TEA for survey processing

U.S. Census Bureau
Center for Statistical Research and Methodology Division

March 19, 2013

Abstract

TEA is a system designed to unify and streamline survey processing, from raw data to editing to imputation to dissemination of output. Its primary focus is in finding observations that are missing data, fail consistency checks, or risk the disclosure of sensitive information, and then using a unified imputation process to fix all of these issues. Beyond this central focus, it includes tools for reading in data, generating fully synthetic data, and other typical needs of the survey processor.

1 Overview

WE INTEND to implement the many steps of survey processing into a single framework, where the interface with which analysts work is common across surveys, the full description of how a survey is processed is summarized in one place, and the code implementing the procedures are internally well-documented and reasonably easy to maintain.

Raw data is often rife with missing items, logical errors, and sensitive information. To ignore these issues risks alienating respondents and data users alike, and so data modification is a necessary part of the production of quality survey and census data. TEA is a statistical library designed to eradicate these errors by creating a framework that is friendly to interact with, effectively handles Census-sized data sets, and does the processing quickly even with relatively sophisticated methods.

This paper gives a detailed overview of the TEA system, its components, and how they're used. If you have this tutorial you should also have a working installation of TEA and a sample directory including `demo.spec`, `demo.R`, and `dc_pums_08.csv`. Basic versions of all of the steps described below are already implemented and running, though the system will evolve and grow as it is applied in new surveys and environments.

2 System basics

TEA implements a two step process for addressing issues with raw data. The first is to identify those failures listed above (missing data, logical errors, or sensitive information), and then, having identified problem data, impute new values to replace the old

ones. Although the term *imputation* is typically used only to describe filling in missing data, we use it broadly to mean any modification of a data item that involves choosing among alternatives, regardless of which of the above failures prompted the fill-in.

In terms of how TEA is implemented, we break the process into two parts: the specification of variable details—such as which file to read in, what values should be top-coded, or the full description of the model used for filling in missing data—and the actual prodecure to be run by the computer based on those inputs. The specification of details as mentioned above will go into a plain text file, herein called the `spec` file. Based on your inputs to the `spec` file (which we will explain later as to what those inputs are/can be), you then run a script in a user-friendly statistical computing framework called **R**. This is where the computing (editing, imputation, etc) takes place. We will explain this in more detail later as well. For now, let's look closer at the `spec` file:

2.1 The spec file

The full specification of the various steps of TEA, from input of data to final production of output tables, that are performed during your implementation of TEA is specified in a single file, the *spec file*. There are several intents to this setup. First, because the spec file is separate from programming languages like R or SAS, it is a simpler grammar that is easier to write, and so analysts can write technical specifications without the assistance of a programmer or a full training in a new programming language. In other words, the spec file allows users whose areas of expertise are not in programming to customize and use TEA in a standardized and acccessible environment. To see why this is the case, observe the following script that is taken from the `demo.spec` file (which you can open using Vi or any other text editor):

```
database: test.db

input {
        input_file :dc_pums_08.csv
        output_table: dc_pums
}
```

In this snippet of the `demo.spec` file, we specified a database to use, an input file to be parsed, and an output table to write our imputations to. Behind the scenes, SQL scripts and C functions are being executed. As we will see, other scripts that are run from the `spec` file perform more complicated behind-the-scenes algorithms. However, before we go through an example of a full `spec` file, let's take a look at the environment and underlying systems in which TEA runs to gain a better understanding of the processes taking place in the `spec` file:

2.2 Environment and underlying systems

TEA is based on three systems: C, R, and SQL. Each provides facilities that complement the others:

SQL is designed around making fast queries from databases, such as finding all observations within a given age range and income band. Any time we need a subset of the data, we will use SQL to describe and pull it. SQL is a relatively simple language, so users unfamiliar with it can probably learn the necessary SQL in a few minutes—in fact, a reader who claims to know no SQL will probably already be able to read and modify the SQL-language conditions in the `checks` sections below.

The TEA system stores data using an SQL database. The system queries the database as needed to provide input to the statistical/mathematical components (R, C function libraries, etc.). Currently, TEA is written to support SQLite as its database interface; however it would be possible to implement other interfaces such as Oracle or MySQL.

Output at each step is also to the database, to be read as input into the next step. Therefore, the state of the data is recorded at each step in the process, so suspect changes may be audited.

Outside the database, to control what happens and do the modeling, the TEA package consists of roughly 5,000 lines of R and C code.

R is a relatively user-friendly system that makes it easy to interact with data sets and write quick scripts to glue together segments of the survey-processing pipeline. R is therefore the interactive front-end for TEA. Users will want to get familiar with the basics of R. As with SQL, users well-versed in R can use their additional knowledge to perform analysis beyond the tools provided by the TEA system.

C is the fastest human-usable system available for manipulating matrices, making draws from distributions, and other basic model manipulations. Most of the numerical work will be in C. The user is not expected to know any C at all, because R procedures are provided that do the work of running the underlying C-based procedures.

Now that we have a better idea of the environments in which TEA is run, let's take a look at an example of a full `spec` file: `demo.spec`:

The configuration system (spec file) is a major part of the user interface with TEA. As is evident from `demo.spec`, there are many components of the `spec` file that all perform certain functions. We begin by explaining the concept of *keys:*

2.3 Keys

Everything in the `spec` file is a key/value pair (or, as may be more familiar to you, a tag: data definition). Each key in the `spec` file has a specific purpose and will be outlined in this tutorial. To begin, we start in the header of `demo.spec`:

```
database: demo.db
id: SSN
```

Here, `database: demo.db` and `id: SSN` are examples of key: value pairs. As is the case in `demo.spec`, you will always need to begin your `spec` file by declaring the database (`database:your_database.db`) key and the unique identifier (`id:your_unique_identifier`) key. The `database:` key identifies the relational database where all of your data will be manipulated during the various processes of TEA. The `id` key provides a column in the database table that serves as the unique identifier for each set of data points in

your data set. Though the `id` key is not strictly necessary, we strongly advise that you include one in your `spec` file to prevent any unnecessary frustration as most of TEA's routines require its use. More information on both of these keys can be found in the appendix of this tutorial.

You may have noticed that keys are assigned values with the following syntax:

```
key:  value
```

This syntax is equivalent to the following form with curly braces:

```
database{
        demo.db
}

id {
        SSN
}
```

Clearly, this form is not as convenient as the `key:  value` form for single values. However, it allows us to have multiple values associated with a single line of data, and even subkeys. For example, take the next line in `demo.spec` (the computer will ignore everything after a #, so those lines are comments for your fellow humans):

```
input {
    #As above, if you have a key with one value, you can use a colon:
    input file: ss08pdc.csv
    overwrite: no
    output table: dc
        #However, with curly braces we can associate multiple values with a single key
        types {
                AGEP: integer
                CATAGE: integer
        }
}
```

In the database, here is what the above will look like:

```
database             ourstudy.db
checks               age > 100 => age = 100
checks               status = 'married' and  age < 16
input/input file     text_in.csv
input/output table   dataset
```

Observe that the subkeys are turned into a slash-separated list. It is worth getting familiar with this internal form, because when you've made a mistake in your `spec` file, the error message that gets printed in **R** will display your keys in the above form. We will discuss this more later in the tutorial when we talk about running your `spec` file in **R**. Here is a succint summation of the syntax rules for keys:

- You can have several values in curly braces, each on a separate line, which are added to the key/value list. Order is preserved.
- If there is a subkey, then its name is merged with the parent key via a slash; you can have as many levels of sub-sub-sub-keys as you wish.
- As a shortcut, you can replace `key {single value}` with `key:single value`.
- Each key takes up exactly one line (unless you explicitly combine lines; see below). Otherwise, white space is irrelevant to the parser. Because humans will also read the spec file, you are strongly encouraged to use indentation to indicate what is a member of what group.
- If you need to continue a value over several lines, put a backslash at the end of each line that continues to the next; the backslash/newline will be replaced with a single space.

We now continue through our `spec` file to the `input` key:

2.4 Input declarations

The `input` key is an important key in your `spec` file that specifies:

1. The `csv` file from which you will draw your data.
2. The option to *overwrite* the csv file currently written into the specified database with a new csv file that you have specified.
3. The `output table` key that specifies the table where you would like to write the data that is read in from the `csv` file.
4. The `types` key that specifies what type of variables are to be written into the output table.

If this all seems confusing, do not fret. The layout of the `spec` file will make more sense as we continue to explain its various features. Again, more information about these keys can be found in the appendix.

We now discuss the `fields` key in more detail.

2.5 Field declarations

The edit-checking system needs to know the type and range of every variable that has an associated edit. If a variable does not have an associated edit, there is no need to declare its range.

The declaration of the edit variables contained in the `fields` key consists of the field name, an optional type (`int`, `text`, or `real`) to be discussed further below, and a list of valid values. Here is a typical example:

```
fields {
    age: 0−100
    sex: M, F
    hh_type: int 1, 2, 4, 8
    income: real
}
```

In the above code, we've declared four variables: age, sex, hh_type, and income and we've declared those four variables in different and valid ways. Declaring variable types for your field is necessary because the edit-checking system will know what to expect when performing its consistency checks later on. You can see that the list of values may be text or numeric, and the range 0–100 will be converted into the full sequence 0, 1, 2, ..., 100. By declaring our variables with a type and range, we can pass the information to the edit-checking system so that it knows what to verify when running its checks for each of these variables.

When declaring a variable, the first word following the `:` may be a type. For instance, for the `hh_type: int 1, 2, 4, 8` field above, we used the word 'int' to indicate that the data values of the field were of type integer. Notice as well that a type does not necessarily need to be declared; in which case the default action is to treat the entry as plain text (which means that numeric entries get converted to text categories, like `"1"`, `"2"`, ...). Keep in mind that if you declare the incorrect type for a field that the edit-checking system may not correctly verify the values of that field in your data set.

As a final note, we warn the user against creating fields with an overly large range of integer values. For a field with a range of possible integer values, the edit-checking system will verify that each data point falls into one of the possible values specified in the range. This can be a problem for a field with an excessively large range of integer values because each data point will have to be compared against all of the possible values in the range. Though this is easily doable for smaller ranges such as 0-100 or even 0-1000, it becomes extremely time consuming for larger ranges such as 0-600000. Instead, we recommend assigning a field with a large range as a real variable.

Auto-detect If your field list consists of a single star, `*` (the wild-card character in SQL that represents all possible inputs), then the data set will be queried for the list of values used in the input data. All values the variable takes on will be valid; all values not used by the variable will not be valid. Keep in mind that this can present problems if there are errors in your input data set. In any case, using `*` may be useful when quickly assembling a first draft of a spec, but we recommend giving an explicit set of values for production. You may precede the `*` with a type designation, like `age: real *`.

As you know, it is often necessary to impute data points in separate categories given the distribution of the data. For this, we use the `recodes` key.

2.6 Recodes

In short, `recode` keys are just new variables that are deterministic functions of existing data sets based on parameters given by you, the analyst. Based on the variables specified in the `fields` key, you can "recode" those fields into other variables (which can be thought of as categories) based on paramters given in a certain syntax. This is often necessary to ensure that you are imputing your data over an accurate distribution. Observe the following typical example of a `recodes` key:

```
recodes {
    CATAGE {
        0 | age between 0 and 15
        1 | age between 16 and 64
        2 | age > 64
    }
}
```

Here, we have declared a new variable CATAGE whose data points are based off of a deterministic function of the variable AGEP. As we will see later, this recode will be called in the `categories` key during imputation so that the data points we are attempting to impute will be imputed in categories based on their recode values rather than collectively as a single set of data points. The `recodes` key is fairly straightforward, although we will learn about some of its more advanced features later. For now, however, we continue our walkthrough of the `spec` file by discussing the `checks` key.

2.7 Checks

The conistency-checking system is rather complex. Ironically, this complexity is what makes the system efficient and reliable: there are typically a few dozen to a few hundred checks that every observation must pass, from sanity checks like *fail if age < 0* to real-world constraints like *fail if age < 16 and status='married'*. Further, every time a change is made (such as by the imputation system) we need to re-check that the new value does not fail checks. For example, an OLS imputation of age could easily generate negative age values, so the consistency checks are essential for verifying that the imputation process is giving us accurate and useable data points. In addition to error checking we can also use consistency checks for other purposes, such as setting *structural zeros* for the raking procedure.

All of the checks that are to be performed are specified here, in the `checks` key. The checks you specify here will be performed on all input variables, as well as on all imputed data values. Let's take a look at an example `checks` key:

```
checks {
        age < 0
        age > 95 => age = 95
       }
```

In the above example, we've indicated that the consistency checking system should verify that age is not less than 0 and that age is not greater than 95. Notice as well that when specifying that age > 95 we've also included the line 'age = 95' to indicate that when an age data point has a value higher than 95 that we should simply top-code it as 95. We haven't included an auto-declaration for age < 0 because if a data-point has a negative age value than it's indicative of a real error rather that should be properly imputed. It is up to you to decide when it is appropriate to utilize the auto-declaration feature.

We've now introduced the keys that precede the `impute` key. Up to this point, all of the `keys` we've discussed have served the purpose of preparing the data in some way to be imputed in the `impute` key. We now describe its functions below.

2.8 Imputation

The `impute` key is fairly comprehensive and has several sub-keys that fulfill various roles in the imputation process. Many of the values of these sub-keys are derived from values found earlier in the `spec` file; such as the fact that `categories` is based off of the variables declared in `recodes`. Take a look at the following example of an `impute` key that is used to outline the imputation of the age variable described in the above keys:

```
impute {
    input table: viewdc
    min group size: 3
    draw count: 3
    seed: 2332

    categories {
        CATAGE
        SEX
    }

    method: hot deck
    output vars: age
}
```

As you can see, there is quite a bit going on here. Let's walk through each of the sub-keys above and see what they're doing:

- The `input table` sub-key specifies the name of a view table to where you will write your recode variables. Recall that a view table is a digital SQL table that takes up very little memory and is very quick to render. In our example, we've indicated that the name of the view table to where our `recode` variables will be written is called 'viewdc'. We'll be able to access this view table from R later when we actually compile our `spec` file.

- `min group size` indicates the minimum number of known data points that can be used to impute any unknown data points. For instance, it wouldn't make much sense to use a single data point to impute five other data points as they would all end up being the same value. To prevent this, we set `min group size: 3`.

- `draw count` determines the number of multiple imputations that should be performed. Because imputation can be done using stochastic models, it is often beneficial to obtain multiple imputations of the same data set. This will be explained in more detail later. For now, just know that the `draw count` sub-key determines the number of multiple imputations that are performed.

- `seed` specifies the pseudo-random number generator seed that is used to obtain random numbers for stochastic imputation models. Keeping this seed the same will ensure that multiple imputations of the same data set will return the same outputs. Be sure to record the seeds you choose for your imputations so that other analysts can reference your data and replicate your imputations if necessary.

- `categories` specifies which recodes will be used to impute the data set for the given variable. For example, if CATAGE is listed in the `categories` key then the data points that correspond to each of the three possible CATAGE values will be imputed separately. While it's not an absolute requirement that the `categories` key be implemented, most data sets are imputed by categories so you will be using this key for almost all of your imputations.

- `method` specifies the type of model that is used to impute your data points. Choosing this model is itself a non-trivial task and is discussed more thoroughly later on in the tutorial.

- `output vars` specifies the variable whose data points are to be imputed. For this reason you could consider `output vars` as both the 'input' variable as well as the 'output' variable to the model specified in `method`.

More can be found on each of the above keys in the appendix.

2.9 More features

This concludes our walkthrough of a typical `spec` file. By now you should have a basic idea of how a `spec` file is implemented within TEA. Before we continue on to explaining more about imputation, the models available in TEA, and other features that are available within the TEA framework, we will mention four more features of the `spec` file.

Group recodes The first feature that we'll mention is that of `group recodes`. Often in an imputation model, after declaring your `recodes`, it is also necessary to declare sub-recodes that are function of those earlier recodes. For example, suppose you had a `spec` file in which you intended to impute data points for various members of households. Then to perform imputations on

```
recodes {
        EARN: case when WAGP<=0 then 'none' when WAGP>0 then 'black' else NULL
            end
        MOVE: case MIG when 1 then 'moved' else 'stayed' end
        DEG: case when SCHL in ('24','23','22','21','20') then 'degree' else 'non-
            degreed' end
        MF: case SEX when '1' then 'Male' when '2' then 'Female' else NULL end
        REL: case RELP when '00' then 'Householder' when '01' then 'Spouse' \
                        when '02' then 'Child' \
                        when '06' then 'Parent' else NULL end
}

group recodes {
    group id column: SERIALNO
    recodes {
                NP: max(SPORDER)
        NHH: sum(RELP='00')
        NSP: sum(RELP='01')
        NUP: sum(RELP='15')
                HHAGE: max(case RELP when '00' then AGEP end)
                SPAGE: max(case RELP when '01' then AGEP end)
                SPORD: max(case RELP when '01' then SPORDER end)
                HHSEX: cast(round(avg(case RELP when '00' then SEX end)) as integer)
                SPSEX: cast(round(avg(case RELP when '01' then SEX end)) as integer)
    }
}
```

Including Up to this point, we've constructed the entire `spec` file in a single file. Though this approach may be appropriate in some scenarios, it's often the case that you and your colleagues would like to make edits to the same `spec` file and update or contruct it concurrently. To do this, you can simply mark subsidiary files to be included in the main spec file. Doing so allows you to easily combine these subsidiary files into one project through which all of TEA's routines will run.

The syntax for including is quite simple. To include a subsidiary file at a certain point in the `spec` file, simply insert the key `include:` *subsidiary_file_name* at the line in the `spec` file where you would like the contents of the subsidiary file to be inserted. For example, if you have written the consistency checks in a file named `consistency`, and the entire rank swapping configuration was in a file named `swap`, then you could use this parent spec file:

```
database: test.db

include: swap

checks {
    include: consistency
}
```

Note that any subsidiary files that you choose to include in your `spec` file must be in the same directory as the `spec` file or it will not be able to find them.

Flagging for disclosure For a given crosstab, there may be cells with a small number of people, perhaps only one. This feature of TEA allows us to flag those cells for later handling.

Any combination of variables could be a crosstab to be checked, but flagging typically focuses on only a few sensetive sets of variables. Here is the section of the spec file describing the flagging. The `key` list gives the variables that will be crossed together. With `combinations: 3`, every set of three variables in the list will be tried. The `frequency` variable indicates that cells with two or fewer observations will be marked.

We are calling this specific form of disclosure avoidance *fingerprinting*, so after this segment of the spec file is in place, call *doFingerprinting()* from **R** to run the procedure. The output is currently in a database table named *vflags*.

```
fingerprint {
        key{
                CATAGE
                SEX
                PUMA
                HISPF
                ESR
                PWGTP
        }
        frequency: 2
        combinations: 3
}
```

Raking Each record in a data set might have several data items which require imputation for different reasons:

- a record could have several inconsistent items that we must replace
- an otherwise consistent record could have a blank item that we wish to impute.
- a record could have a combination of consistent items that could lead to personal identif ication

Each scenario implies slightly different knowledge about the data, and thus each scenario might require a different imputation method to properly use this knowledge.

An overlay is a secondary data table (or set of tables) that gives information regarding the *reason* for imputation. Using missing data as an example, a simple overlay could have an entry for each item in the data, indicating if that item is missing or not. A more complicated overlay could delineate the type of non-response for each data item.

Raking is a method of producing a consistent table of individual cells beginning with just the column and row totals. For this tutorial, we will use it as a disclosure-avoidance technique for crosstabs. The column sums and row sums are guaranteed to not change; all individual cells are recalculated. Thus, provided the column totals have passed inspection for avoiding disclosure, the full table passes.

The key inputs are the set of fields that are going to appear in the final crosstab, and a list of sets of fields whose column totals (and cross-totals) must not change.

In the `spec` file, you will see a specification for a three-way crosstab between the `PUMA`, `rac1p`, and `catage` variables. All pairwise crosstabs must not change, but any other details are free to be changed for the raking.

```
raking {
        all_vars: puma|catage | rac1p

        contrasts{
                catage | rac1p
                        puma | rac1p
                        puma | catage
        }
}
```

As you have seen a few times to this point, once you have the spec file in place you can call the procedure with one **R** function, which in this case is `doRaking()`. But there are several ways to change the settings as you go, so that you can experiment and adjust.

The first is to simply change the spec file and re-run. To do this, you will have to call `read_spec` again:

```
> read_spec("demo.spec"); doRaking() #Run two commands on a line by ending the first
      with a semicolon
> read_spec("demo.spec"); doRaking() #Hit the up−arrow to call up the previous line.
```

Everything you can put in a `spec` file you can put on the command line. The help system will give you the list of inputs (which will also help with writing a `spec` file), and you can use those to tweak settings on the command line:

```
> ?doRaking #What settings are available?
> doRaking(max_iterations=2) #What happens when the algorithm doesn't have time to
> converge?
```

This concludes our discussion of the `spec` file layout and syntax. At this point, you should be able to implement a basic `spec` file to impute any data set. More information about the keys discussed above and others that were not present in `demo.spec` can be found in the appendix and at the r-forge website.

We now continue our tutorial by discussing imputation in more detail.

3 Imputation

Thus far we've seen how to impute a data set using a single `impute` group. In this form, single imputation produces a list of replacement values for those data items that fail consistency checks or initially had no values. For the case of editing, the replacements would be for data that fails consistency checks; for the case of basic imputation, the replacements would be for elements that initially had no values. Recall that as we stipulated earlier in the tutorial, for our purposes we consider a looser definition of imputation that includes the replacement of data points that both intially had no value as well as those that fail consistency checks.

In a similar vein, while single imputation gives us a single list of replacement values, any stochastic imputation method could be repeated to produce multiple lists of replacement values. For instance, we could utilize multiple imputation to calculate the variance of a given statistic, such as average income for a subgroup, as the sum of within-imputation variance and across-imputation variance.

To give a concrete example, consider a randomized hot deck routine, where missing values are filled in by pulling values from similar records. For each record with a missing income:

1. Find the universe of which the record is a member.
2. For each variable in turn:
    - Make a draw from model chosen for the variable, based on the specific universe from (1).

The simplest and most common example is the randomized hot deck, in which each survey respondent has a universe of other respondents whose data can be used to fill in the respondent's missing values. The hot deck model is a simple random draw from the given universe for the given variable.

Given this framework, there are a wealth of means by which universes are formed, and a wealth of models by which a missing value can be filled in using the data in the chosen universe.

Universes The various models described above are typically fit not for the whole data set, but for smaller *universes*, such as a single county, or all males in a given age group. A universe definition is an assertion that the variable set for the records in the universe was generated in a different manner than the variables for other universes.

An assertion that two universes have different generative processes could be construed in several ways:

- different mathematical forms, like CART vs. logistic regression

- One broad form, like logistic regression, but with different variables chosen (that is, the stochastic form is the same but the structural form differs).
- A unified form, like a logistic regression with age, sex, and ancestry as predictors, with different parameter estimates in each universe.

Universe definitions play a central role in the current ACS edit and imputation system. Here, universes allow data analysts to more easily specify particular courses of action in the case of missing or edit-inconsistent data items. To give an extreme example, for the imputation of marital status in ACS group quarters (2008), respondents are placed in two major universes: less than 15 years of age (U1) and 15 or more years of age (U2). The assertion here, thus, is that people older than 15 have a marital status that comes from a different generative process than those people younger than 15. This is true: people younger than 15 years of age cannot be married! Thus in the system, any missing value of marital status in U1 can be set to "never married", and missing values in U2 can be allocated via the ACS hot-deck imputation system.

Now that we are more familar with universes, we can discuss the various models that are available in TEA and the methodology of choosing the one that is appropriate for the imputation being performed.

4 Models

Now that we're more familiar with the concept of universes, we examine how to choose the model that will give us the best results when imputing the data points in a specific universe. Indeed, given an observation with a missing data point and a universe, however specified, there is the problem of using the universe to fill in a value. Randomized hot-deck is again the simplest model: simply randomly select an observation from the universe of acceptable values and fill in. Other models make a stronger effort to find a somehow-optimal value:

• One could find the nearest neighbor to the data point to be imputed (using any of a number of metrics).

• Income is typically log-Normally distributed, and log of this year's income, last year's income, and if available, income reports from administrative records (ADREC) may be modeled as Multivariate Normal (with a high correlation coefficient among variables). After estimating the appropriate Multivariate distribution for the universe, one could make draws from the distribution.

• If the Multivariate Normal seems implausible, one could simply aggregate the data into an empirical multivariate PDF, then smooth the data into a kernel density estimator (KDE). Draws from a specified KDE can be made as easily as draws from a standard Multivariate Normal.

• Count data, such as the number of hospital visits over a given period, are typically modeled as a Poisson distribution. In a manner similar to fitting a Normal, one could

find the best-fitting Poisson distribution and draw from that distribution to fill in the missing observation.

• Bayesian methods: if ADREC are available, they may be used to generate a prior distribution on a variable, which is then updated using non-missing data from the survey.

• One could do a simple regression using the data on hand. For example, within the universe, regress income on age, race, sex, and covariates from ADREC. Once the regression coefficients are calculated, use them to impute income for the record as the predicted value plus an error term.

• Discrete-outcome models, like the Probit or Logit, could be used in a similar manner.

• To expand upon running a single regression, one can conduct a structured search over regression models for the best-fitting regression.[1] Such a search is currently in use for disclosure avoidance in ACS group quarters.

A unified framework would allow comparison across the various imputation schemes and structured tests of the relative merits of each. Though different surveys are likely to use different models for step (2) of the above process, the remainder of the overall routine would not need to be rewritten across surveys.

We now discuss each of the models that are available in TEA.

4.1 Models of TEA

This section describes the models available for use in imputing missing data.

Hot deck This is randomized hot deck. Missing values are filled in by drawing from nonmissing values in the same subset. The assumption underlying the model is *missing completely at random* (MCAR), basically meaning that nonrespondents do not differ in a systematic way from respondents.

This model has no additional keys or options, although the user will probably want an extensive set of category subsets. Example:

```
impute{
    input table: dc
    min group size: 3
    draw count: 5
    id: serialno
                                                                    905,1
                                                                      76%


  categories {
```

[1] Because the imputation step is not an inference step, such a search presents few conceptual difficulties.

```
        agecat
        sex
    }
    output vars: sex
    method: hot deck
}

impute{
    input table: dc
    min group size: 3
    draw count: 5
    id: serialno

    categories {
        agecat
        sex
    }
    method: ols
                                                                    923,1
                                                                        78%

utput vars: agep
    input vars: rac1p, nativity||sex
}
```

Ordinary least squares (aka regression) This is the familiar $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \epsilon$ form. The parameters (including the variance of ϵ) are estimated for the subset of the category where all variables used in the regression are not missing. For a point where y is missing but $(x_1, x_2, \ldots)$ are not,find $\beta_1 x_1 + \beta_2 x_2 + \ldots$, then add a random draw from the distribution of ϵ.

The variables may be specified via the usual SQL, with two exceptions to accommodate the fact that so many survey variables are categorical.

Unless otherwise noted, all dependent variables are taken to be categorical, and so are expanded to a set of dummies. The first category is taken to be the numeraire, and others are broken down into dummy variables that each have a separate term in the regression. The independent variable will always be calculated as a real number, but depending on the type of variable may be rounded to an integer.

If a dependent variable is numeric, list it with a #, such as `variables: #age, sex`.

An *interaction* term is the product of the variables, where for catgories *product* means the smaller subsets generated by the cross of the two variables, such as a sex-age cross of $(M, 0--18), (F, 0--18), (M, 18--35), (F, 18--35)$; for continuous variables *product* has its typical meaning.

Probit and Logit These models behave like OLS, but are aimed at categorical variables. The output variable is not restricted to twoc categories.

seqRegAIC

Distributions The Normal (aka Gaussian) distribution is the archetype of this type of model. First, estimate the mean and standard deviation using the non-missing data in the category. Then fill in the missing data via random draws from the Normal distribution with the calculated parameters. You may use either `method: normal` or `method: gaussian`.

For other situations, other distributions may be peferable. For example, income is typically modeled via `method: lognormal`. Count data may best be modeled via `method: poisson`.

Hot deck is actually a fitting of the Multinomial distribution, in which each bin has elements in proportion to that observed in the data; `method: hot deck` and `method: multinomial` are synonyms.

The *method: multivariate normal* doesn't work yet.

The distribution models have no additional options or keys.

Kernel smoothing A kernel density estimate overlays a Normal distribution over every data point. For example, if a data set consisted of two data points at 10 and one at 12, then there would be a large hump in the final distribution at 10, a smaller hump at 12, and the distribution would be continuous over the entire space of reals.

Thus, kernel smoothing will turn a discrete distribution consisting of values on a few values into a continuous distribution.

Invoke this model using either `method: kernel` or `method: kernel density`.

Internals The Census Bureau often deals with tables that would have 1.5 million cells if written out in full, but where only maybe 20,000 surveys are in hand. Therefore, the algorithm is heavily oriented toward sparse data.

It begins with a long and tedious routine to write SQL to generate the set of possibly-nonzero values, as per the example above. SQL is the appropriate language for generating this list because it is optimized for generating the cross of several variables and for pruning out values that match our criteria. The tedium turns out to be worth it: our test data set takes about 25 seconds to run using the original full-cross '72 algorithm, and runs in under two seconds using the SQL-pruned matrix.

Recall that we had briefly discussed multiple imputation in the `impute` section above. We now discuss this in more detail.

5 Multiple Imputation

A single imputation would produce a list of replacement values for certain data items. Any stochastic imputation method could be repeated to produce multiple lists of replacement values. Variance of a given statistic, such as average income for a subgroup, is then the sum of within-imputation variance and across-imputation variance.

- For each imputation:

  - fill in the data set using the given set of imputed values
  - calculate the within-imputation variance
- Sum the across-imputation and average within-imputation variances to produce an overall variance figure that takes into account uncertainty due to imputation.

The question of what should be reported to the public from a sequence of imputations remains open. The more extensive option would be to provide multiple data sets; the less extensive option would be to simply provide a variance measure for each data point that is not a direct observation.

Interface Figure **??** shows a (slightly abbreviated) configuration file describing the hot deck process for a variable in the SIPP. It is intended to be reasonably readable, and maintainable by an analyst who is a statistician but not a programmer.

Lines 11–18 of the configuration specify the categories used for step (1) above: draws are made from the universe of records with an age in the same age category as the record to be filled in and `num_sipp_jobs_2008` in the same category as well.

Of course, different surveys would have different classification schemes, but this means that each survey would need a new configuration file, not new code.

Line eight indicates that five imputations are to be done for each missing value. Those with extensive experience with multiple imputation often advise that a handful of imputations are sufficient for most purposes.

The sample from Figure **??** focused on the determination of categories in which to do imputation. Figure **??** focuses on regression models that go beyond the simple randomized hot deck of Figure **??**. Lines 5–8 specify the variables that need imputation, and the form of model to be used. The current system will search the set of models of the given form for the one that best fits the known data; Lines 9–14 show the list of variables that could be used as explanatory variables, although a typical model will likely wind up using only around four or five.

Edits The system as written includes a component that checks consistency against a sequence of edits. In line three of the sample spec file of Figure **??**, the `flagearn` variable is declared to have possible values of 0, 1, 3, or 4, but line four specifies that if an imputation returns 4, then it is rejecte d. The imputation routine sketched above does not need to include any edit rules, because this edit step will take those into account; the separation of edits and imputations simplifies the routine.

```
1
2
3
4
5
6
7
8
9
10
```

```
database: sipp.db

|flagearn 0, 1, 3, 4
flagearn = 4

impute_by_groups {
    min_group_size {20}
    iteration_count {5}
    datatab {sippdata}
```

```
        categories {
            15<=agesipp200812<18;
            18<=agesipp200812<22;
            22<=agesipp200812<40;
            40<=agesipp200812<62;
            62<=agesipp200812;
            num_sipp_jobs_2008 = 0;
            num_sipp_jobs_2008 = 1;
            num_sipp_jobs_2008 => 2;
     }

        imputes{
            flagearn~ flagearn;
        };
    }
```

```
database: acs2008.db

impute{
    seqRegAIC{
        vars{
          TI{ model: gam }
            DIS{ model: multinom }
        }
        predlist{ #Sample of predictors that could used for regressions
            SEX; YOE; WKL; MIL; UR;
            SCH; RCGP; POV; SS; MAR;
            LANX; JWTR; TYPGRP; GQINST; FER;
            ESR; DIS; COW; CIT; OCC2;
        }
    }
}
```

6 Interactive R

The specification file described to this point does nothing by itself, but provides information to procedures written in **R** that do the work. In fact, TEA is simply called as a library in **R**, and the `spec` file itself is instantiated by issuing commands from the **R** command prompt described below.

6.1 Loading TEA in R

When you start R (from the directory where the data is located), you are left at a simple command prompt, >, waiting for your input. TEA extends R via a library of functions for survey processing, but you will first need to load the library, with:

```
> library(tea)
```

[You can cut crimson-bordered code blocks and paste them directly onto the **R** command line, while blue-bordered blocks are spec file samples and would be meaningless typed out at the R command prompt.]

Now you have all of the usual commands from **R**, plus those from TEA. You only need to load the library once per **R** session, but it's harmless if you run `library(tea)` several times.

After loading the library by running `> library(tea)`, you would then need to tell **R** to read your `spec` file, perform the checks, perform the imputations, and then finally check out the imputations to an **R** data structure so that you can view them. Observe the following code:

```
> library(tea)
> read_spec("spec")
> doChecks()
> doMImpute()
> checkOutImpute()
```

These commands could be entered in a script file as easily as they're entered on R's command line. You always have the option of creating a `.R` file that has each of the above commands listed sequentually. Observe the following example of a `.R` file:

```
library(tea)
library(ggplot2)
readSpec("demo.spec")
doChecks()
doMImpute()
checkOutImpute()
```

If we assume that the file above is named `demo.R` then we could run the following command from **R**'s command line:

```
> source("demo.R")
```

Then **R** would automatically run each of the scripts specified in `demo.R` and would accomplish the exact same thing as running each command separately through **R**'s command line. Though running `> source("your_file.")` is often quicker and more convenient than entering each command separately on the command line, entering the commands separately can often aid in verifying the results of the consistency-checking step, verifying the results of the imputation, et cetera.

In either case, once your `spec` file has been correctly implemented, your data will be imputed and available for viewing through an **R** data-frame. We examine how this is done in the next subsection.

6.2 Showing the data

Data is stored in two places: the database, and R data frames. Database tables live on the hard drive and can easily be sent to colleagues or stored in backups. R data frames are kept in R's memory, and so are easy to manipulate and view, but are not to be considered permanent. Database tables can be as large as the disk drive can hold; R data frames are held in memory and can clog up memory if they are especially large.

You can use TEA's `show_db_table` function to pull a part of a database table into an R data frame. You probably don't want to see the whole table, so there are various options to limit what you get. Some examples:

```
> teaTable("dc", cols="AGEP, PUMA", where="PUMA=104")
> teaTable("dc", cols="AGEP, PUMA", limit=30, offset=100)
```

The first example pulls two columns, but only where `PUMA=104`. The second example pulls 30 rows, but with an offset of 100 down from the top of the table. You will probably be using the `limit` often; the `offset` allows you to check the middle of the table, if you suspect that the top of the table is not representative.

In fact, there are still more options. Rather than listing them all here, you can get them via R's help system:

```
> ?teaTable
```

This `?name` form should give you a help page for any function, including TEA functions like `?doRaking` or `?doMImpute`. (Yes, TEA function documentation is still a little hit-and-miss.) Depending on R's setup, this may start a paging program that lets you use the arrow keys and page up/page down keys to read what could be a long document. Quit the pager with `q`.

The `show_db_table` function creates an R data frame, and, because the examples above didn't do anything else with it, displays the frame to the screen and then throws it out. Alternatively, you can save the frame and give it a name. R does assignment via `<-`, so name the output with:

```
> p104 <- teaTable("dc", cols="AGEP, PUMA", where="PUMA=104")
```

To display the data frame as is, just give its name at the command line.

```
> p104
```

But you may want to restrict it further, and R gives you rather extensive control over which rows and columns you would like to see.

Another piece of R and spec file syntax: the # indicates a comment to the human reader, and R will ignore everything from a # to the end of the line.

```
> p104[1,1] #The upper-left element
> p104[1,"AGEP"] #The upper-left element, using the column name
> p104[ ,"AGEP"] #With no restriction on the rows, give all rows---the full AGEP vector
```

```
> p104[17, ] #All of row 17

> minors <- p104[, "AGEP"] < 18
> minors #A true/false vector showing which rows are under 18.
> p104[minors, ] #You can use that list of true/falses to pick rows. This gives all rows under
      18yo.
```

These commands could be entered on R's command line as easily as in a script file, so an analyst who needs to verify the results of the consistency-checking step could copy and paste the first three lines of the script onto the R command prompt, where they will run and then return the analyst to the command prompt, where he or she could print subsections of the output tables, check values, modify the spec files and re-run, continue to the imputation step, et cetera.

7 Conclusion

This concludes our tutorial. If you have any questions or comments please feel free to contact us at xxxxxxxxx@email.gov. Also, check out the R-Forge Website at `https://r-forge.r-project.org/projects/tea/` for updates to both the TEA software as well as this tutorial document.

Appendix: keys

This is a reference list of all of the available keys that could appear in a spec file. As a reference, descriptions are brief and assume you already know the narrative of the relevant procedures, in the main text.

Keys are listed using the `group/key/value` notation described in Section 2.1. As described there, one could write a key as either

```
group {
    key : value
}

#or as
group {
    key {
        value
    }
}
```

Here are the keys, in alphabetical order.

raking/thread count:
You can thread either on the R side among several tables, or interally to one table raking. To thread a single raking process, set this to the number of desired threads.

input/primary key:
A list of variables to use as the primary key for the output table. In SQLite, if there

is only one variable in the list as it is defined as an integer, this will create an integer primary key and will thus be identical to the auto-generated ROWID variable.

group recodes/recodes:
A set of recodes like the main set, but each calculation of the recode will be grouped by the group id, so you can use things like `max(age)` or `sum(income)`. Returns 0 on OK, 1 on error.

raking/tolerance:
If the max(change in cell value) from one step to the next is smaller than this value, stop.

input/types:
A list of *keys* of the form: var: type where var is the name of a variable (column) in the output table and type is a valid database type or affinity. The default is to read in all variables as character columns.

id:
Provides a column in the data set that provides a unique identifier for each observation. Some procedures need such a column; e.g., multiple imputation will store imputations in a table separate from the main dataset, and will require a means of putting imputations in their proper place. Other elements of TEA, like flagging for disclosure avoidance, use the same identifier.

rankSwap/max change:
maximal absolute change in value of x allowed. That is, if the swap value for x_i is y, if $|y - x_i| >$ maxchange, then the swap is rejected
default = 1

raking/all vars:
The full list of variables that will be involved in the raking. All others are ignored.

impute/draw count:
How many multiple imputations should we do? Default: 5.

rankSwap/seed:
The random number generator seed for the rank swapping setup.

impute/earlier output table:
If this imputaiton depends on a previous one, then give the fill-in table from the previous output here.

impute/input table:
The table holding the base data, with missing values. Optional; if missing, then I rely on the sytem having an active table already recorded. So if you've already called `doInput()` in R, for example, I can pick up that the output from that routine (which may be a view, not the table itself) is the input to this one.

impute/output table:
Where the fill-ins will be written. You'll still need `checkOutImpute` to produce a completed table.

impute/seed:
The RNG seed

raking/contrasts:
The sets of dimensions whose column/row/cross totals must be kept constant. One contrast to a row; pipes separating variables on one row.

```
raking{
    contrasts{
        age | sex | race
        age | block
    }
}
```

input/indices:
Each row specifies another column of data that needs an index. Generally, if you expect to select a subset of the data via some column, or join to tables using a column, then give that column an index. The `id` column you specified at the head of your spec file is always indexed, so listing it here has no effect.

rankSwap/swap range:
proportion of ranks to use for swapping interval, that is if current rank is r, swap possible from r+1 to r+floor(swaprange*length(x))
default = 0.5

raking/count col:
If this key is not present take each row to be a single observation, and count them up to produce the cell counts to which the system will be raking. If this key is present, then this column in the data set will be used as the cell count.

input/input file:
The text file from which to read the data set. This should be in the usal comma-separated format with the first row listng column names.

recodes:
New variables that are deterministic functions of the existing data sets. There are two forms, one aimed at recodes that indicate a list of categories, and one aimed at recodes that are a direct calculation from the existing fields. For example (using a popular rule that you shouldn't date anybody who is younger than (your age)/2 +7),

```
recodes {
    pants {
        yes | leg_count = 2
        no | #Always include one blank default category at the end.
    }
    youngest_date {
        age/2 + 7
    }
}
```

You may chain recode groups, meaning that recodes may be based on previous recodes. Tagged recode groups are done in the sequence in which they appear in the file. [Because the order of the file determines order of execution, the tags you assign are irrelevant, but I still need distinct tags to keep the groups distinct in my bookkeeping.]

```
recodes [first] {
    youngest_date: (age/7) +7 #for one−line expressions, you can use a colon.
```

```
    oldest_date: (age −7) ∗2
}
recodes [second] {
    age_gap {
        yes | spouse_age > youngest_date && spouse_age < oldest_date
        no |
    }
}
```

If you have edits based on a formula, then I'm not smart enough to set up the edit table from just the recode formula. Please add the new field and its valid values in the fields section, as with the usual variables. If you have edits based on category-style recodes, I auto-declare those, because the recode can only take on the values that you wrote down here.

raking/input table:
The table to be raked.

input/missing marker:
How your text file indicates missing data. Popular choices include "NA", ".", "NaN", "N/A", et cetera.

timeout:
Once it has been established that a record has failed a consistency check, the search for alternatives begins. Say that variables one, two, and three each have 100 options; then there are 1,000,000 options to check against possibly thousands of checks. If a timeout is present in the spec (outside of all groups), then the alternative search halts and returns what it has after the given number of seconds have passed.

raking/max iterations:
If convergence to the desired tolerance isn't achieved by this many iterations, stop with a warning.

input/output table:
The name of the table in the database to which to write the data read in.

database:
The database to use for all of this. It must be the first thing on your line. I need it to know where to write all the keys to come.

raking/run number:
If running several raking processes simultaneously via threading on the R side, specify a separate run_number for each. If single-threading (or if not sure), ignore this.

input/overwrite:
If `n` or `no`, I will skip the input step if the output table already exists. This makes it easy to re-run a script and only sit through the input step the first time.

group recodes:
Much like recodes (qv), but for variables set within a group, like eldest in household. For example,

```
group recodes {
    group id : hh_id
    eldest: max(age)
    youngest: min(age)
```

```
    household_size: count(*)
    total_income: sum(income)
    mean_income: avg(income)
}
```

raking/structural zeros:
A list of cells that must always be zero, in the form of SQL statements.
input/primarky key:
The name of the column to act as the primary key. Unlike other indices, the primary key has to be set on input.
group recodes/group id:
The column with a unique ID for each group (e.g., household number).