Statistical Models in S
edited by
John M. Chambers and Trevor J. Hastie
1992
608 pages
ISBN 0-534-16765-9
Wadsworth and Brooks/Cole Advanced Books and Software

Contents:

New programming functionality has been added to the *New S* language since the publication of the *New S* manual in 1988 (The New S language: A programming environment for data analysis and graphics, by R.A. Becker, J.M. Chambers, and A.R. Wilks). By using this extension to the *New S* language, the ten authors of this manual are able to develop a unified approach to the fitting and analysis of a fairly complete collection of response models (traditional and recent). The book represents the first major effort

in this area. I highly recommended it to anyone interested in using New S, or in applying more recent response models, or in research in statistical computing.

The book is well organized and reads more like a single treatise than it does like a collection of papers. The editors and the authors are to be congratulated for a remarkable job. Each chapter is organized into four primary sections. The first describes the statistical methodology, the second how to use the S functions and data structures, the third how to extend or specialize the given software, and the fourth contains more detail on the computations. Consequently, by reading only the first two primary sections of each chapter one comes away well equipped to use the software in a host of applications. For most readers this will be enough. As interest and circumstance demand, the remaining sections of any chapter can be read with profit. Although early chapters are required reading for later chapters, chapters 7 through 10 can be read independently of one another.

The standard response models of the classic linear model (lm) and the generalized linear model (glm) (including quasi-likelihood models) provide the basic intuition for the design of the unified approach. Newer methodologies like tree-based models for classification and regression, local regression models (loess), and generalized additive models (gam) are treated in similar fashion. Here *similar fashion* is an understatement; any common elements of the analysis in these response models are enforced by the design of the software. For example, with the exception of the non-linear model, the fitting procedure of any response model accepts an extended version of the Wilkinson and Rogers notation for specifying the structural part of the model (1973, Applied Statistics, 22, pp 392-399). (A non-linear model must explicitly define its parameters.) While the commonality is emphasised, specialized treatment in specific circumstances is encouraged. For example, to ensure the correct analysis of variance for some experimental designs the formula specification is extended to allow identification of different error sources for analysis of variance data structures (aov).

Common and specialized behaviour for different response models is easily specified programmatically through the two extensions to the New S language described in this book. The first is given by the twin notions of generic functions and specialized methods. As an example, consider the function *anova*. As its first argument, it takes a fitted *model* data structure and produces an anova style table summarizing the fitted model. *Anova* should (and does) work for any fit produced by an aov an lm, a glm, a gam, and a loess fit. By this it is meant that there is some sense in which we

would like to producing an anova-like table for any of these fits. Yet what should be produced will depend on the kind of data structure given as its first argument. If for example a glm fit is given, then an appropriate *analysis of deviance* table is printed. This specialization is achieved by having the *anova* function automatically dispatch to the function *anova.glm* whenever it is presented with a glm fitted model.

The second extension allows arbitrary S data structures to be related to one another through some kind of inheritance. This is implemented by adding a new attribute called *class* on S data structures. For example, a glm fitted model will have as its class attribute the vector (in New S terminology) given by (*glm*, *lm*). Operationally this means that any generic function (e.g. *anova*) that is called on a glm will look first for a function of the same name but ending in *.glm* (e.g. *anova.glm*) to apply to the argument. If there is one then it is used. If there is not, it looks again but this time for one ending in *.lm* (e.g. *anova.lm*). If the entire vector of class attributes fails to turn up an appropriate function, then finally the ending *.default* is tried (e.g. *anova.default*) – there may or may not be a *.default* method defined.

This extension of New S in the direction of object-oriented programming is important and exciting. The authors, and Chambers in particular, are to be applauded for such a move. The unified approach ti fitting response models is particularly interesting. Ordinarily, I would consider such important work to be above criticism in a book review. But because for many statisticians this version of New S will be their first exposure to the ideas of object-oriented programming I think it is important to highlight some of the weaknesses of the New S approach for the readers.

First, a little history on application of object-oriented programming as applied in statistical computing is in order. In the 1980's a great deal of research work on computing environments for data analysis centred on exploiting the object-oriented paradigm. In 1985 Steve Peters and I wrote a small statistical system called DINDE that was nearly exclusively object-oriented (1988 SIAM journal on Stat. and Sci. Computing). Unfortunately, it required rather specialized hardware. John McDonald at the University of Washington has made publicly available a system called Arizona. The first widely used object-oriented statistical system was Luke Tierney's Lisp-Stat (1990, Wiley and Sons). A new object-oriented statistical system system developed at Waterloo called Quail will be publicly available in March 1992.

Thus New S, represents an important development in the trend of statistical analysis environments becoming object-oriented. One critical thing that distinguishes it from others is that the developers have had to add

the object-oriented aspect to an existing statistical system. This has the strength that the large community of S users will have access to new possibilities that were previously denied them. the attendant weakness however is that the full power of object-oriented programming is not necessarily realized. As is pointed out in Appendix A of the book, New S has much in common with object-oriented languages *but differs in a number of respects related to the nature of S* (p. 457).

Indeed, to me New S is unlike any object-oriented language I know and consequently cannot (yet?) fulfill the promise of object-oriented programming. At best, New S's functional programming style has been extended so that the user can write functions which dispatch to other functions depending only on the value of the *class* attribute of one of its arguments. True, this makes it possible to write functions which are *generic* but it is a far cry from object-oriented programming. Despite the impression given to the casual reader, there is no such thing as a *class* in this New S; there is merely an attribute called *class* which can appear on any S data structure. The generic functions look to this attribute to decide which one of a collection of New S functions (called methods) to invoke. The dispatching is often called *method lookup* and in the New S model is confused with the definition of a class.

In an object-oriented programming language, classes are data structures which can themselves be manipulated. Minimally, they can be related one to another through the notion of inheritance. As an example consider using classes as data structures to describe birds. We might define a general class called *bird* which would be a template data structure representing the properties held by birds in general. A second class called *flightless-bird* could be introduced to represent birds which have evolved to a flightless state (e.g. penguins and ostriches). It is clear that every element of the class *flightless-bird* is also an element of the class *bird*. It is also clear that the converse does not hold; an element of the class *bird* is not necessarily also an element of the class *flightless-bird*. This distinction is reflected in the software by asserting that *flightless-bird* is a subclass of *bird*. Consequently any property of *bird* is inherited by *flightless-bird*. If I had a pet ostrich called Frank, he would be represented in this system as an *instance* of the class *flightless-bird* – a *flightless-bird object*. A generic function that operated on birds might be *fly* which would cause the bird-object to fly from its present position to a new specified position. If applied to the object representing Frank however nothing should happen because Frank is a *flightless-bird*. This is implemented in software by defining a generic function called *fly*

and separate *fly* methods for each of the classes *bird* and *flightless-bird*. The method lookup procedure typically traverses the inheritance hierarchy of the classes to determine which is the most specific method for a given argument to the generic function call. In some systems, this lookup can be redefined.

In the extended New S system, Frank would be represented by making a *bird* data structure and pushing the string *flightless-bird* onto its class attribute vector. No class called *flightless-bird* would exist as a data structure. A separate New S function, fly.flightless-bird, would be defined to represent the fly method for flightless-birds. So far so good. The problem is that because no classes exist, there is absolutely no enforcement of an inheritance hierarchy. In New S style of *object-oriented programming*, objects can be rooutinely created that have contradictory class information. For example, consider two New S *objects*, one having class attribute (*flightless-bird*, *bird*, *animal*) and another with class attribute (*flightless-bird*, *moving-van*, *telescope*). As the class hierarchy is ordered from child to parent to grandparent and so on as one proceeds left to right, both have as their primary class *flightless-bird*. The class *flightless-bird*, like any class in the New S extension, is completely without meaning. The class attribute only determines the method lookup to be used for a particular instance. It would be better named the *method-precedence* attribute.

The absence of the existence of classes and the consequent meaninglessness of a class in New S may be the reason that all of the method-precedences defined for New S models seem completely backward to me. For example, consider implementing generalized linear models (glm) and standard linear models (lm) with genuine classes. Because a linear model is really a special kind of generalized linear model one might naturally define two classes, say *lm* and *glm* and assert that *lm* is a specialized subclass of *glm*. As a consequence, whatever property one expects of a *glm* would also be found on an *lm* since it is simply a special kind of *glm*. through *inheritance*. The two models have statistical meaning and relationships; having class structures preserves and enforces this meaning. By contrast, in the extended New S system, the class attribute of a *generalized-additive-model object* or *gam* is defined to be (*gam*, *glm*, *lm*). I would place (as we have in the Quail system) *gam* at the top of the hierarchy and *lm* at the bottom. The class attribute of a gam would be simply (*gam*) while that of a linear model would be (*lm*, *glm*, *gam*).

I might add that as a method dispatching facility, the New S implementation can dispatch only on the basis of the type of one of its arguments. There are many situations where this is a handicap and one would like dis-

patching to depend on the type of any number of the arguments to a generic function. In many object-oriented systems this is possible, but it is difficult to see how the New S could be extended yet again to accommodate this kind of method-lookup.

In summary, the book and the attendant software are interesting, valuable, and important. The book should be of interest to a wide audience. Again the authors are to be congratulated. As an object oriented programming system the extended New S system is unusual and in this reviewers opinion falls short of fulfilling the promise of object-oriented programming.

R.W. Oldford
Department of Statistics and Actuarial Science
University of Waterloo
Waterloo, Ontario
N2L 3G1
Canada

February 1992