Extending Arc

Sanford Weisberg

School of Statistics, University of Minnesota, St. Paul, MN 55108-6042.

Supported by National Science Foundation Grant DUE 96-52887.

August 15, 2000

Abstract

This paper describes code that can be written to extend various features of *Arc.* Several accompanying files of computer code are available from www.stat.umn.edu/arc/addons.html When unpacked, you will have a directory called `ExtendingArc` that includes the files you need. www.stat.umn.edu/arc/addons.html.

Arc is a *Xlisp-Stat* application for the analysis of regression models that study the dependence of a response y on a set of predictors x. Most of the methodology in the basic *Arc* is described in Cook and Weisberg (1999). An earlier version of *Arc* called *R-code* was described in Cook and Weisberg (1994); *Arc* should be used in place of *R-code.* You need to write code in *Xlisp-Stat* to extend *Arc.* For a description of the object system used in *Xlisp-Stat*, see Tierney (1990). Steele (1990) provides a complete reference guide to common *lisp*, which is the language used in *Xlisp-Stat*.

This paper describes methods for extending *Arc* in several useful ways. We describe first how a new fitting method can be added to *Arc*, using work by Jorge de la Vega on fitting ordinal regression models as an illustration. Several other useful changes are described as well.

1 The Basics

1.1 The Arc prototype

It is helpful to think of *Arc* as a hierarchical structure of objects. At the highest level of the hierarchy is the single object called `arc`; if you simply type `arc` into the text window, the object will be returned. This object keeps track of global constants, gives access to global methods like the help system and computing quantiles and percentage points of probability distributions. Before you read in any data files, all interaction with the program is really with the `arc` object. Developers are unlikely to want to change anything about the `arc` object.

The `arc` object has methods for reading data files and creating *datasets.* *Arc* keeps a list of all the data sets currently in use, and has methods that allow for switching between datasets. The commands

```
> (send arc :datasets)
> (send arc :active-dataset)
```

return, respectively, a list of all the datasets currently loaded into the program, and the currently active dataset.

The complete code for the `arc` prototype is given in the file `arc1.lsp`.

1.2 Dataset-proto

When you read a data file into *Arc*, several objects are created. For example, load the file `hald.lsp` that comes with the program, using the Load item in the Arc menu. The following objects are then created:

1. A `dataset-proto` object is created. The name of this object is either specified by a name given on the data file (the line `dataset = hald` specifies that the name of the dataset is `hald`), or is set in a dialog. All datasets must have unique names. The dataset is added to the list of datasets kept by the `arc` object; in addition, typing `hald` in the listener will return the object.

2. Each variable (column of data) in the data file becomes an object of type `datalist-proto`. For the hald data, there are 5 variables. You can get a list of the datalists, and their names, using the methods:

```
> (send hald :datalists)
(#<Object: 405d7af8, prototype = DATALIST-PROTO>
 #<Object: 405db2d8, prototype = DATALIST-PROTO>
 #<Object: 405d8be8, prototype = DATALIST-PROTO>
 #<Object: 405deb08, prototype = DATALIST-PROTO>
 #<Object: 405dffa8, prototype = DATALIST-PROTO>
 #<Object: 405f4eb8, prototype = DATALIST-PROTO>)
> (send hald :names)
("X1" "X2" "X3" "X4" "Y" "Case-numbers")
```

   Each datalist has a name, data, a type, such as variate, text, factors, interactions, and others; information on missing data symbols, and a text string that describes how the data list was created or what it represents. Developers may have specialized datalists, such as a survival indicator for survival analysis; Section 4 shows how to add new data list types. The last of these datalists, of type `:case-numbers`, is generated automatically by *Arc*. Other special variables of this type include a list of all ones.

3. Each dataset-proto has two associated menus, a dataset menu, which has the same name as the dataset, and a Graph&Fit menu that that provides access to several graph types and to various models for examining the data set. We will show later how to modify both of these menus by adding items for special purposes.

4. Models are created from a dataset's Graph&Fit menu. The model will usually have a menu of its own, giving access to methods that act on the model, but probably not on raw data. The dataset keeps a list of all the models created from it, and this provides a mechanism for the models to interact with each other. For example, with the Hald data, fit the linear regression of Y on the other variates, and then type

```
> (send hald :models)
(#<Object: 3d9a8e8, prototype = NORMALREG-PROTO>)
```

which shows hald currently knows about only one model, and it is of type `normalreg-proto`, which is a generalized linear model with normal errors.

5. Graphs are created from the dataset's Graph&Fit menu or possibly from a model's menu. The items in the Graph&Fit menu list the types of plots that are available in *Arc*: histograms, 2D and 3D scatterplots, boxplots, scatterplot matrices and multipanel plots (which are just histograms or 2D plots with an added slidebar that will replace the values on one of the axes when the slidebar is clicked). This plots will consist of raw data from the dataset or of quantities that can be obtained from the fit of a model, like residuals or fitted values. Plots that require quantities that can be derived from a fit of a model are specified in the model's menu. These latter plots include, for example, added-variable plots and *CERES* plots. When most graphs are created, a function is also created that, when called, will recompute the data in the graph. This permits graphs to be updated when data are changed (for example, by deleting an observation). Draw two graphs from the Graph&Fit menu for the Hald data, and then type

   ```
   > (send hald :graphs)
   (#<Object: 39e4298, prototype = HISTOGRAM-PROTO>
    #<Object: 3a8a5d8, prototype = SCATTERPLOT-PROTO>)
   ```

   so the first graph is a histogram and the second a scatterplot (the newest graph is first in the list of graphs). If you remove a graph from the screen, then that graph is removed from the list of graphs.

2 Adding a Fitting Method

We describe first how to add a new fitting method, and for this we will use the example of fitting an ordinal response model, as written by Jorge de la Vega. The basic procedure to follow is:

1. Create a *stand-alone* prototype that, when given the data of the correct type, will do the computations you want. This prototype need not have any reference to *Arc* at all. The prototype needs to have the following methods:

   (a) An `:isnew` method that creates a new instance of the prototype. This method should return the object itself, so the dataset proto can keep the name of the object in a list.

   (b) A `:compute` method that does the required computations.

   (c) A `:needs-computing` method that returns `t` if the results need to be (re)computed and `nil` otherwise. For example, when a point is deleted from a graph, the dataset prototype changes the value of `:needs-computing` to `t` for all of its models. Then, when the results of the model are redisplayed, or used in a graph, the model will know it needs to recompute itself.

   (d) An `:included` method that is a list of n elements with ith element equal to `t` when the ith case is to be used in computations, and `nil` if it is to be skipped.

(e) A `:display` method that will display any printed output in the text window.

(f) *Arc* codes missing values using the code for not-a-number (returned with the function `(set-missing)`, usually equal to $-\infty$. When data is passed to your prototype by the dataset proto, any case with missing values will have the corresponding value of `included` set to `nil`. You should be sure that your prototype will not choke on the missing values. On the Macintosh and Unix, performing any arithmetic with not-a-number returns not-a-number, which is what one would like. On Windows, however, arithmetic with not-a-number sometimes returns an error message, which is not what is wanted. Consequently, one must check that Windows does the right thing with missing values.

The prototype can inherit from `regression-model-proto` but there is no requirement for this to be true. The example we present below inherits from `glim-proto`, which in turn inherits from `regression-model-proto`.

You may find it convenient to write a function that, when called with the appropriate arguments, will do all the computations you want, put values into important slots, and return an object.

2. Once you have a working method, you need to write the interface with *Arc*. You will probably want to keep to the style of other fitting methods in *Arc*, which means that you will want to have a menu item for your fitting method, which then opens a dialog, and the results of the dialog are used to create the object you need. You can add an item to the Graph&Fit menu for your object by creating. You need to create an `arc-menu-item` for the new fitting method. You then add this item to the list of items in `*arc-fit-menu-items*`. Every Graph&Fit menu will include your new fitting method.

3. You can modify the dialog included with this article to suit your needs, or write your own from scratch.

4. When the menu item is selected, it will call a `dataset-proto` method. You must write the corresponding method that will be responsible for calling your code.

5. Any graphs created by the model should be created in a special way so *Arc* plot controls will work properly.

6. The model's prototype should have a method called `:plotlist` that when called gives a list of methods that will be used in drawing graphs from the Graph&Fit menu.

All these steps can be carried out by modifying the code that accompanies this paper to fit your situation.

2.1 Ordinal Regression

We will consider fitting ordinal regression models, in particular the *proportional odds model*, Agresti (1990, Sec. 9.4.1). Suppose we have a response variable y that consists of J ordered categories. For example, y might represent a scale of mental impairment with four categories "Well", "Mild", "Moderate" and "Impaired". We will code these as 4, 3, 2, and 1, respectively,

where the ordering is important, but the numbering scheme is arbitrary: the "distance" between Well and Mild might be much greater than the "distance" between Moderate and Impaired. Let $F_j(\mathbf{x}) = \Pr(y \leq j|\mathbf{x})$ be the cumulative probability that y is of category j or less, given the value of the predictors $\mathbf{x}$. Using the notation of Cook and Weisberg (1999), but keeping the intercept separate, the logistic proportional odds model assumes that

$$F_j(\mathbf{x}) = 1/(1 + \exp(-(\alpha_j + \boldsymbol{\eta}^T\mathbf{u}))) \tag{1}$$

where $\mathbf{u} = \mathbf{u}(\mathbf{x})$ is a set of *terms* derived from $\mathbf{x}$ (see Cook and Weisberg, 1999, Chapter 8, for the reason for substituting the terms $\mathbf{u}$ for the predictors $\mathbf{x}$). Let $L_j(\mathbf{x}) = \log(F_j/(1 - F_j))$ be the logit transformation of F. In the logit scale the model is linear,

$$L_j(\mathbf{x}) = \alpha_j + \boldsymbol{\eta}^T\mathbf{u} \tag{2}$$

This model implies that

$$L_j(\mathbf{x}_1) - L_j(\mathbf{x}_2) = \boldsymbol{\eta}^T\mathbf{u}_1 - \boldsymbol{\eta}^T\mathbf{u}_1 = \boldsymbol{\eta}^T(\mathbf{u}_1 - \mathbf{u}_2) \tag{3}$$

The name proportional odds model is used for this model because (3) is independent of j and so in logit scale the regression lines are all parallel.

Another view of the proportional odds model is to assume that there is an underlying continuous version y^* of y, such that we observe $y = j$ if $\alpha_{j-1} < y^* \leq \alpha_j$, where $\alpha_0 = -\infty$. As discussed by Agresti, this procedure will produce estimates of the cut-points α_j as well as of the coefficient estimates for the other terms.

To fit this model, the user must specify a response variable, and a set of predictors or terms $\mathbf{u}$. In the fitting, we also require a list of frequencies, the number of observations with a particular value of y and a fixed value of $\mathbf{u}$. Finally, the model presented here uses the logistic kernel mean function, but others like the probit can also be used. When these values are specified, the prototype should fit the model (in this case, using maximum likelihood estimation), estimates and standard errors and an estimate of overall fit should be returned. Also, one should be able to obtain ancillary statistics as needed for graphical methods.

2.1.1 The Function

Jorge de la Vega translated a Matlab function by Johnson and Albert (1999) into lisp to fit the proportional odds model. Here is the first line of the function:

```
(defun ordinalmle (w x f &key (link 0) (maxiter 30) (tol 0.0001))
```

This function takes as input a $n \times k$ matrix w of zeroes and ones such that the ith row of this matrix has exactly one non-zero entry indicating which category was observed. The second argument is an $n \times p$ matrix of terms, including a column of ones for the overall intercept, assumed to be of full column rank. The list `f` of length n gives the frequencies, giving the number of times a particular category occurred with a given set of terms. This is a stand-alone function that uses none of the features of *Arc.* It does not allow missing values, deleted cases, or other modifications. If the computing algorithm converges, the function returns a list of values, given by

```
    (return (list
             :deviance (select deviances 0)
             :deviances (select deviances 1)
             :coef-estimates alpha
             :fit-values mu
             :cov-matrix covar
             :beta beta)))
```

Rather than just returning a list of unlabeled values, the list returned has a name of a quantity (preceded by a colon) followed by the value that was computed for it in the function. This simplifies working with the function and making finding errors easier.

It is not necessary to start by writing a function that does the computation. Experienced *Xlisp-Stat* users may prefer to build a stand-alone prototype for computing everything.

2.1.2 The prototype

Given a function that can compute the maximum likelihood estimates and related statistics, we next define a prototype that will supply the data to the function, and then have helper methods to access the results through printing and plotting. The prototype is defined as follows:

```
(defproto multiordinal-model-proto
      '(fit needs-computing frequencies frequencies-name)
      () glim-proto)
```

The name of the prototype is `multiordinal-model-proto`. It inherits from `glim-proto`, which in turn inherits from `regression-model-proto`, and so it has all the methods and slots defined for those prototypes. We recommend that you follow this procedure even if your model is quite different from a glim-proto. This prototype has four *slots* of its own, frequencies, frequencies-name, fit and needs-computing. All other slots that it will used are inherited from its ancestors.

In *Xlisp-Stat*, an `:isnew` method is responsible for creating an object and filling it with data. The `:isnew` method for `multiordinal-model-proto` is

```
(defmeth multiordinal-model-proto :isnew
     (&rest args &key pweights weights-name
          (frequencies pweights) (frequencies-name weights-name))
 (send self :needs-computing t)
 (send self :frequencies frequencies)
 (send self :frequencies-name frequencies-name)
 (apply #'call-next-method args))
```

Apart from the way estimates are computed, the structure of the multinomial ordinal model is very similar to a generalized linear model. For example, both will have a matrix of terms stored in the `:x` slot, a list of responses stored in `:yvar` (the multinomial model with translate these category labels to the necessary matrix of labels). Both use iterative calculations, and the count limit on iterations and the convergence criterion can be inherited by the new method from the old one. The multinomial models require frequencies, which are not used by generalized linear models. The strategy we follow for now is to equate the glm's weights with the multinomial frequencies; later, we separate these.

In the `:isnew` function above, the slots for frequencies (the values) and frequencies-name (the name of the variable providing the frequencies) are explicitly set. This prototype has its own `:needs-computing` method, as glim-proto's method for this depends on using the glim-proto `:compute` method. Finally, the last line of the method, `(apply #'call-next-method args)`, calls the `:isnew` method for glim-proto, and that method will fill all the other necessary slot-values.

2.1.3 The Compute Method

The bulk of the computing is done by the `ordinalmle` function discussed earlier. The job of the `:compute` method is to (1) reset `:needs-computing` to `nil`; (2) obtain the arguments w, x and frequencies to pass to the function, deleting all non-included or missing cases; (3) store the results for other methods to access. Here is the compute method:

```
(defmeth multiordinal-model-proto :compute ()
  (call-method regression-model-proto :compute)
  (send self :needs-computing nil)
  (let* ((w (send self :w))
         (x (send self :x-matrix))
         (frequencies (select (send self :frequencies)
                              (which (send self :included))))
         (link (if (eq (send self :link) logit-link) 0 1))
         (result (ordinalmle w x frequencies :link link
                   :count-limit (send self :count-limit)
                   :tol (send self :epsilon-dev))))
    (if (not result)
        (error "Convergence failed, maximum iterations exceeded."))
;include command to remove model if error.
    (setf (slot-value 'fit) result)
    t))
```

The first step in the compute method calls the regression-model-proto `:compute` method. The purpose of this is to determine a full-rank subset of the terms for use by `ordinalmle`. This is not a fool-proof way to get the rank, and in fact the rank of the problem might be lower than expected. The next line updates `:needs-computing`, and then the matrices w (computed from the responses) and x (computed from the predictors) is obtained. The other inputs to `ordinalmle` are obtained as well, and the function is executed. If there is a result, then the result is put into the slot value `fit`. The `:compute` method doesn't do any printing.

2.1.4 Accessor/Helper Methods

Several accessor/helper methods are required. These functions either access slot values, or recover particular parts of the fit. Of particular interest is the method `:get-value` that is called by many of the other methods to recovered parts of the fit.

```
(defmeth multiordinal-model-proto :frequencies
           (&optional (new nil set))
    (if set (setf (slot-value 'frequencies) new))
    (slot-value 'frequencies))
```

```
(defmeth multiordinal-model-proto :frequencies-name
           (&optional (new nil set))
    (if set (setf (slot-value 'frequencies-name) new))
    (slot-value 'frequencies-name))

(defmeth multiordinal-model-proto :get-value (what)
    (when (send self :needs-computing) (send self :compute))
    (let ((pos (position what (slot-value 'fit) :test #'eq)))
     (select (slot-value 'fit) (1+ pos))))

(defmeth multiordinal-model-proto :needs-computing
           (&optional (new nil set))
    (if set (setf (slot-value 'needs-computing) new))
    (slot-value 'needs-computing))

(defmeth multiordinal-model-proto :beta ()
    (send self :get-value :beta))

(defmeth multiordinal-model-proto :deviance ()
   (send self :get-value :deviance))

(defmeth multiordinal-model-proto :fit-values ()
  (let* ((inc (which (send self :included)))
         (fit1 (column-list (send self :get-value :fit-values)))
         (n (length (send self :included)))
         (k (length (first fit1)))
         (ans (repeat (list (repeat (set-missing) n)) k)))
    (apply #'bind-columns
     (mapcar #'(lambda (full part) (setf (select full inc) part))
         ans fit1))))

(defmeth multiordinal-model-proto :coef-estimates ()
    (send self :get-value :coef-estimates))

;;; the next method makes the standard display variances item work
(defmeth multiordinal-model-proto :xtxinv ()
    (send self :get-value :cov-matrix))

(defmeth multiordinal-model-proto :cov-matrix ()
    (send self :get-value :cov-matrix))

(defmeth multiordinal-model-proto :standard-errors ()
    (sqrt (diagonal (send self :cov-matrix))))

(defmeth multiordinal-model-proto :deviances ()
  (let* ((inc (which (send self :included)))
         (dev (send self :get-value :deviances))
         (n (length (send self :included)))
```

```
        (ans (repeat (set-missing) n)))
  (setf (select ans inc) dev)
  ans))

(defmeth multiordinal-model-proto :response-categories ()
"Method args: ()
Returns the labels of the response categories."
  (remove-duplicates (qsort (send self :y)) :test #'eq))

(defmeth multiordinal-model-proto :w ()
"Method args: ()
Returns a matrix of the category indicators with deleted cases excluded."
  (let* ((cats (send self :response-categories))
         (values (iseq 1 (length cats)))
         (y (recode-values
              (select (send self :y) (which (send self :included)))
              cats values)))
     (flet ((dummy (val) (mapcar #'(lambda (v) (if (= v val) 1 0)) y)))
        (apply #'bind-columns (mapcar #'dummy values)))))

(defmeth multiordinal-model-proto :parameter-names ()
"Method args: ()
Returns a list of names of the parameters, including the intercept,
followed by category cutpoints."
  (let* ((i (list "Intercept"))
         (v (select (send self :predictor-names) (send self :basis)))
         (c (mapcar #'(lambda (b) (format nil "Cut~a" b))
              (rest (butlast (send self :response-categories))))))
   (append c i v)))

(defmeth regression-model-proto :display-variances
           (&key (correlation nil))
 (let ((m (* (^ (send self :sigma-hat) 2) (send self :xtxinv))))
  (cond
   (correlation
    (format t "Correlation matrix of the coefficient estimates~%")
    (print-correlation-matrix (get-correlation-matrix m)
         (send self :parameter-names) ))
   (t
    (format t
       "Variance-covariance matrix of the coefficient estimates~%")
    (print-covariance-matrix m  (send self :parameter-names))
    (send self :display-variances :correlation t)))))

(defmeth multiordinal-model-proto :x-matrix ()
"Method args: ()
Returns the design matrix, and then removes non-included rows."
  (let* ((x (call-next-method)))
   (select x (which (send self :included)) (iseq (array-dimension x 1)))))
```

2.1.5 The Display Method

The display method gives standard printed output. For the multinomial ordinal model, have estimates of η and of the αs, standard errors, the deviance and the degrees of freedom to print. It is standard in *Arc* to start all output by sending the `:description` method to the model. This prints some useful documentation.

```
(defmeth multiordinal-model-proto :display ()
  (if (send self :needs-computing) (send self :compute))
  (send self :description)
  (let* ((incl (send self :included))
         (n-incl (send self :num-included))
         (predictors (send self :predictor-names))
         (response (send self :response-name))
         (frequencies (send self :frequencies-name))
         (p (length predictors))
         (ps (send self :basis))
         (cats (send self :response-categories))
         (m-2  (- (length cats) 2))
         (coefs (send self :coef-estimates))
         (secoefs (send self :coef-standard-errors))
         (deviance (send self :deviance))
         (link (send self :link))
         (tab (+ 2 (max (cons 8 (mapcar #'length predictors)))))
         (excluded (select (send self :case-labels)
                           (which (mapcar #'not incl)))))
    (format t "~%~%Coefficients of predictors:~%")
    (format t "~va~va~va~va~%"
              (+ 2 tab) "Label   "  16 "Estimate" 14 "Std.Error"
              14 "Est/SE"  )
    (format t "~va~v,4f~v,4f~v,4f~%"
              tab "Constant" 12 (select coefs m-2)
              14  (select secoefs m-2)
              14 (/ (select coefs m-2) (select secoefs m-2 )))
    (dolist (j ps)
     (format t "~va~v,4f~v,4f~v,4f~%"
              tab (select predictors j) 12 (select coefs (+ 1 (+ m-2 j)))
              14 (select secoefs (+ 1 (+ m-2 j)))
              14
              (/ (select coefs (+ 1 (+ m-2 j)) )
              (select secoefs (+ 1 (+ m-2 j)) ))) )

    (format t "~%Cuttof points for categories of ~a:~%" response)
    (format t "~va~va~va~va~%"
                         (+ 2 tab) "Label   " 16 "Estimate" 14 "Std.Error"
                                        14 "Est/SE")
    (format t "~va~v,4f~v,4f~%"
                        tab (select cats 0)12 0 14 0 )
    (dolist (j (iseq m-2))
            (format t "~va~v,4f~v,4f~v,4f~%"
```

```
                tab (select cats (+ 1 j)) 12 (select coefs j) 14
                    (select secoefs j)
                14 (/ (select coefs  j) (select secoefs j)) ))
    (format t "~%Number of cases used:~vd" 15 n-incl)
    (format t "~%Deviance:            ~v,4f~%"  15 deviance)
    ))
```

These methods are sufficient to supply the data and fit a multinomial ordinal model, to display the output in a printed table, and to give the user access to all the computed quantities.

2.2 Interfacing with Arc

To interface with *Arc*, you will want to add an item to the Graph&Fit menu to access your method. Also, you will need to use a dialog to specify which variables you want to use in your model, and after the fit set up a menu for further calculations or graphics. The following function does some of the work for you:

```
(arc-setup-model :menu-item 'multiordinal-menu-item
                 :menu-title (ms "Fit ordinal multinomial model..."
                                 "Fit ordinal multinomial..."
                                 "Fit ordinal &multinomial...")
                :fitname :multiordinal
                :mean-functions (list "Logistic" "Inv-Probit")
                :link-functions (list logit-link probit-link)
                :prefix "MO"
                :types (append *glm-variate-types-default*
                               (list :ones))
                :dialog #'multinomial-dialog
                :make-method :make-glm
                :proto 'multiordinal-model-proto)
```

The argument for `:menu-item` is any symbol you like (as long as it isn't used for something else). The `:menu-title` is the text that will appear in the menu. The function `ms` takes three arguments, and returns its first argument on a Macintosh, second on Unix/Linux, and third on Windows. The & in the third argument underlines the m in multinomial, allowing the use of keyboard shortcuts for this item on Windows. The `:fitname` gives the name you select for models of this type; *Arc* keeps track of models by type.

The next two keywords are relevant for models that have link functions, or, equivalently, their inverses, kernel mean functions. The first of these keywords is a list of text strings that gives short names for the kernel mean function (consistent with the usage in Cook and Weisberg (1999)). The second list gives the corresponding link functions, as used in glim-proto. If you do not use link or mean functions in your method you need not specify these items. The `:prefix` gives the letter used to specify models of this type; for example "L" is used for linear models.

The item `:types` tells the system the types of data you would like to use in the dialog to set up your model. To see the default list, type `*glm-variate-types-default*` in the listener; see also Section 4. For this prototype, we want the vector of ones available in addition to the standard types. The keyword `:proto` gives the name of your prototype for fitting your model.

The keyword `:dialog` allows you to specify a function that should be called to create the dialog you want to use to specify variables. The default is to use the `glm-dialog` that is included in the file glmsetup.lsp. The default is not quite appropriate, since it permits deleting an intercept using offsets, and specifying weights rather than frequencies. A very slight modification of this method is available in the file `ord1.lsp` that creates the correct dialog for this model by deleting the unneeded items and changing the name Weights to Frequencies. The new dialog returns the same arguments oas `glm-dialog`, and this simplifies debugging the program.

Similarly, the `:make-method` specifies the dataset-proto method that will be called to convert the results of the dialog into values that can be used by the prototype. The default is completely adequate for the multinomial ordinal model (if we remember that the argument `weights` contains the frequencies). In other problems (for example, survival analysis might have an indicator for censoring an observation) modification of `:make-glm` might be required. This method is also included in `glmsetup.lsp`.

2.3 The Menu

The `:make-glm` method sends the model that is created the `:menu` method to create a menu. If you don't have a menu of your own, the menu will consist of the items in glm menus. If you want your own menu, you need to specify the items that go in it, using a `:menu-items` method:

```
(defmeth multiordinal-model-proto :menu-items ()
    (mapcar #'(lambda (a) (make-menu-item a self))
                *arc-multiordinal-model-menu-items*))

(rc-menu-item 'multiordinal-summary-item "summary" :display)
(rc-menu-item 'multiordinal-plot-item "Plot probabilities" :plot-fitted-
prob)
(rc-menu-item 'multiordinal-remove-item "Remove model" :remove-menu)


(defparameter *arc-multiordinal-model-menu-items*
    (list  'multiordinal-summary-item
           'display-variances-menu-item
           'multiordinal-plot-item
           'multiordinal-remove-item))
```

This menu has four items. The 'multinomial-summary-item calls the `:display` method. The item 'display-variances-menu-item is used in regression-model-proto, and it finds and prints the variance covariance matrix of the coefficient estimates (using the method `:xtxinv`; by making this method call the `:cov-matrix` method, this menu item can be reused here). The next item calls the `:plot-fitted-prob` method to draw a graph (see below). The final method calls the standard `:remove-menu` item to remove the model from the menu bar.

2.4 Graphics

The method

```
(defmeth multiordinal-model-proto :plotlist () '(:deviances))
```

adds items to the list of quantities that can be plotted from the Graph&Fit menu. Each of the items in this list must correspond to a method in your prototype that, when called, returns a list of n numbers (or numbers and missing value symbols). If you do not have a `:plotlist` method of your own, then the method inherited from glim-proto will be used.

Finally, we give an example of a stand-alone graph for plotting the cumulative probability of success for each of the categories:

```
(defmeth multiordinal-model-proto :plot-fitted-prob ()
"Method args: ()
Plots the linear predictor on the horizontal axis, and fitted
probabilities on the vertical axis."
  (let* ((horiz (matmult (send self :x-matrix) (send self :beta)))
         (vert (transpose (mapcar #'cumsum
                 (transpose (mapcar #'(lambda (a) (coerce a 'list))
                   (column-list (send self :fit-values)))))))
         (dataset (send self :data))
         (gn (send arc :graph-number))
         (or (order horiz))
         (p (send scatterplot-proto :new 2)))
   (send p :start-buffering)
   (send p :variable-label '(0 1)
       '("Linear predictor" "Cumulative probability"))
   (send p :title
      (format nil "[Plot~a](~a) Proportional Odds Model Summary"
      gn (send self :name)))
   (send dataset :intern (format nil "Plot~a" gn) p)
   (send p :add-slot 'owner (send self :data))
   (send dataset :graphs p)
   (defmeth p :close ()
    (send (slot-value 'owner) :delete-graph self) (call-next-method))
   (mapcar #'(lambda (d)
       (send p :add-lines (select horiz or) (select d or))) vert)
   (send p :adjust-to-data)
   (send p :buffer-to-screen)
   p))
```

The plot is first created as a 2D scatterplot; screen buffering is used; axis labels and a title are added. The `intern` method creates a local variable with the same name as the title of the plot, so the user can type methods to the plot. The plot is told about the dataset, and the dataset is told about the plot. This plot does not have the ability to be updated if cases are deleted, but it should.

3 More Generalized Linear Models

If you have read this far, you can see that adding a new generalized linear model type should be very easy. You can figure out how to do this by examining the file `invgau.lsp` that can

be obtained from www.stat.umn.edu/arc/addons.html. This shows both how to define a new link function (inverse kernel mean function), a new error distribution (for the inverse Gaussian distribution), and how to install everything in the menu. The file uses a function called `arc-glm-method`, which has been renamed `arc-setup-model` and is identical to it.

4 New Data Types

The code for data lists is given in the file `datalist.lsp`. The `dataset-proto` keeps track of any number of data lists, models, and graphs. Each data list consists of: (1) a name of the data list; (2) a type for the data list; (3) a slot that hold information about the data; (4) values for the data list and (5) a method for converting the values into a list of lists of data for use with models. The types in *Arc* include:

:variate This is the basic data type, to represent a list of numbers.

:factor Returns $p - 1$ dummy variables when the data in this data list have p distinct values.

:factor1 Returns p dummy variables when the data in this data list have p distinct. values.

:effect Returns $p - 1$ contrasts using "effect" coding, as used in SAS and most other standard statistical programs.

:interaction Returns an interaction with as many degrees of freedom as needed.

:ones A list on n ones.

:case-numbers A list of case numbers.

:case-names A text list of case names.

:text A data list with at least one item that is not a number.

You typically will not need to create your own data types, but this can be useful for some reasons (for example, you may have a dialog that lists several alternatives for different weighting schemes, so you might want a data list of type `:weights` that can be easily identified and put into an appropriate dialog). To create a data type of your own, you need to write two methods, one for `dataset-proto` and one for `datalist-proto`. For example, the `:variate` data type uses this method:

```
(defmeth dataset-proto :variate (&rest args &key data)
"Message args: (&rest args)
Creates a variate or text datalist with data data and name name."
  (let* ((data (coerce data 'list))
         (d (apply #'send datalist-proto :new :variate args)))
        (send self :add-datalist d)
        (send d :name)))
```

A typical call to this method might be

```
(send dataset :variate :data x :name "Fred" :info "Fred's height data")
```

The method makes sure that the argument to `:data` is a list. A `:variate` data list is created. The other arguments `:name` and `:info` are passed to the data list. `:name` is required. You can also pass other arguments, but you will need to adjust the `datalist-proto` to accept them. The `:add-datalist` method checks to make sure that the name does not already exist in the data set.

You need to write a `datalist-proto` method with the same name as the data type. For variates, this is

```
(defmeth datalist-proto :variate (what)
  (case what
    (:values (list (send self :data)))
    (:labels (list (send self :name)))
    (:length (length (slot-value 'data)))
    (:obs-length (length (send self :observed)))))
```

For other types, this can be more complicated. For example, for factors, the method is

```
(defmeth datalist-proto :factor (what &rest args)
  (defmeth self :info () "Factor--first level dropped")
  (case what
    (:values (make-factor (send self :data) ))
    (:labels (factor-levels (send self :data) :prefix (send self :name) ))
    (:length (length (first (send self :values))))
    (:obs-length (length (send self :observed)))))
```

This method first defines the `:info` method for factors to print a meaningful text string. The `:values` are returned as the result of sending the data, a list of factor levels, to the `make-factor` function. Length differs from obs-length if some values are missing. For interactions, the method is

```
(defmeth datalist-proto :interaction (what)
  (case what
    (:values (apply #'rcross-terms
              (mapcar #'(lambda (a) (send a :values)) (send self :parents))))
    (:labels (apply #'cross-names
            (mapcar #'(lambda (a) (send a :labels)) (send self :parents))))
    (:length (send (first (send self :parents)) :length))
    (:obs-length (length (send self :observed)))))
```

This uses the functions `rcross-terms` and `cross-names` to get the values and the list of labels, respectively. The `:parents` gives the names of the variates or factors that were used to define the interaction. With this definition, interactions do not have any data of their own, but use the their parent's data.

5 Adding Smoothers

All 2D graphs have at least two smoother menus: a parametric smoother menu and a nonparametric smoother menu. Histograms have a density estimation slide-bar that is similar. You can easily add new smoothers to these menus by (1) defining the smoother, and (2) appending it to the list of smoothers.

To define a smoother, use the function `arc-smoother`. An example is:

```
(arc-smoother 'my-smooth "My smooth" (rseq .1 1 10) #'my-smooth-f)
```

This function has four required arguments. The first argument is the name of the prototype that defines the smoother. This means that a prototype named `my-smooth` will be created by the call to this function. If needed, you can write additional methods for the prototype. The next item is a short string that will be put into the slide bar to identify the smoother. The third item is a list of values to be put into the slide-bar; the usual values in *Arc* are `(rseq .1 1 10)`. The final item is the name of a function that you must provide that actually computes the smooth. If this function is not `nil`, it must have the calling sequence

```
(defun my-smooth-f (x y b &key weights) ... )
```

where `x` and `y` are the data, and `b` is the bandwidth value read off the slide bar. Alternatively, you can have the computing done in the prototype's compute method. The default method is:

```
(defmeth arc-smoother-proto :compute (x y index &rest args)
  (when (send self :bandwidth index)
    (let* ((b (send self :bandwidth index))
           (ans (funcall (send self :function) x y b)))
   ans)))
```

This method calls the function `my-smooth-f` defined above to do the computing. You may wish to override this default if your smoothing method also computes side effects like standard errors. You may need to modify the value of `b` by multiplying it by scale parameters, sample sizes, and so on. Graphs in *Arc* may have weights attached to each point, and if they are present they will be passed to the smoother through the keyword `:weights`. The function should return a list of two lists (not a list of two vectors), say $(x_0, \hat{y}_0)$, where x_0 is *ordered* and $\hat{y}_0$ are the smoothed values at x_0. The values in x_0 need not be identical to the values in `x`, but they should cover all or nearly all of the range of `x`.

After you create the new smoother, you need to append it to the list of smoothers. For the nonparametric smoothers,

```
(defparameter *nonparametric-smoothers*
  (append *nonparametric-smoothers* (list 'my-smooth)))
```

For parametric smoothers:

```
(defparameter *parametric-smoothers*
  (append *parametric-smoothers* (list 'my-smooth)))
```

Finally, for density estimates, use

```
(defparameter *density-smoothers*
  (append *density-smoothers* (list 'my-smooth)))
```

For example, the file `super-smoother.lsp` in the ftp://ftp.stat.ucla.edu/pub/lisp/xlisp/xlisp-stat/code/statistics/smoothers/supersmoother/ gives an implementation of Friedman's super smoother. It can be added to *Arc* using the following code. First, here is a function that computes the smooth:

```
(defun super-smooth (x y b)
 (let* ((h (if (realp b) (* b (length x)) nil))
        (or (order x))
        (s (send supersmoother-proto :new (select x or) (select y or)
                                     nil h nil)))
  (list (select x or) (send s :smoothed-y))))
```

This function is of the required form. It calls `supersmoother-proto`, ignoring the ability to use weights, and to use "base enhancement." Next, create *Arc* smoother prototype:

```
(arc-smoother 'super-smooth "Super-smooth" '("Opt" .05 .2 .5) #'super-
smooth)
```

This will add an item called "Super-smooth" to the menu. Only four choices for the smoothing parameter are given: "Opt" will choose the bandwidth locally to satisfy an optimality criterion, and .05, .2 and .5 use these fractions of the data. If the value in the slide-bar is not a number (that is, it must be "Opt"), then the value of `h` is set to `nil` in the function `super-smoother`, which is the appropriate value for the calling sequence to `super-smoother-proto`. Finally, define

```
(defparameter *nonparametric-smoothers*
  (append *nonparametric-smoothers* (list 'super-smooth)))
```

Loading this code, and the file `super-smoother.lsp`, is all that is required to use the super-smoother.

6 Adding Items a Menu

In this section, we describe how to add an item to an existing menu. Previously, in the discussion of the multinomial ordinal model, we discussed how to create a menu for a new model type.

6.1 Regression Model Menu

Adding an item to a regression model menu requires two steps. First, create a new `rc-menu-item` for the new item. Then, add the name of the new item to the parameter

```
(defparameter *arc-linear-regression-menu-items*
  (append *arc-linear-regression-menu-items* (list 'new-item)))
```

When the item is selected from the regression menu, it will call the method specified from the regression model. The similar parameters for other models are:

Linear regression	`*arc-linear-regression-menu-items*`
Nonlinear regression	`*arc-nonlinear-regression-menu-items*`
Gen. lin. models	`*arc-glm-regression-menu-items*`

6.2 Dataset and Graph&Fit Menus

Each dataset has two menus, the dataset menu, which has the name of the dataset, and the Graph&Fit menu, which has the items for drawing graphs and fitting models. These menus are designed to be easily extended. The first task is to create an Arc menu item. This is done with the function `rc-menu-item`. For example, the code

```
(rc-menu-item 'plot-of-menu-item
  "Plot of..." :plot-dialog)
```

will create an item called `'plot-of-menu-item` what will print the text "Plot of..." in the menu, and when selected execute the dataset's method `:plot-dialog`. The standard menu items are collected into parameters like the following three:

```
(defparameter *arc-dataset-menu-items*
  (list 'description-menu-item  'display-summary-menu-item
        'table-data-menu-item
        'display-data-menu-item 'display-case-names-menu-item))

(defparameter *arc-data-menu-items*
  (list 'add-to-dataset-menu-item
        'transform-menu-item 'make-factors-menu-item
        'make-interactions-menu-item 'set-case-names-menu-item
        'delete-from-dataset-menu-item
        'rename-datalist-menu-item
        'save-dataset-menu-item
        ))

(defparameter *arc-graphics-menu-items*
  (list 'plot-of-menu-item 'scatterplot-matrix-menu-item
        'boxplot-menu-item
        'multi-panel-plot-menu-item
        'prob-plot-menu-item
        'remove-marks-menu-item ))
```

Append your own menu item to one of these parameters. For example, if you have a new menu item called `'recode-datalist-menu-item` that when called will give you a dialog to recode a variable, you might want to put it with the other data manipulation menu items. You can modify `*arc-data-menu-items*` by

```
(defparameter *arc-data-menu-items*
 (append *arc-data-menu-items* (list 'recode-datalist-menu-item)))
```

For completeness, here is the method that actually creates the menus.

```
(defmeth dataset-proto :make-menu (title)
  (let* ((items (list (mapcar #'(lambda (a) (make-menu-item a self))
                      (combine *arc-dataset-menu-items* 'dash
                               *arc-data-menu-items* 'dash
                               'remove-dataset-menu-item))
                     (mapcar #'(lambda (a) (make-menu-item a self))
```

```
                    (combine *arc-graphics-menu-items* 'dash
                             *arc-fit-glm-menu-items*
                             *arc-fit-menu-items*) )))
      (menus (list (send menu-proto :new title)
                   (send menu-proto :new "Graph&Fit"))))
(mapcar #'(lambda (m i) (apply #'send m :append-items i)) menus items)
(defmeth (first menus) :install ()
 (call-next-method)
 (mapcar #'(lambda (m) (send m :install)) (rest menus)))
(defmeth (first menus) :remove ()
 (call-next-method)
 (mapcar #'(lambda (m) (send m :remove)) (rest menus)))
(first menus)))
```

7 Added Variable Plots

The file `avp.lsp` included with the files of code includes the code that is used to draw added variable plots. We provide some annotation to that code here. The method that is called when the 2D added variable plot item is selected from the regression menu is `:avps`. This method first checks to see how many terms are in the model, and if there is only one, it calls `:avp1`. Otherwise, the added variable plot method will provide added variable plots for all terms in the model, using a slidebar to rotate between the various choices. This is done by setting `index` to be one of the elements of the basis (one of the variables in a full-rank subset of the predictors), initially the first column. The defined function `data` actually computes and returns the quantities that are to appear in the graph; it uses the current value of index to decide which plot is wanted. The function `:make-clone` makes an exact copy of the object for the current model.

Recall that an added-variable plot ofr x_1 is a plot of the residuals from the regression of y on all terms except x_1 (using whatever fitting method is used for the full model; for example, logistic or linear regression) versus the residuals from the linear regression of x_1 on the remaining terms. These quantities are computed in the function `data` defined in the code, with the clone used to compute the first set of residuals and a regression-model used to compute the second.

The `:make-plot` method is the standard method in *Arc* for creating a graph. Its first argument is a function that when called will return the quantities to be plotted. Passing the function rather than the data makes updating the plot easy when the data are changed. The second argument is a list of labels for the axes in the plot. The keyword arguments used here are for the title for the plot and for passing weights, if any, to the plot. The keyword `:mark`, with argument the name of a variable, will set the marks for this and all other graphs; you can't set marks on this graph, but `:make-plot` will automatically add the makes to the added-variable plot if they have already been set.

Next, the slider that selects the various plots is created. All sliders are overlays, and this is created in the standard way. The first argument `basis` gives the numbers of the columns of data that are used. The title here is blank, but in other sliders it is not blank. The length of the slider is determined by the default method `:slider-width`, and varies for Mac, Windows and Unix. The location keyword tells where to put the slider, and the `:locate-next-`

`control` method puts it in the correct place. Finally, the display keyword gives a list of what should be displayed on the slider, in this case the variable names corresponding to the basis. The action for the slider is determined by a `:do-action` method that is also defined in the `:avp` method.

The two methods `:start-next-frame` and `:finish-next-frame` include functions. Whenever the plot is redrawn, the function in `:start-next-frame` is called before the plot is redrawn, and the function in `:finish-next-frame` is called after it is redrawn. This will primarily occur when a point is deleted/restored from this graph or another graph.

The `:do-action` method for the slider is included here as well. It appropriately updates the plot, the regressions and the index, and then calls `:draw-next-frame` to redraw the plot. This completes the code for 2D added-variable plots.

This file also includes the code for 3D added variable plots, which require a dialog to determine what to plot.

8 References

Agresti, A. (1990). *Categorical Data Analysis.* New York: Wiley.

Cook, R. D. and Weisberg, S. (1994). *An Introduction to Regression Graphics*. New York: Wiley.

Cook, R. D. and Weisberg, S. (1999). *Applied Regression Including Computing and Graphics*. New York: Wiley.

Johnson, V. E. and Albert, J H. (1999). *Ordinal Data Modeling*. New York: Springer.

Steele, G. (1990). *Common LISP*, 2nd ed. Digital Press. Available on-line at

http://www-cgi.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/html/cltl/clm/clm.html

Tierney, L. (1990). *Lisp-Stat*. New York: Wiley.