Paper 122-2012

Matching Data Using Sounds-Like Operators and SAS® Compare Functions

Amanda Roesch, Educational Testing Service, Princeton, NJ

ABSTRACT

By combining both sounds-like operators and compare functions, SAS can quickly identify many intended matches between almost any two strings (I say *almost* because some typos and spelling mistakes are too drastic to even be caught by the human eye, let alone a computer program relying on logic). First, using the sounds-like (either SOUNDEX or =*) operator, SAS will pair every match that sounds even relatively close. Yes this will result in numerous pairings, but that's where compare functions are useful. They can parse the possibilities down for further evaluation. The compare functions that will be discussed here include COMPARE, COMPGED and CALL COMPCOST, COMPLEV, and SPEDIS. Finally, using a sort and a well-devised cutoff value, SAS will give a final list of the most likely matches. This method not only increases the prospective matches, it uses logic and reason rather than manipulating the data itself.

INTRODUCTION

Often times, SAS® programmers find themselves faced with what most programmers dread the most: the task of matching the unmatchable, the inconsistent text string. For instance, since computers don't understand that 'McHugh' and 'Mc Hugh' are most likely referring to the same name, it is necessary to go into SAS's bag of tricks and use things such as SUBSTR, COMPRESS, and TRANSWRD functions. The problem is, unless it is a relatively simple difference, a programmer usually has to know exactly what needs to be done to make the two words pair together. Take for example, 'Sesame' and 'Sesmae'. Most likely a typo, but how can SAS be manipulated into matching these two together? The answer lies within sounds-like operators and compare functions.

POSSIBLE MATCHES

Let us start from the beginning. You are given your data and asked to match candidates from one set of data (Name) to another set of data (Birthday). The matching fields in these two sets of data are First Name and Last Name. You find that one of these datasets contains many typos in both fields and it is rare to find a perfect match. So what is your first step? You will use a sounds-like merge to gather all of the possible matches.

SAMPLE DATA

Two SAS data sets will be used in this paper. WORK.Name contains two character variables, First and Last. WORK.Birthday contains three character variables, First_B, Last_B, and Birthday. Note that Last and Last_B both refer to last names, but the variables are modified to match their respective data set. This is also the case for first name.

* * * * * * * * * * * * * *

WORK.Name

OBS	FIRST	LAST
1	George	Washington
2	Thomas	Jefferson

WORK.Birthday

OBS	FIRST_B	LAST_B	BIRTHDAY
1	Goerge	Washnigton	02/22/1732
2	greg	Wa.sh	08/09/1921
3	Thmas	Jefrson	04/13/1743
4	Thomnas	Wasshington	05/13/1961
5	TANK	JEEPERS	03/27/1945

Notice how WORK.Birthday has strings that resemble those in WORK.Name, but there are discrepancies. Also in WORK.Birthday are strings that, while similar to those in WORK.Name, are obviously not correct matches. This paper will show how to identify the match of the WORK.Name observations from the WORK.Birthday data set.

PROC SQL AND THE SOUNDS-LIKE OPERATOR

One method on how to accomplish this is to use a many-to-many merge in PROC SQL, with the criteria that one variable must sound-like another. It is important that the variables to be matched have different names, so the compare functions can be applied in the next step. When matching candidate records together, I find that it is best to start by matching on the first name and last name. Even if more compatible variables are available, it is usually best to start simple. For example:

```
PROC SQL;
  CREATE TABLE Possible_Matches AS
  SELECT *
  FROM Name AS n, Birthday AS b
  WHERE (n.Last =* b.Last_B OR n.First =* b.First_B);
QUIT;
```

OBS	FIRST	LAST	FIRST_B	LAST_B	BIRTHDAY
1	George	Washington	Goerge	Washnigton	02/22/1732
2	George	Washington	greg	Wa.sh	08/09/1921
3	George	Washington	Thomnas	Wasshington	05/13/1961
4	Thomas	Jefferson	Thmas	Jefrson	04/13/1743
5	Thomas	Jefferson	Thomnas	Wasshington	05/13/1961
6	Thomas	Jefferson	TANK	JEEPERS	03/27/1945

Now our dataset 'Possible_Matches' will contain all of the possible matches if a candidate spelled either their first or last name incorrectly (or neither). The 'n.Last =* b.Last_B' expression is the equivalent to coding 'SOUNDEX(n.Last)=SOUNDEX(b.Last_B)'. Notice the potential matches; if either the last name or the first name was similar than they were listed as a possible match.

THE SOUNDEX OPERATOR

To see how the strings are being matched we can create a variable to hold the SOUNDEX equivalent of the variable we plan to merge on for each dataset. For example:

```
DATA Birthday;
  SET Birthday;
  Soundex_Last=SOUNDEX(Last_B);
  Soundex_First=SOUNDEX(First_B);
RUN;
```

WORK.Birthday

OBS	FIRST_B	LAST_B	BIRTHDAY	SOUNDEX	
				FIRST	LAST
1	Goerge	Washnigton	02/22/1732	G62	W25235
2	greg	Wa.sh	08/09/1921	G62	W2
3	Thmas	Jefrson	04/13/1743	T52	J1625
4	Thomnas	Wasshington	05/13/1961	T52	W25235
5	TANK	JEEPERS	03/27/1945	T52	J162

Notice that some strings were assigned the same code as other strings, although they are different such as ‘George’ and ‘greg’. Also notice that, while Observation 5 has a leading blank, the code still begins with the first letter. Each string is given a code, based on the American SoundEX formula patented by Robert C. Russell in 1918.

SOUNDEX FORMULA

The American SoundEX converts each word to a code. This code consists of the first letter in the word and then 3 digits, designated by key letters and their corresponding value. The values are:

Letter	Value
B, P, F, V	1
C, S, G, J, K, Q, X, Z	2
D, T	3
L	4
M, N	5
R	6

The letters A, E, I, O, U, Y, H, and W along with any non-alpha characters are not coded. Double letters (‘DD’) will only be coded once for both letters. Any two letters that have the same code and are next to each other or separated by a ‘W’ or ‘H’ will also be coded as a single number (ex. FLASHCARD would be coded as F426, since SHC would only be assigned a 2). The only time two sequential digits of a code are the same will be if two letters with the same code are separated by a vowel or space. The code cuts off at three digits, but if a word does not have three significant letters, than 0s are appended (ex. Daniel is D540). However, when SAS introduced this operator in version 6.07, it used a variation of this conversion. SAS does not cut off the code at three digits, it uses as many digits as a word requires. Similarly, it does not append 0s. Therefore, the codes designated by SAS will vary in length, whereas the ones assigned by American SoundEX will always be a length of four. Also, SAS considers the letters ‘W’ and ‘H’ to count as vowels whereas the American version just ignores them completely (ex. FLASHCARD will now be coded as F42263). And this is how SAS decides if one thing sounds like another, explaining how our dataset of Possible Matches came to be. For example, WASHINGTON is assigned a value of W25235. The W is the first letter, A is skipped, S=2, H is skipped, I is skipped, N=5, G=2, T=3, O is skipped, and N=5. This will be matched with any other last name that is assigned the same code. As can be seen from the list of possible matched, there are quite a few that are clearly not referring to the same name. But now that we have all of our possible matches, how do we separate the good ones from the bad?

EVALUATING THE POSSIBILITIES

The answer lies within compare functions. We can use compare functions to determine how close two things really are to one another. Compare functions look at the actual word, not at codes that can be devised from it. This is more specific and quite a bit harsher than the SoundEX operator, which is exactly what we need to parse down our pairs. SAS has developed several of these functions, which can be used in various situations. How will we decide which one to use? Or should we use a combination of the functions? A better understanding of the functions is essential to recognizing which one(s) will suit the data best.

SPEDIS FUNCTION

The first of these offered by SAS is the SPEDIS function, standing for Spelling Distance. This was introduced by SAS in version 6.12. SPEDIS accepts two arguments, a query and a keyword, and returns the 'spelling distance' required to convert the keyword into the query. This is coded as SPEDIS (query, keyword). The query is the word that the keyword will be manipulated to match. 'Spelling Distance' can be defined as the cost of operations used in the conversion process. These operations include character insertion, deletion, and replacement. Different 'costs' are associated with each one of these operations, as shown in the table below.

Operation	Cost	Explanation
Match	0	No change
Singlet	25	Delete one of a double letter
Doublet	50	Double a letter
Swap	50	Reverse the order of two consecutive letters
Truncate	50	Delete a letter from the end
Append	35	Add a letter to the end
Delete	50	Delete a letter from the middle
Insert	100	Insert a letter in the middle
Replace	100	Replace a letter in the middle
FirstDel	100	Delete the first letter
FirstIns	200	Insert a letter at the beginning
FirstRep	200	Replace the first letter

These operations' costs are summed and then divided by the length of the query to represent the spelling distance. SPEDIS evaluates all of the possible ways to translate the keyword into the query and then returns the smallest possible value (always rounded down to a whole number). The function evaluates many possible scenarios before returning a value, making it the slowest option of the ones we will list here. Since the keyword is being translated into the query and divided by the length of the query, SPEDIS (A, B) does not necessarily equal SPEDIS (B, A). Example:

```
DATA Spedis;
  SET Possible_Matches;
  First_SPEDIS=SPEDIS(First, First_B);
  Last_SPEDIS=SPEDIS(Last, Last_B);
RUN;
```

****

OBS	FIRST	LAST	FIRST_B	LAST_B	BIRTHDAY	SPEDIS	
						FIRST	LAST
1	George	Washington	Goerge	Washnigton	02/22/1732	8	5
2	George	Washington	greg	Wa.sh	08/09/1921	75	32
3	George	Washington	Thomnas	Wasshington	05/13/1961	108	2
4	Thomas	Jefferson	Thmas	Jefrson	04/13/1743	16	16
5	Thomas	Jefferson	Thomnas	Wasshington	05/13/1961	8	100
6	Thomas	Jefferson	TANK	JEEPERS	03/27/1945	89	92

A matching name requires 0 operations and therefore receives a value of 0. Above you can see that for the relatively simple 'George' and 'Goerge', the cost is 50 for the swap of the 'e' and the 'o', making the distance 50/6=8.3333, which rounds to 8 by the SPEDIS rule. It can be seen that SPEDIS(First, First_B) does not equal SPEDIS(First_B, First) specifically in observations 4 and 5. One requires an inserted character and one requires a deleted character, which are inverse operations. Since deleting and inserting are different costs, the two have different spelling distances. Also, if the length of the two strings is different, when the distance is calculated the two costs would be averaged over a different number of characters since they are only divided by the length of the query.

COMPARE FUNCTION

SAS next established the compare functions beginning with version 9. The first of these is simply COMPARE (string1, string2 <, modifiers>). COMPARE takes the two strings and returns the position of the first character that is different in them. The value will be negative if string1 is alphabetically or numerically ordered before string2 and will be positive if the situation is reversed. If the two strings are switched, than the result will be the negative of previous result, COMPARE (A, B) = -COMPARE (B, A). If the strings match exactly, than the returned value will be zero. The modifiers, which can be used in any of the compare functions, are as follows:

Modifier	Explanation
i or I	Ignores the case in ***string1*** and ***string2***.
l or L	Removes leading blanks in ***string1*** and ***string2*** before comparing the values.
n or N	Removes quotation marks from any argument that is an n-literal and ignores the case of ***string1*** and ***string2***.
: (colon)	Truncates the longer of ***string1*** or ***string2*** to the length of the shorter string, or to one, whichever is greater.

These can be used in conjunction with one another and will be applied in the order that they are given. Using the modifier :N will first truncate the longer string and then remove quotation marks. Trailing blanks are stripped before processing. It is important to note that the sounds-like function cannot pair together a name surrounded by quotes with one that is not. Example:

```
DATA Compare;
  SET Possible_Matches;
  First_COMPARE=COMPARE(First, First_B, ':iln');
  Last_COMPARE=COMPARE(Last, Last_B);
RUN;
```

****

OBS	FIRST	LAST	FIRST_B	LAST_B	BIRTHDAY	COMPARE	
						FIRST	LAST
1	George	Washington	Goerge	Washnigton	02/22/1732	-2	-5
2	George	Washington	greg	Wa.sh	08/09/1921	-2	3
3	George	Washington	Thomnas	Wasshington	05/13/1961	-1	-4
4	Thomas	Jefferson	Thmas	Jefrson	04/13/1743	3	-4
5	Thomas	Jefferson	Thomnas	Wasshington	05/13/1961	-5	-1
6	Thomas	Jefferson	TANK	JEEPERS	03/27/1945	2	1

Take a look at observation 6. Both names from the Birthday data set begin with a blank. However, since First_COMPARE used the modifier to ignore blanks, the value is 2 because the second letter ‘a’ from First_B does not match the second letter (but third character) ‘h’ from First. Last_COMPARE is 1 since ‘T’ in Last_B is not a blank as it is in Last. The values are positive in both because alphabetically the differences from the data set Birthday come first (‘A’ comes before ‘h’; blank comes before ‘J’).

COMPGED FUNCTION

The next compare function is COMPGED, meaning Generalized Edit Distance. The generalized edit distance is the measure of differentiation between two strings. The syntax is COMPGED (string1, string2 <, cutoff> <, modifiers>). The function returns the ‘cost’ of converting the first string into the second string. Similarly to the SPEDIS function, this cost is determined by taking all of the possible ways to match string1 to string2 and evaluating which one would have the lowest cost. There is also an option to set a cutoff, if the COMPGED is greater than or equal to the cutoff, than the value of the cutoff is returned. The modifiers are the same as listed above for the COMPARE function.

****

Also similarly to the SPEDIS function, COMPGED also assigns a cost to each operation required to alter string1 into string2. However, the costs are not the same.

Operation	Cost	Explanation
APPEND	50	When the output string is longer than the input string, add any one character to the end of the output string without moving the pointer.
BLANK	10	Do any of the following: • Add one space character to the end of the output string without moving the pointer. • When the character at the pointer is a space character, advance the pointer by one position without changing the output string. • When the character at the pointer is a space character, add one space character to the end of the output string, and advance the pointer by one position. If the cost for BLANK is set to zero by the COMPCOST function, the COMPGED function removes all space characters from both strings before doing the comparison.
DELETE	100	Advance the pointer by one position without changing the output string.
DOUBLE	20	Add the character at the pointer to the end of the output string without moving the pointer.
FDELETE	200	When the output string is empty, advance the pointer by one position without changing the output string.
FINSERT	200	When the pointer is in position one, add any one character to the end of the output string without moving the pointer.
FREPLACE	200	When the pointer is in position one and the output string is empty, add any one character to the end of the output string, and advance the pointer by one position.
INSERT	100	Add any one character to the end of the output string without moving the pointer.
MATCH	0	Copy the character at the pointer from the input string to the end of the output string, and advance the pointer by one position.
PUNCTUATION	30	Do any of the following: • Add one punctuation character to the end of the output string without moving the pointer. • When the character at the pointer is a punctuation character, advance the pointer by one position without changing the output string. • When the character at the pointer is a punctuation character, add one punctuation character to the end of the output string, and advance the pointer by one position. If the cost for PUNCTUATION is set to zero by the COMPCOST function, the COMPGED function removes all punctuation characters from both strings before doing the comparison.
REPLACE	100	Add any one character to the end of the output string, and advance the pointer by one position.
SINGLE	20	When the character at the pointer is the same as the character that follows in the input string, advance the pointer by one position without changing the output string.
SWAP	20	Copy the character that follows the pointer from the input string to the output string. Then copy the character at the pointer from the input string to the output string. Advance the pointer two positions.
TRUNCATE	10	When the output string is shorter than the input string, advance the pointer by one position without changing the output string.

CALL COMPCOST

These costs can be altered using the CALL COMPCOST function. The structure of this function is CALL COMPCOST ('operation1=', value1 <, 'operation2=', value2, ….>). The operation is one of the operations listed above that you would like to modify the cost of. The value is the new price of the operation; it must be greater than -32767 and less than 32767. The value may also be set to missing, which will set it to its default value. A value of zero will effectively disable the operation. You can list as many as you like within one function. The costs set by this function will be effective until either the data step ends or another CALL COMPCOST is called and changes the operations. The operations can also be abbreviated, for example 'DELETE=' can be 'DEL=' and 'INSERT=' can be 'I=' or 'INS='. This ability to change the values of the operations makes the COMPGED function the most versatile, and probably the best to control for specific situations. However, while it does not take as long to run as the SPEDIS function, it does take longer than the COMPARE function or the COMPLEV function. Example:

```
DATA Compged;
  SET Possible_Matches;
  CALL COMPCOST('SWAP=', 5, 'P=', 0);
  First_COMPGED=COMPGED(First, First_B, ':iln');
  Last_COMPGED=COMPGED(Last, Last_B);
RUN;
```

OBS	FIRST	LAST	FIRST_B	LAST_B	BIRTHDAY	COMPGED	
						FIRST	LAST
1	George	Washington	Goerge	Washnigton	02/22/1732	5	5
2	George	Washington	greg	Wa.sh	08/09/1921	300	600
3	George	Washington	Thomnas	Wasshington	05/13/1961	500	100
4	Thomas	Jefferson	Thmas	Jefrson	04/13/1743	200	200
5	Thomas	Jefferson	Thomnas	Wasshington	05/13/1961	200	900
6	Thomas	Jefferson	TANK	JEEPERS	03/27/1945	300	500

Notice that the values are typically much higher than the SPEDIS ones; this is because the sum of the costs is not averaged over the length of the string. In this example, the cost of SWAP has been changed from 20 to 5 since typos of this sort are relatively common. Now, 'George' and 'Goerge' only has a COMPGED of 5. Also, observation 6 demonstrates the significance of the case modifier. Although Last_B has more characters in common with Last than First_B does with First, because of the case ignorer the value is higher which indicates a worse match. This is due to the fact that every uppercase letter had to be replaced with a lowercase one, which is the same cost as replacing a character with any other character. The value of 'P', for PUNCTUATION, has been set to 0 which disregards the operation. Therefore, in observation 2, Last_COMPGED is just the cost of inserting 6 characters onto Last_B ('ington').

****

COMPLEV FUNCTION

The COMPLEV function is a variation of the COMPGED function designed to be simpler and execute faster. It stands for the Levenshtein Edit Distance. It was also introduced in version 9 of SAS and has the same format as the COMPGED function, COMPLEV (string1, string2 <, cutoff > <, modifiers>). It performs just as COMPGED does, with one exception. Instead of assigning a cost to each operation, it merely counts the number of operations that are done. So if there was a swap and a replacement done, then a value of two would be assigned versus the 120 for COMPGED, using default values. If a truncate and punctuation were done, than that would also receive a value of two versus the 40 for COMPGED, using default values. It is a straightforward way of seeing how much two strings may differ by, but it is both less versatile and less explanatory. CALL COMPCOST is not available to change the costs of the operations (yes, they all count as 1, but you are not able to disable costs that you do not want to count). Also, a simple truncatation counts the same as a more invasive replace, so it will not be as clear how different the two strings actually are. Example:

```
DATA Complev;
  SET Possible_Matches;
  First_COMPLEV=COMPLEV(First, First_B, ':iln');
  Last_COMPLEV=COMPLEV(Last, Last_B);
RUN;
```

OBS	FIRST	LAST	FIRST_B	LAST_B	BIRTHDAY	COMPLEV	
						FIRST	LAST
1	George	Washington	Goerge	Washnigton	02/22/1732	2	2
2	George	Washington	greg	Wa.sh	08/09/1921	3	7
3	George	Washington	Thomnas	Wasshington	05/13/1961	5	1
4	Thomas	Jefferson	Thmas	Jefrson	04/13/1743	2	2
5	Thomas	Jefferson	Thomnas	Wasshington	05/13/1961	2	9
6	Thomas	Jefferson	TANK	JEEPERS	03/27/1945	3	9

As can be seen in the data table, COMPLEV gives an idea of how close two strings are as SPEDIS and COMPGED did, but it is not as descriptive. In observation 1, Last_Compare received a value of 2 when comparing ‘Washington’ with ‘Washnigton’. For the COMPGED example, it was decided that a SWAP was a minimal cost procedure. However, COMPLEV does not allow you to change costs, and therefore is awarded the same value as observation 4, where ‘Jefrson’ is missing two letters when compared to ‘Jefferson’. SPEDIS also has a SWAP valued at less than a DOUBLE and INSERT, the simplest procedure used to adjust observation 4.

COMPARING THE COMPARES

Now that the compare functions have been explained and demonstrated, let’s take a look at how they compare with one another. COMPLEV is the fastest of all of the options, COMPGED is the most flexible, SPEDIS averages the costs over the length of the string, and COMPARE is the simplest. COMPLEV is useful for a large amount of data where you want to count every operation done, no matter what it is. A disadvantage is the inability to not count an operation, such as a truncate or punctuation. COMPGED is useful for very specific comparing, where the CALL COMPCOST function can alter the operation costs to better suit the needs of the data. Both of the COMP functions also allow the modifiers, making them even more versatile. SPEDIS is useful for comparing large strings, so if a string of length 250 is just missing a comma, the cost will be low. COMPARE is useful to find where the first difference occurs, and not really appropriate for matching. There is also the option of combining the functions. If you would like to see how costly the operations performed are, you could combine a COMPGED and COMPLEV. Or if you need to see how different two strings are, but also want to know where the first difference occurs so that you may correct it, SPEDIS and COMPARE can be used. It all depends on the data and the desired result.

PARSING DOWN TO THE PROBABLE MATCHES

Once the data has been assigned the proper comparison values, the obvious non-matches should be removed. There are several ways to do this. One way is to set a threshold and eliminate any matches above said threshold using a simple WHERE= or DATA step. Another way is to sort the data by the original string and then use a 'BY' statement in conjunction with a 'FIRST.' to only take the best match. (This method must be used with caution in case of multiple 'best' matches that have the same value since it will just take the first one in the data set.) For example:

```
DATA Compare_Matches;
  SET Compged;
  Total_Compare=First_Compged+Last_Compged;
RUN;

PROC SORT DATA= Compare_Matches;
  BY Last First Total_Compare;
RUN;

DATA Final_Matches;
  SET Compare_Matches;
  BY Last First;
  IF FIRST.First;
RUN;
```

OBS	FIRST	LAST	FIRST_B	LAST_B	BIRTHDAY
1	George	Washington	Goerge	Washnigton	02/22/1732
2	Thomas	Jefferson	Thmas	Jefrson	04/13/1743

Which are most likely the best matches between the two data sets. 'Thomas' is only missing the 'o', and 'Jefrson' is missing the 'fe'. 'Goerge' and 'Washnigton' just have two letters switched, a common typo. Looking at the other possibilities in the data set, it is safe to assume that we have achieved the desired results.

CONCLUSIONS

Each of the compare functions has advantages and disadvantages. The one (or ones) to use will depend on both the data and the type of match you are looking for. For the sample data that has been shown in this paper, COMPLEV or COMPGED are the ideal functions to use. While there is never 100% chance of success in matching, hopefully this is just one more tool that will help you get there.

****

REFERENCES

CALL COMPCOST Routine
http://support.sas.com/documentation/cdl/en/lrdict/64316/HTML/default/viewer.htm#a002206135.htm

COMPARE Function
http://support.sas.com/documentation/cdl/en/lrdict/64316/HTML/default/viewer.htm#a002206130.htm

COMPGED Function
http://support.sas.com/documentation/cdl/en/lrdict/64316/HTML/default/viewer.htm#a002206133.htm

COMPLEV Function
http://support.sas.com/documentation/cdl/en/lrdict/64316/HTML/default/viewer.htm#a002206137.htm

Fan, Zizhong. 2004. *Matching Character Variables by Sound: A closer look at SOUNDEX function and Sounds-Like Operator (=*).* Rockville, MD: Westat.

SPEDIS Function
http://support.sas.com/documentation/cdl/en/lrdict/64316/HTML/default/viewer.htm#a000245949.htm

Staum, Paulette. 2007. *Fuzzy Matching using the COMPGED Function.* West Nyack, NY: Paul Waldron Consulting.

ACKNOWLEDGEMENTS

CONTACT INFORMATION

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

Amanda Roesch
Educational Testing Service
Princeton, NJ 08540
Phone: (609) 734 - 5511
Email: aroesch@ets.org

****