An R Function for Tabulating Census Data

Griffith Feeney

19 January 2014

1. Introduction

The advantage of tabulating census data in **R** is a seamless transition to analysis using R, a powerful environment that has become a *lingua franca* of statistics throughout the world.

The **R** function `tablew()` produces tables in the form of n-dimensional arrays. Once the environment has been set up, the command

```
tablew(age, sex)
```

will instantiate an appropriately labeled 2-way table (matrix) of population by age and sex. Listing n variables will produce an n-dimensional array.

It is not desirable to load entire census datasets into **R**. For all but the smallest dataset it is impossible. "Vectorizing" datasets minimizes resource demands and maximizes speed by making it possible to bring only needed resources into **R** as they are needed.

Vectorization requires a single preliminary operation easily implemented by a shell script. The process may take many hours for a large datasets, but the script may be run unattended, over night if necessary.

Vectorization has the advantage of making it possible to analyze a small number of variables from a large dataset without producing full file format and codebook information.

2. Requirements

The program (function) `tablew()` meets the following requirements.

1. The output is in tabular form, fully labelled with variable and variable value names.
2. There is no restriction on the number of variables of tabulation. n variables of tabulation result in an n-dimensional array.
3. The output includes cells for all possible combinations of values of the variables of tabulation, including combinations of values that do not occur in the data.
4. The data to be tabulated may be weighted.

But for the last of these requirements, tt table() would provide everything necessary, though certain of aspects of the environment and setup steps below would facilitate the work.

Because a census is by definition a complete enumeration of a specified target population, census data do not usually involve weights. It is necessary to be able to work with samples of census data, and these may involve weights.

3. Required inputs

To meet these requirements, the following information is needed.

1. The weight to be used for each person in the dataset
2. For each variable of tabulation, the codes for this variable for each person in the dataset.
3. For each variable of tabulation, a list of all valid codes and values these codes represent

The following section describes an environment to provide this information.

4. Environment

Variables are represented in **R** by character vectors with one component for each person in the dataset, in the order in which persons appear in the dataset.

Character vectors make it easy to deal gracefully with non-numeric characters. This is useful in general and essential for working with unedited data. The ability to work easily with unedited data is essential for work in national statistical offices.

Weights are represented in **R** by a numeric vector. If the dataset field containing the weights does not include a decimal point, the multiple of ten by which the values in the field must be divided must be given exogenously.

Codebook information is represented in **R** in a character matrix with one row for each valid code, a column for the codes, a column for the values represented by the codes, and an optional third column for abbreviated values. There is a codebook matrix for each variable used.

```
  code short
1 "1"  "Male"
2 "2"  "Female"
3 "9"  "MV"
```

Weights, vectorized variables and variable codebooks are contained in files in the current working directory. Weights and vectorized variables are stored as compressed text files, which R can decompress on the fly when reading.

Variable codebooks are stored in text files that look like this,

```
1;Male
2;Female
9;MV
```

with an optional third semicolon-delimited column for long value names, which allows the second column to be used for abbreviated names.

`tablew()` transparently brings weights, vectorized variables and codebooks into the workspace as they are needed.

The program is described in two stages, minimal code required to accomplished the desired result, and additional code to improve usability and funcationality. The next section describes the "core" code. The following section describes additional code.

5. Procedure and minimal code

Step 1 Preliminaries

The tabulation procedure requires three R objects, a list `codes` giving possible codes for each variable of tabulation, a list `values` of the values represented by these codes,, and a vector `dims` giving the number of values for each variable of tabulation.

The following code produces these objects from the information in the variable codebooks, assumed present in the workspace.

```
codes   <- list()
values  <- list()
dims    <- numeric(0)
for (i in 1:length(vnames)) {
  codebook    <- get(paste(vnames[i],".cb", sep=""))
  codes[[i]]  <- codebook[, 1]
  values[[i]] <- codebook[, 2]
  dims        <- c(dims, length(codes[[i]]))
}
names(codes)  <- vnames
names(values) <- vnames
```

Step 2 Construct dummy table-as-vector

The following code produces a vector `tav` with one component for every cell in the desired output table with components named by the combination of codes represented by the cell.

```
groups <- codes[[1]]
if (length(codes) >= 2) {
  for (i in 2:length(codes)) {
    groups <- as.vector(outer(groups, codes[[i]], FUN="paste", sep=""))
  }
}
tav <- rep(0, times=length(groups))
names(tav) <- groups
```

`codes` is a list whose components are vectors giving codes for the values of the variables of tabulation. `codes[[1]]` is thus a vector giving codes for the first variable of tabulation, and so on.

The first line initializes the `groups` vector to be constructed. The construction is completed by the following loop, cycles through variables of tabulation.

The last two lines create a vector of zeros whose length equals the number of cells in the final n-way table and names the components of this vector by the components of the `groups` vector.

Step 3 Produce a compound-variable-for-tabulation

The following code produces a vector whose components are the concatenation of the codes of the variables of tabulation.

```
cvft <- get(vnames[1])
if (length(vnames) >= 2) {
  for (i in 2:length(vnames)) {
    cvft <- paste(cvft, get(vnames[i]), sep="")
  }
}
```

Step 4 Produce desired n-way table

Only three lines of code are needed. Tabulation proper is accomplished by `rowsum()`, suggested of Thomas Lumley.

```
tav.nz <- rowsum(weights, cvft)[, 1]
```

This produces a vector whose components are the numbers of persons with any combination of codes, valid or otherwise, of variables of tabulation that occur in the dataset, the combinations being names of the components. `tav.nz` does not include combinations that do not occur in the dataset, e.g., numbers of persons at very old ages, whence the `.nz`.

The next line pulls the non-zero values of the final table into the the dummy vector `tav` constructed in Step 1, leaving the initialized zero cells unchanged. It exploits the possibility of indexing vectors by the names, rather than the numbers, of their components.

```
tav[names(tav.nz)] <- tav.nz
```

The third and last line of code formats the table as a multidimensional array.

```
array(tav, dim=dims, dimnames=values)
```

`dims` is a vector, constructed in Step 1, giving the number of values for each variable of tabulation, in the order in which the values are listed in the call to `tablew()`.

`values` is a list, also constructed in Step 1, whose components are vectors for the variables of tabulation, in the same order, giving the values of the variable in the order listed in the codebook.

Table 1 shows code for a minimal `tablew()` function.

6. Supplementary code

Addition 1 `getweights()` and `getvariable()`

Weights and variables can be pulled into the **R** workspace manually at the command line, but it is worth writing a function to do this, particularly for pulling in codebook information.

Table 2 shows a function to pull in weights. Table 3 shows a function to pull in variables. The distinction is worth making because weights do not require codebook information, but do require coercion to numeric format.

`get()` is used when the vector is already present in the workspace because there is no way (that I can find, at least) to exit an R function gracefully before executing all the code.

The message is printed to warn the user that the operation may take more than a few seconds. The division by 100 is specific to the IPUMS International data I am currently working with.

The second section of code is required to put the codebook information in the needed format and to allow for the optional third column. The codebook file is retrieved with `readLines()` because we need leading blanks this is the only thing I could find that works.

Both functions to the global environment, which I consider appropriate here, though it is discouraged in functional programming.

With these convenience functions in hand, we add the following code to the beginning of `tablew()`.

```
getweights("weight")
for (i in 1:length(vnames)) {
  getvariable(vnames[i])
}
```

Addition 2 Automatic assignment of table names

[describe rationale later, for now, just the code]

```
table.name <- paste(vnames, collapse=".")
if (length(vnames) == 1) {
  table.name <- paste(table.name, ".frq", sep="")
}
assign(table.name, x, envir=globalenv())
```

The conditional statement is required to keep from clobbering the vectorized variable tabulated when making 1-way tabulations (frequencies).

Addition 3 Enable restriction to universe

Some tables should be made only for subsets of the full dataset, e.g., tables of women by age and number of children ever born. We might use use `tablew()` to tabulate by age, children ever born and sex and hten index out males, but this is clumsy, inefficient, and easy to avoid.

The following code restricts tabulation to records specified by a logical vector supplied as an additional argument to the function.

```
if (!all(universe)) {
  cat("tablew: Restricting variables to universe ...\n")
  for (i in 1:length(vnames)) {
    x <- get(vnames[i])[universe]
    assign(vnames[i], x)  # NO global assignment here!
  }
  weights <- weight[universe]  # Correct, but not immediately obvious
}
```

The reassignment of vectorized variables is local to the function.

Addition 4 Improved command line interface

The following code makes it possible to type `tablew(age, sex)` in place of `tablew(c("age", "sex"))` at the command line. In practice, this is a very substantial convenience.

```
variable.list <- as.list(substitute(list(...)))[-1L]
vnames  <- character(0)
for (i in 1:length(variable.list)) {
  vnames <- c(vnames, as.character(variable.list[[i]]))
}
```

Table 4 Final `tablew()` code (continued)

```
  # Create compound variable for tabulation [CORE]
  cat("tablew: Creating compound variable for tabulation ...\n")
  cvft <- get(vnames[1])
  if (length(vnames) >= 2) {
    for (i in 2:length(vnames)) {
      cvft <- paste(cvft, get(vnames[i]), sep="")
    }
  }

  # Produce and return cross-tabulation [CORE]
   cat("tablew: Tabulating ...\n")
   tav.nz <- rowsum(weights, cvft)[, 1]
   tav[names(tav.nz)] <- tav.nz
   x <- array(tav, dim=dims, dimnames=values)
```

Table 1 Minimal code for `tablew()`

```
tablew <- function(vnames, weights=weight) {
  # Step 1: Preliminaries
  codes   <- list()
  values  <- list()
  dims    <- numeric(0)
  for (i in 1:length(vnames)) {
    codebook    <- get(paste(vnames[i],".cb", sep=""))
    codes[[i]]  <- codebook[, 1]
    values[[i]] <- codebook[, 2]
    dims        <- c(dims, length(codes[[i]]))
  }
  names(codes)  <- vnames
  names(values) <- vnames

  # Step 2: Construct dummy table-as-vector
  groups <- codes[[1]]
  if (length(codes) >= 2) {
    for (i in 2:length(codes)) {
      groups <- as.vector(outer(groups, codes[[i]], FUN="paste", sep=""))
    }
  }
  tav <- rep(0, times=length(groups))
  names(tav) <- groups

  # Step 3: Construct compound variable for tabulation
  cat("tablew: Creating compound variable for tabulation ...\n")
  cvft <- get(vnames[1])
  if (length(vnames) >= 2) {
    for (i in 2:length(vnames)) {
      cvft <- paste(cvft, get(vnames[i]), sep="")
    }
  }

  # Step 4: Construct and return cross-tabulation
  cat("tablew: Tabulating ...\n")
  tav.nz <- rowsum(weights, cvft)[, 1]
  tav[names(tav.nz)] <- tav.nz
  array(tav, dim=dims, dimnames=values)
}



  # Create table name, assign result to name in global environment [ADD 2]
  table.name <- paste(vnames, collapse=".")
  if (length(vnames) == 1) {
    table.name <- paste(table.name, ".frq", sep="")
```

Table 2 Code for `getweights()`

```
getweights <- function(wname) {
  if (exists(wname)) {
    get(wname)
  } else {
    cat("Getting \"", wname, "\" ...\n", sep="")
    x <- readLines(paste(wname, ".gz", sep=""))  # Compressed data
    x <- as.numeric(x)/100
    assign(wname, x, envir=globalenv())
  }
}
```

Table 3 Code for `getvariable()`

```
getvariable <- function(vname) {
  #   Args: Name of variable, quoted (full path)
  # Value:  Character vector of variable values
  # Get variable (if not already in workspace)
  if (exists(vname)) {
    get(vname)
  } else {
    cat("Getting \"", vname, "\" ...\n", sep="")
    x <- readLines(paste(vname, ".gz", sep=""))  # Compressed data
    assign(vname, x, envir=globalenv())
  }
  # Get codebook for variable (if not weight)
    cbname <- paste(vname,".cb", sep="")
    lines <- readLines(cbname)
    x <- NULL
    for (i in 1:length(lines)) {
      x <- rbind(x, strsplit(lines[i], split=";")[[1]])
    }
    rownames(x) <- 1:dim(x)[1]
    if (dim(x)[2] == 3) {
      colnames(x) <- c("code", "short", "long")
    }
    if (dim(x)[2] == 2) {
      colnames(x) <- c("code", "short")
    }
    assign(cbname, x, envir=globalenv())
}
```

Table 4 Final `tablew()` code

```
tablew <- function(..., weights=weight, \
                   universe=rep(TRUE,times=length(weights))) {
  # Simplify command line [ADD 4]
  variable.list <- as.list(substitute(list(...)))[-1L]
  vnames  <- character(0)
  for (i in 1:length(variable.list)) {
    vnames <- c(vnames, as.character(variable.list[[i]]))
  }

  # Get weights, variables, and codebooks automatically [ADD 1]
  getweights("weight")
  for (i in 1:length(vnames)) {
    getvariable(vnames[i])
  }

  # Get variable codes, short values and dims from codebook [CORE]
  codes   <- list()
  values  <- list()
  dims    <- numeric(0)
  for (i in 1:length(vnames)) {
    codebook    <- get(paste(vnames[i],".cb", sep=""))
    codes[[i]]  <- codebook[, 1]
    values[[i]] <- codebook[, 2]
    dims        <- c(dims, length(codes[[i]]))
  }
  names(codes)  <- vnames
  names(values) <- vnames

  # Construct zero-initialized tabulation-as-vector [CORE]
  groups <- codes[[1]]
  if (length(codes) >= 2) {
    for (i in 2:length(codes)) {
      groups <- as.vector(outer(groups, codes[[i]], FUN="paste", sep=""))
    }
  }
  tav <- rep(0, times=length(groups))
  names(tav) <- groups

  # Restrict variables of tabulation and weights to universe [ADD 3]
  if (!all(universe)) {
    cat("tablew: Restricting variables to universe ...\n")
    for (i in 1:length(vnames)) {
      x <- get(vnames[i])[universe]
      assign(vnames[i], x)  # NO global enviroment here!
    }
    weights <- weight[universe]  # tricky!
  }
```

```
  }
  if (!all(universe)) {
    table.name <- paste(table.name, ".unv.", deparse(substitute(universe)), sep="")
  }
  assign(table.name, x, envir=globalenv())
}
```