The Survey Statistician, 2022, Vol. 86, 32–43.

Book and Software Review

M. S. Diallo. Samplics: A comprehensive library for survey sampling in Python

Mamadou S. Diallo[1]

[1]The Saudi Center for Opinion Polling (SCOP), Saudi Arabia, msdiallo@samplics.org

Abstract

Survey sampling is one of the main tools used by public and private organizations of all sizes to produce statistics to guide decision-making. For example, governments regularly use large national household and non-household surveys to inform policy in numerous sectors. Similarly, opinion polling and market research surveys inform corporations and other entities on populations' views and opinions on issues and products.

Python has become a leading tool for data science and machine learning projects. Yet, survey statisticians did not have any comprehensive library in Python for designing or analyzing complex survey data. With the development of *samplics*, Python users no longer must learn another software to design or analyze complex survey samples. The library *samplics* classes and functions provide a large coverage of survey sampling topics from sample size calculation, sample selection, weight adjustments, estimation, tabulation, t-test to small area estimation. This paper discusses some of the APIs of *samplics*.

Keywords: survey, sampling, sample size, small area estimation, Python.

1 Introduction

Python is a free and open-source software; it has become one of the most popular software during the last decade. Most of its popularity is due to the explosion of data science and its applications. Python is currently one of the software of choice for machine learning and data science due to the availability of comprehensive and user friendly libraries such as scipy, numpy, matplotlib, pandas, scikit-learn, statsmodels, keras, tensorflow, and pytorch. However, until the development of *samplics*, there was no library for survey sampling techniques, see Lohr (2022).

samplics is a Python library for survey sampling techniques. The package is comprehensive and is designed to assist the survey statistician from the conception with sample size calculation to the estimation of population parameters. The main modules of the *samplics* library are sample size calculation, sample selection, weighting, population parameters estimation, tabulation and hypothesis testing, and small area estimation. A Python user no longer needs to move to the R Software or other solutions to design or analyse complex survey samples.

In this paper, we present the main APIs of the *samplics* library. It uses Python versions 3.7.x or newer and the following packages: numpy, pandas, scpy, and statsmodels. To install it, use: pip install samplics. The current version of the library is 0.3.35 (May, 2022), and its manual can be downloaded at https://samplics.readthedocs.io/en/latest/.

2 Sample size calculation

During the design of the survey, investigators collaborate with statisticians to calculate the minimum required sample sizes, see Chow et al. (2018) and Ryan (2013) for a comprehensive review of sample size calculation methods. Often at the design phase, many variables are of interest. For the sample size calculation, the investigators must reduce the number to ideally a single variable or a handful. *samplics* provides the class *SampleSize* for calculating sample size to estimate proportions, means, and totals. Its argument *parameter* can take the values "proportion", "mean" or "total", while the argument *method* takes the values "wald" or "fleiss"; for example:

SampleSize(parameter = "proportion", method = "wald", stratification = False)

To calculate the sample size, we need to provide the expected value through the argument *target* and the desired precision *half_ci* in *SampleSize.calculate*. If we are estimating a mean or a total then the standard deviation, *sigma*, is required. We have:

SampleSize.calculate(target, half_ci, sigma = None, deff = 1.0, resp_rate = 1.0,
number_strata = None, pop_size = None, alpha = 0.05)

We can use this class to calculate the sample size for simple random sampling with replacement when we estimate a proportion:

$$n_0 = \left(\frac{z_{\alpha/2}}{e}\right)^2 p(1-p),$$

where $z_{\alpha/2}$ is the quantile of order $1-\alpha/2$ of the N(0,1) distribution, p is the expected proportion, and e is the margin of error. For example, let's say we want to calculate the sample size to estimate a proportion $p = 0.5$ with a margin of error $e = 0.03$, and for $\alpha = 0.05$. We could use the following code snippet:

```
import samplics
from samplics.sampling import SampleSize
size_prop = SampleSize(parameter="proportion")
size_prop.calculate(target=0.5, half_ci=0.03)
size_samp_size
1068
```

The second line of the code above creates the object *size_prop*. In the third line, we call the method *calculate()* to compute the sample size and update the object *size_prop*. To show the content of the object *size_prop*, we can print the members using *size_prop.__dict__*.

samplics can also calculate the required sample size to conduct hypothesis testing. There are several *samplics* classes for calculating sample size for testing proportions or means in the situation of one or two samples.

The class *SampleSizeMeanOneSample* calculates the minimum required sample size for testing mean with one sample. Let's assume we have one sample and we are interested in the following hypotheses $H_0 : \mu = \mu_0 \quad versus \quad H_a : \mu \neq \mu_0$. The equation for the sample size needed to achieve power $1-\beta$ is $n = \frac{(z_{\alpha/2}+z_\beta)^2\sigma^2}{\epsilon^2}$, where ϵ is the difference $\mu - \mu_0$. For example, let's assume that we want to calculate sample size required to test the difference before and after a treatment. We have that the average before treatment is 1.5 and the same average after treatment is 2. The standard

deviation is 1, and we assume $1 - \beta = 0.8$. We could calculate the sample size by using the following code snippet for simple random sampling with replacement:

```
from samplics.sampling import SampleSizeMeanOneSample
mean_equality = SampleSizeMeanOneSample()
mean_equality.calculate(mean_0=1.5, mean_1=2, sigma=1)
mean_equality.samp_size
32
```

In a stratified design, the population is divided into H partitions or strata. The sample is selected independently from each stratum. The above *samplics* APIs integrates the notion of stratification. When instantiating the objects, we can indicate that it's a stratified design using *stratification = True*. The parameters should then be provided by stratum using Python dictionaries. For example, *mean_0 = {"North": 1.50, "South": 1.65, "West": 1.55, "East": 1.45}* where North, South, West, and East are the strata.

If more convenient, we can use the method *to_dataframe()* to convert the output data dictionary to a Pandas DataFrame, see McKinney (2010) to learn more about Pandas. Similarly, we can use *SampleSizePropOneSample* for testing proportions with one sample. In the context of two samples, we can use *SampleSizeMeanTwoSample* and *SampleSizePropTwoSample*.

3 Sample selection

The class *SampleSelection* implements several popular random selection methods such as simple random sampling (SRS), systematic (SYS), several probability proportional to size (PPS) methods. The available PPS algorithms for selecting samples with unequal probabilities of selection are systematic (*method="pps-sys"*), Brewer (*method="pps-brewer"*), Hanurav-Vijayan (*method="pps-hv"*), Murphy (*method="pps-murphy"*), and Rao-Sampford (*method="pps-rs"*) methods. Let's assume that we have a population of 100 units and we want to select 10 using the SRS method:

```
from samplics.sampling import SampleSelection
srs_sampling = SampleSelection(method="srs")
srs_sample, srs_hits, srs_probs = srs_sampling.select(samp_unit=range(1, 101),
samp_size=10)
```

As shown in the above code snippet, the method *select()* returns a tuple of three numpy arrays, see Harris et al. (2020). The first array provides the selection status of each unit in the population, the second array provides the probabilities of selection, and the third array gives the number of hits/times a unit was selected. If needed, the user may set *to_dataframe=True* to convert the output data to a pandas DataFrame from the tuple of three arrays. The resulting sample is now a Pandas DataFrame with its first 5 observations shown below.

	_samp_unit	_mos	_sample	_hits	_probs
0	1	1.0	False	0	0.1
1	2	1.0	False	0	0.1
2	3	1.0	False	0	0.1
3	4	1.0	False	0	0.1
4	5	1.0	False	0	0.1

The code above returns the entire population with the variable *_sample* indicating the selected units. We can use *sample_only=True* to subset the returned data to only contain the sample. To illustrate the PPS sample selection, we use the code below to randomly generate sizes (using the Unif(0,1) distribution) associated with each of the 100 units in our population:

```
import random
mos = []
for _ in range(100):
mos.append(round(100 * random.random(), 0))
```

Now let's select a sample of 2 units using the PPS Brewer method without replacement:

```
ss_brewer = SampleSelection(method="pps-brewer", with_replacement=False)
ss_brewer_sample = ss_brewer.select( samp_unit=range(1,101), samp_size=2, mos=mos,
to_dataframe=True, sample_only=True)
ss_brewer_sample
```

	_samp_unit	_mos	_sample	_hits	_probs
0	46	67.0	1	1	0.026677
1	50	93.0	1	1	0.037030

As discussed previously, for stratified designs, we provide the information using Python dictionaries where the keys are the strata names and the values are the sample sizes. Then, we provide the stratification variable to the method *select()* using the argument *stratum*.

4 Weighting

The *samplics* module *weighting* provides the algorithms for adjusting the sample weight for non-response, post-stratification, and calibration. The main class is *SampleWeight* and the different type of adjustments are conducted using its methods *adjust()*, *poststratify()*, and *calibrate()*.

4.1 Design weight

The design weight calculation and subsequent weight adjustments are key steps to ensuring the generalization of the sample results to the target population. The initial design weight, w_i, is obtained as the reciprocal of the probability of inclusion π_i, for unit i in the population, $w_i = \frac{1}{\pi_i}$.

samplics has a module *dataset* which provides curated datasets for running the examples. With the code below, we use the dataset module to load two datasets representing primary sampling units (PSUs) and Secondary Sampling Units (SSUs) samples:

```
from samplics.datasets import load_psu_sample, load_ssu_sample
psu_sample_dict = load_psu_sample()
psu_sample = psu_sample_dict["data"]
ssu_sample_dict = load_ssu_sample()
ssu_sample = ssu_sample_dict["data"]
```

We combine the two datasets to form the final sample data and we calculate the inclusion probability as the product of the two stage probabilities:

```
full_sample = pd.merge(
left=psu_sample[["cluster", "region", "psu_prob"]],
right=ssu_sample[["cluster", "household", "ssu_prob"]],
on="cluster")
```

Hence, the design weight follows as the reciprocal of the inclusion probability.

```
full_sample["inclusion_prob"] = full_sample["psu_prob"] * full_sample["ssu_prob"]
full_sample["design_weight"] = 1 / full_sample["inclusion_prob"]
full_sample.head()
```

	cluster	region	psu_prob	household	ssu_prob	inclusion_prob	design_weight
0	7	North	0.187726	72	0.115385	0.021661	46.166667
1	7	North	0.187726	73	0.115385	0.021661	46.166667
2	7	North	0.187726	75	0.115385	0.021661	46.166667
3	7	North	0.187726	715	0.115385	0.021661	46.166667
4	7	North	0.187726	722	0.115385	0.021661	46.166667

4.2 Non-response adjustment

For the purpose of illustrating non-response adjustments, we add non-respondent households into our example. That is, we simulate the non-response status and store it in the variable *response_status* which has four possible values: *ineligible* which indicates that the sampling unit is not eligible for the survey, *respondent* which indicates that the sampling unit responded to the survey, *non-respondent* which indicates that the sampling unit did not respond to the survey, and *unknown* means that we are not able to infer the status of the sampling unit i.e. we do not know whether the sampling unit is eligible or not for the survey.

```
np.random.seed(7)
full_sample["response_status"] = np.random.choice( ["ineligible", "respondent",
"non-respondent", "unknown"], size=full_sample.shape[0], p=(0.10, 0.70, 0.15, 0.05) )
full_sample[["cluster", "region", "design_weight", "response_status"]].head(5)
```

	cluster	region	design_weight	response_status
0	7	North	46.166667	ineligible.
1	7	North	46.166667	respondent.
2	7	North	46.166667	respondent.
3	7	North	46.166667	respondent.
4	7	North	46.166667	unknown.

In general, the sample weights are adjusted by redistributing the sample weights of all eligible units for which there is no sufficient response (nonrespondents) to the sampling units that sufficiently responded to the survey (respondents). This adjustment is done within adjustment/response classes or domains. Note that the determination of the response classes is outside of the scope of this module.

The method *adjust()* has a boolean argument *unknown_to_inelig* which controls how the sample weights of the unknown are redistributed. By default, *adjust()* redistributes the sample weights of the units with unknown eligibility to the ineligibles (*unknown_to_inelig=True*). If we do not wish to redistribute the sample weights of the unknowns to the ineligibles, we set the flag to *False*.

```
status_mapping = {"in":  "ineligible", "rr":  "respondent", "nr":  "non-respondent",
"uk":  "unknown" }

from samplics.weighting import SampleWeight
full_sample["nr_weight"] = SampleWeight().adjust(
samp_weight=full_sample["design_weight"],
adjust_class=full_sample[["region", "cluster"]],
resp_status=full_sample["response_status"],
resp_dict=status_mapping)
full_sample[["cluster", "region", "design_weight", "response_status",
"nr_weight"]].drop_duplicates().head(10)
```

	cluster	region	design_weight	response_status	nr_weight
0	7	North	46.166667	ineligible	49.464286
1	7	North	46.166667	respondent	54.410714
4	7	North	46.166667	unknown	0.000000
11	7	North	46.166667	non-respondent	0.000000
15	10	North	50.783333	non-respondent	0.000000
16	10	North	50.783333	respondent	70.733929
19	10	North	50.783333	ineligible	54.410714
21	10	North	50.783333	unknown	0.000000
30	16	South	62.149123	respondent	66.588346
35	16	South	62.149123	non-respondent	0.000000

Important. The default call of *adjust()* expects the response status variable to have values of "in", "rr", "nr", or "uk" where "in" means ineligible, "rr" means respondent, "nr" means non-respondent, and "uk" means unknown eligibility.

In the call above, if we omit the argument *resp_dict*, then the code would fail with an assertion error message. The current error message is the following: "The response status must only contains values in ('in', 'rr', 'nr', 'uk') or the mapping should be provided using *response_dict* parameter". For the call to run without specifying *resp_dict*, it is necessary that the response status takes only values in the standard codes i.e. ("in", "rr", "nr", "uk").

4.3 Post-stratification

Post-stratification is useful to compensate for under-representation of the sample or to correct for nonsampling error. Post-stratification classes can be formed using variables beyond the ones involved in the sampling design. For example, socio-economic variables such as age group, gender, race and education are often used to form post-stratification classes/cells.

Let's assume that we have a reliable external source e.g. a recent census that provides the number of households by region. The external source has the following control data: 3700 households for East, 1500 for North, 2800 for South and 6500 for West. We use the method *poststratify()* to ensure that the post-stratified sample weights (*ps_weight*) sum to the know control totals by region. Note that the control totals are provided using the Python dictionary *census_households*.

```
census_households = {"East": 3700, "North": 1500, "South": 2800, "West": 6500}
full_sample["ps_weight"] = SampleWeight().poststratify(samp_weight
=full_sample["nr_weight"], control=census_households, domain=full_sample["region"])
full_sample.head(7)
```

	cluster	region	household	design_weight	response_status	nr_weight	ps_weight
0	7	North	72	46.166667	ineligible	49.464286	51.020408
1	7	North	73	46.166667	respondent	54.410714	56.122449
2	7	North	75	46.166667	respondent	54.410714	56.122449
3	7	North	715	46.166667	respondent	54.410714	56.122449
4	7	North	722	46.166667	unknown	0.000000	0.000000
5	7	North	724	46.166667	respondent	54.410714	56.122449
6	7	North	755	46.166667	respondent	54.410714	56.122449

In some surveys, there is interest in keeping relative distribution of strata to some known distribution. For example, WHO EPI vaccination surveys, World Health Organization (2018), often poststratify sample weights to ensure that relative sizes of strata reflect official statistics e.g. census data. Assume that according to census data that East contains 25% of the households, North contains 10%, South contains 20% and West contains 45%. We can post-stratify using the snippet of code below.

```
known_ratios = {"East": 0.25, "North": 0.10, "South": 0.20, "West": 0.45}
full_sample["ps_weight2"] = SampleWeight().poststratify(samp_weight
=full_sample["nr_weight"], factor=known_ratios, domain=full_sample["region"])
```

4.4 Calibration weight

Calibration is a more general concept for adjusting sample weights to sum to known constants; see Deville & Särndal (1992). In *samplics*, we implemented the generalized regression (GREG) class of calibration. Assume that we have $\hat{\mathbf{Y}} = \sum_{i \in s} w_i y_i$ and population totals $\mathbf{X} = (\mathbf{X}_1, ..., \mathbf{X}_p)^T$ are available. Working under the model $Y_i | \mathbf{x}_i = \mathbf{x}_i^T \boldsymbol{\beta} + \epsilon_i$, where $\boldsymbol{\beta}$ is the vector of parameters, and ε_i are independent error terms, for any unit i in the population, the GREG estimator of the population total is $\hat{\mathbf{Y}}_{GR} = \hat{\mathbf{Y}} + (\mathbf{X} - \hat{\mathbf{X}})^T \hat{\mathbf{B}}$ where $\hat{\mathbf{B}}$ is the weighted least squares estimate of β and $\hat{\mathbf{X}}$ is the Horvitz-Thompson estimate of $\mathbf{X}$. The essence of the GREG approach consists of, under the regression model, finding the adjusted weights w_i^* that are the closest to w_i, by minimizing the chi-square distance between them.

Let us simulate three auxiliary variables that are *education*, *poverty* and *under_five* (number of children under five in the household) and assume that we have a total number of under five children of 6300 in the East, 4000 in the North, 6500 in the South and 14000 in the West. Similarly, we have the following number of households per poverty status (*Yes*: in poverty / *No*: not in poverty) and education level (*Low*: less than secondary, *medium*: secondary completed, and *high*: more than secondary):

```
np.random.seed(150)
full_sample["education"] = np.random.choice(
("Low", "Medium", "High"), size=150, p=(0.40, 0.50, 0.10))
full_sample["poverty"] = np.random.choice((0, 1), size=150, p=(0.70, 0.30))
full_sample["under_five"] = np.random.choice((0, 1, 2, 3, 4, 5), size=150,
p=(0.05, 0.35, 0.25, 0.20, 0.10, 0.05))
full_sample[["cluster", "region", "household", "nr_weight", "education", "poverty",
"under_five"]].head()
```

	cluster	region	household	nr_weight	education	poverty	under_five
0	7	North	72	49.464286	High	1	1
1	7	North	73	54.410714	Low	0	3
2	7	North	75	54.410714	Medium	0	2
3	7	North	715	54.410714	Medium	1	2
4	7	North	722	0.000000	Medium	0	2

We now will calibrate the nonresponse weight (*nr_weight*) to ensure that the estimated number of households in poverty is equal to 4,700 and the estimated total number of children under five is 30,800.

The class *SampleWeight* uses the method *calibrate()* to adjust the weight using the GREG approach. The control values must be stored in a Python dictionary i.e. *totals* = {*"poverty": 4700, "under_five": 30800*}. In this case, we have two numerical variables: *poverty* with values in 0, 1 and *under_five* with values in 0, 1, 2, 3, 4, 5. The argument *aux_vars* represents the matrix of covariates.

```
totals = {"poverty":  4700, "under_five":  30800}
full_sample["calib_weight"] = SampleWeight().calibrate(samp_Weight =
full_sample["nr_weight"],
aux_vars = full_sample[["poverty", "under_five"]], control = totals)
full_sample[["cluster", "region", "household", "nr_weight", "calib_weight"]].head()
```

	cluster	region	household	nr_weight	calib_weight
0	7	North	72	49.464286	50.432441
1	7	North	73	54.410714	57.233887
2	7	North	75	54.410714	56.292829
3	7	North	715	54.410714	56.416743
4	7	North	722	0.000000	0.000000

If we want to control by domain then we can do so using the argument *domain* from *calibrate()*. First we update the Python dictionary holding the control values for each domain. Note that the dictionary is now a nested dictionary where the higher level keys hold the domain values i.e. East, North, South and West. Then the higher level values of the dictionary are the dictionaries providing mapping for the auxiliary variables and the corresponding control values.

```
totals_by_domain = {
"East":  "poverty":  1200, "under_five":  6300,
"North":  "poverty":  200, "under_five":  4000,
"South":  "poverty":  1100, "under_five":  6500,
"West":  "poverty":  2200, "under_five":  14000,
}
```

```
full_sample["calib_weight_d"] = SampleWeight().calibrate(
samp_weight = full_sample["nr_weight"],
aux_vars = full_sample[["poverty", "under_five"]],
control = totals_by_domain,
domain = full_sample["region"])
```

```
full_sample[["cluster", "region", "household", "nr_weight", "calib_weight",
"calib_weight_d"]].head()
```

	cluster	region	household	nr_weight	calib_weight	calib_weight_d
0	7	North	72	49.464286	50.432441	40.892864
1	7	North	73	54.410714	57.233887	61.852139
2	7	North	75	54.410714	56.292829	59.371664
3	7	North	715	54.410714	56.416743	47.462625
4	7	North	722	0.000000	0.000000	0.000000

Note that the GREG domain estimates above do not have the additive property. That is the GREG domain estimates do not sum to the overall GREG estimate. To enforce the additive property of the GREG estimates, we must use *additive=True* when calling *calibrate()*.

4.5 Replicate weights

We can use *samplics* to create replicate weights. It is best to create the replicate weights from the design weights and apply the weight adjustments to each replicate. The API for creating the replicate weights is:

```
ReplicateWeight(method, stratification=True, number_reps = 500, fay_coef = 0.0,
random_seed = None)
```

```
ReplicateWeight.replicate(samp_weight, psu, stratum = None, rep_coefs = False,
rep_prefix = None, psu_varname = "_psu", str_varname)
```

The user provides the sample weight to replicate with the sampling design information, namely the PSU and stratification as applicable. In the case of the *Fay*'s method, Dippo et al. (1984) and Fay (1989) , the user may provide *fay_coef*, the coefficient to adjust the original weights.

5 Population parameters estimation

The *samplics* estimation module has two main parts: linearization (Taylor series) and replication based estimations.

5.1 Linearization (Taylor series)

The API for the Taylor-based estimation is as shown below and the argument *parameter* may take the value *mean*, *total*, *proportion*, or *ratio*. The main method of this class is *estimate()* which calculates the point estimates, the uncertainty measures, and other related statistics:

```
TaylorEstimator(parameter, alpha = 0.05, random_seed = None, ciprop_method = "logit")
```

```
TaylorEstimator.estimate(y, samp_weight = None, x = None, stratum = None,
psu = None, ssu = None, domain = None, by = None, fpc = 1.0, deff = False,
coef_variation = False, as_factor = False, remove_nan = False)
```

Some of the parameters of the method *estimate()* are : y is the variable of interest, $samp\_weight$ is the final sampling weight, x is the auxiliary variable in the case of the ratio estimation, $domain$ is the variable for the domain estimation (domain variable), and by is the variable to split the data; split the data then produce estimates for each partition (not domain estimation).

We are going to download the NHANES dataset and use it to estimate the average level of zinc:

```
from samplics.datasets import load_nhanes2
nhanes2_dict = load_nhanes2()
nhanes2 = nhanes2_dict["data"]
```

Now we estimate the average level of zinc:

```
zinc_mean_str = TaylorEstimator("mean")
zinc_mean_str.estimate(y=nhanes2["zinc"], samp_weight=nhanes2["finalwgt"],
stratum=nhanes2["stratid"], psu=nhanes2["psuid"], remove_nan=True)
print(zinc_mean_str)
```

SAMPLICS - Estimation of Mean
Number of strata: 31
Number of psus: 62
Degree of freedom: 31

MEAN	SE	LCI	UCI	CV
87.182067	0.494483	86.173563	88.190571	0.005672

The results of the estimation are stored in the dictionary *zinc_mean_str*. The users can covert the main estimation information into a *pandas DataFrame* by using the method *to_dataframe()*. The method *to_dataframe()* is more useful for domain estimation by producing a table where which row is a level of the domain of interest.

5.2 Replication

The class *replicateEstimator* provides the algorithms for the replication-based estimation. The argument *method* takes the value *brr* for the Balanced Random Replication (BRR) approach, McCarthy (1966); *bootstrap*, Rao & Wu (1992); and *jackknife*, Krewski & Rao (1981). Note that the *Fay*'s method is a generalization of the BBR method. Instead of simply taking half-size samples, we use the full sample every time but with unequal weighting: *fay_coef* for units outside the half-sample and 2 - *fay_coef* for units inside it (BRR is the case *fay_coef=0 or* None). *parameter* takes the same values as in the linearization case.

Let's load the NHANES again and use it estimate the ratio of weight over height. We want to use the BRR replicates weights.

```
from samplics.datasets import load_nhanes2brr
nhanes2brr_dict = load_nhanes2brr()
nhanes2brr = nhanes2brr_dict["data"]
```

Now we are going to estimate the ratio using the BRR replicates weights from the dataset.

```
from samplics.estimation import ReplicateEstimator
brr = ReplicateEstimator(method="brr", parameter="ratio")
ratio_wgt_hgt = brr.estimate(y=nhanes2brr["weight"],
samp_weight=nhanes2brr["finalwgt"],
x=nhanes2brr["height"], rep_weights=nhanes2brr.loc[:,
"brr_1":"brr_32"], remove_nan=True)
print(ratio_wgt_hgt)
```

SAMPLICS - Estimation of Ratio

Number of strata: None
Number of psus: None
Degree of freedom: 16

RATIO	SE	LCI	UCI	CV
0.426082	0.00273	0.420295	0.43187	0.006407

6 Categorical data

With *samplics*, users can analyze categorical data by producing tabulations and conducting t-tests.

There are two main *samplics* classes for tabulation i.e. *Tabulation* for one-way tables and *crossTabulation* for two-way tables. From the NHANES dataset downloaeded using load_nhanes2(), let's tabulate the variables *race* and *diabetes*, we can use the *tabulation* class as follows:

```
diabetes_nhanes = Tabulation("proportion")
diabetes_nhanes.tabulate(vars=nhanes2[["race", "diabetes"]], samp_weight=weight,
stratum=stratum, psu=psu, remove_nan=True)
print(diabetes_nhanes)
```

Tabulation of race
Number of strata: 31
Number of PSUs: 62
Number of observations: 10335
Degrees of freedom: 31.00

variable	category	proportion	stderror	lower_ci	upper_ci
race	1.0	0.879016	0.016722	0.840568	0.909194
race	2.0	0.095615	0.012778	0.072541	0.125039
race	3.0	0.025369	0.010554	0.010781	0.058528
diabetes	0.0	0.965715	0.001820	0.961803	0.969238
diabetes	1.0	0.034285	0.001820	0.030762	0.038197

In the case of two-way tabulation, we use the *crossTabulation* class. The APIs for *crossTabulation* is very similar to *tabulation*. Let's crosstabulate *race* by *diabetes*. We can use the *crossTabulation* class as follows:

```
crosstab_nhanes = CrossTabulation("proportion")
crosstab_nhanes.tabulate(vars=nhanes2[["race", "diabetes"]], samp_weight=weight,
stratum=stratum, psu=psu, remove_nan=True)
print(crosstab_nhanes)
```

Cross-tabulation of race and diabetes
Number of strata: 31
Number of PSUs: 62
Number of observations: 10335
Degrees of freedom: 31.00

race	diabetes	proportion	stderror	lower_ci	upper_ci
1	0.0	0.850866	0.015850	0.815577	0.880392
1	1.0	0.028123	0.001938	0.024430	0.032357
2	0.0	0.089991	0.012171	0.068062	0.118090
2	1.0	0.005646	0.000847	0.004157	0.007663
3	0.0	0.024858	0.010188	0.010702	0.056669
3	1.0	0.000516	0.000387	0.000112	0.002383

1

Pearson (with Rao-Scott adjustment):
Unadjusted - chi2(2): 21.2661 with p-value of 0.0000
Adjusted - F(1.52, 47.26): 14.9435 with p-value of 0.0000

Likelihood ratio (with Rao-Scott adjustment):
Unadjusted - chi2(2): 18.3925 with p-value of 0.0001
Adjusted - F(1.52, 47.26): 12.9242 with p-value of 0.0001.

With categorical data, users may want to compare groups. The class Ttest offers algorithms for testing means and proportions from one or two samples.

When sample sizes are too small for areas to produce reliable and stable domain estimates, the small area estimation (sae) techniques can help improve the precision of the estimates. The module *sae* of *samplics* provides routines for producing small area estimates.

7 Conclusion

As with R, Python now provides an open-source library for the design and analysis of survey sampling. The Python library *samplics* allows Python users to remain in the Python ecosystem when designing and analyzing complex samples. Furthermore, we expect that *samplics* will help bring more survey statisticians and official statistics producers to Python. Our ambition with *samplics* is to create a robust, comprehensive, and easy to use ecosystem for survey sampling and the production of official statistics.

References

Chow, S., Shao, J., Wang, H., Y, & Lokhnygina, Y. (2018). *Sample Size Calculations in Clinical Research, Third Edition*. CRC Press, Taylor & Francis Group.

Deville, J. C. & Särndal, C. E. (1992). Calibration estimators in survey sampling. *Journal of the American Statistical Association*, **87**, 376–382.

Dippo, C. S., Fay, R. E., & Morganstein, D. H. (1984). Computing variances from complex samples with replicate weights. In *Proceedings of the Survey Research Methods Section, ASA* (pp. 489–494).

Fay, R. E. (1989). Theory and application of replicate weighting for variance calculations. In *Proceedings of the Survey Research Methods Section, ASA* (pp. 212–217).

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., Fernandez del Rio, J., Wiebe, M., Peterson, P., Gerard-Marchant, P., Sheppard, K.,

Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., & Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, **585**, 357–362.

Krewski, D. & Rao, J. N. K. (1981). Inference from stratified samples: Properties of the linearization, jackknife and balanced repeated replication methods. *The Annals of Statistics*, **9**, 1010–1019.

Lohr, S. L. (2022). *Sampling: Design and Analysis, Third Edition*. CRC Press, Taylor & Francis Group.

McCarthy, P. J. (1966). *Replication: An Approach to the Analysis of Data from Complex Surveys*.

McKinney, W. (2010). Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference* (pp. 56–61).

Rao, J. N. K. & Wu, C. F. J. (1992). Resampling inference with complex survey data. *Journal of the American Statistical Association*, **83**, 231–241.

Ryan, T. P. (2013). *Sample Size Determination and Power*. Jonh Wiley & Sons, Inc.

World Health Organization (2018). *World Health Organization vaccination coverage cluster surveys: reference manual*. https://apps.who.int/iris/handle/10665/272820.