Paper BtB-03

Data in the Doughnut Hole: Using SAS® to Report on What is NOT There

Sarah Woodruff, Westat, Rockville, MD

Abstract

Typically the request for a table, graph or other report concerns data that currently exists and needs to be explored. However, there is often a need to examine data that is expected to be present but currently is not. Yet how can we report on something that is not there? This paper describes and explores ways to create simulations of what is expected and then match those with data that is actually present.

Making comparisons between what has been anticipated and what actually exists then opens up the ability to report on what is not there. This paper also explores how to test the differences between the groups based on a variety of conditions and how to determine what is legitimately absent. Reporting on the absent data is then described, including how to check that everything which is expected is actually present and be able to report that as well. DATA step techniques are combined with PROC SQL and PROC REPORT to create a step-wise process which can be easily modified to fit most any set of specifications. Let's dive into the sweet task of reporting from inside the doughnut hole!

Keywords: shell data, PROC SQL, missing records, PROC REPORT

Introduction

One of the best reasons to use SAS is its ability to efficiently parse data and create almost infinitely customizable reports. As programmers, we are routinely asked to explore and track existing data, no matter what line of work we are in or our particular industry focus. However, across various applications or areas of research, we may be asked to examine what is NOT present in our data, whether that might be shipments that were not received or clinical visits that did not take place or vehicle options that were not selected. Such information creates a "doughnut hole" in research or reporting; we are surrounded by data that does exist, yet the data that is missing is still of interest to our project.

Since a large portion of my project work focuses on clinical trials, I will be using some clinical research scenarios as the basis for the examples in this paper. Despite this, the ideas behind the examples could be generalized or customized for nearly any scenario in which missing data is relevant to a project. Though I predominantly focus on DATA step coding, I am have been endeavoring to incorporate PROC SQL as both approaches have their strengths. To make this paper useful to as many programmers as possible, I will demonstrate the use of each throughout.

The Ingredients

For the purposes of this paper, I am using a hypothetical project called StudyMedicine. Three existing study data tables are relevant here:

1. Planned Events: the planned schedule of study events and forms

SubjectID	VisitName	FormName	EarlyDate	ExpectedDate	LateDate	ETC.
10002	WEEK01	Form01	10/12/2012	10/15/2012	10/18/2012	<…>
10002	WEEK01	Form01b	10/12/2012	10/15/2012	10/18/2012	<…>
10002	WEEK01	Form05	10/12/2012	10/15/2012	10/18/2012	<…>
10002	WEEK01	Form12	10/12/2012	10/15/2012	10/18/2012	<…>
10002	WEEK02	Form02	10/19/2012	10/22/2013	10/25/2012	<…>
10002	WEEK02	Form04	10/19/2012	10/22/2013	10/25/2012	<…>
10002	WEEK02	Form05	10/19/2012	10/22/2013	10/25/2012	<…>

.
.
And so on for each subject with the individual's respective visits and forms.

2. Study Visits: the study events and their forms that have actually occurred and been recorded

SubjectID	VisitName	FormName	VisitDate	ETC.
10002	WEEK01	Form01	10/16/2012	<…>
10002	WEEK01	Form01b	10/16/2012	<…>
10002	WEEK01	Form12	10/16/2012	<…>
10002	WEEK02	Form02	10/21/2012	<…>
10002	WEEK02	Form04	10/21/2012	<…>

.
.
And so on for each subject with the individual's respective entered visits and forms.

3. Demographics: information about the subjects in the study

SubjectID	Gender	Age	Ethnicity	ETC.
10002	Female	32	Hispanic	<…>
10003	Male	45	Non-Hispanic	<…>
10004	Female	26	Non-Hispanic	<…>

.
.
And so on for each subject.

The Easiest Hole to Find

Many types of clinical studies will include a planned events table in their relational database. At its simplest, this table would list all the visits or "events" that are expected to happen for each subject involved in the study. When an expected schedule accompanies these planned events, then those dates would be included as well. However, research, much like other aspects of life, often does not go as planned. Study visits may be missed or may happen out of order or may not happen within the planned timeframes.

If a planned events table or its equivalent for something non-clinical exists, then the isolation of data which does not exist becomes relatively straightforward. Whether utilizing DATA step code or PROC SQL, the first step is to determine what makes a record unique. For the purposes of this example, the variables SubjectID, VisitName and FormName will create a unique record in the StudyVisits table because the hypothetical project, StudyMedicine, has a set of unique study visits with forms for each participant.

In a DATA step approach, both the table containing the planned events and the table containing the study visit data would need to be sorted by the relevant variables.

```
PROC SORT data = StudyMedicine.PlannedEvents out = PE_UniqueSort NODUPKEY ;
      BY SubjectID VisitName FormName ;
RUN ;

PROC SORT data = StudyMedicine.StudyVisits out = SV_UniqueSort NODUPKEY ;
      BY SubjectID VisitName FormName ;
RUN ;
```

Once both tables are sorted, a DATA step with merge logic can be used to isolate the records that are in the PlannedEvents table but not the StudyVisits table.

```
DATA AbsentStudyVisits ;
      MERGE PE_UniqueSort ( in = PE ) SV_UniqueSort ( in = SV ) ;
            BY SubjectID VisitName FormName ;
      IF PE and not SV ;
RUN ;
```

The equivalent process in PROC SQL can be accomplished in one step as the two tables do not need to be sorted prior to isolating the absent records of interest. In this approach, the two tables are compared on the basis of the SubjectID, VisitName and FormName to specify a unique record; when the combination of those three variables is not present in the StudyVisit table, then the records are output.

```
PROC SQL ;
      CREATE TABLE AbsentStudyVisits as SELECT DISTINCT
      PE.* FROM StudyMedicine.PlannedEvents as PE
            LEFT JOIN
      StudyMedicine.StudyVisits as SV ON
            PE.SubjectID = SV.SubjectID AND
            PE.VisitName = SV.VisitName AND
            PE.FormName = SV.FormName
      WHERE SV.SubjectID is null OR
            SV.VisitName is null OR
            SV.FormName is null;
QUIT ;
```

Note that a LEFT JOIN was used in the above code because the PlannedEvents table was referenced first and the goal was to isolate records present there, but not in the StudyVisits table. Had the StudyVisits table been referenced first, then a RIGHT JOIN would need to be used to isolate the same set of records.

Verifying Data Expectations

In this scenario, it would be reasonable to assume that, since the StudyMedicine project calls for a unique collection of visits with forms for each participant, there will never be more than one record in the StudyVisits table with identical values for SubjectID, VisitName and FormName. However, merging often goes awry when based on untested assumptions and it would be a good idea to verify that this is indeed how the data was entered. With a bit of planning, human error can be easily detected and then investigated.

```
PROC SORT DATA = StudyMedicine.StudyVisits
      /* SV_NotUnique will contain any duplicates based on the combination of
      BY variables. */
      DUPOUT = SV_NotUnique
      /* The use of NODUPKEY will focus on the two variables that create the
      unique record. */
      NODUPKEY
      OUT = SV_Unique ;
      BY SubjectID VisitName FormName ;
RUN ;
```

Building your own Hole

Often, we do not have the advantage of a planned events table or its equivalent in various project work. In such cases, it becomes necessary to build the relevant table, and SAS provides an easy way to do this through the creation of shell tables. Since these shells are being constructed *de novo*, there is a great deal of flexibility to specify what will create a unique record of interest.

For this example, let us say that are SubjectID values are generated as the StudyMedicine project proceeds and that there are specific forms to be completed at each study visit. However, the forms required at each visit vary. A shell for each visit can easily be created.

```
DATA Shell_WEEK01 ;
      /* LENGTH and FORMAT values should match those used in the actual data */
```

```
        LENGTH VisitName $ 20 FormName $ 8 ;
        FORMAT VisitName $20. FormName $8. ;
        /* Created values should also match those used in the actual data */
        VisitName = 'WEEK01' ;
        DO FormName = 'Form01' , 'Form01b' , 'Form05' , 'Form12' ;
              OUTPUT ;
        END ;
RUN ;

DATA Shell_WEEK02 ;
        /* LENGTH and FORMAT values should match those used in the actual data */
        LENGTH VisitName $ 20 FormName $ 8 ;
        FORMAT VisitName $20. FormName $8. ;
        /* Created values should also match those used in the actual data */
        VisitName = 'WEEK02' ;
        DO FormName = 'Form02' , 'Form04' , 'Form05' ;
              OUTPUT ;
        END ;
RUN ;
.
.
.
```

Continue creating tables to cover each instance of a visit that is part of the study. If certain forms are expected for certain subjects, but not for others at a given visit, that conditional logic will be handled later in the process. For now, be sure to include any form that is of interest for a given visit.

Once all the visits within the study are accounted for in their own shells, the shells need to be combined in order to create a full listing of the visits with their respective forms that are expected during the study.

```
DATA Shell_AllWEEKS;
        SET Shell_WEEK01 Shell_WEEK02 <...> ;
RUN ;
```

With the full set of visits and forms created, the next step is to combine this information with the SubjectIDs so that a comprehensive list of forms can be created for each study participant. Since this is fundamentally a Cartesian product, PROC SQL is the best way to create this data set.

```
PROC SQL;
        CREATE TABLE FormsSubjects AS
              SELECT Shell_AllWEEKS.VisitName, Shell_AllWEEKS.FormName,
                    StudyVisits.SubjectID
        FROM StudyMedicine.StudyVisits, Shell_AllWEEKS ;
QUIT ;
```

Now that the study participants have been aligned with all their visits and requisite expected forms, this created data can be conditionally edited if needed. For example, Form04 may only be required at the WEEK02 visit if the participant is female or over a certain age or on a particular treatment. The relevant variables specifying these conditions need to be isolated from the study data and combined with the FormsSubject data set. Then combinations of SubjectID with VisitName and FormName which are not relevant can be deleted.

In a DATA step, the data sets to be merged would need to be sorted by the variables they had in common; for this example, that will only be SubjectID as we are going to look at bringing in demographic information. The process would be as follows:

```
DATA Shell_EditedWEEKS ( DROP = Gender ) ;
      MERGE Shell_AllWEEKS ( in = ALL )
            StudyMedicine.Demographics ( in = DEMO KEEP = SubjectID Gender ) ;
      BY SubjectID ;
      IF ALL ;
      /* Include other conditions as needed */
      IF Gender not in ('Female') and VisitName in ('WEEK02')
         and FormName in ('Form04') THEN DELETE ;
RUN ;
```

As always, PROC SQL is an alternative. Here the addition of the necessary demographic variables to remove the unneeded records is separate from the deletion of those records. For me, this aids in the readability of the code, but if the data sets were large enough, it could be preferable due to performance concerns to combine these two steps.

```
PROC SQL ;
      CREATE TABLE Shell_EditedWEEKS as SELECT
      ALL.* FROM Shell_AllWEEKS as ALL
            LEFT JOIN
      DEMO.* FROM StudyMedicine.Demographics as DEMO ON
            ALL.SubjectID = DEMO.SubjectID ;
      DELETE FROM Shell_EditedWEEKS
      WHERE Gender not in ('Female') and VisitName in ('WEEK02')
            and FormName in ('Form04') ;
QUIT ;
```

Setting Conditions for Being in the Hole

Once we have an established list of what is expected and can compare that to records that are actually present, we may want to impose additional conditions on what qualifies as not being there. For example, in the project StudyMedicine, there could be an established date for each visit, but following that, there could be a window of time in which the forms for the visit could be entered. Therefore, data should not be considered to be in that hole until that window has closed. We need to establish those parameters so they can be applied to the combination of expected and existing data that has already been established.

For the purposes of this example, the dates of the study visits are being established based on the WEEK01 visit as that is the entry point into the study. Once a study visit happens, a three-day window exists to get the forms entered. If the form is not present after that time, it has fallen into the doughnut hole and becomes of interest to track.

Note in this step, baseline data for WEEK01 is being taken from the StudyVisits table. Since dates for those visits already exist, the VisitDate and LateDate values are generated directly. Values for the remaining visits need to be created so this step combines data that is currently present with what is expected to accomplish that goal. Since date values are such a relevant part of this data selection, it is beneficial to format them so they are easily understandable. My preference is for a very full format, like MMDDYY10., but that can be customized to project needs. The final version of ScheduleOfDates should only contain the variables that will be needed going forward. This helps to minimize the size of data sets while also making it easier to track what is being used by the program.

```
DATA ScheduleOfDates ( KEEP = SubjectID VisitName VisitDate LateDate ) ;
      SET StudyMedicine.StudyVisits ( KEEP = SubjectID VisitName EntryDate ) ;
      WHERE VisitName in ('WEEK01') ;
            IF VisitName in ('WEEK01') then VisitDate = EntryDate ;
            IF VisitName in ('WEEK01') then LateDate = VisitDate + 3 ;
            VisitName = 'WEEK02' ; VisitDate = EntryDate + 7 ; LateDate =
```

```
                VisitDate + 3 ; OUTPUT ;
            VisitName = 'WEEK04' ; VisitDate = EntryDate + 21 ; LateDate =
                VisitDate + 3; OUTPUT ;
            /* Continue to add values for all relevant visits */
        FORMAT VisitDate LateDate mmddyy10. ;
        PROC SORT ; BY SubjectID VisitName ;
RUN ;
```

Note that ScheduleOfDates is ultimately sorted by SubjectID and VisitName; this step is important because this data needs to be added to Shell_EditedWeeks so that the relevant dates can be added to the records for each type of visit.

```
DATA Shell_Weeks_Dates ;
    MERGE Shell_EditedWEEKS ScheduleofDates ;
    BY SubjectID VisitName ;
RUN ;
```

However, remember that this sort is only truly required if Shell_Weeks_Dates is created with a DATA step merge. Using PROC SQL to create this table means that sorting is not necessary, although to make the process smooth, specific identification of variables is needed. Since SubjectID and VisitName are on both data sets, if *all* the variables from ScheduleOfDates were selected, a message would go to the log specifying that those two variables were already in the data set being created. In order to avoid this message, only the two additional variables from ScheduleOfDates are selected for inclusion, though the linkage between ScheduleOfDates and Shell_EditedWEEKS proceeds on both SubjectID and VisitName.

```
PROC SQL;
    CREATE TABLE Shell_Weeks_Dates AS
        SELECT Shell_EditedWEEKS.*, ScheduleofDates.VisitDate ,
            ScheduleofDates.LateDate
    FROM Shell_EditedWEEKS, ScheduleofDates
        WHERE Shell_EditedWEEKS.SubjectID = ScheduleofDates.SubjectID
        AND   Shell_EditedWEEKS.VisitName = ScheduleofDates.VisitName ;
QUIT ;
```

This extended shell now needs to be compared to the data available in StudyVisits. Since this comparison is now specific not only to SubjectID and VisitName, but also to FormName, all three variables need to be part of the next linkage step. If proceeding with an additional PROC SQL, the code will look similar to that which created AbsentStudyVisits above, but we are now interested in cases where any one of the three variables remains blank.

The use of the left join to create AbsentStudyVisitsForms below works as it did for the creation of AbsentStudyVisits; since Shell_Weeks_Dates contains all the records and StudyVisits contains those which have actually occurred and been entered, we want to keep records from Shell_Weeks_Dates based on the further conditions provided in the code. Thus, we are working to isolate only the records that exist within the doughnut hole specific to the conditions of the project.

```
PROC SQL ;
    CREATE TABLE AbsentStudyVisitsForms as SELECT
    PE.* FROM Shell_Weeks_Dates as PE
        LEFT JOIN
    StudyMedicine.StudyVisits as SV ON
        PE.SubjectID = SV.SubjectID AND
        PE.VisitName = SV.VisitName AND
        PE.FormName = SV.FormName
```

```
        WHERE SV.SubjectID is null OR
              SV.VisitName is null OR
              SV.FormName is null ;
QUIT ;
```

DATA step code to bring these tables together would require sorting and given the source, it would be prudent to create a separate sorted table with the records from the StudyVisits table.

```
PROC SORT DATA = StudyMedicine.StudyVisits OUT = StudyVisits_Sort ;
        BY SubjectID VisitName FormName ;
RUN ;
```

Particularly if only a limited number of variables are needed from the StudyVisit data, it would be good to include a KEEP statement with the creation of the sorted data set. Alternatively, if there are only a few variables which are *not* needed, then a DROP statement would be useful to get rid of any data that will not be used going forward.

If Shell_Weeks_Dates has been created using PROC SQL then it will need a similar sort performed. If it has been created using a DATA step, then the sort can be incorporated into that piece of code to make the process more efficient.

With both sorted data sets available, an additional DATA step can be used to merge the two, isolating only the records of interest in our hole.

```
DATA AbsentStudyVisitsForms ;
        MERGE Shell_Weeks_Dates ( in = PE ) StudyVisits_Sort ( in = SV ) ;
        BY SubjectID VisitName FormName ;
        IF PE and not SV ;
RUN ;
```

No matter which method is preferable, a data set complete with the date parameters for each visit is now available so it becomes possible to further identify which records genuinely belong in the hole and which do not yet qualify. It is useful to create an indicator for being in this hole; in addition, it may be useful to calculate certain metrics about that status for a record like how many days have passed since a form was expected. In the code below, InTheHole indicates whether a record is there or not and DaysInHole calculates how long that has been the case. SAS makes this work particularly easy for us through the use of the TODAY() function which allows us to make calculations utilizing today's data automatically rather than having to hard code a value.

```
DATA Absent ;
        SET AbsentStudyVisitsForms ;

        IF TODAY() gt LateDate THEN InTheHole = 1 ;
        ELSE InTheHole = 0 ;

        IF InTheHole = 1 THEN DaysInHole = TODAY() - LateDate ;
        ELSE DaysInHole = 0 ;
RUN ;
```

The creation of an indicator variable at this stage of programming would also allow you to explore records which are on the precipice of falling into the doughnut hole, those which are not currently present but which also do not meet the conditions of being truly delinquent or absent. In some cases, these may merit a frequency listing or report of their own, but for now the focus remains on those which are truly in our data hole.

By the very nature of this process, once data shows up, it is no longer in the doughnut hole. Typically when I have

done this type of reporting, there have been others who needed to look at what forms or visits were absent in a given period of time so I have saved a permanent data set to create an archive of the occurrences. If comparison between reporting periods is necessary – for example, if you want to examine how long it is typically taking for records to make it out of the hole – then you will need to put more thought into how you want to maintain your archive. An appropriately structured running data set where records are appended might be appropriate or it may be easier to save data sets based on the given reporting period and compare them as separate entities.

Reporting from the Hole

With all conditions having been applied to both the expected and existing data, now it is time to actually create a report out of the records. Given that the point of this process is to find records which do not exist, the hope is that there will never be anything to report and thus that all expected records are actually in place. Therefore we need a PROC REPORT step which provides options; one option will create a listing of the records that are absent and another option that provides the message that no records were found to be absent. A straightforward way to do this is to create a simple macro to wrap around the reporting code in order to test whether there are records to list or not.

To make the report as informative as possible, it is beneficial to start with both a title and a footnote. In this case, the title provides a description of the report while the footnote provides the name and location of the program generating the report along with the date is was run. The use of DATE() combined with %SYSFUNC allows us to automatically populate with today's date and I choose to match my date format with the date format used in the program.

```
TITLE1 h=3 "StudyMedicine Absent Forms";
FOOTNOTE1 h=0.7 j=l "PROGRAM: \\Drive\Programs\StudyMedicine_Absent_Forms.sas
                     rundate: %sysfunc(DATE(), mmddyy10.)";
```

With the descriptive portions in place, then build the macro that will create the body of the report. Start by opening the macro call and establishing where the report will be saved, the file name under which it will be saved, and the format of the document. For this example, I will show the use of RTF output, but this could easily be modified to accommodate PDF output or further customized to generate Excel output.

```
%MACRO AbsentForms ;

ODS RTF FILE = "\\Drive\Reports\StudyMedicine_Absent_Forms.rtf" ;
```

The file name could also have additional date information appended to it to describe the reporting period, when it was run or something else relevant to the project's use of this data.

The first PROC REPORT stage should cover all variables of interest to describe the records that are absent. For the current example, this includes organizing the records by Site, SubjectID, VisitName and FormName along with calculated dates and the length of time the record has been absent. The use of options helps to customize the PROC REPORT performance, but can be changed based on specific project needs. By the same token, this example includes size specifications on the columns which may not create the desired appearance and can also be altered or further customized. By setting a COMPUTE block to create a break after each Site, a line of white space is added to aid in readability of the final output. On the first two DEFINE statements, ORDER is used to keep repeats of the values of Site and SubjectID from being printed on each line of the report. This also aids in readability and its use could be extended to other variables. For example, if multiple visits had the same name and were then distinguished from each other by an additional variable, then it would be advantageous to apply ORDER to VisitName as well. The remaining variables are set to DISPLAY because all of their information would be unique and should be seen as part of the report. If variables are properly labeled during an earlier DATA or PROC SQL step, then labels do not need to be added here, but PROC REPORT does make the process of specifying custom labels very easy without having to change them on the data set itself.

```
PROC REPORT DATA = Absent missing headline headskip nowindows split='*'
      (header)=[just=center font_size=1.6 background=white]
      style(column)=[just=center font_size=1.1] ;

            WHERE InTheHole = 1 ;
```

```
        COLUMN Site SubjectID VisitName FormName VisitDate LateDate DaysInHole
               <other relevant variables> ;

        DEFINE Site / order 'Site' style(column)=[just=left cellwidth=6.2cm] ;
        DEFINE SubjectID / order 'Subject*ID' style=[cellwidth=1.5cm] ;
        DEFINE VisitName / display 'Visit' style=[cellwidth=2cm] ;
        DEFINE FormName / display 'Form Name' style=[cellwidth=2cm] ;
        DEFINE VisitDate / display 'Visit*Date' style=[cellwidth=2cm] ;
        DEFINE LateDate / display 'Late*Date' style=[cellwidth=2cm] ;
        DEFINE DaysInHole / display "# of Days*Absent" style=[cellwidth=2cm] ;

        < DEINFE statements for other variables >

        COMPUTE AFTER Site ;
               LINE ' ' ;
        ENDCOMP ;
RUN ;
```

The next portion of the macro will check to see how many records actually exist in the Absent data set. This will create the trigger to print an alternate message if there are no records to display. The CALL symputx allows us to automatically take the count of records and put it in the OBS macro variable for later use.

```
DATA _null_ ;
       CALL symputx ("obs" , RecCnt ) ;
              IF 0 THEN SET Absent NOBS = RecCnt ;
       STOP ;
RUN;
```

Once the record count is being held in the OBS macro variable, then its value can be checked and used as a reporting trigger. If there are no records on which to report, then a specific note text is created. This is a generic example, but this text could be as detailed as was deemed appropriate for the report. Once the text is created in a simple DATA step, then that is the basis for an alternate PROC REPORT step. Once again, options have been used to customized the layout and sizes are specified, but any of those could be modified.

```
%IF &obs  = 0 %THEN %DO ;
       DATA Absent_Note ; NOTE = "  ----- No Absent Forms -----  " ;
       RUN ;
       PROC REPORT DATA = Absent_Note headline headskip nowindows
              style(header) = [ cellwidth = 700 cellheight = 90 ]
              style(column) = [ font_size = 7 ] ;
              COLUMNS NOTE ;
              DEFINE NOTE /  display style = [ just = center cellheight = 60 ] ;
       RUN ;
%END ;
```

With the completion of the various PROC REPORT options, there macro code needs to be wrapped up. The ODS destination needs to be closed, then the macro itself needs to be closed and then finally the macro must be called in order to generate the report. The initial ODS statement should be modified based on the type of document output desired.

```
ODS RTF CLOSE ;
%MEND ;

%AbsentForms;
```

Conclusion

The process laid out here allows us to report on data which does not currently exist for us. Sometimes that data is easy to isolate if data structures already exist to specify what is expected. Sometimes that data can be harder to isolate if we need to build the shells that synthesize the expected records and then compare existing records to those. However, once the infrastructure is in place, the possibilities are almost infinite as to how the data can then be conditioned, customized or modified to indicate whether a record is temporarily mislaid or whether it truly has fallen into a data doughnut hole.

References

Harrington, Timothy J. 2002. "An Introduction to SAS® PROC SQL". *Proceedings of the SAS User Group International 2002 Conference*. Orlando, FL: SAS

Hermansen, Sigurd W. 2012. "Database Programming in SAS SQL: A One-Day Introduction for Westat Systems Staff". Rockville, MD: Westat

Lafler, Kirk Paul. 2009. "Exploring PROC SQL® Joins and Join Algorithms". *Proceedings of the SAS Global 2009 Conference*. Washington, DC: SAS

Markovitz, Heidi. 2006. "Dup, Dedup, DUPOUT - New in PROC SORT". *Proceedings of the SouthEast SAS Users Group 2006 Conference*. Atlanta, GA: SESUG

Acknowledgments

I would like to thank Rick Mitchell for encouraging me to continue to write and being a SAS conference inspiration. I would like to thank Michael Raithel and Mike Rhoads at Westat for their administrative and editorial support. I would like to thank Westat for their institutional support of my participation in both local and regional SAS users' groups.

Contact Information

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

Sarah Woodruff
Westat
1600 Research Boulevard, WB 401
Rockville, MD 20850
240-314-7562
SarahWoodruff@Westat.com