JACKKNIFING IN XLISP-STAT

JAN DE LEEUW

ABSTRACT. It is comparatively easy to write functions in Xlisp-Stat that perform the usual Jackknife computations in any specific problem. It suffices to put a loop around the function we are studying, and to collect the results of the loop in an appropriate way. In this note we use the unique properties of Lisp to automate this process.

CONTENTS

1. INTRODUCTION

In Lisp it is common practice to write functions that return functions. See Tierney [7], Section 3.6.3, and Graham [3], Chapters 5 and 15. Applications of these techniques are, for instance, to produce memoized versions of functions, or versions of function that keep a record of the number of times they are called (which is useful in profiling).

In this paper we use Lisp, in particular Xlisp-Stat, to write functions implementing the Jackknife. This makes it possible to illustrate some basic Lisp techniques, and it produces a fairly routine way to associate a standard error and/or confidence interval with any statistic computed in Xlisp-Stat.

Written while visiting the National Institute of Statistical Sciences, Research Triangle Park, NC.

2. The Jackknife

The Jackknife [6, 8] is a popular technique to compute bias corrections, standard errors, and confidence intervals for a wide class of statistics. Currently, it seems to be pushed to the background by Efron's Bootstrap [2] and related resampling techniques, but we feel that the simplicity and the straightforwardness of the Jackknife continue to make it valuable.

Here is a brief outline of the Jackknife. We no attempt to be either comprehensive or rigorous. The expansions and calculations we use can be made rigorous by using results such as those of Hurt [5]. We underline random variables [4], and we use $\triangleq$ for definitions.

2.1. Delta Method.

Suppose Φ is a possible vector-valued function of a sequence of $m-$vectors of multinomial proportions $\underline{p}_n$. Define $\pi \triangleq \mathbf{E}\,(\underline{p}_n)$, and $V(\pi) \triangleq n\mathbf{V}\,(\underline{p}_n)$. We can write

$$\Phi(\underline{p}_n) = \Phi(\pi) + n^{-1/2}G(\pi)\underline{z}_n + o_p(n^{-1/2}), \tag{2.1}$$

where

$$G(\pi) \triangleq \left.\frac{\partial\Phi}{\partial p}\right|_{p=\pi}, \tag{2.2}$$

and

$$\underline{z}_n \triangleq n^{1/2}(\underline{p}_n - \pi). \tag{2.3}$$

Under suitable regularity conditions we have the following expression for the dispersion of $\Phi(\underline{p}_n)$.

$$n\mathbf{E}(\Phi(\underline{p}_n) - \Phi(\pi))(\Phi(\underline{p}_n) - \Phi(\pi))' = G(\pi)V(\pi)G'(\pi) + o(1). \tag{2.4}$$

For bias correction we need an extra term in the expansion. For coordinate s

$$\phi_s(\underline{p}_n) = \phi_s(\pi) + n^{-1/2}g_s'(\pi)\underline{z}_n + \frac{1}{2}n^{-1}\underline{z}_n'H_s(\pi)z_n + o_p(n^{-1}), \tag{2.5}$$

which implies, assuming sufficient smoothness, that

$$n\mathbf{E}\,(\phi_s(\underline{p}_n) - \phi_s(\pi)) = \frac{1}{2}\ \mathrm{tr}\ H_s(\pi)V(\pi) + o(1). \tag{2.6}$$

2.2. Approximating the Derivatives.

The Jackknife can be thought of as a systematic and convenient way to approximate the first and second derivatives numerically. From the programming point of view, this has the major advantage that we do not have to write code to compute the derivatives analytically, which will necessarily be specific to the problem at hand.

The $\underline{p}_n$ are averages of n random variables, which take the unit vectors e_j in $\mathbb{R}^m$ as their values. Define

$$\underline{p}_{n;j} \triangleq \frac{n\underline{p}_n - e_j}{n-1} = \underline{p}_n - \frac{1}{n-1}(e_j - \underline{p}_n), \tag{2.7}$$

and the *pseudo-values*

$$\tilde{\Phi}_{n;j}(\underline{p}_n) \triangleq n\Phi(\underline{p}_n) - (n-1)\Phi(\underline{p}_{n;j}). \tag{2.8}$$

Now

$$\Phi(\underline{p}_{n;j}) = \Phi(\underline{p}_n) - \frac{1}{n-1}G(\underline{p}_n)(e_j - \underline{p}_n) + o_p((n-1)^{-1}), \tag{2.9}$$

and thus

$$\tilde{\Phi}_{n;j}(\underline{p}_n) = \Phi(\underline{p}_n) + G(\underline{p}_n)(e_j - \underline{p}_n) + o_p(1). \tag{2.10}$$

For the average pseudovalue we find

$$\tilde{\Phi}_n(\underline{p}_n) \triangleq \sum_{j=1}^{m} \underline{p}_{nj}\tilde{\Phi}_{n;j}(\underline{p}_n) = \Phi(\underline{p}_n) + o_p(1), \tag{2.11}$$

and for the variance of the pseudo-values

$$\sum_{j=1}^{m} \underline{p}_{nj}(\tilde{\Phi}_{n;j}(\underline{p}_n) - \tilde{\Phi}_n(\underline{p}_n))(\tilde{\Phi}_{n;j}(\underline{p}_n) - \tilde{\Phi}_n(\underline{p}_n))' = G(\underline{p}_n)V(\underline{p}_n)G'(\underline{p}_n) + o_p(1), \tag{2.12}$$

which shows that the variance of the pseudo-values can be used to estimate the variance of $\Phi(\underline{p}_n)$.

To apply bias correction we again need an extra term in the expansion.

$$\phi_s(\underline{p}_{n;j}) = \phi_s(\underline{p}_n) - \frac{1}{n-1}g_s(\underline{p}_n)(e_j - \underline{p}_n) + \frac{1}{2}\frac{1}{(n-1)^2}(e_j - \underline{p}_n)'H_s(\underline{p}_n)(e_j - \underline{p}_n) + o_p((n-1)^{-2}), \tag{2.13}$$

and thus

$$\{\tilde{\Phi}_{n;j}(\underline{p}_n)\}_s = \phi_s(\underline{p}_n) + g_s(\underline{p}_n)(e_j - \underline{p}_n) + \frac{1}{2}\frac{1}{n-1}(e_j - \underline{p}_n)'H_s(\underline{p}_n)(e_j - \underline{p}_n) + o_p((n-1)^{-1}). \tag{2.14}$$

This gives for the average pseudo-value

$$\tilde{\phi}_{ns}(\underline{p}_n) = \phi_s(\underline{p}_n) + \frac{1}{2}\frac{1}{n-1}\ \mathrm{tr}\ H_s(\underline{p}_n)V(\underline{p}_n) + o_p((n-1)^{-1}), \tag{2.15}$$

Thus the average pseudo-value corrects for bias.

3. Some Lisp

3.1. Symbols. In Xlisp-Stat symbols are arrays, with four elements. The elements of the array are the *print name*, the symbol's *value*, the symbol's *function definition*, and the *property list* of the symbol. All of the last three cells can be empty. The following small sessions illustrates how the various components of a symbol are filled. First we see that a `defun` defines a symbol and puts a closure in the function cell. There is nothing in the value cell.

```
> (defun aragon ())
ARAGON
> (symbol-function 'aragon)
#<Closure-ARAGON: #492428>
> (symbol-value 'aragon)
Error: The variable ARAGON is unbound.
Happened in: #<Subr-TOP-LEVEL-LOOP: #44e2a4>
> (symbol-name 'aragon)
"ARAGON"
```

A `setq` also defines a symbol, and puts a value in the value cell.

```
> (setq bilbo 3)
3
> (symbol-value 'bilbo)
3
> (symbol-function 'bilbo)
Error: The function BILBO is unbound.
Happened in: #<Subr-TOP-LEVEL-LOOP: #44e2a4>
> (symbol-name 'bilbo)
"BILBO"
```

We can also create symbols, in which both the value cell and the function cell are empty.

```
> (intern "FRODO")
FRODO
:INTERNAL
> (symbol-function 'frodo)
Error: The function FRODO is unbound.
Happened in: #<Subr-TOP-LEVEL-LOOP: #44e2a4>
> (symbol-value 'frodo)
Error: The variable FRODO is unbound.
Happened in: #<Subr-TOP-LEVEL-LOOP: #44e2a4>
> (symbol-name 'frodo)
"FRODO"
```

3.2. Data Types and Function Types. Xlisp-Stat has many data types. There are strings, characters, numbers, objects, arrays, structures, streams, packages, hash-tables, and lists. There are also various types of fuunctions. This is illustrated by the following session.

```
> (symbol-function 'mean)
#<Subr-MEAN: #10b290>
> (symbol-function 'standard-deviation)
#<Byte-Code-Closure: #1b4620>
> (defun cajun ())
CAJUN
> (symbol-function 'cajun)
#<Closure-CAJUN: #1298a0>
> (symbol-function 'setf)
#<FSubr-SETF: #e91d0>
```

Subr's are built-in functions and fsubr's are built-in special forms, which are functions that do not evaluate their arguments. Functions added as Lisp code are closures, and if the functions are byte-compiled they are byte-code-closures.

3.3. Evaluation. To understand the use of symbols better, we look into how Xlisp evaluates its expressions ([1], page 7).

The first rule is that strings, characters, numbers, objects, prototypes, structures, streams, subrs, fsubrs, and closures evaluate to themselves. Symbols evaluate to the value in their value cell, if there is any. If there is none, it's an error.

Lists are evaluated by looking at the car (first element) of the list first. If it is a symbol, the function cell is retrieved. If it is a lambda-expression, a closure is constructed. If it is a subr or fsubr or closure, it stand for itself. If it is anything else, it's an error.

We then look at the remaining elements of the list. If the car was a subr or closure, the elements are evaluated, and the subr or closure is applied to it. If the car was an fsubr, then the fsubr is called with the remaining elements as unevaluated arguments. It is a macro, then the macro is expanded, and the expanded macros is called with the remaining elements as unevaluated arguments.

```
> (make-symbol "BOZO")
#:BOZO
> bozo
Error: The variable BOZO is unbound.
> (bozo)
Error: The function BOZO is unbound.
> (setf (symbol-function 'bozo)
        #'(lambda (x) (* x x)))
#<Closure: #4907d8>
> (bozo 3)
9
> (setf (symbol-value 'bozo) 3)
3
> (+ bozo bozo)
6
> (bozo bozo)
9
```

4. The Jackknife Code

4.1. A Pseudovalue Closure.

The key function of our Jackknife system is the following.

```
(defun jack-pseudo-values (func)
  #'(lambda (data)
      (let* (
             (n (length data))
             (l (iseq n))
             (f (symbol-function func))
             (p (make-list n))
             )
(dotimes (i n)
(setf (elt p i)
      (funcall f (select data (which (/= i l))))))
(- (* n (make-list n :initial-element (funcall f data)))
   (* (1- n) p))
))
)
```

This function takes as its single argument a symbol `func`, with a non-empty function cell. The function `(symbol-function 'func)` takes a sequence of objects as an argument, and produces a function which computes all pseudo-values. Thus

```
> (jack-pseudo-values 'max)
#<Closure: #4b2404>
> (funcall (jack-pseudo-values 'max) (normal-rand 5))
(1.8794036049709275 2.123838484548437 1.8794036049709275 1.8794036049709275
1.8794036049709275)
```

4.2. Pseudovalue Means and Dispersion. Two additional functions are defined, which return functions computing pseudo-value averages and dispersion. They call `jack-pseudo-values`.

```
(defun jack-average (func)
  #'(lambda (data)
      (average (funcall (jack-pseudo-values func) data)))
)

(defun jack-dispersion (func)
  #'(lambda (data)
      (dispersion (funcall (jack-pseudo-values func) data)))
)
```

We need utilities to actually compute averages and dispersions. The Xlisp-Stat functions `mean` and `standard-deviation` will not do, because they are *vector reducing*. Thus they always return a single number, no matter if one applies them to lists, arrays, or lists of arrays.

```
(defun average (x)
"Args: list
Takes the average of all elements in a list"
(if (not (listp x)) (error "Argument for AVERAGE must be a list"))
  (/ (apply #'+ x) (length x)))

(defun dispersion (x)
"Args: list
Computes the dispersion of all elements in a list.
The list must have either numbers or sequences or arrays
of numbers"
(if (not (listp x)) (error "Argument for DISPERSION must be a list"))
  (let (
        (n (length x))
        (m (average x))
        (s (apply #'+ (mapcar #'(lambda (z) (outer-product z z)) x)))
        )
    (- (/ s n) (outer-product m m))
))
```

We can now investigate in detail what arguments the function that we are Jackknifing can handle, i.e. for what types of functions they produce reasonable results. Let us call the function `foo`, with argument `foo-arg`, returning resulty `foo-result`. Looking at `jack-me` first shows that `foo-arg` must be a sequence, because we apply `length`. It also shows that `foo-result` must be able to be multiplied by a scalar and to be added to itself. Thus `foo-result` can be a number, a vector, a list, an array, a list of arrays, an array of arrays, and so on. If we want to use `jackknife-dispersion`, we see that it uses the `outer-product` function to compute the dispersion. This means that `foo-result` can be an array, which is first displaced to a vector before `outer-product` is applied.

The function `sv-decomp`, for instance, returns a list of three arrays and a logical constant. If we want to Jackknife the singular value decomposition, we have to define a function which makes a list of vectors into a matrix, and which strips off the logical constant from the result.

```
(defun my-svd-decomp (list-of-vectors)
  (select (sv-decomp (apply #'bind-rows list-of-vectors)) '(0 1 2))
)
```

Then we can say `(jackknife-pseudo-values 'my-svd-decomp)`. But we cannot say `(jackknife-dispersion 'my-svd-decomp)`, because `outer-product` does not know what to do with lists of arrays.

4.3. Tying it all together. Finally, we have written a function which takes the symbol `func` and produces three new symbols in the current environment. They are `JACK-PSEUDO-VALUES-OF-FOO`, `JACK-AVERAGE-OF-FOO`, and `JACK-DISPERSION-OF-FOO`. These three symbols have the three closures discussed above in their function cells, so they can be called directly, without using `funcall`.

```
(defun jack-me (func)
(let* (
      (ffunc (symbol-name func))
      (pname (concatenate 'string "JACK-PSEUDO-VALUES-OF-" ffunc))
      (aname (concatenate 'string "JACK-AVERAGE-OF-" ffunc))
      (dname (concatenate 'string "JACK-DISPERSION-OF-" ffunc))
      (pfunc (jack-pseudo-values func))
      (afunc (jack-average func))
      (dfunc (jack-dispersion func))
      (psymb (intern pname))
      (asymb (intern aname))
      (dsymb (intern dname))
      )
  (setf (symbol-function psymb) pfunc)
  (setf (symbol-function asymb) afunc)
  (setf (symbol-function dsymb) dfunc)
(values psymb asymb dsymb)
))
```

Here is a final example. We define

```
(defun singular-values (list-of-vectors)
  (elt (sv-decomp (apply #'bind-rows list-of-vectors)) 1)
)
```

and

```
(setq x (mapcar #'(lambda (n) (coerce (normal-rand n)'vector))
  (repeat 4 100)))
```

This makes a list with 100 vectors of 4 normal deviates, while the function `singular-values` returns a vector of singular values. Thus

```
> (singular-values x)
#(11.536806970914478 10.579640568812247 9.455345232286554 8.573864335865494)
```

We then say

```
> (jack-me 'singular-values)
JACK-PSEUDO-VALUES-OF-SINGULAR-VALUES
JACK-AVERAGE-OF-SINGULAR-VALUES
JACK-DISPERSION-OF-SINGULAR-VALUES
> (jack-average-of-singular-values x)
#(16.84083238208193 15.911544545603567 14.182201645750206
13.365352055410636)
> (print-matrix (jack-dispersion-of-singular-values x))
#2a(
    (  52.7149      -1.44281      -4.59448       1.29799    )
    ( -1.44281       52.5639      -7.75757      -2.77336    )
    ( -4.59448      -7.75757       42.6533       2.54817    )
    (  1.29799      -2.77336       2.54817       36.4671    )
   )
NIL
```

Since in this case the population singular values are all 10, we are not too impressed with the actual performace of the jackknife (it is plausible, that in some cases the singular value decomposition has not converged, in which case the pseudo-values can behave very strangely).

References

1. T. Almy, *XLISP-PLUS: Another Object-oriented Lisp*, 1993, Version 2.1f.
2. B. Efron and R. J. Tibshirani, *Introduction to the Bootstrap*, Chapman & Hall, New York, NY, 1993.
3. P. Graham, *On Lisp. Advanced Techniques for Common Lisp*, Prentice Hall, Englewood Cliffs, NJ, 1994.
4. J. Hemelrijk, *Underlining Random Variables*, Statistica Neerlandica **20** (1966), 1–7.
5. J. Hurt, *Asymptotic Expansions of Functions of Statistics*, Aplikace Matematiky **21** (1976), 444–456.
6. R. G. Miller, *The Jackknife – A Review*, Biometrika **61** (1974), 1–17.
7. L. Tierney, *LISP-STAT. An Object-Oriented Environment for Statistical Computing and Dynamic Graphics*, Wiley, New York, NY, 1990.
8. J. W. Tukey, *Bias and Confidence in not Quite Large Samples*, Annals of Mathematical Statistics **29** (1958), 614–614, Abstract.

UCLA Statistics Program, 8118 Mathematical Sciences Building, University of California at Los Angeles

E-mail address: deleeuw@stat.ucla.edu