Simulations using Arc

Sanford Weisberg

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

Supported by National Science Foundation Grant DUE 96-52887

October 7, 1999

Abstract

Arc is a computer program for the analysis of regression problems. It is described in Cook and Weisberg (1999). Although primarily a menu driven program, it can be used by typing commands at the command line, and it can therefore be used for simulations. Several applications of this are illustrated with a sequence of functions in the file `simulate.lsp` that can be downloaded from Internet.

Arc is a menu driven computer program of the analysis of regression data, as described in Cook and Weisberg (1999). Since *Arc* is written in the *Xlisp-Stat* language, all the functionality of *Xlisp-Stat* is available when you use *Arc*. In particular, it is possible to use *Arc* with commands typed at a command line. In this way, it is possible to use *Arc* as a research tool, for example by computing simulations.

1 Getting Started

WARNING! This material requires more familiarity with computing than most work with *Arc*. When you start *Arc*, all of *Xlisp-Stat* is available as well. In this report, we show how *Xlisp-Stat* code can be used to call *Arc* for simulations. *Xlisp-Stat* is an implementation of the *lisp* language. For those unfamiliar with *lisp*, there are several ways of getting started. Tierney (1990) provides a very readable introduction. Several online references can be obtained from http://www.visualstats.org/. The book *Common Lisp* by Guy L. Steele is a comprehensive source for the *lisp* language, and is available on-line at the location given in the references. The file `simulate.lsp` is available from www.stat.umn.edu/arc, and includes the *Xlisp-Stat* code described in this report. To use it, download the file and put it in your Extras directory in the same directory as the *Xlisp-Stat* program.

2 Simulating a t-test

As a simple example, consider simulating the distribution of the t-test for the hypothesis that $\eta_1 = 0$ in the simple linear regression model

$$\mathrm{E}(y|x) = \eta_0 + \eta_1 x \text{ with } \mathrm{var}(y|x) = \sigma^2$$

One possible simulation might go as follows. First, select a sample size n, and values for η_0 and η_1. Generate a fixed vector x of length n. Then, for each of B simulations,

1. Generate ε of length n from a distribution of interest.

2. Compute $y = \eta_0 + \eta_1 x + \varepsilon$.

3. Fit the simple linear regression of y on x.

4. Save $t = \hat{\eta}_1 / \mathrm{se}(\hat{\eta}_1)$.

The B values of t can then be summarized as needed.

We will take $n = 10$, x standard normal, and $\eta_0 = \eta_1 = 0$, with uniform errors, so we will be interested in the level of the test, not in power, when the errors are sampled from the uniform $(-.5, +.5)$ distribution. The following *Xlisp-Stat* code sets up the problem to match this specification.

```
(def n 10)  Set the sample size
(def eta0 0)  Set the intercept
(def eta1 0)  Set the slope
(def x (normal-rand n))  The predictor is normal random numbers
(def Ey (+ eta0 (* eta1 x)))  The mean values of y given x
(def y (+ Ey (- (uniform-rand 10) .5)))  Add uniform error to Ey
 Finally, create a dataset with these values
(make-dataset :data (list x Ey y)
              :data-names (list "X" "Ey" "Y") :name "sim")
```

The dataset named `sim` with variables X, Y and *Ey* will appear on the menu bar. From the Graph&Fit menu, fit the simple linear regression of Y on X. The regression will probably be named `L1`.

Now for the simulation. The following function does one simulation:

```
(defun sim0 ()
 (send L1 :yvar (+ Ey (- (uniform-rand n) .5)))
 (second (/ (send l1 :coef-estimates)
            (send l1 :coef-standard-errors))))
```

This function will replace Y with the new random values, and then compute and return the t-test (the method `:coef-estimates` checks to see that the data have changed and if so all estimates are recomputed using the modified data).

The function `sim0` can be repeated $B = 1000$ times by typing the obscure *lisp* code

```
(def tvals (mapcar #'(lambda (j) (sim0)) (iseq 1000)))
```

The macro `mapcar` executes the function `#'(lambda (j) (sim))` for every element in the list following the name of the function. The result is a list of $B = 1000$ values of t. If you type

```
(make-dataset :data (list tvals))
```

a new dataset will be created consisting of the simulated values. You can then produce graphs and summary statistics for the sample of $B = 1000$ t-values.

3 First Elaboration

The function `sim1` combines all the above actions into one function, as shown in Table 1. This generalizes the original simulation in several ways:

1. Initial conditions, like sample sizes, values of parameters and distributions, are passed as arguments. The parameters `x-dist` and `error-dist` expect the names of an *Xlisp-Stat* function like `#'normal-rand`, which when executed will return a list of random numbers, in this case normal random numbers. The default argument for `error-dist` is based on uniform random numbers, and since these must be on the range $(-.5, .5)$, the $.5$ must be subtracted from the uniform generator. This is accomplished using an anonymous function, so the default is `#'(lambda (n) (- (uniform-rand n) 0.5))`.

2. The `make-dataset` function creates a data set with name `sim`, and adds it to the menu bar. We show later than this step can be skipped.

3. The `:make-glm` method is the internal *Arc* method that creates the regression model. This method is described more completely in the next section.

4. Each simulation returns the ts for both the intercept and the slope. The function `transpose` takes the B lists, each of length two, and returns two lists, each of length B. If you type

   ```
   (make-dataset :data (simulate :n 10 :B 100)
                 :data-names '("t-int" "t-slope")
                 :name "Sim-output")
   ```

   the output from the simulation will become an *Arc* data set that you can summarize with graphs or save to a file.

The function `sim1` can be modified to study some other aspect of the regression model. For example, if one were interested in both the t-tests and R^2, the last five lines of the function could be replaced by

```
(transpose (mapcar #'(lambda (j)
     (send L1 :yvar (+ Ey (funcall error-dist n)))
     (cons (send L1 :r-squared)
           (/ (send L1 :coef-estimates)
              (send L1 :coef-standard-errors))))
  (iseq B)))
```

This will probaby look very obscure unless you are familiar with *lisp*. The function in the middle (starting with `#'(lambda`) first replaces the response with a new random response. It then gets the new value of R^2, the new t-values, and then uses the `cons` function to add R^2 to the front of the list of the t-values. Thus, each time the function is called, a list of three values, (R^2, t_0, t_1), will be returned. More generally, one could write a regression-model-proto method that returns the statistics of interest. For example, the method

Table 1: The Function `sim1` to Simulate a t-Test.

```
(defun sim1 (&key
            (n 10)    ; sample size
            (B 1000)  ; number of simulations
            (eta (list 0 0)) ; values of the regression parameters
            (x-dist #'normal-rand) ; distribution to generate x
            (error-dist #'(lambda (n) (- (uniform-rand n) .5)))
"Function args:  (&key (n 100) (m 1000) (eta (list 0 0))
            (x-dist #'normal-rand)
            (error-dist #'(lambda (n) (- (uniform-rand n) .5))))
Does B simulations of n observations on the simple regression of
y on x with x sampled from the x-dist distribution.  The true
values of the parameters is given by eta, and the error
distribution is given by error-dist."
; Set up initial conditions
  (let* ((eta0 (first eta))
         (eta1 (second eta))
         (x (funcall x-dist n))
         (Ey (+ eta0 (* eta1 x)))
         (e (funcall error-dist n))
         (d (make-dataset :data (list x Ey (+ Ey e))
                          :data-names (list "x" "Ey" "y")
                          :name "sim")))
; define a regression model, named in this case L1:
      (send d :make-glm :predictors (list "X")
                        :response "Y"
                        :type :normal
                        :name "L1"
                        :weights nil
                        :offset nil
                        :trials nil
                        :mean-function "Identity"
                        :intercept t
                        :display nil)
; do the simulation
     (transpose (mapcar #'(lambda (j)
          (send L1 :yvar (+ Ey (funcall error-dist n)))
          (/ (send L1 :coef-estimates)
             (send L1 :coef-standard-errors)))
       (iseq B)))
     ))
```

```
(defmeth regression-model-proto :simulated-values ()
  (let ((t-vals (/ (send self :coef-estimates)
                   (send self :coef-standard-errors)))
        (r2 (send self :r-squared))
        (s (send self :sigma-hat)))
   (combine s r2 t-vals)))
```

will return $\hat{\sigma}$, R^2 and the t-values. The last five lines of the function could then be replaced by

```
     (transpose (mapcar #'(lambda (j)
          (send L1 :yvar (+ Ey (funcall error-dist n)))
          (send L1 :simulated-values))
       (iseq B)))
```

4 Second Elaboration

The function `sim1` is designed to work in analogy to the way the *Arc* works by first defining a data set, and then deriving a model from the data set. For the purposes of a simulation, this may be an extra, and unnecessary, step. We may wish to go directly from data to a model without first defining a data set. How this is done is shown in the function `sim2` in Table 2. The changes in this function are (1) no data set is created and (2) the regression model is created with the function `make-linear` instead of creating the model through the dataset. Otherwise, this function works the same as `sim1`. One advantage of this approach is that no graphics are used (no menus or dialogs).

The `make-linear` function is just one of several functions available for creating models of various types without first creating a dataset. All the functions are listed in Table 3. All require specification of the terms (argument `:x`) and the response (argument `:y`). Binomial models also require specification of the number of trials (argument `:trials`). Optional arguments can be used to set weights or an offset. You can also set the *link function*. The link function is defined in Section 21.4.1 as the inverse of the kernel mean function; specifying generalized linear models using the link function is more standard than using kernel mean functions, but the two are equivalent. To find out the appropriate link functions you can use for binomial models, for example, type

```
> (link-functions :binomial)
(#<Glim Link Object: LOGIT-LINK>
 #<Glim Link Object: PROBIT-LINK>
 #<Glim Link Object: CLOGLOG-LINK>)
```

For example, to fit a binomial regression model with predictors given in the variable `x`, response in `y`, trials in `n`, but using the probit link (or inverse probit kernel mean function), type

```
(make-binomial :x x :y y :trials n :link probit-link)
```

Table 4 lists functions for making inverse regression models (based on sliced inverse regression or SIR, sliced average variance estimation or SAVE or principal Hessian directions, or pHd). These functions are similar to the functions for making generalized linear models.

Table 2: The Function `sim2` that By-passes the Need for Creating a Dataset.

```
(defun sim2 (&key
            (n 100)   ; sample size
            (B 1000)  ; number of simulations
            p         ; number of predictors
            eta       ; values of the regression coef, including intercept
            (x-dist #'(lambda () (normal-rand (repeat n p))))
                      ; distribution to generate multivariate x
            (error-dist #'(lambda (n) (- (uniform-rand n) .5))))
"Function args:  (&key (n 100) (m 1000) eta p
            (x-dist #'normal-rand)
            (error-dist #'(lambda (n) (- (uniform-rand n) .5)))
            (x-dist #'(lambda () (normal-rand (repeat n p)))))
Does B simulations of n observations on the simple regression of
y on x with x sampled from the x-dist distribution.  The true
values of the parameters is given by eta, and the error
distribution is given by error-dist."
; Set up initial conditions
  (let* ((x (funcall x-dist n))
         (yhat (+ (first eta) (sum (* (rest eta) x))))
         (e (funcall error-dist n))
         (r (make-linear :x x :y (+ yhat e))))
; do the simulation
     (transpose (mapcar #'(lambda (j)
          (send r :yvar (+ Yhat (funcall error-dist n)))
          (send r :simulated-values))
       (iseq B)))
     ))
```

Table 3: Methods for Making Models of Various Types Based on Generalized Linear Models.

```
make-linear
make-binomial
make-poisson
make-gamma
```

These functions create model objects without first creating a dataset. This will be useful primarily for simulation studies. None of these functions use any graphical methods, like menus, graphs and dialogs. In each of the functions below, the following arguments may appear:

REQUIRED arguments:

- `:x` *gives the list of lists of predictors*
- `:y` *gives the response*
- `:trials` *(binomial only) gives the number of trials*

OPTIONAL arguments:

- `:weights` *gives the weights*
- `:intercept t or nil` *for an intercept*
- `:offset` *a list of numbers if there is an offset*
- `:link` *gives the link if not the canonical link*

You can get a list of the link function names using the function

```
     (link-functions :normal)
     (link-functions :poisson)
     (link-functions :gamma)
     (link-functions :binomial)
```

Of course you can write your own link functions as well

Table 4: Methods for Making Inverse Regression Models.

```
make-sir
make-phd
make-save
```

The REQUIRED arguments are:

- `:x` *list of lists of predictors*
- `:y` *list of response*
- `:nslices` *SIR/SAVE only*

The OPTIONAL arguments are:

- `:weights`
- `:method-response either :yvar or :residuals`
- *anything else passed to the :isnew*

A similar function for nonlinear regression models is called `make-nonlinear`. As with the other models, it has required arguments `:x` for the predictors, `:y` for the response, and optional argument `:weights` for weights. Two additional arguments are required. `:mean-function` is a string that specifies the mean function for the response given the predictors. For example, `"th0+th1*(1-exp(th2*x1))"` specfies the mean function $\theta_0 + \theta_1(1 - \exp(\theta_2 x))$. The rules for this string are discussed in the chapter on nonlinear regression available from the *Arc* website. Finally, the argument `:starting-values` is a list of starting values for the parameters. Any other argument that can be passed to a nonlinear model can also be added to this function. The function `sim2` should be taken as a prototype of what can be done in simulations. In most instances, the user will need to write a substitute simulation function based on `sim2`.

5 What to Save

The function `sim2` will at each replication save whatever is returned by the method regression method `:simulated-values`. There is no default for this function, so the user must write this function for each particular application of simulation.

To help in writing this method, it is possible to get a list of the the existing methods for each of the models. For example, the command

```
(display-help inverse-regression-model-proto)
```

will give the name of all methods that for inverse regression models. (Warning: the output is long!). You can get similar lists for regression-model-proto (methods available to all models); glim-proto (for generalized linear models); normalreg-proto, binomialreg-proto, gammareg-proto and poissonreg-proto.

For example suppose you wanted to simulate the significance level of the test for 1D structure using pHd. For a particular data set, the test is computed by the method `phd-test`; the argument `:print nil` suppresses printing. This method returns a list of lists that corresponds to the usual printed output for the pHd test (you can discover this by making a pHd model in the usual way, and then sending the model the `:phd-test` method). The test statistic is the fourth value in the first list. The function `sim3` in Table 5 will simulate this test.

6 More Elaboration

The user is constrained only by imagination and knowledge of *lisp*. One could generate from the joint distribution of (x, y) rather than from the conditional distriubtion of $y|x$, or permute residuals to get a bootstrap or a permutation test. For your own simulations, you can modify these functions to suit your needs.

7 Another Example

Arc uses simulation in envelopes around probability plots of residuals. The general idea was outlined by A. C. Atkinson (1985, Section 4.2), as follows:

Table 5: The Function `sim3` to Simulate a Test Based on pHd.

```
(defun sim3 (&key
            (n 100)   ; sample size
            (B 1000)  ; number of simulations
            p         ; number of predictors
            eta       ; values of the regression coef
            (x-dist #’(lambda () (normal-rand (repeat n p))))
                      ; distribution to generate multivariate x
            (error-dist #’(lambda (n) (- (uniform-rand n) .5))))
"Function args:  (&key (n 100) (m 1000) eta p
            (x-dist #’normal-rand)
            (error-dist #’(lambda (n) (- (uniform-rand n) .5)))
            (x-dist #’(lambda () (normal-rand (repeat n p)))))
Does B simulations of n observations on the simple regression of
y on x with x sampled from the x-dist distribution.  The true
values of the parameters is given by eta, and the error
distribution is given by error-dist."
; Set up initial conditions
  (let* ((x (funcall x-dist n))
         (yhat (+ (first eta) (sum (* (rest eta) x))))
         (e (funcall error-dist n))
         (r (make-phd :x x :y (+ yhat e))))
; do the simulation
     (transpose (mapcar #’(lambda (j)
          (send r :yvar (+ Yhat (funcall error-dist n)))
          (send r :simulated-values))
       (iseq B)))
     ))
(defmeth inverse-regression-model-proto :simulated-values ()
 (fourth (first (send self :phd-test))))
```

1. Draw a probability plot of ordered residuals versus quantiles from a standard distribution like the normal or the half-normal.

2. To calibrate the plot, assume that the estimated values of parameters in the model are true values, and then generate a random response vector according to the model. For a normal linear regression model, the ith random response would be equal to the ith fitted value plus a normal random deviate times the estimated value of σ. For a logistic regression model, the ith random response is a random draw from a binomial distribution with number of trials and probability of success equal to the number of trials and estimated probability of success for the ith case. If `m` is the name of the model, the call

   ```
   (send m :random-response)
   ```

   will return random responses for all cases except those with missing data.

3. Fit the same model as before but with the response given by the random response. Save the residuals (or any other statistic derived from the model).

4. Repeat (2) and (3) 19 times. Add to the original probability plot at each plotting position the minimum and maximum of the 19 simulated probability plots to give an envelope for the original curve.

The method `:random-response` can be useful in many other simulations. As an exercise the reader could write a function that would simulate the null distribution of the deviance (or Pearson's X^2) for a particular log-linear or logistic regression model.

Happy computing!

8 References

Atkinson, A. C. (1985). *Plots, Transformation and Regression*. Oxford: Oxford University Press.

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

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.