On the Usage of the gRbase Package
—
Notice: The functionality described in this note is not being maintained / developed further. Graphical modelling facilities are provided in the gRim package

Claus Dethlefsen
Aalborg Hospital, Aarhus University Hospital, Denmark

Søren Højsgaard
Aalborg University, Denmark

September 25, 2012

Contents

1 Introduction

This document is a supplement to Dethlefsen and Højsgaard (2005) (hereafter called CDSH) which is the formal reference to the `gRbase` package. In CDSH several broader perspectives are outlined and references to litterature as well as to software are given. The present document is more down to earth as it describes what is actually working.

The core of `gRbase` is the `gmData` and `gModel` classes described below.

`gmData` objects: A fundamental element of `gRbase` is a common class for representing data. No matter the actual representation of data, the important characteristics are contained in a graphical metadata (`gmData`) object. It contains the abstraction of data into a meta data object including variable names and types etc. Also, it may be possible to work without data, which may be valuable if the point of interest is in the model alone.

Separating the specification of the variables from data has the benefit, that some properties of a model can be investigated without any reference to data, for example decomposability and collapsibility. The `gmData` class is described in Section 3.

In principle this allows for working with a reference to data, such as a database. This enables modelling although data are unavailable at the time of modelling, or if the data-amount is huge or if the data changes dynamically.

`gModel` objects: A `gModel` object links a model specification to a `gmData` object. The model given by a model specification which can be quite arbitrary but which might be a formula.

When defining a `gModel` object, no fitting is done. This is an important difference between model in `gRbase` and e.g. linear models in the function `lm` in **R**. There are two reasons for this: First that some aspects of a model may be of interest without any reference to data. Secondly once a model is to be fitted to data, there may be several possible "engines" for doing so. For example, one might fit a graphical Gaussian model with maximum likelihood or by working with another type of estimating function. The `gModel` class is described in Section 4.

Some features of `gRbase` will be illustrated in the present paper on the basis of the `rats` dataset in the `gRbase` package. The `rats` dataset is from a hypothetical drug trial, where the weight losses of male and female rats under three different drug treatments have been measured after one and two weeks. The dataset is provided in the `gRbase` package, and is further described in Edwards (2000). We will also refer to the dataset `HairEyeColor` (Snee, 1974), included in **R**.

2 A small sample session

Before describing the core elements of `gRbase`, we present a sample session intended to give the reader a feel for how an end user will use `gRbase`.

Creating a gmData object first, data are created as a `gmData` object from an existing `table` object.

```
library(gRbase)
data(HairEyeColor)
gmdHec  <- as.gmData(HairEyeColor)  # define data
gmdHec
```

```
  varNames shortNames varTypes nLevels
1     Hair          H Discrete       4
2      Eye          E Discrete       4
3      Sex          S Discrete       2
To see the values of the factors use the 'valueLabels' function
To see the data use the 'observations' function
```

```
valueLabels(gmdHec)
```

```
$Hair
[1] "Black" "Brown" "Red"   "Blond"

$Eye
[1] "Brown" "Blue"  "Hazel" "Green"

$Sex
[1] "Male"   "Female"
```

Then, the model with sex independent of hair-colour and eye-colour is defined, fitted (with the `loglm`-engine) and finally the output is analysed using the `anova` procedure to test the model against the saturated model.

```
hecM1 <- hllm(~Hair*Eye+Sex,gmdHec)# indep. of Sex
hecM1 <- fit(hecM1,engine="loglm")
anova(getFit(hecM1))                   # test against saturated model
```

```
Call:
loglm(formula = loglm.formula, data = c(32, 53, 10, 3, 11, 50,
10, 30, 10, 25, 7, 5, 3, 15, 7, 8, 36, 66, 16, 4, 9, 34, 7, 64,
5, 29, 7, 5, 2, 14, 7, 8))

Statistics:
                   X^2 df P(> X^2)
Likelihood Ratio 19.86 15   0.1775
Pearson          19.57 15   0.1892
```

3 The `gmData` class

A `gmData` object contains, by default, information about variable names, variable types, their labels, their levels (for discrete variables), and whether the variables are latent or not. Unique abbreviations (short names) of the variable names are created for ease of use when specifying model formulas. Besides, a `gmData` object may contain data or a reference to data, but need not do so.

3.1 Creating a gmData object from a data frame or a table

Typically one will create a `gmData` object (with data) from a data frame as follows. Section 2 showed how to do this for a table. For a data frame the scheme is the same:

```
data(rats)
gmdRats <- as.gmData(rats)
gmdRats
```

```
  varNames shortNames   varTypes nLevels
1      Sex          S   Discrete       2
2     Drug          D   Discrete       3
3       W1          a Continuous      NA
4       W2          b Continuous      NA
To see the values of the factors use the 'valueLabels' function
To see the data use the 'observations' function
```

Observe, that when an object is printed, only the summary of the variables are printed. Data and value labels are not displayed, but may be accessed separately.

3.2 Creating a gmData object manually

A `gmData` object may be created with `newgmData`:

```
gmdRatsNodata  <-  newgmData(
                          varNames=c("Sex","Drug","W1","W2"),
                          varTypes=c("Discrete", "Discrete",
 "Continuous", "Continuous"),
                          nLevels=c(2,3,NA,NA),
                          valueLabels=list( Sex=c("M", "F"),
 Drug=c("D1", "D2", "D3")))
gmdRatsNodata
```

```
  varNames shortNames   varTypes nLevels
1      Sex          S   Discrete       2
2     Drug          D   Discrete       3
3       W1          a Continuous      NA
4       W2          b Continuous      NA
To see the values of the factors use the 'valueLabels' function
```

```
valueLabels(gmdRatsNodata)
```

```
$Sex
[1] "M" "F"

$Drug
[1] "D1" "D2" "D3"
```

Note that there is some redundancy in the specification above: The value of `nLevels` can be deduced from `valueLabels`. Therefore `nLevels` needs not to be specified. If `valueLabels` are not given, then default labels are created based on nLevels. If neither nLevels nor valueLabels are given, then all discrete variables are assumed to be binary. Following this convention we can write

```
d  <-  newgmData(varNames=c("Sex","Drug","W1","W2"),
                 varTypes=c("Discrete", "Discrete",
 "Continuous", "Continuous"))
d
```

```
  varNames shortNames   varTypes nLevels
1      Sex          S   Discrete       2
2     Drug          D   Discrete       2
3       W1          a Continuous      NA
4       W2          b Continuous      NA
To see the values of the factors use the 'valueLabels' function
```

```
valueLabels(d)
```

```
$Sex
[1] "Sex1" "Sex2"

$Drug
[1] "Drug1" "Drug2"
```

Valid variable types Default is that the valid variable types are as given by the function validVarTypes():

```
validVarTypes()
```

```
[1] "Discrete"   "Ordinal"    "Continuous"
```

The valid variable types can be extended. This could be of relevance if e.g. a variable y takes only strictyly positive values and should be "read as" $\log y$. Then we can extend the valid variable types as:

```
oldtypes <- validVarTypes()
validVartypes <- function() c(oldtypes, "PosCont")
validVartypes()
```

```
[1] "Discrete"   "Ordinal"    "Continuous" "PosCont"
```

3.3 Editing gmData objects

The information contained in a gmData object may be accessed or modified by the methods: varTypes, varNames, nLevels, latent, valueLabels, and observations. For example, to redefine the levels of the variable Sex, we can do:

```
observations(gmdRatsNodata) <- rats
valueLabels(gmdRatsNodata)$Sex <- c("Male","Female")
valueLabels(gmdRatsNodata)
```

```
$Sex
[1] "Male"   "Female"
```

```
$Drug
[1] "D1" "D2" "D3"
```

3.4 Writing new conversion methods

It is also possible to write conversion methods for other data types, if needed. Suppose we have a 2×2 table from cross classifying factors `Aa` and `Bb` and that the counts (in some order) are 12, 20, 33 and 41. We may represent data as e.g.

```
d <- list(varNames=c("Aa","Bb"), nLevels=c(2,2), counts=c(12,20,33,41))
class(d) <- "countsList"
d
```

```
$varNames
[1] "Aa" "Bb"

$nLevels
[1] 2 2

$counts
[1] 12 20 33 41

attr(,"class")
[1] "countsList"
```

A conversion method can be defined as

```
as.gmData.countsList <- function(from){
   ans <- newgmData(varNames=from$varNames, nLevels=from$nLevels)
   observations(ans) <- from$counts
   return(ans)
 }
```

Then we get:

```
gd<-as.gmData(d)
```

```
  varNames shortNames varTypes nLevels
1       Aa          A Discrete       2
2       Bb          B Discrete       2
To see the values of the factors use the 'valueLabels' function
To see the data use the 'observations' function
```

```
valueLabels(gd)
```

```
$Aa
[1] "Aa1" "Aa2"

$Bb
[1] "Bb1" "Bb2"
```

```
observations(gd)
```

```
[1] 12 20 33 41
```

4 The `gModel` class

The general class `gModel` contains a formula object and a `gmData` object. Implementations of different specific graphical model classes can inherit from this class and provide methods for parsing the formula. Here, we illustrate by implementation of a class for hierarchical log–linear models, `hllm`.

For a hierarchical log–linear model, we use the following formula language. The right hand side of the formula is a list of the generators separated by '+'. A generator is specified by variable names with separated by '*'. Commonly used models have short hand notations: saturated model (`~.^.`), main effects (`~.^1`), all kth order interactions (`~.^k`). By an optional argument, `marginal`, it is possible to specify a subset of the variables from the `gmData` object.

The saturated model

```
m1 <- hllm(~.^.,gmdHec)   # saturated
formula(m1)
```

```
~Hair * Eye * Sex
<environment: 0x0000000009495820>
```

The model where sex is independent of hair- and eye-color

```
m2 <- hllm(~Hair*Eye+Sex,gmdHec)
```

The model with all main effects

```
m3 <- hllm(~.^1,gmdHec)   # all main effects
formula(m3)
```

```
~Hair + Eye + Sex
<environment: 0x0000000008ec0948>
```

The saturated model in the hair-eye marginal

```
m4 <- hllm(~.^.,gmdHec,marginal=c("Hair","Eye"))
formula(m4)
```

```
~Hair * Eye
<environment: 0x0000000008d5e6e0>
```

Also, the `gModel` class will have associated methods for making inference, which will be treated in Section ??.

4.1 Model editing

One important aspect of graphical modelling is the ability to interact with the model. Editing the model means `e.g.` that edges are added or removed and the resulting model is further investigated. The package developer needs to provide the methods `addEdge` and `dropEdge` for his model class.

In addition, variables may be added or deleted from the model by the methods `dropVertex` and `addVertex`, which should also be provided by the package developer.

It is up to the package developer to define the body of these methods. The output should be an object similar to the input object. If for example the input object is a fitted object, the returned object should also be fitted with the same engine.

Acknowledgements

The members of the gR project are acknowledged for their inspiration.

References

Claus Dethlefsen and Søren Højsgaard. A common platform for graphical models in R: The gRbase package. *Journal of Statistical Software*, 14:1–12, 2005.

David Edwards. *Introduction to Graphical Modelling.* Springer-Verlag, 2nd edition, 2000.

Ronald D. Snee. Graphical display of two-way contingency tables. *The American Statistician*, 28:9–12, 1974.