# Tea for survey processing: a tutorial

Ben Klemens, Rolando Rodríguez, and David Vernet

May 5, 2015

Raw data is often rife with missing items, logical errors, and sensitive information. To ignore these issues risks alienating respondents and data users alike, 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 via a framework that is user-friendly, effectively handles Census-sized data sets, and processes quickly even with relatively sophisticated methods.

**Some sample use cases**

- A record is missing age, so it gets sent to the imputation module to draw values of age given schooling (if available for the record) and neighborhood characteristics. Did the imputation generate a married 12-year old? Each draw is sent to the editing module to verify every consistency check provided by the user, and strike out failures.
- One record might be for a 110-year old. This is distinctive enough that publishing a table of incomes by age might publicly reveal this person's income. Tea provides a module for finding distinctive cases and blanking out the most distinctive characteristic. Of course, the record then needs to be sent to imputation, and the imputed value(s) need to be edited.
- Crosstabs giving age $\times$ race $\times$ income and are cleared for disclosure concerns, but users want to use statistical techniques based on record-level data, not crosstabs. Tea's raking module generates a synthetic data set that is as close as possible to the constraint of the crosstab (and, of course, meets all constraints).

You can also read this tutorial in your browser at `http://b-k.github.io/tea-tutorial/`.

- Section 1 presents a 'hello, world' example to introduce the basic mechanics of the system, and the format of the spec file used to describe a survey.
- Section 2 describes Tea's syntax for generating new variables based on other variables, like `is_heterosexual_marriage` calculated from the sex of the spouses, or log of income calculated from income.
- Section 3 goes into greater detail on the procedure for imputing missing values, the specific models available, and the use of multiple imputation.
- Section 5 describes *editing*, the process of finding elements that fail validity checks.

- Section 6 describes a table-based method of finding respondents at risk of being identifiable, and blanking out portions of their information so an imputed value can be filled in.
- Section 7 describes Tea's mechanism for generating synthetic microdata with marginal values identical to the source data. This is useful for running bootstrap-style tests and protecting the identities of individual respondents.
- The Appendix in Section 8 provides some other useful information, including some notes on missingness markers in various systems and a suggestion on how to cite Tea in your papers.

**This tutorial is a work in progress.** Tea has an open development model, wherein users have been able to download versions almost since its inception. We follow the same process for this tutorial, posting the draft as we write it, and adding more every week.

**Installing Tea** The interface for Tea is an R package, but depends on several C libraries. That means that installation is not as trivial as typical R packages, but we found the large-scale processing we had to do to be largely impossible when staying exclusively within ecosystem of things installable via `install.package`.

If you are set up to use virtual machines, check out the GovReady Tea machine[1].

Installation is easiest on a system with a package management system. Package managers give you immediate access to thousands of programs, libraries, language compilers, games, and everything else a computer can do. Linux users know their package manager; Mac users can use MacPorts[2] or Fink[3]; Windows users can use Cygwin[4].

Once you have the basic platform set up,

- Install the Apophenia library[5]
- Install the R-to-Apophenia link, Rapophenia[6]
- Install Tea[7]

# 1 System basics

The full specification of the various steps of Tea's process, from input of data to final production of output tables, is specified in a single plain text file, herein called the *spec file*. An R script executes the plan based on the spec file description.

There are several benefits to such a work flow. 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 analysts whose areas of expertise are not in programming can write technical

[1] `https://github.com/GovReady/govready-Tea`
[2] `http://macports.org`
[3] `http://finkproject.org`
[4] `http://cygwin.com`
[5] `http://apophenia.info/setup.html`
[6] `https://github.com/b-k/Rapophenia`
[7] `https://github.com/rodri363/tea`

specifications without the assistance of a programmer or training in a new programming language. Second, this follows the programming adage that data and procedure should be kept separate. Third, we hope that spec files will be published along with data sets, so that users can see the assumptions made during processing.

Throughout this tutorial, code snippets that have a file name can be found in the tutorial repository[8]. In the sidebar on that page, you will find a link to download a zip file of the full set of files, or users familiar with git can clone the repository.

Being plain text, you can open and edit the files using any text editor (Notepad, Kate, vi, ...). From that directory, you can cut/paste the R sample code onto the R command prompt, or run the program using R's the `source` command, e.g., from R:

```
source("hello.R")
```

This leaves you at the R prompt with the Tea library loaded and any variables defined in `hello.R` ready for your interrogation.

If you want to run a script and exit, you can do this via R's `-f` flag. From the command line:

```
R -f hello.R
```

**Hello** Our first example will load in a data set (the Public Use Micro Sample for the American Community Survey, DC, 2008) and do some simple queries, without yet doing any editing or imputation. Here is the spec file:

```
database: test.db

input {
    input file: dc_pums_08.csv
    output table: dc
    overwrite: yes
}

fields {
    age: real
    schl: int 1-24
}
```

Almost everything in the spec file will consist of key:value pairs, which may be organized into groups (and subgroups).

Everything on a line after a # is a comment that the spec parser will ignore. Blank lines are also ignored.

[8] `https://github.com/b-k/tea-tutorial.git`

The first non-comment, non-blank line of every spec file needs to specify a database; more on this requirement in Section 1.1. This will be a file generated on disk (or opened, if it already exists), using the SQLite database library. The name doesn't have to end in `.db`, but we recommend sticking to that custom. If you are an SQLite expert, you can view or modify this database using your favorite SQLite tools.

If your data is already in an SQLite database, then you can skip the `input` step, but in typical cases your data will be in a plain text file. These are typically called CSV files (for comma-separated values), though the delimiter can be set to almost anything; we prefer pipes, | .

The `input` segment of the spec gives the name of the input file, and the name of the table that will be generated from that input.

This spec file tells Tea to overwrite the `dc` table if it already exists. This can be useful for running an analysis from start to finish, but in most cases, the read-in is time-consuming and most later operations will not modify the source data, so the default if the input segment has no `overwrite` key is to assume `overwrite: no`. More on this below.

The spec also includes a section describing two of the fields: `age` is a real number, and `schl` is an integer from 1 to 24. Not all fields have to be specified, but it helps to guarantee that SQLite doesn't treat the entries as text, and will be required when we get to editing. You can have one `fields` section for every table, or several which will be treated as a single unit.

Spacing in the spec file is largely irrelevant, but each key:value pair must end with a newline (or the } closing a group, though we think having the } on its own line looks better). You will see some examples below where the expression is too long for one line, so lines are joined by ending the first line with a backslash.

Now that we have specified the characteristics of the survey, we can move on to `hello.R`, which makes use of that description:

```
library(tea)
readSpec("hello.spec")
doInput()

tt <- teaTable("dc", limit=20, cols=c("serialno", "agep", "schl"))
print(tt)

print(summary(teaTable("dc", cols=c("agep", "schl"))))
```

- The first line, `library(tea)` loads Tea. All functions defined therein can now be used at the R command line.
- The second line reads the spec file, `"hello.spec"`.
- At this point, the database is open, and Tea knows about all of the configuration options in the spec file. Thus, the `doInput()` function does not need input arguments specifying the name of the text file or other details—that can all be read off from the spec.

- After `doInput()` runs, the requested table, `dc`, exists in the database. The `teaTable()` function pulls data from the database into an R data frame. In the example here, the first use of `teaTable` will print to the screen, so limit the number of columns to pull to 20, and request only three columns.
- The second example uses R's `summary()` function to show the means and medians of the age and schooling categories.

We do not print the unremarkable output from running `hello.R`, but you are encouraged to make sure that your Tea installation is working by running it yourself.

### 1.1 Subkeys

Why does the first line of the spec file have to be a database? Because the first step in processing is to convert the spec file into a database table. The table is named `keys`, and you can view it like any other:

```
teaTable("keys")
```

```
                 key   tag count          value
1           database 0 tag     1        test.db
2   input/input file 1 tag     2 dc_pums_08.csv
3 input/output table 1 tag     3             dc
```

The format of the `keys` table is subject to change, so please do not rely on it for processing. However, it can be useful to check that the parser did what you expected. The `fields` and `checks` sections do not get encoded into this table, but every other key:value pair will get one row. Comparing this table to `hello.spec` shows how the keys embedded in a group are written via a `group/key` notation. For example, `input/input file` appeared in the spec as

```
input {
    input file: ...
}
```

That said, here is an alphabetical list of all Tea keys[9].

## 2 Recodes

A *recode* is a variable that is deterministically generated from another variable. The name comes from variables made via simple code switches, like taking three-digit race codes (100, 101, ..., 200, 201, ...) and reducing them to one-digit race codes (1, 2, ...). But they are a versatile solution to many little survey annoyances. If you find yourself referring to a complex expression several times, it might be best to define a variable

[9] `http://rodri363.github.io/tea/keys.html`

based on that expression. It is easy to use a recode to define sets of observations that can then be treated as separate categories for imputation modeling.

There may be characteristics of the group, such as the householder's income and the total income for all members of the household, that are useful for processing each record. Tea handles such characteristics by adding a field giving the group-based value to each record.

On the back-end, the recodes are generated via an SQL table view. We have a table named `dc`, so the recodes will be in a table named `viewdc`. Therefore, all recodes are simple fragments of SQL.

Here is a sample spec file that generates several recodes.

---

```
database: test.db
id: id

input {
    input file: dc_pums_08.csv
    output table: dc
}

fields {
    agep: real
    PINCP: real
    log_in: real
    schl: int 1−24
    has_income: int 0, 1
}

recodes {
    id: serialno∗100 + sporder
    log_in: log10(PINCP+10)
}

group recodes {
   group id: serialno
   hh_in: max(case sporder when 1 then log_in else 0 end)
   house_in: sum(case when PINCP is not null \
                          and PINCP > 0 then PINCP else 0 end)
}

recodes {
    log_house_in: log10(house_in)

    has_income {
        0 | PINCP==0
        1 | PINCP>0
    }

    age_cat {
```

```
            0 | AGEP <= 15
            1 | AGEP between 16 and 29
            2 | AGEP between 30 and 64
            3 | AGEP>= 65
            X |
        }
}

checks {
    has_income=0 and PINCP is null => PINCP=0

    PINCP +0.0 < 0
}
```

- For imputation, we will need a column with a unique identifier, so Tea can record which imputed value goes with which record. Census household data usually gives a household number (here, `serialno`) and a household member number (here, `sporder`), so generate a unique ID by joining the two together.
- Income in US dollars is given by the `PINCP` variable. Because it is often easier to work with log of income, generate a variable with log of income. The `+10` prevents log-of-zero errors.
- The next recode segment works in groups. Each value of the given `group id` is treated as a unit, and the given value is written down for every member of the group. So after the recode step, every observation will have a `hh_in` field giving the householder's income and a `house_in` field giving the total income for the household.
- Calculate the householder income by taking the maximum over PINC for the householder (i.e., the person for whom `sporder==1`) and zero for everybody else. We use this form often to pull the characteristics of one household member.
- If you think the SQL syntax for case statements[10] is awkward, we agree with you. Watch out whether there is a variable name after the `case` keyword [`case pinc when ...`] or not [`case when ...`].
- A recode segment can not refer to fields generated in that recode segment. However, subsequent recode segments can refer to variables generated in previous segments.
- Because SQL `case` statements are difficult to write, but recodes with an if-then format are so common, Tea provides a special syntax. The `has_income` variable will be zero if `PINCP` is zero, and will be one in all other cases. The `age_cat` recode behaves similarly: if `AGEP <=15`, then `age_cat` is set to zero, and so on. The `between` keyword is standard SQL, and is inclusive of the endpoints of the range. The default of `X` will be assigned for any negative or NaN values.
- The spec could also have several `group recode` segments. For example, one segment could give state averages and one could give county averages.

The R script uses some of those recodes.

[10] `http://stackoverflow.com/questions/4622/sql-case-statement-syntax`

```
library(tea)
readSpec("recode.spec")
doInput()

tt <- teaTable("viewdc", limit=20, cols=c("id", "PINCP", "log_in",
                                          "hh_in", "house_in"))
print(tt)


print(summary(teaTable(teaenv$active_tab, cols=c("log_in", "log_house_in"),
                                    where="log_in+log_house_in > -100")))

print(dbGetQuery(teaenv$con, "select avg(log_house_in) from \
                              (select distinct serialno, log_house_in \
                              from viewdc where log_house_in > -100)"))
```

- The `doInput` function does the recodes, so there is no need for a separate `doRecodes` function.
- We can treat the new fields exactly like we do the ones supplied in the input data.
- Tea keeps a few variables in an R environment named `teaenv`. The name of the table after the last step is stored in `teaenv$active_tab`. If there were no recodes, this would be `dc`; because there were recodes, this is `viewdc`. More on tracking the flow of tables below.
- The script gets the average log income and average `log_house_in`, taking care to add a condition to only get those records where that sum is not `-Inf`. But the average of `log_house_in` is an unusual statistic, because it goes per capita: a household of seven counts seven times as much as a household of one.
- The last line of the script gets an average using one value per household. It uses the `dbGetQuery` function from the `RSQLite` library to directly query the database. The function needs a database connection to talk to, and that too is in the `teaenv` environment, stored as `teanev$con`.

**The flow of inputs and outputs** With a few exceptions, every segment can have `input table`, `overwrite`, and `output table` keys. Keeping track of these things will become important as more segments are added and if multiple tables are being used.

If these keys are not given, the output table from the previous segment of the spec will be used as the input table for the next segment.

If you do not specify the output from the recodes, then the output table name at the end of a sequence of recodes is the input table name with `view` at the front. E.g., `intab` → `viewintab`.

The imputation uses `filled` as the default output table name, but you are encouraged to use something else.

The flow is thus typically a chain of steps, like input → recode → join with another table → impute. If you give an instruction to overwrite input every time, then the recode, join, and impute steps will also rebuild their output tables. If you remove the input table (SQL: `"drop table read_in_data"`) then the table will be re-read and all subsequent steps rerun. The exceptions: `checks` and `fields` apply to all tables, the `input` segment takes an `input file`, and the imputations are always re-run.

As above, the R variable `teaenv$active_tab` keeps track of the latest output table.

# 3 Imputation

This section will begin with an overview of the many methods one could use to fill in a blank in a survey, and will then show how Tea implements them.

The first are what we will call mechanical edits, which are deterministic if-then rules. For example, if a birth date is given but an age is not, then fill in the age by calculating the time span between the birth date and the survey.

**Mechanical edits** Mechanical methods are best when there is a single correct answer that can be calculated from available information, such as in an accounting ledger where certain sets of numbers must add up to certain sums.

Here is a simple case from the example below:

```
has_income=0 and PINC is null => PINC=0
```

This reads as an if-then statement: if `has_income=0 and PINC is null`, then set `PINC=0`.

People at extreme ends of a range, such as the exceptionally long-lived or the exceptionally high-income, are a disclosure avoidance risk, so statistical agencies often *top code*, changing all values above a threshold to the threshold itself. To set the income of all people making over a million a year to exactly a million:

```
PINC>1e6 => PINC=1e6
```

Some history and literature: Pierzchala [1990] gives an overview of the problem of editing (including a useful glossary). The models covered in that paper are uniformly in the class of mechanical edits, including several implementations of the method of Fellegi and Holt [1976] and deterministic nearest-neighbor methods, such as that used by Garcia [2003].

This thread has continued into the present day. Chen et al. [2002] discusses three newer systems, each of which focuses on determining the cause of the error and imputing accordingly: the editing system for the American Community Survey (ACS, from the US Census Bureau) consists of a long sequence of if-then rules that specify what substitutions are to be made given every type of failure; the DISCRETE system [Winkler, 1995] uses Fellegi-Holt's method to derive imputation rules from the edits; what is

now CANCEIS [Bankier et al., 2002] uses a relatively sophisticated nearest-neighbor rule for imputation.

**Probabilistic edits** Probabilistic edits assert some underlying distribution or model to the variables, and fill in a missing value by making a draw (sometimes multiple) from the underlying distribution.

The incumbent method is *Hot Deck*, a nonparametric method in which nearby non-missing values are used to generate imputed data [Cresce et al., 2005]. In its simplest form, Hot Deck imputation for a missing value consists of substituting a neighbor's value for the missing value. This is a mechanical imputation: if the value is missing, then fill in using the neighbor's value.

Randomized Hot Deck is probabilistic: select a subset of similar respondents (see below), and assume that the missing value takes on the same distribution as the values in the respondent set. Having established this empirical distribution of values, make a draw (or draws) to fill in the missing value.

Hot Deck does not make assumptions about mathematical relations among variables. For example, there is no need to evaluate and substantiate claims that variables are multivariate Normal, or there is a linear relationship between some set of dependent variables and independent variables. But Hot Deck does make the basic modeling assumption that neighbors are similar, and that values observed in the neighborhood are the most likely values that would have been observed had the missing data been gathered.

Other models make other assumptions. We may assume that income has a Lognormal distribution, in which case we can calibrate the distribution by calculating $\mu$ and $\sigma$ using the known values from the respondent set, and then drawing from the estimated Lognormal distribution.

The *EM algorithm* described below builds and draws from a multivariate empirical distribution, akin to the univariate Hot Deck distribution.

**Categories** There is a broad intuition that imputed values should be taken from records 'close' to the one missing information, perhaps in the physical sense, or in a broader sense, like how we expect an apartment dweller to be more like another apartment dweller than a similar person living in a single-family house, *ceteris paribus*.

Tea thus provides a mechanism to do imputations by categories. Given this part in the `impute` segment of a spec file,

```
categories {
        income_cat
        sex
        puma #public use microdata area
    }
```

and a record with a known `income_cat`, `sex`, and `puma`, only records with the same characteristics will be used to train or fit the model.

These categories 'roll up' if there is insufficient data. If there are not enough observations in the `income_cat` $\times$ `sex` $\times$ `puma` category to build a model, then the

last categorization in the list is thrown out, and the model fit using `income_cat` × `sex`. The roll-up could continue until there are no categories left, in which case the entire data set is used. Set the minimum group size before a roll-up occurs via an `impute/min group size` key.

**Imputing incomes** The following example imputes missing incomes using two methods.

Here is the spec file, with several imputation methods:

```
include: recode.spec

impute {
    input table: viewdc
    vars: has_income
    categories {
        age_cat
        sex
        puma #public use microdata area
    }
    method: hot deck
    output table: has_in
}

common {
    input table: viewdc
    earlier output table: has_in
    min group size: 5
    subset: agep+0.0>15

    categories {
        has_income
        age_cat
        sex
        puma
    }
}

impute {
    vars: PINCP
    paste in: common
    method: hot deck
    output table: hd
}

impute {
    vars: PINCP
    paste in: common
    method: lognormal
```

```
    output table: norm
}
```

---

The spec uses two features to reduce redundancy.

- At the top of the file, the `include` line tells the parser to paste the full contents of the listed file into the current file at this point. We can thus reuse all of the declarations and consistency checks listed in `recode.spec` above.
- The `paste in` lines do a similar thing, but for groups. In this case, `paste in: common` instructs the parser to find the group defined to that point with the name `common` and insert all key/value pairs defined in that group at this point. This makes it easy to write two imputations that have most of their specification in common.

Please note: this is a tutorial consisting of simple demo code, so we make no promises that this is a good model for imputing missing data. We do not know of any Census surveys that use these models for income imputation. But we do believe that the spec format makes the assumptions underlying the imputation clear, and provides an easy means of quickly trying different models.

After every value is imputed, the edits are checked. Above, you saw that `recodes.spec` has an edit specifying that if `has_income=0 and PINC is null` then set `PINC` to zero. This guarantees us that at the end of the imputation of `has_income`, the only people with incomes marked as missing are those who are imputed as having a nonzero income.

**Check out any time you like** Tea is intended for multiple imputation, either in the formal sense (see, e.g., Schafer [1997]) or in the sense of this example, where multiple models are used on the same data set.

To facilitate this, imputations are not written to the `input table` specified in the `impute` section, but to a separate `output table`. For the `has_income` variable, the output table was `has_in`, so after the imputation, we can have a look at that table:

```
teaTable("has_in", limit=5)
```

```
  draw value       id      field
1    0     1   348902 has_income
2    0     0  6462602      PINCP
3    0     0  6462602 has_income
4    0     0 18435903      PINCP
5    0     0 18435903 has_income
```

Each id/field coordinate in the original table that has an imputed value will have a row (or rows) in the fill-in table. Here only one draw was made per imputed value, so the `draw` column is all zeros; if you specify `draw count` to be greater than one, then this will take appropriate values.

This is the output for the imputation of `has_income`, but because the edit rules specify that `(has_income=0 and PINCP is NULL)` is an inconsistent value, the fill-in table also specifies cases where PINCP has to be adjusted to maintain consistency.

The list of imputations is sometimes useful for debugging, but we actually want the complete data set. Check out an imputation into a new database table as needed:

```
checkOutImpute(teaenv$active_tab, "complete_tab", filltab="has_in")
outdata <- teaTable("complete_tab")
```

One imputation often follows another. Here, the model of income depends on the `has_income` variable, so we have to impute `has_income` first. Imputations are done in the sequence listed in the spec file. Use the `earlier output table` key in an `impute` group to specify a table of fill-ins to be checked out before embarking on the imputation described by this group.

If you want to not bother with a fill-in table and write imputations directly to the input table, add the `autofill` key to an imputation.

**R code for imputing income** Here is some R code to

- Blank out some incomes, adjusting and saving the data before doing so.
- Do the several imputations as per the spec file above, via `doMImpute`.
- Check out the imputations in sequence, bind them together, and plot the several log income distributions on one plot.

The sample data used in this tutorial is already-cleaned public use data, so the only missing incomes are for people under 15 (who all have missing income, which tells us that the ACS probably has an edit akin to `age<=15 => PINCP=NULL`). The `pokeHoles` function first records the original distribution as reported by the Census (in the `precut` table), then blanks out the incomes of about 30
of already-blank under-15s). Because `pokeHoles.R` is not really about Tea, we do not print it here, but you can inspect it on the pokeHoles.R page[11] of the repository.

This tutorial will not go into detail regarding the especially R-heavy portions of the code, such as how the `ggplot2` package was used to generate the final plot, though the comments give a sketch of what the code is doing.

---

```
library(tea)
readSpec("incomes.spec")
doInput()


#To prevent a million log(0) errors, give everybody $10
dbGetQuery(teaenv$con, "update viewdc set pincp=pincp+10")

# Dirty up the data by blanking out 30% of incomes
```

[11] `https://github.com/b-k/tea-tutorial/blob/master/pokeHoles.R`

```
source("pokeHoles.R")
pokeHoles("viewdc", "pincp", .3)

doMImpute()

# A function to use checkOutImpute to get imputed PINCPs and save them
# in a format amenable to the plotting routine
getOne <−function(method, fillins){
    filledtab <− paste(fillins, "fill", sep="")
    checkOutImpute(origin="viewdc", dest=filledtab, filltab=fillins,
                        subset="agep+0>15")
    outframe <−teaTable(filledtab, cols="PINCP")
    outframe$PINCP <− log10(outframe$PINCP+10)
    outframe$imp <− paste("via", method)
    return(outframe)
}

# The data sets in this function are two−column: the first
# is the observed value, and the second is the name of the method.
# Put the two imputations and the original data in that format,
# then send to ggplot.
plotWage <− function(outname){
    dfhd <− getOne("Hot Deck", "hd")
    dfnorm <− getOne("Lognormal", "norm")

    dfdc <−teaTable("precut", cols="PINCP", where="PINCP is not null")
    dfdc$PINCP <− log10(dfdc$PINCP+10) #+ runif(length(dfdc$PINCP))
    dfdc$imp <− "original"

    DFall <−rbind(dfdc, dfhd, dfnorm)

    #plot DFall, using imp# as color
    library(ggplot2)
    p <− ggplot(DFall,aes(x=PINCP,color=as.factor(imp)))
    p <− p + geom_density()
    p <− p + theme(axis.text.x=element_text(angle=45,hjust=1,vjust=1))
    bitmap(file=paste(outname,".png", sep=""),width=11∗(10/11),height=8.5∗(10/11),units
        ="in",res=150)
    print(p)
    dev.off()
}

plotWage("log_wageplots")
```

Figure 1 presents the final graphic. The data used for imputation is not pictured, but hot deck makes draws from that distribution, so the hot deck curve could be read as the post-cut data set. The imputation using a lognormal distribution produces a distribution that is lower at the peak and smooths out some of the inflection points in the data.

If we believe that people at extremes of the income spectrum are underrepresented in the complete data, then the fact that Hot Deck simply replicates the nonmissing data may not be desirable.

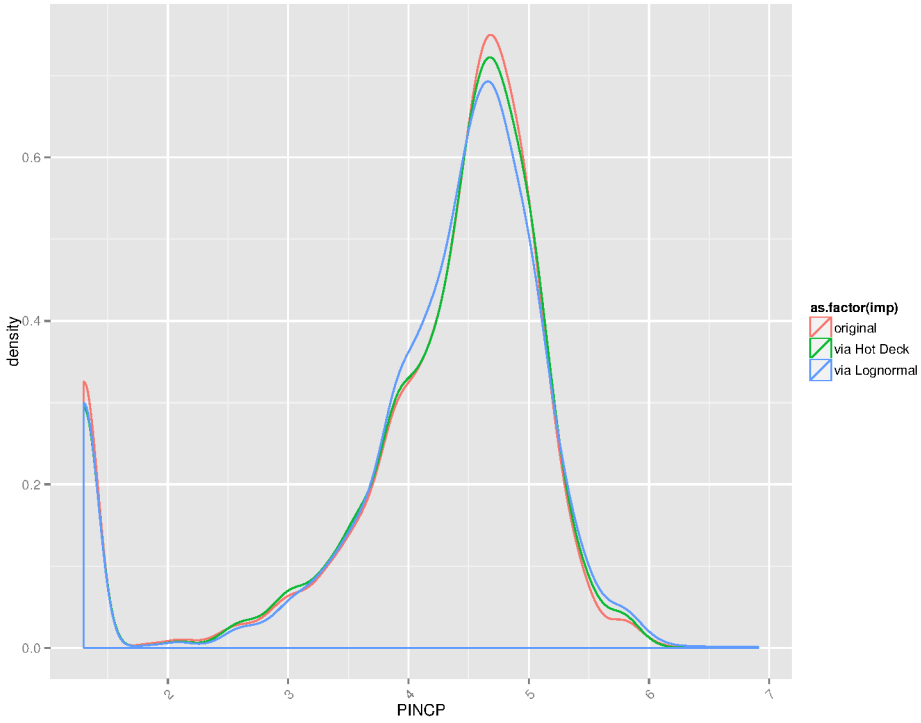


Figure 1: Log wages from the data and imputed using Hot Deck and a Normal distribution.

# 4 Imputation with auxiliary data

Consider two data sets that ostensibly measure the same thing, such as American Community Survey income questions and income questions on tax returns. The biases and assumptions underlying both questions are slightly different. People have a strong incentive to understate taxable income, or might treat exceptional circumstances differently between the two. But we have a strong expectation that the two are closely correlated.

We can expect that for some respondents, we have both observations, for some only one or the other, and for some we have neither.

Tea has an implementation of the expectation-maximization algorithm by Hartley [1958], popularized in **?**. The end result in our two-income example is a table with

ACS incomes on the rows and IRS incomes on the columns. We expect that most observations will be along the main diagonal where ACS income equals IRS income.

Given a person with a known IRS income, we can impute their ACS income by taking the appropriate column and making a random draw from the density in that column. For somebody with no income data, we can make a draw from the entire table.

This method has several advantages over substituting in IRS observations. It accommodates the uncertainty of using an auxiliary data source, and several draws from the IRS data would produce a spread of income values (see the discussion of multiple imputation below). This algorithm becomes especially useful when we go past two dimensions into six or seven, each of which is partially specified.

The primary benefit of Hartley's EM algorithm is that it uses all available information, in a Bayesianly-correct manner, to generate the final table. If a respondent has only ACS or only IRS data, their response is still used to generate the densities in the final table. Tea's implementation offers a few minor benefits: it accommodates near-misses (and so is usable for near-continuous variables like incomes) and is implemented efficiently enough to work for up to about seven-dimensional tables (though it starts to freeze up past that point, especially when many-valued variables like age or income are involved).

**An example** In the code repository, there is a second data set, named `fake_in.csv`. This is indeed fake data, generated via `fake_in.c` (also in the repository), which reads the first 2,000 income observations from the DC PUMS, and multiplies each by a draw from a truncated Normal(0.9, 0.2) distribution. Thus, the fake income data systematically under-reports the actual data.

Our first task in processing is to merge the data sets into one table. Joining data sets is one of the great strengths of SQL databases, and we only have to specify how it is done using a `join` segment in the spec file.

---

```
database: test.db
id: id

input {
    input file: dc_pums_08.csv
    output table: dc
}

fields {
    agep: real
    PINCP: real
    income: real
    log_in: real
    schl: int 0−24
    has_income: int 0, 1
    id: real
    age_cat: cat 0−3, X
```

```
}

checks {
    pincp < 0
}

recodes {
    id: serialno*100 + sporder
    log_in: log10(PINCP+10)
}

recodes {
    has_income {
        0 | PINCP==0
        1 | PINCP>0
    }

    age_cat {
        0 | AGEP <= 15
        1 | AGEP between 16 and 29
        2 | AGEP between 30 and 64
        3 | AGEP>= 65
        X |
    }
}

checks {
    has_income=0 and PINCP is null => PINCP=0

    PINCP +0.0 < 0
}

input {
    input file: fake_in.csv
    output table: fake_incomes
}

join {
    host: viewdc
    add: fake_incomes
    output table: dc_united
}

impute {
    input table: dc_united
    method: em
    vars: income, PINCP, age_cat
    output table: via_em
    subset: agep>15
    near misses: ok
```

```
}

impute {
    categories {
        age_cat
    }
    input table: dc_united
    method: hot deck
    vars: PINCP
    subset: agep>15
    output table: via_hot_deck
}
```

Here is the R code. The `library` and `readSpec` are by now familiar to you, but the script then introduces a new command, `doTable`. Because tea derived the pipeline of input/output tables from your spec file, it knows what steps to take to generate any given table. Inspecting the spec ourselves, we see that `dc_united` depends on `viewdc`, which is the product of recodes on `dc`, which is a read-in from the `dc_pums_08.csv` file, and on `fake_incomes`, which is a read-in from `fake_in.csv`. The `doTable("dc_united")` function will therefore implement all of those steps.

As above, the public-use microdata is complete, so we need to use the `pokeHoles` function to produce some missing data. The script then calls `doMImpute`.

```
library(tea)
readSpec("joined.spec")
doTable("dc_united")

source("pokeHoles.R")
pokeHoles("dc_united", "pincp", .3)

doMImpute()

# Check out an imputation using the chosen method
# Query the absolute difference between the correct PINCP and the imputed
# Get log(diff+10), where the +10 deals with zeros
# Print a summary of the log differences.
getDiffs <- function(method, name){
    filltab <- paste("via_", method, sep="")
    outtab <- paste("complete_", filltab, sep="")
    checkOutImpute(origin="dc_united", dest=outtab, filltab=filltab)
    diffs <- dbGetQuery(teaenv$con, paste("select abs(dcu.pincp - imp.pincp) \
                        from precut dcu, ", outtab, "imp\
                        where dcu.id+0.0 = imp.id \
                        and dcu.agep>15 \
                        and dcu.id in (select id from dc_united where pincp is null)"))
    diffs <- log(diffs+10)
    print(paste("MSEs for income imputed via", name))
```

```
    print(summary(diffs))
}

getDiffs("em", "EM algorithm (w/aux data)")
getDiffs("hot_deck", "Hot deck")
```

Here is the output. Note that, even though the fake data is biased, generated via a process that understates income by 90% of the true values, using it as a covariate gave us better results than not using it.

```
[1] "MSEs for income imputed via EM algorithm (w/aux data)"
abs(dcu.pincp − imp.pincp)
Min. : 2.303
1st Qu.: 7.251
Median : 8.701
Mean : 8.197
3rd Qu.: 9.799
Max. :13.715

[1] "MSEs for income imputed via Hot deck"
abs(dcu.pincp − imp.pincp)
Min. : 2.303
1st Qu.: 9.623
Median :10.483
Mean :10.166
3rd Qu.:11.296
Max. :13.635
```

# 5 Editing

To this point, we have covered imputation of missing values, where those values and edit-associated values get checked. This segment will cover editing the data that is already part of the data set. There are three things that can be done when an edit is hit:

- Keep a tally of which fields in which records failed how many edits. If you only want to get a count of how many fields fail in how many records, try the `CheckDF` function.
- Mechanical edit: make a deterministic change you specify given that the edit was triggered.
- Blank it: use a heuristic specified below to blank out a field, so you can impute it in a subsequent call to `doMImpute()`.

The sample spec for this part of the tutorial will focus on age, income, and schooling, and the presumption that children do not have advanced degrees or significant incomes.

```
database: test.db
id: id

input {
    input file: dc_pums_08.csv
    output table: dc
}

fields {
    agep: real
    PINCP: real
    schl: int 0-24
}

recodes {
    id: serialno*100 + sporder
}

checks {
    agep < 13 and pincp > 0
    schl>12 and agep <=14 and agep>=5 => schl = agep-5

    #13 is unlucky.
    schl=13 => schl=NULL
}

edit {
    input table: viewdc
    output table: ed_imp
}

impute {
    input table: viewdc
    method: normal
    categories {
        agep
    }
    subset: agep>=5
    vars: sex
    previous output table: ed_imp
    output table: imp
}
```

The `checks` section looks much like the sections above. One edit, that if age is less than twelve and income is positive, has no associated mechanical edit, because we do not know whether a fault is in the age or the income field. The other edit has an associated mechanical edit, that if years of schooling look off, we should assign the

U.S.-typical age minus five to schooling. As with most deterministic edits, this comes off as a somewhat strong assumption.

Any time a record needs to be verified, including during edits, these checks will run.

This spec file also has an `edit` segment, which allows you to describe similar logistics to the `impute` segment, like the input table (which is assumed to be the output from the last segment of the spec), the output table, and whether to overwrite.

That segment gets used when you call `doEdit()` from R. Here is a sample program.

```
library(tea)
readSpec("edit.spec")

# Because the data are already clean, let's insert some errors to edit.
dbGetQuery(teaenv$con, "update viewdc \
                        set pincp=1000 where agep = 12")
dbGetQuery(teaenv$con, "update viewdc \
                        set schl=22 where agep between 10 and 14 and id%2 == 0")

doEdit()

checkOutImpute("viewdc", "vv", filltab="ed_imp")
teaTable("vv", cols=c("agep", "schl"), where="agep<15 and schl>19")

doMImpute()
```

The ACS PUMS is already edited, so the first order of business is to insert errors, which is what the two `update` queries do toward the head of the file.

The `doEdit` call will fist handle the mechanical edits. When a record hits a failure with a mechanical edit:

- The change made by the edit is recorded in the output table.
- For the purposes of this record, this modification is exhausted. The edit (here, that `schl>12 and agep<14` is an error) wil still be checked.
- The whole check starts over, with the newly modified record.

The record may encounter additional mechanical edits, or it may get to the end of the edit list with no further changes. An exhausted mechanical edit will not be reapplied to a record, so we are guaranteed forward progress.

But there may still be edit failures. In this case, the next step that `doEdit()` will take is to blank out fields, with the intent of allowing you to impute them later.

**The blanking procedure** At this point in the process, we have checked every edit, and have a tally of every edit that failed for a given record, and know which fields are associated with the edit.

We would like to blank out the value that is implicated in the most edits. Say that an edit regarding age and housing tenure failed. If age appears in a hundred edits and

fails in six, while tenure appears in three edits but fails all three, we are inclined to believe that it is tenure (and perhaps some other variable in other age-related edits) that needs fixing. So we use a modified percentage of edits failed to determine which field to blank.

The modifier is a user-specified salience, which you may set in the `fields` segment:

```
fields {
    age: [1.5] real
    status: [2] cat married, single, divorced
    tenure: cat own, rent
}
```

Here, tenure has the default salience of one, and age has a salience of 1.5. In the example above, the score for age is now 1.5 * 6/100 = .09. The corresponding score for tenure is thus 1*3/3 = 1, so tenure is still going to be blanked if there is a failing edit regarding tenure and age.

After tenure is blanked, all later edits regarding tenure are considered resolved for this record.

**The follow-up imputation and the audit trail** At the end of the `doEdit()` step, we have a fill-in table named `edits`, which is identical to the output from the imputation steps above, except that it may include some `NULL`s, indicating that a certain field in a certain record is to be blanked out.

We could use `checkOutImpute` to fold the list of changes into a new table and check that the edits have been made.

Or, we can use the `previous output table` key in the imputation to have the imputation automatically pull the edits for us.

In this setup, the edits and imputations have all been written to the same table, thus producing a complete audit trail. We can query for every entry with a given record ID to see what happened to the record over the course of things:

```
teaTable("ed_imp", where= "id=72023405")
```

# 6 Disclosure avoidance

# 7 Synthetic data

*Raking* is a method of producing a table of cells consistent with a given list of margins. For example, we may require that any data set we produce have an age $\times$ race $\times$ ethnicity distribution and a sex $\times$ age distribution that exactly matches the complete data in a census.

Raking is a simple method by which the values in the survey table can be incrementally reweighted to match the values in the census. Let *close* indicate Kullback-Leibler

divergence; then the raked table is the closest table to the original data subject to the constraints of the row and column totals specified by the census.

Given more dimensions, we may want certain sets of dimensions to match a reference, such as requiring all block $\times$ sex categories to fit to the reference values.

But consider the case where there was no survey; we can begin with a 'noninformative' table in which all cells are filled with ones, and rake that to the closest table that matches the specified margins.

Raking a noninformative table as per the above definition (or many others) to margins deemed to not be a disclosure risk will produce microdata that also bears no disclosure risk, and can be presented to the public.

Tea's raking algorithm is designed to work well with sparse tables, and makes heavy use of the database for its work. If the marginal cells imply a large number of zeros (which is a near-certainty in high-dimensional tables), those cells will remain zero in the raked table.

# 8 Appendix: additional info

Here are some additional notes about using Tea and its attendant tools.

## 8.1 Ways of expressing missing data

Because Tea is about dealing with missing data, it may be worth getting to know how various systems express missingness.

- SQL has a `NULL` marker.
- The IEEE floating-point standard has a not-a-number (NaN) marker (in fact, billions of NaN markers).
- R has an NA marker distinct from its NaN marker.

These are explicit markers of missingess that can not be mistaken for erroneous values. It is easy to find systems that use ad hoc markers to represent missingness, including zero, -1, an empty string, and so on, especially in older systems that predate the IEEE standard (first published in 1985). We recommend using the `input/missing marker` key to specify this ad hoc value; Tea will then read elements intended as blanks into the database as `NULL`.

The IEEE 754 standard specifies a `NaN` marker to represent math errors such as `0./0.`, and this is often used to represent missing data.[12]

Comparison to `NaN` *always* fails—the standard specifies that even `a==a` is false if `a` is `NaN`. Therefore, there are functions to use to check for missingness. in SQL,

```
a is null
or
a is not null
```

[12]The standard also states that `1./0.` produces `INFINITY` and `-1./0.` produces `-INFINITY`.

are correct ways to check for missings or nonmissings. Note that SQLite allows either `NULL` or `null`. In R

```
is.na(a)
```

will check whether `a` is `NaN`.

Especially careful readers will note that using `NaN` as a missingness marker is potentially ambiguous: a `NaN` could represent truly missing data or it could represent the result of dividing a nonmissing zero by another nonmissing zero. Thus, R has an `NA` marker, which is distinct from its `NaN` marker. The authors of this document have surveyed R users, including the authors of some popular packages, and have not found a single user who takes care to preserve the distinction between `NaN` and `NA` in practice. Note that `NaN`s are a subset of `NA`s, not the other way around:

```
> a<-NaN
> is.na(a)
[1] TRUE


> b<-NA
> is.nan(b)
[1] FALSE
```

Therefore, we recommend using the shorter `is.na`, which catches both `NA`s and `NaN`s.

See also this blog post[13] for bit-level discussion of the types of `NaN`.

## 8.2 Using m4 to reduce repetition

## 8.3 Citing Tea

To date, the only formal publication describing Tea is a UN proceedings paper [Klemens, 2012], so until we get out more publications it is may be the best option when citing Tea in an academic paper:

```
@inproceedings{klemens:tea,
    author={Ben Klemens},
    title= {Tea for Survey Processing},
    booktitle={Proceedings of the Conference of European Statisticians},
    chapter={Working paper 29},
    crossref={UNECE12}
}

@proceedings{UNECE12,
    title={Proceedings of the Conference of European Statisticians},
    series={Work Session on Statistical Data Editing},
```

[13] `http://modelingwithdata.org/arch/00000132.htm`

```
    organization={United Nations Economic Commission for Europe},
    year=2012,
    month=Sep
}
```

# References


M Bankier, P Mason, and P Poirier. Imputation of demographic variables in the 2001 Canadian Census of Population. In *American Statistical Association, Proceedings of the Section on Survey Research Methods*, 2002.

Bor-Chung Chen, Yves Thibaudeau, and William E Winkler. A comparison study of ACS if-then-else, NIM, and DISCRETE edit and imputation systems. In *ASA Joint Statistical Meetings: Section on Survey Research Methods*, 2002.

Arthur R Cresce, Jr, Sally M Obenski, and James Farber. Improving imputation: The plan to examine count, status, vacancy, and item imputation in the Decennial Census. In *Proceedings of the Conference of European Statisticians*, Work Session on Statistical Data Editing. United Nations Economic Commission for Europe, May 2005.

IP Fellegi and D Holt. A systematic approach to automatic edit and imputation. *Journal of the American Statistical Association*, 71:17–35, 1976.

María Garcia. Implied edit generation and error localization for ratio and balancing edits. In *Proceedings of the Conference of European Statisticians*, Work Session on Statistical Data Editing. United Nations Economic Commission for Europe, October 2003.

H O Hartley. Maximum likelihood estimation from incomplete data. *Biometrics*, 14 (2):174–194, June 1958.

Ben Klemens. Tea for survey processing. In *Proceedings of the Conference of European Statisticians*, Work Session on Statistical Data Editing. United Nations Economic Commission for Europe, September 2012.

Mark Pierzchala. A review of the state of the art in automated data editing and imputation. *Journal of Official Statistics*, 6(4):355–377, 1990.

J L Schafer. *Analysis of Incomplete Multivariate Data*. Number 72 in Monographs on Statistics and Applied Probability. Chapman & Hall/CRC, 1997.

William E Winkler. Editing discrete data. In *American Statistical Association, Proceedings of the Section on Survey Research Methods*, pages 108–113, 1995.