Generalized Linear Models in Lisp-Stat

Luke Tierney

January 31, 2021

1 Introduction

This note outlines a simple system for fitting generalized linear models in Lisp-Stat. Three standard models are implemented:

- Poisson regression models
- Binomial regression models
- Gamma regression models

The model prototypes inherit from the linear regression model prototype. By default, each model uses the canonical link for its error structure, but alternate link structures can be specified.

The next section outlines the basic use of the generalized linear model objects. The third section describes a few functions for handling categorical independent variables. The fourth section gives further details on the structure of the model prototypes, and describes how to define new models and link structures. The final section illustrates several ways of fitting more specialized models, using the Bradley-Terry model as an example.

2 Basic Use of the Model Objects

Three functions are available for constructing generalized linear model objects. These functions are called as

```
(poissonreg-model ⟨x⟩ ⟨y⟩ [⟨keyword arguments ...⟩])
(binomialreg-model ⟨x⟩ ⟨y⟩ ⟨n⟩ [⟨keyword arguments ...⟩])
(gammareg-model ⟨x⟩ ⟨y⟩ ⟨keyword arguments ...⟩)
```

The $\langle x \rangle$ and $\langle y \rangle$ arguments are as for the `regression-model` function. The sample size parameter $\langle n \rangle$ for binomial models can be either an integer or a sequence of integers the same length as the response vector. All optional keyword arguments accepted by the `regression-model` function are accepted by these functions as well. Four additional keywords are available: `:link`, `:offset`, `:verbose`, and `:pweights`. The keyword `:link` can be used to specify an alternate link structure. Available link structures include

```
identity-link  log-link     inverse-link  sqrt-link
logit-link     probit-link  cloglog-link
```

By default, each model uses its canonical link structure. The `:offset` keyword can be used to provide an offset value, and the keyword `:verbose` can be given the value `nil` to suppress printing of iteration information. A prior weight vector should be specified with the `:pweights` keyword rather than the `:weights` keyword.

As an example, we can examine a data set that records the number of months prior to an interview when individuals remember a stressful event (originally from Haberman, [?, p. 2]):

```
> (def months-before (iseq 1 18))
MONTHS-BEFORE
> (def event-counts '(15 11 14 17 5 11 10 4 8 10 7 9 11 3 6 1 1 4))
EVENTS-RECALLED
```

The data are multinomial, and we can fit a log-linear Poisson model to see if there is any time trend:

```
> (def m (poissonreg-model months-before event-counts))
Iteration 1: deviance = 26.3164
Iteration 2: deviance = 24.5804
Iteration 3: deviance = 24.5704
Iteration 4: deviance = 24.5704

Weighted Least Squares Estimates:

Constant                 2.80316   (0.148162)
Variable 0            -0.0837691   (0.0167996)

Scale taken as:                1
Deviance:                24.5704
Number of cases:              18
Degrees of freedom:           16
```

Residuals for the fit can be obtained using the `:residuals` message:

```
> (send m :residuals)
(-0.0439191 -0.790305 ...)
```

A residual plot can be obtained using

```
(send m :plot-residuals)
```

The `:fit-values` message returns $X\beta$, the linear predictor without any offset. The `:fit-means` message returns fitted mean response values. Thus the expression

```
(let ((p (plot-points months-before event-counts)))
  (send p :add-lines months-before (send m :fit-means)))
```

constructs a plot of raw counts and fitted means against time.

To illustrate fitting binomial models, we can use the leukemia survival data of Feigl and Zelen [**?**, Section 2.8.3] with the survival time converted to a one-year survival indicator:

```
> (def surv-1 (if-else (> times-pos 52) 1 0))
SURV-1
> surv-1
(1 1 1 1 0 1 1 0 0 1 1 0 0 0 0 0 1)
```

The dependent variable is the base 10 logarithm of the white blood cell counts divided by 10,000:

```
> transformed-wbc-pos
(-1.46968 -2.59027 -0.84397 -1.34707 -0.510826 0.0487902 0 0.530628 -0.616186
 -0.356675 -0.0618754 1.16315 1.25276 2.30259 2.30259 1.64866 2.30259)
```

A binomial model for these data can be constructed by

```
> (def lk (binomialreg-model transformed-wbc-pos surv-1 1))
Iteration 1: deviance = 18.2935
Iteration 2: deviance = 18.0789
Iteration 3: deviance = 18.0761
Iteration 4: deviance = 18.0761

Weighted Least Squares Estimates:

Constant                  0.372897   (0.590934)
Variable 0               -0.985803   (0.508426)

Scale taken as:                  1
Deviance:                  18.0761
Number of cases:                17
Degrees of freedom:             15
```

This model uses the logit link, the canonical link for the binomial distribution. As an alternative, the expression

```
(binomialreg-model transformed-wbc-pos surv-1 1 :link probit-link)
```

returns a model using a probit link.

The `:cooks-distances` message helps to highlight the last observation for possible further examination:

```
> (send lk :cooks-distances)
(0.0142046 0.00403243 0.021907 0.0157153 0.149394 0.0359723 0.0346383
 0.0450994 0.174799 0.0279114 0.0331333 0.0347883 0.033664 0.0170441
 0.0170441 0.0280411 0.757332)
```

This observation also stands out in the plot produced by

```
(send lk :plot-bayes-residuals)
```

3 Tools for Categorical Variables

Four functions are provided to help construct indicator vectors for categorical variables. As an illustration, a data set used by Bishop, Fienberg, and Holland examines the relationship between occupational classifications of fathers and sons. The classes are

Label	Description
A	Professional, High Administrative
S	Managerial, Executive, High Supervisory
I	Low Inspectional, Supervisory
N	Routine Nonmanual, Skilled Manual
U	Semi- and Unskilled Manual

The counts are given by

	Son				
Father	A	S	I	N	U
A	50	45	8	18	8
S	28	174	84	154	55
I	11	78	110	223	96
N	14	150	185	714	447
U	3	42	72	320	411

We can set up the occupation codes as

```
(def occupation '(a s i n u))
```

and construct the son's and father's code vectors for entering the data row by row as

```
(def son (repeat occupation 5))
(def father (repeat occupation (repeat 5 5)))
```

The counts can then be entered as

```
(def counts '(50  45   8  18   8
              28 174  84 154  55
              11  78 110 223  96
              14 150 185 714 447
               3  42  72 320 411))
```

To fit an additive log-linear model, we need to construct level indicators. This can be done using the function `indicators`:

```
> (indicators son)
((0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0)
 (0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0)
 (0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0)
 (0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1))
```

The result is a list of indicator variables for the second through the fifth levels of the variable `son`. By default, the first level is dropped. To obtain indicators for all five levels, we can supply the `:drop-first` keyword with value `nil`:

```
> (indicators son :drop-first nil)
((1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0)
 (0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0)
 (0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0)
 (0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0)
 (0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1))
```

To produce a readable summary of the fit, we also need some labels:

```
> (level-names son :prefix 'son)
("SON(S)" "SON(I)" "SON(N)" "SON(U)")
```

By default, this function also drops the first level. This can again be changed by supplying the `:drop-first` keyword argument as `nil`:

```
> (level-names son :prefix 'son :drop-first nil)
("SON(A)" "SON(S)" "SON(I)" "SON(N)" "SON(U)")
```

The value of the `:prefix` keyword can be any Lisp expression. For example, instead of the symbol `son` we can use the string `"Son"`:

```
> (level-names son :prefix "Son")
("Son(S)" "Son(I)" "Son(N)" "Son(U)")
```

Using indicator variables and level labels, we can now fit an additive model as

```
> (def mob-add
       (poissonreg-model
        (append (indicators son) (indicators father)) counts
        :predictor-names (append (level-names son :prefix 'son)
                                 (level-names father :prefix 'father))))
```

```
Iteration 1: deviance = 1007.97
Iteration 2: deviance = 807.484
Iteration 3: deviance = 792.389
Iteration 4: deviance = 792.19
Iteration 5: deviance = 792.19

Weighted Least Squares Estimates:

Constant                  1.36273   (0.130001)
SON(S)                    1.52892   (0.10714)
SON(I)                    1.46561   (0.107762)
SON(N)                    2.60129   (0.100667)
SON(U)                    2.26117   (0.102065)
FATHER(S)                 1.34475   (0.0988541)
FATHER(I)                 1.39016   (0.0983994)
FATHER(N)                 2.46005   (0.0917289)
FATHER(U)                 1.88307   (0.0945049)

Scale taken as:                 1
Deviance:                  792.19
Number of cases:               25
Degrees of freedom:            16
```

Examining the residuals using

```
(send mob-add :plot-residuals)
```

shows that the first cell is an outlier – the model does not fit this cell well.

To fit a saturated model to these data, we need the cross products of the indicator variables and also a corresponding set of labels. The indicators are produced with the `cross-terms` function

```
> (cross-terms (indicators son) (indicators father))
((0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
 (0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0)
 ...)
```

and the names with the `cross-names` function:

```
> (cross-names (level-names son :prefix 'son)
               (level-names father :prefix 'father))
("SON(S).FATHER(S)" "SON(S).FATHER(I)" ...)
```

The saturated model can now be fit by

```
> (let ((s (indicators son))
        (f (indicators father))
        (sn (level-names son :prefix 'son))
        (fn (level-names father :prefix 'father)))
    (def mob-sat
         (poissonreg-model (append s f (cross-terms s f)) counts
                           :predictor-names
                           (append sn fn (cross-names sn fn)))))

Iteration 1: deviance = 5.06262e-14
Iteration 2: deviance = 2.44249e-15
```

```
Weighted Least Squares Estimates:

Constant                   3.91202   (0.141421)
SON(S)                   -0.105361   (0.20548)
SON(I)                    -1.83258   (0.380789)
SON(N)                    -1.02165   (0.274874)
SON(U)                    -1.83258   (0.380789)
FATHER(S)                -0.579818   (0.236039)
FATHER(I)                 -1.51413   (0.33303)
FATHER(N)                 -1.27297   (0.302372)
FATHER(U)                 -2.81341   (0.594418)
SON(S).FATHER(S)           1.93221   (0.289281)
SON(S).FATHER(I)           2.06417   (0.382036)
SON(S).FATHER(N)           2.47694   (0.346868)
SON(S).FATHER(U)           2.74442   (0.631953)
SON(I).FATHER(S)           2.93119   (0.438884)
SON(I).FATHER(I)           4.13517   (0.494975)
SON(I).FATHER(N)           4.41388   (0.470993)
SON(I).FATHER(U)           5.01064   (0.701586)
SON(N).FATHER(S)            2.7264   (0.343167)
SON(N).FATHER(I)           4.03093   (0.41346)
SON(N).FATHER(N)           4.95348   (0.385207)
SON(N).FATHER(U)           5.69136   (0.641883)
SON(U).FATHER(S)           2.50771   (0.445978)
SON(U).FATHER(I)           3.99903   (0.496312)
SON(U).FATHER(N)           5.29608   (0.467617)
SON(U).FATHER(U)           6.75256   (0.693373)

Scale taken as:                  1
Deviance:              3.37508e-14
Number of cases:                25
Degrees of freedom:              0
```

4 Structure of the Generalized Linear Model System

4.1 Model Prototypes

The model objects are organized into several prototypes, with the general prototype `glim-proto` inheriting from `regression-model-proto`, the prototype for normal linear regression models. The inheritance tree is shown in Figure ??. This inheritance captures the reasoning by analogy to the linear case that is the basis for many ideas in the analysis of generalized linear models. The fitting strategy uses iteratively reweighted least squares by changing the weight vector in the model and repeatedly calling the linear regression `:compute` method.

Convergence of the iterations is determined by comparing the relative change in the coefficients and the change in the deviance to cutoff values. The iteration terminates if either change falls below the corresponding cutoffs. The cutoffs are set and retrieved by the `:epsilon` and `:epsilon-dev` methods. The default values are given by

```
> (send glim-proto :epsilon)
1e-06
> (send glim-proto :epsilon-dev)
0.001
```

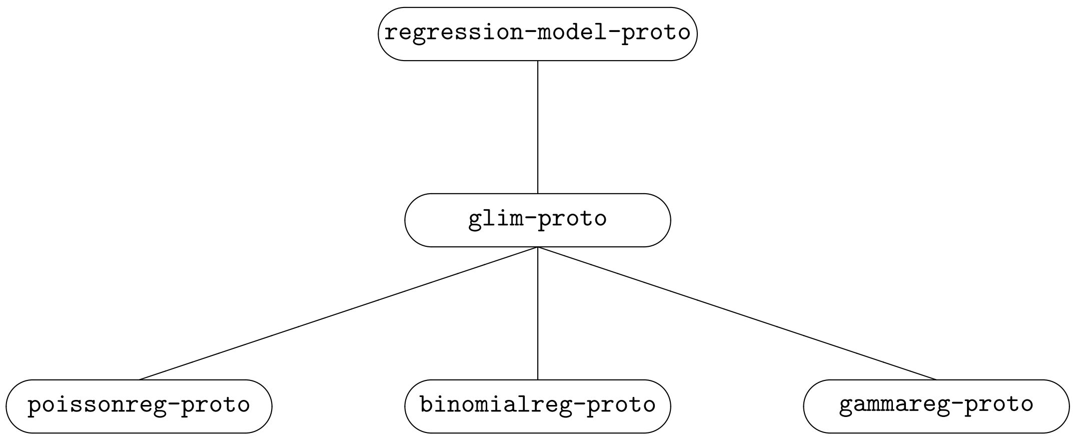

Figure 1: Hierarchy of generalized linear model prototypes.

A limit is also imposed on the number of iterations. The limit is set and retrieved by the `:count-limit` message. The default value is given by

```
> (send glim-proto :count-limit)
30
```

The analogy captured in the inheritance of the `glim-proto` prototype from the normal linear regression prototype is based primarily on the computational process, not the modeling process. As a result, several accessor methods inherited from the linear regression object refer to analogous components of the computational process, rather than analogous components of the model. Two examples are the messages `:weights` and `:y`. The weight vector in the object returned by `:weights` is the final set of weights obtained in the fit; prior weights can be set and retrieved with the `:pweights` message. The value returned by the `:y` message is the artificial dependent variable

$$z = \eta + (y - \mu)\frac{d\eta}{d\mu}$$

constructed in the iteration; the actual dependent variable can be obtained and changed with the `:yvar` message.

The message `:eta` returns the current linear predictor values, including any offset. The `:offset` message sets and retrieves the offset value. For binomial models, the `:trials` message sets and retrieves the number of trials for each observation.

The scale factor is set and retrieved with the `:scale` message. Some models permit the estimation of a scale parameter. For these models, the fitting system uses the `:fit-scale` message to obtain a new scale value. The message `:estimate-scale` determines and sets whether the scale parameter is to be estimated or not.

Deviances of individual observations, the total deviance, and the mean deviance are returned by the messages `:deviances`, `:deviance` and `:mean-deviance`, respectively. The `:deviance` and `:mean-deviance` methods adjusts for omitted observations, and the denominator for the mean deviance is adjusted for the degrees of freedom available.

Most inherited methods for residuals, standard errors, etc., should make sense at least as approximations. For example, residuals returned by the inherited `:residuals` message correspond to the Pearson residuals for generalized linear models. Other forms of residuals are returned by the messages `:chi-residuals`, `:deviance-residuals`, `:g2-residuals`, `:raw-residuals`, `:standardized-chi-residuals`, and `:standardized-deviance-residuals`.

4.2 Error Structures

The error structure of a generalized linear model affects four methods and two slots The methods are called as

```
(send ⟨m⟩ :initial-means)
(send ⟨m⟩ :fit-variances ⟨mu⟩)
(send ⟨m⟩ :fit-deviances ⟨mu⟩)
(send ⟨m⟩ :fit-scale)
```

The `:initial-means` method should return an initial estimate of the means for the iterative search. The default method simply returns the dependent variable, but for some models this may need to be adjusted to move the initial estimate away from a boundary. For example, the method for the Poisson regression model can be defined as

```
(defmeth poissonreg-proto :initial-means () (pmax (send self :yvar) 0.5))
```

which insures that initial mean estimates are at least 0.5.

The `:fit-variances :fit-deviances` methods return the values on the variance and deviance functions for a specified vector of means. For the Poisson regression model, these methods can be defined as

```
(defmeth poissonreg-proto :fit-variances (mu) mu)
```

and

```
(defmeth poissonreg-proto :fit-deviances (mu)
  (flet ((log+ (x) (log (if-else (< 0 x) x 1))))
    (let* ((y (send self :yvar))
           (raw-dev (* 2 (- (* y (log+ (/ y mu))) (- y mu))))
           (pw (send self :pweights)))
      (if pw (* pw raw-dev) raw-dev))))
```

The local function `log+` is used to avoid taking the logarithm of zero.

The final message, `:fit-scale`, is only used by the `:display` method. The default method returns the mean deviance.

The two slots related to the error structure are `estimate-scale` and `link`. If the `estimate-scale` slot is not `nil`, then a scale estimate is computed and printed by the `:dislay` method. The `link` slot holds the link object used by the model. The Poisson model does not have a scale parameter, and the canonical link is the logit link. These defaults can be set by the expressions

```
(send poissonreg-proto :estimate-scale nil)
(send poissonreg-proto :link log-link)
```

The `glim-proto` prototype itself uses normal errors and an identity link. Other error structures can be implemented by constructing a new prototype and defining appropriate methods and default slot values.

4.3 Link Structures

The link function g for a generalized linear model relates the linear predictor η to the mean response μ by

$$\eta = g(\mu).$$

Links are implemented as objects. Table **??** lists the pre-defined link functions, along with the expressions used to return link objects. With one exception, the pre-defined links require no parameters. These link objects can therefore be shared among models. The exception is the power link. Links for binomial models are defined for $n = 1$ trials and assume $0 < \mu < 1$.

Link objects inherit from the `glim-link-proto` prototype. The `log-link` object, for example, is constructed by

Table 1: Link Functions and Expression for Obtaining Link Objects

Link	Formula	Domain	Expression
Identity	μ	$(-\infty, \infty)$	`identity-link`
Logarithm	$\log \mu$	$(0, \infty)$	`log-link`
Inverse	$1/\mu$	$(0, \infty)$	`inverse-link`
Square Root	$\sqrt{\mu}$	$(0, \infty)$	`sqrt-link`
Logit	$\log \frac{\mu}{1-\mu}$	$[0, 1]$	`logit-link`
Probit	$\Phi^{-1}(\mu)$	$[0, 1]$	`probit-link`
Compl. log-log	$\log(-\log(1-\mu))$	$[0, 1]$	`cloglog-link`
Power	μ^k	$(0, \infty)$	`(send power-link-proto :new` $\langle k \rangle$`)`

```
(defproto log-link () () glim-link-proto)
```

Since this prototype can be used directly in model objects, the convention of having prototype names end in `-proto` is not used. The `glim-link-proto` prototype provides a `:print` method that should work for most link functions. The `log-link` object prints as

```
> log-link
#<Glim Link Object: LOG-LINK>
```

The `glim-proto` computing methods assume that a link object responds to three messages:

```
(send ⟨link⟩ :eta ⟨mu⟩)
(send ⟨link⟩ :means ⟨eta⟩)
(send ⟨link⟩ :derivs ⟨mu⟩)
```

The `:eta` method returns a sequence of linear predictor values for a particular mean sequence. The `:means` method is the inverse of `:eta`: it returns mean values for specified values of the linear predictor. The `:derivs` method returns the values of

$$\frac{d\eta}{d\mu}$$

at the specified mean values. As an example, for the `log-link` object these three methods are defined as

```
(defmeth log-link :eta (mu) (log mu))
(defmeth log-link :means (eta) (exp eta))
(defmeth log-link :derivs (mu) (/ mu))
```

Alternative link structures can be constructed by setting up a new prototype and defining appropriate `:eta`, `:means`, and `:derivs` methods. Parametric link families can be implemented by providing one or more slots for holding the parameters. The power link is an example of a parametric link family. The power link prototype is defined as

```
(defproto power-link-proto '(power) () glim-link-proto)
```

The slot `power` holds the power exponent. An accessor method is defined by

```
(defmeth power-link-proto :power () (slot-value 'power))
```

and the `:isnew` initialization method is defined to require a power argument:

```
(defmeth power-link-proto :isnew (power) (setf (slot-value 'power) power))
```

Thus a power link for a particular exponent, say the exponent 2, can be constructed using the expression

```
(send power-link-proto :new 2)
```

To complete the power link prototype, we need to define the three required methods. They are defined as

```
(defmeth power-link-proto :eta (mu) (^ mu (send self :power)))

(defmeth power-link-proto :means (eta) (^ eta (/ (slot-value 'power))))
```

and

```
(defmeth power-link-proto :derivs (mu)
  (let ((p (slot-value 'power)))
    (* p (^ mu (- p 1)))))
```

The definition of the `:means` method could be improved to allow negative arguments when the power is an odd integer. Finally, the `:print` method is redefined to reflect the value of the exponent:

```
(defmeth power-link-proto :print (&optional (stream t))
  (format stream ``#<Glim Link Object: Power Link (~s)>'' (send self :power)))
```

Thus a square link prints as

```
> (send power-link-proto :new 2)
#<Glim Link Object: Power Link (2)>
```

5 Fitting a Bradley-Terry Model

Many models used in categorical data analysis can be viewed as special cases of generalized linear models. One example is the Bradley-Terry model for paired comparisons. The Bradley-Terry model deals with a situation in which n individuals or items are compared to one another in paired contests. The model assumes there are positive quantities $\pi_1, \ldots, \pi_n$, which can be assumed to sum to one, such that

$$P\{i \text{ beats } j\} = \frac{\pi_i}{\pi_i + \pi_j}.$$

If the competitions are assumed to be mutually independent, then the probability $p_{ij} = P\{i \text{ beats } j\}$ satisfies the logit model

$$\log \frac{p_{ij}}{1 - p_{ij}} = \phi_i - \phi_j$$

with $\phi_i = \log \pi_i$. This model can be fit to a particular set of data by setting up an appropriate design matrix and response vector for a binomial regression model. For a single data set this can be done from scratch. Alternatively, it is possible to construct functions or prototypes that allow the data to be specified in a more convenient form. Furthermore, there are certain specific questions that can be asked for a Bradley-Terry model, such as what is the estimated value of $P\{i \text{ beats } j\}$? In the object-oriented framework, it is very natural to attach methods for answering such questions to individual models or to a model prototype.

To illustrate these ideas, we can fit a Bradley-Terry model to the results for the eastern division of the American league for the 1987 baseball season [?]. Table **??** gives the results of the games within this division.

The simplest way to enter this data is as a list, working through the table one row at a time:

Table 2: Results of 1987 Season for American League Baseball Teams

Winning	Losing Team						
Team	Milwaukee	Detroit	Toronto	New York	Boston	Cleveland	Baltimore
Milwaukee	-	7	9	7	7	9	11
Detroit	6	-	7	5	11	9	9
Toronto	4	6	-	7	7	8	12
New York	6	8	6	-	6	7	10
Boston	6	2	6	7	-	7	12
Cleveland	4	4	5	6	6	-	6
Baltimore	2	4	1	3	1	7	-

```
(def wins-losses '( -  7  9  7  7  9 11
                    6  -  7  5 11  9  9
                    4  6  -  7  7  8 12
                    6  8  6  -  6  7 10
                    6  2  6  7  -  7 12
                    4  4  5  6  6  -  6
                    2  4  1  3  1  7  -))
```

The choice of the symbol `-` for the diagonal entries is arbitrary; any other Lisp item could be used. The team names will also be useful as labels:

```
(def teams '("Milwaukee" "Detroit" "Toronto" "New York"
             "Boston" "Cleveland" "Baltimore"))
```

To set up a model, we need to extract the wins and losses from the `wins-losses` list. The expression

```
(let ((i (iseq 1 6)))
  (def low-i (apply #'append (+ (* 7 i) (mapcar #'iseq i)))))
```

constructs a list of the indices of the elements in the lower triangle:

```
> low-i
(7 14 15 21 22 23 28 29 30 31 35 36 37 38 39 42 43 44 45 46 47)
```

The wins can now be extracted from the `wins-losses` list using

```
> (select wins-losses low-i)
(6 4 6 6 8 6 6 2 6 7 4 4 5 6 6 2 4 1 3 1 7)
```

Since we need to extract the lower triangle from a number of lists, we can define a function to do this as

```
(defun lower (x) (select x low-i))
```

Using this function, we can calculate the wins and save them in a variable `wins`:

```
(def wins (lower wins-losses))
```

To extract the losses, we need to form the list of the entries for the transpose of our table. The function `split-list` can be used to return a list of lists of the contents of the rows of the original table. The `transpose` function transposes this list of lists, and the `append` function can be applied to the result to combine the lists of lists for the transpose into a single list:

```
(def losses-wins (apply #'append (transpose (split-list wins-losses 7))))
```

The losses are then obtained by

```
(def losses (lower losses-wins))
```

Either `wins` or `losses` can be used as the response for a binomial model, with the trials given by

```
(+ wins losses)
```

When fitting the Bradley-Terry model as a binomial regression model with a logit link, the model has no intercept and the columns of the design matrix are the differences of the row and column indicators for the table of results. Since the rows of this matrix sum to zero if all row and column levels are used, we can delete one of the levels, say the first one. Lists of row and column indicators are set up by the expressions

```
(def rows (mapcar #'lower (indicators (repeat (iseq 7) (repeat 7 7)))))
(def cols (mapcar #'lower (indicators (repeat (iseq 7) 7))))
```

The function `indicators` drops the first level in constructing its indicators. The function `mapcar` applies `lower` to each element of the indicators list and returns a list of the results. Using these two variables, the expression

```
(- rows cols)
```

constructs a list of the columns of the design matrix.

We can now construct a model object for this data set:

```
> (def wl (binomialreg-model (- rows cols)
                             wins
                             (+ wins losses)
                             :intercept nil
                             :predictor-names (rest teams)))
Iteration 1: deviance = 16.1873
Iteration 2: deviance = 15.7371

Weighted Least Squares Estimates:

Detroit                -0.144948   (0.311056)
Toronto                -0.286871   (0.310207)
New York               -0.333738   (0.310126)
Boston                 -0.473658   (0.310452)
Cleveland              -0.897502   (0.316504)
Baltimore               -1.58134   (0.342819)

Scale taken as:                1
Deviance:                15.7365
Number of cases:              21
Degrees of freedom:           15
```

To fit to a Bradley-Terry model to other data sets, we can repeat this process. As an alternative, we can incorporate the steps used here into a function:

```
(defun bradley-terry-model (counts &key labels)
  (let* ((n (round (sqrt (length counts))))
         (i (iseq 1 (- n 1)))
         (low-i (apply #'append (+ (* n i) (mapcar #'iseq i))))
         (p-names (if labels
                      (rest labels)
                      (level-names (iseq n) :prefix "Choice"))))
    (labels ((tr (x)
               (apply #'append (transpose (split-list (coerce x 'list) n))))
             (lower (x) (select x low-i))
             (low-indicators (x) (mapcar #'lower (indicators x))))
      (let ((wins (lower counts))
            (losses (lower (tr counts)))
            (rows (low-indicators (repeat (iseq n) (repeat n n))))
            (cols (low-indicators (repeat (iseq n) n))))
        (binomialreg-model (- rows cols)
                           wins
                           (+ wins losses)
                           :intercept nil
                           :predictor-names p-names)))))
```

This function defines the function `lower` as a local function. The local function `tr` calculates the list of the elements in the transposed table, and the function `low-indicators` produces indicators for the lower triangular portion of a categorical variable. The `bradley-terry-model` function allows the labels for the contestants to be specified as a keyword argument. If this argument is omitted, reasonable default labels are constructed. Using this function, we can construct our model object as

```
(def wl (bradley-terry-model wins-losses :labels teams))
```

The definition of this function could be improved to allow some of the keyword arguments accepted by `binomialreg-model`.

Using the fit model object, we can estimate the probability of Boston ($i = 4$) defeating New York ($j = 3$):

```
> (let* ((phi (cons 0 (send wl :coef-estimates)))
         (exp-logit (exp (- (select phi 3) (select phi 4)))))
    (/ exp-logit (+ 1 exp-logit)))
0.534923
```

To be able to easily calculate such an estimate for any pairing, we can give our model object a method for the `:success-prob` message that takes two indices as arguments:

```
(defmeth wl :success-prob (i j)
  (let* ((phi (cons 0 (send self :coef-estimates)))
         (exp-logit (exp (- (select phi i) (select phi j)))))
    (/ exp-logit (+ 1 exp-logit))))
```

Then

```
> (send wl :success-prob 4 3)
0.465077
```

If we want this method to be available for other data sets, we can construct a Bradley-Terry model prototype by

```
(defproto bradley-terry-proto () () binomialreg-proto)
```

and add the `:success-prob` method to this prototype:

```
(defmeth bradley-terry-proto :success-prob (i j)
  (let* ((phi (cons 0 (send self :coef-estimates)))
         (exp-logit (exp (- (select phi i) (select phi j)))))
    (/ exp-logit (+ 1 exp-logit))))
```

If we modify the `bradley-terry-model` function to use this prototype by defining the function as

```
(defun bradley-terry-model (counts &key labels)
  (let* ((n (round (sqrt (length counts))))
         (i (iseq 1 (- n 1)))
         (low-i (apply #'append (+ (* n i) (mapcar #'iseq i))))
         (p-names (if labels
                      (rest labels)
                      (level-names (iseq n) :prefix "Choice"))))
    (labels ((tr (x)
               (apply #'append (transpose (split-list (coerce x 'list) n))))
             (lower (x) (select x low-i))
             (low-indicators (x) (mapcar #'lower (indicators x))))
      (let ((wins (lower counts))
            (losses (lower (tr counts)))
            (rows (low-indicators (repeat (iseq n) (repeat n n))))
            (cols (low-indicators (repeat (iseq n) n))))
        (send bradley-terry-proto :new
              :x (- rows cols)
              :y wins
              :trials (+ wins losses)
              :intercept nil
              :predictor-names p-names)))))
```

then the `:success-prob` metod is available immediately for a model constructed using this function:

```
> (def wl (bradley-terry-model wins-losses :labels teams))
Iteration 1: deviance = 16.1873
Iteration 2: deviance = 15.7371
...
> (send wl :success-prob 4 3)
0.465077
```

The `:success-prob` method can be improved in a number of ways. As one example, we might want to be able to obtain standard errors in addition to estimates. A convenient way to provide for this possibility is to have our method take an optional argument. If this argument is `nil`, the default, then the method just returns the estimate. If the argument is not `nil`, then the method returns a list of the estimate and its standard error.

To calculate the standard error, it is easier to start with the logit of the probability, since the logit is a linear function of the model coefficients. The method defined as

```
(defmeth bradley-terry-proto :success-logit (i j &optional stdev)
  (let ((coefs (send self :coef-estimates)))
    (flet ((lincomb (i j)
             (let ((v (repeat 0 (length coefs))))
               (if (/= 0 i) (setf (select v (- i 1)) 1))
               (if (/= 0 j) (setf (select v (- j 1)) -1))
               v)))
      (let* ((v (lincomb i j))
             (logit (inner-product v coefs))
             (var (if stdev (matmult v (send self :xtxinv) v))))
        (if stdev (list logit (sqrt var)) logit)))))
```

returns the estimate or a list of the estimate and approximate standard error of the logit:

```
> (send wl :success-logit 4 3)
-0.13992
> (send wl :success-logit 4 3 t)
(-0.13992 0.305583)
```

The logit is calculated as a linear combination of the coefficients; a list representing the linear combination vector is constructed by the local function `lincomb`.

Standard errors for success probabilities can now be computed form the results of `:success-logit` using the delta method:

```
(defmeth bradley-terry-proto :success-prob (i j &optional stdev)
  (let* ((success-logit (send self :success-logit i j stdev))
         (exp-logit (exp (if stdev (first success-logit) success-logit)))
         (p (/ exp-logit (+ 1 exp-logit)))
         (s (if stdev (* p (- 1 p) (second success-logit)))))
    (if stdev (list p s) p)))
```

For our example, the results are

```
> (send wl :success-prob 4 3)
0.465077
> (send wl :success-prob 4 3 t)
(0.465077 0.0760231)
```

These methods can be improved further by allowing them to accept sequences of indices instead of only individual indices.

Acknowledgements

I would like to thank Sandy Weisberg for many helpful comments and suggestions, and for providing the code for the glim residuals methods.

References

[1] Agresti, A. (1990), *Categorical Data Analysis*, New York, NY: Wiley.

[2] Lindsey, J. K. (1989), *The Analysis of Categorical Data Using GLIM*, Springer Lecture Notes in Statistics No. 56, New York, NY: Springer.

[3] McCullagh, P. and Nelder, J. A. (1989), *Generalized Linear Models*, second edition, London: Chapman and Hall.

[4] Tierney, L. (1990), *Lisp-Stat: An Object-Oriented Environment for Statistical Computing and Dynamic Graphics*, New York, NY: Wiley.