Paper AA-04-2013

Logistics Performance Metrics using SAS® Macros

Michael C. Frick, General Motors - Retired, Warren, MI

ABSTRACT

Performance metrics for time-related processes, such as logistics delivery times, often follow an all too familiar pattern that is highly skewed to the right. In such cases, traditional central tendency statistics are of little use as they are dominated by the out-of-process events in the tail of the distribution and do not adequately describe the distribution of delivery times for in-process events. In this work, I propose a methodology that first splits the distribution into its two main components using a SAS® macro that employs piece-wise linear regression to automatically determine the break point between the main body of the distribution and the tail. Using this automated splitting mechanism, we are able to classify the performance of a large number of individual shipping lanes across three categories: (1) percent of out-of-process events, (2) average performance for in-process events against an expected standard, and (3) variation in performance for in-process events.

INTRODUCTION:

More often than not, measurements of time-related processes that physically move objects through space follow the shape of the distribution in Figure 1. We can see why by examining a rather simple task –driving across town. The majority of drive times will vary somewhat around your expected drive time (minor variation due to traffics lights, funeral processions, bad weather, etc.). Yet, no matter how often you drive it, physics dictates that it has to take a certain minimum amount of time to get there; hence the distribution is truncated on the left. Further, if you drive it often enough, there will be a number of times that it takes considerably longer than normal to get there (flat tire, road closures, etc.); hence the distribution will have an elongated tail on the right.

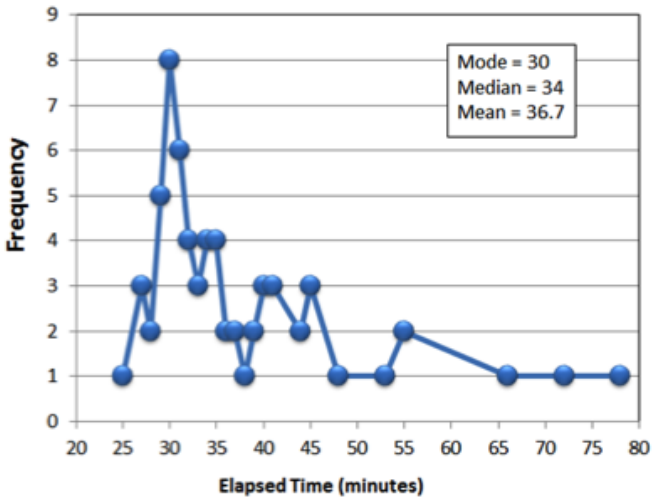

Figure 1: Distribution of Drive Times

Earlier I used the words “expected” and “normal” to quantify our expected drive times. The context is that it usually takes an approximate amount of time for you to arrive at your destination. Let’s return to Figure 1. Notice where the two statistical expected times (mean and median) would land on the distribution. Clearly they would not reflect an individual’s usual expected drive time. In fact, Figure 1 represents my own personal drive times to work over about a 14-week period. When asked (even before I did the study) I would respond that my drive takes about 30 minutes.

Given the shape of the distribution, one might be tempted to fit a theoretical distribution to the data. Figure 2 shows an Inverse Gaussian fit. Finding such a fit allows one to make inferences about how the distribution should behave. But how does one turn that information into performance metrics that can be shared with an operations group, not to mention the practical difficulties of fitting curves across a large number of shipping lanes.

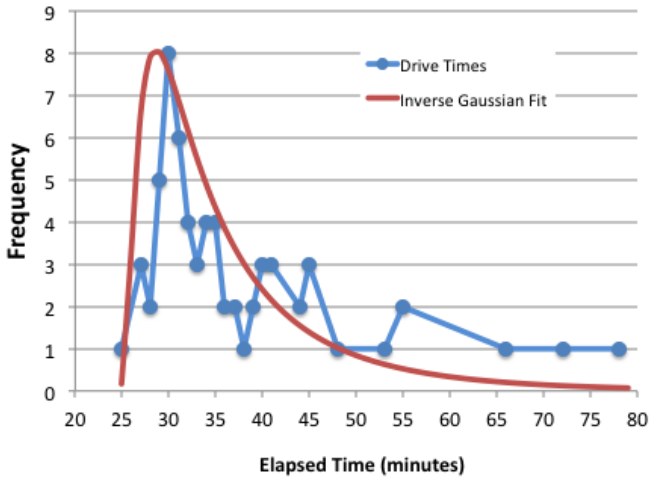

Figure 2: Inverse Gaussian Fit to Drive Times

It is the impact of the tail, which is generally outside of an operational group’s control, which causes a difference between the statistical expected and a personal expected. This difference can be quite large when manufacturing issues delay departures after units have been released for shipment. I remember having a long conversation with a supplier responsible for load-makeup and

shipment from a large manufacturing facility. He refused to believe that his reported average load makeup-time was accurate. The issue was simple, the vast majority of shipments went out as expected (which is what he sees); but a small number had been held in the yard for repairs for enough extra time to move the "statistical" average to a value that was meaningless to him – a serious problem. If you want operational folk to take a performance metric seriously, it must have a physical interpretation that is meaningful to them.

A common response to this phenomenon is to exclude long-delivery records from the measurement process, especially those that are deemed to have somehow been out-of-process. See Frick [1] for a detailed discussion of performance metrics in general as well as the pitfalls of excluding records; but, my main concern here is that simply dropping records ignores/hides failed performance on a potentially large number of customer deliveries. A better response is to break the distribution into its two key components (body and tail) and then measure.

SPLITTING THE DISTRIBUTION:

Let's return a third time to Figure 1 on the previous page. Left to their own devices, most people would have no problem splitting the tail from the main body manually at around 38 minutes, although some might make the case to split it around 45 minutes. Easy enough if we wish to analyze a small number of shipping lanes, but impossible if the number is large. In this section, I describe a SAS macro that employs piece-wise linear regression to automatically determine the break point between the main body of the distribution and the tail. The macro takes advantage of the shape of the distribution. The mode of the distribution will be close to the actual process capability (the personal expected time). Minor disruptions in delivery times will cause the shape of the distribution to be "bell-like" in the immediate vicinity of the mode. Major disruptions will result in random delivery times in the tail to the right of the "bell-like" area.

The distribution can be approximated using 4 straight lines (left panel of Figure 3): (1) one fitted to the truncated tail on the left side of the distribution, (2) one fitted to the left side of the "bell-like" body of the distribution, (3) one fitted to the right side of the "bell-like" body of the distribution, and (4) one fitted to the long tail on the right side of the distribution. As the truncated tail on the left is usually unimportant, the macro focuses on finding the best two lines on

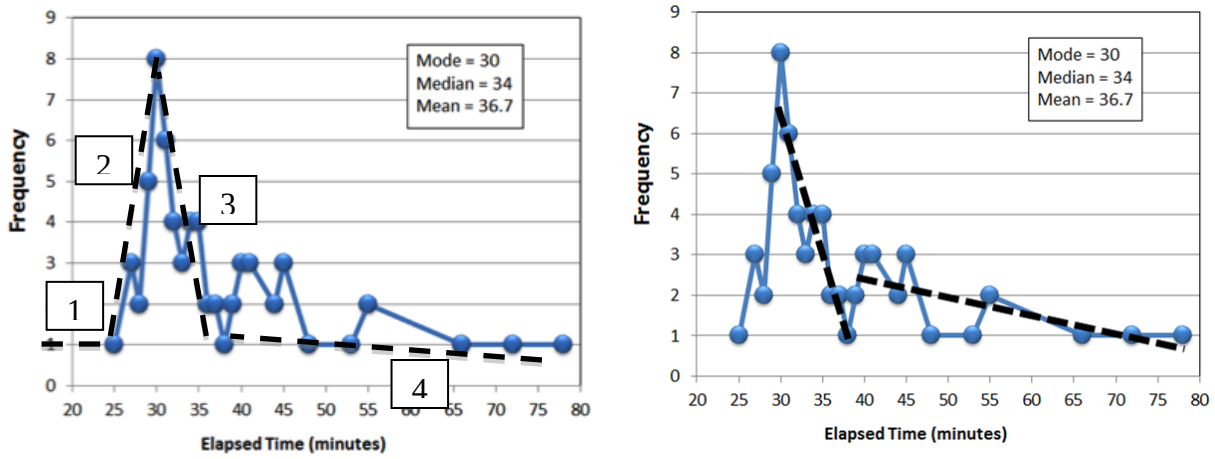

Figure 3: Linearly Approximated Distribution

the right side of the distribution using brute force and piecewise linear regression. It assumes the first line will be anchored on its left at the mode and the second line will be anchored on its right on the far right of the distribution. As the regression lines are only used to find the break point, I am not concerned about continuity at the break point. Hence, the macro simply tries all combinations of break points between the two and selects the two regression lines that minimize the sum of squared residuals. The right panel of Figure 3 shows the two best regression lines for my driving times shown originally in Figure 1. The SAS code for splitting the distribution is given in Appendix A.

PERFORMANCE METRICS:

In an earlier MWSUG paper [1], I presented 9 pitfalls (DO's/DON'Ts) when developing a set of performance metrics that were drawn from over 10 years of working experience with both executive leadership and operational groups in a large supply chain organization. In that work, I discussed the difference between desired performance, current process capability, and current actual performance. Metrics aimed at the comparison of desired and current process capability must be assigned to a non-operational group (e.g. the supply chain planning organization.) They have access to the resources necessary to define and implement initiatives to narrow the gap between current and desired capability. Metrics aimed at the comparison of current process capability and current actual performance must be assigned to an operational group. As the previously defined macro is built to help analyze actual performance, our focus here will be on operational metrics. But what makes a "good" operational performance metric?

To be considered "good", a performance metric should shine a light on potential areas of improvement. Simple pass/fail is not good enough. It should be presented in such a way that the interpretation of the metric has a physical meaning to the group. For our data in Figure 1, many people would first consider using mean, standard deviation, and perhaps the 90th percentile. [1]

So assume we report that our data in Figure 1 has a mean of 36.7, standard deviation of 10.3, and a 90th percentile of 48 and that the current process capability is set at 30. What actions might an operational group take from the above statistics? Most likely, they will see the difference between the mean and the desired performance and decide, if we can improve our process by a week, we will hit our target and improve our process variability at the same time. But we can see from Figure 1 that most of the distribution appears to be following the expected process and it is already centered on our expected time. This is a recipe for disaster. When they can't improve the average, they will look at eliminating issues observed in the tail. But by definition, things in the tail are usually one-time, unexpected events. Since the data in Figure 1 are my own personal drive times, I can tell you that no operational initiative will stop me from being stuck on a freeway that is closed or from getting a traffic citation. One logistics supervisor I used to work with called working on the items in the tail "Whack-a-mole". Management insisted that they work on the 10 worst. They did and could claim victory. But 10 others always seem to crop up and the size of the tail never really changed.

Let's revisit the data after it has been separated into its two components as shown in Figure 3. The panels of Figures 4 & 5 show the results from PROC UNIVARIATE on all 65 data points (top) and just the data points after the tail has been excluded (bottom). Notice how the three central tendency statistics and standard deviation have tightened up after the tail units have been excluded. Plus or minus 3 minutes to an average of 31 is pretty consistent performance.

Despite my whack-a-mole story above, we are still obligated to root-cause drive times in the tail. Notice the clump of drive times between 40-45 minutes in Figure 1. Luckily when I recorded my drive times I decided to keep notes. It turns out that the majority of these drive times came on days I stopped for gas. This may be actionable. In my case, I could have stopped for gas on weekends. If I were to critique my drive time performance, I would make the following observation. If I could just leave for work on time and make sure to get gas on the weekends, I could count on getting to work in very close to 30 minutes on most days (i.e. I need to focus on fixing the tail).

The SAS System

The UNIVARIATE Procedure
Variable: x

Freq: y

Moments

N	65	Sum Weights	65
Mean	36.7384615	Sum Observations	2388
Std Deviation	10.3414774	Variance	106.946154
Skewness	2.14650233	Kurtosis	5.26548097
Uncorrected SS	94576	Corrected SS	6844.55385
Coeff Variation	28.1489124	Std Error Mean	1.2827024

Basic Statistical Measures

Location		Variability	
Mean	36.73846	Std Deviation	10.34148
Median	34.00000	Variance	106.94615
Mode	30.00000	Range	53.00000
		Interquartile Range	10.00000

Figure 4: Univariate Statistics for Drive Times -- Tail Included

the

The SAS System

The UNIVARIATE Procedure
Variable: x

Freq: y

Moments

N	45	Sum Weights	45
Mean	31.5555556	Sum Observations	1420
Std Deviation	3.02681618	Variance	9.16161616
Skewness	0.19506754	Kurtosis	-0.5156267
Uncorrected SS	45212	Corrected SS	403.111111
Coeff Variation	9.59202309	Std Error Mean	0.45121111

Basic Statistical Measures

Location		Variability	
Mean	31.55556	Std Deviation	3.02682
Median	31.00000	Variance	9.16162
Mode	30.00000	Range	13.00000
		Interquartile Range	4.00000

Figure 5: Univariate Statistics for Drive Times -- Tail Excluded

[1] Some may be tempted to use medians and quartiles; but these are bad choices in a metrics environment involving non-technical leadership. Try explaining why the sum of 3 sub-process medians doesn't equal the total process median.

Assume now, that we provide an operational group with metrics based on the mean and standard deviation of the main body of the distribution and the percent of deliveries in the tail. In the case above, the group can ignore the entire body of the distribution as it is performing well. The tail actually starts at the 70th percentile, not at an arbitrarily decided 90th percentile. Given the unusually large amount of volume in the tail, it's clear that the group should dig in to root cause issues. By splitting the data into two pieces (main body and tail) we can now answer the following questions about a particular distribution lane:

1. How many deliveries appear to be following the intended process?
2. Are the units following the intended process, on average, hitting our expected process capability?
3. Are the units following the intended process experiencing too much variation?

Assuming we can define target criteria for these questions, we can determine whether this particular lane is performing well under 3 separate criteria, which in turn provides specific direction on where to shine the spotlight: average time for the main body, variation in the main body, or excessively large tail. Given an automated process for splitting distributions and target criteria for each of the three attributes above, we can now automatically classify a large number of lanes based on their performance and manage process improvement by exception.

DISCUSSION:

After spending over 10 years working with both executive leadership and operational groups on performance metrics, I have found the following to be true:

- Operational groups can sabotage, cheat, or out and out ignore performance metrics for which they see no value.
- Operational groups can move mountains to achieve performance improvements if they see value.

Hence, when developing performance metrics for an operational group, it is imperative that those metrics have the following characteristics:

- The operational group, through its normal work output, must be able to influence improvements.
- The metrics must make intuitive sense to the operational group. There has to be a physical interpretation to the metrics.
- The metrics must be presented in a form that helps point the way towards areas of potential improvement.

In the main body of this work, I have made the case for splitting logistics delivery times into two groups, those that appear to be following the intended delivery process and those that appear to have been derailed (pun intended) while in transit prior to taking measurements. I believe this gives one the best chance at presenting measurements to an operational group that will be consistent with what they see every day.

In the tail of this work (Appendix A), I presented a set of SAS Macros that would enable one to split the tail from the main body of a highly skewed distribution. To implement a measurement process across a large number of shipping lanes, one would need to beef up these macros in the following way:

1. An outer macro would be needed to loop through the macros shown in Appendix A for each measurement lane, capturing the primary statistics for each as it went along.
2. A classification process would need to be built which would join expected performance with actual performance on each lane and assign a judgmental value as to whether the performance on each of the three criteria (average & standard deviation for in-process event, quantity of out of process events) were within tolerance. The process would need to be able to rank the delivery lanes by best opportunity (combination of severity and type of bad performance and the shipping volume for each lane.)
3. The macros in Appendix A would need to be beefed up to provide a number of key checks. (e.g. low volume lanes, lanes without a clear cut mode, zero slope in the tail, etc.)
4. Depending on the number of lanes and if speed is an issue, one might be try to use the shape of the distribution to smarten up the search algorithm (i.e. once the slope of the second regression line is relatively flat do we still need to keep searching for a better fit?

APPENDIX A – TAIL SPLITTING SAS MACROS

In this section, I describe a set of three SAS macros that are able to automatically find the break point between the main body and tail of a highly skewed distribution. (See Appendix B for some helpful hints if you are not familiar with the SAS macro language). The macros assume that the distribution has the shape as shown earlier in Figure 1 and that the best place to split the data is between the mode and max of the distribution. The algorithm therefore, albeit dumb, is straightforward. It starts by placing the mode and the first data point to the right of the mode in one subset. The remaining data points to the right of this subset are placed into a second subset. Individual regressions are run on each subset of data and the combined residual sum of squares is saved for the current candidate break point. The algorithm then moves the second data point to the right of the mode from the second subset to the first subset and recalculates the regressions. This process is continued until all possible break point candidates have been examined. The break point associated with the two regressions that minimize the consolidated sum of squared residuals is selected as the best.

The code segment below shows the SAS macro LOOPER that controls the iterative search for the best break point. The parameter (DSN) is the name of the data set that contains the data points in our distribution shown in Figure 1. The PROC UNIVARIATE finds the mode of the distribution, which is subsequently stored in the macro variable MODE using the SYMPUT function. Next, a macro variable is created for every data point to the right of the mode. The names of the new variables have a prefix of "XVAL" and a numeric suffix such that the first data point to the right of the mode is named XVAL1, the second is named XVAL2, and so on. The value assigned to each new macro variable is the x-coordinate of the data point. A final new macro variable, NLOOPS, is set to the number of data points to the right of the mode. Perhaps the most interesting part of the code is the %DO - %END Loop. These innocent looking 3 lines of code control 19 separate expansions of the macro "REGIT". Each macro expansion corresponds to a unique split of the data set into two subsets. We will return to the %DO Loop as we take a quick look at macro REGIT below.

The best way to see what's going on with the macro variable creation is to look at the result of executing the "%put _user_" command, which dumps the symbol table to the SAS log (Figure 6 on next page). For each entry you can see the Macro Name, the Macro Variable Name, and the text value associated with each Macro variable. The first entry is XVAL14 with the associated text of "48". You will have to ask a SAS guru why the entries appear to have a random order. However, if you scan the list, you can see that MODE has a value of 30 as expected. XVAL1 the first data point to the right of the mode has a value 31. NLOOPS has a value of 19, as there are 19 data points to the right of the mode.

```
%macro looper(dsn);
proc delete data=main_stats;
proc sort data=&dsn.;
  by x;
proc univariate data=&dsn. noprint;
  var x;
  freq y;
  output out=main_stats mode=mode;
data _null_;
  set main_stats;
  call symput('mode',trim(left(put(mode,4.))));
data _null_;
  set &dsn. end=endf;
  retain n 0;
  if x>&mode. then
     do;
       n=n+1;
       call symput('xval' || trim(left(put(n,4.))),trim(left(put(x,6.))));
     end;
  if endf then call symput('nloops',trim(left(put(n,4.))));
run;
%put _user_;
%do nn=1 %to &nloops;
   %regit(&dsn.,&nn.,&mode.,&&xval&nn..);
  %end;
run;
proc sort data=mdls;
  by descending rsq;
options linesize=100;
proc print data=mdls;
  format slope1 sse1 rsq1 sstot1 slope2 sse2 rsq2 sstot2 sse rsq sstot 5.2;
run;
%mend;
```

```
MLOGIC(LOOPER):  %PUT _user_
LOOPER XVAL14 48
LOOPER XVAL15 53
LOOPER DSN xy
LOOPER XVAL12 44
LOOPER XVAL1 31
LOOPER XVAL13 45
LOOPER XVAL2 32
LOOPER XVAL10 40
LOOPER XVAL3 33
LOOPER XVAL11 41
LOOPER XVAL4 34
LOOPER XVAL5 35
LOOPER XVAL6 36
LOOPER XVAL7 37
LOOPER XVAL8 38
LOOPER NLOOPS 19
LOOPER XVAL9 39
LOOPER XVAL18 72
LOOPER MODE 30
LOOPER XVAL19 78
LOOPER XVAL16 55
LOOPER XVAL17 66
SYMBOLGEN:  Macro variable NLOOPS resolves to 19
MLOGIC(LOOPER):  %DO loop beginning; index variable NN; start value
MLOGIC(REGIT):  Beginning execution.
```

Figure 6: Symbol Table Dump

The MACRO REGIT (code segment below) expects 4 parameters. The first, DSN, is just the name of the data set. The second is an iteration counter. The third, X1, is set to the MODE. The fourth, X2, is the current test break point. The first PROC REG operates on data between X1 and X2 (i.e. from the mode to the current break point candidate). The second PROC REG operates on all data points to the right of the current break point candidate. The rest of the macro saves the results from the two regressions, computes the consolidated sum of squared residuals, and then saves the results of this model with those from previous macro calls in a data set named MDLS for later analysis.

```
%macro regit(dsn,loop,x1,x2);
proc delete data=est1 est2 est;
proc reg data=&dsn. noprint
  outest=est1(keep=x _rsq_ _sse_
  rename=(_rsq_=RSQ1 _sse_=SSE1 x=slope1));
  model y=x / rsquare sse;
  where &x1.<=x<=&x2.;
proc reg data=&dsn. noprint
  outest=est2(keep=x _rsq_ _sse_
  rename=(_rsq_=RSQ2 _sse_=SSE2 x=slope2));
  model y=x / rsquare sse;
  where x>&x2.;
data est1;
  set est1;
  SStot1=SSE1/(1-RSQ1);
data est2;
  set est2;
  SStot2=SSE2/(1-RSQ2);
data est;
  merge est1 est2;
data est;
  set est;
  length modelid $ 20;
  modelid=compress("Model: &loop.");
  x1=&x1.;
  x2=&x2.;
  SSE=SSE1+SSE2;
  SStot=SStot1+SStot2;
  RSQ=1-(SSE/SStot);
proc append base=mdls data=est;
  run;
%mend;
```

Now let's return to the %DO Loop shown previously in the macro LOOPER.

```
%do nn=1 %to &nloops;
   %regit(&dsn.,&nn.,&mode.,&&xval&nn..);
  %end;
```

The macro call to expand REGIT can admittedly be a bit overwhelming as it makes use of a SAS macro language construct called a MACRO array. MACRO arrays are described in detail in Carpenter [2]. The 4th time we execute the %DO Loop, we are asking the SAS Macro processor to expand REGIT with the current value of the macro variable DSN (xy), the current value of the macro variable NN (4), the current value of the macro variable MODE (30), and the current value of the macro variable XVAL4 (34). Notice how in the unexpanded version of the macro, page 5, the PROC REGs point to the macro variable DSN while in the expanded version of the macro (Figure 7) they read "proc reg data=xy". The expansion of a SAS macro is nothing more exotic than an insertion of the code in-line at the point of the expansion request coupled with text substitution. Again, notice how the WHERE clause in the PROC REG looks at the macro variables X1[2] and X2 when unexpanded but looks at the values of X1 (30) and X2 (34) for a particular expansion. We can tie this back to the symbol table shown in Figure 6 where you can see that XVAL4 does indeed have a value of 34.

```
MLOGIC(REGIT):  Ending execution.
MPRINT(LOOPER):  ;
MLOGIC(LOOPER):  %DO loop index variable NN is now 4; loop will iterate again.
MLOGIC(REGIT):  Beginning execution.
SYMBOLGEN:  Macro variable DSN resolves to xy
SYMBOLGEN:  Macro variable NN resolves to 4
SYMBOLGEN:  Macro variable MODE resolves to 30
SYMBOLGEN:  && resolves to &.
SYMBOLGEN:  Macro variable NN resolves to 4
SYMBOLGEN:  Macro variable XVAL4 resolves to 34
MLOGIC(REGIT):  Parameter DSN has value xy
MLOGIC(REGIT):  Parameter LOOP has value 4
MLOGIC(REGIT):  Parameter X1 has value 30
MLOGIC(REGIT):  Parameter X2 has value 34
MPRINT(REGIT):   proc delete data=est1 est2 est;

NOTE: Deleting WORK.EST1 (memtype=DATA).
NOTE: Deleting WORK.EST2 (memtype=DATA).
NOTE: Deleting WORK.EST (memtype=DATA).
NOTE: PROCEDURE DELETE used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds


SYMBOLGEN:  Macro variable DSN resolves to xy
MPRINT(REGIT):   proc reg data=xy noprint outest=est1(keep=x _rsq_ _sse_ rename=(_rs
MPRINT(REGIT):   model y=x / rsquare sse;
SYMBOLGEN:  Macro variable X1 resolves to 30
SYMBOLGEN:  Macro variable X2 resolves to 34
MPRINT(REGIT):   where 30<=x<=34;
NOTE: The data set WORK.EST1 has 1 observations and 3 variables.
NOTE: PROCEDURE REG used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds
```

Figure 7: 4th Expansion of SAS Macro REGIT

The third macro, ANALYZE, creates summary statistics for the two subsets that correspond to the best break point. First, it applies a flag to observations in the tail of the distribution and then computes their frequency. Then it calculates univariate statistics for just those observations considered to be in the main body of the distribution. ANALYZE obviously just scratches the surface of functionality one might want to see here. See the Discussion section in the main body of the text for potential enhancements/extension to the macros presented in this appendix.

```
%macro analyze(dsn,mdls);
  proc delete data=x;
  data x;
   set &mdls.;
   if _n_=1;
   keep x1 x2;
  data &dsn.;
   set &dsn.;
if _n_=1 then set x;
   if x>x2 then tail=1;
   else
      tail=0;
 proc print data=&dsn.;
 proc freq data=&dsn.;
  tables tail;
  weight y;
 proc univariate data=&dsn.;
 var x;
  where x<=x2;
  freq y;
run;
%mend;
```

[2] Parameters for SAS macro expansion are positional in this example. So in the macro call, MODE is aligned with X1 as both are in the third position.

APPENDIX B – SAS MACRO LANGUAGE OVERVIEW

This section provides a quick overview of the SAS macro language constructs used in Appendix A. Its primary purpose is to allow a person unfamiliar with SAS macros to be able to follow the above code. To find out everything you might ever have wanted to know about SAS macros, Carpenter's classic reference [2]. Prior to looking at some specific code, a quick note about SAS macro processing. SAS macros are NOT equivalent to Excel [3] macros. Excel macros are chunks of code that are compiled and then executed each time they are invoked. SAS macros are chunks of code passed through a preprocessor prior to compilation and then executed each time they are invoked. That is, an Excel macro is the identical code executed over and over. It can just do it on different sets of data. The code in a SAS macro can be changed on the fly to operate completely differently.

The primary building block of the SAS macro language is the macro variable. The naming convention for defining a new macro variable follows the normal rules for defining a variable in a data step; however, when you reference one in a code segment it must have an "&" as a prefix and should have a "." as a suffix. SAS macro variables are different than other SAS variables in that the only value they can take on are text strings. SAS macro variables are normally defined and assigned a value in one of three ways: (1) a %LET statement (which can occur in open SAS code outside of a macro), (2) an input parameter in the SAS macro definition, (3) a %SYMPUT function inside of a SAS macro. SAS macro variables are generally only visible to SAS code in the macro in which they are defined. Again, Carpenter [2] is a great source for details on the scope of SAS macro variables.

Figure 8 shows a typical macro definition. I have purposely cut and pasted the code segment as a bitmap from the SAS editor rather than pasting it as text. I feel it is important to see the visual queues the editor provides when working with SAS macros. The macro definition starts with the keyword "%MACRO" and is finalized with the keyword "%MEND". Notice that both keywords are shown as a dark blue. The name of the macro is "AgeSubsetter". There are four parameters that can be passed to the macro at the time it is invoked: "DataSetName", "AgeLower", "AgeUpper", and "PrintIt". Notice how the macro variables defined as parameters in the macro definition statement are sprinkled throughout the body of the macrocode segment (shown in an aqua-green color). We will come back to this thought in a minute.

```
%macro AgeSubSetter(DataSetName,AgeLower,AgeUpper,PrintIt);
data tt;
  set &DataSetName.;
  if &AgeLower. <= age <= &AgeUpper.;
run;
%if &PrintIt. = "Y" %then
%do;
proc print data=tt(obs=10);
%end;
run;
%mend;
run;
%AgeSubSetter(xx,16,35,"Y");
run;
%AgeSubSetter(xx,16,35,"N");
run;
```

Figure 8: SAS Macro AgeSubSetter

Since macros are not executed at the time they are defined, we need a mechanism to execute them - which are the macro invocation statements shown at the bottom of Figure 8. The invocation starts with a leading % sign followed by the name of macro and parameters we wish to pass to the macro. The parameters specified in the macro invocation can be any SAS expression that results in a text string. Figure 9 shows the expansion of the macro after it has been invoked with the first invocation in Figure 8. Notice how the strings passed as parameters in Figure 8 are substituted for their corresponding macro variable names from the macro definition to produce the executable block of SAS code shown in Figure 9. This is quite literally a gigantic "COPY and PASTE" and then "REPLACE ALL" type of process.

```
81   %AgeSubSetter(xx,16,35,"Y");
MLOGIC(AGESUBSETTER):  Beginning execution.
MLOGIC(AGESUBSETTER):  Parameter DATASETNAME has value xx
MLOGIC(AGESUBSETTER):  Parameter AGELOWER has value 16
MLOGIC(AGESUBSETTER):  Parameter AGEUPPER has value 35
MLOGIC(AGESUBSETTER):  Parameter PRINTIT has value "Y"
MPRINT(AGESUBSETTER):   data tt;
SYMBOLGEN:  Macro variable DATASETNAME resolves to xx
MPRINT(AGESUBSETTER):   set xx;
SYMBOLGEN:  Macro variable AGELOWER resolves to 16
SYMBOLGEN:  Macro variable AGEUPPER resolves to 35
MPRINT(AGESUBSETTER):   if 16 <= age <= 35;
MPRINT(AGESUBSETTER):   run;

NOTE: There were 46 observations read from the data set WORK.XX.
NOTE: The data set WORK.TT has 20 observations and 1 variables.
NOTE: DATA statement used (Total process time):
      real time           0.02 seconds
      cpu time            0.03 seconds


SYMBOLGEN:  Macro variable PRINTIT resolves to "Y"
MLOGIC(AGESUBSETTER):  %IF condition &PrintIt. = "Y" is TRUE
MPRINT(AGESUBSETTER):   proc print data=tt(obs=10);
MPRINT(AGESUBSETTER):   run;

NOTE: There were 10 observations read from the data set WORK.TT.
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

Figure 9: Expansion of Macro AgeSubsetter # 1

The beauty is you get to decide at execution time (perhaps in the middle of a SAS job) what copy, paste, and replace to process. Refer back to the macro definition in Figure 8. The %IF-%DO-%END construct is a conditional execution statement that is evaluated by the macro processor prior to execution of the SAS code. If the PRINTIT parameter is passed a value of "Y" then PROC PRINT is included in the macro expansion and executed. If it is set to anything else (i.e. "N"), the PROC PRINT is never even shown to the compiler, much less executed. Compare the expanded code segments shown previously in Figure 9 to the one in Figure 10. The choice to include/exclude the PROC PRINT is made by the preprocessor, not the compiler. Think of the power. I can sprinkle all of my code with key debugging aids that are normally turned off but can be turned on as needed. Assuming you have trapped an error code (%SYSERR) after a job/routine has been run, you could automatically re-execute the module with your debugging turned on – a great time saving for large jobs run overnight.

```
MLOGIC(AGESUBSETTER):  Ending execution.
82   run;
83   %AgeSubSetter(xx,16,35,"N");
MLOGIC(AGESUBSETTER):  Beginning execution.
MLOGIC(AGESUBSETTER):  Parameter DATASETNAME has value xx
MLOGIC(AGESUBSETTER):  Parameter AGELOWER has value 16
MLOGIC(AGESUBSETTER):  Parameter AGEUPPER has value 35
MLOGIC(AGESUBSETTER):  Parameter PRINTIT has value "N"
MPRINT(AGESUBSETTER):   data tt;
SYMBOLGEN:  Macro variable DATASETNAME resolves to xx
MPRINT(AGESUBSETTER):   set xx;
SYMBOLGEN:  Macro variable AGELOWER resolves to 16
SYMBOLGEN:  Macro variable AGEUPPER resolves to 35
MPRINT(AGESUBSETTER):   if 16 <= age <= 35;
MPRINT(AGESUBSETTER):   run;

NOTE: There were 46 observations read from the data set WORK.XX.
NOTE: The data set WORK.TT has 20 observations and 1 variables.
NOTE: DATA statement used (Total process time):
      real time           0.02 seconds
      cpu time            0.01 seconds


SYMBOLGEN:  Macro variable PRINTIT resolves to "N"
MLOGIC(AGESUBSETTER):  %IF condition &PrintIt. = "Y" is FALSE
MPRINT(AGESUBSETTER):   run;
MLOGIC(AGESUBSETTER):  Ending execution.
84   run;
```

Figure 10: Expansion of Macro AgeSubSetter #2

The final concept I would like to discuss here is that of a SAS macro array. Technically, the SAS macro language does not have an array capability. But, because, at its heart, it is a text substitution facility, you can fake it out. Let's assume you have defined 3 macro variables Age1, Age2, and Age3 and Age1 has been assigned the value of "16". Then the statement

x = &Age1.; would resolve to x = 16;

When the macro pre-processor sees the "&" sign, it knows it needs to do a text substation. But what if I have a fourth macro variable defined, K., which has been assigned the value of "1" and changed the assignment statement to

x =&&Age&K..

The doubled "&&" sign causes the macro processor to do two successive text substitutions. The first replaces the "&K" with its value of 1, yielding &Age1 while the second replaces &Age1 with its value of 16. If you couple this facility with a macro %Do %End loop using K as the macro loop variable, you in effect have an array capability.

REFERENCES

[1] Frick, Michael 2009, DO's and DON'Ts of Generating Performance Metrics, Proceedings of the Twentieth Midwest SAS Users Group Conference, paper T04-2009.

[2] Carpenter, Art 2004. Carpenter's Complete Guide to the SAS® Macro Language, Second Edition. Cary, NC: SAS Institute Inc.

[3] Excel (Part of Microsoft Office Professional Edition) [computer program]. Microsoft; 2010.

Contact Information

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

Name: Michael C. Frick

Enterprise: General Motors, Retired

Address: 30238 Underwood Drive
City, State ZIP: Warren, MI 48092
Work Phone: 586-573-0977
Fax: N/A
E-mail: mcfdaf001@yahoo.com
Web: N/A