Tools for modelling categorical variables in Lisp-Stat

Thomas Lumley
Dept. Biostatistics
University of Washington
Seattle

March 30, 1996

1 Introduction

This documentation is somewhat disorganised because it reflects three different versions of the tools. They have all been kept because they may all be of some use. The first and most elegant is an object-oriented set, including methods for regression-model-proto and its descendants. The second is the functions ued to implement the object methods. These would be useful for models where a method hasn't yet been written. Finally there are is a set of primitive functions for constructing factor and interaction design matrices.

2 Object oriented methods

`model-formula-proto` is a prototype for a model formula class. To create a new model formula use the `(as-formula)` function

```
(as-formula '((factor a) (factor b) (term x) (interaction (list
a b)) (interaction (list a x) :is-factor (list t nil))))
```

The argument **must** be quoted. I'm sure there is some way of getting round this — the equivalent of `deparse(substitute())` in S but I'm not a good enough LISP hacker to know what it is. Please tell me if you know – it probably involves using a macro rather than a function.

Note that the `:is-factor` keyword in `(interaction)` is optional but that the default is to assume everything is a factor. If one of the variables is continuous and you forget `:is-factor` the results will be obviously wrong (even if you don't run out of stack space). This is the best I can do with a function that accepts lists of numbers as input. The Right Thing to do is probably what S does: force the user to predeclare factor variables as objects and treat everything else as a metric variable. This is not too difficult and I may well do a revised version that makes everything completely object oriented. See the Individual functions section for more documentation — `as-formula` is just an object version of `(formula)`

The formula object responds to messages

`:formula`
:design-matrix
:name-list
:block-names
:block-indices

The first sets or returns the formula (the argument to `(as-formula)`) and recomputes if necessary. The other four return the appropriate things: a design matrix, a list of column labels for it, a list of names for each block of columns (ie each factor, interaction or metric variable), and a list of lists of column numbers for each block.

There is a function `(regression-formula)` which is exactly the same as `(regression-model)` except that it takes a model-formula-proto object instead of the x argument. It then stores the formula in a new slot in the regression-model object and optionally invokes a new method :display-with-formula which gives Wald tests for the various factors, interactions and other terms in the model. The method :display-with-formula will work on `glim-proto`s as well, but a new set of constructor functions similar to `(regression-formula)` would be needed or else the `:isnew` method would need to be changed. Changing the `:isnew` method is the most elegant solution; I haven't done it because I am reluctant to override built-in code at this stage.

My Generalised Estimating Equations prototype deals with model formulae fairly transparently: it will accept a formula object, a design matrix, a sequence or a list of sequences to specify the predictors.

Example:

```
> (def a (repeat (iseq 1 2) 30))
A
>
(def b (repeat (iseq 1 3) 20))
B
>(def x (normal-rand 60))
X
> (def y (+ x a (normal-rand 60)))
Y

> (def formula1 (as-formula '((term x) (factor a) (factor b)
 (interaction (list a b)))))
FORMULA1
```

```
> (send formula1 :name-list) (send formula1 :name-list)
("((A 2) (B 2))" "((A 2) (B 3))" "(B 2)" "(B 3)" "(A 2)" "X")
> (send formula1 :block-names)
("(INTERACTION A B)" "B" "A" "X")
> (send formula1 :block-indices)
((1 0) (3 2) (4) (5))

> (regression-model (send formula1 :design-matrix) y :response-name "Why?"
:predictor-names (send formula1 :name-list))

Least Squares Estimates:

Constant                  0.867491     (0.308863)
((A 2) (B 2))            -0.481680     (0.621703)
((A 2) (B 3))            -0.666604     (0.624665)
(B 2)                     0.782557     (0.441825)
(B 3)                     0.538866     (0.444595)
(A 2)                      1.16702     (0.437606)
X                          1.30912     (0.155051)

R Squared:                0.634662
Sigma hat:                0.976272
Number of cases:                60
Degrees of freedom:             53

#<Object: 1674520, prototype = REGRESSION-MODEL-PROTO>

> (def modela (regression-formula formula1 y :response-name "Why?"))
Block              Wald Chisq     p-value
X                    71.287       0.0000
A                    7.1119       0.0077
B                    3.3431       0.1880
(INTERACTION A B)    1.2326       0.5399
MODELA
>  (send modela :display)

Least Squares Estimates:

Constant                  0.867491     (0.308863)
((A 2) (B 2))            -0.481680     (0.621703)
((A 2) (B 3))            -0.666604     (0.624665)
(B 2)                     0.782557     (0.441825)
(B 3)                     0.538866     (0.444595)
(A 2)                      1.16702     (0.437606)
```

```
X                          1.30912      (0.155051)

R Squared:               0.634662
Sigma hat:               0.976272
Number of cases:               60
Degrees of freedom:            53

NIL

> (send modela :display-with-formula)
Block                Wald Chisq     p-value
X                       71.287      0.0000
A                       7.1119      0.0077
B                       3.3431      0.1880
(INTERACTION A B)       1.2326      0.5399
NIL
> (send modela :display-with-formula nil)
Block                Wald Chisq     p-value
X                       71.287      0.0000
      Variable             Estimate       Std.Err.       p-value
     X                       1.3091     (0.155051)     0.0000
A                       7.1119      0.0077
      Variable             Estimate       Std.Err.       p-value
     (A 2)                   1.1670     (0.437606)     0.0077
B                       3.3431      0.1880
      Variable             Estimate       Std.Err.       p-value
     (B 3)                  0.53887     (0.444595)     0.2255
     (B 2)                  0.78256     (0.441825)     0.0765
(INTERACTION A B)       1.2326      0.5399
      Variable             Estimate       Std.Err.       p-value
     ((A 2) (B 3))         -0.66660     (0.624665)     0.2859
     ((A 2) (B 2))         -0.48168     (0.621703)     0.4385
NIL
```

3 Individual functions

The `(formula)` function takes a quoted list describing a model as its argument and returns `(design-matrix names block-names block-indices)` where `design-matrix` is a design matrix, `names` is a list of names for its columns, `block-names` is a list of names for multicolumn model terms such as categorical predictors and `block-indices` is a list of lists of integers describing which columns of the design matrix comprise which blocks.

The canonical form for input is like this `(formula '((factor a) (factor b) (term x) (interaction (list a b)) (interaction (list a x) :is-factor (list t nil))))`

The function extracts names from the quoted list so if you try something clever to save typing like storing `(interaction (list a x) :is-factor (list t nil))` in a variable you may get silly results. Don't do that, then.

You can change the default variable names (though not the default block names) using the relevant keyword arguments in `factor` and `interaction`. See the documentation for the functions (below) and the example to find out how. You need to specify the new name everywhere in order to make the labels come out consistent. It's probably easier to use the right variable name to begin with.

Note that the `:is-factor` keyword in `(interaction)` is optional but that the default is to assume everything is a factor. If one of the variables is continuous and you forget `:is-factor` the results will be obviously wrong (even if you don't run out of stack space). This is the best I can do with a function that accepts lists of numbers as input.

`(formula-display)` takes a vector of coefficient estimates, a covariance matrix and the output of `(formula)` and displays the Wald chisquare test for each block (factor, term or interaction), with or without the separate estimates and p-values for each variable.

`(formula-display beta cov-mat model-formula &key (block-only t) (intercept t))` `beta` is the coefficient estimates, `cov-mat` is the covariance matrix, `model-formula` is the output of the original formula command. Set `block-only` to `nil` to get tests for each separate variable. If the model does not have an intercept set `intercept` to `nil`. It is obviously better to have this installed as a method for your regression objects.

```
Example:
> (def a (repeat (iseq 1 2) 30))
A
>
(def b (repeat (iseq 1 3) 20))
B
>(def x (normal-rand 60))
X
> (def y (+ x a (normal-rand 60)))
Y
> (def result (formula '((term x) (factor a) (factor b)
 (interaction (list a b)))))
RESULT

>  (def model1 (regression-model (first result) y :predictor-names
(second result)))

Least Squares Estimates:
```

```
Constant                  0.867491     (0.308863)
((A 2) (B 2))            -0.481680     (0.621703)
((A 2) (B 3))            -0.666604     (0.624665)
(B 2)                     0.782557     (0.441825)
(B 3)                     0.538866     (0.444595)
(A 2)                      1.16702     (0.437606)
X                          1.30912     (0.155051)

R Squared:                0.634662
Sigma hat:                0.976272
Number of cases:                60
Degrees of freedom:             53


MODEL1

>  (def model1.cov (* (^ (send model1 :sigma-hat) 2) (send model1 :xtxinv)))
MODEL1.COV
> (formula-display (send model1 :coef-estimates) model1.cov result
 :block-only nil)
Block               Wald Chisq     p-value
X                      71.287      0.0000
      Variable            Estimate       Std.Err.       p-value
     X                      1.3091      (0.155051)     0.0000
A                      7.1119      0.0077
      Variable            Estimate       Std.Err.       p-value
     (A 2)                  1.1670      (0.437606)     0.0077
B                      3.3431      0.1880
      Variable            Estimate       Std.Err.       p-value
     (B 3)                 0.53887      (0.444595)     0.2255
     (B 2)                 0.78256      (0.441825)     0.0765
(INTERACTION A B)      1.2326      0.5399
      Variable            Estimate       Std.Err.       p-value
     ((A 2) (B 3))        -0.66660      (0.624665)     0.2859
     ((A 2) (B 2))        -0.48168      (0.621703)     0.4385

> (formula-display (send model1 :coef-estimates) model1.cov result)

Block               Wald Chisq     p-value
X                      71.287      0.0000
A                      7.1119      0.0077
B                      3.3431      0.1880
(INTERACTION A B)      1.2326      0.5399
```

```
NIL
```

Now we recode the predictors and relabel.

```
>  (def a (repeat (list "Male" "Female") 30))
A
>  (def b (repeat (list "Never" "Past" "Current") 20))
B
> (def result1 (formula '((term x :namebase "Age") (factor a :namebase "Sex")
(factor b :namebase "Smoking") (interaction (list a b)
:namebases (list "Sex" "Smoking")))))
RESULT1
> (formula-display (send model1 :coef-estimates) model1.cov result1)

Block               Wald Chisq     p-value
Age                    71.287      0.0000
Sex                    7.1119      0.0077
Smoking                3.3431      0.1880
(INTERACTION Sex Smoking)    1.2326     0.5399
NIL

> (formula-display (send model1 :coef-estimates) model1.cov result1
:block-only nil)
Block               Wald Chisq     p-value
Age                    71.287      0.0000
      Variable            Estimate       Std.Err.       p-value
     Age                    1.3091      (0.155051)     0.0000
Sex                    7.1119      0.0077
      Variable            Estimate       Std.Err.       p-value
     (Sex Male)             1.1670      (0.437606)     0.0077
Smoking                3.3431      0.1880
      Variable            Estimate       Std.Err.       p-value
     (Smoking Past)        0.53887      (0.444595)     0.2255
     (Smoking Never)       0.78256      (0.441825)     0.0765
(INTERACTION Sex Smoking)    1.2326     0.5399
      Variable            Estimate       Std.Err.       p-value
     ((Sex Male) (Smoking Past))  -0.66660     (0.624665) 0.2859
     ((Sex Male) (Smoking Never))  -0.48168     (0.621703) 0.4385
NIL
```

4 Primitive functions used by these methods

These functions provide a basic level of support for categorical predictor variables to be used in the Lisp-Stat regression functions. You can use just these to get more control over the process. After writing all the formula methods I found out that there were already primitive functions like this provided with `glim-proto`.

There are three main functions: `(factor)`, `(interaction)`, and `(block-test)`. The first constructs a set of treatment contrasts – indicator variables for all but the first level of the factor – and optionally a set of appropriate names. The second takes a list of variables and constructs a matrix of variables for the interaction terms. It allows both continuous and factor variables. Only the highest order interaction terms are included, so to get a matrix for the model $A + B + A : B$ you would need to call `(factor)` on each of A and B and then `(interaction)` on A and B together. This behaviour is useful for two reasons: it makes it much easier to implement a model formula system and it allows non-hierarchical models on the occasions when they are appropriate. The third function calculates and displays the Wald test for a block of variables all having zero coefficient and the individual coefficient estimates, standard errors and Wald p-values.

There are also two functions `(design)` and `(names)` that return the design matrix or list of names from a list of terms of the sort returned by `(factor)` and `(interaction)`.

`(factor x &key namebase)`.

`x` is a sequence and `namebase` is either a string or a symbol. Returns a list containing the design matrix of treatment contrasts and a list of variable names. If namebase is `nil` returns only the design matrix (**not** a list containing only the design matrix)

```
>(factor (list 2 3 4 2 3 4) :namebase "A")
(#2A((0 0) (1 0) (0 1) (0 0) (1 0) (0 1)) ("(A 3)" "(A 4)"))

>(factor (list 2 3 4 2 3 4) :namebase 'A)
(#2A((0 0) (1 0) (0 1) (0 0) (1 0) (0 1)) ("(A 3)" "(A 4)"))

> (factor (list 2 3 4 2 3 4) )
#2A((0 0) (1 0) (0 1) (0 0) (1 0) (0 1))
```

`(interaction (xs &key (is-factor (repeat t (length xs))) (namebases (repeat nil (length xs))) )`

`xs` is a list of sequences, `is-factor` is a list of `t` and `nil` values indicating whether the respective variable should be treated as a factor, `namebases` is a list of strings or symbols. Returns a design matrix (if `namebases` or `(first namebases)` is `nil`) or a list containing a design matrix and a list of names.

```
> (def a (list 1 2 1 2 1 2))
A
> (def b (list 5 6 7 5 6 7))
```

```
B
> (def c (list 0.1 0.2 0.3 0.4 0.5 0.6))
C
> (interaction (list a b) :is-factor (list t t) :namebases (list "A" 'B))
(#2A((0 0) (1 0) (0 0) (0 0) (0 0) (0 1)) ("((A 2) (B 6))" "((A 2) (B 7))"))
> (interaction (list a c) :is-factor (list t nil) :namebases (list "A" "C"))
(#2A((0.0) (0.2) (0.0) (0.4) (0.0) (0.6)) ("((A 2) C)"))
> (interaction (list a b c) :is-factor (list t t  nil)
:namebases (list "A" "B" "C"))
(#2A((0.0 0.0) (0.2 0.0) (0.0 0.0) (0.0 0.0) (0.0 0.0) (0.0 0.6))
 ("(((A 2) (B 6)) C)" "(((A 2) (B 7)) C)"))
>  (interaction (list a b c) :is-factor (list t t  nil) )
#2A((0.0 0.0) (0.2 0.0) (0.0 0.0) (0.0 0.0) (0.0 0.0) (0.0 0.6))
```

`(block-test (index beta covmat &key blockname names (block-only nil))`

`index` is a sequence of integers indicating which components of the coefficient vector are in the block, `beta` is the coefficient vector, `covmat` is the covariance matrix, `blockname` is a string giving the name of the block as a whole. Prints out a table with the Wald test of the hypothesis that all the coefficients in the block are zero and if block-only is nil separate tests for each coefficient.

```
> (def a (binomial-rand 100 4 0.3))
A
> (def b (binomial-rand 100 4 0.3))
B
> (def x (normal-rand 100 ))
X
> (def y (+ a x (normal-rand 100)))
Y
> (def factora (factor b :namebase "B"))
FACTORA
> (def factora (factor a :namebase "A"))
FACTORA
> (def factorb (factor b :namebase "B"))
FACTORB
> (def varx (list x '("X")))
VARX
> (def ab (interaction (list a b) :namebases (list "A" "B")))
AB
> (def ax (interaction (list a x) :namebases (list "A" "X")
:is-factor (list t nil)))
AX
> (def bx (interaction (list b x) :namebases (list "B" "X")
```

```
:is-factor (list t nil)))
BX
> (def abx (interaction (list a b x) :namebases (list "A" "B" "X")
:is-factor (list t t nil)))
ABX

> (def model1 (regression-model (design (list factora varx)) y
:predictor-names (names (list factora varx))))

Least Squares Estimates:

Constant                  0.299999      (0.200000)
(A 1)                     0.677777      (0.255555)
(A 2)                      1.74724      (0.277777)
(A 3)                      2.61529      (0.388888)
(A 4)                      4.33253      (0.699999)
X                          1.00357      (9.175541E-2)

R Squared:                0.700000
Sigma hat:                0.933333
Number of cases:               100
Degrees of freedom:             94

MODEL1

> (def model1-cov (* (send model1 :sigma-hat) (send model1 :xtxinv)))
MODEL1-COV

>(block-test (list 0 1 2 3) (send model1 :coef-estimates) model1-cov
:blockname "A" :names (names (list factora)) )
A                227.13      0.0000
     (A 1)                0.29999      (0.211111)          0.1666
     (A 2)                0.67777      (0.266666)          0.0096
     (A 3)                 1.7472      (0.288888)          0.0000
     (A 4)                 2.6153      (0.400000)          0.0000

> (block-test (list 0 1 2 3) (send model1 :coef-estimates) model1-cov
:blockname "A" :block-only t)
A                227.13      0.0000

>(def model2 (regression-model (design (list factora factorb  ax bx varx)) y
:predictor-names (names (list factora factorb  ax bx varx))))

Least Squares Estimates:
```

```
Constant                  -6.656751E-2  (0.266666)
(A 1)                      0.722222     (0.255555)
(A 2)                       1.75989     (0.288888)
(A 3)                       2.69490     (0.399999)
(A 4)                       4.66799     (0.699999)
(B 1)                      0.533333     (0.255555)
(B 2)                      0.155555     (0.277777)
(B 3)                      0.422222     (0.355555)
((A 1) X)                  0.233333     (0.277777)
((A 2) X)                  0.455555     (0.299999)
((A 3) X)                  0.199999     (0.388888)
((A 4) X)                   1.24579     (0.999999)
((B 1) X)                 -0.288888     (0.277777)
((B 2) X)                  0.133333     (0.299999)
((B 3) X)                 -0.611111     (0.399999)
X                          0.866666     (0.300000)

R Squared:                 0.744444
Sigma hat:                 0.900000
Number of cases:                100
Degrees of freedom:              84

MODEL2

> (def model2-cov (* (send model2 :sigma-hat ) (send model2 :xtxinv)))
MODEL2-COV
> (block-test (list 0 1 2 3) (send model2 :coef-estimates) model2-cov
:blockname "A" :block-only t)
A                 81.223      0.0000
> (block-test (list 4 5 6) (send model2 :coef-estimates) model2-cov
:blockname "B" :block-only t)
B                 45.812      0.0000
> (block-test (iseq 7 10) (send model2 :coef-estimates) model2-cov
:blockname "AX" :block-only t)
AX                3.4602      0.4888
> (block-test (iseq 11 13) (send model2 :coef-estimates) model2-cov
:blockname "BX" :block-only t)
BX                4.8578      0.1888
```