AgEcon SEARCH
RESEARCH IN AGRICULTURAL & APPLIED ECONOMICS

The World's Largest Open Access Agricultural & Applied Economics Digital Library

This document is discoverable and free to researchers across the globe due to the work of AgEcon Search.

Help ensure our sustainability.

Give to AgEcon Search

AgEcon Search
http://ageconsearch.umn.edu
aesearch@umn.edu

*Papers downloaded from **AgEcon Search** may be used for non-commercial purposes and personal study only. No other use, including posting to another Internet site, is permitted without permission from the copyright owner (not AgEcon Search), or as allowed under the provisions of Fair Use, U.S. Copyright Act, Title 17 U.S.C.*

No endorsement of AgEcon Search or its fundraising activities by the author(s) of the following work or their employer(s) is intended or implied.

Check for updates

The Stata Journal (2019)
19, Number 1, pp. 143–184 DOI: 10.1177/1536867X19830912

Updates to the ipfraking ecosystem

Stanislav Kolenikov
Abt Associates
Columbia, MO
stas_kolenikov@abtassoc.com

Abstract. Kolenikov (2014, *Stata Journal* 14: 22–59) introduced the package `ipfraking` for iterative proportional fitting (raking) weight-calibration procedures for complex survey designs. In this article, I briefly describe the original package and updates to the core program and document additional programs that are used to support the process of creating survey weights in the author's production code.

Keywords: st0323_1, ipfraking, ipfraking_report, whatsdeff, totalmatrices, mat2do, xls2row, wgtcellcollapse define, wgtcellcollapse sequence, wgtcellcollapse report, wgtcellcollapse candidate, wgtcellcollapse collapse, survey, svy, calibration, raking, weights, iterative proportional fitting

1 Introduction and background

Large-scale social, behavioral, and health data are often collected via complex survey designs that may involve stratification, multiple stages of selection, unequal probabilities of selection (Korn and Graubard 1995, 1999), or any combination thereof. In an ideal setting, one accounts for varying probabilities of selection by using the Horvitz–Thompson estimator of the totals (Horvitz and Thompson 1952; Thompson 1997), and the remaining sampling fluctuations can be further ironed out by poststratification (Holt and Smith 1979). However, on top of the planned differences in probabilities of obtaining a response from a sampled unit, nonresponse is a practical problem that has been growing more acute in recent years (Groves et al. 2002; Pew Research Center 2012). The analysis weights provided along with the public use microdata by data-collecting agencies are designed to account for unequal probabilities of selection, nonresponse, and other factors affecting imbalance between the population and the sample, thus making the analyses conducted on such microdata generalizable to the target population.

Earlier work (Kolenikov 2014) introduced the package `ipfraking`, which implements calibration of survey weights to known control totals to ensure that the resulting weighted data are representative of the population of interest. The process of calibration is aimed at aligning the sample totals of the key variables with those known for the population as a whole. The remainder of this section provides a condensed treatment of estimation with survey data using calibrated weights; I provided a full description in the previous article.

© 2019 StataCorp LLC

For a given finite population $\mathcal{U}$ of units indexed $i = 1, \ldots, N$, the interests of survey statisticians often lie in estimating the population total of a variable Y:

$$T(Y) = \sum_{i \in \mathcal{U}} Y_i \tag{1}$$

A sample $\mathcal{S}$ of n units indexed by $j = 1, \ldots, n$ is taken from $\mathcal{U}$. If the probability to select the ith unit is known to be π_i, then the probability weights, or design weights, are given by the inverse probability of selection,

$$w_{di} = \pi_i^{-1}$$

where subscript d stands for design probabilities of selection. With these weights, an unbiased (design-based, nonparametric) estimator of the total of (1), according to Horvitz and Thompson (1952), is

$$t(y) = \sum_{j \in \mathcal{S}} \frac{y_j}{\pi_j} \equiv \sum_{j \in \mathcal{S}} w_{dj} y_j$$

Probability weights protect the end user from potentially informative sampling designs in which the probabilities of selection are correlated with outcomes and relieve the user from the need to fully account for the sampling design variables in their analysis. This is required in methods such as multilevel regression with poststratification (Park, Gelman, and Bafumi 2004). Design-based methods generally ensure that inference can be generalized to the finite population even when the statistical models used by analysts and researchers are not specified correctly (Pfeffermann 1993; Binder and Roberts 2003).

Survey statisticians often have auxiliary information on the units in the frame, and such information can be included at the sampling stage to create more efficient designs. Unequal probabilities of selection are then controlled with probability weights, implemented as `[pw=`*varname*`]` in Stata (and can be permanently affixed to the dataset with the `svyset` command; see [SVY] **svyset**).

In many situations, however, usable information is not available beforehand and may appear only in the collected data. For example, the census totals of the age and gender distribution of the population may exist, but age and gender of the sampled units are unknown until they are measured in the survey. One can still capitalize on this additional data by adjusting the weights in such a way that the reweighted data conform to these known figures. The procedures to perform these reweighting steps are generally known as weight calibration (Deville and Särndal 1992; Deville, Särndal, and Sautory 1993; Kott 2006, 2009; Särndal 2007).

Suppose there are several variables, referred to as control variables, that are available for both the population and the sample (age groups, race, gender, educational attainment, etc.). Categorical variables are represented by the 0/1 category indicators, although Kolenikov and Hammer (2015) provide an illustrative example of how the counts of persons in each demographic category within a household (that is, variables taking values 0, 1, 2, ...) can be used to create person-level weights that are

constant within households. Weight calibration aims to adjust the weights via an iterative optimization so that the control totals for the control variables $\mathbf{x}_j = (x_{1j}, \ldots, x_{pj})$, obtained with the calibrated weights w_{cj}, align with the known population totals:

$$\sum_{j \in \mathcal{S}} w_{cj} \mathbf{x}_j = T(\mathbf{X}) \tag{2}$$

The population totals of the control variables in the right-hand side of (2) are assumed to be known from a census or a higher quality survey. Deville and Särndal (1992) framed the problem of finding a suitable set of weights as that of constrained optimization with the control equations (2) serving as constraints. Optimization is targeted at making the discrepancy between the design weights w_{dj} and calibrated weights w_{cj} as close as suitably possible.

The package `ipfraking` (Kolenikov 2014) implements a popular calibration algorithm known as iterative proportional fitting, or raking, that consists of iterative updating (poststratification) of each of the margins. For an in-depth discussion of distinctions between raking and poststratification, see Kolenikov (2016). Since 2014, the continuing code development resulted in additional features that this update documents.

2 Updates to ipfraking program and package

Listed below is the full syntax of `ipfraking`. For a description of its options, see Kolenikov (2014).

2.1 Syntax for ipfraking

`ipfraking` [*if*] [*in*] [`pw=`*varname*], `ctotal(`*matrix_name* [*matrix_name ...*]`)`
{`generate(`*newvar*`)` | `replace`} [`tolerance(#) iterate(#) nodivergence`
`ctrltolerance(#) trace alpha(#) trimhiabs(#) trimhirel(#)`
`trimloabs(#) trimlorel(#) trimfrequency(once` | `sometimes` | `often)`
`double meta nograph loglevel(#) linear`]

2.2 New features of ipfraking

The new features of `ipfraking` concern reporting and diagnostics, an alternative functional form specification, and richer metadata stored in the characteristics of the weight variable.

Reporting of results and errors by `ipfraking` was improved in several ways. It now reports the discrepancy for the worst-fitting category and the number of trimmed observations.

Using example 3 from Kolenikov (2014) with trimming options, we have

```
. capture drop rakedwgt3
. ipfraking [pw=finalwgt], generate(rakedwgt3)
>     ctotal(ACS2011_sex_age Census2011_region Census2011_race)
>     trimhiabs(200000) trimloabs(2000) meta
 Iteration 1, max rel difference of raked weights = 14.95826
 Iteration 2, max rel difference of raked weights = .21474256
 Iteration 3, max rel difference of raked weights = .02754514
 Iteration 4, max rel difference of raked weights = .00511347
 Iteration 5, max rel difference of raked weights = .00095888
 Iteration 6, max rel difference of raked weights = .00018036
 Iteration 7, max rel difference of raked weights = .00003391
 Iteration 8, max rel difference of raked weights = 6.377e-06
 Iteration 9, max rel difference of raked weights = 1.199e-06
 Iteration 10, max rel difference of raked weights = 2.254e-07
The worst relative discrepancy of  3.0e-08 is observed for race == 3
Target value =   20053682; achieved value =   20053682
Trimmed due to the upper absolute limit: 5 weights.
   Summary of the weight changes
               |     Mean    Std. dev.     Min        Max        CV
---------------+--------------------------------------------------
Orig weights   |    11318       7304      2000      79634    .6453
Raked weights  |    22055      18908      4033     200000    .8573
Adjust factor  |   2.1486               0.9220    18.9828
```

(*output omitted*)

```
. char list rakedwgt3[]
  rakedwgt3[source]:          finalwgt
  rakedwgt3[objfcn]:          2.25435521164e-07
  rakedwgt3[maxctrl]:         3.00266819571e-08
  rakedwgt3[converged]:       1
  rakedwgt3[worstcat]:        3
  rakedwgt3[worstvar]:        race
  rakedwgt3[command]:         [pw=finalwgt], generate(rakedwgt3) ctotal(ACS20..
  rakedwgt3[trimloabs]:       trimloabs(2000)
  rakedwgt3[trimhiabs]:       trimhiabs(200000)
  rakedwgt3[trimfrequency]:   sometimes
  rakedwgt3[hash1]:           3644541563
  rakedwgt3[svyset]:          rake( i.sex_age i.region i.race, totals( 11.sex..
  rakedwgt3[mat3]:            Census2011_race
  rakedwgt3[over3]:           race
  rakedwgt3[totalof3]:        _one
  rakedwgt3[Census2011_race]: 7.48567522438e-09
  rakedwgt3[mat2]:            Census2011_region
  rakedwgt3[over2]:           region
  rakedwgt3[totalof2]:        _one
  rakedwgt3[Census2011_region]:
                              3.00266819571e-08
  rakedwgt3[mat1]:            ACS2011_sex_age
  rakedwgt3[over1]:           sex_age
  rakedwgt3[totalof1]:        _one
  rakedwgt3[ACS2011_sex_age]: 4.13778301743e-09
  rakedwgt3[note1]:           Raking controls used: ACS2011_sex_age Census201..
  rakedwgt3[note0]:           1
```

If `ipfraking` determines that the categories do not match in the control totals received from `ctotal()` and those found in the data, it provides a full listing of categories

and explicitly shows the categories not found in one or the other. Using example 2 of Kolenikov (2014), let us modify one of the variables to a nonsensical value:

```
. replace sex_age = 15 if sex_age == 21
(2,056 real changes made)
. ipfraking [pw=finalwgt], generate(rakedwgt2d)
> ctotal(ACS2011_sex_age Census2011_region Census2011_race)
categories of sex_age do not match in the control ACS2011_sex_age and in the
> data (nolab option)
This is what ACS2011_sex_age gives:
  _one:11 _one:12 _one:13 _one:21 _one:22 _one:23
This is what I found in data:
  _one:11 _one:12 _one:13 _one:15 _one:22 _one:23
This is what ACS2011_sex_age has that data don´t:
  _one:21
This is what data have that ACS2011_sex_age doesn´t:
  _one:15
r(111);
```

Option `meta` saves more information in characteristics of the calibrated weight variables that can be used in production diagnostics. The following characteristics are stored with the newly created weight variable (see [P] **char**).

`command`	the full command as typed by the user
matrix_name	the relative matrix difference from the corresponding control total; see [D] **functions**
`trimhiabs`, `trimloabs`, `trimhirel`, `trimlorel`, `trimfrequency`	corresponding trimming options, if specified
`maxctrl`	the greatest `mreldif` between the targets and the achieved weighted totals
`objfcn`	the value of the relative weight change at exit
`converged`	whether `ipfraking` exited because of convergence (`1`) versus because of an increase in the objective function or reaching the limit on the number of iterations (`0`)
`source`	weight variable specified as the `[pw=]` input
`worstvar`	the variable in which the greatest discrepancy between the targets and the achieved weighted totals (`maxctrl`) was observed
`worstcat`	the category of the `worstvar` variable in which the greatest discrepancy was observed

For the control total matrices $\#= 1, 2, \ldots$, the following meta information is stored.

`mat`$\#$	the name of the control total matrix
`totalof`$\#$	the multiplier variable (matrix `coleq`)
`over`$\#$	the margin associated with the matrix (that is, the categories represented by the columns)

Also, `ipfraking` stores the notes regarding the control matrices used and which of the margins did not match the control totals, if any. See [D] **notes**.

The `linear` option provides linear calibration (case 1 of Deville and Särndal [1992]). The weights are calculated analytically:

$$w_{j,\text{lin}} = w_{dj}(1 + \mathbf{x}_j'\lambda), \quad \lambda = \Big(\sum_{j\in\mathcal{S}} w_{dj}\mathbf{x}_j\mathbf{x}_j'\Big)^{-1}\{T(\mathbf{X}) - t(X)\}$$

Because no iterative optimization is required, linear calibration works quickly. However, it undesirably may potentially produce negative weights because the range of weights is not controlled. Because raking works by multiplying the current weights by positive factors, if the input weights are all positive, the output weights will also be positive. Negative weights are not allowed by the official `svy` commands or commands that work with `[pweight]`. In many tasks, running linear weights first, pulling up the negative and small positive weights (`replace weight = 1 if weight <= 1`), and reraking using the "proper" iterative proportional fitting runs faster than raking from scratch. An example of linearly calibrated weights is given below in section 6.

2.3 Utility commands

The original package `ipfraking` provided additional utility commands: `mat2do` and `xls2row`. One of these utility commands, `mat2do`, was updated to provide the option `notimestamp` to omit the time stamps (which tend to unnecessarily throw off the project building and revision control systems).

This update provides two more utility commands, `whatsdeff` and `totalmatrices`.

Design effects

A new utility command, `whatsdeff`, was added to compute the unequal weighting (UWE) design effects (DEFFs) and margins of error. These are common tasks associated with describing survey weights. Specifically, the Transparency Initiative of the American Association for Public Opinion Research (AAPOR 2017) requires that

> For probability samples, the estimates of sampling error will be reported, and the discussion will state whether or not the reported margins of sampling error or statistical analyses have been adjusted for the DEFF due to weighting, clustering, or other factors.

`whatsdeff` *weight_variable* [*if*] [*in*] [`, by(`*varlist*`)`]

The utility command `whatsdeff` calculates the apparent DEFF due to UWE,

$$\text{DEFF}_{\text{UWE}} = 1 + \text{CV}_w^2 = \texttt{1 + r(Var)/(r(mean))\^{}2}$$

using the returned values from `summarize` *weight_variable* (see `help return`). Additionally, it reports the effective sample size, $n/\text{DEFF}_{\text{UWE}}$, and returns the margins of error for the sample proportions that estimate the population proportions of 10% and 50%.

```
. webuse nhanes2, clear
. whatsdeff finalwgt
          Min |      Mean |       Max |        CV |      DEFF |         N |     N eff
--------------+-----------+-----------+-----------+-----------+-----------+-----------
Overall
      2000.00 |  11318.47 |  79634.00 |    0.6453 |    1.4164 |     10351 |   7307.97
. return list
scalars:
                  r(N) =  10351
              r(MOE10) =  .0068792766212984
              r(MOE50) =  .0114654610354974
       r(Neff_Overall) =  7307.974353253639
       r(DEFF_Overall) =  1.416397964696134
```

DEFF can also be broken down by a categorical variable:

```
. whatsdeff finalwgt, by(sex)
          Min |      Mean |       Max |        CV |      DEFF |         N |     N eff
--------------+-----------+-----------+-----------+-----------+-----------+-----------
sex == Male
      2000.00 |  11426.14 |  79634.00 |    0.6578 |    1.4326 |      4915 |   3430.94
sex == Female
      2130.00 |  11221.12 |  61534.00 |    0.6333 |    1.4010 |      5436 |   3880.01
--------------+-----------+-----------+-----------+-----------+-----------+-----------
Overall
      2000.00 |  11318.47 |  79634.00 |    0.6453 |    1.4164 |     10351 |   7307.97
. return list
scalars:
                  r(N) =  10351
              r(MOE10) =  .0068792766212984
              r(MOE50) =  .0114654610354974
       r(Neff_Overall) =  7307.974353253639
       r(DEFF_Overall) =  1.416397964696134
  r(Neff_sex_eq_Femal
    e)                =  3880.00710397866
  r(DEFF_sex_eq_Femal
    e)                =  1.40102836266093
   r(Neff_sex_eq_Male) =  3430.938195872213
   r(DEFF_sex_eq_Male) =  1.432552765279559
```

The estimates of UWE DEFFs that `whatsdeff` produces should be considered a typical magnitude of a DEFF. As pointed out by a referee, in many situations when survey variables are correlated with weights or with the variables that weight calibration is based on, the actual DEFFs reported by postestimation command `estat effect` should

be expected to be lower, provided that variance estimation methods account for calibration properly, for example, via replicate variance estimation as described in Kolenikov (2010) or via the `svy, vce(calibrate)` functionality of the official Stata `svy` suite available in Stata 15.1+ (Valliant and Dever 2018). In other words, for most situations these estimates could be considered an upper bound on this DEFF because this calculation assumes that the weights are independent of the survey variable of descriptive interest.

Conversion of the matrices

A new command, `totalmatrices`, converts the control totals matrices between the formats expected by `ipfraking` and `svycal` (Valliant and Dever 2018).

`totalmatrices` *matrix_list*`, stub(`*name*`)` [`svycal ipfraking replace convert`]

`stub(`*name*`)` provides the naming convention for the converted control total matrices. If the conversion is from `ipfraking` to `svycal`, one matrix whose name is supplied in the `stub()` option will be created. If the conversion is from `svycal` to `ipfraking`, matrices corresponding to each variable will be created and have their names set to concatenation of the stub and the variable name. `stub()` is required.

`svycal` checks that the supplied matrix or matrices are compatible with `svycal` specification of totals as a matrix.

`ipfraking` checks that the supplied matrix or matrices are compatible with `ipfraking`.

`replace` specifies that the matrices with the required names can be overwritten if they already exist in memory.

`convert` is used to request the conversion; otherwise, `totalmatrices` will check only that the format of the inputs seems to be correct.

If you want to convert several matrices from `ipfraking` format to a single matrix in `svycal` format, type

```
. totalmatrices ACS2011_sex_age Census2011_region Census2011_race,
>         ipfraking stub(alltotals) replace convert
It appears that the matrix ACS2011_sex_age is of ipfraking format.
It appears that the matrix Census2011_region is of ipfraking format.
It appears that the matrix Census2011_race is of ipfraking format.
You can now matrix list alltotals to check and then call svycal as:
   svycal [regress|rake] 11.sex_age 12.sex_age 13.sex_age 21.sex_age 22.sex_age
> 23.sex_age 1.region 2.region 3.region 4.region 1.race 2.race 3.race
> [pw=finalwgt], generate(...) totals(alltotals) nocons
I suspect the following would be simpler and could work, too:
   svycal [regress|rake] ibn.sex_age ibn.region ibn.race [pw=finalwgt],
> generate(...) totals(alltotals) nocons
```

If you want to convert a single matrix compatible with `svycal` requirements for its `totals(`*matrix_name*`)` format into a list of matrices compatible with `ipfraking`, type

```
. totalmatrices alltotals, ipfraking stub(totmat_) replace convert
It appears that the matrix alltotals is of the svycal format.
Matrices created:
matrix list totmat_sex_age
matrix list totmat_region
matrix list totmat_race
. matrix list totmat_region
totmat_region[1,4]
            _one:     _one:     _one:     _one:
               1         2         3         4
region  40679030  49205289  85024007  53385843
```

Note that at the moment, `totalmatrices` does not handle conversion of interactions, which is arguably one of the greatest strengths of `svycal`. As noted in section 7, for interactions to work out with `ipfraking`, standalone variables need to be created, and `totalmatrices` would rather have the user do that.

2.4 New commands in the package

I added two new commands to the package, `ipfraking_report` and `wgtcellcollapse`, and I document them in the subsequent sections of this article. The former provides reports on the raked weights, including summaries of the unweighted data, data with the input weights, and data with the calibrated weights. The latter creates a mostly automated flow of collapsing weighting cells that are too detailed (and hence have low sample sizes).

3 Excel reports on raked weights: ipfraking_report

`ipfraking_report using` *filename*`, raked_weight(`*weight_variable*`)`
[`matrices(`*matrix_list*`) by(`*varlist*`) xls replace force`]

The utility command `ipfraking_report` produces a detailed report describing the raked weights and places it into *filename*`.dta` (or, if the `xls` option is specified, both *filename*`.dta` and *filename*`.xls`).

Along the way, `ipfraking_report` runs a regression of the log raking adjustment ratio on the calibration variables. This regression is expected to have R^2 equal to or close to 1 and residual variance equal to or close to 0. This naturally produces high t test values, but the purpose of this regression is not in establishing "significance" of any variable in explaining the outcome (which we know to be predicted with near certainty). Instead, the regression coefficients provide insights regarding which categories received greater versus smaller adjustments (which in turn indicate lower response or coverage rates for the corresponding population subgroups). Conversely, control variables that

are associated with relatively similar adjustment factors may be contributing relatively little to the weight adjustment and may be candidates for removal from the list of control totals.

The regression output, using example 3 from Kolenikov (2014), is as follows:

```
. ipfraking_report using rakedwgt3-report, raked_weight(rakedwgt3) replace
> by(_one)
Margin variable sex_age (total variable: _one; categories: 11 12 13 21 22 23).
Margin variable region (total variable: _one; categories: 1 2 3 4).
Margin variable race (total variable: _one; categories: 1 2 3).
Auxiliary variable _one (categories: 1).
file rakedwgt3-report.dta saved
```

Source	SS	df	MS		
Model	2086.13859	10	208.613859	Number of obs	= 10,351
				F(10, 10340)	> 99999.00
Residual	.78315703	10,340	.000075741	Prob > F	= 0.0000
				R-squared	= 0.9996
Total	2086.92175	10,350	.201634952	Adj R-squared	= 0.9996
				Root MSE	= .0087

__000003	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
sex_age						
11	.0644365	.0002775	232.21	0.000	.0638925	.0649804
12	.4545577	.0003154	1441.25	0.000	.4539395	.455176
13	.6782466	.0002804	2418.71	0.000	.6776969	.6787963
22	.3966406	.0003049	1300.84	0.000	.3960429	.3972383
23	.7304392	.0002726	2679.97	0.000	.7299049	.7309734
region						
NE	-.4455127	.0002536	-1756.49	0.000	-.4460099	-.4450155
MW	-.4428144	.0002335	-1896.53	0.000	-.4432721	-.4423567
W	-.6672675	.0002407	-2772.21	0.000	-.6677393	-.6667957
race						
Black	.3360321	.0002848	1180.08	0.000	.3354739	.3365902
Other	1.613276	.0006303	2559.34	0.000	1.612041	1.614512
_cons	.5864801	.0002455	2388.48	0.000	.5859988	.5869614

```
Raking adjustments for sex_age variable:
  the smallest was        1.798 for category 21 (21)
  the greatest was        3.732 for category 23 (23)
Raking adjustments for region variable (1=NE, 2=MW, 3=S, 4=W):
  the smallest was        0.922 for category 4 (W)
  the greatest was        1.798 for category 3 (S)
Raking adjustments for race variable (1=white, 2=black, 3=other):
  the smallest was        1.798 for category 1 (White)
  the greatest was        9.023 for category 3 (Other)
```

We can see that `ipfraking` had to make greater adjustments to the weights of older females (`sex_age==23`, that is, `sex==2 & age==3`; the adjustment factor for this category was 3.732 versus the low of 1.798 for young women) and especially of individuals of other races (the adjustment factor was 9.023, versus 1.798 for the whites). The diagnostic value is in the differences in the adjustment factors with the same variable. Because no attempt is being made to average across the population or the sample or to assign

the "base" variable, the absolute reported values of the adjustment factors may not be meaningful. In the example above, 1.798 figures both as the greatest adjustment factor of the region variable and as the lowest adjustment factor for the race and sex-by-age interaction. As is easily seen from regression output, this value is the exponent of the intercept `1.798=exp(0.586)`. Because all the "estimates" of the region-specific coefficients are negative, the lowest reported value is less than this baseline value. Because all the "estimates" of the race and sex-by-age indicators are positive, all the category-specific adjustment factors are greater than this baseline value. This is an interplay of the base categories and the differences in the demographic composition within each category of a control total variable vis-a-vis other weighting variables.

3.1 Options for ipfraking_report

`raked_weight(`*weight_variable*`)` specifies the name of the raked weight variable to create the report for. `raked_weight()` is required.

`matrices(`*matrix_list* [*matrix_list* ...]`)` provides a list of known control totals. The `ipfraking_report` command will pick up the raking variables and their categories. Each matrix is expected to be compatible with the matrices consumed by `ipfraking` as control totals, the `ctotal()` option. While the functionality of producing results by different variables is provided with the `by()` option, passing the known control totals with `matrices()` allows comparing the required versus achieved control totals.

`by(`*varlist*`)` specifies a list of additional variables for which the weights are to be tabulated in the raking weights report. The difference with the `matrices()` option is that the control totals for these variables may not be known (or may not be relevant). In particular, `by(_one)`, where `_one` is identically 1, will produce the overall report.

`xls` requests exporting the report to an Excel file.

`replace` specifies that the files produced by `ipfraking_report` (that is, the `.dta` and the `.xls` file if `xls` option is specified) should be overwritten.

`force` requests that `ipfraking_report` provide summaries of weights for a given variable each time it is encountered. The multiple opportunities include being one of the raking margins picked up from the control totals saved by `ipfraking, meta`; being supplied with the `by()` option; and being supplied with the `matrices()` option. The reasons to include a variable multiple times in these options is to see how the weights perform depending on whether a variable with known control totals is included as a raking margin.

3.2 Variables in the raking report

The raking report file contains the following variables.

Variable name	Definition
`Weight_Variable`	the name of the weight variable, `generate()`
`C_Total_Margin_Variable_Name`	the name of the control margin, `rowname` of the corresponding `ctotal()` matrix
`C_Total_Margin_Variable_Label`	the label of the control margin variable
`Variable_Class`	the role of the variable in the report: `Raking margin`: a variable used as a calibration margin (picked up automatically from the `ctotal()` matrix, provided the `meta` option was specified); `Other known target`: supplied with the `matrices()` option of `ipfraking_report`; `Auxiliary variable`: additional variable supplied with the `by()` option of `ipfraking_report`
`C_Total_Arg_Variable_Name`	the name of the multiplier variable
`C_Total_Arg_Variable_Label`	the label of the multiplier variable
`C_Total_Margin_Category_Number`	numeric value of the control total category
`C_Total_Margin_Category_Label`	label of the control total category
`C_Total_Margin_Category_Cell`	an indicator of whether a weighting cell was produced by collapsing categories using `wgtcellcollapse`
`Category_Total_Target`	the control total to be calibrated to (the specific entry in the `ctotal()` matrix)
`Category_Total_Prop`	control total proportion (the ratio of the specific entry in the `ctotal()` matrix to the matrix total)
`Unweighted_Count`	number of sample observations in the category
`Unweighted_Prop`	unweighted proportion
`Unweighted_Prop_Discrep`	difference `Unweighted_Prop`− `Category_Total_Prop`
`Category_Total_SRCWGT`	weighted category total with input weight
`Category_Prop_SRCWGT`	weighted category proportion with input weight

Continued on next page

Variable name	Definition
`Category_Total_Discrep_SRCWGT`	difference `Category_Total_SRCWGT`− `Category_Total_Target`
`Category_Prop_Discrep_SRCWGT`	difference `Category_Prop_SRCWGT`− `Category_Total_Prop`
`Category_RelDiff_SRCWGT`	`reldif(Category_Total_SRCWGT, Category_Total_Target)`
`Overall_Total_SRCWGT`	sum of source weights
`Source`	the name of the matrix from which the totals were obtained
`Comment`	placeholder for comments, to be entered during manual review

For each of the input weights (`SRCWGT` suffix), raked weights (`RKDWGT` suffix), and raking ratio (the ratio of raked and input weights, `RKDRATIO` suffix), the following summaries are provided.

Variable name	Definition
`Min_`*WEIGHT*	minimum of the respective weights
`P25_`*WEIGHT*	25th percentile of the respective weights
`P50_`*WEIGHT*	median of the respective weights
`P75_`*WEIGHT*	75th percentile of the respective weights
`Max_`*WEIGHT*	maximum of the respective weights
`Mean_`*WEIGHT*	mean of the respective weights
`SD_`*WEIGHT*	standard deviation of the respective weights
`DEFF_`*WEIGHT*	apparent UWE DEFF of the respective weights

3.3 Example

Continuing with the example of calibration by region, race, and sex-by-age interaction, we find that a glimpse of the raking report looks as follows:

```
. use rakedwgt3-report, clear
(Weighting report on rakedwgt3)
. list C_Total_Margin_Variable_Name C_Total_Margin_Category_Label
>      Category_Total_Target Category_Total_RKDWGT DEFF_SRCWGT DEFF_RKDWGT,
>      sepby(C_Total_Margin_Variable_Name)
```

	C_Tota..	~y_Label	Categor~t	Categor..	DEFF_SR~T	DEFF_RK~T
1.	sex_age	11	41995394	41995394	1.2148059	1.6259899
2.	sex_age	12	42148662	42148662	1.2462168	1.5716613
3.	sex_age	13	26515340	26515340	1.2241095	1.5460785
4.	sex_age	21	41164255	41164255	1.2325105	1.5639529
5.	sex_age	22	43697440	43697440	1.1937826	1.5175312
6.	sex_age	23	32773080	32773080	1.233902	1.664307
7.	region	NE	40679030	40679030	1.3056639	1.3657837
8.	region	MW	49205289	49205289	1.3475551	1.4909581
9.	region	S	85024007	85024006	1.4950056	1.4912995
10.	region	W	53385843	53385844	1.459859	2.3772667
11.	race	White	1.784e+08	1.784e+08	1.4059259	1.4337901
12.	race	Black	29856865	29856865	1.5173846	1.5092533
13.	race	Other	20053682	20053682	1.3179136	1.2264706
14.	_one	1	.	2.283e+08	1.4164382	1.7349278

The last line, corresponding to the auxiliary variable `_one` identically equal to 1 (this variable was present in the dataset because it was used by `ipfraking` as a multiplier), contains summaries for the sample as a whole. I recommend to always include it (note the use of `ipfraking_report, ... by(_one)` in the syntax in the previous section).

The functionality of `ipfraking_report` is aimed at manual quality control, which typically involves i) review of variables and categories with raking factors that differ the most (in the output such as that shown on page 152) and ii) review of the resulting report file in Excel (for example, for DEFFs and discrepancies between targets and achieved totals).

4 Collapsing weighting cells: wgtcellcollapse

An additional new component of the `ipfraking` package is a tool to semiautomatically collapse weighting cells to achieve a required minimal size of the weighting cell. A typical recommendation is to have cells of size at least 30 to 50.

`wgtcellcollapse` *task* [*if*] [*in*] [, *task_options*]

where *task* is one of the following:

`report` lists the currently defined collapsing rules.

`define` defines collapsing rules explicitly.

`sequence` creates collapsing rules for a sequence of categories.

`candidate` finds rules applicable to a given category.

`label` labels collapsed cells using the original labels after `wgtcellcollapse collapse`.

`collapse` performs cell collapsing.

4.1 wgtcellcollapse report

Syntax

`wgtcellcollapse report, variables(`*varlist*`)` [`break`]

Options

`variables(`*varlist*`)` specifies the list of variables for which to report the collapsing rules. `variables()` is required.

`break` makes `wgtcellcollapse report` exit with an error when technical inconsistencies are encountered.

4.2 wgtcellcollapse define

Syntax

`wgtcellcollapse define, variables(`*varlist*`)` [`from(`*numlist*`) to(`#`)`
`label(`*string*`) max(`#`) clear`]

Options

`variables(`*varlist*`)` specifies the list of variables for which the collapsing rule can be used. `variables()` is required.

`from(`*numlist*`)` specifies the list of categories that can be collapsed according to this rule.

`to(`#`)` specifies the numeric value of the new collapsed category.

`label(`*string*`)` provides the value label to be attached to the new collapsed category.

`max(`#`)` overrides the automatically determined max value of the collapsed variable.

`clear` clears all the rules currently defined.

Example

Let us demonstrate the two subcommands introduced so far with the following toy example.

```
. clear
. set obs 4
number of observations (_N) was 0, now 4
. generate byte x = _n
. label define x_lbl 1 "One" 2 "Two" 3 "Three" 4 "Four"
. label values x x_lbl
. wgtcellcollapse define, variable(x) from(1 2 3) to(123)
. wgtcellcollapse report, variable(x)
Rule (1): collapse together
  x == 1 (One)
  x == 2 (Two)
  x == 3 (Three)
  into x == 123 (123)
  WARNING: unlabeled value x == 123
```

For automated quality control purposes, the `break` option of `wgtcellcollapse report` can be used to abort the execution when encountering technical deficiencies in the rules or in the data. In the above example, the label of the new category 123 was not defined. Should the `break` option be specified, an absence of the category label would be considered a serious enough deficiency to stop with an error:

```
. wgtcellcollapse report, variable(x) break
Rule (1): collapse together
  x == 1 (One)
  x == 2 (Two)
  x == 3 (Three)
  into x == 123 (123)
  ERROR: unlabeled value x == 123
assertion is false
r(9);
. wgtcellcollapse define, variable(x) clear
. wgtcellcollapse define, variable(x) from(1 2 3) to(123)
> label("One through three")
. wgtcellcollapse report, variable(x) break
Rule (1): collapse together
  x == 1 (One)
  x == 2 (Two)
  x == 3 (Three)
  into x == 123 (One through three)
```

4.3 wgtcellcollapse sequence

Syntax

`wgtcellcollapse sequence, variables(`*varlist*`) from(`*numlist*`) depth(#)`

Options

`variables(`*varlist*`)` specifies the list of variables for which the collapsing rule can be used. `variables()` is required.

`from(`*numlist*`)` specifies the sequence of values from which the plausible subsequences can be constructed. `from()` is required.

`depth(#)` specifies the maximum number of the original categories that can be collapsed. `depth()` is required.

Example

Continuing with the toy example introduced above, let us see an example of moderate-length sequences to collapse categories:

```
. clear
. set obs 4
number of observations (_N) was 0, now 4
. generate byte x = _n
. label define x_lbl 1 "One" 2 "Two" 3 "Three" 4 "Four"
. label values x x_lbl
. wgtcellcollapse sequence, variable(x) from(1 2 3 4) depth(3)
. wgtcellcollapse report, variable(x)
Rule (1): collapse together
  x == 1 (One)
  x == 2 (Two)
  into x == 212 (One to Two)
Rule (2): collapse together
  x == 2 (Two)
  x == 3 (Three)
  into x == 223 (Two to Three)
Rule (3): collapse together
  x == 3 (Three)
  x == 4 (Four)
  into x == 234 (Three to Four)
Rule (4): collapse together
  x == 1 (One)
  x == 2 (Two)
  x == 3 (Three)
  into x == 313 (One to Three)
```

```
Rule (5): collapse together
  x == 1 (One)
  x == 223 (Two to Three)
  into x == 313 (One to Three)
Rule (6): collapse together
  x == 3 (Three)
  x == 212 (One to Two)
  into x == 313 (One to Three)
Rule (7): collapse together
  x == 2 (Two)
  x == 3 (Three)
  x == 4 (Four)
  into x == 324 (Two to Four)
Rule (8): collapse together
  x == 2 (Two)
  x == 234 (Three to Four)
  into x == 324 (Two to Four)
Rule (9): collapse together
  x == 4 (Four)
  x == 223 (Two to Three)
  into x == 324 (Two to Four)
```

Note how `wgtcellcollapse sequence` automatically created labels for the collapsed cells.

When creating sequential collapses, `wgtcellcollapse sequence` uses the following conventions in assigning the values for the new collapsed categories:

- First comes the length of the collapsed subsequence (up to `depth(#)`).
- Then comes the starting value of the category in the subsequence (padded by zeros as needed).
- Then comes the ending value of the category in the subsequence (padded by zeros as needed).

In the example above, rules 7 through 9 lead to collapsing into the new category 324, which stands for "the subsequence of length 3 that starts with category 2 and ends with category 4". A numeric value of the collapsed category that reads like 50412 means "the subsequence of length 5 that starts with category 4 and ends with category 12". In that second example, `wgtcellcollapse sequence` padded the value of 4 with an additional 0, so the length of resulting collapsed category value is always (number of digits of the sequence length) + twice (number of digits of the greatest source category).

Note that `wgtcellcollapse sequence` respects the order in which the categories are supplied in the `from()` option and does not sort them. If the categories are supplied in the order 2, 4, 1, and 3, then `wgtcellcollapse sequence` would collapse 2 with 4, 4 with 1, and 1 with 3:

```
. wgtcellcollapse define, var(x) clear
. wgtcellcollapse sequence, var(x) from(2 4 1 3) depth(2)
. wgtcellcollapse report, var(x)
Rule (1): collapse together
  x == 2 (Two)
  x == 4 (Four)
  into x == 224 (Two to Four)
Rule (2): collapse together
  x == 4 (Four)
  x == 1 (One)
  into x == 241 (Four to One)
Rule (3): collapse together
  x == 1 (One)
  x == 3 (Three)
  into x == 213 (One to Three)
```

4.4 wgtcellcollapse candidate

Syntax

`wgtcellcollapse candidate, variable(`*varname*`) category(#)`
`[ maxcategory(#) ]`

Options

`variable(`*varname*`)` specifies the variable to be collapsed. `variable()` is required.

`category(#)` specifies the category to be collapsed. `category()` is required.

`maxcategory(#)` specifies the maximum value of the categories in the candidate rules to be returned.

Example

The rules found are quietly returned through the mechanism of `sreturn` (see [P] **return**) because they are intended to stay in memory sufficiently long for `wgtcellcollapse collapse` to evaluate each rule. Going back to the example from the previous section with sequential collapses of depth 3, we can identify the following candidates for categories 2 and 212 (collapsed values of 1 and 2) and a nonexistent category of 55:

```
. wgtcellcollapse candidate, variables(x) category(2)
. sreturn list
macros:
           s(goodrule) : "1 2 4 7 8"
              s(rule8) : "2:234=324"
              s(rule7) : "2:3:4=324"
              s(rule4) : "1:2:3=313"
              s(rule2) : "2:3=223"
              s(rule1) : "1:2=212"
                s(cat) : "2"
                  s(x) : "x"
. wgtcellcollapse candidate, variables(x) category(2) max(9)
. sreturn list
macros:
           s(goodrule) : "1 2 4 7"
              s(rule7) : "2:3:4=324"
              s(rule4) : "1:2:3=313"
              s(rule2) : "2:3=223"
              s(rule1) : "1:2=212"
                s(cat) : "2"
                  s(x) : "x"
. wgtcellcollapse candidate, variables(x) category(212)
. sreturn list
macros:
           s(goodrule) : "6"
              s(rule6) : "3:212=313"
                s(cat) : "212"
                  s(x) : "x"
. wgtcellcollapse candidate, variables(x) category(55)
. sreturn list
macros:
                s(cat) : "55"
                  s(x) : "x"
```

In the second call to the option, we used `max(9)` to restrict the returned rules to the rules that deal only with the original categories (so rule 8 that involved a collapsed category 234 was omitted). It relies on the naming conventions described in the previous section: any of the collapsed cells would have three-digit values. In the third call, we requested a list of rules that involve a collapsed category `cat(212)`. Requests for nonexisting categories are not considered errors but simply produce empty lists of "good rules".

4.5 wgtcellcollapse label

Syntax

`wgtcellcollapse label, variable(`*varname*`) [verbose force]`

Options

`variable(`*varname*`)` specifies the collapsed variable to be labeled. `variable()` is required.

`verbose` outputs the labeling results. There may be a lot of output.

`force` instructs `wgtcellcollapse label` to use only categories present in the data.

Example

An example is given in section 5.2 below.

4.6 wgtcellcollapse collapse

Syntax

`wgtcellcollapse collapse [`*if*`] [`*in*`], variables(`*varlist*`) mincellsize(#)`
`saving(`*dofile_name*`) [generate(`*newvar*`) replace append feed(`*varname*`)`
`strict sort(`*varlist*`) run maxpass(#) maxcategory(#) zeroes(`*numlist*`)`
`greedy]`

Options

`variables(`*varlist*`)` provides the list of variables whose cells are to be collapsed. When more than one variable is specified, `wgtcellcollapse collapse` proceeds from right to left, that is, first attempts to collapse the rightmost variable. `variables()` is required.

`mincellsize(#)` specifies the minimum cell size for the collapsed cells. For most weighting purposes, values of 30 to 50 can be recommended. `mincellsize()` is required.

`saving(`*dofile_name*`)` specifies the name of the do-file that will contain the cell-collapsing code. `saving()` is required.

`generate(`*newvar*`)` specifies the name of the collapsed variable to be created.

`replace` overwrites the do-file if one exists.

`append` appends the code to the existing do-file.

`feed(`*varname*`)` provides the name of an already existing collapsed variable.

`strict` modifies the behavior of `wgtcellcollapse collapse` to use only collapsing rules for which all participating categories have nonzero counts.

`sort(`*varlist*`)` sorts the dataset before proceeding to collapse the cell. The default sort order is in terms of the values of the collapsed variable. A different sort order may produce a different set of collapsed cells when cells are tied on size.

`run` specifies that the do-file created is run upon completion. This option is typically specified with most runs.

`maxpass(#)` specifies the maximum number of passes through the dataset. The default is `maxpass(10000)`.

`maxcategory(#)` specifies the maximum category value of the variable being collapsed. It is passed to the internal calls to `wgtcellcollapse candidate`; see above.

`zeroes(`*numlist*`)` provides a list of the categories of the collapsed variable that may have zero counts in the data.

`greedy` modifies the behavior of `wgtcellcollapse collapse` to prefer the rules that collapse the maximum number of categories.

Remarks

The primary intent of `wgtcellcollapse collapse` is to create the code that can be used in both a survey data file and a population targets data file that are assumed to have identically named variables. Thus, it not only manipulates the data in memory and collapses the cells but also produces the do-file code that can be recycled in automated weight production. To that effect, when a do-file is created with the `replace` and `saving()` options, the user must specify the `generate()` option to provide the name of the collapsed variable; and when the said do-file is appended with the `append` and `saving()` options, the name of that variable is provided with the `feed()` option.

The algorithm `wgtcellcollapse collapse` uses to identify the cells to be collapsed uses a variation of greedy search. It first identifies the cells with the lowest (positive) counts; finds the candidate rules for the variable or variables to be collapsed; evaluates the counts of the collapsed cells across all of these candidate rules; and uses the rule that produces the smallest size of the collapsed cell across all applicable rules. So when it finds several rules that are applicable to the cell being currently processed that has a size of 5, and the candidate rules produce cells of sizes 7, 10, and 15, `wgtcellcollapse collapse` will use the rule that produces the cell of size 7. The algorithm runs until all cells have sizes of at least `mincellsize(#)` or until `maxpass(#)` passes through the data are executed. In real-world situations with messy data, this basic algorithm often produces inconsistent results generally because it fails to identify empty cells or fully track the cells that have already been collapsed. For that reason, I provide some hints to modify its behavior. Section 5 demonstrates a worked-out example.

Hint 1. Because `wgtcellcollapse collapse` works with the sample data, it will not be able to identify categories that are not observed in the sample (for example, rare categories missing because of unit nonresponse) but may be present in the population. This will lead to errors at the raking stage, when the control total matrices have more categories than the data, forcing `ipfraking` to stop with errors. (See page 147 for the output `ipfraking` provides in this situation.) To help with that, the option `zeroes()` allows the user to pass the categories of the variables that are known to exist in the population but not in the sample.

Hint 2. The behavior of `wgtcellcollapse collapse, zeroes()` leads to undesirable artifacts when collapsing long streaks of sequential zeros. While the edge zero cells would be collapsed with their nonzero neighbors, the zero cell in between may end up being collapsed with some faraway cells, creating collapsing rules with breaks in the sequences. To improve upon that behavior, the option `greedy` makes `wgtcellcollapse collapse` look for a rule that combines as many cells as possible, thus collapsing as many categories with zero counts in one pass as it can.

Hint 3. Other than for dealing with zero cells, the option `strict` should be specified most of the time. It ensures that each cell in a candidate rule being evaluated has some data in it.

Hint 4. If you want to guarantee some specific combination of cells to be collapsed by `wgtcellcollapse collapse`, the most reliable way is to explicitly identify them with the `if` qualifier and specify a very large cell size like `mincellsize(10000)` so that `wgtcellcollapse collapse` makes every possible effort to collapse those cells. Because the resulting cell or cells will fall short of that size, the program will exit with a complaint that this size could not be achieved, but hopefully the cells will be collapsed as needed.

Hint 5. If any of the cells fail to reach the required sizes, the problematic values are returned to the user in the `r(failed)` macro as a space-delimited list and in the `r(cfailed)` as a comma-delimited list. The content of the `r(failed)` macro can be used in code that could read

```
foreach c in `r(failed)´ {
    ...
    * run some diagnostics for each category that failed
    list ... if collapsed_variable == `c´
    ...
}
```

while the content of the `r(cfailed)` macro can be used in code that could read

```
list ... if inlist(collapsed_variable,`r(cfailed)´)
```

Also, these returned values should be used in production code by using the `assert` command (Gould 2003) to ascertain that these macros are empty (that is, no errors were encountered):

```
assert "`r(failed)´" == ""
```

A referee noted that `wgtcellcollapse` could also have utility in preparing for hotdeck imputation procedures. The textbook versions of hotdeck procedures impute missing data by assuming a missing at random model (Rubin 1976) with conditioning on a set of categorical variables, that is, cells of a multivariate table. Akin to weighting procedures, hotdeck procedures are more stable with larger cells, so cell collapsing is often recommended to achieve minimal cell sizes (with an understanding of the bias-versus-variance tradeoff built into these collapsing decisions). For a review of the hotdeck and related imputation methods, see Andridge and Little (2010).

5 Extended motivating example

The primary purpose of developing `wgtcellcollapse` and adding it to the `ipfraking` suite was to address the need to collapse cells of the margin variables so that each cell has a minimum sample size in a way that can be easily made consistent between a sample data and the population targets data. The problem arises when some of the target variables have dozens of categories, most of which have small counts. Example where such needs arise include

- transportation surveys, where many stations will have low counts of boardings, alightings, or both;
- country of origin variables in household surveys, where most countries will have very low counts; and
- continuous age variables that can be collapsed into age groups differently for different values of race or sex.

The workflow of `wgtcellcollapse` is demonstrated with the following simulated transportation dataset of trips along a commuter metro line composed of 21 stations:

```
. use stations, clear
. list station_id, sep(0)
```

	station_id
1.	1. Alewife
2.	2. Brookline
3.	8. Carmenton
4.	11. Dogville
5.	18. East End
6.	24. Framington
7.	26. Grand Junction
8.	30. High Point
9.	36. Irvingtown
10.	39. Johnsville
11.	40. King Street
12.	44. Limerick
13.	47. Moscow City
14.	49. Ninth Street
15.	50. Ontario Lake
16.	53. Picadilly Square
17.	55. Queens Zoo
18.	60. Redline Circle
19.	62. Silver Spring
20.	68. Toledo Town
21.	69. Union Station

Suppose turnstile counts were collected at entrances (`board_id`) and exits (`alight_id`) of these stations, producing the following population figures:

```
. use trip_population, clear
. table board_id daypart, c(sum num_pass) cellwidth(10) mi
```

board_id	daypart AM Peak	Midday	PM Reverse	Night	Weekend
1. Alewife	1423	34	219	113	44
2. Brookline	7198	298	773	169	144
8. Carmenton	19254	181	3739	872	422
11. Dogville	12626	872	3476	769	1270
18. East End	2470	143	1263	145	114
24. Framington	634	50	1296	133	60
26. Grand Junction	2208	233	439	88	166
30. High Point	4319	424	3740	482	115
36. Irvingtown	1221	34	444	30	167
39. Johnsville	93	4	64	2	6
40. King Street	398	46	76	11	13
44. Limerick	1021	19	129	53	34
47. Moscow City	3300	776	984	140	301
49. Ninth Street	38	22	191	5	5
50. Ontario Lake	606	22	80	18	23
53. Picadilly Square	642	71	622	153	69
55. Queens Zoo	331	23	174	15	19
60. Redline Circle	270	4	63	13	3
62. Silver Spring	3402	240	950	206	445
68. Toledo Town	5085	61	744	272	112

```
. table alight_id daypart, c(sum num_pass) cellwidth(10) mi
```

alight_id	daypart AM Peak	Midday	PM Reverse	Night	Weekend
2. Brookline	19	.	3	2	.
8. Carmenton	492	18	56	23	15
11. Dogville	2475	42	423	153	80
18. East End	929	31	193	67	68
24. Framington	404	13	91	28	27
26. Grand Junction	576	20	147	42	41
30. High Point	2189	89	560	165	167
36. Irvingtown	288	10	91	21	18
39. Johnsville	41	.	11	2	1
40. King Street	131	3	38	8	6
44. Limerick	277	9	87	20	18
47. Moscow City	1746	78	556	142	128
49. Ninth Street	88	2	25	3	4
50. Ontario Lake	232	11	70	14	14
53. Picadilly Square	633	33	198	47	47
55. Queens Zoo	230	10	71	13	14
60. Redline Circle	90	2	26	3	4
62. Silver Spring	1134	67	369	91	85
68. Toledo Town	1372	81	444	112	118
69. Union Station	53193	3038	16007	2733	2677

Most people ride the train to the last station, with much smaller traffic at other population centers.

Suppose a survey was administered to a sample of the metro line users, with the following counts of cases collected.

```
. use trip_sample, clear
. table board_id daypart, c(freq) cellwidth(10) mi
```

board_id	daypart				
	AM Peak	Midday	PM Reverse	Night	Weekend
1. Alewife	46	4	11	7	3
2. Brookline	236	4	35	6	7
8. Carmenton	653	4	184	47	24
11. Dogville	410	41	166	35	56
18. East End	85	5	64	4	4
24. Framington	30	3	74	3	1
26. Grand Junction	72	13	23	5	6
30. High Point	158	20	187	25	12
36. Irvingtown	34	2	25	1	15
39. Johnsville	5	1	1	.	.
40. King Street	17	1	2	.	1
44. Limerick	28	.	9	1	3
47. Moscow City	94	31	49	7	13
49. Ninth Street	.	.	9	.	.
50. Ontario Lake	13	1	4	1	1
53. Picadilly Square	23	4	35	7	5
55. Queens Zoo	10	1	14	.	2
60. Redline Circle	13	.	5	.	.
62. Silver Spring	106	18	38	12	17
68. Toledo Town	149	6	33	11	3

```
. table alight_id daypart, c(freq) cellwidth(10) mi
```

alight_id	daypart				
	AM Peak	Midday	PM Reverse	Night	Weekend
2. Brookline	1	.	.	.	.
8. Carmenton	11	1	1	.	1
11. Dogville	85	1	14	6	5
18. East End	36	1	18	1	4
24. Framington	15	1	2	2	2
26. Grand Junction	15	2	8	1	1
30. High Point	73	4	22	11	8
36. Irvingtown	9	.	4	2	2
39. Johnsville	3	.	1	.	.
40. King Street	.	.	3	.	.
44. Limerick	13	.	2	.	2
47. Moscow City	81	6	22	6	6
49. Ninth Street	3	1	1	.	.
50. Ontario Lake	2	.	1	2	1
53. Picadilly Square	23	1	8	3	2
55. Queens Zoo	6	.	5	1	.
60. Redline Circle	5	.	.	.	.
62. Silver Spring	49	.	19	3	9
68. Toledo Town	43	3	24	6	7
69. Union Station	1,709	138	813	128	123

Because only 3,654 surveys were collected from a total of 96,783 riders, we would reasonably expect that there is a need for weighting and nonresponse adjustment. The data available for calibration include the population turnstile counts listed above. We will produce interactions of the day part and the station that will serve as two weighting margins (one for the stations where the metro users boarded and one for the stations where they got off).

First, we need to define the weighting rules. In this case, the stations are numbered sequentially, with the northernmost station Alewife being number 1 and the southernmost station, Union Station, where everybody gets off to rush to their city jobs or attractions, being number 69. Below, we create a list of stations and provide it to `wgtcellcollapse sequence`. We would be collapsing stations along the line with the expectation that travelers boarding or leaving at adjacent stations within the same day part are more similar to one another than the travelers boarding or leaving a particular station at different times of the day. Collapsing rules need to be defined for the `daypart` variable as well—mostly because `wgtcellcollapse collapse` expects all variables to have collapsing rules defined.

```
. use trip_sample, clear
. wgtcellcollapse sequence, var(daypart) from(2 3 4) depth(3)
. levelsof board_id, local(stations_on)
1 2 8 11 18 24 26 30 36 39 40 44 47 49 50 53 55 60 62 68
. levelsof alight_id, local(stations_off)
2 8 11 18 24 26 30 36 39 40 44 47 49 50 53 55 60 62 68 69
. local all_stations: list stations_on | stations_off
. wgtcellcollapse sequence, var(board_id alight_id) from(`all_stations´)
> depth(20)
. save trip_sample_rules, replace
file trip_sample_rules.dta saved
```

The syntax above relies on the stations being in the sequential order, which is how the output of `levelsof` is organized. Otherwise, the internal numeric identifiers of the stations would need to be supplied in the order in which the trains run through them.

The number of collapsing rules for variables `board_id` and `alight_id` created by `wgtcellcollapse sequence` is 2,961 each.

Below, we present the final syntax to produce the collapsed cells. A more detailed version of this article, available upon request from the author, describes the process of building the syntax through trial and (mostly) error. A reader who plans to thoroughly use `wgtcellcollapse` should read the full description.

```
. use trip_sample_rules, clear
. * (1) Run 1
. wgtcellcollapse collapse, variables(daypart board_id) mincellsize(1)
>         zeroes(39 44 49 60) greedy maxcategory(99)
>         generate(dpston5) saving(dpston5.do) replace run
```

(*output omitted*)

```
. * (2) Run 2
. wgtcellcollapse collapse, variables(daypart board_id) mincellsize(20)
>         strict feed(dpston5) saving(dpston5.do) append run
```

(*output omitted*)

```
. assert "`r(failed)´" == ""
. * (3) Run 3
. wgtcellcollapse collapse, variables(daypart alight_id) mincellsize(1)
>         zeroes(2 40 60) greedy maxcategory(99)
>         generate(dpstoff5) saving(dpstoff5.do) replace run
Pass 0 through the data...
  smallest count = 1 in the cell      1000002
Processing zero cells...
  Invoking rule 39:40=23940 to collapse zero cells
  replace dpstoff5 = 1023940 if inlist(dpstoff5, 1000039, 1000040)
Pass 0 through the data...
  smallest count = 1 in the cell      1000002
  Invoking rule 1:2:8=30108 to collapse zero cells
  replace dpstoff5 = 2030108 if inlist(dpstoff5, 2000001, 2000002, 2000008)
Pass 0 through the data...
  smallest count = 1 in the cell      1000002
  Invoking rule 30:36:39:40:44=53044 to collapse zero cells
  replace dpstoff5 = 2053044 if inlist(dpstoff5, 2000030, 2000036, 2000039,
> 2000040, 2000044)
```

(*output omitted*)

```
Pass 0 through the data...
  smallest count = 1 in the cell      1000002
  Invoking rule 53:55:60=35360 to collapse zero cells
  replace dpstoff5 = 5035360 if inlist(dpstoff5, 5000053, 5000055, 5000060)
Pass 0 through the data...
  smallest count = 1 in the cell      1000002
Pass 12 through the data...
  smallest count = 1 in the cell      1000002
  Done collapsing! Exiting...
. * (4) Run 4
. wgtcellcollapse collapse if inlist(daypart,4,5) & inrange(alight_id,49,50),
>       variables(daypart alight_id) mincellsize(1)
>       feed(dpstoff5) zeroes(49) maxcategory(99) saving(dpstoff5.do) append run
Pass 12 through the data...
  smallest count = 1 in the cell      5000050
Processing zero cells...
  Invoking rule 49:50=24950 to collapse zero cells
  replace dpstoff5 = 4024950 if inlist(dpstoff5, 4000049, 4000050)
Pass 12 through the data...
  smallest count = 1 in the cell      5000050
  Invoking rule 49:50=24950 to collapse zero cells
  replace dpstoff5 = 5024950 if inlist(dpstoff5, 5000049, 5000050)
Pass 12 through the data...
  smallest count = 1 in the cell      5024950
```

```
Pass 14 through the data...
  smallest count = 1 in the cell      5024950
  Done collapsing! Exiting...
. * (5) Run 5
. * special cells for weekend
. wgtcellcollapse collapse if daypart==5 & inrange(alight_id,1,36),
>         variables(daypart alight_id) mincellsize(50)
>         strict feed(dpstoff5) saving(dpstoff5.do) append run
Pass 14 through the data...
  smallest count = 1 in the cell      5000026
  Invoking rule 24:26=22426
  replace dpstoff5 = 5022426 if inlist(dpstoff5, 5000024, 5000026)
Pass 15 through the data...
  smallest count = 1 in the cell      5030108
  Invoking rule 11:30108=40111
  replace dpstoff5 = 5040111 if inlist(dpstoff5, 5000011, 5030108)
```

(*output omitted*)

```
Pass 19 through the data...
  smallest count = 10 in the cell      5043040
  Invoking rule 70126:43040=110140
  replace dpstoff5 = 5110140 if inlist(dpstoff5, 5070126, 5043040)
Pass 20 through the data...
  smallest count = 23 in the cell      5110140
  WARNING: could not find any rules to collapse dpstoff5 == 5110140
Pass 21 through the data...
  smallest count = .i in the cell      1000002
  Done collapsing! Exiting...
. * (6) Run 6
. wgtcellcollapse collapse if daypart==5 & inrange(alight_id,44,68),
>         variables(daypart alight_id) mincellsize(50)
>         strict feed(dpstoff5) saving(dpstoff5.do) append run
Pass 20 through the data...
  smallest count = 1 in the cell      5024950
  Invoking rule 24950:35360=54960
  replace dpstoff5 = 5054960 if inlist(dpstoff5, 5024950, 5035360)
Pass 21 through the data...
  smallest count = 2 in the cell      5000044
  Invoking rule 44:47=24447
  replace dpstoff5 = 5024447 if inlist(dpstoff5, 5000044, 5000047)
```

(*output omitted*)

```
Pass 25 through the data...
  smallest count = 27 in the cell      5094468
  WARNING: could not find any rules to collapse dpstoff5 == 5094468
Pass 26 through the data...
  smallest count = .i in the cell      1000002
  Done collapsing! Exiting...
```

```
. * (7) Run 7
. * all other cells
. wgtcellcollapse collapse, variables(daypart alight_id) mincellsize(20)
>         strict feed(dpstoff5) saving(dpstoff5.do) append run
Pass 25 through the data...
  smallest count = 1 in the cell      1000002
  Invoking rule 2:8=20208
  replace dpstoff5 = 1020208 if inlist(dpstoff5, 1000002, 1000008)
Pass 26 through the data...
  smallest count = 1 in the cell      2000011
  Invoking rule 11:18=21118
  replace dpstoff5 = 2021118 if inlist(dpstoff5, 2000011, 2000018)
  (output omitted)
Pass 64 through the data...
  smallest count = 15 in the cell      3054960
  Invoking rule 62:54960=64962
  replace dpstoff5 = 3064962 if inlist(dpstoff5, 3000062, 3054960)
Pass 65 through the data...
  smallest count = 21 in the cell      2200168
  Done collapsing! Exiting...
. assert "`r(failed)´" == ""
```

Each pass identified the smallest cell count, the cell where this low count is found, the rule that can be used to collapse this cell with some other cell (see more on determination of what `wgtcellcollapse` believes to be the best rule below), and Stata code that can be used to apply this collapsing rule.

The collapsed values of `dpston` (daypart-station-on) and `dpstoff` (daypart-station-off) combine the values of the parent variables. The value of `dpston==1000003` indicates the combination of categories `daypart==1` and station number 3. The value of `dpston==1023940` indicates `daypart==1` and sequence of two stations from 39 to 40. The value of `dpston==2053044` indicates `daypart==2` and sequence of five stations from 30 to 44.

The first call to `wgtcellcollapse` uses the options `generate()` and `replace` to create a new variable and a new do-file, while subsequent calls `feed()` this variable back and `append` additional cell-collapsing code to the existing do-file.

The `zeroes()` option specified in calls 1, 3, and 4 notified `wgtcellcollapse` that there are values of `alight_id` that are never observed. (Riders get on the train on these stations and exit in small numbers, but no completed surveys were obtained.) The `mincellsize(1)` option effectively instructed `wgtcellcollapse` to exit once all of these zero cells are identified and merged with nonzero cells. The `maxcategory(99)` option restricts collapsing rules only to those that involve individual stations. It relies on the convention that all the individual station IDs are less than 99 and all the collapsed values are at least `20102` (that is, the first two stations merged together, forming a cell of size 2 that stretched from `01` to `02`). Without these options, `wgtcellcollapse` would be allowed to pick up one of the previously collapsed cells. However, it seems safer to collapse stations with zero count to only one station.

Using the subsampling conditions like `if daypart==5 & inrange(alight_id,1,36)` in calls 5 and 6 effectively specifies one specific collapsing cell that the algorithm could not otherwise identify. A higher target value, `mincellsize(50)`, is used in conjunction to ensure that the algorithm does not exit prematurely. The special missing value `.i` that appears in the smallest count report, as opposed to the actual counts in other runs, is used internally to stop `wgtcellcollapse` after all the relevant cases selected by the `if` qualifier have been processed.

Using the `greedy` option in call 3 made it possible to collapse the streak of zero counts in the midday part from `36. Irvington` to `44. Limerick`. Without it, each individual zero count station would be paired with a nonmissing station, which leads to cells that overlap in space.

After all zero counts stations are processed, the `strict` option should almost always be specified, as is done in runs 2, 5, 6, and 7. It prevents `wgtcellcollapse` from picking up rules that may have skipped categories in them. In other words, it ensures that the collapsed cells are contiguous.

The resulting cells satisfy the sample size requirements of at least 20 cases per cell:

```
. by dpston5, sort:  assert _N >= 20
. by dpstoff5, sort: assert _N >= 20
```

5.1 Pipeline to raking

As its output, `wgtcellcollapse` produced two files, one for each weighting margin, called `dpston.do` and `dpstoff.do`. An interested reader is welcome to `type` them; they contain long sequences of `replace` commands to perform the cell collapsing. These do-files are intended to be run on both the sample and the population data to create identical collapsed categories and produce consistent matrices of the population control totals for `ipfraking`.

```
. use trip_population, clear
. run dpston5.do
. total num_pass, over(dpston5)
Total estimation                  Number of obs   =        719
      1000001: dpston5 = 1000001
      1000002: dpston5 = 1000002
  (output omitted)
      5000011: dpston5 = 5000011
      5026268: dpston5 = 5026268
      5030108: dpston5 = 5030108
      5051836: dpston5 = 5051836
      5093960: dpston5 = 5093960
```

Over	Total	Std. Err.	[95% Conf.	Interval]
num_pass				
1000001	1423	967.7508	-476.9595	3322.959
1000002	7198	4895.91	-2414.011	16810.01
(output omitted)				
5000011	1270	834.301	-367.961	2907.961
5026268	557	364.4324	-158.4805	1272.481
5030108	610	263.2061	93.25444	1126.746
5051836	622	215.5712	198.7749	1045.225
5093960	473	261.8954	-41.17225	987.1723

```
. matrix dpston5 = e(b)
. matrix coleq dpston5 = _one
. matrix rownames dpston5 = dpston5
. run dpstoff5.do
. total num_pass, over(dpstoff5)
Total estimation                  Number of obs   =        719
      1000018: dpstoff5 = 1000018
      1000030: dpstoff5 = 1000030
  (output omitted)
      5000069: dpstoff5 = 5000069
      5094468: dpstoff5 = 5094468
      5110140: dpstoff5 = 5110140
```

Over	Total	Std. Err.	[95% Conf.	Interval]
num_pass				
1000018	929	360.7303	220.7878	1637.212
1000030	2189	868.0319	484.8161	3893.184
(output omitted)				
4000069	2733	728.6906	1302.381	4163.619
4080130	480	132.6806	219.5117	740.4883
4123668	476	72.94794	332.7832	619.2168
5000069	2677	895.7917	918.316	4435.684
5094468	432	87.57763	260.0612	603.9388
5110140	423	120.0254	187.3574	658.6426

```
. matrix dpstoff5 = e(b)
```

```
. matrix coleq dpstoff5 = _one
. matrix rownames dpstoff5 = dpstoff5
. use trip_sample_rules, clear
. run dpston5
. run dpstoff5
. generate byte _one = 1
. ipfraking [pw=_one], ctotal(dpston5 dpstoff5) generate(raked_weight5)
 Iteration 1, max rel difference of raked weights = 37.856256
 Iteration 2, max rel difference of raked weights = .06404821
 Iteration 3, max rel difference of raked weights = .00891802
 Iteration 4, max rel difference of raked weights = .00128619
 Iteration 5, max rel difference of raked weights = .00018966
 Iteration 6, max rel difference of raked weights = .00002818
 Iteration 7, max rel difference of raked weights = 4.198e-06
 Iteration 8, max rel difference of raked weights = 6.257e-07
The worst relative discrepancy of  7.8e-08 is observed for dpstoff5 == 5110140
>
Target value =        423; achieved value =        423
   Summary of the weight changes
                |     Mean    Std. dev.     Min         Max        CV
----------------+-------------------------------------------------------
Orig weights    |        1           0         1           1         0
Raked weights   |   26.487       5.754    13.174      38.634     .2172
Adjust factor   |  26.4869              13.1743     38.6339
. whatsdeff raked_weight5
          Min   |    Mean   |     Max   |     CV    |    DEFF   |    N    |  N eff
----------------+-----------+-----------+-----------+-----------+---------+---------
Overall
        13.17   |   26.49   |    38.63  |   0.2172  |   1.0472  |   3654  | 3489.37
```

5.2 Informative labels

Once the collapsing rules are finalized, several types of category labels can be attached to the resulting collapsed cells. Using the mechanics of labels in multiple languages (see [D] **label language**), `wgtcellcollapse label` defines three "languages" to describe the cells. The language `numbered_ccells` may be convenient for debugging purposes in fine-tuning the collapsing algorithms, while the language `texted_ccells` would prove useful for `ipfraking_report` in creating human-readable labels. In Stata Markup and Control Language output, the `label language` instructions are clickable, so the user can click the command rather than copying and pasting it.

```
. wgtcellcollapse label, variable(dpston5)
(language default renamed unlabeled_ccells)
(language numbered_ccells now current language)
(language texted_ccells now current language)
To attach the numeric labels (of the kind "dpston5==1000001"), type:
   label language numbered_ccells
To attach the text labels (of the kind "dpston5==AM Peak; 1. Alewife"), type:
   label language texted_ccells
The original state, which is also the current state, is:
   label language unlabeled_ccells
```

```
. wgtcellcollapse label, variable(dpstoff5)
To attach the numeric labels (of the kind "dpstoff5==1000018"), type:
   label language numbered_ccells
To attach the text labels (of the kind "dpstoff5==AM Peak; 18. East End"), type:
   label language texted_ccells
The original state, which is also the current state, is:
   label language unlabeled_ccells

. label language numbered_ccells

. tabulate dpstoff5 if daypart==5
```

Long ID of the interaction	Freq.	Percent	Cum.
daypart==5, alight_id==69	123	71.10	71.10
daypart==5, alight_id==94468	27	15.61	86.71
daypart==5, alight_id==110140	23	13.29	100.00
Total	173	100.00	

```
. label language texted_ccells

. tabulate dpstoff5 if daypart==5
```

Long ID of the interaction	Freq.	Percent	Cum.
Weekend; 69. Union Station	123	71.10	71.10
Weekend; 44. Limerick to 68. Toledo Tow	27	15.61	86.71
Weekend; 1. Alewife to 40. King Street	23	13.29	100.00
Total	173	100.00	

```
. label language unlabeled_ccells

. tabulate dpstoff5 if daypart==5
```

Interactions of daypart alight_id, with some collapsing	Freq.	Percent	Cum.
5000069	123	71.10	71.10
5094468	27	15.61	86.71
5110140	23	13.29	100.00
Total	173	100.00	

6 Linear calibrated weights

Using the final set of collapsed categories in the simulated transportation data example, let us demonstrate the linear calibration option of `ipfraking`, added since Kolenikov (2014). In mathematical terms, linear weights explicitly solve the minimization problem of finding a set of weights $\{w_{li}, i = 1, \ldots, n\}$, where the subscript l stands for linear calibration, such that

$$\sum_{i=1}^{n} \frac{(w_{li} - w_{di})^2}{w_{di}} \to \min$$

Deville and Särndal (1992) and Särndal, Swensson, and Wretman (1992) provide explicit treatment of the problem and the resulting analytical expressions that are coded in `ipfraking, linear`. The main advantage of linear weight calibration is a much faster computing time. To demonstrate it, we will time the output by using the immediate timing results, `set rmsg on` (see [R] **set**).

```
. set rmsg on
r; t=0.00 12:18:12
. ipfraking [pw=_one], ctotal(dpston5 dpstoff5) nograph generate(raked_weight5)
 Iteration 1, max rel difference of raked weights = 37.856256
 Iteration 2, max rel difference of raked weights = .06404821
 Iteration 3, max rel difference of raked weights = .00891802
 Iteration 4, max rel difference of raked weights = .00128619
 Iteration 5, max rel difference of raked weights = .00018966
 Iteration 6, max rel difference of raked weights = .00002818
 Iteration 7, max rel difference of raked weights = 4.198e-06
 Iteration 8, max rel difference of raked weights = 6.257e-07
The worst relative discrepancy of  7.8e-08 is observed for dpstoff5 == 5110140
Target value =        423; achieved value =        423
   Summary of the weight changes
               |     Mean    Std. dev.    Min        Max       CV
---------------+--------------------------------------------------
Orig weights   |        1          0         1          1        0
Raked weights  |   26.487      5.754    13.174     38.634    .2172
Adjust factor  |  26.4869              13.1743    38.6339
r; t=0.99 12:18:13
. ipfraking [pw=_one], ctotal(dpston5 dpstoff5) nograph generate(raked_weight5l)
>  linear
Linear calibration
The worst relative discrepancy of  3.7e-14 is observed for dpstoff5 == 5110140
Target value =        423; achieved value =        423
   Summary of the weight changes
               |     Mean    Std. dev.    Min        Max       CV
---------------+--------------------------------------------------
Orig weights   |        1          0         1          1        0
Raked weights  |   26.487     5.7523    12.518     38.204    .2172
Adjust factor  |  26.4869              12.5178    38.2040
r; t=0.43 12:18:13
. set rmsg off
. label variable raked_weight5l "Linear calibrated weights"
```

```
. compare raked_weight5 raked_weight5l
                                        ---------- difference ----------
                                count       minimum      average     maximum
------------------------------------------------------------------------------
raked_w~5<raked_~5l              1896     -1.813144    -.0476911   -3.17e-11
raked_w~5>raked_~5l              1758      2.18e-09     .0514348    2.405758
                           ----------
jointly defined                  3654     -1.813144     3.21e-10    2.405758
                           ----------
total                            3654
```

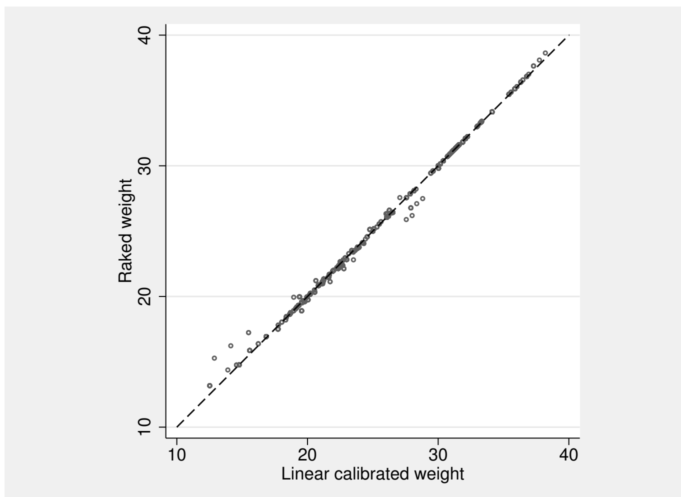

Figure 1. Linear and raked weights

The speed advantages of `linear` calibration are quite clear (0.43 seconds versus 0.99 seconds), although raking convergence in 8 iterations is quite fast in my experience. It is not unusual to see dozens of iterations, and when higher order interactions are being used as raking margins, subtle correlations between the cells arise, slowing down convergence and requiring hundreds of iterations. Linear calibrated and raked weights are similar, as figure 1 demonstrates. However, the lowest of the linearly calibrated weights are slightly smaller than comparable raked weights. The weights produced by linear and raking calibration methods should be expected to agree in general, but the match along the diagonal line of the plot should not be expected to be ideal.

As mentioned before, in extreme situations, linearly calibrated weights may become negative, which creates additional issues. First, Stata's `svy` commands or estimation commands with `pweight` specifications do not accept negative weights and produce error messages when such weights are encountered. This is not a bug but indeed a welcome behavior. Second, negative weights are typically difficult to interpret; within a common, although not technically accurate interpretation of sampling weights as the number of population units that a sampled unit represents, it is puzzling to find a negative

number of such population units. The way I use the linear calibration functionality of `ipfraking` is to produce "preliminary" sets of weights. If the weights at the low end satisfy the natural range restriction (greater than 0, to prevent input data check errors with estimation commands; or in some applications, greater than 1, to satisfy the "number of population units" interpretation that is often desirable), these weights can be "accepted" as final. If they do not, `ipfraking` can be called with trimming syntax such as `trimloabs(1)`. The linear weights can then be used as a starting point to accelerate convergence using the `from()` option.

While the general theory of calibrated estimation (Deville and Särndal 1992) ensures that linear calibrated weights (analyzed as case 1 in that article) and raked weights (case 2) are asymptotically equivalent, this equivalence implicitly requires that the scales of the population control matrices are identical. In practice, different control total variables may come from different sources, and some sources may either have different populations to which they can technically be generalized or come at different scales such as proportions versus population totals. Nearly every dual-frame random-digit dialing survey of the general U.S. population that I dealt with would use the American Community Survey data for demographic variables (which would come with the desirable population scaling) and National Health Interview Survey data for phone use variables (cell phone only, landline only, both, or none), which would come in the form of proportions. While the raking version of `ipfraking` would not have any difficulty incorporating both (with the caveat that the final scale of weights will be determined by the last variable in the `ctotal()` list), the linear version of weights would try to find a middle point between the population totals that are on the scale of millions and proportions that are on the scale of about 1. The results would likely be quite strange.

7 Other packages with similar functionality

Other packages provide similar basic functionality (that is, raked weights, with or without trimming). Kolenikov (2014) provided comparisons with `survwgt` (Winter 2002), `ipfweight` (Bergmann 2011) and `maxentropy` (Wittenberg 2010) and reported that the weights produced by these packages were identical within numeric accuracy.

Concurrently with Kolenikov (2014), another weight calibration package, `sreweight` (Pacifico 2014), was published in the *Stata Journal*. It implements a full range of objective functions from Deville and Särndal (1992) and does so faster than `ipfraking` because the core iterative functionality is implemented in Mata. Finally, Stata 15.1 now provides the `svycal` command, undocumented at the time of the writing of this article, although described and exemplified in detail in Valliant and Dever (2018). Compared with `svycal`, the core functionality of `ipfraking` provides a richer set of trimming specifications. I compared the weights produced by `ipfraking` with those produced by `sreweight` and `svycal` in the case of the basic raking procedure without trimming, and they agree within numeric accuracy:

```
. svycal rake ibn.sex_age ibn.region ibn.race [pw=finalwgt],
>         generate(rakedwgt2a) totals(alltotals) nocons
note: 4.region omitted because of collinearity
note: 3.race omitted because of collinearity
. compare rakedwgt2 rakedwgt2a
                                          difference
                             count       minimum      average      maximum
-------------------------------------------------------------------------
rakedwgt2<rakedw~2a           6843     -.0057653    -.0001227   -3.27e-06
rakedwgt2>rakedw~2a           3508      1.56e-07     .0002394    .0038718
-------------------------------------------------------------------------
jointly defined              10351     -.0057653     3.41e-13    .0038718
-------------------------------------------------------------------------
total                        10351
. assert reldif(rakedwgt2, rakedwgt2a) < c(epsfloat)
```

Weights produced by `ipfraking` also agree with those produced by the R package `survey` (Lumley 2010, 2018), namely, the `survey::calibrate(...,calfun="raking")` function, and those produced by SAS raking macro `RAKE_AND_TRIM()` (Izrael et al. 2017). When trimming options are specified, the results from different packages diverge because trimming operations appear to be implemented differently in each package.

It is unfortunate that so much effort has gone into replicating the functionality by the different authors. The primary distinction of the current `ipfraking` package is the rich ecosystem that goes along with it, aimed at producing survey weights by a survey organization in a way that is efficient, robust, and flexible codewise.

As a practicing survey statistician who needs to experiment with the weights a lot, I believe that `ipfraking` is easier to experiment with than `svycal` or `sreweight` for several reasons. First, `ipfraking` relies on the control totals being carried over from `svy: total` with minimal modifications such as renaming row and column names; passing control totals is more cumbersome with other packages. Second, `ipfraking` produces detailed diagnostics of problems and oddities it encounters along the way, assisting the survey statistician in assessing whether the resulting weights are satisfactory.

For relatively simpler tasks of producing replicate weights and calibrating them at the same time, `survwgt` provides easier syntax. Coding the task with `ipfraking` or any other package would require explicit cycles.

From a code development perspective, I believe that relying on matching the order of control totals and variables, as required by all other community-contributed packages, creates a potential for errors that are easy to make and difficult to catch. If you supplied 20 control totals and 19 variables, at which position in the list should the 20th missing variable be? With `ipfraking` and the official `svycal`, the risk that a control total figure would be associated with a wrong category of the control total variable is much lower because they pair the values and the categories in a single object (via the names attached to the control total matrices) or explicit syntax `value.variable =` # specification of `svycal`. However, the matrix naming is different between `ipfraking` and `svycal`, so I provide a conversion tool, `totalmatrices`, in this update.

Additionally, `ipfraking` can incorporate variables that sum up to different totals, for example, totals from different sources or years, or totals and proportions if unified data are not available, with the side effect of producing weights whose totals agree with the last control total variable. Without trimming, doing so ensures that proportions for each calibration variable are satisfied. Because `maxentropy`, `sreweight`, and `svycal` produce weights by optimization with the goal of satisfying all totals simultaneously, it is unclear what the properties of the resulting weight would be when the scales of control totals differ between variables and whether the resulting weights would produce marginal proportions that agree between the control totals and calibrated weights.

Compared with `ipfraking`, the official `svycal` command handles interactions far more graciously and consistently with the Stata user experience of using factor variables in regression models (see [U] **11.4.3 Factor variables**). It creates the necessary interactions internally on the fly, while `ipfraking` requires explicit generation of interaction variables.

Finally, of all the Stata weight calibration packages, `ipfraking` is unique in offering the possibility of using multipliers other than 0 and 1 (that is, category dummies).

Ultimately, the choice of the package is a matter of personal preference, package familiarity, and coding style.

8 Acknowledgments

The author is grateful to an anonymous referee for a thorough review and thoughtful suggestions, to Tom Guterbock for bug reports and functionality suggestions, and to Jason Brinkley for extensive comments and critique. The opinions stated in this article are of the author only and do not represent the position of Abt Associates.

9 References

AAPOR. 2017. AAPOR terms and conditions for transparency certification. American Association for Public Opinion Research. https://www.aapor.org/AAPOR_Main/media/MainSiteFiles/TI-Terms-and-Conditions-10-4-17.pdf.

Andridge, R. R., and R. J. A. Little. 2010. A review of hot deck imputation for survey non-response. *International Statistical Review* 78: 40–64.

Bergmann, M. 2011. ipfweight: Stata module to create adjustment weights for surveys. Statistical Software Components S457353, Department of Economics, Boston College. http://econpapers.repec.org/software/bocbocode/s457353.htm.

Binder, D. A., and G. R. Roberts. 2003. Design-based and model-based methods for estimating model parameters. In *Analysis of Survey Data*, ed. R. L. Chambers and C. J. Skinner, 29–48. Chichester, UK: Wiley.

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

Deville, J.-C., C.-E. Särndal, and O. Sautory. 1993. Generalized raking procedures in survey sampling. *Journal of the American Statistical Association* 88: 1013–1020.

Gould, W. 2003. Stata tip 3: How to be assertive. *Stata Journal* 3: 448.

Groves, R. M., D. A. Dillman, J. L. Eltinge, and R. J. A. Little, eds. 2002. *Survey Nonresponse*. New York: Wiley.

Holt, D., and T. M. F. Smith. 1979. Post stratification. *Journal of the Royal Statistical Society, Series A* 142: 33–46.

Horvitz, D. G., and D. J. Thompson. 1952. A generalization of sampling without replacement from a finite universe. *Journal of the American Statistical Association* 47: 663–685.

Izrael, D., M. P. Battaglia, A. A. Battaglia, and S. W. Ball. 2017. You do not have to step on the same rake: SAS raking macro—generation IV. SAS Global Forum 2017. https://support.sas.com/resources/papers/proceedings17/0470-2017-poster.pdf.

Kolenikov, S. 2010. Resampling variance estimation for complex survey data. *Stata Journal* 10: 165–199.

———. 2014. Calibrating survey data using iterative proportional fitting (raking). *Stata Journal* 14: 22–59.

———. 2016. Post-stratification or non-response adjustment? *Survey Practice* 9(3). https://www.surveypractice.org/article/2809-post-stratification-or-non-response-adjustment.

Kolenikov, S., and H. Hammer. 2015. Simultaneous raking of survey weights at multiple levels. *Survey Methods: Insights from the Field*. https://surveyinsights.org/?p=5099.

Korn, E. L., and B. I. Graubard. 1995. Analysis of large health surveys: Accounting for the sampling design. *Journal of the Royal Statistical Society, Series A* 158: 263–295.

———. 1999. *Analysis of Health Surveys*. New York: Wiley.

Kott, P. S. 2006. Using calibration weighting to adjust for nonresponse and coverage errors. *Survey Methodology* 32: 133–142.

———. 2009. Calibration weighting: Combining probability samples and linear prediction models. In *Sample Surveys: Inference and Analysis, vol. 29B*, ed. D. Pfeffermann and C. R. Rao, 55–82. Oxford: Elsevier.

Lumley, T. S. 2010. *Complex Surveys: A Guide to Analysis Using R*. Hoboken, NJ: Wiley.

———. 2018. *Survey analysis in R*. http://r-survey.r-forge.r-project.org/survey/.

Pacifico, D. 2014. sreweight: A Stata command to reweight survey data to external totals. *Stata Journal* 14: 4–21.

Park, D. K., A. Gelman, and J. Bafumi. 2004. Bayesian multilevel estimation with poststratification: State-level estimates from national polls. *Political Analysis* 12: 375–385.

Pew Research Center. 2012. Assessing the representativeness of public opinion surveys. Technical report, Pew Research Center for People and Press. http://www.people-press.org/2012/05/15/assessing-the-representativeness-of-public-opinion-surveys/.

Pfeffermann, D. 1993. The role of sampling weights when modeling survey data. *International Statistical Review* 61: 317–337.

Rubin, D. B. 1976. Inference and missing data. *Biometrika* 63: 581–592.

Särndal, C.-E. 2007. The calibration approach in survey theory and practice. *Survey Methodology* 33: 99–119.

Särndal, C.-E., B. Swensson, and J. Wretman. 1992. *Model Assisted Survey Sampling*. New York: Springer.

Thompson, M. E. 1997. *Theory of Sample Surveys*. London: Chapman & Hall.

Valliant, R., and J. A. Dever. 2018. *Survey Weights: A Step-by-Step Guide to Calculation*. College Station, TX: Stata Press.

Winter, N. 2002. survwgt: Stata module to create and manipulate survey weights. Statistical Software Components S427503, Department of Economics, Boston College. https://ideas.repec.org/c/boc/bocode/s427503.html.

Wittenberg, M. 2010. An introduction to maximum entropy and minimum cross-entropy estimation using Stata. *Stata Journal* 10: 315–330.

About the author

Stanislav (Stas) Kolenikov is a principal scientist at Abt Associates. His work involves applications of statistical methods in collecting survey data for public opinion research, public health, economic policy making, transportation, and other disciplines that rely on primary survey data collection. Within survey methodology, his expertise includes advanced sampling techniques, survey weighting, calibration, missing data imputation, variance estimation, nonresponse analysis and adjustment, small area estimation, and mode effects. Besides survey statistics, Stas has extensive experience developing and applying statistical methods in social sciences, focusing on structural equation modeling and microeconometrics. He has been writing Stata programs since 1998, or Stata 5.