Fuzzy Matching using the COMPGED Function
Paulette Staum, Paul Waldron Consulting, West Nyack, NY

ABSTRACT
Matching data sources based on imprecise text identifiers is much easier if you use the COMPGED function. COMPGED computes a “generalized edit distance” that summarizes the degree of difference between two text strings. Several additional techniques can be helpful. CALL COMPCOST can adjust the weighting algorithm used by COMPGED. Removing commonplace words and adjusting for the effect of different text string lengths can improve results. Also, eliminating implausible potential matches can improve efficiency.

INTRODUCTION
Sometimes data about the same person, business, or other entity is available from multiple sources. How do you combine information from different sources? In an ideal world (for programmers, if not for individualists), all data sources would share a common unique identifier for each person, business, or other entity. This might be something like a Social Security Number.

Often there is no unique identifier. Combining different data sources must be done on the basis of names, addresses or other identifiers. These identifiers will not always match, even when they refer to the same individual or entity. This is a standard problem often called “fuzzy matching”. It is frequently used to do “fuzzy merging” of two data sources. The essence of the standard solution is to compute a distance based on the differences between the two identifiers. If the distance is small, then the two identifiers are assumed to be substantially the same. The same techniques can be used to look for approximate matches within one data source.

Often, the matching process needs to be adjusted or tuned based on the specific characteristics of the text to be matched. This can be complicated and time-consuming. If you or your organization have the resources, consider using a commercial package intended to solve these kinds of problems. Alternatives include SAS products like Data Integration and Text Mining, as well as products from other vendors of fuzzy matching products. If you do not have access to this kind of software, this paper introduces some methods for building your own application and tuning it for specific tasks.

SAS has a number of functions designed to compare text strings and determine the degree of their similarity or difference. If some of the identifiers are numbers or dates, they can be compared after they are converted to text strings by using the PUT or PUTN function. If you know exactly how the text strings should be related, you might be able to use the Perl regular expression (PRX) functions and CALL routines to evaluate the distance between the strings. For more general problems, you can use SAS functions like COMPGED, COMPLEV, SPEDIS, SOUNDEX and its counterpart, the sounds-like operator (=*).

Which function is best for fuzzy matching? SAS documentation characterizes the strengths and weaknesses of these functions. COMPGED computes a “generalized edit distance” summarizing the degree of difference between two text strings. COMPLEV is faster, but less comprehensive in its evaluation and not as generally useful for text string matching. SPEDIS is slower. SOUNDEX and =* are primarily intended for English language text strings and encode the sound a text string represents. Based on these characteristics and some experimentation, COMPGED is the best function for fuzzy matching.

CALCULATE DISTANCE WITH COMPGED
To calculate the generalized edit distance between two text strings, the COMPGED function converts the first string into the second string. This is similar to the word game in which you try to change one word into another word by changing one character at a time. COMPGED moves through the first string one character at a time, trying to build the second string. It might replace one character in the first string with another, it might insert a new character in the second string, or skip a character in the first string. It assigns costs to each operation (insertion, deletion, replacement) and adds them up step by step. It chooses the lowest cost method of building the second string from the first string. This determines the distance between the two strings.
COMPGED has 2 required arguments and 2 optional arguments.

- The first two required arguments are the text strings to compare.
- The third argument is an optional cutoff value. If the 2 input strings are very dissimilar, you might not care how dissimilar they are. To make the comparison process much more efficient, comparisons are stopped whenever the distance is above the cutoff.
- The fourth argument is an optional string of modifiers. Among other things, you can use "i" to ignore case differences and 'L' to remove leading blanks.

Here are some examples of how to use COMPGED and the results it produces.

* 1: If the strings are identical, then the COMPGED output distance is zero.
  ```sas
  data _null_;  
  SameScore = compged('My story is absolutely consistent', 'My story is absolutely consistent');  
  put SameScore=;  
  run;  
  SameScore=0
  ```

* 2: A higher distance indicates more difference than a lower distance.
  ```sas
  data _null_;  
  ShortScore = compged('the story ','the history');  
  LongScoreSameDiff = compged('the story of my life','the history of my life');  
  LongScoreMoreDiff = compged('the story of my life','the history of his life');  
  put ShortScore= LongScoreSameDiff= LongScoreMoreDiff=;  
  run;  
  ShortScore=200 LongScoreSameDiff=200 LongScoreMoreDiff=500
  ```

* 3: The generalized edit distance is not symmetric by default.
  ```sas
  data _null_;  
  ForwardScore=compged('the story goes on','the story');  
  BackwardScore=compged('the story','the story goes on');  
  put ForwardScore= BackwardScore=;  
  run;  
  ForwardScore=320 BackwardScore=80
  ```

CALCULATE DISTANCES FOR ALL PAIRS

What is the best way to compare thousands of text strings stored in two data sets? To find fuzzy matches between two data sets, you need to compare every value of the key variable in one data set with every value of the key variable in another data set. This can be done by creating a cross-product of the two data sets. Suppose one data set has key values equal to A, B, C and the other has the key values equal to X, Y, Z. The cross-product will have the combinations A:X, A:Y, A:Z, B:X, B:Y, B:Z, C:X, C:Y, C:Z. Note that as the number of observations in the input data set increases, the number of observations in the output data set increases even more rapidly.

Let’s look at an example. Using realistic names and addresses as sample data might raise confidentiality issues. To avoid this problem, and to demonstrate the generality of the fuzzy matching task, our sample data will be comparable text strings gathered from various Internet sites. (See the References for sources.) We’ll combine all the text strings from the sites into one data set. We’ll search through the data set looking for duplicates. This is simpler than merging and matching multiple pairs of inputs. The data set will include a sequential identifier and a variable identifying the source internet site. The input data set looks like this.
### Text String Table

<table>
<thead>
<tr>
<th>ID</th>
<th>Source Web Site</th>
<th>Text String</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>How many graduate students does it take to screw in a light bulb?</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>How many developers does it take to change a light bulb?</td>
</tr>
<tr>
<td>3</td>
<td>B</td>
<td>How many paranoids does it take to change a lightbulb?</td>
</tr>
</tbody>
</table>

In the sample code below:

- **PROC SQL** performs a reflexive join to create a cross product of the input data set with itself, so that each text string is compared with every other text string in the input.

- The WHERE condition compares pairs of text strings once (A:B), not twice. It excludes comparisons in the reverse order (B:A) by only comparing text strings when the id in the first copy of the data set is less than the id in the second copy of the data set, i.e. when `a.id < b.id`.

- **COMPGED** is used to calculate distance. The modifiers 'I' and 'L' ignore case and leading blanks.

- The WHERE condition also limits the output to pairs of text strings with low distances. You set the maximum distance that is acceptable for two text strings to be considered similar enough to be matched. The macro variable `maxscore` is set to that maximum distance and will depend on the characteristics of your data.

```plaintext
%let maxscore=999;
proc sql;
create table matches_sql as
select a.joke as joke1, b.joke as joke2,
   compged(a.joke,b.joke,&maxscore,'il' ) as gedscore
from jokes a, jokes b
where a.id < b.id
   and calculated gedscore < &maxscore
order by calculated gedscore;
quit;
```

The results are:

<table>
<thead>
<tr>
<th>Joke1</th>
<th>Joke2</th>
<th>GED Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q: How many graduate students does it take to screw in a light bulb? A: Only one, but it may take upwards of five years for him to get it done.</td>
<td>Q: How many graduate students does it take to screw in a light bulb? A: Only one, but it may take upwards of five years for him to get it done.</td>
<td>0</td>
</tr>
<tr>
<td>Q: How many graduate students does it take to screw in a light bulb? A: It all depends on the size of the grant.</td>
<td>Q: How many graduate students does it take to screw in a light bulb? A: It all depends on the size of the grant.</td>
<td>0</td>
</tr>
<tr>
<td>Q: How many developers does it take to change a light bulb? A: That's a hardware issue</td>
<td>Q: How many developers does it take to change a light bulb? A: That's a hardware problem</td>
<td>510</td>
</tr>
<tr>
<td>Q: How many developers does it take to change a light bulb? A: It works in my office</td>
<td>Q: How many developers does it take to change a light bulb? A: It works on my computer</td>
<td>610</td>
</tr>
</tbody>
</table>
The first line is a joke that appears in identical form on two web sites. The second line is a data entry error that repeated the same joke from the same source in the input data set. The third and fourth lines are slightly different phrasings of the same joke on two web sites. The matches look good. Later we’ll see some of the non-matches.

The next example demonstrates a nested loop technique for creating a cross-product data set. The major differences between the code below and the PROC SQL above are:

- The input data set is read once in the outer DATA step loop and a second time in an inner DO loop.
- The “IF _N_ < I” limits the distance calculation to once per pair of text strings.
- The “IF GEDScore < &MaxScore” limits the output to pairs of text strings with low distances.

```sas
%let maxscore=999;
data matches;
set jokes(rename=(joke=joke1)) end=eof1 nobs=nobs1;
do i = 1 to nobs1;
  set jokes(rename=(joke=joke2)) point=i;
gedscore=compged(joke1,joke2,&maxscore,'i' );
  if _n_ < i then do;
    if gedscore < &maxscore then output matches;
  end;
end;
keep joke1 joke2 gedscore;
run;
```

The results are identical to the results of the SQL step above.

**COPE WITH UNCERTAINTY**

Are you curious what happens if maxscore is set higher? If you change the cutoff from 999 to 1500, there is one additional match.

<table>
<thead>
<tr>
<th>Joke1</th>
<th>Joke2</th>
<th>GED Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q: How many paranoids does it take to change a lightbulb?</td>
<td>Q: How many paranoids does it take to change a lightbulb?</td>
<td>1300</td>
</tr>
<tr>
<td>A: Who wants to know?</td>
<td>A: What do you mean by that?</td>
<td></td>
</tr>
</tbody>
</table>

If you increase maxscore to 2000, the additional pairs are a mixture of matched pairs, pairs that do not match, and pairs that reasonable people might consider either matched or not matched. Usually, it is not possible to choose a distance which cleanly separates the matches from the non-matches. Here are some of the additional results.

<table>
<thead>
<tr>
<th>Joke1</th>
<th>Joke2</th>
<th>GED Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q: How many paranoids does it take to change a lightbulb?</td>
<td>Q: How many astronomers does it take to change a light bulb?</td>
<td>1880</td>
</tr>
<tr>
<td>A: Who wants to know?</td>
<td>A: None, astronomers prefer the dark.</td>
<td></td>
</tr>
<tr>
<td>Q: How many developers does it take to change a light bulb?</td>
<td>Q: How many developers does it take to change a light bulb?</td>
<td>1900</td>
</tr>
<tr>
<td>A: It works on my computer</td>
<td>A: That's a hardware issue</td>
<td></td>
</tr>
<tr>
<td>Q: How many paranoids does it take to change a lightbulb?</td>
<td>Q: How many developers does it take to change a light bulb?</td>
<td>1990</td>
</tr>
<tr>
<td>A: Who wants to know?</td>
<td>A: It works in my office</td>
<td></td>
</tr>
</tbody>
</table>

Applications Big and Small

NESUG 2007
To evaluate the matching process and choose the maximum distance for matches, sort the potential matches in order from lowest distance to the highest distance. Scan the list of potential matches looking for two boundaries:

- the highest distance for a match
- the lowest distance for a non-match

Sometimes you don’t have enough information to know which pairs are matches and which are non-matches, so these boundary distances are estimates. If you can estimate these two boundary distances, you can assign each potential match to one of three categories:

- Definitely matched
- Uncertain
- Definitely not matched

You can use the proportion of uncertain matches to evaluate alternative match scoring algorithms. If you are lucky, the range of distances in which your algorithm produces uncertain results will be small. For some processes, manual intervention to flag ambiguous pairs as matches (or not) is possible. Otherwise, if your process must be fully automated, someone will need to make a decision about what rules to apply. Choosing these rules depends on the relative costs and benefits of false matches and false non-matches. Ask two questions:

- What are the consequences of a missing a valid match (a false non-match)?
- What are the consequences of an incorrect match (a false match)?

The consequences depend on how your output will be used. If you are creating a mailing list for an offer to potential customers, false matches and false non-matches probably don’t matter much. Maybe you didn’t send someone any offer, maybe you sent two offers. On the other hand, if you are mailing warnings for a safety recall, a failure to notify someone would be a serious mistake.

In a research study, a false match might mean that you will join data from two sources for two different people and then wonder why the results for that person make no sense. Or worse yet, you might not notice. Poor matching can seriously dilute the results of a research study by introducing unnecessary randomness. Failing to identify correct matches will reduce your sample size, but assuming the ability to identify matches is randomly distributed, it will not otherwise dilute your chances of finding effects.

Now that we have the basics of fuzzy matching, we will discuss some techniques for improving the results.

**PRE-SCREEN PAIRS**

At this point in your project, you might notice that your comparison jobs are not running as fast as you would like. So far, our sample code has compared all possible pairs of text strings. A cross product of the 2 input data sources was created and distances calculated for all pairs. For large quantities of data, this is a very time-consuming computation. If both sources had 10,000 input rows, the number of pair-wise comparisons would be 100 million. The examples above cut this number in half by comparing A:B and skipping comparing B:A. Also, eliminate any identical duplicates before you begin the fuzzy matching process. Checking for exact matches is much faster than checking for fuzzy matches.

What else can you do to speed up processing? Wherever possible, a qualifying or screening test should be used to reduce the number of pairs that are compared. For last names, you might choose to compare only names that begin with the same letter. All the “A” names like Adams and Anderson would be compared, but Adams and Smith would not be compared. Similarly, you could limit address matching to addresses in the same state.

For our sample data, we can use the ‘source web site’ variable to limit the number of comparisons. If we know that each source web site does not contain the same text string twice, there is no need to compare text strings from the same source. The next example adds this test in **bold**.
%let maxscore=999;
data matches_data_if;
set jokes(rename=(joke=joke1 source=source1)) end=eof1 nobs=nobs1;
do i = 1 to nobs1;
   if _n_ < i then do;
      set jokes(rename=(joke=joke2 source=source2)) point=i;
      if source1^=source2 then do;
         gedscore=compged(joke1,joke2,&maxscore,'iL' );
      end;
      if gedscore < &maxscore then output;
   end;
end;
keep joke1 joke2 gedscore;
run;

The results now contain 3 matches instead of 4 matches. The accidental duplicate from one source (due to a
data entry error) is no longer included in the output.

ADJUST WEIGHTS WITH CALL COMPCOST
COMPGED calculates the distance between two strings by adding up costs for a series of operations that start
with string 1 and build string 2. What are the possible operations and what are default costs assigned to each
operation? The SAS documentation for COMPGED provides a complete list of operations and default costs. Here
is a condensed summary:
• inserting or replacing a character at the beginning of the string (cost=200)
• inserting, replacing or ignoring a character not at the beginning of the string (cost=100)
• appending a character to the output string after there is no more input (cost=50)
• ignoring punctuation (cost=30) or unmatched blanks (cost=10)
• repeating a character, changing a double character to a single character, or swapping two adjacent
   characters (cost=20)
• copying a character from the first string to the second string (cost=0 - This is free.)

You can use CALL COMPCOST to describe a different list of costs. The costs will be used by the COMPGED
function until they are changed by another CALL COMPCOST or a step boundary occurs. The arguments for
CALL COMPCOST are a list of operations and associated values, like this:

CALL COMPCOST(operation-1, value-1 <,operation-2, value-2 ...>);

This can be used to adjust the costs to achieve symmetric results, so that the distance between text string 1 and
2 equals the distance between text string 2 and 1. To do this, assign equal costs to the operations within each of
the following pairs: INSERT and DELETE, FINSERT and FDELETE (for the first characters), APPEND and
TRUNCATE, DOUBLE and SINGLE.

The example below uses the DATA step approach to create a cross-product data set of matches because there is
no obvious way to use CALL COMPCOST in PROC SQL. The added code is in bold. It changes the weights
used by COMPGED so that the distance between a pair of text strings and the reverse pair is the same. It weights
changes to the punch-line at the end of the joke text strings more heavily, and de-emphasizes the importance of
variations in punctuation.
• APPEND is increased from the default of 50 to 200.
• TRUNCATE is increased from the default of 10 to 200.
• PUNCTUATION is reduced from the default of 30 to 10.
data matches data wt;
    set jokes(rename=(joke=joke1 source=source1)) end=eof1 nobs=nobs1;
    if _n_ = 1 then do;
        call compcost( 'fdelete=',200, 'finsert=',200, 'freplace=',200,
                       'delete=',100, 'insert=',100, 'replace=',100,
                       'append=',200, 'truncate=',200,
                       'double=',20, 'single=',20, 'swap=',20,
                       'blank=',10, 'punctuation=',10,
                       'match=',0 );
    end;
    do i = 1 to nobs1;
        if _n_ < i then do;
            set jokes(rename=(joke=joke2 source=source2)) point=i;
            if source1^=source2 then do;
                gedscore=compged(joke1,joke2,&maxscore,'iL' );
                if gedscore < &maxscore then output;
            end;
        end;
    end;
    keep joke1 joke2 gedscore;
run;

The new output has one fewer pseudo-match than the previous output, because the calculated distance increased from 1990 to 2320.

<table>
<thead>
<tr>
<th>Joke1</th>
<th>Joke2</th>
<th>GED Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q: How many paranoids does it take to change a lightbulb?</td>
<td>Q: How many developers does it take to change a light bulb?</td>
<td>2320</td>
</tr>
<tr>
<td>A: Who wants to know?</td>
<td>A: It works in my office</td>
<td></td>
</tr>
</tbody>
</table>

Note that in version 9.1, there is a COMPCOST bug. It does not affect this example, but consider using the workaround in SAS Note SN-V9-016881 for other situations.

**ADJUST FOR DIFFERENT STRING LENGTHS**

Text strings which are equally similar (when evaluated by the percentage of text that is different) can have very different distances as calculated by COMPGED. If there are merely extra characters at the end of one string, this is not heavily weighted by COMPGED. However, if there are different characters within the strings in the same proportions, the longer strings will be considered to be more different.

You might choose to look at the distance in proportion to the length of the input strings. In this simplified example, the first pair of strings each has a length of 25. There is one character that is different, for a rate of 4% of the characters being different. The second pair of strings each has a length of 124, with 5 characters different, for essentially the same rate.

As you can see below, the distances are 100 vs. 500, but the distances divided by the average length of the strings are both 4, reflecting the proportionately similar degree of randomness in the text strings.
data _null_;  
a = "The fat cat sat on a mat.";  
b = "The fat bat sat on a mat.";  
c = "My motto is a foolish consistency is the hobgoblin of little minds, adored by little 
     statesmen and philosophers and divines.";  
d = "My motto is a foolish consistency is the hodgoblin of little mines, abored by little 
     statesmen and philosophers and bovines.";  
a_b = compged(a,b);  
c_d = compged(c,d);  
put a_b= c_d=;  
a_b_len=( length(a) + length(b) ) / 2;  
c_d_len=( length(c) + length(d) ) / 2;  
pput a_b_len= c_d_len=;  
a_b_score = round(a_b / a_b_len , .1);  
c_d_score = round(c_d / c_d_len , .1);  
pput a_b_score= c_d_score=;  
run;
data _null_;  
a = " The quick brown fox jumps over the lazy dog.";
b = " A quick brown fox jumps over a lazy dog.";  
c = " The quick brown fox jumps over the lazy slob.";
if _n_=1 then do;
    commonidx = prxparse("s/\sthe \sa \san \sor \sand /i");
    if commonidx=0 then put 'Parsing error';
end;
retain commonidx;
length a1 b1 $600;
a1 = prxchange(commonidx,-1,a);
b1 = prxchange(commonidx,-1,b);
c1 = prxchange(commonidx,-1,c);
put a1= b1= c1=;
a1_b1 = compged(a1,b1);
a1_c1 = compged(a1,c1);
put a1_b1= a1_c1=;
run;

a1=quick brown fox jumps over lazy dog.
b1=quick brown fox jumps over lazy dog.
c1=quick brown fox jumps over lazy slob.

a1_b1=0 a1_c1=300

You can see that the PRX functions handily create text strings with the common words removed. Comparing these reduced text strings leaves the more substantive differences to determine the distances between pairs of text strings. Once the articles are removed from the sentences, the difference in the nouns stands out.

**STANDARDIZE TERMS**

Some words or phrases within text strings should not be ignored or removed, but they may need to be translated into standard terminology. One familiar example of this is nicknames. Bill, Billy, Will, William, and Willy are often equivalent. Results will be improved if equivalent terms are standardized. Specific knowledge about the input data is essential for this step.

The translation can be done using one of several alternatives.

- a hash table for lookups
- PRX functions or the TRANWRD function
- a format to map alternatives to the standard

The translation list can be kept in an Excel spreadsheet so that it can be updated easily. The spreadsheet can be imported into a SAS data set and then read into the hash table or format automatically.

**ADJUST RELATIVE CONTRIBUTIONS OF IDENTIFIERS**

For some matching processes, more than one identifier can be matched. For example, name, address fields, Social Security Number, and/or date of birth can all contribute to identifying individuals. (Numbers and dates can be converted to text strings and evaluated by COMPGED.) For each pair of individuals, there should be a distance calculated for each identifier. The distances for all the identifiers can be combined in different ways. The criteria for matching could be:

- The number of identifiers with low distances might be required to be above some threshold.
- A weighted sum of distances for all identifiers could be used to emphasize the fields that provide the most information. (For example, there tend to be many more possible last names than first names. A match on last name might be more informative and more heavily weighted than a match on first name.)
- A weighted sum of distances might always omit the largest distance, to allow for the possibility of one bad identifying field.
CONCLUSIONS
No single approach to fuzzy matching will automatically solve all types of problems or handle all types of input. Fuzzy matching is an iterative process of matching, followed by review of the results and changes to the matching algorithms. These steps are repeated until you are satisfied with the results or no further improvements can be made. There are always likely to be some pairs which can not be definitely classified as matches or non-matches.

When using Base SAS for fuzzy matching, three techniques can be powerful:

- The COMPGED function calculates a distance that quantifies the difference between two text strings.
- The CALL COMPCOST routine allows you to customize the COMPGED distance calculation.
- Pre-processing of the text strings and post-processing of the distances can improve the ability to distinguish between matches and non-matches.

REFERENCES
There are two essential references:

SAS Online Help for the COMPGED function and the CALL COMPCOST routine


Other helpful articles include:


Schreier, Howard. 2004, “Using Edit-Distance Functions to Identify ‘Similar’ E-mail Addresses” Proceedings of the 29th Annual SAS Users Group International Conference.

Sources for sample data:
http://www.cs.bgu.ac.il/~omri/Humor/lightbulb.html
http://www.eyrie.org/~thad/strange/lightbulbs.html
http://www.netfunny.com/rft/jokes/88old/bulb.html
http://www.twisted.dk/light.htm

ACKNOWLEDGMENTS
SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Thanks to Joel Gordon of MDRC for the benefit of learning about some of his thinking about fuzzy matching. There is much more to be learned.

CONTACT INFORMATION
Your comments and questions are valued and encouraged. Please contact the author at:

Paulette W. Staum
Paul Waldron Consulting
2 Tupper Lane
West Nyack, NY 10994
staupt@optonline.net