Package ‘survey’

January 21, 2020

**Title**  Analysis of Complex Survey Samples

**Description**  Summary statistics, two-sample tests, rank tests, generalised linear models, cumulative link models, Cox models, loglinear models, and general maximum pseudolikelihood estimation for multistage stratified, cluster-sampled, unequally weighted survey samples. Variances by Taylor series linearisation or replicate weights. Post-stratification, calibration, and raking. Two-phase subsampling designs. Graphics. PPS sampling without replacement. Principal components, factor analysis.

**Version**  3.37

**Author**  Thomas Lumley

**Maintainer**  `Thomas Lumley <t.lumley@auckland.ac.nz>`

**License**  GPL-2 | GPL-3

**Depends**  R (>= 3.1.0), grid, methods, Matrix, survival

**Imports**  stats, graphics, splines, lattice, minqa, numDeriv, mitools

(>= 2.4)

**Suggests**  foreign, MASS, KernSmooth, hexbin, RSQLite, quantreg, parallel, CompQuadForm, DBI, AER

**URL**  [http://r-survey.r-forge.r-project.org/survey/](http://r-survey.r-forge.r-project.org/survey/)

**NeedsCompilation**  no

**Repository**  CRAN

**Date/Publication**  2020-01-21 09:40:02 UTC

---

**R topics documented:**

anova.svyglm .......................................................... 3
api ................................................................................. 5
as.fpc ................................................................. 8
as.svrepdesign .......................................................... 9
as.svydesign2 ...................................................... 11
barplot.svystat ....................................................... 12
bootweights ............................................................. 13
brrweights .............................................................. 14
### R topics documented:

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>calibrate</td>
<td>17</td>
</tr>
<tr>
<td>compressWeights</td>
<td>23</td>
</tr>
<tr>
<td>confint.svylm</td>
<td>24</td>
</tr>
<tr>
<td>crowd</td>
<td>25</td>
</tr>
<tr>
<td>dimnames.DBIsvydesign</td>
<td>26</td>
</tr>
<tr>
<td>election</td>
<td>27</td>
</tr>
<tr>
<td>estweights</td>
<td>28</td>
</tr>
<tr>
<td>fpc</td>
<td>30</td>
</tr>
<tr>
<td>ftable.svystat</td>
<td>31</td>
</tr>
<tr>
<td>hadamard</td>
<td>32</td>
</tr>
<tr>
<td>hospital</td>
<td>34</td>
</tr>
<tr>
<td>HR</td>
<td>35</td>
</tr>
<tr>
<td>make.calfun</td>
<td>35</td>
</tr>
<tr>
<td>marginpred</td>
<td>37</td>
</tr>
<tr>
<td>mu284</td>
<td>38</td>
</tr>
<tr>
<td>nhanes</td>
<td>39</td>
</tr>
<tr>
<td>nonresponse</td>
<td>40</td>
</tr>
<tr>
<td>open.DBIsvydesign</td>
<td>41</td>
</tr>
<tr>
<td>paley</td>
<td>42</td>
</tr>
<tr>
<td>pchisqsum</td>
<td>44</td>
</tr>
<tr>
<td>postStratify</td>
<td>45</td>
</tr>
<tr>
<td>psrsq</td>
<td>47</td>
</tr>
<tr>
<td>rake</td>
<td>48</td>
</tr>
<tr>
<td>regTermTest</td>
<td>50</td>
</tr>
<tr>
<td>scd</td>
<td>52</td>
</tr>
<tr>
<td>SE</td>
<td>53</td>
</tr>
<tr>
<td>stratsample</td>
<td>54</td>
</tr>
<tr>
<td>subset.survey.design</td>
<td>55</td>
</tr>
<tr>
<td>surveyoptions</td>
<td>56</td>
</tr>
<tr>
<td>surveysummary</td>
<td>57</td>
</tr>
<tr>
<td>svrepdesign</td>
<td>60</td>
</tr>
<tr>
<td>svrVar</td>
<td>64</td>
</tr>
<tr>
<td>svy.varcoef</td>
<td>64</td>
</tr>
<tr>
<td>svyby</td>
<td>65</td>
</tr>
<tr>
<td>svycdf</td>
<td>68</td>
</tr>
<tr>
<td>svyciprop</td>
<td>69</td>
</tr>
<tr>
<td>svycontrast</td>
<td>71</td>
</tr>
<tr>
<td>svycoplot</td>
<td>73</td>
</tr>
<tr>
<td>svycxph</td>
<td>74</td>
</tr>
<tr>
<td>svyCprod</td>
<td>76</td>
</tr>
<tr>
<td>svycralpha</td>
<td>78</td>
</tr>
<tr>
<td>svydesign</td>
<td>79</td>
</tr>
<tr>
<td>svyfactanal</td>
<td>82</td>
</tr>
<tr>
<td>svyglm</td>
<td>83</td>
</tr>
<tr>
<td>svyhist</td>
<td>86</td>
</tr>
<tr>
<td>svyivreg</td>
<td>88</td>
</tr>
<tr>
<td>svykappa</td>
<td>89</td>
</tr>
<tr>
<td>svykappa</td>
<td>89</td>
</tr>
</tbody>
</table>
### anova.svyglm

Model comparison for glms.

#### Description

A method for the `anova` function, for use on `svyglm` objects. With a single model argument it produces a sequential anova table, with two arguments it compares the two models.

#### Usage

```r
## S3 method for class 'svyglm'
anova(object, object2 = NULL, test = c("F", "Chisq"),
       method = c("LRT", "Wald"), tolerance = 1e-05, ..., force = FALSE)
## S3 method for class 'svyglm'
AIC(object, ..., k=2)
## S3 method for class 'svyglm'
BIC(object, ..., maximal)
```
Arguments

object A `svyglm` object.
object2 Optionally, another `svyglm` object.
test Use (linear combination of) F or chi-squared distributions for p-values. F is usually preferable.
method Use weighted deviance difference (LRT) or Wald tests to compare models
tolerance For models that are not symbolically nested, the tolerance for deciding that a term is common to the models.
... For AIC and BIC, optionally more `svyglm` objects
force Force the tests to be done by explicit projection even if the models are symbolically nested (eg, for debugging)
maximal A `svyglm` model that `object` (and ... if supplied) are nested in.
k Multiplier for effective df in AIC. Usually 2. There is no choice of k that will give BIC

Details

The reference distribution for the LRT depends on the misspecification effects for the parameters being tested (Rao and Scott, 1984). If the models are symbolically nested, so that the relevant parameters can be identified just by manipulating the model formulas, `anova` is equivalent to `regTermTest`. If the models are nested but not symbolically nested, more computation using the design matrices is needed to determine the projection matrix on to the parameters being tested. In the examples below, `model1` and `model2` are symbolically nested in `model0` because `model0` can be obtained just by deleting terms from the formulas. On the other hand, `model2` is nested in `model1` but not symbolically nested: knowing that the model is nested requires knowing what design matrix columns are produced by `stype` and `as.numeric(stype)`. Other typical examples of models that are nested but not symbolically nested are linear and spline models for a continuous covariate, or models with categorical versions of a variable at different resolutions (eg, smoking yes/no or smoking never/former/current).

A saddlepoint approximation is used for the LRT with numerator df greater than 1.

AIC is defined using the Rao-Scott approximation to the weighted loglikelihood (Lumley and Scott, 2015). It replaces the usual penalty term p, which is the null expectation of the log likelihood ratio, by the trace of the generalised design effect matrix, which is the expectation under complex sampling. For computational reasons everything is scaled so the weights sum to the sample size.

BIC is a BIC for the (approximate) multivariate Gaussian models on regression coefficients from the maximal model implied by each submodel (ie, the models that say some coefficients in the maximal model are zero) (Lumley and Scott, 2015). It corresponds to comparing the models with a Wald test and replacing the sample size in the penalty by an effective sample size. For computational reasons, the models must not only be nested, the names of the coefficients must match.

Value

Object of class `seqanova.svyglm` if one model is given, otherwise of class `regTermTest` or `regTermTestLRT`
Note

At the moment, AIC works only for models including an intercept.

References


See Also

regTermTest, pchisqsum

Examples

data(api)
dclus2<-svydesign(id=~dnum+snum, weights=~pw, data=apiclus2)

model0<-svyglm(I(sch.wide=="Yes")~ell+meals+mobility, design=dclus2, family=quasibinomial())
model1<-svyglm(I(sch.wide=="Yes")~ell+meals+mobility+as.numeric(stype), design=dclus2, family=quasibinomial())
model2<-svyglm(I(sch.wide=="Yes")~ell+meals+mobility+stype, design=dclus2, family=quasibinomial())

anova(model2)
anova(model0, model2)
anova(model1, model2)
anova(model1, model2, method="Wald")

AIC(model0, model1, model2)
BIC(model0, model2, maximal=model2)

api

Student performance in California schools

Description

The Academic Performance Index is computed for all California schools based on standardised testing of students. The data sets contain information for all schools with at least 100 students and for various probability samples of the data.
Usage

data(api)

Format

The full population data in api.pop are a data frame with 6194 observations on the following 37 variables.

cds  Unique identifier
stype  Elementary/Middle/High School
name  School name (15 characters)
sname  School name (40 characters)
snum  School number
dname  District name
dnum  District number
cname  County name
cnum  County number
flag  reason for missing data
pcttest  percentage of students tested
api00  API in 2000
api99  API in 1999
target  target for change in API
growth  Change in API
sch.wide  Met school-wide growth target?
comp.imp  Met Comparable Improvement target
both  Met both targets
awards  Eligible for awards program
meals  Percentage of students eligible for subsidized meals
ell  ‘English Language Learners’ (percent)
yr.rnd  Year-round school
mobility  percentage of students for whom this is the first year at the school
acs.k3  average class size years K-3
acs.46  average class size years 4-6
acs.core  Number of core academic courses
pct.resp  percent where parental education level is known
not.hsg  percent parents not high-school graduates
hsg  percent parents who are high-school graduates
some.col  percent parents with some college
col.grad  percent parents with college degree
grad.sch  percent parents with postgraduate education
avg.ed  average parental education level
full  percent fully qualified teachers
emer  percent teachers with emergency qualifications
enroll  number of students enrolled
api.stu  number of students tested.

The other data sets contain additional variables pw for sampling weights and fpc to compute finite population corrections to variance.

Details

apipop is the entire population, apisrs is a simple random sample, apiclus1 is a cluster sample of school districts, apistrat is a sample stratified by stype, and apiclus2 is a two-stage cluster sample of schools within districts. The sampling weights in apiclus1 are incorrect (the weight should be 757/15) but are as obtained from UCLA.

Source

Data were obtained from the survey sampling help pages of UCLA Academic Technology Services; these pages are no longer on line.

References

The API program has been discontinued at the end of 2018. Information is archived at https://www.cde.ca.gov/re/pr/api.asp

Examples

library(survey)
data(api)
mean(apipop$api00)
sum(apipop$enroll, na.rm=TRUE)

#stratified sample
dstrat<-svydesign(id=~1, strata=~stype, weights=~pw, data=apistrat, fpc=~fpc)
survey(dstrat)
svymean(~api00, dstrat)
svytotal(~enroll, dstrat, na.rm=TRUE)

# one-stage cluster sample
dclus1<-svydesign(id=~dnum, weights=~pw, data=apiclus1, fpc=~fpc)
survey(dclus1)
svymean(~api00, dclus1)
svytotal(~enroll, dclus1, na.rm=TRUE)

# two-stage cluster sample
dclus2<-svydesign(id=~dnum+snum, fpc=~fpc1+fpc2, data=apiclus2)
survey(dclus2)
svymean(~api00, dclus2)
svytotal(~enroll, dclus2, na.rm=TRUE)

# two-stage 'with replacement'
dclus2wr<-svydesign(id=~dnum+snum, weights=~pw, data=apiclus2)
summary(dclus2wr)
svymean(~api00, dclus2wr)
svytotal(~enroll, dclus2wr, na.rm=TRUE)

# convert to replicate weights
rclus1<-as.svrepdesign(dclus1)
summary(rclus1)
svymean(~api00, rclus1)
svytotal(~enroll, rclus1, na.rm=TRUE)

# post-stratify on school type
pop.types<-xtabs(~stype, data=apipop)
rclus1p<-postStratify(rclus1, ~stype, pop.types)
dclus1p<-postStratify(dclus1, ~stype, pop.types)
summary(dclus1p)
summary(rclus1p)
svymean(~api00, dclus1p)
svytotal(~enroll, dclus1p, na.rm=TRUE)
svymean(~api00, rclus1p)
svytotal(~enroll, rclus1p, na.rm=TRUE)

---

**as.fpc**

*Package sample and population size data*

**Description**

This function creates an object to store the number of clusters sampled within each stratum (at each stage of multistage sampling) and the number of clusters available in the population. It is called by svydesign, not directly by the user.

**Usage**

```r
as.fpc(df, strata, ids, pps=FALSE)
```

**Arguments**

- `df` A data frame or matrix with population size information
- `strata` A data frame giving strata at each stage
- `ids` A data frame giving cluster ids at each stage
- `pps` if TRUE, fpc information may vary within a stratum and must be specified as a proportion rather than a population sizes
Details

The population size information may be specified as the number of clusters in the population or as the proportion of clusters sampled.

Value

An object of class `survey_fpc`

See Also

`svydesign`, `svyrecvar`

---

### as.svrepdesign

**Convert a survey design to use replicate weights**

#### Description

Creates a replicate-weights survey design object from a traditional strata/cluster survey design object. JK1 and JKn are jackknife methods, BRR is Balanced Repeated Replicates and Fay is Fay’s modification of this, bootstrap is Canty and Davison’s bootstrap, subbootstrap is Rao and Wu’s \((n-1)\) bootstrap, and mrbbootstrap is Preston’s multistage rescaled bootstrap.

#### Usage

```r
as.svrepdesign(design, type=c("auto", "JK1", "JKn", "BRR", "bootstrap", "subbootstrap","mrbbootstrap","Fay"), fay.rho = 0, fpc=NULL,fpctype=NULL,..., compress=TRUE, mse=getOption("survey.replicates.mse"))
```

#### Arguments

- `design`: Object of class `survey.design`
- `type`: Type of replicate weights. "auto" uses JKn for stratified, JK1 for unstratified designs
- `fay.rho`: Tuning parameter for Fay’s variance method
- `fpc`: Passed to `jk1weights`, `jknweights`, `brrweights`, `bootweights`, `subbootweights`, or `mrbweights`.
- `fpctype`,...
- `compress`: Use a compressed representation of the replicate weights matrix.
- `mse`: if TRUE, compute variances from sums of squares around the point estimate, rather than the mean of the replicates

#### Value

Object of class `svyrep.design`. 
References


See Also

brrweights, svydesign, svrepdesign, bootweights, subbootweights, mrbweights

Examples

data(scd)
scddes<-svydesign(data=scd, prob=~1, id=~ambulance, strata=~ESA, nest=TRUE, fpc=rep(5,6))
scdnofpc<-svydesign(data=scd, prob=~1, id=~ambulance, strata=~ESA, nest=TRUE)

# convert to BRR replicate weights
scd2brr <- as.svrepdesign(scdnofpc, type="BRR")
scd2fay <- as.svrepdesign(scdnofpc, type="Fay", fay.rho=0.3)
# convert to JK1 weights
scd2jkn <- as.svrepdesign(scdnofpc, type="JKn")

# convert to JK1 weights with finite population correction
scd2jknf <- as.svrepdesign(scddes, type="JKn")

## with user-supplied hadamard matrix
scd2brr1 <- as.svrepdesign(scdnofpc, type="BRR", hadamard.matrix=paley(11))

svyratio(~alive, ~arrests, design=scd2brr)
svyratio(~alive, ~arrests, design=scd2brr1)
svyratio(~alive, ~arrests, design=scd2fay)
svyratio(~alive, ~arrests, design=scd2jkn)
svyratio(~alive, ~arrests, design=scd2jknf)

data(api)
## one-stage cluster sample
dclus1<-svydesign(id=~dnum, weights=~pw, data=apiclus1, fpc=~fpc)
## convert to JK1 jackknife
rclus1<-as.svrepdesign(dclus1)
## convert to bootstrap
bclus1<-as.svrepdesign(dclus1,type="bootstrap", replicates=100)

svymean(~api00, dclus1)
svytotal(~enroll, dclus1)
svymean(~api00, rclus1)
svytotal(~enroll, rclus1)

svymean(~api00, bclus1)
svytotal(~enroll, bclus1)

dclus2<-svydesign(id = ~dnum + snum, fpc = ~fpc1 + fpc2, data = apiclus2)
mrbclus2<-as.svrepdesign(dclus2, type="mrb",replicates=100)
svytotal(~api00+stype, dclus2)
svytotal(~api00+stype, mrbclus2)

---

as.svydesign2  Update to the new survey design format

Description

The structure of survey design objects changed in version 2.9, to allow standard errors based on multistage sampling. as.svydesign converts an object to the new structure and .svycheck warns if an object does not have the new structure.

You can set options(survey.want.obsolete=TRUE) to suppress the warnings produced by .svycheck and options(survey.ultimate.cluster=TRUE) to always compute variances based on just the first stage of sampling.

Usage

as.svydesign2(object)
.svycheck(object)

Arguments

object produced by svydesign

Value

Object of class survey.design2

See Also

svydesign, svyrecvar
Description

Draws a barplot or dotplot based on results from a survey analysis. The default barplot method already works for results from `svytable`.

Usage

```r
## S3 method for class 'svystat'
barplot(height, ...)
## S3 method for class 'svrepstat'
barplot(height, ...)
## S3 method for class 'svyby'
barplot(height, beside=TRUE, ...)

## S3 method for class 'svystat'
dotchart(x,...,pch=19)
## S3 method for class 'svrepstat'
dotchart(x,...,pch=19)
## S3 method for class 'svyby'
dotchart(x,...,pch=19)
```

Arguments

- `height, x` Analysis result
- `beside` Grouped, rather than stacked, bars
- `...` Arguments to `barplot` or `dotchart`
- `pch` Overrides the default in `dotchart.default`

Examples

```r
data(api)
dclus1<-svydesign(id=~dnum, weights=~pw, data=apiclus1, fpc=~fpc)
a<-svymean(~stype, dclus1)
barplot(a)
barplot(a, names.arg=c("Elementary","High","Middle"), col="purple",
       main="Proportions of school level")

b<-svyby(~enroll+api.stu, ~stype, dclus1, svymean)
barplot(b,beside=TRUE,legend=TRUE)
dotchart(b)
```
bootweights

Compute survey bootstrap weights

Description

Bootstrap weights for infinite populations ('with replacement' sampling) are created by sampling with replacement from the PSUs in each stratum. subbootweights() samples n-1 PSUs from the n available (Rao and Wu), bootweights samples n (Canty and Davison).

For multistage designs or those with large sampling fractions, mrbweights implements Preston’s multistage rescaled bootstrap. The multistage rescaled bootstrap is still useful for single-stage designs with small sampling fractions, where it reduces to a half-sample replicate method.

Usage

bootweights(strata, psu, replicates = 50, fpc = NULL, fpctype = c("population", "fraction", "correction"), compress = TRUE)
subbootweights(strata, psu, replicates = 50, compress = TRUE)
mrbweights(clusters, stratas, fpcs, replicates=50, multicore=getOption("survey.multicore"))

Arguments

strata Identifier for sampling strata (top level only)
stratas data frame of strata for all stages of sampling
psu Identifier for primary sampling units
clusters data frame of identifiers for sampling units at each stage
replicates Number of bootstrap replicates
fpc Finite population correction (top level only)
fpctype Is fpc the population size, sampling fraction, or 1-sampling fraction?
fpcs survey_fpc object with population and sample size at each stage
compress Should the replicate weights be compressed?
multicore Use the multicore package to generate the replicates in parallel

Value

A set of replicate weights

warning

With multicore=TRUE the resampling procedure does not use the current random seed, so the results cannot be exactly reproduced even by using set.seed()
Note

These bootstraps are strictly appropriate only when the first stage of sampling is a simple or stratified random sample of PSUs with or without replacement, and not (eg) for PPS sampling. The functions will not enforce simple random sampling, so they can be used (approximately) for data that have had non-response corrections and other weight adjustments. It is preferable to apply these adjustments after creating the bootstrap replicate weights, but that may not be possible with public-use data.

References


See Also

as.svrepdesign

brrweights

Compute replicate weights

Description

Compute replicate weights from a survey design. These functions are usually called from as.svrepdesign rather than directly by the user.

Usage

brrweights(strata, psu, match = NULL,
           small = c("fail","split","merge"),
           large = c("split","merge","fail"),
           fay.rho=0, only.weights=FALSE,
           compress=TRUE, hadamard.matrix=NULL)

jk1weights(psu,fpc=NULL,
           fpctype=c("population","fraction","correction"),
           compress=TRUE)

jknweights(strata,psu, fpc=NULL,
           fpctype=c("population","fraction","correction"),
           compress=TRUE,
           lonely.psu=getOption("survey.lonely.psu"))
Arguments

- **strata**: Stratum identifiers
- **psu**: PSU (cluster) identifier
- **match**: Optional variable to use in matching.
- **small**: How to handle strata with only one PSU
- **large**: How to handle strata with more than two PSUs
- **fpc**: Optional population (stratum) size or finite population correction
- **fpctype**: How fpc is coded.
- **fay.rho**: Parameter for Fay’s extended BRR method
- **only.weights**: If TRUE return only the matrix of replicate weights
- **compress**: If TRUE, store the replicate weights in compressed form
- **hadamard.matrix**: Optional user-supplied Hadamard matrix for brrweights
- **lonely.psu**: Handling of non-certainty single-PSU strata

Details

JK1 and JKn are jackknife schemes for unstratified and stratified designs respectively. The finite population correction may be specified as a single number, a vector with one entry per stratum, or a vector with one entry per observation (constant within strata). When fpc is a vector with one entry per stratum it may not have names that differ from the stratum identifiers (it may have no names, in which case it must be in the same order as unique(strata)). To specify population stratum sizes use fpctype="population", to specify sampling fractions use fpctype="fraction" and to specify the correction directly use fpctype="correction"

The only reason not to use compress=TRUE is that it is new and there is a greater possibility of bugs. It reduces the number of rows of the replicate weights matrix from the number of observations to the number of PSUs.

In BRR variance estimation each stratum is split in two to give half-samples. Balanced replicated weights are needed, where observations in two different strata end up in the same half stratum as often as in different half-strata. BRR, strictly speaking, is defined only when each stratum has exactly two PSUs. A stratum with one PSU can be merged with another such stratum, or can be split to appear in both half samples with half weight. The latter approach is appropriate for a PSU that was deterministically sampled.

A stratum with more than two PSUs can be split into multiple smaller strata each with two PSUs or the PSUs can be merged to give two superclusters within the stratum.

When merging small strata or grouping PSUs in large strata the match variable is used to sort PSUs before merging, to give approximate matching on this variable.

If you want more control than this you should probably construct your own weights using the Hadamard matrices produced by hadamard.
Value

For brrweights with only.weights=FALSE a list with elements

weights two-column matrix indicating the weight for each half-stratum in one particular set of split samples
wstrata New stratum variable incorporating merged or split strata
strata Original strata for distinct PSUs
psu Distinct PSUs
npairs Dimension of Hadamard matrix used in BRR construction
sampler function returning replicate weights
compress Indicates whether the sampler returns per PSU or per observation weights

For jk1weights and jk1nweights a data frame of replicate weights and the scale and rscale arguments to svrVar.

References

Levy and Lemeshow "Sampling of Populations". Wiley.
Shao and Tu "The Jackknife and Bootstrap". Springer.

See Also

hadamard, as.svrepdesign, svrVar, surveyoptions

Examples

data(scd)
scdnofpc<-svydesign(data=scd, prob=~1, id=~ambulance, strata=~ESA, nest=TRUE)

## convert to BRR replicate weights
scdbrr <- as.svrepdesign(scdnofpc, type="BRR")
svymean(~alive, scdbrr)
svyratio(~alive, ~arrests, scdbrr)

## with user-supplied hadamard matrix
scdbrr1 <- as.svrepdesign(scdnofpc, type="BRR", hadamard.matrix=paley(11))
svymean(~alive, scdbrr1)
svyratio(~alive, ~arrests, scdbrr1)
calibrate

Description

Calibration, generalized raking, or GREG estimators generalise post-stratification and raking by calibrating a sample to the marginal totals of variables in a linear regression model. This function reweights the survey design and adds additional information that is used by svyrecvar to reduce the estimated standard errors.

Usage

```r
calibrate(design,...)
```

## S3 method for class 'survey.design2'

calibrate(design, formula, population,
aggregate.stage=NULL, stage=0, variance=NULL,
bounds=c(-Inf,Inf), calfun=c("linear","raking","logit"),
maxit=50, epsilon=1e-7, verbose=FALSE, force=FALSE, trim=NULL,
bounds.const=FALSE, sparse=FALSE,...)

## S3 method for class 'svyrep.design'

calibrate(design, formula, population, compress=NA,
aggregate.index=NULL, variance=NULL, bounds=c(-Inf,Inf),
calfun=c("linear","raking","logit"),
maxit=50, epsilon=1e-7, verbose=FALSE, force=FALSE, trim=NULL,
bounds.const=FALSE, sparse=FALSE,...)

## S3 method for class 'twophase'

calibrate(design, phase=2, formula, population,
calfun=c("linear","raking","logit","rrz"),...)

grake(mm, ww, calfun, eta=rep(0, NCOL(mm)), bounds, population, epsilon,
verbose, maxit, variance=NULL)

cal_names(formula, design,...)
```

Arguments

design Survey design object
formula Model formula for calibration model, or list of formulas for each margin
population Vectors of population column totals for the model matrix in the calibration model, or list of such vectors for each cluster, or list of tables for each margin. Required except for two-phase designs
compress compress the resulting replicate weights if TRUE or if NA and weights were previously compressed
stage See Details below
variance Coefficients for variance in calibration model (heteroskedasticity parameters) (see Details below)
aggregate.stage
An integer. If not NULL, make calibration weights constant within sampling units at this stage.

aggregate.index
A vector or one-sided formula. If not NULL, make calibration weights constant within levels of this variable

bounds
Bounds for the calibration weights, optional except for calfun="logit"

bounds.const
Should be TRUE if bounds have been specified as constant values rather than multiplicative values

trim
Weights outside this range will be trimmed to these bounds.

... Options for other methods
calfun
Calibration function: see below

maxit
Number of iterations

epsilon
Tolerance in matching population total. Either a single number or a vector of the same length as population

verbose
Print lots of uninteresting information

force
Return an answer even if the specified accuracy was not achieved

phase
Phase of a two-phase design to calibrate (only phase=2 currently implemented.)

mm
Model matrix

ww
Vector of weights

eta
Starting values for iteration

sparse
Use sparse matrices for faster computation

Details

The formula argument specifies a model matrix, and the population argument is the population column sums of this matrix. The function cal_names shows what the column names of this model matrix will be.

For the important special case where the calibration totals are (possibly overlapping) marginal tables of factor variables, as in classical raking, the formula and population arguments may be lists in the same format as the input to rake.

If the population argument has a names attribute it will be checked against the names produced by model.matrix(formula) and reordered if necessary. This protects against situations where the (locale-dependent) ordering of factor levels is not what you expected.

Numerical instabilities may result if the sampling weights in the design object are wrong by multiple orders of magnitude. The code now attempts to rescale the weights first, but it is better for the user to ensure that the scale is reasonable.

The calibrate function implements linear, bounded linear, raking, bounded raking, and logit calibration functions. All except unbounded linear calibration use the Newton-Raphson algorithm described by Deville et al (1993). This algorithm is exposed for other uses in the grake function. Unbounded linear calibration uses an algorithm that is less sensitive to collinearity. The calibration function may be specified as a string naming one of the three built-in functions or as an object of class calfun, allowing user-defined functions. See make.calfun for details.
The bounds argument can be specified as global upper and lower bounds e.g. bounds=c(0.5,2) or as a list with lower and upper vectors e.g. bounds=list(lower=lower,upper=upper). This allows for individual boundary constraints for each unit. The lower and upper vectors must be the same length as the input data. The bounds can be specified as multiplicative values or constant values. If constant, bounds.const must be set to TRUE.

Calibration with bounds, or on highly collinear data, may fail. If force=TRUE the approximately calibrated design object will still be returned (useful for examining why it failed). A failure in calibrating a set of replicate weights when the sampling weights were successfully calibrated will give only a warning, not an error.

When calibration to the desired set of bounds is not possible, another option is to trim weights. To do this set bounds to a looser set of bounds for which calibration is achievable and set trim to the tighter bounds. Weights outside the bounds will be trimmed to the bounds, and the excess weight distributed over other observations in proportion to their sampling weight (and so this may put some other observations slightly over the trimming bounds). The projection matrix used in computing standard errors is based on the feasible bounds specified by the bounds argument. See also trimWeights, which trims the final weights in a design object rather than the calibration adjustments.

For two-phase designs calfun="rrz" estimates the sampling probabilities using logistic regression as described by Robins et al (1994). estWeights will do the same thing.

Calibration may result in observations within the last-stage sampling units having unequal weight even though they necessarily are sampled together. Specifying aggregate.stage ensures that the calibration weight adjustments are constant within sampling units at the specified stage; if the original sampling weights were equal the final weights will also be equal. The algorithm is as described by Vanderhoeft (2001, section III.D). Specifying aggregate.index does the same thing for replicate weight designs; a warning will be given if the original weights are not constant within levels of aggregate.index.

In a model with two-stage sampling, population totals may be available for the PSUs actually sampled, but not for the whole population. In this situation, calibrating within each PSU reduces with second-stage contribution to variance. This generalizes to multistage sampling. The stage argument specifies which stage of sampling the totals refer to. Stage 0 is full population totals, stage 1 is totals for PSUs, and so on. The default, stage=NULL is interpreted as stage 0 when a single population vector is supplied and stage 1 when a list is supplied. Calibrating to PSU totals will fail (with a message about an exactly singular matrix) for PSUs that have fewer observations than the number of calibration variables.

The variance in the calibration model may depend on covariates. If variance=NULL the calibration model has constant variance. If variance is not NULL it specifies a linear combination of the columns of the model matrix and the calibration variance is proportional to that linear combination. Alternatively variance can be specified as a vector of values the same length as the input data specifying a heteroskedasticity parameter for each unit.

The design matrix specified by formula (after any aggregation) must be of full rank, with one exception. If the population total for a column is zero and all the observations are zero the column will be ignored. This allows the use of factors where the population happens to have no observations at some level.

In a two-phase design, population may be omitted when phase=2, to specify calibration to the phase-one sample. If the two-phase design object was constructed using the more memory-efficient
method="approx" argument to `twophase`, calibration of the first phase of sampling to the population is not supported.

**Value**

A survey design object.

**References**


Lumley T, Shaw PA, Dai JY (2011) "Connections between survey calibration estimators and semiparametric models for incomplete data" International Statistical Review. 79:200-220. (with discussion 79:221-232)


**See Also**

`postStratify`, `rake` for other ways to use auxiliary information

`twophase` and vignette("epi") for an example of calibration in two-phase designs


`make.calfun` for user-defined calibration distances.

`trimWeights` to trim final weights rather than calibration adjustments.

**Examples**

data(api)
dclus1<-svydesign(id=~dnum, weights=~pw, data=apiclus1, fpc=~fpc)

cal_names(~stype, dclus1)

pop.totals<-c(`(Intercept)`=6194, stypeH=755, stypeM=1018)

## For a single factor variable this is equivalent to
## `postStratify`

(dclus1g<-calibrate(dclus1, ~stype, pop.totals))

svymean(~api00, dclus1g)
svytotal(~enroll, dclus1g)
svytotal(~stype, dclus1g)

## Make weights constant within school district
(dclus1agg2 <- calibrate(dclus1, ~stype, pop.totals, aggregate=1))
svymean(~api00, dclus1agg2)
svytotal(~enroll, dclus1agg2)
svytotal(~stype, dclus1agg2)

## Now add sch.wide

### Finally, calibrate on 1999 API and school type

cal_names(~stype+api99, dclus1)
(dclus1g3 <- calibrate(dclus1, ~stype+api99, c(pop.totals, api99=3914069)))
svymean(~api00, dclus1g3)
svytotal(~enroll, dclus1g3)
svytotal(~stype, dclus1g3)

## Same syntax with replicate weights

rclus1 <- as.svrepdesign(dclus1)
(rclus1g3 <- calibrate(rclus1, ~stype+api99, c(pop.totals, api99=3914069)))
svymean(~api00, rclus1g3)
svytotal(~enroll, rclus1g3)
svytotal(~stype, rclus1g3)

## Bounded weights

### Bounded weights
range(weights(dclus1g3)/weights(dclus1))
dclus1g3b <- calibrate(dclus1, ~stype+api99, c(pop.totals, api99=3914069), bounds=c(0.6,1.6))
range(weights(dclus1g3b)/weights(dclus1))
svymean(~api00, dclus1g3b)
svytotal(~enroll, dclus1g3b)
svytotal(~stype, dclus1g3b)
## Individual boundary constraints as constant values

The first weight will be bounded at 40, the rest free to move.

```r
bnds <- list(
  lower = rep(-Inf, nrow(apiclus1)),
  upper = c(40, rep(Inf, nrow(apiclus1)-1)))
```

```r
dclusl1g3b1 <- calibrate(dclusl1, ~stype+api99, c(pop.totals, api99=3914069),
                         bounds=bnds, bounds.const=TRUE)
```

```r
svytotal(~api.stu, dclusl1g3b1)
```

## trimming

```r
dclusl1tr <- calibrate(dclusl1, ~stype+api99, c(pop.totals, api99=3914069),
                       bounds=c(0.5,2), trim=c(2/3,3/2))
```

```r
svytotal(~stype,dclusl1tr)
```

```r
range(weights(dclusl1tr)/weights(dclusl1))
```

```r
rclusl1tr <- calibrate(rclusl1, ~stype+api99, c(pop.totals, api99=3914069),
                        bounds=c(0.5,2), trim=c(2/3,3/2))
```

```r
svytotal(~stype,rclusl1tr)
```

## Input in the same format as rake() for classical raking

```r
pop.table <- xtabs(~stype+sch.wide, apipop)
```

```r
pop.table2 <- xtabs(~stype+comp.imp, apipop)
```

```r
dclusl1r<-rake(dclusl1, list(~stype+sch.wide, ~stype+comp.imp),
                list(pop.table, pop.table2))
```

```r
gclusl1r<-calibrate(dclusl1, formula=list(~stype+sch.wide, ~stype+comp.imp),
                      population=list(pop.table, pop.table2), calfun="raking")
```

```r
svytotal(~stype, dclusl1r)
```

```r
svytotal(~stype, gclusl1r)
```

## generalised raking

```r
dclusl1g3c <- calibrate(dclusl1, ~stype+api99, c(pop.totals, api99=3914069),
                        calfun="raking")
```

```r
range(weights(dclusl1g3c)/weights(dclusl1))
```

```r
(dclusl1g3d <- calibrate(dclusl1, ~stype+api99, c(pop.totals, api99=3914069),
                         calfun=cal.logit, bounds=c(0.5,2.5)))
```

```r
range(weights(dclusl1g3d)/weights(dclusl1))
```

## Ratio estimators are calibration estimators

```r
dstrat<-svydesign(id=~1,strata=~stype, weights=~pw, data=apistrat, fpc=~fpc)
```

```r
svytotal(~api.stu,dstrat)
```

```r
common<-svyratio(~api.stu, ~enroll, dstrat, separate=FALSE)
predict(common, total=3811472)
```

```r
pop<-3811472
```

```r
## Individual boundary constraints as constant values
```
## equivalent to (common) ratio estimator

dstratg1 <- calibrate(dstrat, enroll - 1, pop, variance = 1)
svytotal(~api.stu, dstratg1)

# Alternatively specifying the heteroskedasticity parameters directly

dstratgh <- calibrate(dstrat, enroll - 1, pop, variance = apistrat$enroll)
svytotal(~api.stu, dstratgh)

---

### compressWeights

**Compress replicate weight matrix**

**Description**

Many replicate weight matrices have redundant rows, such as when weights are the same for all observations in a PSU. This function produces a compressed form. Methods for `as.matrix` and `as.vector` extract and expand the weights.

**Usage**

```r
compressWeights(rw, ...)  
## S3 method for class 'svyrep.design'
compressWeights(rw, ...)
## S3 method for class 'repweights_compressed'
as.matrix(x, ...)
## S3 method for class 'repweights_compressed'
as.vector(x, ...)
```

**Arguments**

- `rw` A set of replicate weights or a `svyrep.design` object
- `x` A compressed set of replicate weights
- `...` For future expansion

**Value**

An object of class `repweights_compressed` or a `svyrep.design` object with `repweights` element of class `repweights_compressed`

**See Also**

- `jknweights`, `as.svrepdesign`

**Examples**

```r
data(api)
dclus1 <- svydesign(id = ~dnum, weights = ~pw, data = apiclus1, fpc = ~fpc)
rclus1c <- as.svrepdesign(dclus1, compress = TRUE)
rclus1f <- as.svrepdesign(dclus1, compress = FALSE)
```
Description

Computes confidence intervals for regression parameters in \texttt{svyglm} objects. The default is a Wald-type confidence interval, adding and subtracting a multiple of the standard error. The \texttt{method="likelihood"} is an interval based on inverting the Rao-Scott likelihood ratio test. That is, it is an interval where the working model deviance is lower than the threshold for the Rao-Scott test at the specified level.

Usage

\begin{verbatim}
## S3 method for class 'svyglm'
confint(object, parm, level = 0.95, method = c("Wald", "likelihood"), ddf = Inf, ...)
\end{verbatim}

Arguments

- \texttt{object} \texttt{svyglm} object
- \texttt{parm} numeric or character vector indicating which parameters to construct intervals for.
- \texttt{level} desired coverage
- \texttt{method} See description above
- \texttt{ddf} Denominator degrees of freedom for "likelihood" method, to use a t distribution rather than normal. If NULL, use \texttt{object$df.residual}
- ... for future expansion

Value

A matrix of confidence intervals

References


See Also

\texttt{confint}

Examples

\begin{verbatim}
data(api)
dclus2<-svydesign(id=~dnum+snum, fpc=~fpc1+fpc2, data=apiclus2)

m<-svyglm(I(comp.imp=="Yes")~stype*emer+ell, design=dclus2, family=quasibinomial)

confint(m)
confint(m, method="like", ddf=NULL, parm=c("ell","emer"))
\end{verbatim}
crowd

Household crowding

Description

A tiny dataset from the VPLX manual.

Usage

data(crowd)

Format

A data frame with 6 observations on the following 5 variables.

- `rooms`: Number of rooms in the house
- `person`: Number of people in the household
- `weight`: Sampling weight
- `cluster`: Cluster number
- `stratum`: Stratum number

Source

Manual for VPLX, Census Bureau.

Examples

data(crowd)

## Example 1-1
i1.1<-as.svrepdesign(svydesign(id=~cluster, weight=~weight,data=crowd))
i1.1<-update(i1.1, room.ratio=rooms/person, overcrowded=factor(person>rooms))
svymean(~rooms+person+room.ratio,i1.1)
svytotal(~rooms+person+room.ratio,i1.1)
svymean(~rooms+person+room.ratio,subset(i1.1,overcrowded==TRUE))
svytotal(~rooms+person+room.ratio,subset(i1.1,overcrowded==TRUE))

## Example 1-2
i1.2<-as.svrepdesign(svydesign(id=~cluster,weight=~weight,strata=~stratum, data=crowd))
svymean(~rooms+person,i1.2)
svytotal(~rooms+person,i1.2)
**dimnames.DBIsvydesign**  
*Dimensions of survey designs*

**Description**

`dimnames` returns variable names and row names for the data variables in a design object and `dim` returns dimensions. For multiple imputation designs there is a third dimension giving the number of imputations. For database-backed designs the second dimension includes variables defined by `update`. The first dimension excludes observations with zero weight.

**Usage**

```r
## S3 method for class 'survey.design'
dim(x)
## S3 method for class 'svyimputationList'
dim(x)
## S3 method for class 'survey.design'
dimnames(x)
## S3 method for class 'DBIsvydesign'
dimnames(x)
## S3 method for class 'svyimputationList'
dimnames(x)
```

**Arguments**

- **x**  
  Design object

**Value**

A vector of numbers for `dim`, a list of vectors of strings for `dimnames`.

**See Also**

`update.DBIsvydesign`, `with.svyimputationList`

**Examples**

```r
data(api)
dclus1 <- svydesign(ids=~dnum, weights=~pw, data=apiclus1, fpc=~fpc)
dim(dclus1)
dimnames(dclus1)
colnames(dclus1)
```
Description

A sample of voting data from US states or counties (depending on data availability), sampled with probability proportional to number of votes. The sample was drawn using Tille’s splitting method, implemented in the “sampling” package.

Usage

data(election)

Format

election is a data frame with 4600 observations on the following 8 variables.

County  A factor specifying the state or country
TotPrecincts  Number of precincts in the state or county
PrecinctsReporting  Number of precincts supplying data
Bush  Votes for George W. Bush
Kerry  Votes for John Kerry
Nader  Votes for Ralph Nader
votes  Total votes for those three candidates
p  Sampling probability, proportional to votes

election_pps is a sample of 40 counties or states taken with probability proportional to the number of votes. It includes the additional column wt with the sampling weights.
election_insample indicates which rows of election were sampled.
election_jointprob are the pairwise sampling probabilities and election_jointHR are approximate pairwise sampling probabilities using the Hartley-Rao approximation.

Source
.

Examples

data(election)
## high positive correlation between totals
plot(Bush~Kerry, data=election, log="xy")
## high negative correlation between proportions
plot(I(Bush/votes)~I(Kerry/votes), data=election)

## Variances without replacement
## Horvitz-Thompson type
estweights

Estimated weights for missing data

Description

Creates or adjusts a two-phase survey design object using a logistic regression model for second-phase sampling probability. This function should be particularly useful in reweighting to account for missing data.

Usage

estWeights(data, formula, ...)  
## S3 method for class 'twophase'
estWeights(data, formula=NULL, working.model=NULL, ...)
## S3 method for class 'data.frame'
estWeights(data, formula=NULL, working.model=NULL,
            subset=NULL, strata=NULL, ...)

Arguments

data  twophase design object or data frame
formula  Predictors for estimating weights
working.model  Model fitted to complete (ie phase 1) data
subset  Subset of data frame with complete data (ie phase 1). If NULL use all complete cases
strata  Stratification (if any) of phase 2 sampling
...  for future expansion
Details

If `data` is a data frame, `estWeights` first creates a two-phase design object. The `strata` argument is used only to compute finite population corrections, the same variables must be included in `formula` to compute stratified sampling probabilities.

With a two-phase design object, `estWeights` estimates the sampling probabilities using logistic regression as described by Robins et al (1994) and adds information to the object to enable correct sandwich standard errors to be computed.

An alternative to specifying `formula` is to specify `working.model`. The estimating functions from this model will be used as predictors of the sampling probabilities, which will increase efficiency to the extent that the working model and the model of interest estimate the same parameters (Kulich \& Lin 2004).

The effect on a two-phase design object is very similar to `calibrate`, and is identical when `formula` specifies a saturated model.

Value

A two-phase survey design object.

References


Lumley T, Shaw PA, Dai JY (2011) "Connections between survey calibration estimators and semiparametric models for incomplete data" International Statistical Review. 79:200-220. (with discussion 79:221-232)

See Also

`postStratify`, `calibrate`, `twophase`

Examples

data(airquality)

## ignoring missingness, using model-based standard error
summary(lm(log(Ozone)~Temp+Wind, data=airquality))

## Without covariates to predict missingness we get
## same point estimates, but different (sandwich) standard errors
daq<-estWeights(airquality, formula=~1,subset=~I(!is.na(Ozone)))
summary(svyglm(log(Ozone)~Temp+Wind,design=daq))

## Reweighting based on weather, month
d2aq<-estWeights(airquality, formula=~Temp+Wind+Month,
The \textit{fpc} data frame has 8 rows and 6 columns. It is artificial data to illustrate survey sampling estimators.

\textbf{Usage}

\begin{verbatim}
data(fpc)
\end{verbatim}

\textbf{Format}

This data frame contains the following columns:

- \texttt{stratid}  Stratum ids
- \texttt{psuid}   Sampling unit ids
- \texttt{weight}  Sampling weights
- \texttt{nh}      number sampled per stratum
- \texttt{Nh}      population size per stratum
- \texttt{x}       data

\textbf{Source}

http://www.stata-press.com/data/r7/fpc.dta

\textbf{Examples}

\begin{verbatim}
data(fpc)
fpc

withoutfpc<-svydesign(weights=~weight, ids=~psuid, strata=~stratid, variables=~x, data=fpc, nest=TRUE)

withoutfpc
svymean(~x, withoutfpc)

withfpc<-svydesign(weights=~weight, ids=~psuid, strata=~stratid, fpc=~Nh, variables=~x, data=fpc, nest=TRUE)

withfpc
svymean(~x, withfpc)
\end{verbatim}
## Other equivalent forms

```r
withfpc<-svydesign(prob=~I(1/weight), ids=~psuid, strata=~stratid, fpc=~Nh, variables=~x, data=fpc, nest=TRUE)
svymean(~x, withfpc)

withfpc<-svydesign(weights=~weight, ids=~psuid, strata=~stratid, fpc=~I(nh/Nh), variables=~x, data=fpc, nest=TRUE)
svymean(~x, withfpc)

withfpc<-svydesign(weights=~weight, ids=~interaction(stratid,psuid), strata=~stratid, fpc=~I(nh/Nh), variables=~x, data=fpc)
svymean(~x, withfpc)

withfpc<-svydesign(ids=~psuid, strata=~stratid, fpc=~Nh, variables=~x, data=fpc, nest=TRUE)
svymean(~x, withfpc)

withfpc<-svydesign(ids=~psuid, strata=~stratid, fpc=~I(nh/Nh), variables=~x, data=fpc, nest=TRUE)
svymean(~x, withfpc)
```

---

### ftable.svystat

**Lay out tables of survey statistics**

#### Description

Reformat the output of survey computations to a table.

#### Usage

```r
## S3 method for class 'svystat'
ftable(x, rownames,...)
## S3 method for class 'svrepstat'
ftable(x, rownames,...)
## S3 method for class 'svyby'
ftable(x,...)
```

#### Arguments

- **x**
  - Output of functions such as svymean, svrepmean, svyby
hadamard

rownames List of vectors of strings giving dimension names for the resulting table (see examples)

... Arguments for future expansion

Value

An object of class "ftable"

See Also

ftable

Examples

data(api)
dclus1<-svydesign(id=~dnum, weights=~pw, data=apiclus1, fpc=~fpc)
a<-svymean(~interaction(stype,comp.imp), design=dclus1)
b<-ftable(a, rownames=list(stype=c("E","H","M"),comp.imp=c("No","Yes")))
b

a<-svymean(~interaction(stype,comp.imp), design=dclus1, deff=TRUE)
b<-ftable(a, rownames=list(stype=c("E","H","M"),comp.imp=c("No","Yes")))
round(100*b,1)

rclus1<-as.svrepdesign(dclus1)
a<-svytotal(~interaction(stype,comp.imp), design=rclus1)
b<-ftable(a, rownames=list(stype=c("E","H","M"),comp.imp=c("No","Yes")))
b
round(b)

a<-svyby(~api99 + api00, ~stype + sch.wide, rclus1, svymean, keep.var=TRUE)
ftable(a)
print(ftable(a),digits=2)

b<-svyby(~api99 + api00, ~stype + sch.wide, rclus1, svymean, keep.var=TRUE, deff=TRUE)
print(ftable(b),digits=2)

d<-svyby(~api99 + api00, ~stype + sch.wide, rclus1, svymean, keep.var=TRUE, vartype=c("se","cvpct"))
round(ftable(d),1)

hadamard

Hadamard matrices

Description

Returns a Hadamard matrix of dimension larger than the argument.
hadamard

Usage

hadamard(n)

Arguments

n
lower bound for size

Details

For most n the matrix comes from 
paley. The $36 \times 36$ matrix is from Plackett and Burman (1946)
and the $28 \times 28$ is from Sloane's library of Hadamard matrices.

Matrices of dimension every multiple of 4 are thought to exist, but this function
doesn't know about all of them, so it will sometimes return matrices that are larger than necessary. The excess is at most
4 for $n < 180$ and at most 5% for $n > 100$.

Value

A Hadamard matrix

Note

Strictly speaking, a Hadamard matrix has entries +1 and -1 rather than 1 and 0, so $2 \times \text{hadamard}(n) - 1$
is a Hadamard matrix

References

Sloane NJA. A Library of Hadamard Matrices http://neilsloane.com/hadamard/
Plackett RL., Burman JP. (1946) The Design of Optimum Multifactorial Experiments Biometrika,
Vol. 33, No. 4 pp. 305-325

See Also

brrweights, paley

Examples

par(mfrow=c(2,2))
## Sylvester-type
image(hadamard(63),main=quote("Sylvester: "*64==2^6))
## Paley-type
image(hadamard(59),main=quote("Paley: "*60==59+1))
## from NJ Sloane's library
image(hadamard(27),main=quote(" Stored: "*28))
## For n=90 we get 96 rather than the minimum possible size, 92.
image(hadamard(90),main=quote(" Constructed: "*96==2*3^2*(11+1))))
hospital

Sample of obstetric hospitals

Description

The hospital data frame has 15 rows and 5 columns.

Usage

data(hospital)

Format

This data frame contains the following columns:

- **hospno**  Hospital id
- **oblevel** level of obstetric care
- **weighta** Weights, as given by the original reference
- **tothosp** total hospitalisations
- **births** births
- **weightats** Weights, as given in the source

Source

http://www.ats.ucla.edu/stat/books/sop/hospsamp.dta

References


Examples

data(hospital)
hospdes<-svydesign(strata=~oblevel, id=~hospno, weights=~weighta, fpc=~tothosp, data=hospital)
hosprep<-as.svrepdesign(hospdes)
svytotal(~births, design=hospdes)
svytotal(~births, design=hosprep)
Wrappers for specifying PPS designs

Description

The Horvitz-Thompson estimator and the Hartley-Rao approximation require information in addition to the sampling probabilities for sampled individuals. These functions allow this information to be supplied.

Usage

HR(psum=NULL, strata = NULL)
ppsmat(jointprob, tolerance = 1e-04)

Arguments

- **psum**: The sum of squared sampling probabilities for the population, divided by the sample size, as a single number or as a vector for stratified sampling.
- **strata**: Stratum labels, of the same length as `psum`, if `psum` is a vector.
- **jointprob**: Matrix of pairwise sampling probabilities for the sampled individuals.
- **tolerance**: Tolerance for deciding that the covariance of sampling indicators is zero.

Value

An object of class `HR` or `ppsmat`, suitable for supplying as the `pps` argument to `svydesign`.

See Also

- `election` for examples of PPS designs

Examples

HR(0.1)

Calibration metrics

Description

Create calibration metric for use in `calibrate`. The function \( F \) is the link function described in section 2 of Deville et al. To create a new calibration metric, specify \( F - 1 \) and its derivative. The package provides `cal.linear`, `cal.raking`, `cal.logit`, which are standard, and `cal.sinh` from the CALMAR2 macro, for which \( F \) is the derivative of the inverse hyperbolic sine.
Usage

make.calfun(Fm1, dF, name)

Arguments

Fm1 Function \( F - 1 \) taking a vector \( u \) and a vector of length 2, bounds.

dF Derivative of \( Fm1 \) wrt \( u \): arguments \( u \) and bounds

name Character string to use as name

Value

An object of class "calfun"

References


See Also

calibrate

Examples

str(cal.linear)
cal.linear$Fm1
cal.linear$dF

hellinger <- make.calfun(Fm1=function(u, bounds) ((1-u/2)^-2)-1,
dF= function(u, bounds) (1-u/2)^-3 ,
name="hellinger distance")

hellinger

data(api)
dclus1<-svydesign(id=~dnum, weights=~pw, data=apiclus1, fpc=~fpc)

svymean(~api00, calibrate(dclus1, ~api99, pop=c(6194, 3914069),
calfun=hellinger))

svymean(~api00, calibrate(dclus1, ~api99, pop=c(6194, 3914069),
calfun=cal.linear))

svymean(~api00, calibrate(dclus1, ~api99, pop=c(6194,3914069),
calfun=cal.raking))
Standardised predictions (predictive margins) for regression models.

**Description**

Reweights the design (using `calibrate`) so that the adjustment variables are uncorrelated with the variables in the model, and then performs predictions by calling `predict`. When the adjustment model is saturated this is equivalent to direct standardization on the adjustment variables.

The `svycoxph` and `svykmlist` methods return survival curves.

**Usage**

```r
marginpred(model, adjustfor, predictat, ...) 
## S3 method for class 'svycoxph'
marginpred(model, adjustfor, predictat, se=FALSE, ...)
## S3 method for class 'svykmlist'
marginpred(model, adjustfor, predictat, se=FALSE, ...)
## S3 method for class 'svyglm'
marginpred(model, adjustfor, predictat, ...) 
```

**Arguments**

- **model**: A regression model object of a class that has a `marginpred` method
- **adjustfor**: Model formula specifying adjustment variables, which must be in the design object of the model
- **predictat**: A data frame giving values of the variables in `model` to predict at
- **se**: Estimate standard errors for the survival curve (uses a lot of memory if the sample size is large)
- **...**: Extra arguments, passed to the `predict` method for `model`

**See Also**

- `svypredmeans` for the method of Graubard and Korn implemented in SUDAAN.
- `calibrate`
- `predict.svycoxph`

**Examples**

```r
## generate data with apparent group effect from confounding
set.seed(42)
df<-data.frame(x=rnorm(100))
df$time<-rexp(100)*exp(df$x-1)
df$status<-1
df$group<-(df$x+rnorm(100))>0
des<-svydesign(id=~1,data=df)
newdf<-data.frame(group=c(FALSE,TRUE), x=c(0,0))
```
## Cox model

```r
m0 <- svycoxph(Surv(time, status) ~ group, design=des)
m1 <- svycoxph(Surv(time, status) ~ group + x, design=des)
## conditional predictions, unadjusted and adjusted
cpred0 <- predict(m0, type="curve", newdata=newdf, se=TRUE)
cpred1 <- predict(m1, type="curve", newdata=newdf, se=TRUE)
## adjusted marginal prediction
mpred <- marginpred(m0, adjustfor=~x, predictat=newdf, se=TRUE)
```

```r
plot(cpred0)
lines(cpred1[[1]], col="red")
lines(cpred1[[2]], col="red")
lines(mpred[[1]], col="blue")
lines(mpred[[2]], col="blue")
```

## Kaplan--Meier

```r
s2 <- svykm(Surv(time, status>0) ~ group, design=des)
p2 <- marginpred(s2, adjustfor=~x, predictat=newdf, se=TRUE)
plot(s2)
lines(p2[[1]], col="green")
lines(p2[[2]], col="green")
```

## logistic regression

```r
logisticm <- svyglm(group ~ time, family=quasibinomial, design=des)
newdf$times <- c(0.1, 0.8)
logisticpred <- marginpred(logisticm, adjustfor=~x, predictat=newdf)
```

---

### mu284

**Two-stage sample from MU284**

---

#### Description

The MU284 population comes from Sarndal et al, and the complete data are available from Statlib. These data are a two-stage sample from the population, analyzed on page 143 of the book.

#### Usage

```r
data(mu284)
```

#### Format

A data frame with 15 observations on the following 5 variables.

- `id1` identifier for PSU
- `n1` number of PSUs in population
- `id2` identifier for second-stage unit
- `y1` variable to be analysed
- `n2` number of second-stage units in this PSU
nhanes

Source
(downloaded from StatLib, which is no longer active)

Examples

data(mu284)
(dmu284<-svydesign(id=~id1+id2,fpc=~n1+n2, data=mu284))
ytotal<-svytotal(~y1, dmu284))
vcov(ytotal)

nhanes Cholesterol data from a US survey

Description
Data extracted from NHANES 2009-2010 on high cholesterol.

Usage
data(nhanes)

Format
A data frame with 8591 observations on the following 7 variables.
SDMVPSU  Primary sampling units
SDMVSTRA  Sampling strata
WTMEC2YR  Sampling weights
HI_CHOL  Numeric vector: 1 for total cholesterol over 240mg/dl, 0 under 240mg/dl
race  1=Hispanic, 2=non-Hispanic white, 3=non-Hispanic black, 4=other
agecat  Age group (0,19] (19,39] (39,59] (59,Inf]
RIAGENDR  Gender: 1=male, 2=female

Source

Examples
data(nhanes)
design <- svydesign(id=~SDMVPSU, strata=~SDMVSTRA, weights=~WTMEC2YR, nest=TRUE,data=nhanes)
design
nonresponse  Experimental: Construct non-response weights

Description

Functions to simplify the construction of non-response weights by combining strata with small numbers or large weights.

Usage

nonresponse(sample.weights, sample.counts, population)
sparsesCells(object, count=0, totalweight=Inf, nrweight=1.5)
neighbours(index, object)
joinCells(object, a, ...)
## S3 method for class 'nonresponse'
weights(object, ...)

Arguments

sample.weights  table of sampling weight by stratifying variables
sample.counts  table of sample counts by stratifying variables
population  table of population size by stratifying variables
object  object of class "nonresponse"
count  Cells with fewer sampled units than this are "sparse"
nrweight  Cells with higher non-response weight than this are "sparse"
totalweight  Cells with average sampling weight times non-response weight higher than this are "sparse"
index  Number of a cell whose neighbours are to be found
a, ...  Cells to join

Details

When a stratified survey is conducted with imperfect response it is desirable to rescale the sampling weights to reflect the nonresponse. If some strata have small sample size, high non-response, or already had high sampling weights it may be desirable to get less variable non-response weights by averaging non-response across strata. Suitable strata to collapse may be similar on the stratifying variables and/or on the level of non-response.

nonresponse() combines stratified tables of population size, sample size, and sample weight into an object. sparseCells identifies cells that may need combining. neighbours describes the cells adjacent to a specified cell, and joinCells collapses the specified cells. When the collapsing is complete, use weights() to extract the nonresponse weights.

Value

nonresponse and joinCells return objects of class "nonresponse", neighbours and sparseCells return objects of class "nonresponseSubset"
Examples

data(api)
## pretend the sampling was stratified on three variables
poptable<-xtabs(~sch.wide+comp.imp+stype,data=apipop)
sample.count<-xtabs(~sch.wide+comp.imp+stype,data=apiclus1)
sample.weight<-xtabs(pw~sch.wide+comp.imp+stype, data=apiclus1)

## create a nonresponse object
nr<-nonresponse(sample.weight,sample.count, poptable)

## sparse cells
sparseCells(nr)

## Look at neighbours
neighbours(3,nr)
neighbours(11,nr)

## Collapse some contiguous cells
nr1<-joinCells(nr,3,5,7)

## sparse cells now
sparseCells(nr1)
nr2<-joinCells(nr1,3,11,8)

## one relatively sparse cell
sparseCells(nr2)
## but nothing suitable to join it to
neighbours(3,nr2)

## extract the weights
weights(nr2)

open.DBIsvydesign

Open and close DBI connections

Description

A database-backed survey design object contains a connection to a database. This connection will be broken if the object is saved and reloaded, and the connection should ideally be closed with close before quitting R (although it doesn’t matter for SQLite connections). The connection can be reopened with open.

Usage

## S3 method for class 'DBIsvydesign'
open(con, ...)
## S3 method for class 'DBIsvydesign'
close(con, ...)
Arguments

- `con`: Object of class DBIsvydesign
- `...`: Other options, to be passed to `dbConnect` or `dbDisconnect`.

Value

The same survey design object with the connection opened or closed.

See Also

- `svydesign`
- DBI package

Examples

```r
## Not run:
library(RSQLite)
dbclus1<-svydesign(id=~dnum, weights=~pw, fpc=~fpc,
data="apiclus1", dbtype="SQLite",
dbname=system.file("api.db",package="survey"))

dbclus1
close(dbclus1)
dbclus1
try(svymean(~api00, dbclus1))

dbclus1<-open(dbclus1)
open(dbclus1)
svymean(~api00, dbclus1)

## End(Not run)
```

---

**paley**

**Paley-type Hadamard matrices**

Description

Computes a Hadamard matrix of dimension \((p + 1) \times 2^k\), where \(p\) is a prime, and \(p+1\) is a multiple of 4, using the Paley construction. Used by `hadamard`.

Usage

```r
paley(n, nmax = 2 * n, prime=NULL, check=!is.null(prime))

is.hadamard(H, style=c("0/1","+-"), full.orthogonal.balance=TRUE)
```
Arguments

- **n**: Minimum size for matrix
- **nmax**: Maximum size for matrix. Ignored if prime is specified.
- **prime**: Optional. A prime at least as large as n, such that prime+1 is divisible by 4.
- **check**: Check that the resulting matrix is of Hadamard type
- **H**: Matrix
- **style**: "0/1" for a matrix of 0s and 1s, "+-" for a matrix of ±1.
- **full.orthogonal.balance**: Require full orthogonal balance?

Details

The Paley construction gives a Hadamard matrix of order p+1 if p is prime and p+1 is a multiple of 4. This is then expanded to order \((p + 1) \times 2^k\) using the Sylvester construction.

paley knows primes up to 7919. The user can specify a prime with the prime argument, in which case a matrix of order \(p + 1\) is constructed.

If check=TRUE the code uses is.hadamard to check that the resulting matrix really is of Hadamard type, in the same way as in the example below. As this test takes \(n^3\) time it is preferable to just be sure that prime really is prime.

A Hadamard matrix including a row of 1s gives BRR designs where the average of the replicates for a linear statistic is exactly the full sample estimate. This property is called full orthogonal balance.

Value

For paley, a matrix of zeros and ones, or NULL if no matrix smaller than nmax can be found.
For is.hadamard, TRUE if H is a Hadamard matrix.

References


See Also

hadamard

Examples

```r
M<-paley(11)
is.hadamard(M)
## internals of is.hadamard(M)
H<-2*M-1
## HH^T is diagonal for any Hadamard matrix
H%*%t(H)
```
The distribution of a quadratic form in $p$ standard Normal variables is a linear combination of $p$ chi-squared distributions with 1df. When there is uncertainty about the variance, a reasonable model for the distribution is a linear combination of $F$ distributions with the same denominator.

**Usage**

```r
pchisqsum(x, df, a, lower.tail = TRUE,
          method = c("satterthwaite", "integration", "saddlepoint"))
pFsum(x, df, a, ddf=Inf, lower.tail = TRUE,
       method = c("saddlepoint", "integration", "satterthwaite"), ...)
```

**Arguments**

- `x` Observed values
- `df` Vector of degrees of freedom
- `a` Vector of coefficients
- `ddf` Denominator degrees of freedom
- `lower.tail` lower or upper tail?
- `method` See Details below
- `...` arguments to `pchisqsum`

**Details**

The "satterthwaite" method uses Satterthwaite’s approximation, and this is also used as a fallback for the other methods. The accuracy is usually good, but is more variable depending on `a` than the other methods and is anticonservative in the right tail (eg for upper tail probabilities less than $10^{-5}$). The Satterthwaite approximation requires all $a>0$.

"integration" requires the CompQuadForm package. For `pchisqsum` it uses Farebrother’s algorithm if all $a>0$. For `pFsum` or when some $a<0$ it inverts the characteristic function using the algorithm of Davies (1980). These algorithms are highly accurate for the lower tail probability, but they obtain the upper tail probability by subtraction from 1 and so fail completely when the upper tail probability is comparable to machine epsilon or smaller.

If the CompQuadForm package is not present, a warning is given and the saddlepoint approximation is used.

"saddlepoint" uses Kuonen’s saddlepoint approximation. This is moderately accurate even very far out in the upper tail or with some $a=0$ and does not require any additional packages. It is implemented in pure R and so is slower than the "integration" method.

The distribution in `pFsum` is standardised so that a likelihood ratio test can use the same `x` value as in `pchisqsum`. That is, the linear combination of chi-squareds is multiplied by `ddf` and then divided by an independent chi-squared with `ddf` degrees of freedom.
Value

Vector of cumulative probabilities

References

Davies RB (1980) "Algorithm AS 155: The Distribution of a Linear Combination of chi-squared
P. Duchesne, P. Lafaye de Micheaux (2010) "Computing the distribution of quadratic forms: Fur-
ther comparisons between the Liu-Tang-Zhang approximation and exact methods", Computational
Statistics and Data Analysis, Volume 54, (2010), 858-862
Farebrother R.W. (1984) "Algorithm AS 204: The distribution of a Positive Linear Combination of
Kuonen D (1999) Saddlepoint Approximations for Distributions of Quadratic Forms in Normal

See Also

pchisq

Examples

```r
x <- 2.7*rnorm(1001)^2+rnorm(1001)^2+0.3*rnorm(1001)^2
x.thin<-sort(x)[1+(0:50)*20]
p.invert<-pchisqsum(x.thin,df=c(1,1,1),a=c(2.7,1,.3),method="int",lower=FALSE)
p.satt<-pchisqsum(x.thin,df=c(1,1,1),a=c(2.7,1,.3),method="satt",lower=FALSE)
p.sadd<-pchisqsum(x.thin,df=c(1,1,1),a=c(2.7,1,.3),method="sad",lower=FALSE)

plot(p.invert, p.satt,type="l",log="xy")
abline(0,1,lty=2,col="purple")
plot(p.invert, p.sadd,type="l",log="xy")
abline(0,1,lty=2,col="purple")
pchisqsum(20, df=c(1,1,1),a=c(2.7,1,.3), lower.tail=FALSE,method="sad")
pFsum(20, df=c(1,1,1),a=c(2.7,1,.3), ddf=49,lower.tail=FALSE,method="sad")
pFsum(20, df=c(1,1,1),a=c(2.7,1,.3), ddf=1000,lower.tail=FALSE,method="sad")
```

Description

Post-stratification adjusts the sampling and replicate weights so that the joint distribution of a set of
post-stratifying variables matches the known population joint distribution. Use rake when the full
joint distribution is not available.
Usage

postStratify(design, strata, population, partial = FALSE, ...)
## S3 method for class 'svyrep.design'
postStratify(design, strata, population, partial = FALSE, compress=NULL,...)
## S3 method for class 'survey.design'
postStratify(design, strata, population, partial = FALSE, ...)

Arguments

design  A survey design with replicate weights
strata   A formula or data frame of post-stratifying variables, which must not contain missing values.
population  A table, xtabs or data.frame with population frequencies
partial   if TRUE, ignore population strata not present in the sample
compress  Attempt to compress the replicate weight matrix? When NULL will attempt to compress if the original weight matrix was compressed
...      arguments for future expansion

Details

The population totals can be specified as a table with the strata variables in the margins, or as a data frame where one column lists frequencies and the other columns list the unique combinations of strata variables (the format produced by as.data.frame acting on a table object). A table must have named dimnames to indicate the variable names.

Compressing the replicate weights will take time and may even increase memory use if there is actually little redundancy in the weight matrix (in particular if the post-stratification variables have many values and cut across PSUs).

If a svydesign object is to be converted to a replication design the post-stratification should be performed after conversion.

The variance estimate for replication designs follows the same procedure as Valliant (1993) described for estimating totals. Rao et al (2002) describe this procedure for estimating functions (and also the GREG or g-calibration procedure, see calibrate)

Value

A new survey design object.

Note

If the sampling weights are already post-stratified there will be no change in point estimates after postStratify but the standard error estimates will decrease to correctly reflect the post-stratification.

References

See Also

rake, calibrate for other things to do with auxiliary information
compressWeights for information on compressing weights

Examples

data(api)
dclus1<-svydesign(id=~dnum, weights=~pw, data=apiclus1, fpc=~fpc)
rclus1<-as.svrepdesign(dclus1)

svymean(~api00, rclus1)
svytotal(~enroll, rclus1)

# post-stratify on school type
pop.types <- data.frame(stype=c("E","H","M"), Freq=c(4421,755,1018))
# or: pop.types <- xtabs(~stype, data=apipop)
# or: pop.types <- table(stype=apipop$stype)

rclus1p<-postStratify(rclus1, ~stype, pop.types)
summary(rclus1p)
svymean(~api00, rclus1p)
svytotal(~enroll, rclus1p)

## and for svydesign objects
dclus1p<-postStratify(dclus1, ~stype, pop.types)
summary(dclus1p)
svymean(~api00, dclus1p)
svytotal(~enroll, dclus1p)

<table>
<thead>
<tr>
<th>psrsq</th>
<th>Pseudo-Rsquareds</th>
</tr>
</thead>
</table>

Description

Compute the Nagelkerke and Cox–Snell pseudo-r-squared statistics, primarily for logistic regression. A generic function with methods for glm and svyglm. The method for svyglm objects uses the design-based estimators described by Lumley (2017)

Usage

psrsq(object, method = c("Cox-Snell", "Nagelkerke"), ...)

Arguments

  object       A regression model (glm or svyglm)
  method       Which statistic to compute
  ...          For future expansion
Value

Numeric value

References


See Also

AIC.svyglm

Examples

data(api)
dclus2<-svydesign(id=~dnum+snum, weights=~pw, data=apiclus2)

model1<-svyglm(I(sch.wide=="Yes")~ell+meals+mobility+as.numeric(stype),
  design=dclus2, family=quasibinomial())

psrsq(model1, type="Nagelkerke")

---

**rake**  
*Raking of replicate weight design*

**Description**

Raking uses iterative post-stratification to match marginal distributions of a survey sample to known population margins.

**Usage**

```r
rake(design, sample.margins, population.margins, control = list(maxit = 10, epsilon = 1, verbose=FALSE), compress=NULL)
```

**Arguments**

- `design`  
  A survey object

- `sample.margins`  
  list of formulas or data frames describing sample margins, which must not contain missing values

- `population.margins`  
  list of tables or data frames describing corresponding population margins

- `control`  
  `maxit` controls the number of iterations. Convergence is declared if the maximum change in a table entry is less than `epsilon`. If `epsilon<1` it is taken to be a fraction of the total sampling weight.

- `compress`  
  If `design` has replicate weights, attempt to compress the new replicate weight matrix? When `NULL`, will attempt to compress if the original weight matrix was compressed
Details

The sample margins should be in a format suitable for `postStratify`.

Raking (aka iterative proportional fitting) is known to converge for any table without zeros, and for any table with zeros for which there is a joint distribution with the given margins and the same pattern of zeros. The ‘margins’ need not be one-dimensional.

The algorithm works by repeated calls to `postStratify` (iterative proportional fitting), which is efficient for large multiway tables. For small tables `calibrate` will be faster, and also allows raking to population totals for continuous variables, and raking with bounded weights.

Value

A raked survey design.

See Also

`postStratify`, `compressWeights`

`calibrate` for other ways to use auxiliary information.

Examples

data(api)
dclus1 <- svydesign(id=~dnum, weights=~pw, data=apiclus1, fpc=~fpc)
rclus1 <- as.svrepdesign(dclus1)

svymean(~api00, rclus1)
svytotal(~enroll, rclus1)

## population marginal totals for each stratum
pop.types <- data.frame(stype=c("E","H","M"), Freq=c(4421,755,1018))
pop.schwide <- data.frame(sch.wide=c("No","Yes"), Freq=c(1072,5122))

rclus1r <- rake(rclus1, list(~stype,~sch.wide), list(pop.types, pop.schwide))

svymean(~api00, rclus1r)
svytotal(~enroll, rclus1r)

## marginal totals correspond to population
xtabs(~stype, apiopop)
svytable(~stype, rclus1r, round=TRUE)
xtabs(~sch.wide, apiopop)
svytable(~sch.wide, rclus1r, round=TRUE)

## joint totals don’t correspond
xtabs(~stype+sch.wide, apiopop)
svytable(~stype+sch.wide, rclus1r, round=TRUE)

## Do it for a design without replicate weights
dclus1r<-rake(dclus1, list(~stype,~sch.wide), list(pop.types, pop.schwide))

svymean(~api00, dclus1r)
regTermTest

Wald test for a term in a regression model

Description

Provides Wald test and working Wald and working likelihood ratio (Rao-Scott) test of the hypothesis that all coefficients associated with a particular regression term are zero (or have some other specified values). Particularly useful as a substitute for anova when not fitting by maximum likelihood.

Usage

regTermTest(model, test.terms, null=NULL, df=NULL,
method=c("Wald","WorkingWald","LRT"), lrt.approximation="saddlepoint")

Arguments

model A model object with coef and vcov methods
regTermTest

- **test.terms**: Character string or one-sided formula giving name of term or terms to test
- **null**: Null hypothesis values for parameters. Default is zeros
- **df**: Denominator degrees of freedom for an F test. If NULL these are estimated from the model. Use Inf for a chi-squared test.
- **method**: If "Wald", the Wald-type test; if "LRT" the Rao-Scott test based on the estimated log likelihood ratio; if "WorkingWald" the Wald-type test using the variance matrix under simple random sampling
- **lrt.approximation**: method for approximating the distribution of the LRT and Working Wald statistic; see pchisqsum.

Details

The Wald test uses a chisquared or F distribution. The two working-model tests come from the (misspecified) working model where the observations are independent and the weights are frequency weights. For categorical data, this is just the model fitted to the estimated population crosstabulation. The Rao-Scott LRT statistic is the likelihood ratio statistic in this model. The working Wald test statistic is the Wald statistic in this model. The working-model tests do not have a chi-squared sampling distribution: we use a linear combination of chi-squared or F distributions as in pchisqsum. I believe the working Wald test is what SUDAAN refers to as a "Satterthwaite adjusted Wald test".

To match other software you will typically need to use lrt.approximation="satterthwaite"

Value

An object of class regTermTest or regTermTestLRT.

Note

The "LRT" method will not work if the model had starting values supplied for the regression coefficients. Instead, fit the two models separately and use anova(model1,model2,force=TRUE)

References


See Also

anova, vcov, contrasts, pchisqsum
Examples

```r
data(esoph)
model1 <- glm(cbind(ncases, ncontrols) ~ agegp + tobgp * alcgp, data = esoph, family = binomial())
anova(model1)

regTermTest(model1, "tobgp")
regTermTest(model1, "tobgp:alcgp")
regTermTest(model1, ~alcgp+tobgp:alcgp)
```

```r
data(api)
dclus2<–svydesign(id=~dnum+snum, weights=~pw, data=apiclus2)
model2<–svyglm(I(sch.wide=="Yes")~ell+meals+mobility, design=dclus2, family=quasibinomial())
regTermTest(model2, ~ell)
regTermTest(model2, ~ell, df=NULL)
regTermTest(model2, ~ell, method="LRT", df=Inf)
regTermTest(model2, ~ell+meals, method="LRT", df=NULL)
regTermTest(model2, ~ell+meals, method="WorkingWald", df=NULL)
```

scd  

**Survival in cardiac arrest**

Description

These data are from Section 12.2 of Levy and Lemeshow. They describe (a possibly apocryphal) study of survival in out-of-hospital cardiac arrest. Two out of five ambulance stations were sampled from each of three emergency service areas.

Usage

```r
data(scd)
```

Format

This data frame contains the following columns:

- **ESA** Emergency Service Area (strata)
- **ambulance** Ambulance station (PSU)
- **arrests** estimated number of cardiac arrests
- **alive** number reaching hospital alive

Source

Examples

data(scd)

## survey design objects
scddes<-svydesign(data=scd, prob=~1, id=~ambulance, strata=~ESA,
nest=TRUE, fpc=rep(5,6))
scdnofpc<-svydesign(data=scd, prob=~1, id=~ambulance, strata=~ESA,
nest=TRUE)

# convert to BRR replicate weights
scd2brr <- as.svrepdesign(scdnofpc, type="BRR")
# or to Rao-Wu bootstrap
scd2boot <- as.svrepdesign(scdnofpc, type="subboot")

# use BRR replicate weights from Levy and Lemeshow
repweights<-2*xcbind(c(1,0,1,0,1,0), c(1,0,0,1,0,1),
c(0,1,1,0,0,1),
c(0,1,0,1,0,0))
scdrep<-svrepdesign(data=scd, type="BRR", repweights=repweights)

# ratio estimates
svyratio(~alive, ~arrests, design=scddes)
svyratio(~alive, ~arrests, design=scdnofpc)
svyratio(~alive, ~arrests, design=scd2brr)
svyratio(~alive, ~arrests, design=scd2boot)
svyratio(~alive, ~arrests, design=scdrep)

# or a logistic regression
summary(svyglm(cbind(alive,arrests-alive)~1, family=quasibinomial, design=scdnofpc))
summary(svyglm(cbind(alive,arrests-alive)~1, family=quasibinomial, design=scdrep))

# Because no sampling weights are given, can't compute design effects
# without replacement: use deff="replace"
svymean(~alive+arrests, scddes, deff=TRUE)
svymean(~alive+arrests, scddes, deff="replace")

---

**SE**

Extract standard errors

Description

Extracts standard errors from an object. The default method is for objects with a `vcov` method.

Usage

SE(object, ...)

## Default S3 method:
SE(object, ...)

## S3 method for class 'svrepstat'
SE(object, ...)

---
Arguments

object An object
... Arguments for future expansion

Value

Vector of standard errors.

See Also

vcov

stratsample Take a stratified sample

Description

This function takes a stratified sample without replacement from a data set.

Usage

stratsample(strata, counts)

Arguments

strata Vector of stratum identifiers; will be coerced to character
counts named vector of stratum sample sizes, with names corresponding to the values
of as.character(strata)

Value

vector of indices into strata giving the sample

See Also

sample

The "sampling" package has many more sampling algorithms.

Examples

data(api)
s<-stratsample(api$stype, c("E"=5,"H"=4,"M"=2))
table(api$stype[s])
Description

Restrict a survey design to a subpopulation, keeping the original design information about number of clusters, strata. If the design has no post-stratification or calibration data the subset will use proportionately less memory.

Usage

## S3 method for class 'survey.design'
subset(x, subset, ...)

## S3 method for class 'svyrep.design'
subset(x, subset, ...)

Arguments

x A survey design object
subset An expression specifying the subpopulation
... Arguments not used by this method

Value

A new survey design object

See Also

svydesign

Examples

data(fpc)
dfpc<-svydesign(id=psuid, strat=stratid, weight=weight, data=fpc, nest=TRUE)
dsub<-subset(dfpc, x>4)
summary(dsub)
svymean(~x, design=dsub)

## These should give the same domain estimates and standard errors
svyby(~x,-I(x>4),design=dfpc, svymean)
summary(svyglm(x-I(x>4)+0,design=dfpc))

data(api)
dclus1<-svydesign(id=dnum, weights=pw, data=apiclus1, fpc=fpc)
rclus1<-as.svrepdesign(dclus1)
svymean(~enroll, subset(dclus1, sch.wide=="Yes" & comp.imp=="Yes"))
svymean(~enroll, subset(rclus1, sch.wide=="Yes" & comp.imp=="Yes"))
surveyoptions

Options for the survey package

Description

This help page documents the options that control the behaviour of the survey package.

Details

All the options for the survey package have names beginning with "survey". Four of them control standard error estimation.

options("survey.replicates.mse") controls the default in svrepdesign and as.svrepdesign for computing variances. When options("survey.replicates.mse") is TRUE, the default is to create replicate weight designs that compute variances centered at the point estimate, rather than at the mean of the replicates. The option can be overridden by specifying the mse argument explicitly in svrepdesign and as.svrepdesign. The default is FALSE.

When options("survey.ultimate.cluster") is TRUE, standard error estimation is based on independence of PSUs at the first stage of sampling, without using any information about subsequent stages. When FALSE, finite population corrections and variances are estimated recursively. See svyrecvar for more information. This option makes no difference unless first-stage finite population corrections are specified, in which case setting the option to TRUE gives the wrong answer for a multistage study. The only reason to use TRUE is for compatibility with other software that gives the wrong answer.

Handling of strata with a single PSU that are not certainty PSUs is controlled by options("survey.lonely.psu"). The default setting is "fail", which gives an error. Use "remove" to ignore that PSU for variance computation, "adjust" to center the stratum at the population mean rather than the stratum mean, and "average" to replace the variance contribution of the stratum by the average variance contribution across strata. As of version 3.4-2 as.svrepdesign also uses this option.

The variance formulas for domain estimation give well-defined, positive results when a stratum contains only one PSU with observations in the domain, but are not unbiased. If options("survey.adjust.domain.lonely") is TRUE and options("survey.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 replicate-weight designs, nor (currently) for raked, post-stratified, or calibrated designs.

The fourth option is options("survey.want.obsolete"). This controls the warnings about using the deprecated pre-2.9.0 survey design objects.

The behaviour of replicate-weight designs for self-representing strata is controlled by options("survey.drop.replicates"). When TRUE, various optimizations are used that take advantage of the fact that these strata do not contribute to the variance. The only reason ever to use FALSE is if there is a bug in the code for these optimizations.

The fifth option controls the use of multiple processors with the multicore package. This option should not affect the values computed by any of the survey functions. If TRUE, all functions that are able to use multiple processors will do so by default. Using multiple processors may speed up calculations, but need not, especially if the computer is short on memory. The best strategy is probably to experiment with explicitly requesting multicore=TRUE in functions that support it, to see if there is an increase in speed before setting the global option.
surveysummary

Summary statistics for sample surveys

Description

Compute means, variances, ratios and totals for data from complex surveys.

Usage

```r
## S3 method for class 'survey.design'
svymean(x, design, na.rm=FALSE, deff=FALSE, ...)
## S3 method for class 'twophase'
svymean(x, design, na.rm=FALSE, deff=FALSE, ...)
## S3 method for class 'svyrep.design'
svymean(x, design, na.rm=FALSE, rho=NULL,
     return.replicates=FALSE, deff=FALSE, ...)
## S3 method for class 'survey.design'
svyvar(x, design, na.rm=FALSE, ...)
## S3 method for class 'svyrep.design'
svyvar(x, design, na.rm=FALSE, rho=NULL,
     return.replicates=FALSE, ..., estimate.only=FALSE)
## S3 method for class 'survey.design'
svytotal(x, design, na.rm=FALSE, deff=FALSE, ...)
## S3 method for class 'twophase'
svytotal(x, design, na.rm=FALSE, deff=FALSE, ...)
## S3 method for class 'svyrep.design'
svytotal(x, design, na.rm=FALSE, rho=NULL,
     return.replicates=FALSE, deff=FALSE, ...)
## S3 method for class 'svystat'
coef(object, ...)
## S3 method for class 'svrepstat'
coef(object, ...)
## S3 method for class 'svystat'
v cov(object, ...)
## S3 method for class 'svrepstat'
v cov(object, ...)
## S3 method for class 'svystat'
confint(object, parm, level = 0.95, df = Inf, ...)
## S3 method for class 'svrepstat'
confint(object, parm, level = 0.95, df = Inf, ...)
## S3 method for class 'svystat'
cv(object, ...)
## S3 method for class 'svrepstat'
cv(object, ...)
## S3 method for class 'svystat'
deff(object, quietly=FALSE, ...)
make.formula(names)
```

Arguments

- `x` A formula, vector or matrix
design: survey.design or svyrep.design object

na.rm: Should cases with missing values be dropped?

rho: parameter for Fay’s variance estimator in a BRR design

return.replicates: Return the replicate means?

deff: Return the design effect (see below)

object: The result of one of the other survey summary functions

quietly: Don’t warn when there is no design effect computed

estimate.only: Don’t compute standard errors (useful when svyvar is used to estimate the design effect)

parm: a specification of which parameters are to be given confidence intervals, either a vector of numbers or a vector of names. If missing, all parameters are considered.

level: the confidence level required.

df: degrees of freedom for t-distribution in confidence interval, use degf(design) for number of PSUs minus number of strata

...: additional arguments to methods, not currently used

names: vector of character strings

**Details**

These functions perform weighted estimation, with each observation being weighted by the inverse of its sampling probability. Except for the table functions, these also give precision estimates that incorporate the effects of stratification and clustering.

Factor variables are converted to sets of indicator variables for each category in computing means and totals. Combining this with the `interaction` function allows crosstabulations. See `ftable.svystat` for formatting the output.

With `na.rm=TRUE`, all cases with missing data are removed. With `na.rm=FALSE` cases with missing data are not removed and so will produce missing results. When using replicate weights and `na.rm=FALSE` it may be useful to set `options(na.action="na.pass")`, otherwise all replicates with any missing results will be discarded.

The `svytotal` and `svreptotal` functions estimate a population total. Use `predict` on `svyratio` and `svyglm`, to get ratio or regression estimates of totals.

`svyvar` estimates the population variance. The object returned includes the full matrix of estimated population variances and covariances, but by default only the diagonal elements are printed. To display the whole matrix use `as.matrix(v)` or `print(v,covariance=TRUE)`.

The design effect compares the variance of a mean or total to the variance from a study of the same size using simple random sampling without replacement. Note that the design effect will be incorrect if the weights have been rescaled so that they are not reciprocals of sampling probabilities. To obtain an estimate of the design effect comparing to simple random sampling with replacement, which does not have this requirement, use `deff="replace"`. This with-replacement design effect is the square of Kish’s "deft".

The design effect for a subset of a design conditions on the size of the subset. That is, it compares the variance of the estimate to the variance of an estimate based on a simple random sample of the
same size as the subset, taken from the subpopulation. So, for example, under stratified random sampling the design effect in a subset consisting of a single stratum will be 1.0.

The cv function computes the coefficient of variation of a statistic such as ratio, mean or total. The default method is for any object with methods for SE and coef.

make.formula makes a formula from a vector of names. This is useful because formulas as the best way to specify variables to the survey functions.

**Value**

Objects of class "svystat" or "svrepstat", which are vectors with a "var" attribute giving the variance and a "statistic" attribute giving the name of the statistic.

These objects have methods for vcov, SE, coef, confint, svycontrast.

**Author(s)**

Thomas Lumley

**See Also**

svydesign, as.svrepdesign, svrepdesign for constructing design objects.
degf to extract degrees of freedom from a design.
svyquantile for quantiles
ftable.svystat for more attractive tables
svyciprop for more accurate confidence intervals for proportions near 0 or 1.
svyttest for comparing two means.
svycontrast for linear and nonlinear functions of estimates.

**Examples**

data(api)

## one-stage cluster sample
dclus1<-svydesign(id=~dnum, weights=~pw, data=apiclus1, fpc=~fpc)

svymean(~api00, dclus1, deff=TRUE)
svymean(~factor(stype),dclus1)
svymean(~interaction(stype, comp.imp), dclus1)
svyquantile(~api00, dclus1, c(.25,.5,.75))
svytotal(~enroll, dclus1, deff=TRUE)
svyratio(~api.stu, ~enroll, dclus1)

v<-svyvar(~api00+api99, dclus1)

v
print(v, cov=TRUE)
as.matrix(v)
# replicate weights - jackknife (this is slower)
dstrat<-svydesign(id=~1,strata=~stype, weights=~pw,
data=apistrat, fpc=~fpc)
jkstrat<-as.svrepdesign(dstrat)

svymean(~api00, jkstrat)
svymean(~factor(stype),jkstrat)
svyvar(~api00+api99, jkstrat)

svyquantile(~api00, jkstrat, c(.25,.5,.75))
svytotals(~enroll, jkstrat)
svyratio(~api.stu, ~enroll, jkstrat)

# coefficients of variation
cv(svytotal(~enroll,dstrat))
cv(svyratio(~api.stu, ~enroll, jkstrat))

# extracting information from the results
coef(svytotal(~enroll,dstrat))
vcov(svymean(~api00+api99,jkstrat))
SE(svymean(~enroll, dstrat))
confint(svymean(~api00+api00, dclus1))
confint(svymean(~api00+api00, dclus1), df=degf(dclus1))

# Design effect
svymean(~api00, dstrat, deff=TRUE)
svymean(~api00, dstrat, deff="replace")
svymean(~api00, jkstrat, deff=TRUE)
svymean(~api00, jkstrat, deff="replace")
(a<-svytotal(~enroll, dclus1, deff=TRUE))
deff(a)

---

svrepdesign  Specify survey design with replicate weights

Description

Some recent large-scale surveys specify replication weights rather than the sampling design (partly
for privacy reasons). This function specifies the data structure for such a survey.

Usage

svrepdesign(variables , repweights , weights , data,...)
## Default S3 method:
svrepdesign(variables = NULL, repweights = NULL, weights = NULL,
data = NULL, type = c("BRR", "Fay", "JK1", "JKn","bootstrap",
"ACS","successive-difference","JK2","other"),

---

svrepdesign  Specify survey design with replicate weights

Description

Some recent large-scale surveys specify replication weights rather than the sampling design (partly
for privacy reasons). This function specifies the data structure for such a survey.

Usage

svrepdesign(variables , repweights , weights , data,...)
## Default S3 method:
svrepdesign(variables = NULL, repweights = NULL, weights = NULL,
data = NULL, type = c("BRR", "Fay", "JK1", "JKn","bootstrap",
"ACS","successive-difference","JK2","other"),
combined.weights=TRUE, rho = NULL, bootstrap.average=NULL, 
  scale=NULL, rscales=NULL,fpc=NULL, fpctype=c("fraction","correction"),
  mse=getOption("survey.replicates.mse"),...)
## S3 method for class 'imputationList'
svrepdesign(variables=NULL, repweights,weights,data,
  mse=getOption("survey.replicates.mse"),...)
## S3 method for class 'character'
svrepdesign(variables=NULL,repweights=NULL, weights=NULL,data=NULL,
  type=c("BRR","Fay","JK1", "JKn","bootstrap","ACS","successive-difference","JK2","other"),
  combined.weights=TRUE, rho=NULL, bootstrap.average=NULL, scale=NULL,rscales=NULL,
  fpc=NULL,fpctype=c("fraction","correction"),mse=getOption("survey.replicates.mse"),
  dbname,...)
## S3 method for class 'svyrep.design'
image(x, ..., 
  col=grey(seq(.5,1,length=30)), type.=c("rep","total"))

Arguments

variables formula or data frame specifying variables to include in the design (default is all)
repweights formula or data frame specifying replication weights, or character string specifying a regular expression that matches the names of the replication weight variables
weights sampling weights
data data frame to look up variables in formulas, or character string giving name of database table
type Type of replication weights
combined.weights TRUE if the repweights already include the sampling weights. This is usually the case.
rho Shrinkage factor for weights in Fay’s method
bootstrap.average For type="bootstrap", if the bootstrap weights have been averaged, gives the number of iterations averaged over
scale, rscales Scaling constant for variance, see Details below
fpc,fpctype Finite population correction information
mse If TRUE, compute variances based on sum of squares around the point estimate, rather than the mean of the replicates
dbname name of database, passed to DBI::dbConnect()
dbtype Database driver: see Details
x survey design with replicate weights
... Other arguments to image
col Colors
type. "rep" for only the replicate weights, "total" for the replicate and sampling weights combined.
Details

In the BRR method, the dataset is split into halves, and the difference between halves is used to estimate the variance. In Fay’s method, rather than removing observations from half the sample they are given weight $\rho$ in one half-sample and $2-\rho$ in the other. The ideal BRR analysis is restricted to a design where each stratum has two PSUs, however, it has been used in a much wider class of surveys. The scale and $rscales$ arguments will be ignored (with a warning) if they are specified.

The JK1 and JKn types are both jackknife estimators deleting one cluster at a time. JKn is designed for stratified and JK1 for unstratified designs.

The successive-difference weights in the American Community Survey automatically use $scale = 4/ncol(repweights)$ and $rscales=rep(1,ncol(repweights))$. This can be specified as type="ACS" or type="successive-difference". The scale and $rscales$ arguments will be ignored (with a warning) if they are specified.

JK2 weights (type="JK2"), as in the California Health Interview Survey, automatically use $scale=1$, $rscales=rep(1,ncol(repweights))$. The scale and $rscales$ arguments will be ignored (with a warning) if they are specified.

Averaged bootstrap weights ("mean bootstrap") are used for some surveys from Statistics Canada. Yee et al (1999) describe their construction and use for one such survey.

The variance is computed as the sum of squared deviations of the replicates from their mean. This may be rescaled: $scale$ is an overall multiplier and $rscales$ is a vector of replicate-specific multipliers for the squared deviations. That is, $rscales$ should have one entry for each column of $repweights$ if the replication weights incorporate the sampling weights (combined.weights=TRUE) or for type="other" these must be specified, otherwise they can be guessed from the weights.

A finite population correction may be specified for type="other", type="JK1" and type="JKn". $fpc$ must be a vector with one entry for each replicate. To specify sampling fractions use $fpctype="fraction"$ and to specify the correction directly use $fpctype="correction"$

$repweights$ may be a character string giving a regular expression for the replicate weight variables. For example, in the California Health Interview Survey public-use data, the sampling weights are "rakedw0" and the replicate weights are "rakedw1" to "rakedw80". The regular expression "rakedw[1-9]" matches the replicate weight variables (and not the sampling weight variable).

data may be a character string giving the name of a table or view in a relational database that can be accessed through the DBI interface. For DBI interfaces $dbtype$ should be the name of the database driver and $dbname$ should be the name by which the driver identifies the specific database (eg file name for SQLite).

The appropriate database interface package must already be loaded (eg RSQLite for SQLite). The survey design object will contain the replicate weights, but actual variables will be loaded from the database only as needed. Use close to close the database connection and open to reopen the connection, eg, after loading a saved object.

The database interface does not attempt to modify the underlying database and so can be used with read-only permissions on the database.

To generate your own replicate weights either use as.svrepdesign on a survey.design object, or see brrweights, bootweights, jk1weights and jknweights

The model.frame method extracts the observed data.
svrepdesign

Value

Object of class svyrep.design, with methods for print, summary, weights, image.

Note

To use replication-weight analyses on a survey specified by sampling design, use as.svrepdesign to convert it.

References


See Also

as.svrepdesign, svydesign, brrweights, bootweights

Examples

data(scd)
# use BRR replicate weights from Levy and Lemeshow
repweights<-2*cbind(c(1,0,1,0,1,0), c(1,0,0,1,0,1), c(0,1,0,0,1,0),
c(0,1,0,1,0,1))
scdrep<-svrepdesign(data=scd, type="BRR", repweights=repweights, combined.weights=FALSE)
svyratio(~alive, ~arrests, scdrep)

## Not run:
## Needs RSQLite
library(RSQLite)
db_rclus1<-svrepdesign(weights=~pw, repweights="wt[1-9]+", type="JK1", scale=(1-15/757)*14/15, data="apiclus1rep", dbtype="SQLite", dbname=system.file("api.db",package="survey"), combined=FALSE)
svymean(~api00+api99,db_rclus1)
summary(db_rclus1)

## closing and re-opening a connection
close(db_rclus1)
db_rclus1
try(svymean(~api00+api99,db_rclus1))
db_rclus1<-open(db_rclus1)
svymean(~api00+api99,db_rclus1)

## End(Not run)
svrVar  
Compute variance from replicates

Description

Compute an appropriately scaled empirical variance estimate from replicates. The mse argument specifies whether the sums of squares should be centered at the point estimate (mse=TRUE) or the mean of the replicates. It is usually taken from the mse component of the design object.

Usage

svrVar(thetas, scale, rscales, na.action=getOption("na.action"), mse=getOption("survey.replicates.mse"), coef)

Arguments

thetas  
matrix whose rows are replicates (or a vector of replicates)

scale  
Overall scaling factor

rscales  
Scaling factor for each squared deviation

na.action  
How to handle replicates where the statistic could not be estimated

mse  
if TRUE, center at the point estimated, if FALSE center at the mean of the replicates

coef  
The point estimate, required only if mse==TRUE

Value

covariance matrix.

See Also

svrepdesign, as.svrepdesign, brrweights, jklweights, jknweights

svy.varcoef  
Sandwich variance estimator for glms

Description

Computes the sandwich variance estimator for a generalised linear model fitted to data from a complex sample survey. Designed to be used internally by svyglm.

Usage

svy.varcoef(glm.object, design)
svyby

Arguments

- glm.object: A glm object
- design: A survey.design object

Value

A variance matrix

Author(s)

Thomas Lumley

See Also

svyglm, svydesign, svyCprod

svyby

Survey statistics on subsets

Description

Compute survey statistics on subsets of a survey defined by factors.

Usage

svyby(formula, by, design,...)

## Default S3 method:
svyby(formula, by, design, FUN, ..., deff=FALSE,keep.var = TRUE, keep.names = TRUE,verbose=FALSE, vartype=c("se","ci","ci","cv","cvpct","var"), drop.empty.groups=TRUE, covmat=FALSE, return.replicates=FALSE, na.rm.by=FALSE, na.rm.all=FALSE, multicore=getOption("survey.multicore"))

## S3 method for class 'svyby'
SE(object,...)

## S3 method for class 'svyby'
deff(object,...)

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

## S3 method for class 'svyby'
confint(object, parm, level = 0.95, df =Inf,...)
unwtd.count(x, design, ...)
Arguments

formula, x
A formula specifying the variables to pass to FUN (or a matrix, data frame, or vector)

by
A formula specifying factors that define subsets, or a list of factors.

design
A `svydesign` or `svrepdesign` object

FUN
A function taking a formula and survey design object as its first two arguments.

... Other arguments to FUN

deff
Request a design effect from FUN

keep.var
If FUN returns a `svystat` object, extract standard errors from it

keep.names
Define row names based on the subsets

verbose
If TRUE, print a label for each subset as it is processed.

vartype
Report variability as one or more of standard error, confidence interval, coefficient of variation, percent coefficient of variation, or variance

drop.empty.groups
If FALSE, report NA for empty groups, if TRUE drop them from the output

na.rm.by
If true, omit groups defined by NA values of the by variables.

na.rm.all
If true, check for groups with no non-missing observations for variables defined by formula and treat these groups as empty

covmat
If TRUE, compute covariances between estimates for different subsets (currently only for replicate-weight designs). Allows `svycontrast` to be used on output.

return.replicates
Only for replicate-weight designs. If TRUE, return all the replicates as the "replicates" attribute of the result

multicore
Use multicore package to distribute subsets over multiple processors?

parm
A specification of which parameters are to be given confidence intervals, either a vector of numbers or a vector of names. If missing, all parameters are considered.

level
The confidence level required.

df
Degrees of freedom for t-distribution in confidence interval, use `degf(design)` for number of PSUs minus number of strata.

object
An object of class "svyby"

Details

The variance type "ci" asks for confidence intervals, which are produced by `confint`. In some cases additional options to FUN will be needed to produce confidence intervals, for example, `svyquantile` needs `ci=TRUE` or `keep.var=FALSE`.

`unwtd.count` is designed to be passed to `svyby` to report the number of non-missing observations in each subset. Observations with exactly zero weight will also be counted as missing, since that’s how subsets are implemented for some designs.

Parallel processing with `multicore=TRUE` is useful only for fairly large problems and on computers with sufficient memory. The `multicore` package is incompatible with some GUIs, although the Mac Aqua GUI appears to be safe.
Value

An object of class "svyby": a data frame showing the factors and the results of FUN. For unwtd.count, the unweighted number of non-missing observations in the data matrix specified by x for the design.

Note

The function works by making a lot of calls of the form FUN(formula,subset(design,by==i)), where formula is re-evaluated in each subset, so it is unwise to use data-dependent terms in formula. In particular, svyby(~factor(a),~b,design=d,svymean), will create factor variables whose levels are only those values of a present in each subset. Use update.survey.design to add variables to the design object instead.

Note

Asking for a design effect (deff=TRUE) from a function that does not produce one will cause an error or incorrect formatting of the output. The same will occur with keep.var=TRUE if the function does not compute a standard error.

See Also

svytable and ftable.svystat for contingency tables, ftable.svyby for pretty-printing of svyby

Examples

data(api)
dclus1<-svydesign(id=~dnum, weights=~pw, data=apiclus1, fpc=~fpc)
svyby(~api99, ~stype, dclus1, svymean)
svyby(~api99, ~stype, dclus1, svyquantile, quantiles=0.5, ci=TRUE, vartype="ci")
## without ci=TRUE svyquantile does not compute standard errors
svyby(~api99, ~stype, dclus1, svyquantile, quantiles=0.5, keep.var=FALSE)
svyby(~api99, list(school.type=apiclus1$stype), dclus1, svymean)
svyby(~api99+api00, ~stype, dclus1, svymean, deff=TRUE, vartype="ci")
svyby(~api99+api00, ~stype+sch.wide, dclus1, svymean, keep.var=FALSE)
## report raw number of observations
svyby(~api99+api00, ~stype+sch.wide, dclus1, unwtd.count, keep.var=FALSE)

rclus1<-as.svrepdesign(dclus1)
svyby(~api99, ~stype, rclus1, svymean)
svyby(~api99, ~stype, rclus1, svyquantile, quantiles=0.5)
svyby(~api99, list(school.type=apiclus1$stype), rclus1, svymean, vartype="cv")
svyby(~enroll, ~stype, rclus1, svytotal, deff=TRUE)
svyby(~api99+api00, ~stype+sch.wide, rclus1, svymean, keep.var=FALSE)
## report raw number of observations
svyby(~api99+api00, ~stype+sch.wide, rclus1, unwtd.count, keep.var=FALSE)
## comparing subgroups using covmat=TRUE
mns<-svyby(~api99, ~stype, rclus1, svymean,covmat=TRUE)
vcov(mns)
svycontrast(mns, c(E = 1, M = -1))

str(svyby(~api99, ~stype, rclus1, svymean, return.replicates=TRUE))

## extractor functions
(a<-svyby(~enroll, ~stype, rclus1, svytotal, deff=TRUE, verbose=TRUE, vartype=c("se","cv","cvpct","var")))
deff(a)
SE(a)
cv(a)
coef(a)
confint(a, df=degf(rclus1))

## ratio estimates
svyby(~api.stu, by=~stype, denominator=~enroll, design=dclus1, svyratio)

## empty groups
svyby(~api00,~comp.imp+sch.wide,design=dclus1,svymean)
svyby(~api00,~comp.imp+sch.wide,design=dclus1,svymean,drop.empty.groups=FALSE)

---

svycdf

Cumulative Distribution Function

Description

Estimates the population cumulative distribution function for specified variables. In contrast to svyquantile, this does not do any interpolation: the result is a right-continuous step function.

Usage

svycdf(formula, design, na.rm = TRUE,...)

## S3 method for class 'svycdf'

print(x,...)

## S3 method for class 'svycdf'

plot(x,xlab=NULL,...)

Arguments

formula one-sided formula giving variables from the design object
design survey design object
na.rm remove missing data (case-wise deletion)?
... other arguments to plot.stepfun
x object of class svycdf
xlab a vector of x-axis labels or NULL for the default labels
svyciprop

Value

An object of class svycdf, which is a list of step functions (of class stepfun)

See Also

svyquantile, svyhist, plot.stepfun

Examples

data(api)
dstrat <- svydesign(id = ~1, strata = ~stype, weights = ~pw, data = apistrat,
                   fpc = ~fpc)
cdf.est<-svycdf(~enroll+api00+api99, dstrat)
cdf.est
## function
cdf.est[[1]]
## evaluate the function
cdf.est[[1]](800)
cdf.est[[2]](800)

## compare to population and sample CDFs.
opar<-par(mfrow=c(2,1))
cdf.pop<-ecdf(apiipop$enroll)
cdf.samp<-ecdf(apistrat$enroll)
plot(cdf.pop,main="Population vs sample", xlab="Enrollment")
lines(cdf.samp,col.points="red")

plot(cdf.pop, main="Population vs estimate", xlab="Enrollment")
lines(cdf.est[[1]],col.points="red")
par(opar)

svyciprop

Confidence intervals for proportions

Description

Computes confidence intervals for proportions using methods that may be more accurate near 0 and 1 than simply using confint(svymean()).

Usage

svyciprop(formula, design, method = c("logit", "likelihood", "asin", "beta", "mean","xlogit"), level = 0.95, df=degf(design),...)
Arguments

- **formula**: Model formula specifying a single binary variable
- **design**: Survey design object
- **method**: See Details below. Partial matching is done on the argument.
- **level**: Confidence level for interval
- **df**: Denominator degrees of freedom, for all methods except "beta". Use Inf for confidence intervals based on a Normal distribution, and for "likelihood" and "logit" use NULL for the default method in glms (currently degf(design)-1, but this may be improved in the future)

... For "mean" and "asin", this is passed to `confint.svystat`

Details

The "logit" method fits a logistic regression model and computes a Wald-type interval on the log-odds scale, which is then transformed to the probability scale.

The "likelihood" method uses the (Rao-Scott) scaled chi-squared distribution for the loglikelihood from a binomial distribution.

The "asin" method uses the variance-stabilising transformation for the binomial distribution, the arcsine square root, and then back-transforms the interval to the probability scale.

The "beta" method uses the incomplete beta function as in `binom.test`, with an effective sample size based on the estimated variance of the proportion. (Korn and Graubard, 1998)

The "xlogit" method uses a logit transformation of the mean and then back-transforms to the probability scale. This appears to be the method used by SUDAAN and SPSS COMPLEX SAMPLES.

The "mean" method is a Wald-type interval on the probability scale, the same as `confint(svymean())`

All methods undercover for probabilities close enough to zero or one, but "beta", "likelihood", "logit", and "logit" are noticeably better than the other two. None of the methods will work when the observed proportion is exactly 0 or 1.

The `confint` method extracts the confidence interval; the `vcov` and `SE` methods just report the variance or standard error of the mean.

Value

The point estimate of the proportion, with the confidence interval as an attribute

References


See Also

- `svymean`, `yrbs`
Examples

```r
data(api)
dclus1 <- svydesign(id = ~dnum, fpc = ~fpc, data = apiclus1)

svyciprop(~I(ell==0), dclus1, method = "li")
svyciprop(~I(ell==0), dclus1, method = "lo")
svyciprop(~I(ell==0), dclus1, method = "as")
svyciprop(~I(ell==0), dclus1, method = "be")
svyciprop(~I(ell==0), dclus1, method = "me")
svyciprop(~I(ell==0), dclus1, method = "xl")

## reproduces Stata svy: mean
svyciprop(~I(ell==0), dclus1, method = "me", df = degf(dclus1))
## reproduces Stata svy: prop
svyciprop(~I(ell==0), dclus1, method = "lo", df = degf(dclus1))
```

```r
rclus1 <- as.svrepdesign(dclus1)
svyciprop(~I(emer==0), rclus1, method = "li")
svyciprop(~I(emer==0), rclus1, method = "lo")
svyciprop(~I(emer==0), rclus1, method = "as")
svyciprop(~I(emer==0), rclus1, method = "be")
svyciprop(~I(emer==0), rclus1, method = "me")
```

---

svycontrast

**Linear and nonlinear contrasts of survey statistics**

Description

Computes linear or nonlinear contrasts of estimates produced by survey functions (or any object with coef and vcov methods).

Usage

```r
svycontrast(stat, contrasts, ...)
```

Arguments

- **stat**: object of class svrepstat or svystat
- **contrasts**: A vector or list of vectors of coefficients, or a call or list of calls
- **...**: For future expansion
Details

If contrasts is a list, the element names are used as names for the returned statistics.

If an element of contrasts is shorter than coef(stat) and has names, the names are used to match up the vectors and the remaining elements of contrasts are assumed to be zero. If the names are not legal variable names (eg `0.1`) they must be quoted (eg "0.1")

If contrasts is a "call" or list of "call"s, the delta-method is used to compute variances, and the calls must use only functions that deriv knows how to differentiate. If the names are not legal variable names they must be quoted with backticks (eg `0.1`).

Value

Object of class svrepstat or svystat

See Also

regTermTest, svyglm

Examples

data(api)
dclus1<-svydesign(id=~dnum, weights=~pw, data=apiclus1, fpc=~fpc)
a <- svytotal(~api00+enroll+api99, dclus1)
svycontrast(a, list(avg=c(0.5,0,0.5), diff=c(1,0,-1)))
## if contrast vectors have names, zeroes may be omitted
svycontrast(a, list(avg=c(api00=0.5,api99=0.5), diff=c(api00=1,api99=-1)))

## nonlinear contrasts
svycontrast(a, quote(api00/api99))
svyratio(~api00, ~api99, dclus1)

## Example: standardised skewness coefficient
moments<-svymean(~I(api00^3)+I(api00^2)+I(api00), dclus1)
svycontrast(moments,
quote((`I(api00^3)`-3*`I(api00^2)`*`I(api00)`+ 3*`I(api00)`*`I(api00)`^2-`I(api00)`^3)/
  (`I(api00^2)`-`I(api00)`^2)^1.5))

## Example: geometric means
meanlogs <- svymean(~log(api00)+log(api99), dclus1)
svycontrast(meanlogs,
  list(api00=quote(exp(`log(api00)`)), api99=quote(exp(`log(api99)`))))
svycoplot

Conditioning plots of survey data

Description

Draws conditioned scatterplots ('Trellis' plots) of survey data using hexagonal binning or transparency.

Usage

svycoplot(formula, design, style = c("hexbin", "transparent"), basecol = "black", alpha = c(0, 0.8), hexscale = c("relative", "absolute"), ...)  

Arguments

- formula: A graph formula suitable for xyplot
- design: A survey design object
- style: Hexagonal binning or transparent color?
- basecol: The fully opaque 'base' color for creating transparent colors. This may also be a function; see svyplot for details
- alpha: Minimum and maximum opacity
- hexscale: Scale hexagons separate for each panel (relative) or across all panels (absolute)
- ...: Other arguments passed to grid.hexagons or xyplot

Value

An object of class trellis

Note

As with all 'Trellis' graphs, this function creates an object but does not draw the graph. When used inside a function or non-interactively you need to print() the result to create the graph.

See Also

svyplot

Examples

data(api)
dclus2<-svydesign(id=~dnum+snum, weights=~pw,
                 data=apiclus2, fpc=~fpc1+fpc2)

svycoplot(api00~api99|sch.wide*comp.imp, design=dclus2, style="hexbin")
svycoplot(api00~api99|sch.wide*comp.imp, design=dclus2, style="hexbin", hexscale="absolute")
svycoplot(api00~api99|sch.wide, design=dclus2, style="trans")
svycoxph

svycoxph

Survey-weighted Cox models.

Description

Fit a proportional hazards model to data from a complex survey design.

Usage

svycoxph(formula, design, subset=NULL, rescale=TRUE, ...)

## S3 method for class 'svycoxph'
predict(object, newdata, se=FALSE,
   type=c("lp", "risk", "expected", "terms","curve"),...)

## S3 method for class 'svycoxph'
AIC(object, ..., k = 2)

Arguments

formula
  Model formula. Any cluster() terms will be ignored.
design
  survey.design object. Must contain all variables in the formula
subset
  Expression to select a subpopulation
rescale
  Rescale weights to improve numerical stability
object
  A svycoxph object
newdata
  New data for prediction
se
  Compute standard errors? This takes a lot of memory for type="curve"
type
  "curve" does predicted survival curves. The other values are passed to predict.coxph()
...  
  For AIC, more models to compare the AIC of. For svycoxph, other arguments passed to coxph.
k
  The penalty per parameter that would be used under independent sampling: AIC has k=2

Details

The main difference between svycoxph function and the robust=TRUE option to coxph in the survival package is that this function accounts for the reduction in variance from stratified sampling and the increase in variance from having only a small number of clusters.

Note that strata terms in the model formula describe subsets that have a separate baseline hazard function and need not have anything to do with the stratification of the sampling.
The AIC method uses the same approach as \texttt{AIC.svyglm}, though the relevance of the criterion this optimises is a bit less clear than for generalised linear models.

The standard errors for predicted survival curves are available only by linearization, not by replicate weights (at the moment). Use \texttt{withReplicates} to get standard errors with replicate weights. Predicted survival curves are not available for stratified Cox models.

The standard errors use the delta-method approach of Williams (1995) for the Nelson-Aalen estimator, modified to handle the Cox model following Tsiatis (1981). The standard errors agree closely with \texttt{survfit.coxph} for independent sampling when the model fits well, but are larger when the model fits poorly. I believe the standard errors are equivalent to those of Lin (2000), but I don’t know of any implementation that would allow a check.

\section*{Value}

An object of class \texttt{svycoxph} for \texttt{svycoxph}, an object of class \texttt{svykm} or \texttt{svykmlist} for \texttt{predict(,type="curve")}.

\section*{Warning}

The standard error calculation for survival curves uses memory proportional to the sample size times the square of the number of events.

\section*{Author(s)}

Thomas Lumley

\section*{References}

Binder DA. (1992) Fitting Cox’s proportional hazards models from survey data. Biometrika 79: 139-147


\section*{See Also}

\texttt{coxph, predict.coxph}

\texttt{svykm} for estimation of Kaplan-Meier survival curves and for methods that operate on survival curves.

\texttt{regTermTest} for Wald and (Rao-Scott) likelihood ratio tests for one or more parameters.

\section*{Examples}

```r
## Somewhat unrealistic example of nonresponse bias.
data(pbc, package="survival")
pbc$randomized<-with(pbc, !is.na(trt) & trt>0)
biasmodel<-glm(randomized~age*edema, data=pbc, family=binomial)
```
svyCprod <- fitted(biasmodel)
if (is.null(pbc$albumin)) pbc$albumin <- pbc$alb  # pre 2.9.0

dpbc <- svydesign(id=~1, prob=~randprob, strata=~edema, data=subset(pbc, randomized))
rpbc <- as.svrepdesign(dpbc)

(model <- svycoxph(Surv(time,status>0)~log(bili)+protime+albumin, design=dpbc))

svycoxph(Surv(time,status>0)~log(bili)+protime+albumin, design=rpbc)

s <- predict(model, se=TRUE, type="curve",
            newdata=data.frame(bili=c(3,9), protime=c(10,10), albumin=c(3.5,3.5)))
plot(s[[1]], ci=TRUE, col="sienna")
lines(s[[2]], ci=TRUE, col="royalblue")
quantile(s[[1]], ci=TRUE)
confint(s[[2]], parm=365*(1:5))

svyCprod

Computations for survey variances

Description

Computes the sum of products needed for the variance of survey sample estimators. svyCprod is used for survey design objects from before version 2.9, onestage is called by svyrecvar for post-2.9 design objects.

Usage

svyCprod(x, strata, psu, fpc, nPSU, certainty=NULL, postStrata=NULL,
lonely.psu=getOption("survey.lonely.psu"))
onestage(x, strata, clusters, nPSU, fpc,
        lonely.psu=getOption("survey.lonely.psu"), stage=0, cal)

Arguments

x A vector or matrix
strata A vector of stratum indicators (may be NULL for svyCprod)
psu A vector of cluster indicators (may be NULL)
clusters A vector of cluster indicators
fpc A data frame (svyCprod) or vector (onestage) of population stratum sizes, or NULL
nPSU Table (svyprod) or vector (onestage) of original sample stratum sizes (or NULL)
certainty logical vector with stratum names as names. If TRUE and that stratum has a single PSU it is a certainty PSU
postStrata Post-stratification variables
lonely.psu  One of "remove", "adjust", "fail", "certainty", "average". See Details below

stage  Used internally to track the depth of recursion

cal  Used to pass calibration information at stages below the population

Details

The observations for each cluster are added, then centered within each stratum and the outer product is taken of the row vector resulting for each cluster. This is added within strata, multiplied by a degrees-of-freedom correction and by a finite population correction (if supplied) and added across strata.

If there are fewer clusters (PSUs) in a stratum than in the original design extra rows of zeroes are added to x to allow the correct subpopulation variance to be computed.

See `postStratify` for information about post-stratification adjustments.

The variance formula gives 0/0 if a stratum contains only one sampling unit. If the certainty argument specifies that this is a PSU sampled with probability 1 (a "certainty" PSU) then it does not contribute to the variance (this is correct only when there is no subsampling within the PSU – otherwise it should be defined as a pseudo-stratum). If certainty is `FALSE` for this stratum or is not supplied the result depends on lonely.psu.

The options are "fail" to give an error, "remove" or "certainty" to give a variance contribution of 0 for the stratum, "adjust" to center the stratum at the grand mean rather than the stratum mean, and "average" to assign strata with one PSU the average variance contribution from strata with more than one PSU. The choice is controlled by setting `options(survey.lonely.psu)`. If this is not done the factory default is "fail". Using "adjust" is conservative, and it would often be better to combine strata in some intelligent way. The properties of "average" have not been investigated thoroughly, but it may be useful when the lonely PSUs are due to a few strata having PSUs missing completely at random.

The "remove"and "certainty" options give the same result, but "certainty" is intended for situations where there is only one PSU in the population stratum, which is sampled with certainty (also called 'self-representing' PSUs or strata). With "certainty" no warning is generated for strata with only one PSU. Ordinarily, `svydesign` will detect certainty PSUs, making this option unnecessary.

For strata with a single PSU in a subset (domain) the variance formula gives a value that is well-defined and positive, but not typically correct. If `options("survey.adjust.domain.lonely")` is `TRUE` and `options("survey.lonely.psu")` is "adjust" or "average", and no post-stratification or G-calibration has been done, strata with a single PSU in a subset will be treated like those with a single PSU in the sample. I am not aware of any theoretical study of this procedure, but it should at least be conservative.

Value

A covariance matrix

Author(s)

Thomas Lumley
svycralpha

Cronbach’s alpha

Description

Compute Cronbach’s alpha coefficient of reliability from survey data. The formula is equation (2) of Cronbach (1951) only with design-based estimates of the variances.

Usage

svycralpha(formula, design, na.rm = FALSE)

Arguments

formula One-sided formula giving the variables that make up the total score
design survey design object
na.rm TRUE to remove missing values

Value

A number

References


Examples

data(api)
dstrat<-svydesign(id = ~1, strata = ~stype, weights = ~pw, data = apistrat, fpc = ~fpc)
svycralpha(~ell+mobility+avg.ed+emer+meals, dstrat)
svydesign

Survey sample analysis.

Description

Specify a complex survey design.

Usage

svydesign(ids, probs=NULL, strata = NULL, variables = NULL, fpc=NULL, 
data = NULL, nest = FALSE, check.strata = !nest, weights=NULL, pps=FALSE,...)

## Default S3 method:
svydesign(ids, probs=NULL, strata = NULL, variables = NULL, 
fpc=NULL, data = NULL, nest = FALSE, check.strata = !nest, weights=NULL, 
pps=FALSE, variance=c("HT","YG"),...)

## S3 method for class 'imputationList'
svydesign(ids, probs = NULL, strata = NULL, variables = NULL, 
fpc = NULL, data, nest = FALSE, check.strata = !nest, weights = NULL, pps=FALSE, 
...)

## S3 method for class 'character'
svydesign(ids, probs = NULL, strata = NULL, variables = NULL, 
fpc = NULL, data, nest = FALSE, check.strata = !nest, weights = NULL, pps=FALSE, 
dbtype = "SQLite", dbname, ...)

Arguments

ids          Formula or data frame specifying cluster ids from largest level to smallest level, 
~0 or ~1 is a formula for no clusters.
probs        Formula or data frame specifying cluster sampling probabilities
strata       Formula or vector specifying strata, use NULL for no strata
variables    Formula or data frame specifying the variables measured in the survey. If NULL, 
the data argument is used.
fpc          Finite population correction: see Details below
weights      Formula or vector specifying sampling weights as an alternative to prob
data          Data frame to look up variables in the formula arguments, or database table 
name, or imputationList object, see below
nest          If TRUE, relabel cluster ids to enforce nesting within strata
check.strata If TRUE, check that clusters are nested in strata.
pps          "brewer" to use Brewer’s approximation for PPS sampling without replace-
ment. "overton" to use Overton’s approximation. An object of class HR to use 
the Hartley-Rao approximation. An object of class ppsmat to use the Horvitz-
Thompson estimator.
dbtype       name of database driver to pass to dbDriver
dbname       name of database (eg file name for SQLite)
svydesign

variance For pps without replacement, use variance="YG" for the Yates-Grundy estimator instead of the Horvitz-Thompson estimator

... for future expansion

Details

The svydesign object combines a data frame and all the survey design information needed to analyse it. These objects are used by the survey modelling and summary functions. The id argument is always required, the strata, fpc, weights and probs arguments are optional. If these variables are specified they must not have any missing values.

By default, svydesign assumes that all PSUs, even those in different strata, have a unique value of the id variable. This allows some data errors to be detected. If your PSUs reuse the same identifiers across strata then set nest=TRUE.

The finite population correction (fpc) is used to reduce the variance when a substantial fraction of the total population of interest has been sampled. It may not be appropriate if the target of inference is the process generating the data rather than the statistics of a particular finite population.

The finite population correction can be specified either as the total population size in each stratum or as the fraction of the total population that has been sampled. In either case the relevant population size is the sampling units. That is, sampling 100 units from a population stratum of size 500 can be specified as 500 or as 100/500=0.2. The exception is for PPS sampling without replacement, where the sampling probability (which will be different for each PSU) must be used.

If population sizes are specified but not sampling probabilities or weights, the sampling probabilities will be computed from the population sizes assuming simple random sampling within strata.

For multistage sampling the id argument should specify a formula with the cluster identifiers at each stage. If subsequent stages are stratified strata should also be specified as a formula with stratum identifiers at each stage. The population size for each level of sampling should also be specified in fpc. If fpc is not specified then sampling is assumed to be with replacement at the top level and only the first stage of cluster is used in computing variances. If fpc is specified but for fewer stages than id, sampling is assumed to be complete for subsequent stages. The variance calculations for multistage sampling assume simple or stratified random sampling within clusters at each stage except possibly the last.

For PPS sampling without replacement it is necessary to specify the probabilities for each stage of sampling using the fpc arguments, and an overall weight argument should not be given. At the moment, multistage or stratified PPS sampling without replacement is supported only with pps="brewer", or by giving the full joint probability matrix using ppsmat. [Cluster sampling is supported by all methods, but not subsampling within clusters].

The dim, ",", "<-" and na.action methods for survey.design objects operate on the dataframe specified by variables and ensure that the design information is properly updated to correspond to the new data frame. With the "<-" method the new value can be a survey.design object instead of a data frame, but only the data frame is used. See also subset.survey.design for a simple way to select subpopulations.

The model.frame method extracts the observed data.

If the strata with only one PSU are not self-representing (or they are, but svydesign cannot tell based on fpc) then the handling of these strata for variance computation is determined by options("survey.lonely.psu"). See svyCprod for details.
data may be a character string giving the name of a table or view in a relational database that can be accessed through the DBI interfaces. For DBI interfaces dbtype should be the name of the database driver and dbname should be the name by which the driver identifies the specific database (e.g., file name for SQLite).

The appropriate database interface package must already be loaded (e.g., RSQLite for SQLite). The survey design object will contain only the design meta-data, and actual variables will be loaded from the database as needed. Use close to close the database connection and open to reopen the connection, e.g., after loading a saved object.

The database interface does not attempt to modify the underlying database and so can be used with read-only permissions on the database.

If data is an imputationList object (from the "mitools" package), svydesign will return a svyimputationList object containing a set of designs. Use with.svyimputationList to do analyses on these designs and MIcombine to combine the results.

Value
An object of class survey.design.

Author(s)
Thomas Lumley

See Also
as.svrepdesign for converting to replicate weight designs, subset.survey.design for domain estimates, update.survey.design to add variables.
mtools package for using multiple imputations
svyrecvar and svyCprod for details of variance estimation
election for examples of PPS sampling without replacement.
http://faculty.washington.edu/tlumley/survey/ for examples of database-backed objects.

Examples

data(api)
# stratified sample
dstrat<-svydesign(id=~1, strata=~stype, weights=~pw, data=apistrat, fpc=~fpc)
# one-stage cluster sample
dclus1<-svydesign(id=~dnum, weights=~pw, data=apiclus1, fpc=~fpc)
# two-stage cluster sample: weights computed from population sizes.
dclus2<-svydesign(id=~dnum+snum, fpc=~fpc1+fpc2, data=apiclus2)

## multistage sampling has no effect when fpc is not given, so
## these are equivalent.
dclus2wr<-svydesign(id=~dnum+snum, weights=weights(dclus2), data=apiclus2)
dclus2wr2<-svydesign(id=~dnum, weights=weights(dclus2), data=apiclus2)

## syntax for stratified cluster sample
##(though the data weren't really sampled this way)
svyfactanal(id=~dnum, strata=~stype, weights=~pw, data=apistrat, nest=TRUE)

## PPS sampling without replacement
data(election)
dpps<- svydesign(id=~1, fpc=~p, data=election_pps, pps="brewer")

## database example: requires RSQLite
## Not run:
library(RSQLite)
dclus1<-svydesign(id=~dnum, weights=~pw, fpc=~fpc, data="apiclus1", dbtype="SQLite", dbname=system.file("api.db",package="survey"))

## End(Not run)

svyfactanal |
--- |
**Factor analysis in complex surveys (experimental).**

Description

This function fits a factor analysis model or SEM, by maximum weighted likelihood.

Usage

```r
svyfactanal(formula, design, factors,
            n = c("none", "sample", "degf","effective", "min.effective"), ...)
```

Arguments

- `formula` - Model formula specifying the variables to use
- `design` - Survey design object
- `factors` - Number of factors to estimate
- `n` - Sample size to be used for testing: see below
- `...` - Other arguments to pass to `factanal`.

Details

The population covariance matrix is estimated by `svyvar` and passed to `factanal`.

Although fitting these models requires only the estimated covariance matrix, inference requires a sample size. With `n="sample"`, the sample size is taken to be the number of observations; with `n="degf"`, the survey degrees of freedom as returned by `degf`. Using "sample" corresponds to standardizing weights to have mean 1, and is known to result in anti-conservative tests.

The other two methods estimate an effective sample size for each variable as the sample size where the standard error of a variance of a Normal distribution would match the design-based standard error estimated by `svyvar`. With `n="min.effective"` the minimum sample size across the variables is used; with `n="effective"` the harmonic mean is used. For `svyfactanal` the test of model adequacy is optional, and the default choice, `n="none"`, does not do the test.
svyglm

Survey-weighted generalised linear models.

Description

Fit a generalised linear model to data from a complex survey design, with inverse-probability weighting and design-based standard errors.

Usage

## S3 method for class 'survey.design'
svyglm(formula, design, subset=NULL,
family=stats::gaussian(),start=NULL, rescale=TRUE, ...)

## S3 method for class 'svyrep.design'
svyglm(formula, design, subset=NULL,
family=stats::gaussian(),start=NULL, rescale=NULL, ..., rho=NULL,
return.replicates=FALSE, na.action,multicore=getOption("survey.multicore"))

## S3 method for class 'svyglm'
summary(object, correlation = FALSE, df.resid=NULL,
...)

## S3 method for class 'svyglm'

Value

An object of class factanal

References

.

See Also

factanal

The lavaan.survey package fits structural equation models to complex samples using similar techniques.

Examples

data(api)
dclus1<-svydesign(id=~dnum, weights=~pw, data=apiclus1, fpc=~fpc)

svyfactanal(~api99+api00+hsg+meals+ell+emer, design=dclus1, factors=2)

svyfactanal(~api99+api00+hsg+meals+ell+emer, design=dclus1, factors=2, n="effective")

###Population dat for comparison
factanal(~api99+api00+hsg+meals+ell+emer, data=apipop, factors=2)
predict(object,newdata=NULL,total=NULL,  
  type=c("link","response","terms"),  
  se.fit=(type != "terms"),vcov=FALSE,...)

## S3 method for class 'svrepglm'
predict(object,newdata=NULL,total=NULL,  
  type=c("link","response","terms"),  
  se.fit=(type != "terms"),vcov=FALSE,  
  return.replicates=!is.null(object$replicates),...)

### Arguments

- **formula**: Model formula
- **design**: Survey design from `svydesign` or `svrepdesign`. Must contain all variables in the formula
- **subset**: Expression to select a subpopulation
- **family**: family object for `glm`
- **start**: Starting values for the coefficients (needed for some uncommon link/family combinations)
- **rescale**: Rescaling of weights, to improve numerical stability. The default rescales weights to sum to the sample size. Use FALSE to not rescale weights. For replicate-weight designs, use TRUE to rescale weights to sum to 1, as was the case before version 3.34.
- **rho**: For replicate BRR designs, to specify the parameter for Fay's variance method, giving weights of rho and 2-rho
- **return.replicates**: Return the replicates as the replicates component of the result? (for predict, only possible if they were computed in the `svyglm` fit)
- **object**: A `svyglm` object
- **correlation**: Include the correlation matrix of parameters?
- **na.action**: Handling of NAs
- **multicore**: Use the `multicore` package to distribute replicates across processors?
- **df.resid**: Optional denominator degrees of freedom for Wald tests
- **newdata**: new data frame for prediction
- **total**: population size when predicting population total
- **type**: linear predictor (link) or response
- **se.fit**: if TRUE, return variances of predictions
- **vcov**: if TRUE and se=TRUE return full variance-covariance matrix of predictions
svyglm

Details

For binomial and Poisson families use family=quasibinomial() and family=quasipoisson() to avoid a warning about non-integer numbers of successes. The ‘quasi’ versions of the family objects give the same point estimates and standard errors and do not give the warning.

If df.resid is not specified the df for the null model is computed by degf and the residual df computed by subtraction. This is recommended by Korn and Graubard and is correct for PSU-level covariates but is potentially very conservative for individual-level covariates. To get tests based on a Normal distribution use df.resid=Inf, and to use number of PSUs-number of strata, specify df.resid=degf(design).

Parallel processing with multicore=TRUE is helpful only for fairly large data sets and on computers with sufficient memory. It may be incompatible with GUIs, although the Mac Aqua GUI appears to be safe.

predict gives fitted values and sampling variability for specific new values of covariates. When newdata are the population mean it gives the regression estimator of the mean, and when newdata are the population totals and total is specified it gives the regression estimator of the population total. Regression estimators of mean and total can also be obtained with calibrate.

Value

svyglm returns an object of class svyglm. The predict method returns an object of class svystat.

Author(s)

Thomas Lumley

References


See Also

glm, which is used to do most of the work.
regTermTest, for multiparameter tests
calibrate, for an alternative way to specify regression estimators of population totals or means
svytest for one-sample and two-sample t-tests.

Examples

data(api)

dstrat<-svydesign(id=~1,strata=~stype, weights=~pw, data=apistrat, fpc=~fpc)
dclus2<-svydesign(id=~dnum+snum, weights=~pw, data=apiclus2)
rstrat<-as.svrepdesign(dstrat)
rclus2<-as.svrepdesign(dclus2)
summary(svyglm(api00~ell+meals+mobility, design=dstrat))
summary(svyglm(api00~ell+meals+mobility, design=clus2))
summary(svyglm(api00~ell+meals+mobility, design=rstrat))
summary(svyglm(api00~ell+meals+mobility, design=rclus2))

## use quasibinomial, quasipoisson to avoid warning messages
summary(svyglm(sch.wide~ell+meals+mobility, design=dstrat,
family=quasibinomial()))

## Compare regression and ratio estimation of totals
api.ratio <- svyratio(~api.stu,~enroll, design=dstrat)
pop<-data.frame(enroll=sum(apiipop$enroll, na.rm=TRUE))
npop <- nrow(apiipop)
predict(api.ratio, pop$enroll)

## regression estimator is less efficient
api.reg <- svyglm(api.stu~enroll, design=dstrat)
predict(api.reg, newdata=pop, total=npop)

## same as calibration estimator
svytotal(~api.stu, calibrate(dstrat, ~enroll, pop=c(npop, pop$enroll)))

## svyglm can also reproduce the ratio estimator
api.reg2 <- svyglm(api.stu~enroll-1, design=dstrat, 
family=quasi(link="identity",var="mu"))
predict(api.reg2, newdata=pop, total=npop)

## higher efficiency by modelling variance better
api.reg3 <- svyglm(api.stu~enroll-1, design=dstrat, 
family=quasi(link="identity",var="mu^3"))
predict(api.reg3, newdata=pop, total=npop)

## true value
sum(apiipop$api.stu)

svyhist

svyhist(formula, design, breaks = "Sturges",
include.lowest = TRUE, right = TRUE, xlab = NULL,
main = NULL, probability = TRUE, freq = !probability, ...)
svyboxplot(formula, design, all.outliers=FALSE,...)
svyhist

Arguments

- formula: One-sided formula for svyhist, two-sided for svyboxplot
- design: A survey design object
- xlab: x-axis label
- main: Main title
- probability, freq: Y-axis is probability density or frequency
- all.outliers: Show all outliers in the boxplot, not just extremes
- breaks, include.lowest, right: As for hist...
- ...: Other arguments to hist or bxp

Details

The histogram breakpoints are computed as if the sample were a simple random sample of the same size.

The grouping variable in svyboxplot, if present, must be a factor.

The boxplot whiskers go to the maximum and minimum observations or to 1.5 interquartile ranges beyond the end of the box, whichever is closer. The maximum and minimum are plotted as outliers if they are beyond the ends of the whiskers, but other outlying points are not plotted unless all.outliers=TRUE. svyboxplot requires a two-sided formula; use variable~1 for a single boxplot.

Value

As for hist, except that when probability=FALSE, the return value includes a component count_scale giving a scale factor between density and counts, assuming equal bin widths.

See Also

svyplot

Examples

data(api)
dstrat <- svydesign(id = ~1, strata = ~stype, weights = ~pw, data = apistrat, fpc = ~fpc)
opar<-par(mfrow=c(1,3))
svyhist(~enroll, dstrat, main="Survey weighted",col="purple",ylim=c(0,1.3e-3))
hist(apistrat$enroll, main="Sample unweighted",col="purple",prob=TRUE,ylim=c(0,1.3e-3))
hist(apipop$enroll, main="Population",col="purple",prob=TRUE,ylim=c(0,1.3e-3))
par(mfrow=c(1,1))
svyboxplot(enroll~stype,dstrat,all.outliers=TRUE)
svyboxplot(enroll~1,dstrat)
par(opar)
svyivreg  Two-stage least-squares for instrumental variable regression

Description

Estimates regressions with endogenous covariates using two-stage least squares. The function uses ivreg from the AER package for the main computations, and follows the syntax of that function.

Usage

svyivreg(formula, design, ...)

Arguments

formula  formula specification(s) of the regression relationship and the instruments. See Details for details

Design  A survey design object

...  For future expansion

Details

Regressors and instruments for svyivreg are specified in a formula with two parts on the right-hand side, e.g., \( y \sim x1 + x2 \mid z1 + z2 + z3 \), where \( x1 \) and \( x2 \) are the regressors and \( z1, z2, \) and \( z3 \) are the instruments. Note that exogenous regressors have to be included as instruments for themselves. For example, if there is one exogenous regressor \( ex \) and one endogenous regressor \( en \) with instrument \( in \), the appropriate formula would be \( y \sim ex + en \mid ex + in \). Equivalently, this can be specified as \( y \sim ex + en \mid .-en + in \), i.e., by providing an update formula with a \( . \) in the second part of the formula.

Value

An object of class svyivreg

References


See Also

ivreg
svykappa  

\textit{Cohen's kappa for agreement}

\textbf{Description}

Computes the unweighted kappa measure of agreement between two raters and the standard error. The measurements must both be factor variables in the survey design object.

\textbf{Usage}

\begin{verbatim}
svykappa(formula, design, ...)
\end{verbatim}

\textbf{Arguments}

- \textit{formula} \enspace \textit{one-sided formula giving two measurements}
- \textit{design} \enspace \textit{survey design object}
- \textit{...} \enspace \textit{for future expansion}

\textbf{Value}

Object of class \texttt{svystat}

\textbf{See Also}

\texttt{svycontrast}

\textbf{Examples}

\begin{verbatim}
data(api)
dclus1<-svydesign(id=~dnum, weights=~pw, data=apiclus1, fpc=~fpc)
svykappa(~comp.imp+sch.wide, dclus1)

svykappa(~stype+stypecopy,dclus1)
\end{verbatim}

\svykappa

\textit{Estimate survival function.}

\textbf{Description}

Estimates the survival function using a weighted Kaplan-Meier estimator.
Usage

svykm(formula, design, se=FALSE, ...)  
## S3 method for class 'svykm'
plot(x, xlab="time", ylab="Proportion surviving",  
ylim=c(0,1), ci=NULL, lty=1,...)
## S3 method for class 'svykm'
lines(x, xlab="time", type="s", ci=FALSE, lty=1,...)
## S3 method for class 'svykmlist'
plot(x, pars=NULL, ci=FALSE,...)
## S3 method for class 'svykm'
quantile(x, probs=c(0.75,0.5,0.25), ci=FALSE, level=0.95,...)
## S3 method for class 'svykm'
confint(object, parm, level=0.95,...)

Arguments

formula Two-sided formula. The response variable should be a right-censored Surv object
design survey design object
se Compute standard errors? This is slow for moderate to large data sets
... in plot and lines methods, graphical parameters
x a svykm or svykmlist object
xlab, ylab, ylim, type as for plot
lty Line type, see par
ci Plot (or return, forquantile) the confidence interval
pars A list of vectors of graphical parameters for the separate curves in a svykmlist object
object A svykm object
parm vector of times to report confidence intervals
level confidence level
probs survival probabilities for computing survival quantiles (note that these are the complement of the usual quantile input, so 0.9 means 90% surviving, not 90% dead)

Details

When standard errors are computed, the survival curve is actually the Aalen (hazard-based) estimator rather than the Kaplan-Meier estimator.

The standard error computations use memory proportional to the sample size times the square of the number of events. This can be a lot.

In the case of equal-probability cluster sampling without replacement the computations are essentially the same as those of Williams (1995), and the same linearization strategy is used for other designs.
Confidence intervals are computed on the log(survival) scale, following the default in survival package, which was based on simulations by Link (1984).

Confidence intervals for quantiles use Woodruff’s method: the interval is the intersection of the horizontal line at the specified quantile with the pointwise confidence band around the survival curve.

Value

For svykm, an object of class svykm for a single curve or svykmlist for multiple curves.

References


Woodruff RS (1952) Confidence intervals for medians and other position measures. JASA 57, 622-627.

See Also

predict.svycoxph for survival curves from a Cox model

Examples

data(pbc, package="survival")
pbc$randomized <- with(pbc, !is.na(trt) & trt>0)
biasmodel<-glm(randomized~age*edema, data=pbc)
pbc$randprob<-fitted(biasmodel)

dpbc<-svydesign(id=~1, prob=~randprob, strata=~edema, data=subset(pbc,randomized))
s1<-svykm(Surv(time,status>0)>1, design=dpbc)
s2<-svykm(Surv(time,status>0)>I(bili>6), design=dpbc)

plot(s1)
plot(s2)
plot(s2, lwd=2, pars=list(lty=c(1,2),col=c("purple","forestgreen")))

quantile(s1, probs=c(0.9,0.75,0.5,0.25,0.1))
s3<-svykm(Surv(time,status>0)>I(bili>6), design=dpbc,se=TRUE)
plot(s3[[2]],col="purple")

confint(s3[[2]], parm=365*(1:5))
quantile(s3[[1]], ci=TRUE)
svyloglin  

*Loglinear models*

**Description**

Fit and compare hierarchical loglinear models for complex survey data.

**Usage**

```r
svyloglin(formula, design, ...)  
update(object, formula, ...)  
anova(object, object1, ..., integrate=FALSE)  
print(x, pval=c("F", "saddlepoint", "lincom", "chisq"), ...)  
coef(object, ..., intercept=FALSE)
```

**Arguments**

- `formula`: Model formula
- `design`: survey design object
- `object, object1`: loglinear model from `svyloglin`
- `pval`: p-value approximation: see Details
- `integrate`: Compute the exact asymptotic p-value (slow)?
- `...`: not used
- `x`: anova object

**Details**

The loglinear model is fitted to a multiway table with probabilities estimated by `svymeans` and with the sample size equal to the observed sample size, treating the resulting table as if it came from iid multinomial sampling, as described by Rao and Scott. The variance-covariance matrix does not include the intercept term, and so by default neither does the `coef` method. A Newton-Raphson algorithm is used, rather than iterative proportional fitting, so starting values are not needed.

The `anova` method computes the quantities that would be the score (Pearson) and likelihood ratio chi-squared statistics if the data were an iid sample. It computes four p-values for each of these, based on the exact asymptotic distribution (see `pchisqsum`), a saddlepoint approximateion to this distribution, a scaled chi-squared distribution, and a scaled F-distribution. When testing the two-way interaction model against the main-effects model in a two-way table the score statistic and p-values match the Rao-Scott tests computed by `svychi2q`.

The `anova` method can only compare two models if they are for exactly the same multiway table (same variables and same order). The update method will help with this. It is also much faster to
use update than svyloglin for a large data set: its time complexity depends only on the size of the
model, not on the size of the data set.
It is not possible to fit a model using a variable created inline, eg I(x<10), since the multiway table
is based on all variables used in the formula.

Value
Object of class "svyloglin"

References

See Also
svychisq, svyglm,pchisqsum

Examples
data(api)
dclus1<-svydesign(id=~dnum, weights=~pw, data=apiclus1, fpc=~fpc)
a<-svyloglin(~stype+comp.imp,dclus1)
b<-update(a,~.^2)
an<-anova(a,b)
an
print(an, pval="saddlepoint")

## Wald test
regTermTest(b, ~stype:comp.imp)

## linear-by-linear association
d<-update(a,~as.numeric(stype):as.numeric(comp.imp))
an1<-anova(a,d)
an1

svylogrank

Compare survival distributions

Description
Computes a weighted version of the logrank and stratified logrank tests for comparing two or more
survival distributions. The generalization to complex samples is based on the characterization of the
logrank test as the score test in a Cox model, Under simple random sampling with replacement, this
function with rho=0 and gamma=0 is almost identical to the robust score test in the survival package.
Usage

svylogrank(formula, design, rho=0, gamma=0, method=c("small","large","score"), ...)

Arguments

formula      Model formula with a single predictor and optionally a strata term. The predictor must be a factor if it has more than two levels.
design       A survey design object
rho, gamma   Coefficients for the Harrington/Fleming G-rho-gamma tests. The default is the logrank test, rho=1 gives a generalised Wilcoxon test
method       "small" works faster when a matrix with dimension number of events by number of people fits easily in memory; "large" works faster for large data sets; "score" works by brute-force construction of an expanded data set, and is for debugging
...          for future expansion.

Value

A vector containing the z-statistic for comparing each level of the variable to the lowest, the chisquared statistic for the logrank test, and the p-value.

References


See Also

svykm, svycoxph.

Examples

library("survival")
data(nwtco)
## stratified on case status
dcchs<-twophase(id=list(~seqno,~seqno), strata=list(NULL,~rel),
        subset=~I(in.subcohort | rel), data=nwtco, method="simple")
svylogrank(Surv(edrel,rel)~factor(stage),design=dcchs)
data(pbc, package="survival")
pbc$randomized <- with(pbc, !is.na(trt) & trt>0)
biasmodel<-.glm(randomized~age*edema,data=pbc)
pbc$randprob<-fitted(biasmodel)
dpbc<-svydesign(id=~1, prob=~randprob, strata=~edema, data=subset(pbc,randomized))
svylogrank(Surv(time,status==2)~trt,design=dpbc)
svylogrank(Surv(time,status==2)~trt,design=dpbc,rho=1)
svymle

Maximum pseudolikelihood estimation in complex surveys

Description

Maximises a user-specified likelihood parametrised by multiple linear predictors to data from a complex sample survey and computes the sandwich variance estimator of the coefficients. Note that this function maximises an estimated population likelihood, it is not the sample MLE.

Usage

svymle(loglike, gradient = NULL, design, formulas, start = NULL, control = list(), na.action = "na.fail", method = NULL, lower = NULL, upper = NULL, ...

# S3 method for class 'svymle'
summary(object, stderr = c("robust", "model"),...)

Arguments

loglike     vectorised loglikelihood function
gradient    Derivative of loglike. Required for variance computation and helpful for fitting
design      a survey.design object
formulas    A list of formulas specifying the variable and linear predictors: see Details below
start       Starting values for parameters
control     control options for the optimiser: see the help page for the optimiser you are using.
lower,upper Parameter bounds for bobyqa
na.action   Handling of NAs
method      "nlm" to use nlm,"uobyqa" or "bobyqa" to use those optimisers from the minqa package; otherwise passed to optim
...         Arguments to loglike and gradient that are not to be optimised over.
object      svymle object
stderr      Choice of standard error estimator. The default is a standard sandwich estimator. See Details below.
Details

Optimization is done by \texttt{nlm} by default or if \texttt{method}="\texttt{nlm}". Otherwise \texttt{optim} is used and \texttt{method} specifies the method and \texttt{control} specifies control parameters.

The design object contains all the data and design information from the survey, so all the formulas refer to variables in this object. The \texttt{formulas} argument needs to specify the response variable and a linear predictor for each freely varying argument of \texttt{loglike}.

Consider for example the \texttt{dnorm} function, with arguments \texttt{x}, \texttt{mean}, \texttt{sd} and \texttt{log}, and suppose we want to estimate the mean of \( y \) as a linear function of a variable \( z \), and to estimate a constant standard deviation. The \texttt{log} argument must be fixed at FALSE to get the loglikelihood. A \texttt{formulas} argument would be \texttt{list(\#y, mean=\#z, sd=\#1)}. Note that the data variable \( y \) must be the first argument to \texttt{dnorm} and the first formula and that all the other formulas are labelled. It is also permitted to have the data variable as the left-hand side of one of the formulas: eg \texttt{list( mean=\#z, sd=\#1)}.

The two optimisers from the \texttt{minqa} package do not use any derivatives to be specified for optimisation, but they do assume that the function is smooth enough for a quadratic approximation, ie, that two derivatives exist.

The usual variance estimator for MLEs in a survey sample is a ‘sandwich’ variance that requires the score vector and the information matrix. It requires only sampling assumptions to be valid (though some model assumptions are required for it to be useful). This is the \texttt{stderr="robust"} option, which is available only when the \texttt{gradient} argument was specified.

If the model is correctly specified and the sampling is at random conditional on variables in the model then standard errors based on just the information matrix will be approximately valid. In particular, for independent sampling where weights and strata depend on variables in the model the \texttt{stderr="model"} should work fairly well.

Value

An object of class \texttt{svymle}

Author(s)

Thomas Lumley

See Also

\texttt{svydesign}, \texttt{svyglm}

Examples

data(api)

dstrat<-svydesign(id=~1, strata=~stype, weight=~pw, fpc=~fpc, data=apistrat)

## fit with glm
m0 <- svyglm(api00~api99+ell,family="gaussian",design=dstrat)
## fit as mle (without gradient)
m1 <- svymle(loglike=dnorm,gradient=NULL, design=dstrat,
  formulas=list(mean=api00~api99+ell, sd=\#1),
  }
## with gradient

```r
gr <- function(x, mean, sd, log) {
  dm <- -2 * (x - mean) / (2 * sd^2)
  ds <- (x - mean) * 2 * (2 * (2 + sd)) / (2 * sd^2)^2 - sqrt(2 * pi) / (sd * sqrt(2 * pi))
  cbind(dm, ds)
}
```

```r
m2 <- svymle(loglike = dnorm, gradient = gr, design = dstrat,
            formulas = list(mean = api00 ~ api99 + ell, sd = ~1),
            start = list(c(80, 1, 0), c(20)), log = TRUE, method = "BFGS")
```

## Using offsets

```r
m3 <- svymle(loglike = dnorm, gradient = gr, design = dstrat,
            formulas = list(mean = api00 ~ api99 + offset(ell) + ell, sd = ~1),
            start = list(c(80, 1, 0), c(20)), log = TRUE, method = "BFGS")
```

## demonstrating multiple linear predictors

```r
m3 <- svymle(loglike = dnorm, gradient = gr, design = dstrat,
            formulas = list(mean = api00 ~ api99 + offset(ell) + ell, sd = ~stype),
            start = list(c(80, 1, 0), c(20, 0, 0)), log = TRUE, method = "BFGS")
```

## More complicated censored lognormal data example

```r
# showing that the response variable can be multivariate
data(pbc, package = "survival")
pbc$rrandomized <- with(pbc, !is.na(trt) & trt > 0)
biasmodel <- glm(randomized ~ age * edema, data = pbc)
pbc$randprob <- fitted(biasmodel)
dpbc <- svydesign(id = ~1, prob = randprob, strata = ~edema,
data = subset(pbc, randomized))
```

## censored logNormal likelihood

```r
lcens <- function(x, mean, sd) {
  ifelse(x[, 2] == 1,
    dnorm(log(x[, 1]), mean, sd, log = TRUE),
    pnorm(log(x[, 1]), mean, sd, log = TRUE, lower.tail = FALSE)
  )
}
```

```r
gcens <- function(x, mean, sd) {
  dz <- -dnorm(log(x[, 1]), mean, sd) / pnorm(log(x[, 1]), mean, sd, lower.tail = FALSE)
  dm <- ifelse(x[, 2] == 1,
```
svyolr

Proportional odds and related models

Description

Fits cumulative link models: proportional odds, probit, complementary log-log, and cauchit.

Usage

svyolr(formula, design, ...)  # S3 method for class 'survey.design2'
svyolr(formula, design, start, ..., na.action = na.omit, method = c("logistic", "probit", "cloglog", "cauchit"))  # S3 method for class 'svyrep.design'
svyolr(formula, design, ..., return.replicates=FALSE, multicore=getOption("survey.multicore"))
svyplot 99

Arguments

formula Formula: the response must be a factor with at least three levels
design survey design object
... dots
start Optional starting values for optimization
na.action handling of missing values
multicore Use multicore package to distribute computation of replicates across multiple processors?
method Link function
return.replicates return the individual replicate-weight estimates

Value

An object of class svyolr

Author(s)

The code is based closely on polr() from the MASS package of Venables and Ripley.

See Also

svyglm, regTermTest

Examples

data(api)
dclus1<-svydesign(id=~dnum, weights=~pw, data=apiclus1, fpc=~fpc)
dclus1<-update(dclus1, mealcat=cut(meals,c(0,25,50,75,100)))
m<-svyolr(mealcat~avg.ed+mobility+stype, design=dclus1)
m

## Use regTermTest for testing multiple parameters
regTermTest(m, ~avg.ed+stype, method="LRT")
Usage

svyplot(formula, design,...)
## Default S3 method:
svyplot(formula, design, style = c("bubble", "hex", "grayhex", "subsample","transparent"),
sample.size = 500, subset = NULL, legend = 1, inches = 0.05,
amount=NULL, basecol="black",
alpha=c(0, 0.8),xbins=30,...)

Arguments

  formula       A model formula
  design        A survey object (svydesign or svrepdesign)
  style         See Details below
  sample.size   For style="subsample"
  subset        expression using variables in the design object
  legend        For style="hex" or "grayhex"
  inches        Scale for bubble plots
  amount        list with x and y components for amount of jittering to use in subsample plots,
or NULL for the default amount
  basecol       base color for transparent plots, or a function to compute the color (see below),
or color for bubble plots
  alpha         minimum and maximum opacity for transparent plots
  xbins         Number of (x-axis) bins for hexagonal binning
  ...           Passed to plot methods

Details

Bubble plots are scatterplots with circles whose area is proportional to the sampling weight. The
two "hex" styles produce hexagonal binning scatterplots, and require the hexbin package from
Bioconductor. The "transparent" style plots points with opacity proportional to sampling weight.
The subsample method uses the sampling weights to create a sample from approximately the pop-
ulation distribution and passes this to plot
Bubble plots are suited to small surveys, hexagonal binning and transparency to large surveys where
plotting all the points would result in too much overlap.
basecol can be a function taking one data frame argument, which will be passed the data frame of
variables from the survey object. This could be memory-intensive for large data sets.

Value

None

References

Lumley T, Scott A (2017) "Fitting Regression Models to Survey Data" Statistical Science 32: 265-
278
svyprcomp

Sampling-weighted principal component analysis

Description

Computes principal components using the sampling weights.

See Also

symbols for other options (such as colour) for bubble plots.

svytable for plots of discrete data.

Examples

data(api)
dstrat<-svydesign(id=~1,strata=~stype, weights=~pw, data=apistrat, fpc=~fpc)
svyplot(api00~api99, design=dstrat, style="bubble")
svyplot(api00~api99, design=dstrat, style="transparent",pch=19)

## these two require the hexbin package
svyplot(api00~api99, design=dstrat, style="hex", xlab="1999 API",ylab="2000 API")
svyplot(api00~api99, design=dstrat, style="grayhex",legend=0)

dclus2<-svydesign(id=~dnum+snum, weights=~pw, data=apiclus2, fpc=~fpc1+fpc2)
svyplot(api00~api99, design=dclus2, style="subsample")
svyplot(api00~api99, design=dclus2, style="subsample",
        amount=list(x=25,y=25))
svyplot(api00~api99, design=dstrat, basecol=function(df){c("goldenrod","tomato","sienna")[as.numeric(df$stype)]},
        style="transparent",pch=19,alpha=c(0,1))
legend("topleft",col=c("goldenrod","tomato","sienna"), pch=19, legend=c("E","H","M"))

## For discrete data, estimate a population table and plot the table.
plot(svytable(~sch.wide+comp.imp+stype,design=dstrat))
fourfoldplot(svytable(~sch.wide+comp.imp+stype,design=dstrat,round=TRUE))

## To draw on a hexbin plot you need grid graphics, eg,
library(grid)
h<-svyplot(api00~api99, design=dstrat, style="hex", xlab="1999 API",ylab="2000 API")
s<-svysmooth(api00~api99,design=dstrat)
grid.polyline(s$sapi99$x,s$sapi99$y,vp=h$plot.vp@hexVp.on,default.units="native",
gp=gpar(col="red",lwd=2))
svyprcomp

Usage

svyprcomp(formula, design, center = TRUE, scale. = FALSE, tol = NULL, scores = FALSE, ...)
  # S3 method for class 'svyprcomp'
biplot(x, cols=c("black","darkred"),xlabs=NULL,
       weight=c("transparent","scaled","none"),
       max.alpha=0.5,max.cex=0.5,xlim=NULL,ylim=NULL,pc.biplot=FALSE,
       expand=1,xlab=NULL,ylab=NULL, arrow.len=0.1, ...)

Arguments

formula model formula describing variables to be used
design survey design object.
center Center data before analysis?
scale. Scale to unit variance before analysis?
tol Tolerance for omitting components from the results; a proportion of the standard
deviation of the first component. The default is to keep all components.
scores Return scores on each component? These are needed for biplot.
x A svyprcomp object
cols Base colors for observations and variables respectively
xlabs Formula, or character vector, giving labels for each observation
weight How to display the sampling weights: "scaled" changes the size of the point
label, "transparent" uses opacity proportional to sampling weight, "none"
changes neither.
max.alpha Opacity for the largest sampling weight, or for all points if weight!="transparent"
max.cex Character size (as a multiple of par("cex")) for the largest sampling weight, or
for all points if weight!="scaled"
xlim,ylim,xlab,ylab Graphical parameters
expand,arrow.len See biplot
pc.biplot See link{biplot.prcomp}
... Other arguments to prcomp, or graphical parameters for biplot

Value

svyprcomp returns an object of class svyprcomp, similar to class prcomp but including design
information

See Also

prcomp,biplot.prcomp
Examples

data(api)
dclus2<-svydesign(id=~dnum+snum, fpc=~fpc1+fpc2, data=apiclus2)

pc <- svyprcomp(~api99+api00+ell+hsg+meals+emer, design=dclus2, scale=TRUE, scores=TRUE)

biplot(pc, xlabs=~dnum, weight="none")

biplot(pc, xlabs=~dnum, max.alpha=1)

biplot(pc, weight="scaled", max.cex=1.5, xlabs=~dnum)

svypredmeans

Predictive marginal means

Description

Predictive marginal means for a generalised linear model, using the method of Korn and Graubard (1999) and matching the results of SUDAAN. The predictive marginal mean for one level of a factor is the probability-weighted average of the fitted values for the model on new data where all the observations are set to that level of the factor but have whatever values of adjustment variables they really have.

Usage

svypredmeans(adjustmodel, groupfactor)

Arguments

adjustmodel A generalised linear model fit by svyglm with the adjustment variable but without the factor for which predictive means are wanted

groupfactor A one-sided formula specifying the factor for which predictive means are wanted. Can use, eg, ~interaction(race,sex) for combining variables

Value

An object of class svystat with the predictive marginal means and their covariance matrix.

Note

It is possible to supply an adjustment model with only an intercept, but the results are then the same as svymean

It makes no sense to have a variable in the adjustment model that is part of the grouping factor, and will give an error message or NA.
svyquantile

Quantiles for sample surveys

Description

Compute quantiles for data from complex surveys.

Usage

```r
svyquantile(x, design, quantiles, alpha=0.05,
  ci=FALSE, method = "linear", f = 1,
  interval.type=c("Wald","score","betaWald"), na.rm=FALSE,se=ci,
  ties=c("discrete","rounded"), df=NULL,...)
```

References


See Also

svyglm

Worked example using National Health Interview Survey data: https://gist.github.com/tslumley/2e74cd0ac12a671d2724

Examples

```r
data(nhanes)
nhanes_design <- svydesign(id=~SDMVPSU, strata=~SDMVSTRA, weights=~WTMEC2YR, nest=TRUE,data=nhanes)
agesexmodel<-svyglm(HI_CHOL~agecat+RIAGENDR, design=nhanes_design,family=quasibinomial)
## high cholesterol by race/ethnicity, adjusted for demographic differences
means<-svypredmeans(agesexmodel, ~race)
means
## relative risks compared to non-Hispanic white
svycontrast(means,quote(`1`/`2`))
svycontrast(means,quote(`3`/`2`))
```
svyquantile

Arguments

x A formula, vector or matrix
design survey.design or svyrep.design object
quantiles Quantiles to estimate
method see approxfun
f see approxfun
ci Compute a confidence interval? (relatively slow; needed for svyby)
se Compute standard errors from the confidence interval length?
alpha Level for confidence interval
interval.type See Details below
ties See Details below
df Degrees of freedom for a t-distribution. Inf requests a Normal distribution, NULL uses degf. Not relevant for type="betaWald"
return.replicates Return the replicate means?
na.rm Remove NAs?
... arguments for future expansion
object Object returned by svyquantile.survey.design

details

The definition of the CDF and thus of the quantiles is ambiguous in the presence of ties. With ties="discrete" the data are treated as genuinely discrete, so the CDF has vertical steps at tied observations. With ties="rounded" all the weights for tied observations are summed and the CDF interpolates linearly between distinct observed values, and so is a continuous function. Combining interval.type="betaWald" and ties="discrete" is (close to) the proposal of Shah and Vaish(2006) used in some versions of SUDAAN.

Interval estimation for quantiles is complicated, because the influence function is not continuous. Linearisation cannot be used directly, and computing the variance of replicates is valid only for some designs (eg BRR, but not jackknife). The interval.type option controls how the intervals are computed.

For survey.design objects the default is interval.type="Wald". A 95% Wald confidence interval is constructed for the proportion below the estimated quantile. The inverse of the estimated CDF is used to map this to a confidence interval for the quantile. This is the method of Woodruff (1952). For "betaWald" the same procedure is used, but the confidence interval for the proportion is computed using the exact binomial cdf with an effective sample size proposed by Korn & Graubard (1998).

If interval.type="score" we use a method described by Binder (1991) and due originally to Francisco and Fuller (1986), which corresponds to inverting a robust score test. At the upper and lower limits of the confidence interval, a test of the null hypothesis that the cumulative distribution function is equal to the target quantile just rejects. This was the default before version 2.9. It is much slower than "Wald", and Dorfman & Valliant (1993) suggest it is not any more accurate.
Standard errors are computed from these confidence intervals by dividing the confidence interval length by $2\cdot \text{qnorm}(\alpha/2)$.

For replicate-weight designs, ordinary replication-based standard errors are valid for BRR and Fay's method, and for some bootstrap-based designs, but not for jackknife-based designs. interval.type="quantile" gives these replication-based standard errors. The default, interval.type="probability" computes confidence on the probability scale and then transforms back to quantiles, the equivalent of interval.type="Wald" for survey.design objects (with alpha=0.05).

There is a confint method for svyquantile objects; it simply extracts the pre-computed confidence interval.

**Value**

returns a list whose first component is the quantiles and second component is the confidence intervals. For replicate weight designs, returns an object of class svyrepstat.

**Author(s)**

Thomas Lumley

**References**


Woodruff RS (1952) Confidence intervals for medians and other position measures. JASA 57, 622-627.

**See Also**

svyk for quantiles of survival curves

svyciprop for confidence intervals on proportions.

**Examples**

```r
data(api)
## population
quantile(api$api00,c(.25,.5,.75))
```
svyranktest

Design-based rank tests

Description

Design-based versions of k-sample rank tests. The built-in tests are all for location hypotheses, but the user could specify others.

Usage

svyranktest(formula, design,
    test = c("wilcoxon", "vanderWaerden", "median", "KruskalWallis"), ...)

Arguments

  formula    Model formula y~g for outcome variable y and group g
  design     A survey design object
Which rank test to use: Wilcoxon, van der Waerden’s normal-scores test, Mood’s test for the median, or a function $f(r, N)$ where $r$ is the rank and $N$ the estimated population size. "KruskalWallis" is a synonym for "wilcoxon" for more than two groups.

... for future expansion

Details

These tests are for the null hypothesis that the population or superpopulation distributions of the response variable are different between groups, targeted at population or superpopulation alternatives. The 'ranks' are defined as quantiles of the pooled distribution of the variable, so they do not just go from 1 to $N$; the null hypothesis does not depend on the weights, but the ranks do.

The tests reduce to the usual Normal approximations to the usual rank tests under iid sampling. Unlike the traditional rank tests, they are not exact in small samples.

Value

Object of class htest

References


See Also

svyttest, svylogrank

Examples

data(api)
dclus1<-svydesign(id=~dnum, weights=~pw, fpc=~fpc, data=apiclus1)

svyranktest(ell~comp.imp, dclus1)
svyranktest(ell~comp.imp, dclus1, test="median")

svyranktest(ell~stype, dclus1)
svyranktest(ell~stype, dclus1, test="median")

## upper quartile
svyranktest(ell~comp.imp, dclus1, test=function(r,N) as.numeric(r>0.75*N))

quantiletest<-function(p){
  rval<-function(r,N) as.numeric(r>(N*p))
  attr(rval,"name")<-paste(p,"quantile")
  rval
}
svyratio  

Ratio estimation

Description

Ratio estimation and estimates of totals based on ratios for complex survey samples. Estimating domain (subpopulation) means can be done more easily with svymean.

Usage

```r
## S3 method for class 'survey.design2'
svyratio(numerator=formula, denominator,
         design, separate=FALSE, na.rm=FALSE, formula, covmat=FALSE, deff=FALSE,...)
## S3 method for class 'svyrep.design'
svyratio(numerator=formula, denominator, design,
         na.rm=FALSE, formula, covmat=FALSE, return.replicates=FALSE, deff=FALSE, ...)
## S3 method for class 'twophase'
svyratio(numerator=formula, denominator, design,
         separate=FALSE, na.rm=FALSE, formula,...)
## S3 method for class 'svyratio'
predict(object, total, se=TRUE,...)
## S3 method for class 'svyratio_separate'
predict(object, total, se=TRUE,...)
## S3 method for class 'svyratio'
SE(object,...,drop=TRUE)
## S3 method for class 'svyratio'
coef(object,...,drop=TRUE)
## S3 method for class 'svyratio'
confint(object, parm, level = 0.95, df =Inf,...)
```

Arguments

- `numerator`: formula, expression, or data frame giving numerator variable(s)
- `denominator`: formula, expression, or data frame giving denominator variable(s)
- `design`: survey design object
- `object`: result of svyratio
- `total`: vector of population totals for the denominator variables in object, or list of vectors of population stratum totals if separate=TRUE
- `se`: Return standard errors?
separate Estimate ratio separately for strata
na.rm Remove missing values?
covmat Compute the full variance-covariance matrix of the ratios
deff Compute design effects
return.replicates Return replicate estimates of ratios
drop Return a vector rather than a matrix
parm a specification of which parameters are to be given confidence intervals, either a vector of numbers or a vector of names. If missing, all parameters are considered.
level the confidence level required.
df degrees of freedom for t-distribution in confidence interval, use\( \text{ddegf}(\text{design}) \) for number of PSUs minus number of strata
... Other unused arguments for other methods

Details

The separate ratio estimate of a total is the sum of ratio estimates in each stratum. If the stratum totals supplied in the total argument and the strata in the design object both have names these names will be matched. If they do not have names it is important that the sample totals are supplied in the correct order, the same order as shown in the output of \text{summary}(\text{design}).

When design is a two-phase design, stratification will be on the second phase.

Value

\text{svyratio} \text{returns an object of class \text{svyratio}. The predict method returns a matrix of population totals and optionally a matrix of standard errors.}

Author(s)

Thomas Lumley

References


See Also

\text{svydesign} \\
\text{svymean} for estimating proportions and domain means \\
\text{calibrate} for estimators related to the separate ratio estimator.
Examples

data(scd)

### survey design objects
scddes<-svydesign(data=scd, prob=~1, id=~ambulance, strata=~ESA, nest=TRUE, fpc=rep(5,6))
scdnofpc<-svydesign(data=scd, prob=~1, id=~ambulance, strata=~ESA, nest=TRUE)

# convert to BRR replicate weights
scd2brr <- as.svrepdesign(scdnofpc, type="BRR")

# use BRR replicate weights from Levy and Lemeshow
repweights<2*cbind(c(1,0,1,0,1,0), c(1,0,0,1,0,1), c(0,1,1,0,0,1),
c(0,1,0,1,1,0))
scdrep<-svrepdesign(data=scd, type="BRR", repweights=repweights)

### ratio estimates
svyratio(~alive, ~arrests, design=scddes)
svyratio(~alive, ~arrests, design=scdnofpc)
svyratio(~alive, ~arrests, design=scd2brr)
svyratio(~alive, ~arrests, design=scdrep)

data(api)
dstrat<-svydesign(id=~1,strata=~stype, weights=~pw, data=apistrat, fpc=~fpc)

### domain means are ratio estimates, but available directly
svyratio(~I(api.stu*(comp.imp=="Yes")), ~as.numeric(comp.imp=="Yes"), dstrat)
svymean(~api.stu, subset(dstrat, comp.imp=="Yes"))

### separate and combined ratio estimates of total
(sep<-svyratio(~api.stu, ~enroll, dstrat,separate=TRUE))
(com<-svyratio(~api.stu, ~enroll, dstrat))

stratum.totals<-list(E=1877350, H=1013824, M=920298)
predict(sep, total=stratum.totals)
predict(com, total=sum(unlist(stratum.totals)))

SE(com)
coef(com)
coef(com, drop=FALSE)
confint(com)

svyrecvar

Variance estimation for multistage surveys

Description

Compute the variance of a total under multistage sampling, using a recursive descent algorithm.
svyrecvar

Usage

svyrecvar(x, clusters, stratas, fpcs, postStrata = NULL, lonely.psu = getOption("survey.lonely.psu"), one.stage=getOption("survey.ultimate.cluster"))

Arguments

x  Matrix of data or estimating functions
clusters  Data frame or matrix with cluster ids for each stage
stratas  Strata for each stage
fpcs  Information on population and sample size for each stage, created by as.fpc
postStrata  post-stratification information as created by postStratify or calibrate
lonely.psu  How to handle strata with a single PSU
one.stage  If TRUE, compute a one-stage (ultimate-cluster) estimator

Details

The main use of this function is to compute the variance of the sum of a set of estimating functions under multistage sampling. The sampling is assumed to be simple or stratified random sampling within clusters at each stage except perhaps the last stage. The variance of a statistic is computed from the variance of estimating functions as described by Binder (1983).

Use one.stage=FALSE for compatibility with other software that does not perform multi-stage calculations, and set options(survey.ultimate.cluster=TRUE) to make this the default.

The idea of a recursive algorithm is due to Bellhouse (1985). Texts such as Cochran (1977) and Sarndal et al (1991) describe the decomposition of the variance into a single-stage between-cluster estimator and a within-cluster estimator, and this is applied recursively.

If one.stage is a positive integer it specifies the number of stages of sampling to use in the recursive estimator.

If pps="brewer", standard errors are estimated using Brewer’s approximation for PPS without replacement, option 2 of those described by Berger (2004). The fpc argument must then be specified in terms of sampling fractions, not population sizes (or omitted, but then the pps argument would have no effect and the with-replacement standard errors would be correct).

Value

A covariance matrix

Note

A simple set of finite population corrections will only be exactly correct when each successive stage uses simple or stratified random sampling without replacement. A correction under general unequal probability sampling (e.g. PPS) would require joint inclusion probabilities (or, at least, sampling probabilities for units not included in the sample), information not generally available.

The quality of Brewer’s approximation is excellent in Berger’s simulations, but the accuracy may vary depending on the sampling algorithm used.
svyrecvar

References


Brewer KRW (2002) Combined Survey Sampling Inference (Weighing Basu’s Elephants) [Chapter 9]


See Also

svrVar for replicate weight designs

svyCprod for a description of how variances are estimated at each stage

Examples

data(mu284)
dmu284<-svydesign(id=~id1+id2,fpc=~n1+n2, data=mu284)
svytotal(~y1, dmu284)

data(api)
# two-stage cluster sample
dclus2<-svydesign(id=~dnum+snum, fpc=~fpc1+fpc2, data=apiclus2)
summary(dclus2)
svymean(~api00, dclus2)
svytotal(~enroll, dclus2,na.rm=TRUE)

# bootstrap for multistage sample
mrclus2<-as.svrepdesign(dclus2, type="mrb", replicates=100)
svytotal(~enroll, mrclus2, na.rm=TRUE)

# two-stage 'with replacement'
dclus2wr<-svydesign(id=~dnum+snum, weights=~pw, data=apiclus2)
summary(dclus2wr)
svymean(~api00, dclus2wr)
svytotal(~enroll, dclus2wr,na.rm=TRUE)
svysmooth  Scatterplot smoothing and density estimation

Description

Scatterplot smoothing and density estimation for probability-weighted data.

Usage

svysmooth(formula, design, ...)  
## Default S3 method:  
svysmooth(formula, design, method = c("locpoly", "quantreg"),  
   bandwidth = NULL, quantile, df = 4, ...)  
## S3 method for class 'svysmooth'  
plot(x, which=NULL, type="l", xlabs=NULL, ylab=NULL,...)  
## S3 method for class 'svysmooth'  
lines(x,which=NULL,...)  
make.panel.svysmooth(design,bandwidth=NULL)

Arguments

formula  One-sided formula for density estimation, two-sided for smoothing  
design  Survey design object  
method  local polynomial smoothing for the mean or regression splines for quantiles  
bandwidth  Smoothing bandwidth for "locpoly" or NULL for automatic choice  
quantile  quantile to be estimated for "quantreg"  
df  Degrees of freedom for "quantreg"  
which  Which plots to show (default is all)  
type  as for plot  
xlabs  Optional vector of x-axis labels  
ylab  Optional y-axis label  
...  More arguments  
x  Object of class svysmooth

Details

svysmooth does one-dimensional smoothing. If formula has multiple predictor variables a separate one-dimensional smooth is performed for each one.

For method="locpoly" the extra arguments are passed to locpoly from the KernSmooth package, for method="quantreg" they are passed to rq from the quantreg package. The automatic choice of bandwidth for method="locpoly" uses the default settings for dp1k and dpill in the KernSmooth package.
make_panel_svysmooth() makes a function that plots points and draws a weighted smooth curve through them, a weighted replacement for panel.smooth that can be passed to functions such as termplot or plot.lm. The resulting function has a span argument that will set the bandwidth; if this is not specified the automatic choice will be used.

**Value**

An object of class svysmooth, a list of lists, each with x and y components.

**See Also**

svyhist for histograms

**Examples**

```r
data(api)
dstrat<-svydesign(id=~1,strata=~stype, weights=~pw, data=apistrat, fpc=~fpc)

smth<-svysmooth(api00~api99+ell,dstrat)
dens<-svysmooth(~api99, dstrat,bandwidth=30)
dens1<-svysmooth(~api99, dstrat)
qsmth<-svysmooth(api00~ell,dstrat, quantile=0.75, df=3,method="quantreg")

plot(smth)
plot(smth, which="ell",lty=2,ylim=c(500,900))
lines(qsmth, col="red")

svyhist(~api99,design=dstrat)
lines(dens, col="purple",lwd=3)
lines(dens1, col="forestgreen",lwd=2)

m<-svyglm(api00~sin(api99/100)+stype, design=dstrat)
termplot(m, data=model.frame(dstrat), partial.resid=TRUE, se=TRUE,
smooth=make.panel.svysmooth(dstrat))
```

---

**svystandardize**

*Direct standardization within domains*

**Description**

In health surveys it is often of interest to standardize domains to have the same distribution of, eg, age as in a target population. The operation is similar to post-stratification, except that the totals for the domains are fixed at the current estimates, not at known population values. This function matches the estimates produced by the (US) National Center for Health Statistics.

**Usage**

```r
svystandardize(design, by, over, population, excluding.missing = NULL)
```
svystandardize

Arguments

- **design**: survey design object
- **by**: A one-sided formula specifying the variables whose distribution will be standardised
- **over**: A one-sided formula specifying the domains within which the standardisation will occur, or ~1 to use the whole population.
- **population**: Desired population totals or proportions for the levels of combinations of variables in `by`
- **excluding.missing**: Optionally, a one-sided formula specifying variables whose missing values should be dropped before calculating the domain totals.

Value

A new survey design object of the same type as the input.

Note

The standard error estimates do not exactly match the NCHS estimates

References


See Also

- postStratify, svyby

Examples

```r
## matches http://www.cdc.gov/nchs/data/databriefs/db92_fig1.png
data(nhanes)
popage <- c(55901, 77670, 72816, 45364)
design <- svydesign(id=~SDMVPSU, strata=~SDMVSTRA, weights=~WTMEC2YR, data=nhanes, nest=TRUE)
stdes <- svystandardize(design, by=~agecat, over=~race+RIAGENDR,
                        population=popage, excluding.missing=~HI_CHOL)
svyby(~HI_CHOL, ~race+RIAGENDR, svymean, design=subset(stdes, agecat!="(0,19]"))
```

```r
data(nhanes)
nhanes_design <- svydesign(ids = ~ SDMVPSU, strata = ~ SDMVSTRA,
                           weights = ~ WTMEC2YR, nest = TRUE, data = nhanes)

## These are the same
nhanes_adj <- svystandardize(update(nhanes_design, all_adults = "1"),
                           by = ~ agecat, over = ~ all_adults,
                           population = c(55901, 77670, 72816, 45364),
                           excluding.missing = ~ HI_CHOL)
```
svysurvreg

svymean(~I(HI_CHOL == 1), nhanes_adj, na.rm = TRUE)

nhanes_adj <- svystandardize(nhanes_design,
    by = ~ agecat, over = ~ 1,
    population = c(55901, 77670, 72816, 45364),
    excluding.missing = ~ HI_CHOL)
svymean(~I(HI_CHOL == 1), nhanes_adj, na.rm = TRUE)

svysurvreg

Fit accelerated failure models to survey data

Description

This function calls survreg from the 'survival' package to fit accelerated failure (accelerated life) models to complex survey data, and then computes correct standard errors by linearisation. It has the same arguments as survreg, except that the second argument is design rather than data.

Usage

## S3 method for class 'survey.design'
svysurvreg(formula, design, weights=NULL, subset=NULL, ...)

Arguments

formula Model formula
design Survey design object, including two-phase designs
weights Additional weights to multiply by the sampling weights. No, I don’t know why you’d want to do that.
subset subset to use in fitting (if needed)
... Other arguments of survreg

Value

Object of class svysurvreg, with the same structure as a survreg object but with NA for the log-likelihood.

Note

The residuals method is identical to that for survreg objects except the weighted option defaults to TRUE.
Examples

data(pbc, package="survival")
pbc$randomized <- with(pbc, !is.na(trt) & trt>0)
biasmodel<-glm(randomized~age*edema,data=pbc)
pbc$randprob<-fitted(biasmodel)
dpbc<-svydesign(id=~1, prob=~randprob, strata=~edema, data=subset(pbc,randomized))

model <- svysurvreg(Surv(time, status>0)~bili+protime+albumin, design=dpbc, dist="weibull")
summary(model)

svytable

Contingency tables for survey data

Description

Contingency tables and chisquared tests of association for survey data.

Usage

## S3 method for class 'survey.design'
svytable(formula, design, Ntotal = NULL, round = FALSE,...)
## S3 method for class 'svyrep.design'
svytable(formula, design, Ntotal = sum(weights(design, "sampling")), round = FALSE,...)
## S3 method for class 'survey.design'
svychisq(formula, design, statistic = c("F", "Chisq","Wald","adjWald","lincom","saddlepoint"),na.rm=TRUE,...)
## S3 method for class 'svyrep.design'
svychisq(formula, design, statistic = c("F", "Chisq","Wald","adjWald","lincom","saddlepoint"),na.rm=TRUE,...)
## S3 method for class 'svytable'
summary(object, statistic = c("F","Chisq","Wald","adjWald","lincom","saddlepoint"),...)
degf(design, ...)
## S3 method for class 'survey.design2'
degf(design, ...)
## S3 method for class 'svyrep.design'
degf(design, tol=1e-5,...)

Arguments

formula  Model formula specifying margins for the table (using + only)
design   survey object
statistic See Details below
Ntotal   A population total or set of population stratum totals to normalise to.
The `svytable` function computes a weighted crosstabulation. This is especially useful for producing graphics. It is sometimes easier to use `svytotal` or `svymean`, which also produce standard errors, design effects, etc.

The frequencies in the table can be normalised to some convenient total such as 100 or 1.0 by specifying the `Ntotal` argument. If the formula has a left-hand side the mean or sum of this variable rather than the frequency is tabulated.

The `Ntotal` argument can be either a single number or a data frame whose first column gives the (first-stage) sampling strata and second column the population size in each stratum. In this second case the `svytable` command performs 'post-stratification': tabulating and scaling to the population within strata and then adding up the strata.

As with other `xtabs` objects, the output of `svytable` can be processed by `ftable` for more attractive display. The summary method for `svytable` objects calls `svychisq` for a test of independence. `svychisq` computes first and second-order Rao-Scott corrections to the Pearson chisquared test, and two Wald-type tests.

The default (`statistic="F"`) is the Rao-Scott second-order correction. The p-values are computed with a Satterthwaite approximation to the distribution and with denominator degrees of freedom as recommended by Thomas and Rao (1990). The alternative `statistic="Chisq"` adjusts the Pearson chisquared statistic by a design effect estimate and then compares it to the chisquared distribution it would have under simple random sampling.

The `statistic="Wald"` test is that proposed by Koch et al (1975) and used by the SUDAAN software package. It is a Wald test based on the differences between the observed cells counts and those expected under independence. The adjustment given by `statistic="adjWald"` reduces the statistic when the number of PSUs is small compared to the number of degrees of freedom of the test. Thomas and Rao (1990) compare these tests and find the adjustment benefical.

`statistic="lincom"` replaces the numerator of the Rao-Scott F with the exact asymptotic distribution, which is a linear combination of chi-squared variables (see `pchisqsum`, and `statistic="saddlepoint"` uses a saddlepoint approximation to this distribution. The `CompQuadForm` package is needed for `statistic="lincom"` but not for `statistic="saddlepoint"`. The saddlepoint approximation is especially useful when the p-value is very small (as in large-scale multiple testing problems).

For designs using replicate weights the code is essentially the same as for designs with sampling structure, since the necessary variance computations are done by the appropriate methods of `svytotal` and `svymean`. The exception is that the degrees of freedom is computed as one less than the rank of the matrix of replicate weights (by `degf`).

At the moment, `svychisq` works only for 2-dimensional tables.
Value

The table commands return an xtabs object, svychisq returns a htest object.

Note

Rao and Scott (1984) leave open one computational issue. In computing 'generalised design effects' for these tests, should the variance under simple random sampling be estimated using the observed proportions or the the predicted proportions under the null hypothesis? svychisq uses the observed proportions, following simulations by Sribney (1998), and the choices made in Stata.

References


Koch, GG, Freeman, DH, Freeman, JL (1975) "Strategies in the multivariate analysis of data from complex surveys" International Statistical Review 43: 59-78


Thomas, DR, Rao, JNK (1990) "Small-sample comparison of level and power for simple goodness-of-fit statistics under cluster sampling" JASA 82:630-636

See Also

svytotal and svymean report totals and proportions by category for factor variables.
See svyby and ftable.svystat to construct more complex tables of summary statistics.
See svyloglin for loglinear models.
See regTermTest for Rao-Scott tests in regression models.
See https://notstatschat.rbind.io/2019/06/08/design-degrees-of-freedom-brief-note/ for an explanation of the design degrees of freedom with replicate weights.

Examples

data(api)
xtabs(~sch.wide+stype, data=apipop)
dclus1<-svydesign(id=~dnum, weights=~pw, data=apiclus1, fpc=~fpc)
summary(dclus1)
(tbl <- svytable(~sch.wide+stype, dclus1))
plot(tbl)
fourfoldplot(svytable(~sch.wide+comp.imp+stype,design=dclus1,round=TRUE), conf.level=0)
sychisq(~sch.wide+stype, dclus1)
svyttest

Design-based t-test

Description
One-sample or two-sample t-test. This function is a wrapper for svymean in the one-sample case and for svyglm in the two-sample case. Degrees of freedom are degf(design)-1 for the one-sample test and degf(design)-2 for the two-sample case.

Usage
svyttest(formula, design, ...)

Arguments

formula Formula, outcome~group for two-sample, outcome~0 or outcome~1 for one-sample
design survey design object
...

for methods

Value
Object of class htest

See Also
t.test

Examples
data(api)dclus2<-svydesign(id=~dnum+snum, fpc=~fpc1+fpc2, data=apiclus2)tt<-svyttest(enroll~comp.imp, dclus2)tt

confint(tt, level=0.9)

svyttest(I(api00-api99)~0, dclus2)
trimWeights  

Trim sampling weights

Description

Trims very high or very low sampling weights to reduce the influence of outlying observations. In a replicate-weight design object, the replicate weights are also trimmed. The total amount trimmed is divided among the observations that were not trimmed, so that the total weight remains the same.

Usage

trimWeights(design, upper = Inf, lower = -Inf, ...)  

## S3 method for class 'survey.design2'
trimWeights(design, upper = Inf, lower = -Inf, strict=FALSE,...)  

## S3 method for class 'svyrep.design'
trimWeights(design, upper = Inf, lower = -Inf,compress=FALSE,...)

Arguments

design  
A survey design object

upper  
Upper bound for weights

lower  
Lower bound for weights

strict  
The reapportionment of the ‘trimmings’ from the weights can push other weights over the limits. If trim=TRUE the function calls itself recursively to prevent this.

compress  
Compress the replicate weights after trimming.

...  
Other arguments for future expansion

Value

A new survey design object with trimmed weights.

See Also

`calibrate` has a `trim` option for trimming the calibration adjustments.

Examples

data(api)
dclus1<-svydesign(id=~dnum, weights=~pw, data=apiclus1, fpc=~fpc)

corrected.weights<-c(~(Intercept)=6194, stypeH=755, stypeM=1018, api99=3914069)
dclus1g<-calibrate(dclus1, ~stype+api99, pop.totals)

summary(weights(dclus1g))
dclus1t<-trimWeights(dclus1g,lower=20, upper=45)
summary(weights(dclus1t))
dclus1tt<-trimWeights(dclus1g, lower=20, upper=45,strict=TRUE)
summary(weights(dclus1tt))

svymean(~api99+api00+stype, dclus1g)
svymean(~api99+api00+stype, dclus1t)
svymean(~api99+api00+stype, dclus1tt)

twophase

Two-phase designs

Description

In a two-phase design a sample is taken from a population and a subsample taken from the sample, typically stratified by variables not known for the whole population. The second phase can use any design supported for single-phase sampling. The first phase must currently be one-stage element or cluster sampling.

Usage

twophase(id, strata = NULL, probs = NULL, weights = NULL, fpc = NULL, subset, data, method=c("full","approx","simple"))
twophasevar(x,design)
twophase2var(x,design)

Arguments

id list of two formulas for sampling unit identifiers
strata list of two formulas (or NULLs) for stratum identifies
probs list of two formulas (or NULLs) for sampling probabilities
weights Only for method="approx", list of two formulas (or NULLs) for sampling weights
fpc list of two formulas (or NULLs) for finite population corrections
subset formula specifying which observations are selected in phase 2
data Data frame will all data for phase 1 and 2
method "full" requires (much) more memory, but gives unbiased variance estimates for general multistage designs at both phases. "simple" or "approx" uses the standard error calculation from version 3.14 and earlier, which uses much less memory and is correct for designs with simple random sampling at phase one and stratified random sampling at phase two.
x probability-weighted estimating functions
design two-phase design
Details

The population for the second phase is the first-phase sample. If the second phase sample uses stratified (multistage cluster) sampling without replacement and all the stratum and sampling unit identifier variables are available for the whole first-phase sample it is possible to estimate the sampling probabilities/weights and the finite population correction. These would then be specified as NULL.

Two-phase case-control and case-cohort studies in biostatistics will typically have simple random sampling with replacement as the first stage. Variances given here may differ slightly from those in the biostatistics literature where a model-based estimator of the first-stage variance would typically be used.

Variance computations are based on the conditioning argument in Section 9.3 of Sarndal et al. Method "full" corresponds exactly to the formulas in that reference. Method "simple" or "approx" (the two are the same) uses less time and memory but is exact only for some special cases. The most important special case is the two-phase epidemiologic designs where phase 1 is simple random sampling from an infinite population and phase 2 is stratified random sampling. See the tests directory for a worked example. The only disadvantage of method="simple" in these cases is that standardization of margins (marginpred) is not available.

For method="full", sampling probabilities must be available for each stage of sampling, within each phase. For multistage sampling this requires specifying either fpc or probs as a formula with a term for each stage of sampling. If no fpc or probs are specified at phase 1 it is treated as simple random sampling from an infinite population, and population totals will not be correctly estimated, but means, quantiles, and regression models will be correct.

Value

twophase returns an object of class twophase2 (for method="full") or twophase. The structure of twophase2 objects may change as unnecessary components are removed.

twophase2var and twophase2var return a variance matrix with an attribute containing the separate phase 1 and phase 2 contributions to the variance.

References


See Also

svydesign, svyrecvar for multi*stage* sampling
calibrate for calibration (GREG) estimators.
estWeights for two-phase designs for missing data.
The "epi" and "phase1" vignettes for examples and technical details.

Examples

```r
## two-phase simple random sampling.
data(pbc, package="survival")
pbc$randomized<-with(pbc, !is.na(trt) & trt>0)
pbc$id<-1:nrow(pbc)
d2pbc<-twophase(id=list(~id,~id), data=pbc, subset=~randomized)
svymean(~bili, d2pbc)

## two-stage sampling as two-phase
data(mu284)
ii<-with(mu284, c(1:15, rep(1:5,n2[1:5]-3)))
mu284.1$id<-1:nrow(mu284.1)
mu284.1$sub<-rep(c(TRUE,FALSE),c(15,34-15))
dmu284<-svydesign(id=~id1+id2, fpc=~n1+n2, data=mu284)
## first phase cluster sample, second phase stratified within cluster
d2mu284<-twophase(id=list(~id1,-id), strata=list(NULL,~id1),
                  fpc=list(~n1,NULL), data=mu284.1, subset=~sub)
svytotal(~y1, dmu284)
svytotal(~y1, d2mu284)
svymean(~y1, dmu284)
svymean(~y1, d2mu284)

## case-cohort design: this example requires R 2.2.0 or later
library("survival")
data(nwtco)
dcchs<-twophase(id=list(~seqno,~seqno), strata=list(NULL,~rel),
                 subset=~I(in.subcohort | rel), data=nwtco)
svycoxph(Surv(edrel,rel)~factor(stage)+factor(histol)+I(age/12), design=dcchs)

## Using survival::cch
subcoh <- nwtco$in.subcohort
selccoh <- with(nwtco, rel==1|subcoh==1)
ccoh.data <- nwtco[selccoh,]
ccoh.data$subcohort <- ~subcoh[selccoh]
cch(Surv(edrel, rel) ~ factor(stage) + factor(histol) + I(age/12),
    data =ccoh.data, subcoh = ~subcohort, id=~seqno, cohort.size=4028, method="LinYing")

## two-phase case-control
## Similar to Breslow & Chatterjee, Applied Statistics (1999) but with
## a slightly different version of the data set
nwtco$incc2<-as.logical(with(nwtco, ifelse(rel | instit==2,1,rbinom(nrow(nwtco),1,.1))))
dccs2<-twophase(id=list(~seqno,~seqno), strata=list(NULL,~interaction(rel,instit)),
                 data=nwtco, subset=~incc2)
dccs8<-twophase(id=list(~seqno,~seqno), strata=list(NULL,~interaction(rel,stage,instit)),
                 data=nwtco, subset=~incc2)
```
## update.survey.design

### Description

Update the data variables in a survey design, either with a formula for a new set of variables or with an expression for variables to be added.

### Usage

```r
## S3 method for class 'survey.design'
update(object, ...)  
## S3 method for class 'twophase'
update(object, ...)  
## S3 method for class 'svyrep.design'
update(object, ...)  
## S3 method for class 'DBIsvydesign'
update(object, ...)  
```

### Arguments

- `object` a survey design object
- `...` Arguments `tag=expr` add a new variable `tag` computed by evaluating `expr` in the survey data.
Details

Database-backed objects may not have write access to the database and so update does not attempt to modify the database. The expressions are stored and are evaluated when the data is loaded.

If a set of new variables will be used extensively it may be more efficient to modify the database, either with SQL queries from the R interface or separately. One useful intermediate approach is to create a table with the new variables and a view that joins this table to the table of existing variables.

There is now a base-R function `transform` for adding new variables to a data frame, so I have added `transform` as a synonym for `update` for survey objects.

Value

A survey design object

See Also

svydesign, svrepdesign, twophase

Examples

data(api)
dstrat<-svydesign(id=~1, strata=~stype, weights=~pw, data=apistrat, fpc=~fpc)
dstrat<-update(dstrat, apidiff=api00-api99)
svymean(~api99+api00+apidiff, dstrat)

weights.survey.design  Survey design weights

Description

Extract weights from a survey design object.

Usage

```r
## S3 method for class 'survey.design'
weights(object, ...)

## S3 method for class 'svyrep.design'
weights(object,
        type=c("replication","sampling","analysis"), ...)

## S3 method for class 'survey_fpc'
weights(object,final=TRUE,...)
```

Arguments

- **object**: Survey design object
- **type**: Type of weights: "analysis" combines sampling and replication weights.
- **final**: If FALSE return a data frame with sampling weights at each stage of sampling.
- **...**: Other arguments ignored
Value

vector or matrix of weights

See Also

svydesign, svrepdesign, as.fpc

Examples

data(scd)

scddes<-svydesign(data=scd, prob=-1, id=-ambulance, strata=-ESA,
                   nest=TRUE, fpc=rep(5,6))
repweights<-2*cbind(c(1,0,1,0,1,0), c(1,0,0,1,0,1), c(0,1,1,0,0,1), c(0,1,0,1,1,0))
scdrep<-svrepdesign(data=scd, type="BRR", repweights=repweights)

weights(scdrep)
weights(scdrep, type="sampling")
weights(scdrep, type="analysis")
weights(scddes)

---

with.svyimputationList

Analyse multiple imputations

Description

Performs a survey analysis on each of the designs in a svyimputationList object and returns a list of results suitable for MIcombine. The analysis may be specified as an expression or as a function.

Usage

```r
## S3 method for class 'svyimputationList'
with(data, expr, fun, ..., multicore=getOption("survey.multicore"))
## S3 method for class 'svyimputationList'
subset(x, subset,...)
```

Arguments

data, x
A svyimputationList object

expr
An expression giving a survey analysis

fun
A function taking a survey design object as its argument

...
for future expansion

multicore
Use multicore package to distribute imputed data sets over multiple processors?

subset
An logical expression specifying the subset
Value

A list of the results from applying the analysis to each design object.

See Also

MIcombine, in the mitools package

Examples

```r
library(mitools)
data.dir<-system.file("dta",package="mitools")
files.men<-list.files(data.dir,pattern="m\.\.dta\$",full=TRUE)
men<-imputationList(lapply(files.men, foreign::read.dta))
files.women<-list.files(data.dir,pattern="f\.\.dta\$",full=TRUE)
women<-imputationList(lapply(files.women, foreign::read.dta))
men<-update(men, sex=1)
women<-update(women,sex=0)
all<-rbind(men,women)
designs<-svydesign(id=~id, strata=~sex, data=all)
designs
results<-with(designs, svymean(~drkfre))
MIcombine(results)
summary(MIcombine(results))
```

withPV.survey.design  Analyse plausible values in surveys

Description

Repeats an analysis for each of a set of 'plausible values' in a survey data set, returning a list suitable for mitools::MIcombine. The default method works for both standard and replicate-weight designs but not for two-phase designs.

Usage

```r
## S3 method for class 'survey.design'
withPV(mapping, data, action, rewrite=TRUE, ...)
```

Arguments

mapping  A formula or list of formulas describing each variable in the analysis that has plausible values. The left-hand side of the formula is the name to use in the analysis; the right-hand side gives the names in the dataset.
data  A survey design object, as created by svydesign or svrepdesign
withReplicates

Compute variances by replicate weighting

Description

Given a function or expression computing a statistic based on sampling weights, withReplicates evaluates the statistic and produces a replicate-based estimate of variance.

Usage

```r
withReplicates design, theta, ..., return.replicates=FALSE
```

## S3 method for class 'svyrep.design'
```r
withReplicates(design, theta, rho = NULL, ..., scale.weights=FALSE, return.replicates=FALSE)
```

## S3 method for class 'svrepvar'
```r
```
withReplicates

withReplicates(design, theta, ..., return.replicates=FALSE)
## S3 method for class 'svrepstat'
withReplicates(design, theta, ..., return.replicates=FALSE)

Arguments

design A survey design with replicate weights (eg from svrepdesign) or a suitable object with replicate parameter estimates
theta A function or expression: see Details below
rho If design uses BRR weights, rho optionally specifies the parameter for Fay’s variance estimator.
... Other arguments to theta
scale.weights Divide the probability weights by their sum (can help with overflow problems)
return.replicates Return the replicate estimates as well as the variance?

Details

The method for svyrep.design objects evaluates a function or expression using the sampling weights and then each set of replicate weights. The method for svrepvar objects evaluates the function or expression on an estimated population covariance matrix and its replicates, to simplify multivariate statistics such as structural equation models.

For the svyrep.design method, if theta is a function its first argument will be a vector of weights and the second argument will be a data frame containing the variables from the design object. If it is an expression, the sampling weights will be available as the variable .weights. Variables in the design object will also be in scope. It is possible to use global variables in the expression, but unwise, as they may be masked by local variables inside withReplicates.

For the svrepvar method a function will get the covariance matrix as its first argument, and an expression will be evaluated with .replicate set to the variance matrix.

For the svrepstat method a function will get the point estimate, and an expression will be evaluated with .replicate set to each replicate. The method can only be used when the svrepstat object includes replicates.

Value

If return.replicates=FALSE, the weighted statistic, with the variance matrix as the "var" attribute. If return.replicates=TRUE, a list with elements theta for the usual return value and replicates for the replicates.

See Also

svrepdesign, as.svrepdesign, svrVar
Examples

```r
data(scd)
repweights<-2*cbind(c(1,0,1,0,1,0), c(1,0,1,0,1,0), c(0,1,0,1,0,1), c(0,1,0,1,0,1))
scdrep<-svrepdesign(data=scd, type="BRR", repweights=repweights)

a<-svyratio(~alive, ~arrests, design=scdrep)
print(a$ratio)
print(a$var)
withReplicates(scdrep, quote(sum(.weights*alive)/sum(.weights*arrests)))
withReplicates(scdrep, function(w, data)
  sum(w*data$alive)/sum(w*data$arrests))

data(api)
dclus1<-svydesign(id=~dnum, weights=~pw, data=apiclus1, fpc=~fpc)
rclus1<-as.svrepdesign(dclus1)
varmat<-svyvar(~api00+api99+ell+meals+hsg+mobility,rclus1,return.replicates=TRUE)
withReplicates(varmat, quote( factanal(covmat=.replicate, factors=2)$unique) )

data(nhanes)
nhanesdesign <- svydesign(id=~SDMVPSU, strata=~SDMVSTRA, weights=~WTMEC2YR, nest=TRUE, data=nhanes)
logistic <- svyglm(HI_CHOL~race+agecat+RIAGENDR, design=as.svrepdesign(nhanesdesign),
  family=quasibinomial, return.replicates=TRUE)
fitted<-predict(logistic, return.replicates=TRUE, type="response")
sensitivity<-function(pred,actual) mean(pred>0.1 & actual)/mean(actual)
withReplicates(fitted, sensitivity, actual=logistic$y)
```

## Not run:
library(quantreg)
data(api)
## one-stage cluster sample
dclus1<-svydesign(id=~dnum, weights=~pw, data=apiclus1, fpc=~fpc)
## convert to bootstrap
bclus1<-as.svrepdesign(dclus1,type="bootstrap", replicates=100)
## median regression
withReplicates(bclus1, quote(coef(rq(api00~api99, tau=0.5, weights=.weights))))
## End(Not run)

---

### yrbs

One variable from the Youth Risk Behaviors Survey, 2015.

### Description

Design information from the Youth Risk Behaviors Survey (YRBS), together with the single variable ‘Never/Rarely wore bike helmet’. Used as an analysis example by CDC.
Usage

data("yrbs")

Format

A data frame with 15624 observations on the following 4 variables.

weight  sampling weights
stratum  sampling strata
psu     primary sampling units
qn8  1=Yes, 2=No

Source


References


Examples

data(yrbs)

yrbs_design <- svydesign(id=~psu, weight=~weight, strata=~stratum, data=yrbs)
yrbs_design <- update(yrbs_design, qn8yes=2~qn8)

ci <- svyciprop(~qn8yes, yrbs_design, na.rm=TRUE, method="xlogit")

ci

## to print more digits: matches SUDAAN and SPSS exactly, per table 3 of reference
coeff(ci)
SE(ci)
attr(ci,"ci")
Index

*Topic **algebra**
  paley, 42
*Topic **category**
  svytable, 118
*Topic **datasets**
  api, 5
crowd, 25
election, 27
fpc, 30
hospital, 34
mu284, 38
nhanes, 39
scd, 52
yrbs, 132
*Topic **distribution**
  pchisqsum, 44
*Topic **hplot**
  barplot.svystat, 12
  svycdf, 68
  svycoplot, 73
  svyhist, 86
  svyplot, 99
  svyprcomp, 101
  svysmooth, 114
*Topic **htest**
  svyranktest, 107
  svytable, 118
  svytest, 121
*Topic **manip**
  as.fpc, 8
  as.svydesign2, 11
  calibrate, 17
  compressWeights, 23
dimnames.DBIsvydesign, 26
  estweights, 28
  ftable.svystat, 31
  nonresponse, 40
  postStratify, 45
  rake, 48
  subset.survey.design, 55
  svyby, 65
  svydesign, 79
  update.survey.design, 126
*Topic **models**
  SE, 53
  svymle, 95
*Topic **multivariate**
  svyfactanal, 82
  svyprcomp, 101
*Topic **optimize**
  svymle, 95
*Topic **regression**
  anova.svyglm, 3
  psrsq, 47
  regTermTest, 50
  svy.varcoef, 64
  svycoxph, 74
  svyglm, 83
  svypredmeans, 103
*Topic **survey**
  anova.svyglm, 3
  as.fpc, 8
  as.svrepdesign, 9
  as.svydesign2, 11
  barplot.svystat, 12
  bootweights, 13
  brrrWeights, 14
  calibrate, 17
  compressWeights, 23
  confint.svyglm, 24
  dimnames.DBIsvydesign, 26
  election, 27
  estweights, 28
  ftable.svystat, 31
  hadamard, 32
  HR, 35
  make.calfun, 35
  marginpred, 37
nonresponse, 40
open.DBI, 41
paley, 42
pchisqsum, 44
postStratify, 45
psrsq, 47
rake, 48
stratsample, 54
subset.survey.design, 55
surveyoptions, 56
surveysummary, 57
svrepdesign, 60
svyVar, 64
svy.varcoef, 64
svyby, 65
svycdf, 68
svyciprop, 69
svycontrast, 71
svycplot, 73
svycoplot, 74
svycpsr, 76
svycralpha, 78
svydesign, 79
svyfactanal, 82
svyglm, 83
svyhist, 86
svyivreg, 88
svykappa, 89
svykm, 89
svyloglin, 92
svylogrank, 93
svymle, 95
svyolr, 98
svyplot, 99
svyprcomp, 101
svypredmeans, 103
svyquantile, 104
svyranktest, 107
svyratio, 109
svyrecvar, 111
svysmooth, 114
svystandardize, 115
svysurvc, 117
svytetable, 118
svyttest, 121
trimWeights, 122
twophase, 123
update.survey.design, 126
weights.survey.design, 127
with.svyimputationList, 128
withPV.survey.design, 129
withReplicates, 130
*Topic survival
svycoxph, 74
svykm, 89
svylogrank, 93
svysurvreg, 117
*Topic univar
surveysummary, 57
svydesign, 79
svyquantile, 104
*Topic utilities
svyCprod, 76
.svycheck (as.svydesign2), 11
].nonresponse (nonresponse), 40
].repweights_compressed
 (compressWeights), 23
].survey.design (subset.survey.design), 55
].svyrep.design (svrepdesign), 60
].twophase (twophase), 123
AIC.svycoxph (svycoxph), 74
AIC.svyglm, 48, 75
AIC.svyglm (anova.svyglm), 3
anova, 3, 50, 51
anova.svyglm, 3
anova.svyloglin (svyloglin), 92
api, 5
apiclus1 (api), 5
apiclus2 (api), 5
apipop (api), 5
apisrs (api), 5
apistat (api), 5
approxfun, 105
as.fpc, 8, 112, 128
as.matrix.repweights (compressWeights), 23
as.matrix.repweights_compressed
 (compressWeights), 23
as.svrepdesign, 9, 14, 16, 23, 59, 62–64, 81,
 131
as.svydesign2, 11
as.vector.repweights_compressed
 (compressWeights), 23
barplot, 12
barplot.svrepstat (barplot.svystat), 12
barplot.svby (barplot.svystat), 12
barplot.svystat, 12
BIC.svyglm (anova.svyglm), 3
binom.test, 70
biplot, 102
biplot.prcomp, 102
biplot.svprcomp (svyprcomp), 101
bootstratum (bootweights), 13
bootweights, 10, 13, 62
brrweights, 10, 14, 33, 62–64
bxp, 87
cal.linear (make.calfun), 35
cal.logit (make.calfun), 35
cal.raking (make.calfun), 35
cal.sinh (make.calfun), 35
cal_names (calibrate), 17
calibrate, 17, 29, 35–37, 46, 47, 49, 85, 110, 112, 122, 124
close, 62, 81
close.BDISvydesign (open.BDISvydesign), 41
coeff, 50
coeff.svrepstat (surveysummary), 57
coeff.svby (svby), 65
coeff.svyglm (svyglm), 83
coeff.svyloglin (svyloglin), 92
coeff.svymle (svymle), 95
coeff.svyratio (svyratio), 109
coeff.svystat (surveysummary), 57
compressWeights, 23, 47, 49
confint, 24
confint.svrepstat (surveysummary), 57
confint.svby (svby), 65
confint.svyglm, 24
confint.svykm (svykm), 89
confint.svratio (svyratio), 109
confint.svystat, 70
confint.svystat (surveysummary), 57
confint.svytest (svytest), 121
contrasts, 51
coxph, 74, 75
crowd, 25
cv (surveysummary), 57
deff (surveysummary), 57
deff.svby (svby), 65
defg, 59, 82, 83, 105
defg (svytable), 118
deriv, 72
dim.BDISvydesign
(dimnames.BDISvydesign), 26
dim.repwights_compressed
(compressWeights), 23
dim.survey.design
(dimnames.BDISvydesign), 26
dim.svyimputationList
(dimnames.BDISvydesign), 26
dim.svyrep.design
(dimnames.BDISvydesign), 26
dim.twophase (dimnames.BDISvydesign), 26
dimnames.BDISvydesign, 26
dimnames.repwights_compressed
(compressWeights), 23
dimnames.survey.design
(dimnames.BDISvydesign), 26
dimnames.svyimputationList
(dimnames.BDISvydesign), 26
dimnames.svyrep.design
(dimnames.BDISvydesign), 26
dimnames.twophase
(dimnames.BDISvydesign), 26
dnorm, 96
dotchart (barplot.svystat), 12
election, 27, 35, 81
election_insamp (election), 27
election_jointHR (election), 27
election_jointprob (election), 27
election_pps (election), 27
estWeights, 19, 124
estWeights (estweights), 28
estweights, 28
extractAIC.svycoxph (svycoxph), 74
factanal, 82, 83
fpc, 30
ftable, 32
ftable.svrepstat (ftable.svystat), 31
ftable.svby, 67
ftable.svby (ftable.svystat), 31
ftable.svystat, 31, 58, 59, 67, 120
glm, 65, 85
grake (calibrate), 17
hadamard, 15, 16, 32, 42, 43
INDEX

hist, 87
hospital, 34
HR, 35, 79

image, 61
image.svyrep.design(svrepdesign), 60
interaction, 58
is.hadamard(paley), 42
ivreg, 88

jk1weights, 62, 64
jk1weights (brrweights), 14
jkweights, 23, 62, 64
jkweights (brrweights), 14
joinCells (nonresponse), 40

lines.svykm(svykm), 89
lines.svysmooth(svysmooth), 114

make.calfun, 18, 20, 35
make.formula(surveysummary), 57
make.panel.svysmooth(svysmooth), 114
marginpred, 37, 124
model.frame.svyrep.design
(svrepdesign), 60
model.frame.twophase(twophase), 123
mrbweights, 10
mrbweights (bootweights), 13
mu284, 38
multistage(svyrecvar), 111
multistage.phase1(twophase), 123

na.exclude.survey.design(svydesign), 79
na.exclude.twophase(twophase), 123
na.fail.survey.design(svydesign), 79
na.fail.twophase(twophase), 123
na.omit.survey.design(svydesign), 79
na.omit.twophase(twophase), 123
neighbours (nonresponse), 40

nhanes, 39
nlm, 96
nonresponse, 40

onestage(svyCprod), 76
onestage.phase1(twophase), 123
onestrat(svyCprod), 76
onestrat.phase1(twophase), 123
open, 62, 81
open.DBIsvydesign, 41
optim, 95, 96

paley, 33, 42
panel.smooth, 115
par, 90
pchisq, 45
pchisqsum, 5, 44, 51, 92, 93, 119
pFsum(pchisqsum), 44
plot, 100
plot.lm, 115
plot.stepfun, 68, 69
plot.svycdf(svycdf), 68
plot.svykm(svykm), 89
plot.svykm(svykm), 89
plot.svysmooth(svysmooth), 114
postStratify, 20, 29, 45, 49, 77, 78, 112, 116
ppsmat, 79, 80
ppsmat (HR), 35
prcomp, 102
predict.coxph, 75
predict.svrepglm(svylm), 83
predict.svcoph, 37, 91
predict.svcoph(svycoph), 74
predict.svyglm(svylm), 83
predict.svyratio(svyratio), 109
predict.svyratio(svyratio), 109
print.anova(svyloglin), 92
print.nonresponse (nonresponse), 40
print.nonresponseSubset (nonresponse), 40
print.regTermTest (regTermTest), 50
print.summary.svyrep.design
(svrepdesign), 60
print.summary.svytable(svytable), 118
print.summary.twophase(twophase), 123
print.svycdf(svycdf), 68
print.svymle(svymle), 95
print.svyquantile(svyquantile), 104
print.svyratio(svyratio), 109
print.svyratio(svyratio), 109
print.svyrep.design(svrepdesign), 60
print.svysmooth(svysmooth), 114
print.twophase(twophase), 123
psrsq, 47

qr, 119
quantile, 90
quantile.svykm(svykm), 89

rake, 18, 20, 45, 47, 48
regTermTest, 4, 5, 50, 72, 75, 85, 99, 120
residuals.svrepglm (svyglm), 83
residuals.svyglm (svyglm), 83

sample, 54
scd, 52
SE, 53, 59
SE.svyby (svyby), 65
SE.svyquantile (svyquantile), 104
SE.svyratio (svyratio), 109
sparseCells (nonresponse), 40
stepfun, 69
strata, 94
stratsample, 54
subbootweights, 10
subbootweights (bootweights), 13
subset.survey.design, 55, 80, 81
subset.svyimputationList
(with.svyimputationList), 128
subset.svyrep.design
(subset.survey.design), 55
subset.twophase (twophase), 123
summary.svrepglm (svyglm), 83
summary.svreptable (svytable), 118
summary.svyglm (svyglm), 83
summary.svymle (svymle), 95
summary.svyrep.design (svrepdesign), 60
summary.svytable (svytable), 118
summary.twophase (twophase), 123
survey.adjust.domain.lonely
(surveyoptions), 56
survey.drop.replicates (surveyoptions), 56
survey.lonely.psu (surveyoptions), 56
survey.multicore (surveyoptions), 56
survey.replicates.mse (surveyoptions), 56
survey.ultimate.cluster
(surveyoptions), 56
survey.want.obsolete (surveyoptions), 56
surveyoptions, 16, 56, 78
surveysummary, 57
svrepdesign, 10, 59, 60, 64, 84, 127, 128, 131
svreptable (svytable), 118
svrVar, 16, 64, 113, 131
svy.varcoef, 64
svyboxplot (svyhist), 86
svyby, 65, 105, 116, 120
svycdf, 68
svychisq, 92, 93
svychisq (svytable), 118
svyciprop, 59, 69, 106
svycontrast, 59, 66, 71, 89
svycoplot, 73
svycoxph, 74, 94
svyCprod, 65, 76, 80, 81, 113
svyralpha, 78
svydesign, 9–11, 35, 42, 55, 59, 63, 65, 78, 79, 84, 96, 110, 124, 127, 128
svyfactanal, 82
svyglm, 3, 4, 24, 47, 58, 64, 65, 83, 93, 96, 99, 103, 104, 121
svyhist, 69, 86, 115
svyivreg, 88
svykappa, 89
svykappa, 75, 89, 94, 106
svyloglin, 92, 120
svylogrank, 93, 108
svymean, 70, 92, 103, 109, 110, 119–121
svymean (surveysummary), 57
svymle, 95
svyolr, 98
svyplot, 73, 87, 99
svypcomp, 101
svypredmeans, 37, 103
svyquantile, 59, 68, 69, 104
svyranktest, 107
svyratio, 58, 109
svyrecvar, 9, 11, 56, 76, 78, 81, 111, 124
svyrecvar (phase1 (twophase), 123
svysmooth, 114
svystandardize, 115
svysurvreg, 117
svyt, 12, 67, 101, 118
svytotal, 119, 120
svytotal (surveysummary), 57
svytest, 59, 85, 108, 121
svyvar, 82
svyvar (surveysummary), 57
symbols, 101
t.test, 121
table, 46
termplot, 115
termplot, 115
transform, 127
trimWeights, 19, 20, 122
twophase, 20, 29, 123, 127
twophase2var (twophase), 123
twophasevar (twophase), 123

unwtd.count (svyby), 65
update.DBIsvydesign, 26
update.DBIsvydesign
(update.survey.design), 126
update.survey.design, 67, 81, 126
update.svyloglin (svyloglin), 92
update.svyrep.design
(update.survey.design), 126
update.twophase (update.survey.design), 126

vcov, 50, 51, 53, 54
vcov.svrepsat (surveysummary), 57
vcov.svyglm (svyglm), 83
vcov.svymle (svymle), 95
vcov.svystat (surveysummary), 57

weights.nonresponse (nonresponse), 40
weights.survey.design, 127
weights.survey_fpc
(weights.survey.design), 127
weights.svyrep.design
(weights.survey.design), 127
with.svyimputationList, 26, 81, 128, 130
withPV.survey.design, 129
withReplicates, 75, 130

xtabs, 46
xyplot, 73

yrbs, 70, 132