

## Data Cleaning 101

Ronald Cody, Ed.D., Robert Wood Johnson Medical School, Piscataway, NJ

### INTRODUCTION

One of the first and most important steps in any data processing task is to verify that your data values are correct or, at the very least, conform to some a set of rules. For example, a variable called GENDER would be expected to have only two values; a variable representing height in inches would be expected to be within reasonable limits. Some critical applications require a double entry and verification process of data entry. Whether this is done or not, it is still useful to run your data through a series of data checking operations. We will describe some basic ways to help identify invalid character and numeric data values, using SAS® software.

### A SAMPLE DATA SET

In order to demonstrate data cleaning techniques, we have constructed a small raw data file called PATIENTS.TXT. We will use this data file and, in later sections, a SAS data set created from this raw data file, for many of the examples in this text. The program to create this data set can be found at the end of this paper. Here is a short description of the variables it contains:

### DESCRIPTION OF THE FILE PATIENTS.TXT

The data file PATIENTS.TXT contains both character and numeric variables from a typical clinical trial. A number of data errors were included in the file so that you can test the data cleaning programs that we develop in this text. The file PATIENTS.TXT is located in a directory (folder) called C:\CLEANING. This is the directory that will be used throughout this text as the location for data files, SAS data sets, SAS programs, and SAS macros. You should be able to modify the INFILE and LIBNAME statements to fit your own operating environment. The layout for the data file PATIENTS.TXT is as follows:

Variable Name	Description	Variable Type	Valid Values
PATNO	Patient Number	Character	Numerals
GENDER	Gender	Character	'M' or 'F'
VISIT	Visit Date	MMDDYY10.	Any valid date
HR	Heart Rate	Numeric	40 to 100
SBP	Systolic Blood Pres.	Numeric	80 to 200
DBP	Diastolic Blood Pres.	Numeric	60 to 120
DX	Diagnosis Code	Character	1 to 3 digits
AE	Adverse Event	Character	'0' or '1'

There are several character variables which should have a limited number of valid values. For this exercise, we expect values of GENDER to be 'F' or 'M', values of DX the numerals 1 through 999, and values of AE (adverse events) to be 0 or 1. A very simple approach to identifying invalid character values in this file is to use PROC FREQ to list all the unique values of these variables. Of course, once invalid values are identified using this technique, other means will have to be employed to locate specific records (or patient numbers) corresponding to the invalid values.

### CHECKING FOR INVALID CHARACTER VALUES

Let's start with some techniques that are useful with character data, especially variables that take on a relatively few valid values.

One very simple technique is to run PROC FREQ on all of the character variables that represent a limited number of categories, such as gender or a demographic code of some sort. The appropriate PROC FREQ statements to list the unique values (and their frequencies) for the variables GENDER, DX, and AE is shown next:

```
PROC FREQ DATA=PATIENTS;
  TITLE "FREQUENCY COUNTS";
  TABLES GENDER DX AE / NOCUM
                                         NOPERCENT;
RUN;
```

To simplify the output from PROC FREQ, we elected to use the NOCUM (no cumulative statistics) and NOPERCENT (no percentages) TABLES options since we only want frequency counts for each of the unique character values. Here is a partial listing of the output from running this program:

#### FREQUENCY COUNTS

The FREQ Procedure

```

                Gender
GENDER      Frequency
-----
2              1
F              12
M              13
X              1
f              2
```

Frequency Missing = 1

Let's focus in on the frequency listing for the variable GENDER. If valid values for GENDER are 'F' and 'M' (and missing), this listing points out several data errors. The values '2' and 'X' both occur once. Depending on the situation, the lower case value 'f' could be considered an error or not. If lower case values were entered into the file by mistake but the value, aside from the case, was correct, we could change all lower case values to upper case with the UPCASE function. At this point, it is necessary to run additional programs to identify the location of these errors. Running PROC FREQ is still a useful first step in identifying errors of these types and it is also useful as a last step after the data have been cleaned, to ensure that all the errors have been identified and corrected.

### USING A DATA STEP TO IDENTIFY INVALID CHARACTER VALUES

Our next task is the use a Data Step to identify invalid data values and to determine where they occur in the raw data file or the SAS data set (by listing the patient number). We will check GENDER, DX, and AE, using several different methods.

First, we write a simple data step that reports invalid data values using PUT statements in a `_NULL_` Data Step. Here is the program:

```
DATA _NULL_ ;
  INFILE "C:\PATIENTS.TXT" PAD;
  FILE PRINT; ***SEND OUTPUT TO THE
              OUTPUT WINDOW;
  TITLE "LISTING OF INVALID DATA";
  ***NOTE: WE WILL ONLY INPUT THOSE
          VARIABLES OF INTEREST;
  INPUT @1  PATNO    $3.
        @4  GENDER  $1.
        @24 DX      $3.
        @27 AE      $1.;

  ***CHECK GENDER;
  IF GENDER NOT IN ('F','M',' ') THEN
    PUT PATNO= GENDER=;
  ***CHECK DX;
  IF VERIFY(DX,' 0123456789') NE 0
    THEN PUT PATNO= DX=;
  ***CHECK AE;
  IF AE NOT IN ('0','1',' ') THEN PUT
    PATNO= AE=;
RUN;
```

Before we discuss the output, let's spend a moment looking over the program. First, notice the use of `DATA _NULL_`. Since the only purpose of this program is to identify invalid data values, there is no need to create a SAS data set. The `FILE PRINT` statement causes the results of any subsequent `PUT` statements to be sent to the output window (or output device). `GENDER` and `AE` are checked using the `IN` statement. The statement:

```
IF X IN ('A','B','C') THEN . . .;
```

Is equivalent to:

```
IF X = 'A' OR X = 'B' OR X = 'C' THEN . . .;
```

That is, if `X` is equal to any of the value in the list following the `IN` statement, the expression is evaluated as true. We want an error message printed when the value of `GENDER` is not one of the acceptable values ('F','M', or missing). We therefore place a `NOT` in front of the whole expression, triggering the error report for invalid values of `GENDER` or `AE`.

There are several alternative ways that the gender checking statement can be written. The method we used above, uses the `IN` operator. A straightforward alternative to the `IN` operator is:

```
IF NOT (GENDER EQ 'F' OR GENDER EQ 'M' OR
        GENDER = ' ') THEN PUT PATNO= GENDER=;
```

Another possibility is:

```
IF GENDER NE 'F' AND GENDER NE 'M' AND
   GENDER NE ' ' THEN PUT PATNO= GENDER=;
```

While all of these statements checking for `GENDER` and `AE` produce the same result, the `IN` statement is probably the easiest to write, especially if there are a large number of possible values to check. Always be sure to consider whether you want to identify missing values as invalid or not. In the statements above, we are

allowing missing values as valid codes. If you want to flag missing values, do not include it in the list of valid codes.

If you want to allow lower case M's and F's as valid values, you can add the single line

```
GENDER = UPCASE(GENDER);
```

right before the line which checks for invalid gender codes. As you can probably guess, the `UPCASE` function changes all lower case letters to upper case letters.

A statement similar to the gender checking statement is used to test the adverse events.

There are so many valid values for `DX` (any numeral from 1 to 999), that the approach we used for `GENDER` and `AE` would be inefficient (and wear us out typing) if we used it to check for invalid `DX` codes. The `VERIFY` function is one of the many possible ways we can check to see if there is a value other than the numerals 0 to 9 or blank as a `DX` value. (Note that an imbedded blank would not be detected with this code.) The `VERIFY` function has the form:

```
VERIFY (CHARACTER_VAR, VERIFY_STRING)
```

where the `verify_string` is either a character variable or a series of character values placed in single or double quotes. The `VERIFY` function returns the first position in the `character_var` that contains a character that is not in the `verify_string`. If there are no characters in the `character_var` that are not in the `verify_string`, the function returns a zero. Wow, that sounds complicated. To make this clearer, let's look at how we can use the `VERIFY` function to check for valid `GENDER` values. We write:

```
IF VERIFY (GENDER, 'FM ') NE 0 THEN PUT PATNO=
   GENDER=;
```

Notice that we included a blank in the `verify_string` so that missing values will be considered valid. If `GENDER` has a value other than an 'F', 'M', or missing, the `verify` function will return the position of the invalid character in the string. But, since the length `GENDER` is one, any invalid value for `GENDER` will return a one.

### USING PROC PRINT WITH A WHERE STATEMENT TO LIST INVALID DATA VALUES

While `PROC MEANS` and `PROC UNIVARIATE` can be useful as a first step in data cleaning for numeric variables, they can produce large volumes of output and may not give you all the information you want, and certainly not in a concise form. One way to check each numeric variable for invalid values is to use `PROC PRINT`, followed by the appropriate `WHERE` statement.

Suppose we want to check all the data for any patient having a heart rate outside the range of 40 to 100, a systolic blood pressure outside the range of 80 to 200 and a diastolic blood pressure outside the range of 60 to 120. For this example, we will not flag missing values as invalid. A `PROC PRINT` statement, followed by a `WHERE` statement, can be used to list data values outside a given range. The `PROC PRINT` statements below will report all patients with out-of-range values for heart rate, systolic blood pressure, or diastolic blood pressure.

### USING A WHERE STATEMENT WITH PROC PRINT TO LIST OUT-OF-RANGE DATA

```
PROC PRINT DATA=CLEAN.PATIENTS;
  WHERE HR NOT BETWEEN 40 AND 100 AND
```

```

HR IS NOT MISSING
SBP NOT BETWEEN 80 AND 200 AND
SBP IS NOT MISSING
DBP NOT BETWEEN 60 AND 120 AND
DBP IS NOT MISSING;
TITLE "OUT-OF-RANGE VALUES FOR NUMERIC
VARIABLES";
ID PATNO;
VAR HR SBP DBP;
RUN;

```

The resulting listing is:

Out-of-range Values for Numeric Variables

PATNO	HR	SBP	DBP
004	101	200	120
008	210	.	.
009	86	240	180
010	.	40	120
011	68	300	20
014	22	130	90
017	208	.	84
321	900	400	200
020	10	20	8
023	22	34	78

#### USING A DATA STEP TO CHECK FOR INVALID VALUES

A simple DATA \_NULL\_ step can also be used to produce a report on out-of-range values. We proceed as we did in Chapter 1. Here is the program:

#### USING A DATA \_NULL\_ DATA STEP TO LIST OUT-OF-RANGE DATA VALUES

```

DATA _NULL_;
  INFILE "C:\CLEANING\PATIENTS.TXT" PAD;
  FILE PRINT; **OUTPUT TO THE OUTPUT WINDOW;
  TITLE "LISTING OF PATIENT NUMBERS AND INVALID
DATA VALUES";
  ***NOTE: WE WILL ONLY INPUT THOSE VARIABLES OF
INTEREST;
  INPUT @1 PATNO $3.
        @15 HR 3.
        @18 SBP 3.
        @21 DBP 3.;
  ***CHECK HR;
  IF (HR LT 40 AND HR NE .) OR HR GT 100 THEN
  PUT PATNO= HR=;
  ***CHECK SBP;
  IF (SBP LT 80 AND SBP NE .) OR SBP GT 200 THEN
  PUT PATNO= SBP=;
  ***CHECK DBP;
  IF (DBP LT 60 AND DBP NE .) OR DBP GT 120 THEN
  PUT PATNO= DBP=;
RUN;

```

We don't need the parentheses in the IF statements above since the AND operator is evaluated before the OR operator. However, since this author can never seem to remember the order of operation of Boolean operators, the parentheses do no harm.

The output from the above program is shown next:

```

OR
OR
Listing of Patient Numbers and Invalid Data
Values
PATNO=004 HR=101
PATNO=008 HR=210
PATNO=009 SBP=240
PATNO=009 DBP=180
PATNO=010 SBP=40
PATNO=011 SBP=300
PATNO=011 DBP=20
PATNO=014 HR=22
PATNO=017 HR=208
PATNO=321 HR=900
PATNO=321 SBP=400
PATNO=321 DBP=200
PATNO=020 HR=10
PATNO=020 SBP=20
PATNO=020 DBP=8
PATNO=023 HR=22
PATNO=023 SBP=34

```

Notice that a statement such as "IF HR LE 40" includes missing values since missing values are interpreted by SAS programs as the smallest possible value. Therefore, a statement like:

```
IF HR LT 40 OR HR GT 100 THEN PUT PATNO= HR=;
```

will produce a listing for patients which includes missing heart rates as well as out-of-range values (which may be what you want), because missing values are certainly less than 40 (they are less than any nonmissing number).

#### USING USER DEFINED FORMATS TO DETECT INVALID VALUES

Another way to check for invalid values of a character variable from raw data is to use user-defined formats. There are several possibilities here. One, we can create a format that formats all valid character values as is and formats all invalid values to a single error code. We start out with a program that simply assigns formats to the character variables and uses PROC FREQ to list the number of valid and invalid codes. Following that, we will extend the program, using a Data Step, to identify which ID's have invalid values. The following program uses formats to convert all invalid data values to a single value:

```

PROC FORMAT;
  VALUE $GENDER 'F','M' = 'VALID'
                ' '      = 'MISSING'
                OTHER    = 'MISCODED';
  VALUE $DX '001' - '999' = 'VALID'
            ' '          = 'MISSING'
            OTHER       = 'MISCODED';
  VALUE $AE '0','1' = 'VALID'
            ' '     = 'MISSING'
            OTHER  = 'MISCODED';
RUN;

```

```

PROC FREQ DATA=CLEAN.PATIENTS;
  TITLE "USING FORMATS";
  FORMAT GENDER $GENDER.
         DX      $DX.
         AE      $AE.;
  TABLES GENDER DX AE / NOCUM NOPERCENT
          NOCOL NOROW;
RUN;

```

For the variables GENDER and AE, which have specific valid values, we list each of the valid values on the range to the left of the equal sign in the VALUE statement and format each of these values with the value 'Valid'. You may choose to lump the missing value with the valid values if that is appropriate, or you may want to keep track of missing values separately as we did. Finally, any value other than the valid values or a missing value will be formatted as 'Miscoded'. All that is left is to run PROC FREQ to count the number of 'Valid', 'Missing', and 'Miscoded' values. Output from PROC FREQ is as follows:

Using FORMATS to Identify Invalid Values

```

Gender
-----
GENDER      Frequency
-----
Miscoded    4
Valid       25

Frequency Missing = 1

Diagnosis Code
-----
DX           Frequency
-----
Valid        21
Miscoded     2

Frequency Missing = 7

Adverse Event?
-----
AE           Frequency
-----
Valid        28
Miscoded     1

Frequency Missing = 1

```

This isn't particularly useful. It doesn't tell us which observations (patient numbers) contain missing or invalid values. Let's modify the program, adding a Data Step, so that ID's with invalid character values are listed.

```

PROC FORMAT;
  VALUE $GENDER 'F','M' = 'VALID'
              ' '      = 'MISSING'
              OTHER    = 'MISCODED';
  VALUE $DX '001' - '999' = 'VALID'
          ' '          = 'MISSING'
          OTHER        = 'MISCODED';
  VALUE $AE '0','1' = 'VALID'
          ' '      = 'MISSING'
          OTHER    = 'MISCODED';
RUN;

DATA _NULL_;
  INFILE "C:PATIENTS,TXT" PAD;
  FILE PRINT; ***SEND OUTPUT TO THE
              OUTPUT WINDOW;
  TITLE "LISTING OF INVALID DATA VALUES";
  ***NOTE: WE WILL ONLY INPUT THOSE

```

```

          VARIABLES OF INTEREST;
INPUT @1  PATNO  $3.
      @4  GENDER $1.
      @24 DX     $3.
      @27 AE     $1.;

IF PUT(GENDER,$GENDER.) = 'MISCODED'
  THEN PUT PATNO= GENDER=;
IF PUT(DX,$DX.) = 'MISCODED' THEN PUT
PATNO= DX=;
IF PUT(AE,$AE.) = 'MISCODED' THEN PUT
PATNO= AE=;
RUN;

```

The "heart" of this program is the PUT function. To review, the PUT function is similar to the INPUT function. It takes the form:

```
CHARACTER_VAR = PUT(VARIABLE,FORMAT)
```

where character\_var is a character variable which contains the formatted value of variable. The result of a PUT function is always a character variable and it is frequently used to perform numeric to character conversions. In this instance, the first argument of the PUT function is a character variable. In the program above, the result of the PUT function for any invalid data values would be the value 'Miscoded'.

Output from this program is:

```

Listing of Invalid Data Values
PATNO=002 DX=X
PATNO=003 GENDER=X
PATNO=004 AE=A
PATNO=010 GENDER=f
PATNO=013 GENDER=2
PATNO=002 DX=X
PATNO=023 GENDER=f

```

### CHECKING FOR INVALID NUMERIC VALUES

The techniques for checking for invalid numeric data are quite different from the techniques we used with character data. Although there are usually many different values a numeric variable can take on, there are several techniques that we can use to help identify data errors. One simple technique is to examine some of the largest and smallest data values for each numeric variable. If we see values such as 12 or 1200 for a systolic blood pressure, we can be quite certain that an error was made, either in entering the data values or on the original data collection form.

There are also some internal consistency methods that can be used to identify possible data errors. If we see that most of the data values fall within a certain range of values, then any values that fall far enough outside that range may be data errors. We will develop programs based on these ideas in this chapter.

### USING PROC MEANS, PROC TABULATE, AND PROC UNIVARIATE TO LOOK FOR OUTLIERS

One of the simplest ways to check for invalid numeric values is to run either PROC MEANS or PROC UNIVARIATE. By default, PROC MEANS will list the minimum and maximum values, along with the n, mean, and standard deviation. PROC UNIVARIATE is somewhat more useful in detecting invalid values, providing us with a listing of the five lowest and five highest values, along with stem-and-leaf plots and box plots. Let's first look at how we can use PROC MEANS for very simple checking of numeric variables.



We certainly get lots of information from PROC UNIVARIATE, perhaps too much. Starting off, we get some descriptive univariate statistics (hence the procedure name) for each of the variables listed in the VAR statement. Most of these statistics are not very useful in the data checking operation. The number of nonmissing observations (N), the number of observations not equal to zero (Num ^= 0) and the number of observations greater than zero (Num > 0) are probably the only items in this list of interest to us at this time.

One of the most important sections of the Univariate output, for data checking purposes, is the section labeled "Extremes." Here we see the five lowest and five highest values for each of our variables. For example, for the variable HR (heart rate), we see three possible data errors under the column labels "Lowest" (10, 22, and 22) and three possible data errors in the "Highest" column (208, 210, and 900). Obviously, having knowledge of reasonable values for each of your variables is essential if this information is to be of any use. Next to the listing of the lowest and highest values, is the observation number containing this value. What would be more useful would be the patient or subject number you assigned to each patient. This is easily accomplished by adding an ID statement to PROC UNIVARIATE. You list the name of your identifying variable following the keyword ID. The values of this ID variable are then used in place of the Obs column. Here are the modified PROC UNIVARIATE statements and the "Extremes" column for heart rate that result:

```
PROC UNIVARIATE DATA=PATIENTS PLOT;
  TITLE "USING PROC UNIVARIATE";
  ID PATNO;
  VAR HR SBP DBP;
RUN;
```

The section of output showing the Extremes for the variable heart rate (HR) follows:

```
Extremes
  Lowest  ID      Highest  ID
      10(020  )      88(007  )
      22(023  )     101(004  )
      22(014  )     208(017  )
      48(022  )     210(008  )
      58(019  )     900(321  )

Missing Value      .
Count              3
% Count/Nobs      10.00
```

Notice that the column labeled ID now contains the values of our ID variable (PATNO), making it easier to locate the original patient data and check for errors.

### USING A DATA STEP TO CHECK FOR INVALID VALUES

While PROC MEANS and PROC UNIVARIATE can be useful as a first step in data cleaning for numeric variables, they can produce large volumes of output and may not give you all the information you want, and certainly not in a concise form. One way to check each numeric variable for invalid values is to do some data checking in the Data Step.

Suppose we want to check all the data for any patient having a heart rate outside the range of 40 to 100, a systolic blood pressure outside the range of 80 to 200 and a diastolic blood

pressure outside the range of 60 to 120. A simple DATA \_NULL\_ step can easily accomplish this task. Here it is:

```
DATA _NULL_;
  INFILE "C:PATIENTS,TXT" PAD;
  FILE PRINT; ***SEND OUTPUT TO THE
                OUTPUT WINDOW;
  TITLE "LISTING OF INVALID DATA VALUES";
  ***NOTE: WE WILL ONLY INPUT THOSE
                VARIABLES OF INTEREST;
  INPUT @1  PATNO    $3.
        @15 HR      3.
        @18 SBP     3.
        @21 DBP     3.;
  ***CHECK HR;
  IF (HR LT 40 AND HR NE .) OR HR GT 100
    THEN PUT PATNO= HR=;
  ***CHECK SBP;
  IF (SBP LT 80 AND SBP NE .) OR SBP GT
    200 THEN PUT PATNO= SBP=;
  ***CHECK DBP;
  IF (DBP LT 60 AND DBP NE .) OR DBP GT
    120 THEN PUT PATNO= DBP=;
RUN;
```

We don't need the parentheses in the IF statements above since the AND operator is evaluated before the OR operator. However, since this author can never seem to remember the order of operation of Boolean operators, the parentheses do no harm and make sure the program is doing what you want it to.

The output from the above program is shown next:

#### Listing of Invalid Data Values

```
PATNO=004 HR=101
PATNO=008 HR=210
PATNO=009 SBP=240
PATNO=009 DBP=180
PATNO=010 SBP=40
PATNO=011 SBP=300
PATNO=011 DBP=20
PATNO=014 HR=22
PATNO=017 HR=208
PATNO=321 HR=900
PATNO=321 SBP=400
PATNO=321 DBP=200
PATNO=020 HR=10
PATNO=020 SBP=20
PATNO=020 DBP=8
PATNO=023 HR=22
PATNO=023 SBP=34
```

Notice that the checks for values below some cutoff also check that the value is not missing. Remember that missing values are interpreted by SAS programs as being more negative than any nonmissing number. Therefore, we cannot just write a statement like:

```
IF HR LT 40 OR HR GT 100 THEN PUT
  PATNO= HR=;
```

because missing values will be included in the error listing.

### USING FORMATS FOR RANGE CHECKING

Just as we did with character values in Chapter 1, we can use user-defined formats to check for out of range data values using PROC FORMAT. The program below uses formats to find invalid data values, based on the same ranges used above.

```
***PROGRAM TO CHECK FOR OUT OF RANGE
VALUES, USING USER DEFINED FORMATS;
PROC FORMAT;
  VALUE HR_CHK 40-100, . = 'OK';
  VALUE SBP_CHK 80-200, . = 'OK';
  VALUE DBP_CHK 60-120, . = 'OK';
RUN;
```

```
DATA _NULL_;
  INFILE "C:PATIENTS.TXT" PAD;
  FILE PRINT; ***SEND OUTPUT TO THE
  OUTPUT WINDOW;
  TITLE "LISTING OF INVALID VALUES";
  ***NOTE: WE WILL ONLY INPUT THOSE
  VARIABLES OF INTEREST;
  INPUT @1 PATNO $3.
         @15 HR 3.
         @18 SBP 3.
         @21 DBP 3.;
  IF PUT(HR,HR_CHK.) NE 'OK' THEN PUT
  PATNO= HR=;
  IF PUT(SBP,SBP_CHK.) NE 'OK' THEN PUT
  PATNO= SBP=;
  IF PUT(DBP,DBP_CHK.) NE 'OK' THEN PUT
  PATNO= DBP=;
RUN;
```

This is a fairly simple and efficient program. The user-defined formats HR\_CHK, SBP\_CHK, and DBP\_CHK all assign the format 'OK' for any data value in the acceptable range. In the Data Step, the PUT function is used to test if a value outside the valid range was found. For example, a value of 22 for heart rate would not fall within the range of 40 to 100 or missing and the format OK would not be assigned. The result of the PUT function for heart rate is not equal to 'OK' and the argument of the IF statement is true. The appropriate PUT statement is then executed and the invalid value is printed to the print file.

Output from this program is shown next:

#### Listing of Invalid Values

```
PATNO=004 HR=101
PATNO=008 HR=210
PATNO=009 SBP=240
PATNO=009 DBP=180
PATNO=010 SBP=40
PATNO=011 SBP=300
PATNO=011 DBP=20
PATNO=014 HR=22
PATNO=017 HR=208
PATNO=321 HR=900
PATNO=321 SBP=400
PATNO=321 DBP=200
PATNO=020 HR=10
PATNO=020 SBP=20
PATNO=020 DBP=8
PATNO=023 HR=22
PATNO=023 SBP=34
```

### EXTENDING PROC UNIVARIATE TO LOOK FOR LOWEST AND HIGHEST VALUES BY PERCENTAGE

Let's return to the problem of locating the "n" lowest and "n" highest values for each of several numeric variables in our data set. Remember we used PROC UNIVARIATE to list the five lowest and five highest values for our three numeric variables. First of all, this procedure printed lots of other statistics that we didn't need (or want). Can we write a program that will list any number of low and high values? Yes, and here's how to do it.

The approach we will use is to have PROC UNIVARIATE output a data set containing the cutoff values on the lower and upper range of interest. The program we describe here will list the bottom and top "n" percent of the values.

Here is a program, using PROC UNIVARIATE, that will print out the bottom and top "n" percent of the data values.

```
***SOLUTION USING PROC UNIVARIATE AND
PERCENTILES;

***THE TWO MACRO VARIABLES THAT FOLLOW
DEFINE THE LOWER AND UPPER PERCENTILE
CUT POINTS;
%LET LOW_PER=20; {1}
%LET UP_PER=80; {2}

***CHOOSE A VARIABLE TO OPERATE ON;
%LET VAR = HR; {3}

PROC UNIVARIATE DATA=PATIENTS NOPRINT;
  VAR &VAR;
  ID PATNO;
  OUTPUT OUT=TMP PCTLPTS=&LOW_PER &UP_PER
  PCTLPRE = L_; {5}
RUN;

DATA HILO;
  SET PATIENTS(KEEP=PATNO &VAR); {6}
  ***BRING IN LOWER AND UPPER CUTOFFS
  FOR VARIABLE;
  IF _N_ = 1 THEN SET TMP; {7}
  IF &VAR LE L_&LOW_PER THEN DO;
    RANGE = 'LOW ';
    OUTPUT;
  END;
  ELSE IF &VAR GE L_&UP_PER THEN DO;
    RANGE = 'HIGH';
    OUTPUT;
  END;
RUN;

PROC SORT DATA=HILO(WHERE=(&VAR NE .));
  BY DESCENDING RANGE &VAR; {8}
RUN;

PROC PRINT DATA=HILO;
  TITLE "LOW AND HIGH VALUES";
  ID PATNO;
  VAR RANGE &VAR;
RUN;
```

Let's go through this program step by step. To make the program somewhat general, we are using several macro variables. Lines {1} and {2} assign the lower and upper percentiles to the macro variables LOW\_PER and UP\_PER (low percentage and upper percentage). To test out our program, we have set these variables to 20 and 80 respectively. In statement {3}, a macro variable (VAR) is assigned the value of one of our numeric

variables to be checked. PROC UNIVARIATE can be used to create an output data set containing information that is normally printed out by the procedure. Since we only want the output data set and not the listing from the procedure, we use the NOPRINT option {4}. As we did before, we are supplying PROC UNIVARIATE with an ID statement so that the ID variable (PATNO in this case) will be included in the output data set. Line {5} defines the name of the output data set and what information we want it to include. The keyword OUT= names our data set (TMP) and PCTLPTS= instructs the program to create two variables; one to hold the value of the VAR variable at the 20<sup>th</sup> percentile and the other for the 80<sup>th</sup> percentile. In order for this procedure to create the variable names for these two variables, the keyword PCTLPRE= (percentile prefix) is used. Since we set the prefix to L\_, the procedure will create two variables, L\_20 and L\_80. The cut points you choose are combined with your choice of prefix to create these two variable names. Data set TMP contains only one observation and three variables, PATNO (because of the ID statement), L\_20 and L\_80. The value of L\_20 is 58 and the value of L\_80 is 88, the 20<sup>th</sup> and 80<sup>th</sup> percentile cutoffs, respectively. The remainder of the program is easier to follow.

We want to add the two values of L\_20 and L\_80 to every observation in the original PATIENTS data set. We do this with a trick. The SET statement in line {6} brings in an observation from the PATIENTS data set, keeping only the variables PATNO and HR (since the macro variable &VAR was set to HR). Line {7} is executed only on the first iteration of this data step (when \_N\_ is equal to 1). Since all variables brought in with a SET statement are automatically retained, the values for L\_20 and L\_80 will be added to every observation in the data set HILO.

Finally, for each observation coming in from PATIENTS, the value of HR is compared to the lower and upper cutoff points defined by L\_20 and L\_80. If the value of HR is at or below the value of L\_20, RANGE is set to the value 'LOW' and the observation is added to data set HILO. Likewise, if the value of HR is at or above the value of L\_80, RANGE is set to 'HIGH' and the observation is added to data set HILO. Before we print out the contents of data set HILO, we sort it first (8) so that the low values and high values are grouped and within these groups, the values are in order (from lowest to highest). The keyword DESCENDING is used in the first level sort so that the LOW values are listed before the HIGH values ('H' comes before 'L' in an alphabetical sort). Within each of these two groups, the data values are listed from low to high. It would probably be nicer for the HIGH values to be listed from highest to lowest, but it would not be worth the effort. Below, is the final listing from this program:

Low And High Values For Variables

PATNO	RANGE	HR
020	LOW	10
014	LOW	22
023	LOW	22
022	LOW	48
003	LOW	58
019	LOW	58
001	HIGH	88
007	HIGH	88
004	HIGH	101
017	HIGH	208
008	HIGH	210

```
321      HIGH      900
```

### CREATING ANOTHER WAY TO FIND LOWEST AND HIGHEST VALUES

There is always (usually?) more than one way to solve any SAS problem. Here is a program to list the 10 lowest and 10 highest values for a variable in a SAS data set:

```
***ASSIGN VALUES TO TWO MACRO VARIABLES;
%LET VAR = HR;
%LET IDVAR = PATNO;

PROC SORT DATA=PATIENTS(KEEP=&IDVAR
                        &VAR {1})
    WHERE=(&VAR NE .)
    OUT=TMP;
    BY &VAR;
RUN;

DATA _NULL_;
    TITLE "TEN LOWEST AND TEN HIGHEST VALUES FOR
          VAR";
    SET TMP NOBS=NUM_OBS;    {2}
    HIGH = NUM_OBS - 9;    {3}
    FILE PRINT;

    IF _N_ LE 10 THEN DO;    {4}
        IF _N_ = 1 THEN PUT / "TEN LOWEST VALUES" ;
        PUT "&ID = " &IDVAR @15 &VAR;
    END;

    IF _N_ GE HIGH THEN DO;    {5}
        IF _N_ = HIGH THEN PUT / "TEN HIGHEST
VALUES";
        PUT "&ID = " &IDVAR @15 &VAR;
    END;

RUN;
```

To make the program as efficient as possible, a KEEP= data set option is used with PROC SORT {1}. In addition, only the nonmissing observations are placed in the sorted temporary data set TMP, because of the WHERE= data set option. Data set TMP will contain only the ID variable and the variable to be checked, in order from lowest to highest. Therefore, the first 10 observations in this data set are the 10 lowest, nonmissing values for the variable to be checked. We use the NOBS= data set option (line {2}) to obtain the number of observations in data set TMP. Since this data set only contains nonmissing values, the 10 highest values for our variable will start with observation NUM\_OBS - 9. This program uses a DATA \_NULL\_ and PUT statements to provide the listing of low and high values. As an alternative, you could create a temporary data set and use PROC PRINT to provide the listing.

One final note: this program does not check if there are fewer than 20 nonmissing observations for the variable to be checked. That would probably be overkill. If you had that few observations, you wouldn't really need a program at all, just a PROC PRINT!

We ran this program on our PATIENTS data set for the heart rate variable (HR) with the following result:

Ten Lowest and Ten Highest Values for HR

Ten Lowest Values

```
PATNO = 020 10
PATNO = 014 22
PATNO = 023 22
PATNO = 022 48
PATNO = 003 58
PATNO = 019 58
PATNO = 012 60
PATNO = 123 60
PATNO = 028 66
PATNO = 003 68
```

Ten Highest Values

```
PATNO = 006 82
PATNO = 002 84
PATNO = 002 84
PATNO = 009 86
PATNO = 001 88
PATNO = 007 88
PATNO = 004 101
PATNO = 017 208
PATNO = 008 210
PATNO = 321 900
```

#### CHECKING A RANGE USING AN ALGORITHM BASED ON STANDARD DEVIATION

One way of deciding what constitutes reasonable cutoffs for low and high data values is to use an algorithm based on the data values themselves. For example, you could decide to flag all values more than two standard deviations from the mean. However, if you had some severe data errors, the standard deviation could be so badly inflated that obviously incorrect data values might lie within two standard deviations. A possible workaround for this would be to compute the standard deviation after removing some of the lowest and highest values. For example, you could compute a standard deviation of the middle 50% of your data and use this to decide on outliers. Another popular alternative is to use an algorithm based on the interquartile range (the difference between the 25<sup>th</sup> percentile and the 75<sup>th</sup> percentile). We will present some programs and based on these ideas in this section.

Let's first see how we could identify data values more than two standard deviations from the mean. We can use PROC MEANS to compute the standard deviations and a short Data Step to select the outliers. Here is such a program:

```
LIBNAME CLEAN "C:\CLEANING";
PROC MEANS DATA=PATIENTS NOPRINT;
  VAR HR SBP DBP;
  OUTPUT OUT=MEANS (DROP=_TYPE_ _FREQ_)
    MEAN=M_HR M_SBP M_DBP
    STD=S_HR S_SBP S_DBP;
RUN;

DATA _NULL_;
  FILE PRINT;
  %LET N_SD = 2;
  SET CLEAN.PATIENTS;
  IF _N_ = 1 THEN SET MEANS;
  ARRAY RAW[3] HR SBP DBP;
  ARRAY MEAN[3] M_HR M_SBP M_DBP;
```

```
  ARRAY STD[3] S_HR S_SBP S_DBP;

  DO I = 1 TO 3;
    IF RAW[I] LT MEAN[I] - &N_SD*STD[I] AND
    RAW[I] NE .
    OR RAW[I] GT MEAN[I] + &N_SD*STD[I] THEN
    PUT PATNO= RAW[I]=;
  END;
RUN;
```

The PROC MEANS step computes the mean and standard deviation for each of the numeric variables in our data set. To make the program more flexible, the number of standard deviations above or below the mean that we would like to report, is assigned to a macro variable (N\_SD). In order to compare each of the raw data values against the limits defined by the mean and standard deviation, we need to combine the values in the single observation data set created by PROC MEANS to the original data set. We use the same trick we used earlier, that is, we execute a SET statement only once, when \_N\_ is equal to one. Since all the variables brought into the program data vector (PDV) are retained when we use a SET statement, these summary values will be available in each observation in the PATIENTS data set. Finally, to save some typing, three arrays were created to hold the original raw variables, the means, and the standard deviations, respectively. The IF statement at the bottom of this data step prints out the ID variable and the raw data value for any value above or below the designated cutoff.

The results of running this program on our PATIENTS data set with N\_SD set to two follows:

Statistics for Numeric Variables

```
PATNO=009 DBP=180
PATNO=011 SBP=300
PATNO=321 HR=900
PATNO=321 SBP=400
PATNO=321 DBP=200
PATNO=020 DBP=8
```

How would we go about computing cutoffs based on the middle 50% of our data? Calculating a mean and standard deviation on the middle 50% of the data (called trimmed statistics by robust statisticians) is easy if you first use PROC RANK (with a GROUPS= option) to identify quartiles and to use this information in a subsequent PROC MEANS step to compute the mean and standard deviation of the middle 50% of your data. Your decision on how many of these trimmed standard deviation units should be used to define outliers is somewhat of a trial and error process. Obviously (well, maybe not that obvious) the standard deviation computed on the middle 50% of your data will be smaller than the standard deviation computed from all of your data, even if your data are normally distributed. The difference between the two will be even larger if there are some dramatic outliers in your data. (We will demonstrate this later in this section.) As an approximation, if your data were normally distributed, the trimmed standard deviation is approximately 2.6 times smaller than the untrimmed value. So, if your original cutoff was plus or minus two standard deviations, you might choose 5 or 5.2 trimmed standard deviations as your cutoff scores. What follows is a program that computed trimmed statistics and uses them to identify outliers:

```
PROC RANK DATA=CLEAN.PATIENTS OUT=TMP GROUPS=4;
  VAR HR;
  RANKS R_HR;
RUN;

PROC MEANS DATA=TMP NOPRINT;
```

```

WHERE R_HR IN (1,2);  ***THE MIDDLE 50%;
VAR HR;
OUTPUT OUT=MEANS(DROP=_TYPE_ _FREQ_)
      MEAN=M_HR  STD=S_HR;
RUN;

DATA _NULL_;
  TITLE "OUTLIERS BASED ON TRIMMED SD";
  FILE PRINT;
  %LET N_SD = 5.25;
  SET CLEAN.PATIENTS;
  IF _N_ = 1 THEN SET MEANS;
  IF HR LT M_HR - &N_SD*S_HR AND HR NE .
  OR HR GT M_HR + &N_SD*S_HR THEN PUT PATNO=
      HR=;
RUN;

```

There is one slight complication here, compared to the earlier non-trimmed version of the program. The middle 50% of the observations can be different for each of the numeric variables you want to test. So, if you want to run the program for several variables, it would be convenient to assign the name of the numeric variable to be tested, to a macro variable. We will do this next, but first, a brief explanation of the program. PROC RANK is used with the GROUPS= options to create a new variable (R\_HR) which will have values of 0,1,2, or 3, depending on which quartile the value lies. Since we want both the original value for HR and the rank value, we use a RANKS statement which allows us to give a new name to the variable that will hold the rank of the variable listed in the VAR statement. All that is left to do, is to run PROC MEANS as we did before, except that a WHERE statement selects the middle 50% of the data values. What follows is the same as the earlier program, except that arrays are not needed since we can only process one variable at a time. Output from the program is shown below:

#### Outliers Based on Trimmed SD

```

PATNO=008 HR=210
PATNO=014 HR=22
PATNO=017 HR=208
PATNO=321 HR=900
PATNO=020 HR=10
PATNO=023 HR=22

```

Notice that the method based on a non-trimmed standard deviation reported only one HR as an outlier (PATNO=321, HR=900) while the method based on a trimmed mean identified six values. The reason? The heart rate value of 900 inflated the non-trimmed standard deviation so much that none of the other values fell within two standard deviations.

#### CONCLUSION

We have only seen the tip of the iceberg (the SAS® iceberg for those who saw the video years ago) with respect to data cleaning techniques. Tasks which we did not discuss here include checking for duplicate ID's, checking ID's across multiple files, counting of missing values, checking for invalid date values, and so forth. You are invited to purchase the soon to be release SAS book on data cleaning, by the author of this paper, for more, in depth coverage of this important topic.

#### PROGRAM TO CREATE THE SAS DATA SET PATIENTS

```

*-----*
| PROGRAM NAME: PATIENTS.SAS |
| PURPOSE: TO CREATE DATA SET PATIENTS |
| DATE: DECEMBER 17, 1998 |
*-----*

```

```

DATA PATIENTS;
INFILE "C:PATIENTS,TXT" PAD;
INPUT @1 PATNO $3.
      @4 GENDER $1.
      @5 VISIT MMDDYY10.
      @15 HR 3.
      @18 SBP 3.
      @21 DBP 3.
      @24 DX $3.
      @27 AE $1.;

LABEL PATNO = "PATIENT NUMBER"
      GENDER = "GENDER"
      VISIT = "VISIT DATE"
      HR = "HEART RATE"
      SBP = "SYSTOLIC BLOOD PRESSURE"
      DBP = "DIASTOLIC BLOODPRESSURE"
      DX = "DIAGNOSIS CODE"
      AE = "ADVERSE EVENT?";

FORMAT VISIT MMDDYY10.;
RUN;

```

#### DATA SET PATIENTS,TXT

```

001M11/11/1998 88140 80 10
002F11/13/1998 84120 78 X0
003X10/21/1998 68190100 31
004F01/01/1999101200120 5A
XX5M05/07/1998 68120 80 10
006 06/15/1999 72102 68 61
007M08/32/1998 88148102 0
008F08/08/1998210 70
009M09/25/1999 86240180 41
010F10/19/1999 40120 10
011M13/13/1998 68300 20 41
012M10/12/98 60122 74 0
013208/23/1999 74108 64 1
014M02/02/1999 22130 90 1
002F11/13/1998 84120 78 X0
003M11/12/1999 58112 74 0
015F 82148 88 31
017F04/05/1999208 84 20
019M06/07/1999 58118 70 0
123M15/12/1999 60 10
321F 900400200 51
020F99/99/9999 10 20 8 0
022M10/10/1999 48114 82 21
023F12/31/1998 22 34 78 0
024F11/09/199876 120 80 10
025M01/01/1999 74102 68 51
027FNOTAVAIL NA166106 70
028F03/28/1998 66150 90 30
029M05/15/1998 41
006F07/07/1999 82148 84 10

```

#### CONTACT INFORMATION

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

Ronald Cody  
 Robert Wood Johnson Medical School  
 Dept of Environmental and Community Medicine  
 675 Hoes Lane  
 Piscataway, NJ 08854  
 Work Phone: (732)235-4490  
 Fax: (732)235-4569  
 Email: CODY@UMDNJ.EDU