Paper 1314-2014

DATA CLEANING: LONGITUDINAL STUDY CROSS-VISIT CHECKS

Lauren Parlett,PhD

Johns Hopkins University Bloomberg School of Public Health, Baltimore, MD

ABSTRACT

Cross-visit checks are a vital part of data cleaning for longitudinal studies. The nature of longitudinal studies encourages repeatedly collecting the same information. Sometimes, these variables are expected to remain static, go away, increase, or decrease over time. This presentation reviews the naïve and the better approaches at handling one-variable and two-variable consistency checks. For a single-variable check, the better approach features the new ALLCOMB function, introduced in SAS® 9.2. For a two-variable check, the better approach uses a BY PROCESSING variable to flag inconsistencies. This paper will provide you the tools to enhance your longitudinal data cleaning process.

INTRODUCTION

One of my first tasks as a data analyst was to "clean" the data collected in a longitudinal study that began long before I was hired. Researchers of the longitudinal study collected regular data on the subjects on a yearly basis. The same forms would be used for each visit.

Not only did the collected data have to make sense across forms, but also the data had to make sense over time. For example, if the subject had indicated at his first visit that his mother was dead, the data should not show her alive at a later visit. That sort of inconsistency had to be flagged for further investigation. My predecessor had adequately written code for the cross visit checks; however, I could not shake the feeling that the code could be more efficient and more dynamic.

In this paper I will provide examples of cross visit checks for one and two variable scenarios. For each, I will go through my initial naïve approach and then come at the problem from a more sophisticated approach. Since the ALLCOMB function is an important part of these methods, SAS® 9.2 or higher is required.

SETTING UP YOUR DATA

The data set used in this paper is completely artificial in order to illustrate concepts and possible pitfalls. This data is for a pretend study about the association between vision and stroke over time. Presented here are 5 individuals (*ID*) who each had 4 visits (*Visit*). At each visit there were four pieces of information collected: whether the person accompanying them -- their informant-- was new (*NewInfrmnt*), the date of birth of the informant (*InfrmntDOB*), whether the subject's vision was normal (*NormVision*), and whether a stroke had occurred in the past (*StrkHist*). Missing or unknown values are denoted by the value 99. Otherwise, variable coding is standard.

```
DATA mydata;
INPUT ID Visit NewInfrmnt InfrmntDOB MMDDYY10. NormVision StrkHist;
DATALINES;
10 1 1 01/13/1970 1 0
10 2 0 01/13/1970 0 0
10 3 0 02/22/1968 1 1
10 4 0 02/22/1968 1 1
11 1 1 03/04/1977 1 0
11 2 0 03/04/1977 0 0
11 3 0 03/04/1977 99 0
11 4 0 03/04/1977 1 1
12 1 1 04/14/1955 1 0
12 2 0 04/14/1955 1 0
12 3 0 04/14/1955 99 0
12 4 0 04/14/1955 1 0
13 1 1 05/21/1966 1 1
13 2 1 06/16/1980 1 99
13 3 0 06/16/1980 1 0
```

```
13 4 . 06/16/1980 1 1
14 1 1 07/08/1972 1 0
14 2 0 07/09/1972 1 0
14 3 0 07/09/1972 1 0
14 4 0 07/09/1972 1 99
;
RUN;
```

The first step in setting up the data is to convert the missing code to a missing value. Then transpose it so that there is one observation per subject such that each row contains all of the variable values across the visits. Only one variable will be transposed at a time. We will start with vision.

```
DATA mydata (DROP=i);
   SET mydata;

   ARRAY vars {*} _NUMERIC_;

   DO i=1 to HBOUND(vars);
          IF vars{i} = 99 THEN vars{i}=.;
   END;
RUN;

PROC SORT data=mydata;
   BY ID;
RUN;

PROC TRANSPOSE data=mydata PREFIX=v OUT=vision_trans(DROP=_NAME_);
   BY ID;              *Each subject gets a row;
   ID Visit;           *Names each new variable according to visit number;
   VAR Normvision;
RUN;
```

After running that code, the *mydata* dataset will look like this:

ID	v1	v2	v3	v4
10	1	0	1	1
11	1	0	.	1
12	1	1	.	1
13	1	1	1	1
14	1	1	1	1

Output 1. Study Data after Transposing by Visit Number

Each subject is on one row. The vision values for each visit are in separate columns. Now we are ready to begin checking across visits.

ONE-VARIABLE CHECK

NAÏVE APPROACH

My first approach to the problem was, admittedly, simplistic, though it did produce the expected results. Someone could present with normal vision and then subsequently have poor vision, but not vice versa. My approach was to type out logic code (IF/THEN statements) to compare each visit. Ultimately, the code should have identified IDs 10 and 11 for further review. My code looked like this:

```
DATA vision_check;
   SET vision_trans;

   *Compare against Visit 1;
   IF v1=0 AND v2=1 THEN flag=1;
   IF v1=0 AND v3=1 THEN flag=1;
```

```
    IF v1=0 AND v4=1 THEN flag=1;

    *Compare against Visit 2;
    IF v2=0 AND v3=1 THEN flag=1;
    IF v2=0 AND v4=1 THEN flag=1;

    *Compare against Visit 3;
    IF v3=0 AND v4=1 THEN flag=1;

    IF flag=1 THEN OUTPUT;
RUN;
```

The output dataset *vision_check*:

ID	v1	v2	v3	v4	flag
10	1	0	1	1	1
11	1	0	.	1	1

Output 2. One-Variable Naive Approach: Dataset after Flagging Discrepancies Using IF/THEN Statements

Nothing wrong there! But things could get a little hairy if the study ended up lasting another 2 years. How many lines of logic code would that be for this one variable? You can calculate on this own using the formula for permutations where *n* = the total number of visits and *r*= the number of visits rearranged at one time.

$$P(n,r) = \frac{n!}{(n-r)!}$$

For example, four visits with two visits evaluated at a time would require 12 lines of IF/THEN code.

$$P(4,2) = \frac{4!}{(4-2)!} = \frac{(4 \times 3 \times 2 \times 1)}{(2 \times 1)} = 12$$

Six visits with two comparisons would require 30 lines. Seven visits with two comparisons? 42 lines. That is a lot of hand coding. And doing that for each variable would take a very long time. There should be an easier way. That is when I tried to use an array and do loop to cut down on the number of lines.

```
DATA vision_checkv2 (DROP=i);
    SET vision_trans;

    ARRAY visit {*} v:;

    DO i=2 TO HBOUND(visit);
           IF visit{i-1} > visit {i} THEN flag=1;
    END;

    IF flag=1 THEN OUTPUT;
RUN;
```

That was a lot faster than coding 6 or 30 logic lines. But, there was something wrong with the *vision_checkv2* dataset.

ID	v1	v2	v3	v4	flag
10	1	0	1	1	1
11	1	0	.	1	1
12	1	1	.	1	1

Output 3. One-Variable Naive Approach: Dataset after Flagging Discrepancies Using Array and Do Loop

The code caught a missing value for subject 12. In SAS, a missing value (.) is considered less than 1; therefore, that observation was flagged as having a discrepancy where one did not exist. I tried again with another condition on the comparison. If either value is missing, the code will skip to the next number of the do loop.

```
DATA vision_checkv3 (DROP = i);
   SET vision_trans;

   ARRAY visit {*} v:;

   DO i=2 TO HBOUND(visit);
          IF visit{i} =. OR visit{i-1}=. THEN CONTINUE;
          IF visit{i} > visit {i-1} THEN flag=1;
   END;

   IF flag=1 THEN OUTPUT;
RUN;
```

And what do we get for *vision_checkv3*?

ID	v1	v2	v3	v4	flag
10	1	0	1	1	1

Output 4. One-Variable Naive Approach: Dataset after Flagging Discrepancies Using Array and Do Loop, Controlling for Missing Values

That's not right either! Subject 11 is now gone because the code could not compare the second and fourth visits where there was a discrepancy.

BETTER APPROACH

The answer involves a rarely-used SAS function ALLCOMB that is available in SAS 9.2 and later. The purpose of the ALLCOMB function is to generate combinations of variables in a specific order. Order matters in our checks (we must check earlier visit versus a later visit and in that order), so that is why we use ALLCOMB rather than ALLPERM. This seems at odds with our first naïve approach where permutations dictated the number of lines, but that was because we could interchange the order of the logic statements without a change in the outcome. That is not the case with this one-variable approach.

ALLCOMB works on elements in an array. The function parameters are the count (or row), the number of chosen elements, and the source of the elements. The result of the ALLCOMB function is a matrix of all possible combinations *in a sorted order*. The sorted order algorithm will output the same matrix every time it is run.

The process for the code is summarized below.

1. Set the data step with the transposed vision dataset.
2. Put the visits into an explicit array (*visit*).
3. Calculate the number of visits in the array (*n*).
4. Declare the number of visits to evaluate at one time (*k*).
5. Calculate the number of times the DO LOOP will have to run through the rows of the ALLCOMB matrix (*ncomb*).
6. Set up a DO LOOP where that will run *ncomb* times to compare *k* visits at a time.
7. Call the ALLCOMB function to interchange two visits and select a certain row (*i*).
8. Write a logic statement comparing the early and later visits, accounting for missing values.

Here is the code for the one-variable approach:

```
DATA vision_onevar (DROP = n k ncomb i);
   SET vision_trans;

   ARRAY visit {*} v:;

   n = dim(visit);
   k = 2;
   ncomb = comb(n,k);
   DO i=1 TO ncomb;
          CALL allcomb(i, k, of visit[*]);
          IF visit{1} < visit{2} AND visit[1]^=. THEN flag=1;
```

```
    END;

    IF flag=1 THEN OUTPUT;
RUN;
```

The resulting *vision_onevar* dataset:

ID	v1	v2	v3	v4	flag
10	1	1	0	1	1
11	.	1	0	1	1

Output 5. One-Variable Better Approach: Dataset after Flagging Discrepancies Using ALLCOMB Function

It flagged the proper IDs, just like the first naïve way, but this code can accommodate many additional visits without adding lines to the data step. Note that the values are no longer with the right visits. They are in reverse order. That can be fixed by copying the visits values to temporary variables before the do loop and then restoring them after the DO loop. There is code in the two-variable better approach that can serve as your template for this. Finally, the less than operator can be changed to greater than or not equal to depending on your particular coding scheme and logic needs.

TWO-VARIABLE CHECK

What if your consistency check depends on two variables rather than one? The value of one variable is allowed to change if another variable for that same visit also changes? That is where the two-variable approach comes in.

NAÏVE APPROACH

Our example involves the variables *NewInfrmnt* and *InfrmntDOB*. The date of birth of the informant should not change unless the informant is new (*NewInfrmnt*=1). According to this logic, subjects 10 and 14 should be flagged for further review.

My first instinct was to expand the better code from the one-variable check where the ALLCOMB function is used. First, I transposed the two variables and added the prefixes "new" and "dob" to their new columns. Next, I merged the resulting datasets. Before going into the comparison, I stored the original values in temporary variables since the ALLCOMB function ultimately reverses the values. Then, two ALLCOMB function calls are employed rather than just one.

The code:

```
DATA inft_check (DROP= n k ncomb i j m temp:);
    MERGE inft_trans inft_trans2;
    BY ID;

    ARRAY new {*} new:;
    ARRAY dob {*} dob:;
    ARRAY temp1 {4};
    ARRAY temp2 {4};

    DO i=1 to HBOUND(new);
           temp1{i} = new{i};
           temp2{i} = dob{i};
    END;


    n = dim(new);
    k = 2;
    ncomb = comb(n,k);
    DO j=1 TO ncomb;
           CALL allcomb(j, k, of new[*]);
           CALL allcomb(j, k, of dob{*});
           IF dob{1} ^= dob{2} AND new[2]=0 AND dob{1}^=. AND dob{2}^=. THEN flag=1;

    END;
```

```
    DO m=1 TO HBOUND(new);
           new{m} = temp1{m};
           dob{m} = temp2{m};
    END;

    FORMAT dob1-dob4 MMDDYY9.;
    IF flag=1 THEN OUTPUT;
RUN;
```

The resulting *inft_check* dataset:

ID	new1	new2	new3	new4	dob1	dob2	dob3	dob4
10	1	0	0	0	01/13/70	01/13/70	02/22/68	02/22/68
13	1	1	0	.	05/21/66	06/16/80	06/16/80	06/16/80
14	1	0	0	0	07/08/72	07/09/72	07/09/72	07/09/72

Output 6. Two-Variable Naive Approach: Dataset after Flagging Discrepancies Using Two ALLCOMB Functions

That's not right. What went wrong? It turns out that the code compared visit 1 and visit 3 for subject 13 and found that a new informant was not specified even though the dates of birth were different. So, it flagged subject 13. However, the new informant WAS specified for visit 2, so subsequent visits had no need to indicate a new informant was present. The two ALLCOMB calls approach was not the way to go.

BETTER APPROACH

A more elegant solution requires no function calls. Rather than using the transposed dataset, the two-variable approach makes use of the long dataset. Keep that in mind if your data is structured as wide.

The code:

```
PROC SORT data=mydata;
   BY ID InfrmntDOB;
RUN;

DATA inft_twovar;
   SET mydata (KEEP= ID Visit InfrmntDOB NewInfrmnt);
   BY ID InfrmntDOB;

   IF first.InfrmntDOB and NewInfrmnt=0 THEN flag=1;

   FORMAT InfrmntDOB MMDDYY9.;
   IF flag=1 THEN OUTPUT;
RUN;
```

This approach relies on a BY processing variable (*first.InfrmntDOB*). When a dataset is SET with one or more BY variables, the processing makes first. and last. variables at the beginning and end of each group. In this better approach, if the observation is the first of a group of date of births for a subject, *first.InformntDOB* is equal to 1. If, in that same observation, there is no new informant declared (*NewInfrmt*=0), then the observation is flagged as having a discrepancy.

The resulting *inft_twovar* dataset:

ID	Visit	New Infrmnt	Infrmnt DOB	flag
10	3	0	02/22/68	1
14	2	0	07/09/72	1

Output 7. Two-Variable Better Approach: Dataset after Flagging Discrepancies Using BY Variables

The dataset has flagged the proper IDs. It shows you the visit where the error occurred. Keep in mind that this method

will not flag visits where there is a new date of birth and the new informant variable is missing. Those should be flagged and hand-checked using other code.

CONCLUSION

Checking for cross-visit consistency is an important part of data cleaning. By doing this, you can catch typos, inconsistent assessments, and more. Sometimes the naïve ways of coding work, such as the first naïve way I tried in the one variable example. Unfortunately, those solutions are not always dynamic and can balloon when additional follow up visits are added to the study. Since longitudinal data collection (or any data collection!) is bound to have missing or unknown values, using the newly offered-- though rarely talked about-- function **ALLCOMB** is essential for checking more than just one visit versus the next visit. When examining two variables, using the BY statement in a DATA step and the resulting BY processing variables (FIRST. or LAST.) is a good way to flag discrepancies when one value relies on another. The examples provided in this paper can be scaled up and supplemented according to your specific data cleaning needs.

ACKNOWLEDGEMENTS

This work was completed while working on the Johns Hopkins University BIOCARD study. Funding for the BIOCARD study is provided by the National Institute of Aging, NIH, grant U01AG033655.

Special thanks to the SAS Support Community for providing ideas about the two-variable cross visit check. Also, thank you to Joseph Guido for reviewing this paper.

CONTACT INFORMATION

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

Lauren Parlett, PhD
Johns Hopkins University
Bloomberg School of Public Health
Department of Epidemiology
615 N. Wolfe St. Rm W6024
Baltimore, MD 21205
(410) 502 - 5448 (phone)
(410) 502 - 4623 (fax)