Working with Unknown Values

The gdata package

by Gregor Gorjanc

Introduction

Unknown or missing values can be represented in various ways. For example SAS uses . (dot), while R uses `NA`, which we can read as Not Available. When we import data into R, say via `read.table` or its derivatives, conversion of blank fields to `NA` (according to `read.table` help) is done for `logical`, `integer`, `numeric` and `complex` classes. Additionally, the `na.strings` argument can be used to specify values that should also be converted to `NA`. Inversely, there is an argument `na` in `write.table` and its derivatives to define value that will replace `NA` in exported data. There are also other ways to import/export data into R as described in the *R* Data Import/Export manual (R Development Core Team, 2006). However, all approaches lack the possibility to define unknown value(s) for some particular column. It is possible that an unknown value in one column is a valid value in another column. For example, I have seen many datasets where values such as 0, -9, 999 and specific dates are used as column specific unknown values.

This note describes a set of functions in package **gdata**[1] (Warnes , 2006): `isUnknown`, `unknownToNA` and `NAToUnknown`, which can help with testing for unknown values and conversions between unknown values and `NA`. All three functions are generic (S3) and were tested (at the time of writing) to work with: `integer`, `numeric`, `character`, `factor`, `Date`, `POSIXct`, `POSIXlt`, `list`, `data.frame` and `matrix` classes.

Description with examples

The following examples show simple usage of these functions on `numeric` and `factor` classes, where value `0` (beside `NA`) should be treated as an unknown value:

```
> library("gdata")
> xNum <- c(0, 6, 0, 7, 8, 9, NA)
> isUnknown(x=xNum)
[1] FALSE FALSE FALSE FALSE FALSE FALSE
    TRUE
```

The default unknown value in `isUnknown` is `NA`, which means that output is the same as `is.na` — at least for atomic classes. However, we can pass the argument `unknown` to define which values should be treated as unknown:

```
> isUnknown(x=xNum, unknown=0)
[1] TRUE FALSE  TRUE FALSE FALSE FALSE
    FALSE
```

This skipped `NA`, but we can get the expected answer after appropriately adding `NA` into the argument `unknown`:

```
> isUnknown(x=xNum, unknown=c(0, NA))
[1] TRUE FALSE  TRUE FALSE FALSE FALSE
    TRUE
```

Now, we can change all unknown values to `NA` with `unknownToNA`. There is clearly no need to add `NA` here. This step is very handy after importing data from an external source, where many different unknown values might be used. Argument `warning=TRUE` can be used, if there is a need to be warned about "original" `NA`s:

```
> xNum2 <- unknownToNA(x=xNum, unknown=0)
[1] NA  6 NA  7  8  9 NA
```

Prior to export from R, we might want to change unknown values (`NA` in R) to some other value. Function `NAToUnknown` can be used for this:

```
> NAToUnknown(x=xNum2, unknown=999)
[1] 999   6 999   7   8   9 999
```

Converting `NA` to a value that already exists in `x` issues an error, but `force=TRUE` can be used to overcome this if needed. But be warned that there is no way back from this step:

```
> NAToUnknown(x=xNum2, unknown=7,
             force=TRUE)
[1] 7 6 7 7 8 9 7
```

Examples below show all peculiarities with class `factor`. `unknownToNA` removes `unknown` value from levels and inversely `NAToUnknown` adds it with a warning. Additionally, `"NA"` is properly distinguished from `NA`. It can also be seen that the argument `unknown` in functions `isUnknown` and `unknownToNA` need not match the class of `x` (otherwise factor should be used) as the test is internally done with `%in%`, which nicely resolves coercing issues.

```
> xFac <- factor(c(0, "BA", "RA", "BA",
                   NA, "NA"))
[1] 0    BA   RA   BA   <NA> NA
Levels: 0 BA NA RA
> isUnknown(x=xFac)
[1] FALSE FALSE FALSE FALSE  TRUE FALSE
> isUnknown(x=xFac, unknown=0)
[1]  TRUE FALSE FALSE FALSE FALSE FALSE
> isUnknown(x=xFac, unknown=c(0, NA))
[1]  TRUE FALSE FALSE FALSE  TRUE FALSE
```

[1] package version 2.3.1

```
> isUnknown(x=xFac, unknown=c(0, "NA"))
[1]  TRUE FALSE FALSE FALSE FALSE  TRUE
> isUnknown(x=xFac, unknown=c(0, "NA", NA))
[1]  TRUE FALSE FALSE FALSE  TRUE  TRUE

> xFac <- unknownToNA(x=xFac, unknown=0)
[1] <NA> BA   RA   BA   <NA> NA
Levels: BA NA RA
> xFac <- NAToUnknown(x=xFac, unknown=0)
[1] 0  BA RA BA 0  NA
Levels: 0 BA NA RA
Warning message:
new level is introduced: 0
```

These two examples with classes numeric and factor are fairly simple and we could get the same results with one or two lines of R code. The real benefit of the set of functions presented here is in list and data.frame methods, where data.frame methods are merely wrappers for list methods.

We need additional flexibility for list/data.frame methods, due to possibly having multiple unknown values that can be different among list components or data.frame columns. For these two methods, the argument unknown can be either a vector or list, both possibly named. Of course, greater flexibility (defining multiple unknown values per component/column) can be achieved with a list.

When a vector/list object passed to the argument unknown is not named, the first value/component of a vector/list matches the first component/column of a list/data.frame. This can be quite error prone, especially with vectors. Therefore, I encourage the use of a list. In case vector/list passed to argument unknown is named, names are matched to names of list or data.frame. If lengths of unknown and list or data.frame do not match, recycling occurs.

The example below illustrates the application of the described functions to a list which is composed of previously defined and modified numeric (xNum) and factor (xFac) classes. First, function isUnknown is used with 0 as an unknown value. Note that we get FALSE for NAs as has been the case in the first example.

```
> xList <- list(a=xNum, b=xFac)
$a
[1]  0  6  0  7  8  9 NA

$b
[1] 0  BA RA BA 0  NA
Levels: 0 BA NA RA
> isUnknown(x=xList, unknown=0)
$a
[1] TRUE FALSE  TRUE FALSE FALSE FALSE
    FALSE

$b
[1] TRUE FALSE FALSE FALSE  TRUE FALSE
```

We need to add NA as an unknown value. However, we do not get the expected result this way!

```
> isUnknown(x=xList, unknown=c(0, NA))
$a
[1] TRUE FALSE  TRUE FALSE FALSE FALSE
    FALSE

$b
[1] FALSE FALSE FALSE FALSE FALSE FALSE
```

This is due to matching of values in the argument unknown and components in a list; i.e., 0 is used for component a and NA for component b. Therefore, it is less error prone and more flexible to pass a list (preferably a named list) to the argument unknown, as shown below.

```
> xList1 <- unknownToNA(x=xList,
+    unknown=list(b=c(0, "NA"), a=0))
$a
[1] NA  6 NA  7  8  9 NA

$b
[1] <NA> BA   RA   BA   <NA> <NA>
Levels: BA RA
```

Changing NAs to some other value (only one per component/column) can be accomplished as follows:

```
> NAToUnknown(x=xList1,
+    unknown=list(b="no", a=0))
$a
[1] 0 6 0 7 8 9 0

$b
[1] no BA RA BA no no
Levels: BA no RA

Warning message:
new level is introduced: no
```

A named component .default of a list passed to argument unknown has a special meaning as it will match a component/column with that name and any other not defined in unknown. As such it is very useful if the number of components/columns with the same unknown value(s) is large. Consider a wide data.frame named df. Now .default can be used to define unknown value for several columns:

```
> df <- unknownToNA(x=df,
+                   unknown=(.default=0,
+                            col1=999,
+                            col2="unknown"))
```

If there is a need to work only on some components/columns you can of course "skip" columns with standard R mechanisms, i.e., by subsetting list or data.frame objects:

```
> cols <- c("col1", "col2")
> df[, cols] <- unknownToNA(x=df[, cols],
+    unknown=(col1=999,
+             col2="unknown"))
```

Summary

Functions `isUnknown`, `unknownToNA` and `NAToUnknown` provide a useful interface to work with various representations of unknown/missing values. Their use is meant primarily for shaping the data after importing to or before exporting from R. I welcome any comments or suggestions.

Bibliography

R Development Core Team. *R Data Import/Export*, 2006. URL `http://cran.r-project.org/manuals.html`. ISBN 3-900051-10-0. 24

G. R. Warnes. *gdata: Various R programming tools for data manipulation*, 2006. URL `http://cran.r-project.org/src/contrib/Descriptions/gdata.html`. R package version 2.3.1. Includes R source code and/or documentation contributed by Ben Bolker, Gregor Gorjanc and Thomas Lumley. 24

Gregor Gorjanc
University of Ljubljana, Slovenia
`gregor.gorjanc@bfro.uni-lj.si`

A New Package for Fitting Random Effect Models

The npmlreg package

by Jochen Einbeck, John Hinde, and Ross Darnell

Introduction

Random effects have become a standard concept in statistical modelling over the last decades. They enter a wide range of applications by providing a simple tool to account for such problems as model misspecification, unobserved (latent) variables, unobserved heterogeneity, and the like. One of the most important model classes for the use of random effects is the generalized linear model. Aitkin (1999) noted that “the literature on random effects in generalized linear models is now extensive,” and this is certainly even more true today.

However, most of the literature and the implemented software on generalized linear mixed models concentrates on a normal random effect distribution. An approach that avoids specifying this distribution parametrically was provided by Aitkin (1996a), using the idea of ’Nonparametric Maximum Likelihood’ (NPML) estimation (Laird, 1978). The random effect distribution can be considered as an unknown mixing distribution and the NPML estimate of this is a finite discrete distribution. This can be determined by fitting finite mixture distributions with varying numbers of support points, where each model is conveniently fitted using a straightforward EM algorithm.

This approach is implemented in GLIM4 (Aitkin and Francis, 1995). Despite being a quite powerful tool, the current GLIM-based software is computationally limited and the GLIM system is no longer widely used. Though the alternatives C.A.MAN (Böhning et al., 1992) and the Stata program `gllamm` (Skrondal and Rabe-Hesketh, 2004) cover parts of GLIMs capacities (in the latter case based on Newton-Raphson instead of EM), no R implementation of NPML estimation existed. The package **npmlreg** (Einbeck et al., 2006), which we wish to introduce to the R community in this article, is designed to fill this gap.

NPML estimation

Assume there is given a set of explanatory vectors $x_1, \ldots, x_n$ and a set of observations $y_1, \ldots, y_n$ sampled from an exponential family distribution[1] $f(y_i|\beta, \phi_i)$ with dispersion[2] parameter ϕ_i. In a generalized linear model, predictors and response are assumed to be related through a link function h,

$$\mu_i \equiv E(y_i|\beta, \phi_i) = h(\eta_i) \equiv h(x_i'\beta),$$

and the variance $Var(y_i|\beta, \phi_i) = \phi_i v(\mu_i)$ depends on a function $v(\mu_i)$ which is entirely determined by the choice of the particular exponential family. However, often the actual variance in the data is larger than the variance according to this strict mean-variance relationship. This effect is commonly called overdispersion, reasons for which might be, e.g., correlation in the data or important explanatory variables not included in the model. In order to account for additional unexplained variability of the individual observations, a random effect z_i with density $g(z)$ is in-

[1] In the present implementation, Gaussian, Poisson, Binomial, and Gamma distributed responses are supported

[2] For binomial and Poisson models, $\phi_i \equiv 1$. For Gaussian and Gamma models, the dispersion may be specified as constant, i.e., $\phi_i \equiv \phi$, or as depending on the observation i. The theory in this section is provided for the most general case, i.e., variable dispersion.