically unacceptable). For this reason, the possibility that function `e.calibrate` ends up with a *false convergence* of the special purpose calibration task cannot, in general, be ruled out. Note that functions `e.calibrate` and `check.cal` are able to detect such false convergence events and warn the user about it.

A satisfactory countermeasure to this issue is to set calibration bounds to any interval that does not include zero (see argument `bounds` of `e.calibrate`).
A second, very attractive alternative would be to run the special purpose calibration task *jointly* with some ordinary calibration task, as the joint calibration constraints would then define a *non-homogeneous* system. This solution can be obtained straightforwardly using function `pop.fuse`.

Author(s)

Diego Zardetto

References

Lesage, E. (2011). *"The use of estimating equations to perform a calibration on complex parameters"*. Survey Methodology. 37.

See Also

`e.calibrate` to calibrate weights, `svystatB` to compute estimates and sampling errors of Multiple Regression Coefficients, `pop.desc` to obtain a natural language description of control totals (including those for *special purpose calibration* tasks), `pop.fuse` to fuse population totals prepared for ordinary and special purpose calibration tasks.

Examples

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

#############################################################################
# Calibrate on the regression coefficients of a *single model* known at the #
# *overall population level*                                                #
#############################################################################
# Suppose you know the coefficients of the following linear model, which you
# obtained fitting the model on some external source (e.g. Census or register
# data):
model0 <- y ~ emp.num*emp.cl

# Here, use the sbs sampling frame available in ReGenesees to simulate the
# external source and compute the values of the regression coefficients:
B0 <- coef(lm(model0, data = sbs.frame))
B0

# Now, prepare the survey design for calibration:
sbsdes0 <- prep.calBeta(design = sbsdes, model = model0, Beta = B0)

# Have a look at the freshly created *synthetic* auxiliary variables:
head(sbsdes0$variables)

# Then, prepare the control totals dataframe for the calibration task:
pop0 <- pop.calBeta(sbsdes0)

# Have a look...
class(pop0)
pop.desc(pop0)
# ...and note that all the control totals are zero!
pop0

# Lastly, calibrate:
sbscal0 <- e.calibrate(sbsdes0, pop0)

# Check that calibration estimates of regression coefficients now match the
```

```
# input B0 values derived from the external source:
svystatB(sbscal0, model0)
B0
# OK


#############################################################################
# Calibrate simultaneously on the regression coefficients of *different #
# models* known at the *overall population level*                       #
#############################################################################
# Suppose you know the coefficients of the following linear models, which you
# obtained fitting the models on some external sources
model1 <- va.imp2 ~ emp.num:emp.cl + nace.macro - 1
model2 <- y ~ dom3 - 1

# Here, use the sbs sampling frame available in ReGenesees to simulate the
# external sources and compute the values of the regression coefficients:
B1 <- coef(lm(model1, data = sbs.frame))
B2 <- coef(lm(model2, data = sbs.frame))

  ## First, just for illustration, calibrate only on B1 regression coefficients:
  sbsdes1 <- prep.calBeta(sbsdes, model = model1, Beta = B1)
  pop1 <- pop.calBeta(sbsdes1)
  sbscal1 <- e.calibrate(sbsdes1, pop1)

  # Check that calibration estimates of regression coefficients now match the
  # input B1 values derived from the external source...
  svystatB(sbscal1, model1)
  B1
  # ...but, of course, do *not* match those of B2:
  svystatB(sbscal1, model2)
  B2
  # OK

  ## Second, just for illustration, calibrate only on B2 regression coefficients:
  sbsdes2 <- prep.calBeta(sbsdes, model = model2, Beta = B2)
  pop2 <- pop.calBeta(sbsdes2)
  sbscal2 <- e.calibrate(sbsdes2, pop2)

  # Check that calibration estimates of regression coefficients now match the
  # input B2 values derived from the external source...
  svystatB(sbscal2, model2)
  B2
  # ...but, of course, do *not* match those of B1:
  svystatB(sbscal2, model1)
  B1
  # OK

## Now, calibrate *simultaneously* on *B1 and B2* regression coefficients
# Prepare the survey design for the joint calibration task:
sbsdes1_2 <- prep.calBeta(sbsdes, model = list(model1, model2), Beta = list(B1, B2))

# Prepare the control totals dataframe for the joint calibration task:
pop1_2 <- pop.calBeta(sbsdes1_2)

# Have a look to the control totals (note the presence of two models and
# Beta vectors):
```

```
pop.desc(pop1_2)
pop1_2

# Lastly, run the calibration:
sbscal1_2 <- e.calibrate(sbsdes1_2, pop1_2)

# Check that calibration estimates of regression coefficients now match *both*
# the B1 and B2 values derived from the external sources:
svystatB(sbscal1_2, model1)
B1
svystatB(sbscal1_2, model2)
B2
# OK


##############################################################################
# Calibrate simultaneously on the regression coefficients of a *single model* #
# known for *different subpopulations*                                        #
##############################################################################
# NOTE: In this case, both *global* and *partitioned* calibration tasks are
#       possible, and both will be illustrated below.

# Suppose you know the coefficients of the following linear model, which you
# obtained fitting the model on some external source (e.g. Census or register
# data) *within subpopulations* defined by some factor variable(s):
model <- va.imp2 ~ emp.num:emp.cl + nace.macro - 1

# Here, use the sbs sampling frame available in ReGenesees to simulate the
# external source and suppose the subpopulations are defined by variable 'dom3'.
# Thus, compute the values of the regression coefficients as follows:
B <- by(sbs.frame, sbs.frame$dom3, function(df) coef(lm(model, data = df)))
B

## Let's start with the *global* solution
# Prepare the survey design for the calibration task (note that the 'by'
# argument is used):
sbsdes.g <- prep.calBeta(sbsdes, model, Beta = B, by = ~dom3)

# Prepare the control totals for the calibration task:
pop.g <- pop.calBeta(sbsdes.g)

# Have a look to the control totals (note the presence of one Beta vector *for
# each domain*):
pop.desc(pop.g)
pop.g

# Run the calibration:
sbscal.g <- e.calibrate(sbsdes.g, pop.g)

# Check that calibration estimates of regression coefficients now match the
# input B values derived from the external source *for each domain*:
svystatB(sbscal.g, model, ~dom3)
B
# OK

## Let's proceed with the *partitioned* solution
# Prepare the survey design for the calibration task (note that 'by' and
```

```
# 'partition' arguments are used):
sbsdes.p <- prep.calBeta(sbsdes, model, Beta = B, by = ~dom3, partition = TRUE)

# Prepare the control totals for the calibration task:
pop.p <- pop.calBeta(sbsdes.p)

# Have a look to the control totals (note the presence of one Beta vector *for
# each domain*):
pop.desc(pop.p)
pop.p

# Run the calibration:
sbscal.p <- e.calibrate(sbsdes.p, pop.p)

# Check that calibration estimates of regression coefficients now match the
# input B values derived from the external source *for each domain*:
svystatB(sbscal.p, model, ~dom3)
B
# OK

## Lastly, check that calibration weights obtained using the *global* and
## *partitioned* solution are the same:
g.range(sbscal.g)
g.range(sbscal.p)
all.equal(weights(sbscal.g), weights(sbscal.p))
# OK


############################################################
# BONUS TIP: Calibration on the mean (or on domain means) #
#            of one variable or multiple variables.       #
############################################################
# Since the domain mean of a numeric variable can be thought as a
# regression coefficient (see the 'Examples' section of ?svystatB),
# you can use the ReGenesees facilities documented above to
# *calibrate on the mean (or on domain means)* of *one variable
# or multiple variables*.
# NOTE: The examples below cover the following cases of calibration on:
#       (i)   The overall mean of a single variable.
#       (ii)  The means of a single variable within domains of just
#             one type.
#       (iii) The domain means of several variables, with multiple
#             and different domain types.

# Load artificial household survey data and define a survey design:
data(data.examples)
exdes <- e.svydesign(data = example, ids = ~towcod + famcod,
                     strata = ~SUPERSTRATUM, weights = ~weight)
exdes <- des.addvars(exdes, ones = 1)

## CASE (i): Calibrate on the overall mean of a single variable
# Suppose you know with satisfactory accuracy the *average income* of your
# target population (but you do *not* have reliable information on the
# *total income*, nor on the *total number of individuals*):
income.AVG <- 1270

# You can calibrate on *average income* as follows:
```

```
exdes.new <- prep.calBeta(exdes, income ~ 1, Beta = income.AVG)
pop.new <- pop.calBeta(exdes.new)
excal.new <- e.calibrate(exdes.new, pop.new)

# Now, check that calibration estimate of average income now match the known
# value derived from the external source *without residual uncertainty*:
svystatTM(excal.new, ~income, estimator = "Mean")
income.AVG

# ...while there is *residual uncertainty* in the estimates of the numerator and
# denominator totals:
svystatTM(excal.new, ~income + ones, estimator = "Total")
# OK

## CASE (ii): Calibrate on the means of a single variable within domains
#             of just one kind
# You can calibrate on *domain means* along the lines illustrated above (note,
# however, that argument 'by' would provide an alternative way to achieve
# the same result).
# Suppose you know with satisfactory accuracy the *average income* by the
# crossclassification of sex and marital status:
income.AVG.sex.marstat <- c(f.married   = 1310, m.married   = 1260,
                            f.unmarried = 1150, m.unmarried = 1200,
                            f.widowed   = 1380, m.widowed   = 1300)

# Run the calibration on *average income by sex:marstat* as follows:
exdes.new <- prep.calBeta(exdes, income ~ sex:marstat -1,
                          Beta = income.AVG.sex.marstat)
pop.new <- pop.calBeta(exdes.new)
excal.new <- e.calibrate(exdes.new, pop.new)

# Now, check that calibration estimates of average income by domains now match
# the known values derived from the external source *without residual
# uncertainty*:
svystatTM(excal.new, ~income, ~sex:marstat, estimator = "Mean")
income.AVG.sex.marstat

# ...while there is *residual uncertainty* in the estimates of the numerator and
# denominator totals:
svystatTM(excal.new, ~income + ones, ~sex:marstat, estimator = "Total")
# OK

## CASE (iii): Calibrate on the domain means of several variables, with multiple
#              and different domain types.
# Suppose you know with satisfactory accuracy:
# - the average income by sex:
# - the average income by marstat:
# - the average of variable z by age (variable 'age5c', 5 classes):
income.AVG.sex <- c("f" = 1245, "m" = 1250)
income.AVG.marstat <- c("married" = 1260, "unmarried" = 1230, "widowed" = 1290)
z.AVG.age5c <- c("1" = 125, "2" = 130, "3" = 135, "4" = 125, "5" = 140)

# Run the calibration as follows:
exdes.new <- prep.calBeta(exdes, model = list(income ~ sex -1,
                                              income ~ marstat -1,
                                              z ~ age5c -1),
                                 Beta = list(income.AVG.sex,
```

```
                                              income.AVG.marstat,
                                              z.AVG.age5c)
                            )
pop.new <- pop.calBeta(exdes.new)
excal.new <- e.calibrate(exdes.new, pop.new)

# Now, check that calibration estimates match the known domain means derived
# from the external source:
svystatTM(excal.new, ~income, ~sex, estimator = "Mean")
income.AVG.sex
svystatTM(excal.new, ~income, ~marstat, estimator = "Mean")
income.AVG.marstat
svystatTM(excal.new, ~z, ~age5c, estimator = "Mean")
z.AVG.age5c
# OK
```

`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 four options for variance estimations which can be freely set and modified by the user:

- `RG.ultimate.cluster`
- `RG.lonely.psu`
- `RG.adjust.domain.lonely`
- `RG.warn.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). For without replacement sampling designs, this approach is known to underestimate the true multistage variance, while - at the same time - overestimating its true first-stage component. Anyway, the underestimation error becomes negligible if the PSUs' sampling fractions across strata are very small. When sampling with replacement, the Ultimate Cluster approach is no longer an approximation, but rather an exact result. Hence, be `options("RG.ultimate.cluster")` `TRUE` or `FALSE`, if one does not specify first-stage finite population corrections, **ReGenesees** will produce exactly the same variance estimates.

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 07] 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 purposes. 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 a single PSU with observations falling 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.

If `options("RG.warn.domain.lonely")` is set to `TRUE`, a warning message is raised whenever an estimation domain happens to contain just a single PSU belonging to a stratum. The default is `FALSE`.

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"*, https://CRAN.R-project.org/package=survey.

Wolter, K.M. (2007) *"Introduction to Variance Estimation"*, Second Edition, 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 SSUs in SR strata and PSUs 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:
old.op <- 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*

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 22 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)

dom3 An unplanned estimation domain, factor with 4 levels

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

Examples

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

`smooth.strat.jump` *Smooth Weights to Cope with Stratum Jumpers*

Description

Given a stratified one-stage unit sampling design object, this function smooths survey weights to mitigate estimation issues that may arise from stratum jumpers.

Usage

```
smooth.strat.jump(design, curr.strata, method = c("MinChange", "Beaumont"))
```

Arguments

`design`	Object of class `analytic` containing the weights to be smoothed. It must be a one-stage unit sampling design object. Moreover, it must be a non-calibrated object. See 'Details'.
`curr.strata`	Formula identifying the *current* strata variable, as observed at survey-time (see 'Details').
`method`	The smoothing method (see 'Details'). The default method `'MinChange'` smooths the weight of stratum jumpers almost without affecting the weights of other units. Method `'Beaumont'` adopts a much more aggressive smoothing strategy, which may result in significant modifications of the weights also for units that are *not* stratum jumpers (see 'Examples').

Details

In business surveys, stratum jumpers are sampling units (e.g. firms or establishments) whose stratum information observed at survey-time happens to differ from the stratum information that was available in the sampling frame at design-time.

If the current value of the strata variable (i.e. the one observed at survey-time) is reliable, stratum jumpers are evidence of frame imperfections (typically, the frame was not up-to-date). Empirically, stratum jumpers are often units that underwent a fast growth in size from sampling-time to survey-time.

In most enterprise surveys the sampling design is such that smaller firms receive a smaller inclusion probability (and hence a larger design weight). Therefore, stratum jumpers often have a "too large" weight, in the sense that they would have received a smaller weight had their actual size been known at sampling-time. When these units also happen to have a large value of interest variable `y`, they may become influential in estimation. Even though stratum jumpers may exist in household surveys too, business surveys are much more exposed to the risk that stratum jumpers unduly influence estimation, as their target populations are typically highly skewed with respect to many interest variables. As a consequence, in business surveys, stratum jumpers can result in inefficient (and, under some circumstances, even biased) design-based estimators.

Despite stratum jumpers are actually a concern only when both their design weight `w` and their `y` value (as measured at survey-time) conspire to yield an influential value of `w * y`, function `smooth.strat.jump` tries to mitigate their potential adverse impact by smoothing the weights *without* using any information on `y`. Such a choiche is driven by two major aims. First, the methodology must preserve the universality of the weights (i.e. the same weights must be used to compute estimates for whatever interest variables `y`). Second, the methodology should not require explicit modeling efforts and be easy enough to scale to production settings that need automated and replicable procedures.

Argument design identifies the survey design object that is (possibly) affected by stratum jumpers and contains the weights that will be smoothed. In case no stratum jumpers are found, the function will raise an error. Object design can only be a one-stage unit sampling design. Moreover, it must be a non-calibrated object. Should any of these conditions be false, the function would raise an error. Note, lastly, that function smooth.strat.jump will not smooth further weights that have already been smoothed. This is a deliberate design choice, devised to discourage over-smoothing and cosmetic adjustments of the survey weights.

Formula curr.strata defines the *current* strata variable, as observed at survey-time. This is different from the *design* strata variable used to build object design using function e.svydesign, as the latter was measured at sampling-time. Function smooth.strat.jump will flag as stratum jumpers all the units whose *current* stratum differs from the *design* one. Note that, in case *current* strata become available only after object design was created, you may use function des.addvars to add this new column to the old object.

The weight smoothing process entails two steps:

1. The weights are smoothed according to a given method (see below): w -> w1
2. The weights w1 of all units are scaled by a global factor so as to preserve the initial overall sum of weights: w1 -> w2 = scale * w1 with scale = sum(w)/sum(w1)

Argument method controls the smoothing algorithm. Two methods can be selected: 'MinChange' (the default) and 'Beaumont' (which implements the proposal of [Beaumont, Rivest 09]). Note that the methods only differ with respect to step 1., as step 2. is identical for both of them.

The step 1 working mechanism of these methods can be summarized as follows:

- **'MinChange'**

Only the weights of stratum jumpers are smoothed, by setting their value to the average weight of units belonging to the same *current* stratum. Weights of all other units are left unchanged.

- **'Beaumont'**

The weights of all units are smoothed, by setting their value to the average weight of units belonging to the same *current* stratum, with the exception of *minimum weight* units, whose weights are left unchanged. Therefore all weights, excluding only minimum weights within each current stratum, are smoothed. Note that this smoothing affects all current strata, even those that do *not* include any stratum jumper.

In summary, both methods often lead to very similar smoothed weights for units that are stratum jumpers. However method 'Beaumont' smooths the weights of all other (i.e. *non* stratum jumpers) units much more aggressively than method 'MinChange' (which only minimally alters them in step 2. to preserve the overall sum of weights). Moreover, method 'MinChange' treats stratum jumpers that grew in size and those that decreased in size on the same footing, whereas method 'Beaumont' typically smooths the weights of the former more than those of the latter (owing to its minimum weight preservation constraint, see 'Examples').

Note that every call to smooth.strat.jump generates, by side effect, a diagnostics data structure named strat.jump.status into the .GlobalEnv (see 'Examples'). This is a data frame with one row for each stratum jumper unit, whith the following columns:

```
Column                 Meaning
IDS....................Unit identifier
```

```
W.....................Initial weight
DES_STR...............Design stratum
DES_STR_W_AVG.........Average of initial weights within the design stratum
CURR_STR..............Current stratum
CURR_STR_W_AVG........Average of initial weights within the current stratum
N_JUMP_DES_CURR_STR....Number of stratum jumpers that jumped between the
                       design stratum and the current stratum (NOTE: in any
                       direction)
W_SMOOTH_UNSC.........Unscaled smoothed weight (as obtained after step 1)
W_SMOOTH..............Smoothed weight (scaled, as obtained after step 2)
```

Value

An object of the same class as `design`. The data frame it contains (stored in its `$variables` slot) includes the smoothed weights columns and a column that flags the stratum jumpers. The name of the smoothed weights column is obtained by pasting the name of the initial weights column with the string `".smooth"`. Stratum jumpers are identified by a new (`logical`) column named `is.jumper`.

Methodological Remark

Smoothing survey weights is a model-based approach, see [Beaumont 08] (e.g.`method = 'Beaumont'` basically models the smoothed weights as a function of the current strata using a one-way ANOVA model plus constraints). Therefore Horvitz-Thompson-like estimators that use smoothed weights - instead of design weights - cannot be guaranteed to be design-unbiased. Of course, the need to smooth the weights arises precisely because the existence of stratum jumpers already signals a departure from the ideal conditions of design-based inference.

As the design-unbiasedness of Horvitz-Thompson estimators in probability sampling rests on using design weights that are reciprocals of inclusion probabilities, smoothing methods that change the design weights the least appear preferable in a design-based perspective. For this reason, function `smooth.strat.jump` adopts the `'MinChange'` method by default. One could, nonetheless, argue that the `'Beaumont'` method could sometimes perform better (e.g. lead to more efficient estimates) from a model-based perspective.

Regardless the choice of argument `method`, in order to reduce any possible design-bias introduced by smoothing the weights, users are advised to *calibrate the smoothed weights* using any auxiliary information available from external sources that are *more up-to-date* than the sampling frame.

Author(s)

Diego Zardetto

References

Beaumont, J. F. (2008). A new approach to weighting and inference in sample surveys. Biometrika, 95(3), 539-553.

Beaumont, J. F., Rivest, L. P. (2009). Dealing with outliers in survey data. In Handbook of statistics (Vol. 29, pp. 247-279). Elsevier.

See Also

`e.svydesign` to bind survey data and sampling design metadata and `e.calibrate` for calibrating smoothed survey weights by leveraging auxiliary information that is more up-to-date than the sampling frame (a warmly suggested option).

Examples

```
##############################################################################
# Build [Beaumont, Rivest 09] example dataset, containing:                   #
# - a first stratum 'A' that, at survey time, contains one large weight stratum  #
#   jumper received from design-stratum 'B'                                  #
# - a second stratum 'B' that, at survey time, does not contain stratum jumpers  #
#                                                                            #
# and enhance it with:                                                       #
# - a third stratum 'C' that, at survey time, does not contain stratum jumpers   #
# - a fourth stratum 'D' that, at survey time, does not contain stratum jumpers  #
# - a fifth stratum 'E' that, at survey time, contains two stratum jumpers, both #
#   received from design-stratum 'D', one with small and one with medium weight  #
##############################################################################
BR <- data.frame(
       id = 1:90,
       des.strata =  factor(rep(c("A", "B", "C", "D", "E"), c(9, 41, 10, 20, 10))),
       curr.strata = factor(rep(c("A", "B", "C", "D", "E"), c(10, 40, 10, 19, 11))),
      w = c(rep(c(1, 31), c(9, 41)), c(27, 28, 22, 26, 11, 12, 13, 30, 17, 21), 2:21, 11:2)
    )
BR$curr.strata[61] <- "E"

# Have a look at the data:
BR

# Have a look at the jumps:
with(BR, table(des.strata, curr.strata))

# Use the BR data frame to build a one stage stratified unit sampling design:
BRdes <- e.svydesign(data=BR, ids=~id, strata=~des.strata, weights=~w)


## Now smooth the weights:
## Method: MinChange (the default)
M.smooth <- smooth.strat.jump(BRdes, ~curr.strata)
M.smooth

# Have a look at the new columns:
head(M.smooth$variables)

# Inspect the effects of smoothing on the stratum jumpers:
strat.jump.status

## Method: Beaumont
B.smooth <- smooth.strat.jump(BRdes, ~curr.strata, method = "Beaumont")
B.smooth

# Inspect the effects of smoothing on the stratum jumpers:
strat.jump.status

## As anticipated, smoothed weights of stratum jumpers are mostly similar for
## both methods. However the methods differ significantly when it comes to non
## stratum jumpers. This is clearly shown in the following plots.

## Plot 1 - START
opar <- par("mfcol" = c(1, 2))
```

```
# M.smooth
with(M.smooth$variables, plot(w, w.smooth, pch = c(19, 15)[1 + is.jumper],
     col = c("black", "red")[1 + is.jumper], cex = c(1,1.2)[1 + is.jumper],
     xlab = "Original Weights", ylab = "Smoothed Weights",
     main = "method: MinChange"))
abline(0:1, col = "limegreen", lwd = 2, lty = 2)

legend("topleft",
legend = c("Stratum Jumper", "Non Stratum Jumper"),
col = c("red", "black"),
pch = c(15, 19),
bty = "n",
text.col = "black",
inset = c(0.05, 0.05)
)

# B.smooth
with(B.smooth$variables, plot(w, w.smooth, pch = c(19, 15)[1 + is.jumper],
     col = c("black", "red")[1 + is.jumper], cex = c(1,1.2)[1 + is.jumper],
     xlab = "Original Weights", ylab = "Smoothed Weights",
     main = "method: Beaumont"))
abline(0:1, col = "limegreen", lwd = 2, lty = 2)

legend("topleft",
legend = c("Stratum Jumper", "Non Stratum Jumper"),
col = c("red", "black"),
pch = c(15, 19),
bty = "n",
text.col = "black",
inset = c(0.05, 0.05)
)

par(opar)
## Plot 1 - END

## Plot 2 - START
# M.smooth
with(M.smooth$variables, plot(w, pch = 20, col = curr.strata,
     ylab = "Original and Smoothed Weights",
     main = "method: MinChange \n(colors identify different current strata)"))
with(M.smooth$variables, points(w.smooth, pch = 0, col = curr.strata))

legend("topright",
legend = c("Original Weight", "Smoothed Weight"),
col = rep("grey", 2),
pch = c(20, 0),
text.col = "black",
inset = c(0.01, 0.01)
)

# B.smooth
with(B.smooth$variables, plot(w, pch = 20, col = curr.strata,
     ylab = "Original and Smoothed Weights",
     main = "method: Beaumont \n(colors identify different current strata)"))
with(B.smooth$variables, points(w.smooth, pch = 0, col = curr.strata))

legend("topright",
```

```
legend = c("Original Weight", "Smoothed Weight"),
col = rep("grey", 2),
pch = c(20, 0),
text.col = "black",
inset = c(0.01, 0.01)
)
## Plot 2 - END

# Although, as seen above, non-negligible differences in smoothed weights exist
# at unit level, both methods perform similarly in terms of strata averages...
## Initial weights, design strata:
with(BR, tapply(w, des.strata, mean))

## Initial weights, current strata:
with(BR, tapply(w, curr.strata, mean))

## Smoothed weights, current strata, method MinChange:
with(M.smooth$variables, tapply(w.smooth, curr.strata, mean))

## Smoothed weights, current strata, method Beaumont:
with(B.smooth$variables, tapply(w.smooth, curr.strata, mean))

# ...as expected
```

---

svystat *Compute Many Estimates and Errors in Just a Single Shot*

Description

Computes many estimates and errors (e.g. for disparate estimation domains) in just a single shot, primarily to use them in fitting GVF models. Can handle estimators of all kinds.

Usage

```
svystat(design, kind = c("TM", "R", "S", "SR", "B", "Q", "L"),
        by = NULL, group = NULL, forGVF = TRUE,
        combo = -1, ...)

## S3 method for class 'gvf.input.gr'
plot(x, ...)

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

Arguments

design	Object of class analytic (or inheriting from it) containing survey data and sampling design metadata.
kind	character specifying the summary statistics function to call: it may be 'TM' (i.e. svystatTM, the default), 'R' (i.e. svystatR), 'S' (i.e. svystatS), 'SR' (i.e. svystatSR), 'B' (i.e. svystatB), 'Q' (i.e. svystatQ), and 'L' (i.e. svystatL).
by	Formula specifying the variables that define the "estimation domains". If NULL (the default option) estimates refer to the whole population.
group	Formula specifying a partition of the population into "groups": the output will be returned separately for each group. If NULL (the default option) the output is returned as a whole.
forGVF	Select TRUE (the default) if you want to use the output to fit a GVF model. Otherwise, the output will be simply a set of summary statistics objects.
combo	An integer which is only meaningful if by is passed. Requests to compute outputs for all the domains determined by crossing the by variables *up to* a given order (see 'Details').
...	For function svystat, additional arguments to the summary statistic function implied by kind. Otherwise, further arguments passed to or from other methods.
x	The object of class gvf.input.gr to plot.
object	An object of class svystat.gr containing survey statistics.

Details

This function can compute *all* the summary statistics provided by **ReGenesees**, and is principally meant to return a lot of them in just a single shot.

If forGVF = TRUE the output will be ready to feed **ReGenesees** GVF fitting infrastructure, otherwise it will consist simply of a set of summary statistic objects.

Use argument kind to specify the summary statistic you need. The default value 'TM' selects function svystatTM, which yields Totals and Means. All the arguments needed by the summary statistic function implied by kind (e.g. argument y for svystatTM when kind = 'TM') will be passed on through argument '...'.

As usual in summary statistics, argument by can be used to request domain estimates.

The group formula (if any) specifies a way of partitioning the population into groups: the output will be reported separately for each group. In the GVF context, a *"grouped"* output will permit to fit *separate* GVF models inside different groups (and hence to compute separate variance predictions for different groups).

Note that group and by share identical syntax and semantics as model formulae, despite they have different purposes in function svystat (as explained above).

Parameter combo is *only* meaningful if by is passed. Its purpose is to allow computing estimates and errors simultaneously for many estimation domains.

If the by formula involves n variables, specifying combo = m requests to compute outputs for all the domains determined by all the interactions of by variables *up to* order m (with -1 <= m <= n), as follows:

```
COMBO          MEANING
m = -1.......'no combo', i.e. treat 'by' formula as usual (the default);
```

```
m =  0.......'order zero' combination, i.e. just a single domain:
             the whole population;
m =  1.......'order zero' plus 'order one' combinations, the latter being
             all the marginal domains defined by 'by' variables;
m =  n........combinations of any order, the maximum being the one with
             all 'by' variables interacting simultaneously.
```

The `plot` method can be used *only* when `forGVF = TRUE` and produces a matrix (or many matrices, if `group` is passed) of scatterplots with polynomial smoothers.

Methods `coef`, `SE`, `VAR`, `cv`, `deff`, and `confint` can be used *only* when `forGVF = FALSE`, to extract estimates and variability statistics.

Value

An object storing estimates and errors, whose detailed structure depends on input parameters' values.

If `forGVF = FALSE`, a set of summary statistics possibly stored into a list (with class `svystat.gr` in the most general case).

If `forGVF = TRUE` and argument `group` is *not* passed, an object of class `gvf.input`.

If `forGVF = TRUE` and argument `group` is passed, an object of class `gvf.input.gr`. This is a list of objects of class `gvf.input`, each one pertaining to a different population group.

Author(s)

Diego Zardetto

See Also

`estimator.kind` to assess what kind of estimates are stored inside a survey statistic object, `gvf.input` as an alternative to prepare the input for GVF model fitting, `GVF.db` to manage **ReGenesees** archive of registered GVF models, `fit.gvf` to fit GVF models, `plot.gvf.fit` to get diagnostic plots for fitted GVF models, `drop.gvf.points` to drop alleged outliers from a fitted GVF model and simultaneously refit it, and `predictCV` to predict CV values via fitted GVF models.

Examples

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

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


#############################################################################
# svystat as an alternative way to compute 'ordinary' summary statistics #
#############################################################################
## Total number of employees
svystat(sbsdes,y=~emp.num,forGVF=FALSE)
# equivalent to:
svystatTM(sbsdes,y=~emp.num)

## Average number of employees per enterprise
svystat(sbsdes,y=~emp.num,estimator="Mean",forGVF=FALSE)
# equivalent to:
```

```
svystatTM(sbsdes,y=~emp.num,estimator="Mean")

## Average value added per employee by economic activity macro-sector
## (nace.macro):
svystat(sbsdes,kind="R",num=~va.imp2,den=~emp.num,by=~nace.macro,forGVF=FALSE)
# equivalent to:
svystatR(sbsdes,num=~va.imp2,den=~emp.num,by=~nace.macro)

## Counts of employees by classes of number of employees (emp.cl) crossed
## with economic activity macro-sector (nace.macro):
svystat(sbsdes,y=~emp.num,by=~emp.cl:nace.macro,forGVF=FALSE)
# equivalent to:
svystatTM(sbsdes,y=~emp.num,by=~emp.cl:nace.macro)

## Provided forGVF = FALSE, you can use estimator.kind on svystat output:
stat<-svystat(sbsdes,kind="R",num=~va.imp2,den=~emp.num,by=~emp.cl:nace.macro,
      group=~region,forGVF=FALSE)
stat
estimator.kind(stat,sbsdes)


############################################################
# Understanding syntax and semantics of argument 'combo' #
############################################################
# Load household data:
data(data.examples)

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

# Add convenience variable 'ones' to estimate counts:
houdes<-des.addvars(houdes,ones=1)

## To facilitate understanding, let's for the moment keep forGVF = FALSE.
## Let's use estimates and errors of counts of individuals by sex and
## five age classes (age5c):
svystat(houdes,y=~ones,by=~age5c:sex,forGVF=FALSE)

## Now let's play with argument 'combo':
   # combo = -1
   # -> 'no combo', i.e. treat 'by' formula as usual
   svystat(houdes,y=~ones,by=~age5c:sex,forGVF=FALSE,combo=-1)

   # combo = 0
   # -> 'order zero' combination, i.e. just a single domain: the whole population
   svystat(houdes,y=~ones,by=~age5c:sex,forGVF=FALSE,combo=0)

   # combo = 1
   # -> 'order zero' plus 'order one' combinations, the latter being all the
   #    marginal domains defined by 'by' variables
   svystat(houdes,y=~ones,by=~age5c:sex,forGVF=FALSE,combo=1)

   # combo = 2
   # -> since 'by' has 2 variables, this means combinations of any order up to
   #    the maximum
   svystat(houdes,y=~ones,by=~age5c:sex,forGVF=FALSE,combo=2)
```

```
   # combo = 3
   # -> yields an error, as 'combo' cannot exceed the number of 'by' variables
   #    (2 in this example)
## Not run:
   svystat(houdes,y=~ones,by=~age5c:sex,forGVF=FALSE,combo=3)

## End(Not run)


#########################################################################
# svystat as an alternative way to prepare input data for GVF models #
#########################################################################
## The same estimates and errors of the last example above, now with
## forGVF = TRUE: note the different output data format
svystat(houdes,y=~ones,by=~age5c:sex,combo=2)

## Note that the agile command above is indeed equivalent to the following
## lengthier, cumbersome statement:
gvf.input(houdes,
          svystatTM(houdes,y=~ones),
          svystatTM(houdes,y=~ones,by=~age5c),
          svystatTM(houdes,y=~ones,by=~sex),
          svystatTM(houdes,y=~ones,by=~age5c:sex)
          )


  ###############################################
  # Using argument 'group' to prepare input data #
  # for separate GVF models                      #
  ###############################################
  ## The same estimates and errors of the last example above, now prepared
  ## separately for different regions (regcod):
  svystat(houdes,y=~ones,by=~age5c:sex,combo=2,group=~regcod)

  ## Again the same estimates and errors, prepared separately for groups
  ## defined crossing marital status (marstat) and region:
  svystat(houdes,y=~ones,by=~age5c:sex,combo=2,group=~marstat:regcod)

  ## NOTE: Output has class "gvf.input.gr". This will tell ReGenesees' GVF
  ##       fitting facilities to handle estimates and errors pertaining to
  ##       different groups independently of each other.


## NOTE: Parameter combo allows svystat to gather a huge amount of estimates and
##       errors in just a single slot, as the number of estimation domains grows
##       exponentially with the number of by variables.
##       See, for instance, the following example:
out <- svystat(houdes,y=~ones,by=~age5c:marstat:sex:regcod,combo=4)
dim(out)
head(out)
plot(out)


#################################################
# Minor details: accessor functions and plotting #
#################################################
```

```
  ## Accessor functions work only when forGVF = FALSE
# Average value added per employee by nace.macro:
out <- svystat(sbsdes,kind="R",num=~va.imp2,den=~emp.num,by=~nace.macro,forGVF=FALSE)
out
# Access CV values and confidence intervals:
cv(out)
confint(out)

# The same as above, separately for regions:
out <- svystat(sbsdes,kind="R",num=~va.imp2,den=~emp.num,by=~nace.macro,group=~region,forGVF=FALSE)
out
# Access CV values and confidence intervals:
cv(out)
confint(out)


  ## Plot function works only when forGVF = TRUE
# Counts of individuals by sex, marstat and age5c, and all their interactions:
out <- svystat(houdes,y=~ones,by=~age5c:marstat:sex,combo=3)
# Plot GVF input:
plot(out)

# The same as above, grouped by region:
out <- svystat(houdes,y=~ones,by=~age5c:marstat:sex,combo=3,group=~regcod)
# Plot GVF inputs, separately by groups (regions):
plot(out)
```

---

svystatB *Estimation of Population Regression Coefficients in Subpopulations*

Description

Computes estimates, standard errors and confidence intervals for Multiple Regression Coefficients in subpopulations.

Usage

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

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

```
confint(object, ...)
## S3 method for class 'svystatB'
summary(object, ...)
```

Arguments

design	Object of class analytic (or inheriting from it) containing survey data and sampling design metadata.
model	Formula specifying the linear model whose coefficients have to be estimated.
by	Formula specifying the variables that define the "estimation domains" (see 'Details'). 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 (see 'Details').
object	An object of class svystatB.
...	Additional arguments to coef, ..., confint methods (if any).

Details

This function computes weighted estimates for Multiple Regression Coefficients 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 model identifies the regression model whose population coefficients have to be estimated (for details on model specification, see e.g. lm). The design variables referenced by model should be numeric or factor (variables of other types - e.g. character - will need to be converted in advance, e.g. using function des.addvars).

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 svystatB 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. Note that, to prevent obvious collinearity issues, the variables referenced by argument by must *not* appear in the input model formula: otherwise, the program will stop and print an error message.

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`.

Missing values (`NA`) in model variables should be avoided. If `na.rm=FALSE` (the default) they generate an error. If `na.rm=TRUE`, observations containing NAs in model variables are dropped, and estimates gets computed on non missing values only. This implicitly assumes that missing values hit interest variables *completely at random*: should this not be the case, computed estimates would be *biased*.

The `summary` method invoked on regression coefficients (say `b`) estimated via `svystatB`, gives p-values and significance codes for the component-wise test `b = 0`. Such values are computed assuming that the distribution of the regression coefficients estimators is normal (which is asymptotically true for large scale surveys). This assumption has the advantage of overcoming the problem of choosing the "right" statistic and assessing its "right" number of degrees of freedom when using data from a complex survey (see e.g. [Korn, Graubard 1990]).

Value

An object inheriting from the `data.frame` class, whose detailed structure depends on input parameters' values. In special cases (see Section 'Collinearity, Aliasing and Impacts in Domain Estimation'), a *list* object.

Collinearity, Aliasing and Impacts in Domain Estimation

Function `svystatB` overcomes problems arising from exact collinearity between model variables via *'aliasing'* (see the 'Examples' Section). Put simply, aliasing discards redundant (i.e. collinear) regressors, yielding exact estimates and standard errors for non-aliased regression coefficients (namely, the same results that would be obtained with a reduced - no collinearity - model). Note that for the way aliasing works, the order of the terms in the linear model formula definitely matters.

Collinearity between variables may manifest itself in subsets of the sample, and with different patterns across subsets. In domain estimation, this phenomenon can have an impact on the structure of `svystatB`'s output. In fact, owing to aliasing, the estimable regression coefficients - for the same input linear model - can be different across domains. In such cases, the domain estimates produced by `svystatB` can no longer be stored in a `data.frame`, and the output object will instead be a *list* (see the 'Examples' Section). Note that for these `svystatB`'s return objects (whose `class` is `svystatB.by.list`) no variance `extractors` are currently available.

Note also that, for the reasons above, the usage of `svystatB` via function `svystat` is restricted: it is not allowed to specify `svystat`'s arguments `by` and `group` when `kind == "B"`.

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.

Korn, E.L., Graubard, B.I. (1990) *"Simultaneous testing of regression coefficients with complex survey data: Use of Bonferroni t statistics"*. The American Statistician, 44, 270-276.

See Also

Estimators of Totals and Means svystatTM, Ratios between Totals svystatR, Shares svystatS, Ratios between Shares svystatSR, Quantiles svystatQ, Complex Analytic Functions of Totals and/or Means svystatL, and all of the above svystat.

Examples

```
#######################################################
# A simple regression model with a single predictor. #
# Let's compare the estimated regression coefficient #
# to its true value computed on the sampling frame.  #
#######################################################

# Load sbs data:
data(sbs)

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

# The population scatterplot of y vs emp.num reveals a linear
# behaviour:
plot(sbs.frame$emp.num,sbs.frame$y,
     col=rgb(50,205,50,100,maxColorValue=255),pch=16)

# Compute the population fit of the linear regression
# model y~emp.num-1 (no intercept):
pop.fit<-lm(y~emp.num-1,data=sbs.frame)
abline(pop.fit,col="red",lwd=2,lty=2)

# The obtained population R-squared is quite significant
# (greater than 0.7):
pop.R2<-summary(pop.fit)$r.squared
pop.R2

# The population regression coefficient is:
B<-coef(pop.fit)
B

# Now let's estimate B on the basis of the sbs sample and
# let's build a 95% confidence interval for the obtained estimate:
svystatB(sbsdes,y~emp.num-1,conf.int=TRUE)

# Thus, the confidence interval covers the true value of B.

# Notice that using ReGenesees Complex Estimators function
# svystatL, you would have obtained exactly the same results:
sbsdes<-des.addvars(sbsdes,y4emp.num=y*emp.num,
                           emp.num.sq=emp.num^2)
svystatL(sbsdes,expression(y4emp.num/emp.num.sq),
         conf.int=TRUE)
```

```
##################################
# A multiple regression example. #
##################################

# Let's estimate the coefficients of a model describing
# value added (variable va.imp2) as a linear function
# of number of employees by region and of nace.macro:
b <- svystatB(sbsdes,va.imp2~emp.num:region+nace.macro,vartype="cvpct")
b

# To obtain p-values and significance codes for the
# component-wise test t=0, you can exploit the
# summary method:
summary(b)

# Notice that estimators normality is assumed.


########################################
# Obtaining domain means via regression. #
########################################

# The domain mean of a numeric variable can be thought
# as a regression coefficient. Suppose you need the
# average number of employees by macro-sector, you can
# do as follows:
svystatB(sbsdes,emp.num~nace.macro-1)

# ...which, indeed, gives exactly the same results of:
svystatTM(sbsdes,y=~emp.num,by=~nace.macro,estimator="Mean")


#########################
# Handling collinearity. #
#########################

# Function svystatB overcomes problems arising from exact
# collinearity between model variables via 'aliasing'.
# To understand how aliasing works, let's build a manifestly
# redundant linear model:
svystatB(sbsdes,y~emp.num+I(2*emp.num)+I(3*va.imp2)+va.imp2-1)

# The obtained warning message shows that order definitely matters
# in aliasing, indeed:
svystatB(sbsdes,y~emp.num+I(2*emp.num)+va.imp2+I(3*va.imp2)-1)

# Notice also that aliasing gives exact estimates and standard errors
# for non-aliased regression coefficients (i.e. the same results that
# would be obtained with a reduced - no collinearity - model):
svystatB(sbsdes,y~emp.num+va.imp2-1)


#############################################
# Handling missing values in model variables. #
#############################################
```

```
# Load fpcdat:
data(fpcdat)

# Now, let's introduce some NAs in survey data:
fpcdat$y[c(1,3)]<-NA
fpcdat$x[c(3,5)]<-NA

# Create a design object:
fpcdes<-e.svydesign(data=fpcdat,ids=~psu+ssu,strata=~stratum,weights=~w,
        fpc=~fpc1+fpc2)

# Let's estimate regression coefficients of model z~y+x
  # na.rm=FALSE (the default) leads to an error:
  ## Not run:
  svystatB(fpcdes,z~y+x)

## End(Not run)

  # whereas na.rm=TRUE simply drops all the cases
  # with missing data in model variables:
  svystatB(fpcdes,z~y+x,na.rm=TRUE)


###################################
# Handling non positive weights. #
###################################

# Non positive direct weights are not allowed, anyway some
# calibrated weights can sometimes turn out to be <= 0. The
# corrisponding observations would be dropped by svystatB.

  # Prepare a template for population totals:
  pop<-pop.template(fpcdes,~z+pl.domain-1)

  # Fill it with silly values in order to obtain some negative g-weights:
  pop[1,]<-c(20000,90,10,90)

  # Calibrate:
  fpccal<-e.calibrate(fpcdes,pop)

  # We got 2 negative calibrated weights:
  g.range(fpccal)
  sum(weights(fpccal)<=0)

  # Now, let's estimate regression coefficients of model z~y+x
  # and pay attantion to the warnings:
  svystatB(fpccal,z~y+x,na.rm=TRUE)


#####################################################################
# Domain estimates of simple and multiple regression coefficients. #
#####################################################################

# Estimate the coefficients of the simple regression y ~ emp.num by domains
# obtained crossing region and nace.macro:
bb <- svystatB(sbsdes, model= va.imp2 ~emp.num, by= ~region:nace.macro)
```

```
bb

# Obtain p-values and significance codes via the summary method:
summary(bb)

# You have yet another method to estimate domain means of numeric variables.
# Suppose you need the average number of employees by macro-sector, you can
# do as follows:
svystatB(sbsdes, model= emp.num ~1, by= ~nace.macro)

# ...which gives exactly the same results of:
svystatB(sbsdes, model= emp.num ~nace.macro -1)

# ...and, of course, of:
svystatTM(sbsdes, y= ~emp.num, by= ~nace.macro, estimator= "Mean")


# One multiple regression example:
svystatB(sbsdes, model = y ~ va.imp2 + emp.num, by = ~region:nace.macro)


# A case of differential aliasing across domain (note the warning messages).
# A list-like object is returned:
svystatB(sbsdes, va.imp2 ~emp.num:emp.cl + nace.macro, by= ~region:public)
```

---

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 (see 'Details').
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 mathematical expression of a Complex Estimator, as specified by argument expr, can involve *'parameters'*, that is *symbols* representing given, non-random, scalar, numeric *values*. For each parameter appearing in expr, the value corresponding to its symbol will be searched following R standard scoping rules, see e.g. the first example in Section 'Examples' for a practical illustration.

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`.

Missing values (`NA`) in interest variables should be avoided. If `na.rm=FALSE` (the default) they generate NAs in estimates (or even an error, if `design` is calibrated). If `na.rm=TRUE`, observations containing NAs are dropped, and estimates gets computed on non missing values only. This implicitly assumes that missing values hit interest variables *completely at random*: should this not be the case, computed estimates would be *biased*.

Value

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

Warning

When the linearized variable corresponding to a Complex Estimator is ill defined (because the estimator gradient is singular at the Taylor series expansion point), SE estimates returned by `svystatL` are `NaN`.

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`, Shares `svystatS`, Ratios between Shares `svystatSR`, Multiple Regression Coefficients `svystatB`, Quantiles `svystatQ`, and all of the above `svystat`.

Examples

```
####################################################
# A first example: the Ratio Estimator of a Total.  #
####################################################
```

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

# Recall that ratio estimators of Totals rely on auxiliary
# information. Thus, suppose you want to estimate the total
# of income and suppose you know from an external source that
# the population size is, say, 1E6:
pop <- 1E6

# To obtain the ratio estimator of total income, you can do as follows:
  ## A) Directly plug the numeric value of pop into expr
svystatL(des, expression(1E6 * (income/ones)), vartype = "cvpct")

  ## B) Treat pop as a parameter and let R find its actual value (1E6) inside
  ##    the calling environment of svystatL (the .GlobalEnv)
svystatL(des, expression(pop * (income/ones)), vartype = "cvpct")

# NOTE: Method B) can be very useful for simulation purposes, as it avoids
#       having to directly type in numbers when invoking svystatL (something
#       that only a human in an interactive R session could do).

# By comparing the latter result with the ordinary estimator of the mean...
svystatTM(des,~income,vartype="cvpct")

# ...one can appreciate the variance reduction stemming from the correlation
# between numerator and denominator:
Corr(des, expression(income), expression(ones))


############################################################
# A complex example: estimation of the Population Standard #
# Deviation of a variable.                                 #
############################################################

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

# Suppose you want to estimate the standard deviation of the
# population distribution of value added (va.imp2):
sbsdes<-des.addvars(sbsdes,va.imp2.sq=va.imp2^2)
svystatL(sbsdes,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)
```

```
##############################################
# Estimation of Geometric and Harmonic Means. #
##############################################

## 1. Harmonic Mean
 # Recall that the the harmonic mean of a positive variable,
 # say z, can be computed as 1/mean(1/z). Thus, for instance,
 # to get a survey estimate of the harmonic mean of emp.num,
 # you can do as follows:
sbsdes<-des.addvars(sbsdes,emp.num.m1=1/emp.num)
h<-svystatL(sbsdes,expression( ones/emp.num.m1 ),
            conf.int=TRUE)
h

 # You can easily verify that the obtained estimate is close
 # to the true value (as computed from the sampling frame) and
 # covered by the 95% confidence interval:
1/mean(1/sbs.frame$emp.num)

## 2. Geometric Mean
 # Recall that the the geometric mean of a non negative variable,
 # say z, can be computed as exp(mean(log(z))). Thus, for instance,
 # to get a survey estimate of the geometric mean of emp.num,
 # you can do as follows:
sbsdes<-des.addvars(sbsdes,log.emp.num=log(emp.num))
g<-svystatL(sbsdes,expression( exp(log.emp.num/ones) ),
            conf.int=TRUE)
g

 # You can easily verify that the obtained estimate is close
 # to the true value (as computed from the sampling frame) and
 # covered by the 95% confidence interval:
exp(mean(log(sbs.frame$emp.num)))

## 3. Comparison with the arithmetic mean
 # If you compute the arithmetic mean estimate:
a<-svystatTM(sbsdes,~emp.num,estimator="Mean")
a

 #...you easily verify the expected hierachy,
 # i.e. harmonic <= geometric <= arithmetic:
H<-coef(h)
G<-coef(g)
A<-coef(a)
stopifnot(H <= G && G <= A)


#####################################################################
# Further complex examples: estimation of Population Regression #
# Coefficients (for a model with a single predictor).           #
#####################################################################

# Suppose you want to estimate of the slope of the population
# regression y vs. emp.num. You can do as follows:

  ## 1. No intercept model: y ~ emp.num - 1
   # Get survey estimate:
```

```
  sbsdes<-des.addvars(sbsdes,y4emp.num=y*emp.num,
                              emp.num.sq=emp.num^2)
  svystatL(sbsdes,expression(y4emp.num/emp.num.sq),
           conf.int=TRUE)

  # Compare with the actual slope from the population fit:
  pop.fit<-lm(y~emp.num-1,data=sbs.frame)
  coef(pop.fit)

  # ...a very good agreement.

 ## 2. The model with intercept: y ~ emp.num
  # Get survey estimate:
  svystatL(sbsdes,expression(  (ones*y4emp.num  - y*emp.num)/
                               (ones*emp.num.sq - emp.num^2)
                             ),
           conf.int=TRUE)

  # Compare with the actual slope from the population fit:
  pop.fit<-lm(y~emp.num,data=sbs.frame)
  coef(pop.fit)

  # ...again a very good agreement.

# Notice that both results above could be obtained also
# by using ReGenesees specialized function svystatB:

  ## 1.
  svystatB(sbsdes,y~emp.num-1,conf.int=TRUE)

  ## 2.
  svystatB(sbsdes,y~emp.num,conf.int=TRUE)

# Notice also - incidentally - that the estimate of the intercept
# turns out to be less accurate than the one we obtained for the slope,
# with about a 6% overestimation.
```

---

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 variable of interest? The default is FALSE (see 'Details').
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 svystatQ 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.

Missing values (NA) in interest variables should be avoided. If na.rm=FALSE (the default) they generate NAs in estimates (or even an error, if design is calibrated). If na.rm=TRUE, observations containing NAs are dropped, and estimates gets computed on non missing values only. This implicitly assumes that missing values hit interest variables *completely at random*: should this not be the case, computed estimates would be *biased*.

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, Shares svystatS, Ratios between Shares svystatSR, Multiple Regression Coefficients svystatB, Complex Analytic Functions of Totals and/or Means svystatL, and all of the above svystat.

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 ratios.
den	Formula defining the denominator variables for the ratios.
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 (see 'Details').
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 Ratios: 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`.

Missing values (`NA`) in interest variables should be avoided. If `na.rm=FALSE` (the default) they generate NAs in estimates (or even an error, if `design` is calibrated). If `na.rm=TRUE`, observations containing NAs are dropped, and estimates gets computed on non missing values only. This implicitly assumes that missing values hit interest variables *completely at random*: should this not be the case, computed estimates would be *biased*. Notice that the `na.rm=TRUE` option is only allowed for a single Ratio, i.e. if `num` and `den` reference a *single* interest variable.

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` estimates are either `NaN` or `Inf`, and `NaN` is returned for the corresponding SE 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`, Shares `svystatS`, Ratios between Shares `svystatSR`, Multiple Regression Coefficients `svystatB`, Quantiles `svystatQ`, Complex Analytic Functions of Totals and/or Means `svystatL`, and all of the above `svystat`.

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)


################################################
# Household-level averages in household surveys. #
################################################

# For an introduction on this topic, see ?svystatTM examples.

  # Load survey data:
  data(data.examples)

  # Define the survey design (variable famcod identifies households)
  exdes<-e.svydesign(data=example,ids=~towcod+famcod,strata=~stratum,
         weights=~weight)

  # Collapse strata to eliminate lonely PSUs
  exdes<-collapse.strata(design=exdes,block.vars=~sr:procod)

  # Now add new convenience variables to the design object:
    ## 'ones':       to estimate individuals counts
    ## 'housize':    to classify individuals by household size
    ## 'houdensity': to estimate households counts
  exdes<-des.addvars(exdes,
                     ones=1,
                     housize=factor(ave(famcod,famcod,FUN = length)),
                     houdensity=ave(famcod,famcod,FUN = function(x) 1/length(x))
                    )

  # Estimate the average number of household components by region:
  svystatR(exdes,num=~ones,den=~houdensity,by=~regcod,
           vartype="cvpct",conf.int=TRUE)

  # Estimate the average household income for the whole population:
  svystatR(exdes,num=~income,den=~houdensity,vartype="cvpct",
           conf.int=TRUE)

  # ...for household size categories:
  svystatR(exdes,num=~income,den=~houdensity,by=~housize,
           vartype="cvpct",conf.int=TRUE)
```

```
# ...and for province and household size:
svystatR(exdes,num=~income,den=~houdensity,by=~housize:procod,
         vartype="cvpct")
```

---

svystatS *Estimation of Shares in Subpopulations*

Description

Calculates estimates, standard errors and confidence intervals for Shares of a numeric variable within subpopulations.

Usage

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

## S3 method for class 'svystatS'
coef(object, ...)
## S3 method for class 'svystatS'
SE(object, ...)
## S3 method for class 'svystatS'
VAR(object, ...)
## S3 method for class 'svystatS'
cv(object, ...)
## S3 method for class 'svystatS'
deff(object, ...)
## S3 method for class 'svystatS'
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.
classes	Formula defining the population groups whose y shares must be estimated.
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 (see 'Details').
object	An object of class svystatS.
...	Additional arguments to coef, ..., confint methods (if any).

Details

This function computes weighted estimates for Shares of a numeric 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 Taylor linearization technique.

Shares are a special case of Ratios. Therefore, at the price of some additional (and possibly heavy) data preparation effort, shares could also be estimated using function svystatR. However, svystatS makes estimation by far easier, in particular when shares have to be estimated for many population groups and/or within many domains.

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

The mandatory argument classes identifies population groups whose shares of y have to be estimated. The design variables referenced by classes must be of class factor. Groups can be identified by crossing factors, e.g. statement classes = ~C1:C2 selects as groups the subpopulations determined by crossing the levels of factors C1 and C2.

The optional argument by specifies the variables defining the "estimation domains", that is the subpopulations within which shares of y by classes must be estimated. If by=NULL (the default option), the estimates produced by svystatS refer to the whole population. Estimation domains must be defined by a formula: for instance 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 svystatS 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.

Missing values (NA) in interest variables should be avoided. If na.rm=FALSE (the default) they generate NAs in estimates (or even an error, if design is calibrated). If na.rm=TRUE, observations containing NAs are dropped, and estimates gets computed on non missing values only. This implicitly assumes that missing values hit interest variables *completely at random*: should this not be the case, computed estimates would be *biased*.

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, Ratios between Shares svystatSR, Multiple Regression Coefficients svystatB, Quantiles svystatQ, Complex Analytic Functions of Totals and/or Means svystatL, and all of the above svystat.

Examples

```
# Load household data:
data(data.examples)

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

# Add convenience variable 'ones' to estimate counts:
des<-des.addvars(des,ones=1)

### Simple examples to illustrate the syntax:
# Shares of income for sex classes:
svystatS(des, y=~income, classes=~sex, vartype="cvpct")

# Shares of income for sex and 5 age classes:
svystatS(des, y=~income, classes=~age5c:sex, vartype="cvpct")

# Shares of income for sex classes within region domains:
svystatS(des, y=~income, classes=~sex, by=~regcod, vartype="cvpct")

# Shares of income for sex classes within domains defined by crossing region and
# 5 age classes:
svystatS(des, y=~income, classes=~sex, by=~age5c:regcod, vartype="cvpct")

# MARGINAL, CONDITIONAL and JOINT relative frequencies (see also ?svystatTM)
  # MARGINAL: e.g. proportions of people by provinces:
  svystatS(des, y=~ones, classes=~procod, vartype="cvpct")
  # CONDITIONAL: e.g. proportions of people by sex within provinces:
  svystatS(des, y=~ones, classes=~sex, by=~procod, vartype="cvpct")
  # JOINT: e.g. proportions of people cross-classified by sex and procod:
  svystatS(des, y=~ones, classes=~sex:procod, vartype="cvpct")


### One more complicated example:
#########################################################
# Shares of income held by people for income quintiles #
#########################################################
    # First: estimate income quintiles
```

```
inc.Q5 <- svystatQ(des, y=~income, probs=seq(0.2, 0.8, 0.2), ties="rounded")
inc.Q5

    # Second: add a convenience factor variable classifying people by income
    # quintiles
des<-des.addvars(des, quintile = cut(income, breaks = c(0, coef(inc.Q5), Inf),
                                     labels = 1:5, include.lowest=TRUE)
                 )

    # Third: estimate income shares by income quintiles
svystatS(des, y=~income, classes=~quintile, vartype="cvpct")

### NOTE: Procedure above yields *correct point estimates* of income shares by
###       income quintiles, while *variance estimation is approximated* since
###       we neglected the sampling variability of the estimated quintiles.
```

`svystatSR` *Estimation of Share Ratios in Subpopulations*

Description

Calculates estimates, standard errors and confidence intervals for Ratios between Shares of a numeric variables in subpopulations.

Usage

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

## S3 method for class 'svystatSR'
coef(object, ...)
## S3 method for class 'svystatSR'
SE(object, ...)
## S3 method for class 'svystatSR'
VAR(object, ...)
## S3 method for class 'svystatSR'
cv(object, ...)
## S3 method for class 'svystatSR'
deff(object, ...)
## S3 method for class 'svystatSR'
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.
`classes`	Formula defining the population groups among which ratios of `y` shares must be estimated.

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 (see 'Details').
object	An object of class svystatSR.
...	Additional arguments to coef, ..., confint methods (if any).

Details

This function computes weighted estimates for Ratios between Shares of a numeric 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 Taylor linearization technique.

Ratios of Shares are a special case of Ratios. Therefore, at the price of some additional and heavy data preparation effort, ratios of shares could also be estimated using function svystatR. However, svystatSR makes estimation by far easier, in particular when share ratios have to be estimated for many population groups and/or within many domains.

The mandatory argument classes identifies population groups whose ratios of y shares have to be estimated. Note that ratios of shares will be estimated and returned *for all the ordered pairs of population groups* defined by classes. Therefore, if classes defines G groups, svystatSR will have to compute estimates and sampling errors for G * (G -1) share ratios. To prevent combinatorial explosions (e.g. G = 20 would generate 380 share ratios), classes formula can reference just a *single* design variable, which must be a factor.

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 svystatSR 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 svystatSR 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`.

Missing values (`NA`) in interest variables should be avoided. If `na.rm=FALSE` (the default) they generate NAs in estimates (or even an error, if `design` is calibrated). If `na.rm=TRUE`, observations containing NAs are dropped, and estimates gets computed on non missing values only. This implicitly assumes that missing values hit interest variables *completely at random*: should this not be the case, computed estimates would be *biased*.

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`, Shares `svystatS`, Multiple Regression Coefficients `svystatB`, Quantiles `svystatQ`, Complex Analytic Functions of Totals and/or Means `svystatL`, and all of the above `svystat`.

Examples

```
# Load household data:
data(data.examples)

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

# Add convenience variable 'ones' to estimate counts:
des<-des.addvars(des,ones=1)

### Simple examples to illustrate the syntax:
# Population sex ratios:
svystatSR(des, y=~ones, classes=~sex, vartype="cvpct")

# Population sex ratios within provinces:
svystatSR(des, y=~ones, classes=~sex, by=~procod, vartype="cvpct")

# Ratios of population shares for 5 age classes:
# NOTE: This yields 5*(5-1)=20 ratios
svystatSR(des, y=~ones, classes=~age5c, vartype="cvpct")
```

```
### One more complicated example:
#########################################################################
# Ratios between shares of income held by people for income quintiles #
#########################################################################
    # First: estimate income quintiles
inc.Q5 <- svystatQ(des, y=~income, probs=seq(0.2, 0.8, 0.2), ties="rounded")
inc.Q5

    # Second: add a convenience factor variable classifying people by income
    # quintiles
des<-des.addvars(des, quintile = cut(income, breaks = c(0, coef(inc.Q5), Inf),
                                     labels = 1:5, include.lowest=TRUE)
                 )

    # Third: estimate income shares by income quintiles
QS5 <- svystatSR(des, y=~income, classes=~quintile, vartype="cvpct")
QS5

### Therefore, for instance, the *S80/S20 income quintile share ratio* is:
S80.20 <- QS5["quintile5/quintile1",]
S80.20

### NOTE: Procedure above yields *correct point estimates* of income quintile
###       share ratios, while *variance estimation is approximated* since
###       we neglected the sampling variability of the estimated quintiles.
```

---

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 (see 'Details').
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.

Missing values (NA) in interest variables should be avoided. If na.rm=FALSE (the default) they generate NAs in estimates (or even an error, if design is calibrated). If na.rm=TRUE, observations containing NAs are dropped, and estimates gets computed on non missing values only. This implicitly assumes that missing values hit interest variables *completely at random*: should this not be the case, computed estimates would be *biased*. Notice that the na.rm=TRUE option is only allowed if y references a *single* interest variable.

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, Shares svystatS, Ratios between Shares svystatSR, Quantiles svystatQ, Multiple Regression Coefficients svystatB, Complex Analytic Functions of Totals and/or Means svystatL, and all of the above svystat.

Examples

```
# Load survey data:
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)

# MARGINAL relative frequency distributions
  # Estimation of the relative frequency distribution of the qualitative
  # variable age5c for the whole population:
  svystatTM(des,~age5c,estimator="Mean")

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

# JOINT relative frequency distributions
  # 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")
    # *Third Solution* (exploiting estimators of Shares, see also ?svystatS):
    # Add new variable 'ones' to estimate counts of final units (individuals)
    # and estimate the share of people for classes of sex and marstat
    des2 <- des.addvars(des, ones=1)
    svystatS(des2,~ones,classes=~sex:marstat)


# 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)


# Estimating totals in domains for "incomplete" partitions: more on
# the AsIs operator I()

  # Estimation of the total income (plus cvpct) ONLY in region 7:
  svystatTM(des,~I(income*(regcod=="7")),vartype="cvpct")
  # Alternative solution (adding a new variable to des):
  des2 <- des.addvars(des, inc_reg7=I(income*(regcod=="7")))
  svystatTM(des2,~inc_reg7,vartype="cvpct")

  # Estimation of the total income (plus cvpct) ONLY in regions 6 and 10:
  svystatTM(des,~I(income*as.numeric(regcod %in% c("6","10"))),vartype="cvpct")
  # Alternative solution (adding a new variable to des):
  des2 <- des.addvars(des, inc_reg6.10=I(income*(regcod %in% c("6","10"))))
  svystatTM(des2,~inc_reg6.10,vartype="cvpct")

  # Compare with the corresponding estimates for the "complete" partition,
  # i.e. for regions:
  svystatTM(des,~income,~regcod,vartype="cvpct")


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

# but see also ?collapse.strata for a better alternative.


#####################################################
# Household-level estimation in household surveys. #
#####################################################

# Large scale household surveys typically adopt a 2-stage sampling
# design with municipalities as PSUs and households as SSUs, in order
# to eventually collect information on each individual belonging to
```

```
# sampled SSUs. In such a framework (up to possible total nonresponse
# effects), each individual inside a sampled household shares the
# same direct weight, which, in turn, equals the household weight.
# This implies that it is very easy to build estimates referred to
# SSU-level (households) information, despite estimators actually
# involve only individual values. Some examples are given below.

  # Load survey data:
  data(data.examples)

  # Define the survey design (variable famcod identifies households)
  exdes<-e.svydesign(data=example,ids=~towcod+famcod,strata=~stratum,
         weights=~weight)

  # Collapse strata to eliminate lonely PSUs
  exdes<-collapse.strata(design=exdes,block.vars=~sr:procod)

  # Now add new convenience variables to the design object:
    ## 'ones':       to estimate individuals counts
    ## 'housize':    to classify individuals by household size
    ## 'houdensity': to estimate households counts
  exdes<-des.addvars(exdes,
                     ones=1,
                     housize=factor(ave(famcod,famcod,FUN = length)),
                     houdensity=ave(famcod,famcod,FUN = function(x) 1/length(x))
                    )

  # Estimate the total number of households:
  nhou<-svystatTM(exdes,~houdensity,vartype="cvpct")
  nhou

  # Estimate the total number of individuals:
  nind<-svystatTM(exdes,~ones,vartype="cvpct")
  nind

  # Thus the average number of individuals per household is:
  coef(nind)/coef(nhou)

  # ...which can be obtained also as a ratio (along with
  # its estimated sampling variability):
  svystatR(exdes,~ones,~houdensity,vartype="cvpct")

  # Estimate the number and proportion of individuals living in households
  # of given sizes:
  nind.by.housize<-svystatTM(exdes,~housize,vartype="cvpct")
  nind.by.housize

  pind.by.housize<-svystatTM(exdes,~housize,estimator="Mean",var="cvpct")
  pind.by.housize

  # Estimate the number of households by household size:
  nhou.by.housize<-svystatTM(exdes,~houdensity,~housize,vartype="cvpct")
  nhou.by.housize

  # Notice that estimates of individuals and household counts are consistent,
  # indeed:
  coef(nind.by.housize)/coef(nhou.by.housize)
```

`trimcal` *Trim Calibration Weights while Preserving Calibration Constraints*

Description

This function trims calibration weights to a bounded interval, while preserving all the calibration constraints.

Usage

```
trimcal(cal.design, w.range = c(-Inf, Inf),
        maxit = 50, epsilon = 1e-07, force = TRUE)
```

Arguments

`cal.design`	Object of class `cal.analytic` containing the calibration weights to be trimmed.
`w.range`	The interval to which trimmed calibration weights must be bound (see 'Details'). The default is `c(-Inf,Inf)`, which would leave the calibration weights *unchanged*.
`maxit`	The same as in function `e.calibrate`.
`epsilon`	The same as in function `e.calibrate`.
`force`	The same as in function `e.calibrate`.

Details

Extreme calibration weights might determine unstable estimates and inflate sampling error estimates. To avoid this risk, extreme weights may be *trimmed* by using some suitable procedure (see e.g. [Potter 90], [Valliant, Dever, Kreuter 13]). Despite no rigorous justifications exist for any proposed trimming method, sometimes practitioners are (or feel) compelled to apply a trimming step before estimation. This happens more frequently when interest variables are highly skewed at the population level, like in business surveys or in social surveys with a focus on economic variables (e.g. income, see [Verma, Betti, Ghellini 07], [EUROSTAT 16]).

Unfortunately, the most common trimming techniques do *not* preserve the calibration constraints: if the input weights to the trimming algorithm are calibrated, typically the trimmed weights will *not* reproduce the calibration totals. As a consequence, users have to calibrate again the weights after trimming and iterate through the trimming and calibration steps, until a set of weights is obtained that respects both the trimming bounds and the calibration controls.

Function `trimcal` overcomes this limitation: it allows to trim calibration weights to a specific interval while *simultaneously* preserving all the calibration totals. To achieve this result, a *constrained trimming algorithm* is used, which is similar in spirit to the GEM (Generalized Exponential Method) of [Folsom, Singh 2000], but adopts the range restricted *euclidean* distance - instead of the *logit* - for numerical stability considerations.

When `w.range` is passed, *both* the trimming limits it defines must be *positive*. In other words, all calibration weights have to be positive after trimming. The purpose of this condition is to enable sound variance estimation on the trimmed object (see also below).

Note that `trimcal` is allowed to trim only *already calibrated* weights, i.e. the input object `cal.design` must be of class `cal.analytic`. This is a deliberate design choice, as trimming design (or initial) weights is methodologically unsound.

Note also that, in case the original calibration weights were asked to be constant within clusters selected at a given sampling stage (via argument aggregate.stage of e.calibrate), trimcal will *preserve* that property (see 'Examples').

Note lastly that trimcal will not trim further weights that have *just* been trimmed. This is again a deliberate design choice, devised to discourage over-trimming and cosmetic adjustments of the survey weights. Of course, it is instead entirely legitimate to calibrate again a trimmed object: after that, a further trimming step will be allowed.

From a variance estimation perspective, the trimmed object returned by function trimcal is treated as an ordinary calibrated object. More precisely, the trimming step is regarded as a *"finalization"* of the weight adjustment procedure which generated cal.design, i.e. as a completion of the previous calibration step. Call w, w.cal and w.cal.trim the starting weights, the calibrated weights of cal.design and the trimmed calibration weights as computed by function trimcal, respectively. Variance estimates computed on the trimmed object will pretend that one passed from w to w.cal.trim *directly* (w -> w.cal.trim), rather then in two steps (w -> w.cal -> w.cal.trim). Note incidentally that function get.residuals, when invoked on a trimmed object, will behave consistently with the variance estimation approach documented here.

Value

A calibrated object of class cal.analytic, storing *trimmed* calibration weights.

Trimming Process Diagnostics

Function trimcal exploits a *constrained* trimming algorithm to adjust the calibration weights so that (i) they fall within a bounded interval but (ii) still preserve all the calibration totals. *When this task is unfeasible, the algorithm will fail*. As a consequence, the adjusted weights returned in the output object will respect the range restrictions set by w.range, but some of the calibration constraints will be broken. Exactly as for function e.calibrate, in order to asses the degree of violation of the calibration constraints introduced by trimming, the user can exploit function check.cal (or, equivalently, the diagnostic data structure ecal.status available in the .GlobalEnv).

Methodological Warning

Trimming the calibration weights can result in introducing a *bias* in calibration estimates. Of course, one must hope that this *unknown* bias will turn out to be small compared to the *unknown* gain in precision obtained by trimming. In any case - since the actual effect of trimming weights on the MSE of the estimators is unclear - function trimcal should be used sparingly and carefully.

Author(s)

Diego Zardetto

References

Potter, F.J. (1990) *"A study of procedures to identify and trim extreme sampling weights"*. Proceedings of the Survey Research Methods Section, American Statistical Association, pp. 225-230.

Folsom, R.E., Singh, A.C. (2000) *"The generalized exponential model for sampling weight calibration for extreme values, nonresponse, and poststratification"*. Proceedings of the Section on Survey Research Methods, American Statistical Association, pp. 598-603.

Verma, V., Betti, G., Ghellini, G. (2007) *Cross-sectional and longitudinal weighting in a rotational household panel: applications to EU-SILC*, Statistics in Transition, 8(1), pp. 5-50.

Valliant, R., Dever, J., Kreuter, F. (2013) *"Practical Tools for Designing and Weighting Survey Samples"*. Springer-Verlag, New York.

EUROSTAT (2016) *"EU statistics on income and living conditions (EU-SILC) methodology - data quality"*, https://ec.europa.eu/eurostat/statistics-explained/index.php/EU_statistics_on_income_and_living_conditions_(EU-SILC)_methodology_%E2%80%93_data_quality.

See Also

e.calibrate for calibrating survey weights within **ReGenesees** and check.cal to check if calibration constraints have been fulfilled.

Examples

```
######################
## Data preparation ##
######################
# Load sbs data:
data(sbs)
# Build a design object:
sbsdes<-e.svydesign(data=sbs,ids=~id,strata=~strata,weights=~weight,fpc=~fpc)
# Build a population totals template and fill it with actual known totals:
pop<-pop.template(sbsdes, calmodel=~(emp.num+ent):emp.cl:nace.macro-1, ~region)
pop <- fill.template(sbs.frame, pop)
# Calibrate:
sbscal <- e.calibrate(sbsdes, pop)
# Have a look at the calibration weights distribution:
summary(weights(sbscal))


######################
## Trimming examples ##
######################
## Example 1
# Now suppose we want to trim these calibration weights to, say, the bounded
# interval [0.5, 20]. Let's use our trimcal() function:
sbstrim <- trimcal(sbscal, c(0.5, 20))

# Have a look at the trimmed object:
sbstrim

# Let's first verify that the trimmed calibration weights actually obey the
# imposed range restrictions...
summary(weights(sbstrim))
# ...ok, as it must be.

# Second, let's verify that the trimmed object still preserves all the
# calibration constraints:
check.cal(sbstrim)

# or, more explicitly:
all.equal(aux.estimates(sbscal,  template=pop),
          aux.estimates(sbstrim, template=pop))
# ...ok, as it must be.

# Let's have a look at the scatterplots of calibrated and trimmed weights:
plot(weights(sbsdes), weights(sbscal), pch = 20, col = "red",
```

```
     xlab = "Direct weights", ylab = "Calibration (red) and Trimmed (blue) weights")
points(weights(sbsdes), weights(sbstrim), pch = 20, col = "blue")
abline(h = c(0.5, 20), col = "green")

# Last, compute estimates and estimated sampling errors on the trimmed object
# as you would do on ordinary calibrated objects, e.g.
  # before trimming:
svystatTM(sbscal, y = ~va.imp2, by = ~nace.macro, estimator = "Mean")
  # after trimming:
svystatTM(sbstrim, y = ~va.imp2, by = ~nace.macro, estimator = "Mean")


## Example 2
# If w.range is too tight, constrained trimming can fail:
sbstrim2 <- trimcal(sbscal, c(1, 20))

# As a consequence, the trimmed weights will respect the range restrictions...
summary(weights(sbstrim2))

# ...but some of the calibration constraints will be broken:
check.cal(sbstrim2)


## Example 3
# If calibration weights were asked to be constant within clusters, the same
# will hold true for the trimmed calibration weights.

# Load household data:
data(data.examples)

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

# Calibrate asking that all individuals within any household share the same
# calibration weight (re-use an example from ?e.calibrate):
descal <- e.calibrate(design=des,df.population=pop04p,
                      calmodel=~x1+x2+x3-1,partition=~regcod,calfun="logit",
                      bounds=bounds,aggregate.stage=2)

# Have a look at the calibration weights distribution:
summary(weights(descal))

# Trim the calibration weights to, say, the bounded interval [150, 850]
destrim <- trimcal(descal, c(150, 850))

# Do trimmed calibration weights obey the imposed range restrictions?
summary(weights(destrim))

# Verify that trimmed weights are still equal within households:
any( tapply( weights(destrim), destrim$variables$famcod,
             function(x) {length(unique(x)) > 1} ) )

# FALSE, as it must be.


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

```
## Allowed and forbidden trimming policies ##
##############################################
# Let's illustrate some design restrictions on function trimcal():
# 1) Trimming limits must be both positive:
## Not run:
trimcal(sbscal, c(-0.05, 18))

## End(Not run)

# 2) Trimming design (or direct, or initial) weights is not allowed, you can
#    only trim calibration weights:
## Not run:
trimcal(sbsdes, c(1, 18))

## End(Not run)

# 3) You cannot trim further weights that have just been trimmed:
## Not run:
trimcal(sbstrim, c(1, 18))

## End(Not run)

# 4) You can calibrate again trimmed weights...
pop2<-pop.template(sbsdes, calmodel=~(emp.num+ent):area-1)
pop2<-fill.template(sbs.frame,pop2)
sbscal2<-e.calibrate(sbstrim, pop2)
summary(weights(sbscal2))

# ...after that, a further trimming step is allowed:
sbstrim2 <- trimcal(sbscal2, c(0.6, 19))
sbstrim2
summary(weights(sbstrim2))
```

UWE *Unequal Weighting Effect*

Description

Computes the Unequal Weighting Effect for the current and initial weights of a design object.

Usage

```
UWE(design, by = NULL)
```

Arguments

design	Object of class `analytic` (or inheriting from it).
by	Formula specifying variables that define "estimation domains". If `NULL` (the default option) the UWE refer to the whole sample.

Details

Function UWE computes the Unequal Weighting Effect for the current (w) and initial (w0) weights of a design object, plus the corresponding variance inflation (or deflation) factor (UWE(w) / UWE(w0)) induced by changing the weights from w0 to w (w0 -> w).

Following Kish's definition [Kish 92], the UWE is calculated as 1 plus the relative sample variance of the weights: UWE(w) = 1 + RelVar(w).

The *current* weights, w, of design are the weights that would be returned by weights(design) and would be used for estimation purposes by functions svystatTM, svystatR, etc.

The *initial* weights, w0, of design depend on the nature of object design:

- If design is the outcome of a **'weight-changing pipeline'**, w0 -> w1 -> ... -> w, i.e. it was obtained by the application of an arbitrary *chain* of **ReGenesees** functions that *modify the weights* (e.g. smooth.strat.jump, e.calibrate, ext.calibrated, trimcal, ...), then the initial weights, w0, are the weights of the starting design object in the pipeline.
- If design is an initial design object generated by function e.svydesign, then the initial weights, w0, are taken as equal to current weights, w0 = w.

Note that, when design is the outcome of a 'weight-changing pipeline', function UWE provides a measure of the overall, cumulative impact of all the adjustments the weights underwent throughout the pipeline.

To assess the effect, in terms of UWE and variance inflation, of just a single processing step of the pipeline, you can call function UWE on the input and output designs of that step and compare the results (basically, by taking suitable ratios).

Value

A data.frame, with one single row (if by = NULL) or one row for each domain (if by is passed), and the following columns:

```
Column        Meaning
UWE.curr......Current Unequal Weighting Effect
UWE.ini.......Initial Unequal Weighting Effect
VAR.infl......Variance Inflation Factor ( UWE.curr / UWE.ini )
```

Methodological Remark

Kish's UWE is a model-based tool that can be useful for diagnostic purposes. However, its values must be interpreted with some caution, exactly as it is necessary to do for model-based estimates of Kish's Deff.

In particular, UWE is - by construction - only sensitive to variations of the *sample* variance of the weights. Therefore, it is unable to discriminate weight adjustments which, despite adding variability to the weights at sample level, might result in reductions of the *sampling* variance for some estimators. This is often the case of calibration, which may well make survey weights more unequal, but nonetheless cause their reciprocals to become more correlated to some interest variables.

In any case, the UWE can turn out handy when comparing the potential outcomes of performing the same kind of weight adjustment under slightly different settings (e.g. calibration with different bounds or distance functions, trimming with different thresholds, etc.).

Author(s)

Diego Zardetto

References

Kish, L. (1992). Weighting for unequal Pi. Journal of Official Statistics, 8, 183-200.

See Also

ReGenesees functions which define survey weights (`e.svydesign`), or modify survey weights (e.g. `smooth.strat.jump`, `e.calibrate`, `ext.calibrated`, `trimcal`, ...).

Examples

```
################################################
# Compute the UWE along the following example #
# of weight-changing pipeline:                 #
# 1) Smooth for stratum jumpers                #
# 2) Adjust for nonresponse                    #
# 3) Calibrate to known population totals      #
# 4) Consistently trim calibration weights     #
#                                              #
# NOTE: To perform 1) and 2) I will first      #
#       A) simulate some stratum jumpers.      #
#       B) simulate some nonresponse.          #
################################################

## Load sbs data:
data(sbs)

## -- A) Simulate stratum jumpers
# Create the strata variable observed at survey time by cloning the
# strata variable at sampling time
sbs$curr.strata <- sbs$strata

# Now inject some (say ~250) random stratum jumpers
set.seed(12345)          # (fix the RNG seed for reproducibility)
sbs$curr.strata[sample(1:nrow(sbs), 250)] <- sbs$curr.strata[sample(1:nrow(sbs), 250)]

# Resulting number of stratum jumpers:
tt <- table(sbs$strata, sbs$curr.strata)
sum(tt[row(tt) != col(tt)])

## -- B) Simulate nonresponse
# Assume a response propensity that increases with enterprise size (as
# measured by number of employees)
levels(sbs$emp.cl)
p.resp <- c(.4, .6, .8, .95, .99)

# Tie response probabilities to sample observations:
pr <- p.resp[unclass(sbs$emp.cl)]

# Now, randomly select a subsample of responding units from sbs:
set.seed(12345)          # (fix the RNG seed for reproducibility)
rand <- runif(1:nrow(sbs))
sbs.r <- sbs[rand < pr, ]

# This implies an overall response rate of about 73%:
nrow(sbs.r) / nrow(sbs)
```

```
## -- 0) Create the respondent design object
# NOTE: I'll keep using the original fpc column for the sake of the examples,
#       but they should be recomputed in real applications...
sbsdes<-e.svydesign(data=sbs.r,ids=~id,strata=~strata,weights=~weight,fpc=~fpc)

## -- 1) Smooth for stratum jumpers
# Use method 'MinChange'
sbssmooth <- smooth.strat.jump(sbsdes, ~curr.strata)

# Have a look
sbssmooth

## -- 2) Adjust for nonresponse
# Use a simple Response Homogeneity Model approach, with size classes
# as RHGs. Perform the RHG weight adjustment via calibration

# Compute enterprise counts by size classes from the frame
N.RHG <- pop.template(sbssmooth, calmodel= ~emp.cl - 1)
N.RHG <- fill.template(sbs.frame, N.RHG)

# Calibrate to achieve the RHG adjustment
sbsRHG <- e.calibrate(sbssmooth, N.RHG)

# Have a look
sbsRHG

# -- 3) Calibrate to known population totals
# Now calibrate again in order to reduce estimators variance, by using further
# available auxiliary information, e.g. the total number of employees (emp.num)
# and enterprises (ent) inside the domains obtained by crossing nace.macro
# and region:
pop <- pop.template(sbsRHG, calmodel = ~emp.num + ent-1,
                    partition = ~nace.macro:region)
pop <- fill.template(sbs.frame, pop)

# Calibrate to improve estimation efficiency
sbscal <- e.calibrate(sbsRHG, pop)

# Have a look
sbscal

# -- 4) Consistently trim calibration weights
# Say one wants to avoid weights that are less then 1 and above 50:
sbstrim <- trimcal(sbscal, c(1, 50))

# Have a look
sbstrim

## -- UWE calculation along the weights-changing pipeline
# Object sbstrim is the output of the weights-changing pipeline, as
# one easily recognizes when printing it:
sbstrim

# UWE of initial object
UWE(sbsdes)

# UWE at step 1), i.e. smoothing for stratum jumpers
```

```
UWE(sbssmooth)

# UWE of step 2), i.e. nonresponse RHG adjustment
UWE(sbsRHG)

# UWE at step 3), i.e. calibration for efficiency improvement
UWE(sbscal)

# UWE at step 4), i.e. consistent trimming of calibration weights
UWE(sbstrim)

# End
```

`weights` *Retrieve Sampling Units Weights*

Description

Extracts the *current* weights of 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`, `svystatS`, `svystatSR`, `svystatQ`, `svystatB`, `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 partitioned 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, svystatS, svystatSR, svystatB, 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 confidence 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)
```

Zapsmall *Zapsmall Data Frame Columns and Numeric Vectors*

Description

Put to zero values "close" to zero.

Usage

```
## Default S3 method:
Zapsmall(x, digits = getOption("digits"), ...)
## S3 method for class 'data.frame'
Zapsmall(x, digits = getOption("digits"), except = NULL, ...)
```

Arguments

x	A data.frame with numeric columns or a numeric vector.
digits	Integer indicating the precision to be used.
except	Indices of columns not to be zapped (if any).
...	Arguments for future expansion.

Details

This function "extends" to data.frame objects function zapsmall from the package **base**.
The method for class data.frame 'zaps' values close to zero occurring in columns of x. Argument except can be used to prevent specific columns from being zapped.
The default method is a bare copy of the original function from package **base**.

Value

An object of the same class of x, with values "close" to zero zapped to zero.

Author(s)

Diego Zardetto

See Also

The original function zapsmall from package **base**.

Examples

```
# Create a test data frame with columns containing
# values of different orders of magnitude:
data <- data.frame(a = pi*10^(-8:1), b = c(rep(1000,8), c(1E-5, 1E-6)))

# Print on screen the test data frame:
data

# Compare with its zapped version:
Zapsmall(data)
```

`%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 sparsity 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 `factor`s 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 data frame). 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 data frames), with the net result of washing-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 data frame 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, age5c 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))
               )

 ## 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))
                )

 ## 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))
                )

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

```
 # thus, tpp < tp < t, as expected.
 # Notice also that we obtained identical calibrated weights:
all.equal(weights(cal07),weights(cal07p))
all.equal(weights(cal07),weights(cal07pp))

 # as it must be.
```

Index