DSC 2001 Proceedings of the 2nd International Workshop on Distributed Statistical Computing
March 15-17, Vienna, Austria
http://www.ci.tuwien.ac.at/Conferences/DSC-2001
K. Hornik & F. Leisch (eds.) *ISSN 1609-395X*

Modeling with Objects

Robert Gentleman

Abstract

The 1991 revision of the S language [2] introduced several new concepts. In this paper we show how three of these: *dataframes*, *formulas*, and *object oriented programming* can be used to simplify the manipulation of complex data. We consider the case where the data contain a longitudinal component and show how using an appropriate data structure provides an abstraction that allows these data to be handled and modeled in a very natural way. The ideas extend to other situations where components of the data collected have complex structure.

1 Introduction

In 1991 a major revision of the S language was released, [2]. In that release several important new concepts were introduced. In this paper we consider ways of using three of these to provide an easier to use modeling paradigm. The three concepts that we will consider are *dataframes*, *formulas*, and *object oriented programming*. In order to make the ideas more concrete a simple example for modeling longitudinal data will be created.

We will show how these three concepts allow us to easily fit models to complex data in a very natural way. These concepts extend to other situations with complex non–standard data. We will explore other uses for these ideas as well. These ideas can be used in either implementation of the S language.

Dataframes are generally thought of as (slightly) enhanced matrices. While this is true, we should think of them as rectangular arrays containing arbitrary elements. The flexibility of a dataframe is greatly enhance by the observation that an element of a data frame can be any S object; including a dataframe.

A dataframe constitutes a collection of objects that are arranged in a cases by variables format. While this can be restrictive it is not as restrictive as is commonly believed. Within a column of a dataframe all the objects must be of the same basic type, but that type is arbitrary. We will show how data which do not, at first glance, appear to be rectangular can be stored in a rectangular format through the use of objects. Dataframes are the fundamental data structure used for fitting models in S. While they have some shortcomings and one can certainly argue about some of the details of their implementation they by and large have provided a very useful service.

A great deal of abstraction is gained by using complex objects to store data. In this paper we consider a simple class of objects designed to represent collections of longitudinal data. However, other situations which would lend themselves to this approach include using *connections* to represent large data objects or using the notion of a *proxy* connection (as provided in `RPgSQL`) to an object stored in a database. For these two examples we want to use R and its modeling capabilities on large data sets without having to manipulate the data sets in R's internal storage.

This work was motivated by a manuscript written by J. Lindsey [3] and by an examination of the modeling framework developed for the package `nlme`, [1]. The discussion in this paper is intended to help find a more general solution to the problems solved in these other works. In this paper we consider only the notion of using objects to represent complex data structures. The separate issue of allowing the user to explicitly specify parameters in formulas is touched on lightly but for the most part this issue is left to future research.

2 The Example

We consider the following example throughout this paper. We have data on some individuals that we would like to model. A longitudinal data object consists of a short time series of observations on some covariate. We will model this using objects with a time component and a covariate component. If several time–varying covariates are collected at the same time it may be sensible to make the covariate component of the object a matrix.

For our example we will assume that the response is not longitudinal but this is only for simplicity. The methodology described here is capable of dealing with objects as responses. We assume that for each observation a number of covariates have been collected. Some of these can be simple but we assume that at least one is longitudinal. We make no assumption about the length or times at which observations are made. These can vary from person to person and (within a person) from covariate to covariate. We do assume that for any covariate the times are in the same units across individuals and generally it will be simpler if all longitudinal objects in a dataframe have the same time scale.

A simple, realistic setting consistent with the preceeding description is the study of repeated bouts of treatment (or hospitalization) for asthma. Fixed (or per subject) covariates could include gender, age, and location. Longitudinal covariates could include daily pollen counts, temperature, or patient level variables such as treatment dosage. In this setting each individual has a time–varying response, some fixed covariates and some time–varying covariates.

It is not uncommon for these data to be arranged in a large rectangular array with missing values filled in as required or alternatively in a ragged array. Anyone who has had to manipulate longitudinal data represented in either of these formats knows how cumbersome and error prone the manipulations can be. The analyst must remember a great deal and small changes to the analysis or the data can create time consuming programming requirements.

One can argue that most of the difficulties are caused by the choice of data structure. The data storage representation does not match the physical representation and as a result the manipulation of these data becomes problematic. Many of the storage and handling issues can be removed by storing the longitudinal observations as a longitudinal object. Using this paradigm each individual has the same number of covariates and using a dataframe to store the data is both possible and useful. Later we will see that this abstraction makes the modeling aspects much simpler as well.

With the adoption of a longitudinal data object we can store the data in a rectangular format — or more precisely in a dataframe. The issue of how to put raw data into the longitudinal objects is no less complex than the issues that arise when using the raw data without the adoption of data objects. However, once these issues have been resolved we can then use the standard modeling paradigm. The savings really come from this point forward in the analysis process. We will not address the issue of populating the data structures further in this paper except to note that any solution to the problem requires the adoption of some form of markup language and in that case adopting an XML strategy may be advantageous.

3 Longitudinal Data

We adopt a very simply structure for a longitudinal object. These objects will have class `lgtdl` and they will have two components; a `time` component and a covariate component. The two components must be the same length — we must have a covariate value for each observation time. The name of the covariate should correspond to the name of the variable being collected. A `lgtdl` object will respond to requests for its `time` component and for its `cov` component. The user should not make assumptions about the implementation of the object. It is worth reiterating that there is no presumption that the times are the same for all individuals or that they are evenly spaced. They should simply be the times at which the covariate was measured. We note that the covariate could, in principle, be an array but in our examples it will always be a vector. Finally we note that access to the covariate component is through the `getcov` function.

In the implementation discussed here the `lgtdl` class will inherit from the `data.frame` class. This means that objects of class `lgtdl` will behave like dataframes unless we specifically override that behavior. In some situations it may be more appropriate for `lgtdl` objects to behave as if they were times series. In those cases an alternative implementation (probably with a different class name) would have the objects inherit from the `ts` class.

We will store the entire data set in a dataframe. The details of how to do this are given in the next section. Each individual will be represented by a row in the dataframe. They may have many different `lgtdl` objects associated with them. For example the response might be times at which they need an inhaler. The covariates could include daily pollen counts, blood pressure, pollution counts, stress, gender, etc. There is no need for the longitudinal objects to share a common time frame. We simply construct a new one for each different time–varying covariate. However, when they do not some caution on the part of the data analyst will be needed.

When modeling longitudinal data we are often interested in functionals of the time–varying covariates. For example, the response may depend on the maximum value of the covariate over a time interval or on the value of the covariate at a particular point in time, or perhaps on the integral of the covariate over time. We obtain values for these functionals by applying *methods* to the `lgtdl` objects.

When the time–varying covariate is represented as an object it becomes easy to manipulate it using generic functions and their associated methods. The various functionals of interest can be programmed up as methods for the appropriate task and class of object. These need to be written only once for the class and can then be used in a variety of situations. Using methods encourages code reuse and can make it easier to provide efficient, correct implementations since the programming needs only be done once.

Probably the most important functionals will be those that estimate the value of the longitudinal covariate at some fixed points in time. Various interpolation methods will need to be implemented. Since these functions are methods and are likely to be reused it is worth doing this once with some care.

Another advantage to using objects to represent longitudinal data is that we can define other methods for them such as plotting methods like: `plot.lgtdl`, `lines.lgtdl`, and `points.lgtdl` provide very useful abstractions for graphing the data.

4 Dataframes

In S a dataframe is an array of objects where the cases or observations are the rows and the columns are variables. Within a column all the data must be of the same type. Dataframes are the fundamental data structure used by most of the modeling software.

Dataframes can be argued to have some deficiencies but in large part these are implementation issues. For example, the fact that all non–numeric data are converted to factors when installed in a dataframe upsets a number of users. Another implementation decision that causes some problems is the behavior when the data argument is a list. In that case it is presumed that each element of the list is intended to be a column of the result. This makes it difficult to install complex structures into a dataframe.

Both of these problems can be overcome by the use of a special operator, `I()`. Any argument to `data.frame` that has this operator applied to it will be placed in the dataframe unchanged. When such a column is extracted from a dataframe it has class `AsIs`. Software that interacts with dataframes must be aware of this behavior and implement appropriate methods for dealing with it.

In an interactive S session typing the name of a dataframe causes it to be printed on the screen. The process involves calling the print method for the object. When a dataframe contains complex objects some consideration of how the object will be printed must be taken. The designer of the software can specify how a cell containing a complex object should appear when the dataframe is printed through the generic function `toString`.

For `lgtdl` objects I have chosen to print the string `lgtdl` and the length of the time series. Users can override this decision by implementing their own version of `toString.lgtdl`. The method is given below.

```
toString.lgtdl <- function (x, width, ...)
{
    n <- dim(x)[1]
    return(paste("lgtdl, length = ", n, sep = ""))
}
```

5 Formulas

We now turn our attention to the second component we need for modeling complex data. In S a formula is a syntactic structure that is created with the $\sim$ operator. In the R dialect this syntax is given some semantic meaning through the assignation of the currently active environment to the formula. Thus, in R, a formula has access to the environment in which it was created. This rather simple change is actually very useful and will eventually lead to more robust modeling code.

The simplest form of a formula is the representation, `y` $\sim$ `x` . This indicates that the response variable `y` is to be modeled and that model will depend on the covariate `x`. In almost all cases `y` is numeric. However, in many cases the modeling process will be less error prone and simpler if left hand side of a formula were an object. The `survival` packages (`survival5` is the most recent) define an object of class `surv` that can be used on the left hand side of an equation.

The `survival5` example is slightly different from the approach being proposed here. In that case the `Surv` object is actually an n by two matrix with a class attached. This object is used as a column in the dataframe. The proposal being made here is to have n objects, one for each observation.

The fitting of generalized linear models could be greatly enhanced by the introduction of `Binomial` and `Multinomial` classes with well defined properties. Using objects with these classes on the left hand side of a formula would make the modeling more transparent and probably less error prone. Currently when fitting Binomial data one must create a matrix for the left hand side; the first column containing the number of successes and the second the number of failures. However, in other situations one stores the number of trials and the number of successes.

When one of the covariates is a complex object the situation is much simpler. In this case one need only determine which functional, or collection of functionals will be included in the model. The formula is then specified with the methods applied to the variables and the *correct* behavior should occur. Suppose that we have a response, `y`, and a longitudinal covariate, say `pcount`. We think that the response is related to `pcount` at time 10 (in some units) and also to the integral of `pcount` from days 5 through 12. We can specify this with the following formula.

```
y ~ interplinear(pcount, 10)+integrate(pcount,from=5,to=12)
```

We presume the existence of correct implementations of the methods `interplinear.lgtdl` and `integrate.lgtdl`.

6 Parameters

The modeling capabilities of S would be substantially enhanced if formula's could be turned into functions. In the process the user should be able to explicitly indicate which symbols are to be interpreted as parameters and which are to be interpreted as variables. It is interesting to consider a formula from two different points of view. Given the data it provides a simple explanation of the relationship between the response and the covariates and hence aids in the estimation of the parameters. On the other hand, with the parameters fixed it can be thought of as a prediction of the response for new values of the covariate.

A fairly standard, although undocumented, modeling paradigm is to use the following sequence of calls.

```
  mt <- terms(formula)
  mf <- match.call()
  #set some components to NULL
  mf[[1]] <- "model.frame"
  #make it a call to model.frame
  mf <- eval(mf, parent.frame())
  #now mf contains the model frame
  #perform other manipulations
  x <- model.matrix(mt, mf, contrasts)
  #x is the model matrix
```

Unfortunately, once we move away from linear models this sequence of calls is no longer sufficient. The code for `nls` is much more complicated. Most of the complication arises from the fact that with linear models one does not need to explicitly state the parameters. They can be inferred from a symbolic description of the model as was presented in Wilkinson and Rogers [4]. But for nonlinear models this does not hold. With nonlinear models the parameters must be included in the description of the model.

Two approaches to solving this problem have been used in the S language. One was to use parameterized data frames. That is, the parameters, and their values, were associated with the data. R has never really implemented parameterized data frames and the ones in S can be shown to be deficient in some regards. Further, it does not seem that this is the appropriate place to store that information.

The second approach is to attempt some automatic means of determining which symbols represent parameters and which symbols represent data. Such an approach cannot work properly in all settings but is more convenient than having to specify parameters. In many situations the programmer will need to be able to directly specify which symbols represent parameters and which represent data. An automatic scheme could coexist with such a mechanism.

Further, if one wants to consider hierarchical models, graphical models or some Bayesian analyses it seems that we should have an explicit representation for parameters. I currently have no concrete suggestions for what that representation should be. Some research and exploratory testing is needed to find a representation that will allow us to easily move between Bayesian analyses and frequentist analyses. In some situations the parameters will be numeric while in other situations they will be associated with prior distributions. These ideas will be explored in future work.

7 Example

Here we provide some explicit code for the example discussed previously. The package `lgtdl` is available from the `CRAN` web site. But please remember that its purpose is pedagogical and that it does not represent a definitive implementation of these ideas (at least at the time of this writing).

```
>   l1<-lgtdl(time=c(1,3,5), cov=c(4,6,8))
>   l2<-lgtdl(time=c(11,13,15), cov=c(66,45,88))
>
>  v <- data.frame(a=c(1,10),b=I(list(l1,l2)))
#we control the printing of complex objects through
#format and toString
#for lgtdl we print the length (number of obs)
> v
   a                b
1  1 lgtdl, length = 3
2 10 lgtdl, length = 3

#if you extract a column that was inserted with I() then
#it has class AsIs. So any modeling, which is based
> y<-v[[2]]
> class(y)
[1] "AsIs"

#for the lgtdl stuff we define the interpolation methods
#interpprev.AsIs and then on to interpprev.lgtdl
```

8 Remarks

The ideas presented in this paper are quite simple and require virtually no changes to either implementation of the S language. However, they provide a powerful abstraction that will help data analysts fit models to complex data. The major points are:

- Dataframes can contain arbitrary objects.
- Using objects to represent some data structures allows us to represent complex data in a rectangular, cases by variables, format.
- Adopting this paradigm leads to simpler data handling and model fitting.
- The use of objects in the modeling code should lead to more robust model fitting.

References

[1] Douglas M. Bates and Jose C. Pinheiro. *Mixed Effects Models in S and S-Plus*. Springer-Verlag, 2000.

[2] John M. Chambers and Trevor J. Hastie. *Statistical Models in S*. Wadsworth, 1991.

[3] J. K. Lindsey. Data objects and model formulae for the future: some proposals. *Unpublished manuscript*, 1999.

[4] G. N. Wilkinson and C. E. Rogers. Symbolic description of factorial models for analysis of variance. *Applied Statistics*, 22:392–399, 1973.