Paper 196-26

Fast and Effective Text Standardization (Coding) without the Latest "Stuff"

Erik H. Dilts, Rho Inc., Chapel Hill, North Carolina

Abstract

Computer equipment and software packages are being upgraded and revised at a remarkable rate, with each component driving the other. This makes many new things possible, and can make old things faster or otherwise more effective, but it can also make those old things obsolete. The cost of keeping even a small company's equipment "up-to-date" can be very high. Therefore, it is important to write applications that will be workable on existing hardware for as long as possible.

This paper demonstrates the concepts of a flexible coding system for adverse events, medications, or any other character data that requires standardization, and can be used on a low-end computer system running SAS® version 6.12. It is intended for an audience with some experience in application design.

Introduction

Text standardization, commonly called coding, is the process by which a set of "imperfect" data (e.g. adverse event fields) is match-merged with a dictionary data set of standard terms, i.e. the COSTART dictionary. Some terms in the data will have exact matches in the dictionary, but many will not. These non-exact matches then require human decision-making to select the proper match for a given character field. For example: a data field such as "HEDACHE" will not find a match among the standard terms in the dictionary file, but a person can recognize the misspelled word "HEADACHE" and match it with the corresponding dictionary record.

A varying amount of the process, based on the type and source of the data, can be done programmatically, but there will always be some data that must be matched by human decision. Therefore it is important for any coding system to make that part of the process as easy as possible. It makes sense to think of a coding program as two distinct parts: the automatic and the manual matching. Both of these parts can be made quite powerful, that is, they can be made to do more of the work or make the existing work easier, but they generally come at the expense of program speed due to equipment limitations. The coding application outlined here is designed to run at an acceptable speed on a 200 MHz system with 32 MB of memory while still providing a great deal of coding "power" to the user.

The application outlined was designed under SAS version 6.12 using Screen Control Language (SCL) for the automatic matching portion and SAS/AF® for the manual matching user interface.

Setup

Flexibility was a high priority design requirement of this application, so it was written with the ability to match any character field in one data set to any character field in another data set. This overall program flexibility initially results in an increase in the overall setup time for any given project, but the time decreases significantly as the application is used more. Most jobs will involve one of the common dictionaries such as MedDRA™, COSTART, WHOART, and ICD-9 which, once created, can be reused.

The majority of these dictionaries come in non-SAS (usually text-based) formats. These must be converted into SAS data sets for the application to reference them. The programs used to create these data sets can also be reused, usually with minor changes if any, when updates or revisions are received. This design allows the user the ability to reference specialized dictionaries for individual clients, for example, a particular subset of the COSTART dictionary.

The application requires several pieces of information before starting a project. The locations of the data set to be coded and the data set to be coded against are needed, as are the variable names from each that are to be used for matching. This data can be obtained in several ways.

The first, and most preferred, method is to create an SCL list (or data set) containing the necessary information for each variable or data set to be coded before the user starts. The application reads this list at startup and displays a popup menu based on the list, then reads in the four aforementioned values based upon the user's choice. This creates a testable, repeatable, and error-resistant platform on which to code any character field.

The second method allows the user greater flexibility, but with increased responsibility comes the increased possibility for error. The FILEDIALOG function gives the user a familiar, Windows-like dialog box in which to choose the data set to be coded. Following that, the VARLIST function brings up a list of the variables in the chosen data set so that the user may pick the variable to be coded. This process is then repeated for the dictionary data set.

Automatic Coding

This aspect of coding systems probably varies the most from application to application. How much it varies is also often unknowingly based on the capabilities of the programmer's system, which may not match the user's. The number one variant is the close-match mapping technique.

The most common technique used is called "fuzzy" matching. This is generally done with various "word-crunching" algorithms, such as the SOUNDEX function included with SAS. The problem with fuzzy matching is that it requires a tremendous amount of system resources. Running a few hundred non-exact-matching terms against a dictionary of several thousand terms using fuzzy matching takes a significant amount of time on a fast computer, and is generally totally unacceptable from the user's point of view on a system that is even just two or three years old. If fuzzy matching is implemented, it should always be an option that can be turned off by the user

if it is not needed or runs too slowly on their equipment.

The second variant in coding application automatic mapping is the match-merge technique used for exact matches. One possibility is the data step merge shown in Example 1. This method requires extensive pre-processing, including making a temporary copy of a large data set and time-consuming sorting. Another possible method is SQL, demonstrated in Example 2. Using SQL for merging data sets has several advantages. Since the data sets do not need to be sorted, they can be used without making temporary copies. Furthermore, the variables to be matched are not required to have the same name.

Example 1:

```
data dict (rename=(drgname=med));
  set drug.dict;
run;

proc sort data=dict; by med;
proc sort data=sugi.testdata out=test; by med;

data merge_1;
  merge dict (in=ina)
        test (in=inb);
  by med;
  if ina and inb;
run;
```

```
4    data dict (rename=(drgname=med));
5      set drug.dict;
6    run;

NOTE: The data set WORK.DICT has 63364 observations and 4 variables.
NOTE: The DATA statement used 0.76 seconds.


7
8    proc sort data=dict; by med;

NOTE: The data set WORK.DICT has 63364 observations and 4 variables.
NOTE: The PROCEDURE SORT used 0.98 seconds.


9    proc sort data=sugi.testdata out=test; by med;
10

NOTE: The data set WORK.TEST has 44 observations and 1 variables.
NOTE: The PROCEDURE SORT used 0.0 seconds.


11   data merge_1;
```

```
12     merge dict (in=ina)
13           test (in=inb);
14     by med;
15     if ina and inb;
16   run;

NOTE: The data set WORK.MERGE_1 has 66 observations and 4 variables.
NOTE: The DATA statement used 0.28 seconds.
```

Example 2:

```
proc sql;
    create table merge_2 as
      SELECT * FROM
        drug.dict as dict
      RIGHT JOIN
        sugi.testdata as test
      ON dict.drgname
      =  test.med;
quit;

data merge_2 (drop=drgname);
  set merge_2;
run;
```

```
23   proc sql;
24       create table merge_2 as
25         SELECT * FROM
26           dict
27         RIGHT JOIN
28           sugi.testdata as test
29         ON dict.drgname
30         =  test.med;
NOTE: Table WORK.MERGE_2 created, with 66 rows and 5 columns.

31
32   quit;
NOTE: The PROCEDURE SQL used 1.26 seconds.


33
34   data merge_2 (drop=drgname);
35     set merge_2;
36   run;

NOTE: The data set WORK.MERGE_2 has 66 observations and 4 variables.
NOTE: The DATA statement used 0.05 seconds.
```

For both examples, the data set TEST contains forty-four records, all of which map to at least one term in the DICT data set, which contains 63,364 records. The tests were run on the following setups: a new system with an 800 MHz processor and 128 MB memory as the fast hardware and an approximately 4 year-old system with a 200 MHz processor and 32 MB memory as the slow hardware. Remember: since the test data set only contains forty-four observations times would be much higher under production usage.

Results:
Data Step Merge (Example 1):
Fast hardware: 2.02 seconds
Slow hardware: 40.01 seconds

Proc SQL Merge (Example 2):
Fast hardware: 1.31 seconds
Slow hardware: 11.55 seconds

The advantages are obvious. The data step merge, by itself, is faster than the SQL procedure in this instance, but the preparation necessary for that method negates its advantage. The only extra step the SQL method needs is for the dictionary-matching variable to be dropped. As a very simple step usually performed locally, the time it adds is negligible. There are other methods available to accomplish this task, but for coding, the SQL procedure is usually the most effective choice. Note that it also "scales down" to the slower hardware much better than the data step merge.

Manual Coding

Automatics coding usually only maps exact matches. The exceptions are those applications that implement fuzzy matching, which adds an extra dimension to manual coding: the verification of possible matches. As this paper is geared toward lower-end systems, this will not be discussed further here. Since there are often many terms left to be coded after the computer has had its turn, it is necessary to make the manual coding process as fast and convenient as possible while minimizing the possibility of errors. There are two components in SAS/AF that work splendidly for this.

The first is the INDEX function, which, when combined with the Extended Input Field and the popmenu function, provides the capability for a fast and powerful partial string search engine. The application creates an SCL list containing all the unique values of the matching variable in the dictionary data set when the manual coding screen is instantiated. Then, when the search field is activated, a labeled section of code in the frame's code runs with the same name as the name of the field, which in Example 3 is WHERESTR.

Example 3:

```
WHERESTR:
  * Make sublist of valid terms containing entered string;
  cdlist=makelist();
  length=listlen(cdlist);
  do i = 1 to length;
    string = getitemc(cdlist,i);
    x = index(upcase(string),upcase(wherestr));
    if x > 0 then rc = insertc (cdlist,string,-1);
  end;
  if listlen(cdlist)=0 then do;
    _msg_= "No match found.";
  end;

  else do;
    * Pop up term list;
    rc = popmenu(cdlist);
    if rc > 0 then do;
      p=rc;  *p is for list position;
      choice = getitemc(cdlist,p);
      rc = dellist(cdlist);
    end;
  end;
RETURN;
```

This makes the field behave as follows: when the user enters any text and presses Enter, a search is performed which attempts to find that string in each dictionary term, regardless of its position in that term. Thus, if the user enters "CILLIN", a list pops up with all dictionary terms that contain that string, from "BACAMPICILLIN" to "PROCAINE PENICILLIN G" and many other variants, which is a most effective way of shortening the list of possible choices. If the user chooses an item from the list, the selection is stored in the variable **choice** in Example 3.

The second tool to help the user is the Extended Text Entry Field with its FEEDBACK method overridden as shown in Example 4.

Example 4:

```
* Remove warnings;
_self_=_self_; _frame_=_frame_;

* Set up program variable;
length TXT $ 200;
FEEDBACK: method event $ 20 line offset 8;
  * Overrides the _FEEDBACK_ method;
  call send (_self_,'_GET_TEXT_',TXT);
  len = length(TXT);
  if len = 0 then return;
if (event = "RETURN") then
    call send (_self_,'_SEND_EVENT_','feedback return',TXT);
if (event ^= "PRINTABLE") then return;
```

```
* Search through list of codes;
rc=searchc(cdlist,TXT,1,1,'y','y');
  if (rc > 0) then do;
    TXT = getitemc(cdlist, rc);
    call send(_SELF_,"_SET_TEXT_",TXT);
    call send(_SELF_,"_SET_SELECT_", 1, len+1, 1, 200);
  end;
endmethod;
```

This creates a field that fills itself in as the user types based on the available choices from the dictionary, contained in the previously mentioned SCL list. For example, in coding a term such as "CAFFEINE", the user would type "C" and the first dictionary term that begins with "C" would appear in the field, then the first that begins with "CA" and so on. In this case the user would only have to type "CAFFE" before the correct term pops up, at which time they would press the Enter key to select the term as it is displayed in the field. This may not seem like much, but it speeds up the process a great deal, in addition to requiring less repetitive typing on the user's part.

Conclusion

Most companies cannot afford to upgrade each time the latest software release or a faster processor comes out. While applications written to take advantage of the latest tools might be very impressive, they remain out of the reach of many users because of the "obsolescence" of their system setups. This paper defines the concepts behind a character field standardization (coding) system that maintains good functionality while still being practical for users with older systems. While it defines one specific application, its underlying theme is that it is important to keep these users in mind when writing programs for SAS or any other platform.

Author Contact

Erik H. Dilts
Rho, Inc.
100 Eastowne Drive
Chapel Hill, NC 27514
Dilts@rhoworld.com