Paper 034-2013

A Flock of C-Stats, or Efficiently Computing Multiple Statistics for Hundreds of Variables

Steven Raimi, Marketing Associates, Detroit, MI and Wilmington, DE
Bruce Lund, Marketing Associates, Detroit, MI and Wilmington, DE

ABSTRACT

In other presentations, the authors have provided macros that efficiently compute univariate statistics for hundreds of variables at a time. The classic example is when a modeler must fit a binary model (two-valued target) and has available hundreds of potential numeric predictors. Such situations may occur when third-party data sets are added to in-house transactional data for direct marketing or credit scoring applications. The paper describes the SAS® code to compute these statistics, focusing on the techniques that make these macros efficient. Topics include macro techniques for identifying and managing the input variables, restructuring the incoming data, and using hash objects to swiftly count the number of distinct values for each variable.

INTRODUCTION

This paper provides SAS code for computing c-statistics, as well as other measures of predictive power, for hundreds of numeric predictor variables in an efficient manner. The authors outline their SAS macro %XC_STAT that incorporates this SAS code, and discuss the programming techniques that enable that efficiency. Data dictionary tables querying, hash object programming, leveraging the output of the SUMMARY step, and recoding to maximize the use of array processing are covered.

Binning (coarse classing) of numeric predictors is generally a good practice before beginning the modeling process. Binning removes distortion from outliers and better reveals trends with respect to the target. %XC_STAT supports binning by collapsing the levels of predictors where the number of levels exceeds a user-provided parameter _N. This binning is accomplished by a PROC RANK step for variables whose number of levels exceeds _N. The determination of the number of levels of the predictor variables is accomplished in the first step of %XC_STAT by a hash coding routine.

We have found that the macro processes hundreds of numeric predictors, in datasets of 500,000 to 1,000,000 records, in less than a minute.

The concepts in this paper were discussed in Raimi and Lund (2011, 2012).

C-STATISTIC

Consider a data set with a numeric (or ordered) predictor X, and a binary target variable Y with values 0 and 1.

- Let M give the count of pairs of observations where one observation has Y = 0 and the other has Y = 1. Such pairs are called “informative pairs”.
- Let C (for Concordant) be the count of informative pairs with a greater value of X when Y=1 than the value of X when Y=0.
- Let T (for ties) be the count of informative pairs where the values of X are equal.

The formula for the c-statistic is: (C + 0.5*T) / M

Computing the c-statistic from a frequency table

The c-statistic can easily be computed from a frequency table. Consider the table of X and Y values where $f_r(i)$ gives the frequency for the (i, r) cell where i = 1 ... K and r = 0, 1.

Table 1 - Frequency Table

	Y	
X	Y = 0	Y = 1
X_1	$f_0(1)$	$f_1(1)$
X_2	$f_0(2)$	$f_1(2)$
...	...	...
X_K	$f_0(K)$	$f_1(K)$
SUM	$\sum f_0(i)$	$\sum f_1(i)$

The values for M, C, and T can be given in terms of the frequency table values as shown:

$$M = \sum_{i=1}^{K} f_0(i) * \sum_{i=1}^{K} f_1(i)$$
$$C = \sum_{i=1}^{K-1} \sum_{j=i+1}^{K} f_0(i) * f_1(j)$$
$$T = \sum_{i=1}^{K} f_0(i) * f_1(i).$$

Then c-statistic = (C + 0.5*T) / M.

We'll call MAX(C_STAT, 1 - C_STAT) the "magnitude" of the c-statistic.[1] The magnitude is at least 0.50.

The c-statistic for Table 2 is computed by:
(C + 0.5*T) / M = [(2 + 2 + 1) + 0.5*(0 + 1 + 2)] / (5 * 2) = 0.65

Table 2 – Frequency Table

	Y	
X	Y = 0	Y = 1
1	2	0
2	1	1
3	2	1
SUM	5	2

Other Computational Methods for c-Statistic

PROC FREQ or PROC NPAR1WAY, followed by simple DATA STEP computations, can compute the c-statistic for a single predictor. See SAS code provided by Xie (2009). Hermansen (2008) provides SAS SQL code to compute the c-statistic.

In the section below SAS code is provided which gives a simplification of the %XC_STAT code for efficiently computing c-statistics for virtually any number of predictor variables. The computations utilize (C + 0.5*T) / M.

Converting Data to Frequency Tables

This section gives SAS code which could be adapted to compute c-statistics for hundreds of variables in an efficient manner. The code below is skeletal. It involves no input data processing or macro coding.

STEP1: The input data is processed by PROC SUMMARY with MISSING option. In the example data set TARGET is the binary target variable (no missing values) and X1-X2 are numeric predictors which may have missing values.

```
DATA DATASET; input X1 X2 TARGET @@; datalines;
1 1 0 1 1 1 . 5 0 6 3 0
6 3 1 6 5 0 6 5 1 4 5 0
4 6 1 4 7 0 4 8 1 . . 0
;

PROC SUMMARY DATA = DATASET NOPRINT MISSING;
CLASS X1 X2 TARGET;
TYPES (X1 X2) * TARGET;
OUTPUT OUT = __X__SUMOUT1;

PROC PRINT; run;
```

Obs	X1	X2	TARGET	_TYPE_	_FREQ_
1	.	.	0	3	1
2	.	1	0	3	1
3	.	1	1	3	1
4	.	3	0	3	1
5	.	3	1	3	1
6	.	5	0	3	3

[1] If c-statistic for X and Y is "c1", then reversing the coding of Y (0 to 1, 1 to 0), a new c-statistic is obtained "c2" where c1 = 1 - c2.

7	.	5	1	3	1
8	.	6	1	3	1
9	.	7	0	3	1
10	.	8	1	3	1
11	.	.	0	5	2
12	1	.	0	5	1
13	1	.	1	5	1
14	4	.	0	5	2
15	4	.	1	5	2
16	6	.	0	5	2
17	6	.	1	5	2

The output data set __X__SUMOUT1 includes an observation for each combination of a value of a predictor and a value of the TARGET as well as the value of _FREQ_, which gives the count of raw observations for this combination.

In a practical case there would be far fewer observations in __X__SUMOUT1 than in the input DATASET. The variable _TYPE_ delineates the observations related to X1 from those related to X2. _TYPE_ plays a crucial role in the following steps.

STEP2: The next step involves somewhat tedious book-keeping. The values of X1 and X2 are each recoded to begin at 1 and to end with the number of their levels (3 for X1 and 6 for X2). This step is needed to set up the array processing in STEP3. Note the new variable named RECODE.

```
DATA __X__SUMOUT2(DROP = VL I)  __X__VL(KEEP = _TYPE_ VL);
SET __X__SUMOUT1 END = EOF;
    BY _TYPE_;
    ARRAY V{*} X1 - X2;
    RETAIN RECODE NEW_VALUE OLD_VALUE 0;
    IF FIRST._TYPE_
    THEN DO;
        RECODE = 0;
        NEW_VALUE = .;
        OLD_VALUE = .;
        END;
    DO I = 2 to 1 by -1;
/* V{I} > . DETERMINES WHICH VARIABLE IS BEING RECODED */
        IF V{I} > . THEN NEW_VALUE = V{I};
        END;
    IF NEW_VALUE NE OLD_VALUE
    THEN DO;
        DO I = 2 TO 1 BY -1;
            IF V{I} > .
            THEN DO;
                RECODE = RECODE + 1;
                END;
            END;
        END;
    OUTPUT __X__SUMOUT2;
    IF LAST._TYPE_ THEN DO;
        DO I = 2 TO 1 BY -1;
            IF V{I} > . THEN VL = RECODE;
            END;
        OUTPUT __X__VL;
        END;
    OLD_VALUE = NEW_VALUE;
```

```
PROC PRINT DATA = __X__SUMOUT2;
PROC PRINT DATA = __X__VL; run;
```

Obs	X1	X2	TARGET	_TYPE_	_FREQ_	RECODE	NEW_ VALUE	OLD_ VALUE
1	.	.	0	3	1	0	.	.
2	.	1	0	3	1	1	1	.
3	.	1	1	3	1	1	1	1
4	.	3	0	3	1	2	3	1
5	.	3	1	3	1	2	3	3
6	.	5	0	3	3	3	5	3
7	.	5	1	3	1	3	5	5
8	.	6	1	3	1	4	6	5
9	.	7	0	3	1	5	7	6
10	.	8	1	3	1	6	8	7
11	.	.	0	5	2	0	.	.
12	1	.	0	5	1	1	1	.
13	1	.	1	5	1	1	1	1
14	4	.	0	5	2	2	4	1
15	4	.	1	5	2	2	4	4
16	6	.	0	5	2	3	6	4
17	6	.	1	5	2	3	6	6

Obs	_TYPE_	VL
1	3	6
2	5	3

STEP3: The values of the cells in the frequency tables for X1 * TARGET and X2 * TARGET are computed and used in the formula [C + 0.5*T] / M to compute the magnitude of the c-statistic, given as MAX(C_STAT, 1 - C_STAT).

```
DATA __X__C_STAT;
MERGE __X__SUMOUT2 __X__VL;
    BY _TYPE_;
    ARRAY V{*} X1 - X2;
/* ASSUMES 10 IS THE MAXIMUM NUMBER OF LEVELS FOR A PREDICTOR */
    ARRAY F0 {10} F0_1 - F0_10;
    ARRAY F1 {10} F1_1 - F1_10;
    RETAIN CURR_VAR_NUM 0;
    RETAIN F0_1 - F0_10;
    RETAIN F1_1 - F1_10;
    IF FIRST._TYPE_ THEN DO;
        DO I = 1 TO 10;
            F0{I} = 0;
            F1{I} = 0;
            END;
        CURR_VAR_NUM + 1;
        END;
    DO VAR_I = 1 TO 2;
        IF V(VAR_I) > .
        THEN DO;
            IF TARGET = 0 THEN F0{RECODE} = _FREQ_;
            ELSE IF TARGET = 1 THEN F1{RECODE} = _FREQ_;
            END;
        END;
    IF LAST._TYPE_ THEN DO;
        C_PAIR  = 0;
        F0_TOTAL = 0;
        F1_TOTAL = 0;
        C_STAT  = 0;
        DO I = 1 TO VL;
            F0_TOTAL = F0_TOTAL + F0{I};
            F1_TOTAL = F1_TOTAL + F1{I};
            END;
        DO I = 1 TO VL;
            DO J = 1 TO VL;
                IF I < J THEN C_STAT = C_STAT  + F0{I}*F1{J}; /* CONCORDANT PAIRS */
                IF I = J THEN DO;
```

```
                C_STAT  = C_STAT + .5*FO{I}*F1{I}; /* TIED PAIRS */
                END;
            END;
        END;
    C_PAIR  = FO_TOTAL * F1_TOTAL; /* INFORMATIVE PAIRS */
    C_STAT  = MAX( C_STAT  / C_PAIR,  1 - C_STAT  / C_PAIR );
    OUTPUT;
    END; /* END OF: IF LAST._TYPE_ THEN DO; */

PROC FORMAT;
VALUE __ACTIVE
    . = 'INACTIVE'
    OTHER = 'ACTIVE';

PROC PRINT DATA = __X__C_STAT; VAR X1 X2 VL C_STAT FO_1 - FO_10 F1_1 - F1_10;
FORMAT X1 X2 __ACTIVE. C_STAT 7.4;
run;
```

O b s	X 1	X 2	V L	C _ S T A T	F 0 _ 1	F 1 _ 1
1	INACTIVE	ACTIVE	6	0.5500	1 1 3 0 1 0 0 0 0 0	0 1 1 1 1 0 1 0 0 0
2	ACTIVE	INACTIVE	3	0.5000	1 2 2 0 0 0 0 0 0 0	0 1 2 2 0 0 0 0 0 0

INFORMATION VALUE STATISTIC

The information value (IV) of a predictor and the target can be given as a formula involving an X-Y frequency table.

Information Value IV is not computed for Table 2 since F1(1) = 0 and LOG is not defined at 0. If F1(1) is re-set to be 1 in Table 3, then IV = .0924 as shown by the calculation below:

Table 3 – Frequency Table (Table 2 modified)

X	Y = 0	Y = 1	b: Col % Y = 0	g: Col % Y = 1	Log(g/b)	g - b	(g - b) * Log(g/b)
1	2	1	0.4	0.333	-0.1833	-0.067	0.0123
2	1	1	0.2	0.333	0.5098	0.133	0.0678
3	2	1	0.4	0.333	-0.1833	-0.067	0.0123
SUM	5	3				IV =	0.0924

This formula can be included in STEP3 as shown:

```
IF LAST._TYPE_ THEN DO;
    IV = 0;
< STATEMENTS TO DEFINE FO{I} F1{I} FO_TOTAL F1_TOTAL AS GIVEN PREVIOUSLY >
    DO I = 1 TO VL;
        IF (F1{I} > 0 & FO{I} > 0)
        THEN DO;
            IV = IV + LOG( (F1{I}*FO_TOTAL)/(FO{I}*F1_TOTAL) ) *
                              (F1{I}/F1_TOTAL - FO{I}/FO_TOTAL);
            END;
        ELSE IF (F1{I} = 0 OR FO{I} = 0)
        THEN DO;
            IV = .;
            END;
        END;
    END;  /* END of IF LAST.TYPE THEN DO; */
```

X-STATISTIC

The x-statistic is defined as the "logistic model c-statistic" of a predictor X as a class-variable in the logistic regression of Y against X as shown: PROC LOGISTIC; CLASS X; MODEL Y = X;

It is not necessary to run PROC LOGISTIC to compute the x-statistic. The x-statistic can be computed from an X-Y frequency table according to the formula below.

Let $M = \sum_{i=1}^{K} f_0(i) * \sum_{i=1}^{K} f_1(i)$

Let $Z = \sum_{i=1}^{K-1} \sum_{j=i+1}^{K} \text{Abs}\ (f_0(i)*f_1(j) - f_0(j)*f_1(i))$. (Abs = absolute value)

Then x-statistic = 0.5 * [Z / M + 1]

The x-statistic ranges between 0.5 and 1.0. For example, the x-statistic for Table 2 is 0.75 as seen in the calculation below:

x-stat = 0.5 * [(2 + 2 + 1)/10 + 1] = 0.75

This formula can be included in STEP3 as shown:

```
IF LAST._TYPE_ THEN DO;
    X_STAT = 0;
< STATEMENTS TO DEFINE FO{I} F1{I} C_PAIR AS GIVEN PREVIOUSLY >
    DO I = 1 TO VL;
        DO J = 1 TO VL;
            IF I < J THEN X_STAT = X_STAT  + ABS(FO{I}*F1{J} - F1{I}*FO{J});
            END;
        END;
        X_STAT  = .5 * (X_STAT  / C_PAIR  + 1);
    END;  /* END of IF LAST.TYPE THEN DO; */
```

Lift of x-Statistic over c-Statistic

We introduce a statistic that measures the potential benefit of finding a non-monotonic transformation of X.[2] A measure which compares c-statistic and x-statistic is:

Lift = (x-statistic - max(c-statistic, 1 - c-statistic)) / max(c-statistic, 1 - c-statistic)

Since a large number of levels of X can inflate the x-statistic, the Adjusted Lift is defined.

Adjusted Lift = Lift / (K – 1)

Adjusted Lift computes the "lift per degree of freedom". The Adjusted Lift is generally comparable across predictors.

BINNING THE VALUES OF PREDICTOR VARIABLES

Binning (collapsing) the levels of numeric predictors X for a binary logistic regression is generally a good practice. Binning removes distortion from outliers and better reveals trends with respect to the target.

A binned variable is suitable for a weight-of-evidence (WOE) transformation, a common practice in preparing variables in credit risk modeling.[3]

The information value (IV) and the x-statistic are not useful when X has many levels (perhaps 15 or more) since each reflects over-fitting of the model PROC LOGISTIC; CLASS X; MODEL Y = X; Both the IV and x-statistic are decreasing as levels of the X are collapsed.

Binning is somewhat subjective. %XC_STAT provides a solution (based on PROC RANK) but other methods of binning could be used as a preliminary step.

---

[2] See Raimi, S. and Lund, B. (2012).
[3] See Finlay p146.

THE PARAMETERS OF %XC_STAT

%XC_STAT(Dataset, _N, Target, Exclude_cols);

Dataset	Dataset to be analyzed.
_N	User-defined maximum number of distinct values a predictor variable may have in order to be processed as-is. Variables with more distinct values are processed by PROC RANK with GROUPS = _**N** to reduce the number of levels to, or below, _**N** prior to statistical calculations.
Target	The response variable. Target has numeric values of 0 and 1 with no missing values.
Exclude_cols	The name(s) of numeric variables to be left out of the analysis. (Character variables are automatically excluded).

For example: %XC_STAT(MYDATA, 10, Y, EXCLUDE1 EXCLUDE2 EXCLUDE3);

In this example any numeric variable to be analyzed which has more than 10 levels is processed through PROC RANK with GROUPS=10. In this case the statistics (c-stat, etc.) are computed for the ranked values.

THE HIGH-LEVEL PROCESSING FLOW OF %XC_STAT

Perform input processing steps such as:
- Verify TARGET is numeric and has values 0 and 1 and no missing
- Obtain preliminary list of eligible predictor variables (numeric variables except for those in Exclude_cols) and save count in &VAR_NUM.

An example is given in: **AN EXAMPLE OF %XC_STAT INPUT PROCESSING STEPS.**

Create Macro Variables for Use in PROC RANK, PROC SUMMARY, and DATA STEPS

- Eliminate numeric variables with all missing values or where count of distinct levels is equal to count of rows in DATASET and also exceeds MAX_LEVELS_PERMITTED = min(&_N, 50). (*)
- After eliminations, update &VAR_NUM and save names of final predictors in &&V&I where I = 1 to &VAR_NUM
- Define VAL_LEVELS&I to be the number of distinct levels of the predictor where I = 1 to &VAR_NUM
- %LET USE_RANKS = "Y" if any VAL_LEVELS&I exceeds MAX_LEVELS_PERMITTED

The values for VAL_LEVELS&I are determined via a hash coding step. Details of hash coding to populate VAL_LEVELS&I are given in **DETERMINING LEVEL COUNTS FOR EACH VARIABLE**.

(*) Removes numeric variables such as a sequence number or ID number.

PROC RANKS (if "&USE_RANKS" = "Y")

```
PROC RANK DATA = &DATASET OUT = RANKOUT GROUPS = &MAX_LEVELS_PERMITTED;
    VAR
    %DO I = 1 %TO &VAR_NUM;
        %IF %CMPRES(&&VAL_LEVELS&I) GT &MAX_LEVELS_PERMITTED
            AND %CMPRES(&&VAL_LEVELS&I) NE &DATASET_ROWS %THEN
            &&V&I;
        %END;
        ;  /* ';' is needed to complete the VAR statement */
    RANKS
    %DO I = 1 %TO &VAR_NUM;
        %IF %CMPRES(&&VAL_LEVELS&I) GT &MAX_LEVELS_PERMITTED
            AND %CMPRES(&&VAL_LEVELS&I) NE &DATASET_ROWS %THEN
            &&V&I;
        %END;
        ;  /* ';' is needed to complete the RANKS statement */
```

Only variables whose levels exceed &MAX_LEVELS_PERMITTED are actually ranked.

PROC SUMMARY and 2 DATA STEPS

STEP1, STEP2, and STEP3 from the beginning of the paper are generalized, and included in the macro %XC_STAT.

AN EXAMPLE OF %XC_STAT INPUT PROCESSING STEPS

Identify Relevant Variables using Dictionary Tables

The practical utility of the %XC_STAT macro would be quite limited if the user had to enter the name of every variable to be considered. Therefore, %XC_STAT queries the SAS version of data dictionary tables (especially SASHELP.VCOLUMN) to automate selection of all appropriate columns. The following code builds a list of all numeric variables (except the Target variable, and any excluded by the exclude_cols parameter).

First Step:

```
*** Convert exclude_cols parameter into proper form for an IN values list by replacing blanks
with double quote-space-double quote, surrounded by parentheses and a start or end quote;

   %LET KILLTHESE=%STR();
   %IF NOT("%CMPRES(&EXCLUDE_COLS)" = %STR("")
           OR "%CMPRES(&EXCLUDE_COLS)" = %STR(%" %")
          )
   %THEN %DO;
      %LET CLEAN = %SYSFUNC(COMPBL(&EXCLUDE_COLS));
      %LET TRANSLATED =
                          %QSYSFUNC(TRANSTRN(
                                             &CLEAN,
                                             %STR( ),
                                             %STR(" ")
                                             )
                                    );
      %LET KILLTHESE=%UPCASE(%STR(%")&TRANSLATED%STR(%"));
   %END;
```

Second Step:

```
***  Create preliminary macro var list. Program assumes variables will be processed in physical
     column order, so order by NPOS;
***  Before this code is run the %XC_STAT macro parameter DATASET has been decomposed into
     &USER_LIBRARY and &USER_MEMBER;
***  Also before this code is run %XC_STAT has determined a preliminary count of numeric
     predictors (excluding EXCLUDE_COLS) and saved this count in &VAR_NUM;

   PROC SQL NOPRINT;
      SELECT NAME, TYPE
      INTO   :PRELIMV1-:PRELIMV%CMPRES(&VAR_NUM),
             :VTYPE1-:VTYPE%CMPRES(&VAR_NUM)
      FROM   SASHELP.VCOLUMN
      WHERE  MEMNAME="%UPCASE(&USER_MEMBER)"
      AND    TYPE = 'num'
      AND    UPCASE(NAME) NE "%UPCASE(&TARGET)"

   %IF &KILLTHESE NE %STR() %THEN
      AND    UPCASE(NAME) NOT IN (&KILLTHESE);

   %IF &USER_LIBRARY NE %THEN
      AND    LIBNAME = "%UPCASE(&USER_LIBRARY)";

      ORDER BY NPOS;
   QUIT;
```

Note: In later steps the "PRELIMV" list will be reduced by removing predictor variables whose values are all missing and predictor variables whose count of distinct values equals the number of data set rows (such as a sequence number) while also exceeding &MAX_LEVELS_PERMITTED.

Discussion:
It took some testing to get the WHERE clause right. The NAME and LIBNAME columns in the VCOLUMN table retain the case of the users' libname and column definition entries, so matching UPCASE (or LOWCASE) functions on both the column reference and the IN list entries have to be used.

Matching case is also critical for MEMNAME, but we found that a formula around the MEMNAME column reference is a BAD idea – that will make the query return information on every column in every table in every dataset available in your session, which is even more time-consuming than it sounds. The only recourse when that happens is to forcibly terminate the session. Instead, we rely on the fact that MEMNAME is stored exclusively in upper case.

HASH OBJECT – DETERMINING LEVEL COUNTS FOR EACH VARIABLE

Another preprocessing step in %XC_STAT finds the count of levels for each predictor. These counts are used to decide which variables to bin via the %XC_STAT macro parameter _N through PROC RANK GROUPS = &_N. Standard data step code could be used to step through the dataset's rows and keep a variable which accumulates distinct values. However, that would require a large variable (large enough to keep an arbitrary number of values) – one for each separate variable being examined. This quickly overruns memory when dealing with high cardinality columns (where the ratio of distinct values to rows is high). PROC SQL enables you to use SELECT DISTINCT var1, var2…varx, but that yields the distinct number of value COMBINATIONS, not distinct values for each column. SELECT DISTINCT var1, DISTINCT var2,…,DISTINCT varx quickly exhausts system memory. Fortunately, a better method is available.

Hash Objects Overview:
SAS will be introducing some future data step enhancements as Data Step Component Objects. The first (and so far only) of these are Hash Objects, introduced in SAS v9. Hash objects are memory-resident packages of attributes (analogous to variables) and methods that access and modify those attributes or perform related functions. They directly address RAM memory to efficiently store, search, retrieve, and sort data. These objects are gone at the end of the data step in which they're used – they are not part of the output dataset (though you could save the internal data for use later, if desired). There is only ONE copy of the data in each hash object (unlike dataset variables, where the defined length is in memory once for every dataset row). You can use many hash objects, if desired, and still remain comfortably within your memory constraints.

To use a hash object, one must first define the object (the DECLARE() method). By default, there is no data associated with the hash object at this point (although you can specify a dataset to be loaded – see details in the SAS hash object tip sheet [4]). The hash object is merely defined, not usable – one must create an instance of it. The next step is to define the database column(s) to be used as the key to the internal data attributes, which is done using the DECLAREKEY() method. If you want to carry other values within the hash object (to serve as an in-memory lookup table, for example), you also use the DECLAREDATA() method.

The hash object thus has data attributes that match the specified dataset columns. The programmer has to make sure that the definitions for these attributes match the names and definitions of the associated data columns. When dataset rows are fetched, the hash object data attributes are automatically synchronized to the values of the current row. But the hash object does more than just reflect the current dataset row's values. The hash object maintains dynamic arrays of values, one array for the key and one for each additional data column defined. If the programmer executes an ADD() method, the data attributes (mirroring their dataset column current values) are stored in newly created entries in these arrays. All array indices are based on the hashed value of the key column(s). They can thus serve as lookups, and can be applied to other programming challenges.

First Step:
Hash objects are particularly good for identifying or manipulating desired subsets of the dataset. One such purpose is examining all values of a variable, and retaining a copy of those meeting particular criteria – in our case, the set of distinct values in the dataset. We use a separate hash object for each variable of interest. Because all the data and operations are done in memory, hash objects are very fast. The logic for counting unique values is summarized in this pseudo-code:

Pseudo-code:
```
For each variable, do
        Establish an empty list of unique values
        For each value, do
                Is the current value found on the list?
                If No, add the value to the list
```

[4] http://support.sas.com/rnd/base/datastep/dot/hash-tip-sheet.pdf

```
                Else, skip to next loop iteration
        End
End
```

In the following code we show the processing of just one predictor variable in order to remove macro processing complexities from the discussion.

SAS code:

```
DATA __X__COUNT_LEVELS (DROP=INCR_VCOUNT TOTAL);
      RETAIN INCR_VCOUNT 1 TOTAL 0;
      SET &DATASET END=EOF;
      *** Create the hash object at the start of the data step;
      IF _N_ = 1 THEN DO;
            *** removed macro looping, just showing 1st variable (H1);
            *** Define the object reference H1 as a hash object;
            *** We specify that the hash is ordered, which makes the object store its values in
            *** order according to the key value.  This is not necessary for our purposes,
            *** but we left it to simplify future enhancements;
            DECLARE HASH H1(ORDERED:"YES");
            *** Instantiate a copy of the just-defined hash object H1;
            H1 = _NEW_ HASH();
            *** Set the variable name held in &V1, which carries the key value;
            RC=H1.DEFINEKEY("&V1");
            *** Finalize and end the hash object definition;
            RC=H1.DEFINEDONE());
      END;
```

Next Step:
The following section of the %XC_STAT program's data step uses the REF() method to both find the current value of &V1, and to add it to the hash object if it's not already there.

SAS code:

```
      *** removed macro looping, just showing 1st variable;
      *** search for current value of each variable, and add it if not found;
      *** The ref() function does both;
      H1.REF();
```

Explanation of SAS Code:
That is all it takes to perform the pseudo-code described above! The REF() method does not need any parameters, because the value of the dataset column referred to as &V1 is automatically used as the key value.

Final Step:
The final section of the data step code writes the resulting count for each variable to a macro array used by the PROC RANK and subsequent parts of the macro program. We use call symput() to put the value of "levels" into the macro variable VAL_LEVELS. We also set Y/N flag USE_RANKS to determine whether there are any variables that have to be processed by the PROC RANK, or if that section should be skipped.

```
      IF EOF THEN DO;
      *** Initialize macro variable to NO;
         CALL SYMPUT("USE_RANKS","NO");
      *** removed macro looping, just showing 1st variable;
            ***  Our code ignores missing values;
            ***  Find the key value . in the object;
            ***  rc is always 0 if a hash object method succeeds.
          RC = H1.FIND(KEY: .);
          IF (RC = 0) THEN MISS1 = 1;
          ELSE MISS1 = 0;
            ***  objectname.num_items is an automatically defined attribute of the hash object
            ***  that holds the number of items in objectname;
            LEVELS = H1.NUM_ITEMS - MISS1;
            ***  There is a val_levels for each variable of interest.  Macro variable is named
                 val_levels1 because the example is for variable 1, using hash object h1;
```

```
            CALL SYMPUT("VAL_LEVELS1",LEVELS);
            ***  LEVELS holds the number of value levels for the current variable (the full code
                 loops over the variable list).  If ANY of them exceed &MAX_LEVELS_PERMITTED,
                 USE_RANKS is set to YES;
            ***  &DATASET_ROWS was previously given the value of the count of rows in &DATASET;
            IF LEVELS GT &MAX_LEVELS_PERMITTED AND LEVELS NE &DATASET_ROWS
            THEN CALL SYMPUT("USE_RANKS","YES");
      END;
   RUN;
```

CONCLUSION

The factor that makes the XC_STAT macro most useful is the speed with which it delivers numerous univariate statistics. This paper discussed how a combination of hash objects, PROC SUMMARY output, and both macro and dataset array programming deliver that speed. This allows modelers to spend more of their time analyzing information-rich predictors, rather than hours to generate statistics for low-value fields that would otherwise be dictated by due diligence.

HOW TO OBTAIN A COPY OF THE MACRO: %XC_STAT macro is available from the authors.

REFERENCES

Borowiak, Kenneth W, (2006). "A Hash Alternative to the PROC SQL Left Join", *NESUG 2006, Proceedings*, Northeast SAS Users Group, Inc, Paper DA07.

Dorfman, P. and Vyverman, K. (2006). "Data Step Hash Objects as Programming Tools", SUGI 31, *Proceedings*, SAS Users Group International, Paper 241-31.

Finlay, S. (2010). *Credit Scoring, Response Modelling and Insurance Rating*, New York: Palgrave MacMillan.

Hermansen, S. W. (2008). "Evaluating Predictive Models: Computing and Interpreting the c Statistic", *SAS Global Forum 2008*, Paper 143-2008.

Raimi, S. and Lund, B. (2011). "Before Logistic Modeling - A Toolkit for Identifying and Transforming Relevant Predictors", *MWSUG 2011, Proceedings*, Midwest SAS Users Group, Inc., Paper SA-03-2011.

Raimi, S. and Lund, B. (2012). "Efficiently Screening Predictor Variables for Logistic Models", *NESUG 2012, Proceedings*, Northeast SAS Users Group, Inc, Paper SA9.

Xie, L. (2009). "AUC calculation using Wilcoxon Rank Sum Test", *SAS Blog: SAS Programming for Data Mining Applications*, http://www.sas-programming.com/2009/10/auc-calculation-using-wilcoxon-rank-sum.html, October 23, 2009.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Steven Raimi
Marketing Associates, LLC
777 Woodward Ave, Suite 500
Detroit, MI, 48226
steve@raimis.com

Bruce Lund
Marketing Associates, LLC
777 Woodward Ave, Suite 500
Detroit, MI, 48226
blund@marketingassociates.com