Quick, Call the "FUZZ": Using Fuzzy Logic

Richann Watson, DataRich Consulting; Louise Hadden, Abt Associates Inc.

ABSTRACT

SAS® practitioners are frequently called upon to do a comparison of data between two different data sets and find that the values in synonymous fields do not line up exactly. A second quandary occurs when there is one data source to search for particular values, but those values are contained in character fields in which the values can be represented in myriad different ways. This paper discusses robust, if not warm and fuzzy, techniques for comparing data between, and selecting data in, SAS data sets in not so ideal conditions.

INTRODUCTION

SAS has provided a number of tools which can perform "fuzzy matching". Among these tools are "wild card" searches in a where statement using the LIKE (alias ?) and CONTAINS operators; string searches in an if statement using operators and functions; "fuzzy" comparison functions such as COMPGED and SPEDIS; PROC FCMP; PROC GEOCODE, and data driven control tables enabling user-defined formats to standardize data. The purpose of this paper is to introduce, by example, each of these tools and techniques.

WHERE STATEMENT WILD CARD TECHNIQUES

There are times when you may need to subset records in a data file but specifying the full search string may be an arduous and error prone process, or there may be multiple search strings that have a similar, identifiable pattern. You could default to the standard logic of *VAR in ('FULL VALUE1' 'FULL VALUE2' etc.)*, or, you can use the following "fuzzy" techniques.

CONTAINS or ? Operators

If you are using a sub-setting WHERE statement you can use a question mark (?) or the word CONTAINS instead of an equal (=). The CONTAINS or ? will allow look for records that have values that contain what is specified.

The CONTAINS and ? operators used in the WHERE statement are demonstrated using the ROSTER data set below (Data Display 1).

FIRSTNAME	LASTNAME	GENDER	SCORE	AGE	EMAIL
Jan	Write	F	1.00000000000012	44.9999999999990	Jan_Write@mail.org
Lucy	Smyth	F	1.00000000000121	53.9383983572895	Lucy_Smyth@mail.org
Kris	Johnson	F	1.00324325660746	39.3566050650240	Kris_Johnson@mail.org
Chris	Jones	M	0.00000000000121	48.8268309377139	Chris_Jones@mail.org
Tracey	Smith	F	0.00000000000012	46.4499999999991	Tracey_Smith@mail.org
Tracy	Besley	M	0.00324325660746	47.4999999999990	Tracy_Besley@mail.org
Tracie	Smith-jones	F	-2.00000000000921	35.6659822039699	Tracie_Smith-jones@mail.org
Chrys	Jones-Wright	F	-2.00000000000092	34.1546885694730	Chrys_Jones-Wright@mail.org
Jon	Wright	M	1.00000000000038	46.8145106091718	Jon_Wright@mail.org
John	Hall	M	1.00000000000384	42.9949999999900	John_Hall@mail.org
Timothy	Bones	M	-12.00000000000920	48.3504449007529	Timothy_Bones@mail.org
Jason	Jaones	M	-12.00000000000091	48.7118412046543	Jason_Jaones@mail.org
Tyler	ones	M	-5.00413456081936	42.9499999999990	Tyler_ones@mail.org

Data Display 1: ROSTER

In order to subset the ROSTER data set for people that have 'Jones' in their last name, the CONTAINS or ? operators can minimize the chance of error by searching for any record that contains the word 'Jones' in the variable LASTNAME (SAS Program 1).

```
data jones_1;
   set roster;
   where LASTNAME ? 'Jones';
run;
```

```
data jones_2;
   set roster;
   where LASTNAME contains 'Jones';
run;
```

SAS Program 1: Illustration of CONTAINS and ? operators on WHERE statement

Regardless of which operator is used they will both yield the same results (Data Display 2).

FIRSTNAME	LASTNAME	GENDER
Chris	Jones	M
Chrys	Jones-Wright	F

Data Display 2: Subset of ROSTER for 'Jones' using CONTAINS or ? operators

It is important to note that the CONTAINS or ? operators are case sensitive, thus if casing should be ignored, then it should be used in conjunction with the LOWCASE function or UPCASE function to force the variable to be one case.

LIKE Operator

Similar to the CONTAINS operator, the LIKE operator searches for specific strings based on a wild-card (% or _) or a pattern. In addition, the LIKE operator is also case sensitive.

The LIKE operator is illustrated using the ROSTER data set. In this example, we want to find all the people with 'Jones' in the last name. We could use the CONTAINS or ? operator if all we wanted was to look for records with the value of 'Jones', or we could use the LIKE operator. Both methods will yield the same results.

```
data jones_3;
   set roster;
   where LASTNAME like '%Jones%';
run;
```

SAS Program 2: Illustration of LIKE operator using only a wild card

Data Display 3 shows the data produced from SAS Program 2. So why use the LIKE operator if it does the same thing as the CONTAINS or ? operators? With the LIKE operator there is a bit more flexibility with regard to where the value specified is found within the variable.

FIRSTNAME	LASTNAME	GENDER
Chris	Jones	M
Chrys	Jones-Wright	F

Data Display 3: Subset of ROSTER for 'Jones' using LIKE operator and wild card

In some cases, it may be necessary to only keep records where the specified text is found at the beginning of, or at the end of, a string as illustrated in SAS Program 3 and Data Display 4. SAS Program 3 searches for records where the first two characters of the value for LASTNAME is 'Jo'.

```
data jones_4;
   set roster;
   where LASTNAME like 'Jo%';
run;
```

SAS Program 3: Illustration of LIKE operator using search string at beginning only

FIRSTNAME	LASTNAME	GENDER
Kris	Johnson	F
Chris	Jones	M
Chrys	Jones-Wright	F

Data Display 4: Subset of ROSTER for last names that start with 'Jo'

If you need to broaden the search to look for any records where 'ones' is in the last name and is preceded by at least one character, you could use a search pattern including an underscore: in this case, a search pattern of '_ones'. The '_' indicates that there is one character that precedes 'ones'. Each '_' indicates a character. Thus '__ones' would indicate there are two characters that precede 'ones', and a search patter of '_ones_' would indicate there is one character before and after the 'ones'. However, we wanted at least one character to precede 'ones' so we would use a search pattern with a combination of '%_' (SAS Program 4 and Data Display 5).

```
data jones_5a;
   set roster;
   where LASTNAME like '%_ones%';
run;
```

SAS Program 4: Illustration of LIKE operator using wild card and string pattern

FIRSTNAME	LASTNAME	GENDER
Chris	Jones	M
Tracie	Smith-jones	F
Chrys	Jones-Wright	F
Timothy	Bones	M
Jason	Jaones	M

Data Display 5: Subset of ROSTER for records with 'ones' in last name and at least one-character preceding 'ones'

If we had only used '%ones%', then the record for 'Tyler ones' would have been retrieved when it should not have been, since it did not have at least one character preceding the 'ones' (SAS Program 5 and Data Display 6).

```
data jones_5b;
   set roster;
   where LASTNAME like '%ones%';
run;
```

SAS Program 5: Illustration of LIKE operator using wild card

FIRSTNAME	LASTNAME	GENDER
Chris	Jones	M
Tracie	Smith-jones	F
Chrys	Jones-Wright	F
Timothy	Bones	M
Jason	Jaones	M
Tyler	ones	M

Data Display 6: Subset of ROSTER for records with 'ones' in last name

With the flexibility of the LIKE operator we can use a combination of wildcards to get the desired output. For example, we could search for records that start with 'J' and contain 'nes' and have at least one character between 'J' and 'nes' as shown in SAS Program 6 and Data Display 7.

```
data jones_6;
   set roster;
   where LASTNAME like 'J%_nes%';
run;
```

SAS Program 6: Illustration of LIKE operator searching for first character using wild card and string pattern

FIRSTNAME	LASTNAME	GENDER
Chris	Jones	M
Chrys	Jones-Wright	F
Jason	Jaones	M

Data Display 7: Subset of ROSTER for records that start with 'J' and contain 'nes'

If we need to search strings containing one of the wild cards, we can take advantage of an ESCAPE clause. For example, we could search for records with email addresses that contain underscores between the first and last name, as shown in SAS Program 7 and Data Display 8. Note that the escape character used must not be an underscore or percent sign.

```
data jones_7;
   set roster;
   where email like 'Ch___^_Jo%' escape '^';
run;
```

SAS Program 7: Illustration of LIKE operator with an escape clause using wild card and string pattern

FIRSTNAME	LASTNAME	GENDER	EMAIL
Chris	Jones	M	Chris_Jones@mail.org
Chrys	Jones-Wright	F	Chrys_Jones-Wright@mail.org

Data Display 8: Subset of ROSTER for all records that have email with underscores with first name starting with 'Ch' and last name starting with 'Jo'

= (Sounds like) Operator*

The sounds like operator (=*) allows you to search for records that may phonetically sound like the string that is provided. SAS Program 8 and Data Display 9 illustrate the use of the sounds like operator.

```
data tracey;
   set roster;
   where FIRSTNAME =* 'Tracey';
run;
```

SAS Program 8: Illustration of =* (sounds like) operator

FIRSTNAME	LASTNAME	GENDER
Tracey	Smith	F
Tracy	Besley	M
Tracie	Smith-jones	F

Data Display 9: Subset of ROSTER for all records that have first name that sounds like 'Tracey'

The downside to these operators is that they will only work on the WHERE statement. They cannot be used on an IF statement.

SEARCHING FOR A STRING WITH IF STATEMENTS

There are several different SAS functions-based options that can be explored when using IF statements for which wild card techniques are not applicable. These "fuzzy" search techniques are explored below.

Character Comparison on IF Statement

If you are interested in only matching on a string that is at the beginning of the value in the variable, then the you can use '=:' to do a character comparison. By default, SAS will truncate the longer value to the length of the shorter value when doing the comparison and then it will compare each character in both strings. SAS Program 9 and Data Display 10 provide an illustration of the use of the character comparison operator.

```
data jones_7;
   set roster;
   if LASTNAME =: 'Jones';
run;
```

SAS Program 9: Illustration of character comparison =:

FIRSTNAME	LASTNAME	GENDER
Chris	Jones	M
Chrys	Jones-Wright	F

Data Display 10: Subset of ROSTER for last names that start with Jones

The disadvantage of this technique is that it will only match on the first part of a character string, so if you need to have a match on any other parts of a character string, then additional techniques should be implemented.

INDEX Function

If you need to search for a string anywhere within a variable or another string, then the INDEX function could be utilized. With the INDEX function you would need to provide two arguments: the variable or string to be searched and the string that you are searching for.

Syntax: **INDEX**(*source*, *excerpt*)

```
data jones_8a;
   set roster;
   if index(LASTNAME, 'Jones');
run;
```

SAS Program 10: Illustration of INDEX function

FIRSTNAME	LASTNAME	GENDER
Chris	Jones	M
Chrys	Jones-Wright	F

Data Display 11: Subset of ROSTER for last name that contains Jones using INDEX function

The downside of using INDEX is that it is case sensitive. If you need to look for values regardless of case status, you could use the UPCASE function or LOWCASE function to force the source string to be one case and then specify the excerpt string to be the same case. SAS Program 11 and Data Display 12 show that LASTNAME = 'Smith-jones' was missed in the initial program execution because 'jones' was not proper case.

```
data jones_8b;
   set roster;
   if index(upcase(LASTNAME),
'JONES');
run;
```

SAS Program 11: Illustration of INDEX with UPPER function

FIRSTNAME	LASTNAME	GENDER
Chris	Jones	M
Tracie	Smith-jones	F
Chrys	Jones-Wright	F

Data Display 12: Subset of ROSTER for last name that contains Jones regardless of casing using INDEX function

FIND Function

An alternative to using the INDEX function with either the UPCASE or LOWCASE function to ensure you capture all possible case statuses, is the use of the FIND function. The FIND function works in a similar fashion to the INDEX function, with the difference being that the FIND function allows for a start position and/or modifier(s) including instructions regarding case.

Syntax: **FIND**(*string*, *substring* <, *modifier(s)*> <, *start-position*>)

FIND(*string*, *substring* <, *start-position*> <, *modifier(s)*>)

The modifiers that can be used are described in Table 1. When specifying the modifier(s), it should be enclosed in quotation marks. If specifying both modifiers, then they will both be enclosed in a single set of quotation marks.

Modifier	Description
i or I	Indicates that casing should be ignored when searching for the substring within the string
t or T	Indicates that trailing blanks should be removed from ***both*** the string and substring

Table 1: FIND Modifiers

Notice that SAS Program 12 yields the same set of records as SAS Program 11. In both sample code the casing of the last name was essentially ignored so that all possible records that contained the search string were retrieved.

```
data jones_9a;
   set roster;
   if find(LASTNAME,'jones','i');
run;
```

SAS Program 12: Illustration of FIND function with 'i' modifier

FIRSTNAME	LASTNAME	GENDER
Chris	Jones	M
Tracie	Smith-jones	F
Chrys	Jones-Wright	F

Data Display 13: Subset of ROSTER for last name that contains Jones regardless of casing using FIND function

The start-position indicates where in the string the search should start from. Notice in SAS Program 13 that the order in which the modifier(s) and start-position are specified in the function are interchangeable.

```
data jones_9b;
   set roster;
   if find(LASTNAME, 'jones', 'i', 5);
run;
```

```
data jones_9c;
   set roster;
   if find(LASTNAME, 'jones', 5, 'i');
run;
```

SAS Program 13: Illustration of FIND function with 'i' modifier and start position searching left to right

By specifying the starting search position as 5, the search will only pick up any records where 'jones' is found in positions 5 or greater in the source string (Data Display 14).

FIRSTNAME	LASTNAME	GENDER
Tracie	Smith-jones	F

Data Display 14: Subset of ROSTER for last name that contains Jones regardless of casing but starting in position 5 searching left to right

If the value is a negative number, it indicates that the search will be from right to left, while a positive number searches from left to right. The default search is left to right. Note that the position of the start is the absolute value of the start-position. For example, start-position = 5 and start-position = -5 both start the search in position 5 of the string. The sign just indicates the direction of the search.

In the example illustrated by SAS Program 14 and Data Display 15, notice that the search starts in the same position as the search done in SAS Program 13 with the difference being that it will search the first 5 characters only for the value of 'jones' (Data Display 15).

```
data jones_9d;
   set roster;
   if find(LASTNAME, 'jones', -5, 'i');
run;
```

SAS Program 14: Illustration of FIND function with 'i' modifier and start position searching right to left

FIRSTNAME	LASTNAME	GENDER
Chris	Jones	M
Chrys	Jones-Wright	F

Data Display 15: Subset of ROSTER for last name that contains Jones regardless of casing but starting in position 5 searching right to left

PRXMATCH Function

PRXMATCH function uses either a regular expression ID or PERL-regular expressions and can be used to search for a string within another string. With the various options indicated in the expression, you can indicate that you wish to ignore casing.

Syntax: **PRXMATCH**(*regular-expression-id* | *perl-regular-expression*, *source*)

Notice that SAS Program 15 yields the same results as SAS Program 12 (see Data Display 13 and Data Display 16).

```
data jones_10;
   set roster;
   if prxmatch('m/Jones/i', LASTNAME);
run;
```

SAS Program 15: Illustration of PRXMATCH function with 'i' option

FIRSTNAME	LASTNAME	GENDER
Chris	Jones	M
Tracie	Smith-jones	F
Chrys	Jones-Wright	F

Data Display 16: Subset of ROSTER for last name that contains Jones regardless of casing using PRXMATCH function

However, with the PRXMATCH function, you can search for more than one string within another. As illustrated in SAS Program 16 and Data Display 17, records that contain the values of 'Jones' or 'Jaones' regardless of casing in the source string are retrieved.

```
data jones_11;
   set roster;
   if prxmatch('m/Jones|Jaones/oi', LASTNAME);
run;
```

SAS Program 16: Illustration of PRXMATCH function with 'i' and 'o' options

FIRSTNAME	LASTNAME	GENDER
Chris	Jones	M
Tracie	Smith-jones	F
Chrys	Jones-Wright	F
Jason	Jaones	M

Data Display 17: Subset of ROSTER for last name that contains Jones or Jaones regardless of casing

Multiple options or metacharacters can be used to build the desired search string. For example, if you wanted to only retrieve the records from a source string that contains the certain values that are found at the beginning of the source string, then the use of '^' metacharacter indicates the beginning position and will only retrieve records that find the search string at the beginning of the source string. SAS Program 17 and Data Display 18 demonstrate the use of the '^' metacharacter. Notice that the record for 'Tracie Smith-jones' is not retrieved because 'Jones' was not at the beginning of the source string.

```
data jones_12;
   set roster;
   if prxmatch( 'm/^Jones|Jaones/oi', LASTNAME);
run;
```

SAS Program 17: Illustration of PRXMATCH function with 'i' and 'o' options and '^' metacharacter

FIRSTNAME	LASTNAME	GENDER
Chris	Jones	M
Chrys	Jones-Wright	F
Jason	Jaones	M

Data Display 18: Subset of ROSTER for last name that contains Jones or Jaones regardless of casing at beginning of source string

Below are several resources when using PRX functions:

- A complete list of metacharacters: https://support.sas.com/documentation/cdl//en/lefunctionsref/69762/HTML/default/viewer.htm#p0s9ilagexmjl8n1u7e1t1jfnzlk.htm.
- A SAS tip reference sheet: https://support.sas.com/rnd/base/datastep/perl_regexp/regexp-tip-sheet.pdf.
- A tool to check syntax of PRX expression: https://regex101.com/.

FUZZY COMPARISONS

CHARACTER FUZZY COMPARISONS

Character strings, especially strings describing names and addresses, are notoriously dirty and prone to spacing, length and punctuation issues. Any real-world comparison of character strings or selection based on character strings needs to be both flexible and configurable, i.e. the degree of "sameness" needs to be quantifiable. SAS provides several character functions that allow you to make a fuzzy comparison: COMPARE, COMPGED, COMPLEV, SOUNDEX and SPEDIS. Each of these functions use a different fuzzy algorithm and can be used in conjunction with one another to achieve a (subjectively) optimal match. Use of these functions produces inexact results by definition, and results must be reviewed carefully. Examples and explanations of each of these functions follow.

COMPARE Function

The COMPARE function determines the first character at which two character variables differ and returns the position of the first character difference. If there are no differences between the two variables, then it will return a 0.

Syntax: **COMPARE**(*string-1*, *string-2* <, *modifier(s)*>)

The modifiers that can be used are described in Table 2. When specifying the modifier(s), it should be enclosed in quotation marks. If specifying multiple modifiers, then all modifiers will be enclosed in a single set of quotation marks.

Modifier	Description
i or I	Indicates that casing should be ignored for both strings.
l or L	Indicates that leading blanks should be removed from both strings before comparison.
n or N	Indicates that quotation marks from either argument should be removed and that casing should be ignored.
: (colon)	Indicates that the longer string should be truncated to the length of the shorter string. If this is not specified, then the shorter string is padded with blank spaces to the length of the longer string.

Table 2: COMPARE Modifiers

The ROSTER data in Data Display 1 is augmented to include additional values for FIRSTNAME and LASTNAME (Data Display 19). The differences between the original values and the additional values are highlighted in pink.

FIRSTNAME	LASTNAME	FIRSTNAME2	LASTNAME2
Jan	Write	Jan	Wright
Lucy	Smyth	Lucy	Smith
Kris	Johnson	Chris	Johnson
Chris	Jones	Chris	Jones
Tracey	Smith	Tracey	Smith
Tracy	Besley	Tracey	Besley
Tracie	Smith-jones	Tracie	Smith-Jones
Chrys	Jones-Wright	Chris	Jones-Wright
Jon	Wright	Jon	Wright
John	Hall	John	Hall
Timothy	Bones	Timothy	Jones
Jason	Jaones	Jason	Jones
Tyler	ones	Tyler	Jones

Data Display 19: Augmented ROSTER (NEWROSTER)

SAS Program 18 and Data Display 20 illustrate the use of COMPARE function without the use of a modifier and with the use of the 'i' modifier.

```
data roster_compare;
   set newroster;
   FNCOMP = compare(FIRSTNAME, FIRSTNAME2);
   LNCOMP = compare(LASTNAME, LASTNAME2);
   FNCOMP_I = compare(FIRSTNAME, FIRSTNAME2, 'i');
   LNCOMP_I = compare(LASTNAME, LASTNAME2, 'i');
run;
```

SAS Program 18: Illustration of COMPARE function without modifiers and with 'i' modifier

FIRSTNAME	LASTNAME	FIRSTNAME2	LASTNAME2	FNCOMP	FNCOMP_I	LNCOMP	LNCOMP_I
Jan	Write	Jan	Wright	1	1	4	4
Lucy	Smyth	Lucy	Smith	0	0	3	3
Kris	Johnson	Chris	Johnson	1	1	0	0
Chris	Jones	Chris	Jones	0	0	0	0
Tracey	Smith	Tracey	Smith	0	0	0	0
Tracy	Besley	Tracey	Besley	5	5	0	0
Tracie	Smith-jones	Tracie	Smith-Jones	0	0	7	0
Chrys	Jones-Wright	Chris	Jones-Wright	4	4	0	0
Jon	Wright	Jon	Wright	0	0	0	0
John	Hall	John	Hall	0	0	0	0
Timothy	Bones	Timothy	Jones	0	0	-1	-1
Jason	Jaones	Jason	Jones	0	0	-2	-2
Tyler	ones	Tyler	Jones	0	0	1	1

Data Display 20: Illustration of COMPARE function without modifiers and with 'i' modifier

With the use of the 'i' modifier notice that 'Smith-jones' and 'Smith-Jones' are considered the same value. Notice that for Jason Jaones (Jones) the COMPARE function returns a -2. This is because if 'Jaones' and 'Jones' were sorted 'Jaones' would come first in a sort sequence. Therefore, the sign on the value returned indicates the order of the arguments. A negative number represents the first argument occurring first in a sort sequence and a positive number represents the second argument occurring first in the sort sequence.

SAS Program 19 and Data Display 21 demonstrate the use of the 'l' modifier to remove any leading blanks and also shows the use of multiple modifiers in one function call.

```
data roster_compare2;
   set newroster;
   FNCOMP_L = compare(FIRSTNAME, FIRSTNAME2, 'l');
   LNCOMP_L = compare(LASTNAME, LASTNAME2, 'l');
   FNCOMP_IL = compare(FIRSTNAME, FIRSTNAME2, 'il');
   LNCOMP_IL = compare(LASTNAME, LASTNAME2, 'il');
run;
```

SAS Program 19: Illustration of COMPARE function with 'l' and 'il' modifier(s)

FIRSTNAME	LASTNAME	FIRSTNAME2	LASTNAME2	FNCOMP_L	FNCOMP_IL	LNCOMP_L	LNCOMP_IL
Jan	Write	Jan	Wright	0	0	4	4
Lucy	Smyth	Lucy	Smith	0	0	3	3
Kris	Johnson	Chris	Johnson	1	1	0	0
Chris	Jones	Chris	Jones	0	0	0	0
Tracey	Smith	Tracey	Smith	0	0	0	0
Tracy	Besley	Tracey	Besley	5	5	0	0
Tracie	Smith-jones	Tracie	Smith-Jones	0	0	7	0
Chrys	Jones-Wright	Chris	Jones-Wright	4	4	0	0
Jon	Wright	Jon	Wright	0	0	0	0
John	Hall	John	Hall	0	0	0	0
Timothy	Bones	Timothy	Jones	0	0	-1	-1
Jason	Jaones	Jason	Jones	0	0	-2	-2
Tyler	ones	Tyler	Jones	0	0	1	1

Data Display 21: Illustration of COMPARE function with 'l' and 'il' modifier(s)

In SAS Program 18 we noticed that it yielded a mismatch for the first name of the first record. However, with the use of the 'l' modifier, there is now a match.

COMPGED Function

The COMPGED function determines how close the two arguments are in regard to matching (i.e., it determines the generalized edit distance between the two values).

Syntax: **COMPGED**(*string-1*, *string-2* <, *cutoff* > <, *modifier(s)*>)

The cutoff is a numeric value that is returned if the generalized edit distance is greater than the cutoff value. The modifiers that can be used are the same as those used for COMPARE, described in Table 2.

Refer to the following SAS documentation for details on computing the generalized edit distance
https://documentation.sas.com/?docsetId=lefunctionsref&docsetTarget=p1r4l9jwgatggtn1ko81fyjys4s7.htm&docsetVersion=9.4&locale=en

SAS Program 20 and SAS Program 21 are used to demonstrate how the generalized edit distance can be calculated based on differences in the two strings.

```
data roster_compged;
   set newroster;
   FNCOMP = compged(FIRSTNAME, FIRSTNAME2);
   LNCOMP = compged(LASTNAME, LASTNAME2);
   FNCOMP_I = compged(FIRSTNAME, FIRSTNAME2, 'i');
   LNCOMP_I = compged(LASTNAME, LASTNAME2, 'i');
run;
```

SAS Program 20: Illustration of COMPGED function without modifiers and with 'i' modifier

FIRSTNAME	LASTNAME	FIRSTNAME2	LASTNAME2	FNCOMP	FNCOMP_I	LNCOMP	LNCOMP_I
Jan	Write	Jan	Wright	20	20	210	210
Lucy	Smyth	Lucy	Smith	0	0	100	100
Kris	Johnson	Chris	Johnson	300	300	0	0
Chris	Jones	Chris	Jones	0	0	0	0
Tracey	Smith	Tracey	Smith	0	0	0	0
Tracy	Besley	Tracey	Besley	100	100	0	0
Tracie	Smith-jones	Tracie	Smith-Jones	0	0	100	0
Chrys	Jones-Wright	Chris	Jones-Wright	100	100	0	0
Jon	Wright	Jon	Wright	0	0	0	0
John	Hall	John	Hall	0	0	0	0
Timothy	Bones	Timothy	Jones	0	0	200	200
Jason	Jaones	Jason	Jones	0	0	100	100
Tyler	ones	Tyler	Jones	0	0	200	200

Data Display 22: Illustration of COMPGED function without modifiers and with 'i' modifier

In the first row of Data Display 22, we see that for the first name the generalized edit distance without modifiers and with the modifier to ignore casing both yield a value of 20. This is because for FIRSTNAME2 there are two blank spaces that precede the value 'Jan'. The unit cost for each blank space is 10 units, therefore, the total unit cost is 20 units. For the last name we have to replace 'gh' with 'te' and truncate the value. The unit cost to replace a character is 100 units per character and the unit cost to truncate the output string is 10 units per character truncated. Since we had to replace two characters and truncate one character, the total unit cost is 210.

```
data roster_compged2;
   set newroster;
   FNCOMP_L = compged(FIRSTNAME, FIRSTNAME2, 'l');
   LNCOMP_L = compged(LASTNAME, LASTNAME2, 'l');
   FNCOMP_IL = compged(FIRSTNAME, FIRSTNAME2, 150, 'il');
   LNCOMP_IL = compged(LASTNAME, LASTNAME2, 200, 'il');
run;
```

SAS Program 21: Illustration of COMPGED function with 'l' and 'il' modifier(s) and cutoff option

FIRSTNAME	LASTNAME	FIRSTNAME2	LASTNAME2	FNCOMP_L	FNCOMP_IL	LNCOMP_L	LNCOMP_IL
Jan	Write	Jan	Wright	0	0	210	200
Lucy	Smyth	Lucy	Smith	0	0	100	100
Kris	Johnson	Chris	Johnson	300	150	0	0
Chris	Jones	Chris	Jones	0	0	0	0
Tracey	Smith	Tracey	Smith	0	0	0	0
Tracy	Besley	Tracey	Besley	100	100	0	0
Tracie	Smith-jones	Tracie	Smith-Jones	0	0	100	0
Chrys	Jones-Wright	Chris	Jones-Wright	100	100	0	0
Jon	Wright	Jon	Wright	0	0	0	0
John	Hall	John	Hall	0	0	0	0
Timothy	Bones	Timothy	Jones	0	0	200	200
Jason	Jaones	Jason	Jones	0	0	100	100
Tyler	ones	Tyler	Jones	0	0	200	200

Data Display 23: Illustration of COMPGED function with 'l' and 'il' modifier(s) and cutoff option

In the first row of Data Display 23, we notice that the total unit cost is 210 as previously determined. Even if we ignore casing for the last name it would still have a unit cost of 210, yet the value for LNCOMP_IL = 200. This is because we have implemented a cutoff value. For the last name the cutoff value of 200 was specified, indicating that regardless of the final generalized edit distance, the value would be cutoff at 200. The same is true for the first name, we implemented a cutoff value of 150 and even though for the third row it shows that the generalized edit distance is 300, for FNCOMP_IL, the value shown is 150.

COMPLEV Function

COMPLEV function is similar to the COMPGED function but instead of computing the generalized edit distance it computes the Levenshtein edit distance. COMPLEV will count the number of operations (e.g., insertions, deletions, replacements) needed to convert one string to the same value of the other.

Syntax: **COMPLEV**(*string-1*, *string-2* < *cutoff* > <, *modifier(s)*>)

Continuing with the use of the new roster data, we use SAS Program 22 to show how the Levenshtein edit distance is determined.

```
data roster_complev;
   set newroster;
   FNCOMP = complev(FIRSTNAME, FIRSTNAME2);
   LNCOMP = complev(LASTNAME, LASTNAME2);
   FNCOMP_I = complev(FIRSTNAME, FIRSTNAME2, 'i');
   LNCOMP_I = complev(LASTNAME, LASTNAME2, 'i');

run;
```

SAS Program 22: Illustration of COMPLEV function without modifiers and with 'i' modifier

FIRSTNAME	LASTNAME	FIRSTNAME2	LASTNAME2	FNCOMP	FNCOMP_I	LNCOMP	LNCOMP_I
Jan	Write	Jan	Wright	2	2	3	3
Lucy	Smyth	Lucy	Smith	0	0	1	1
Kris	Johnson	Chris	Johnson	2	2	0	0
Chris	Jones	Chris	Jones	0	0	0	0
Tracey	Smith	Tracey	Smith	0	0	0	0
Tracy	Besley	Tracey	Besley	1	1	0	0
Tracie	Smith-jones	Tracie	Smith-Jones	0	0	1	0
Chrys	Jones-Wright	Chris	Jones-Wright	1	1	0	0
Jon	Wright	Jon	Wright	0	0	0	0
John	Hall	John	Hall	0	0	0	0
Timothy	Bones	Timothy	Jones	0	0	1	1
Jason	Jaones	Jason	Jones	0	0	1	1
Tyler	ones	Tyler	Jones	0	0	1	1

Data Display 24: Illustration of COMPLEV function without modifiers and with 'i' modifier

For the first row in Data Display 24, two operations are required to get FIRSTNAME and FIRSTNAME2 to match because we need to remove two blank spaces. Three operations are needed to make LASTNAME and LASTNAME2 match because we would need to replace 'te' with 'gh' which counts as two operations, and the 't' in LASTNAME2 needs to be truncated which counts as the third operation.

The concept of cutoff for COMPLEV is the same as with COMPGED. Instead of using a cutoff based on operation unit costs, the cutoff is based on number of operations.

SOUNDEX Function

The SOUNDEX function determines how much two character variables sound alike. It works best with the English language. It is equivalent to using =* (sounds like) on a WHERE statement.

Syntax: **SOUNDEX**(*argument*)

With the SOUNDEX function vowels and the letters 'H', 'W' and 'Y' are excluded except when it is the first character in the argument when determining if the argument sounds like a specific value. Other characters in the English alphabet are assigned one of the following values:

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

The value generated from SOUNDEX is the first character in the argument and then for each character in the argument that is not excluded is assigned one of the values above. If there are two or more consecutive characters assigned the same numeric value, then only the first one is kept.

To demonstrate the use of SOUNDEX, we execute the following data step (SAS Program 23) so that we can calculate the value generated from SOUNDEX using FIRSTNAME. The results are found in Data Display 25

```
data roster_soundex;
   set roster;
   FN_SOUND = soundex(FIRSTNAME);
run;
```

SAS Program 23: Illustration of SOUNDEX function

FIRSTNAME	LASTNAME	FN_SOUND
Jan	Write	J5
Lucy	Smyth	L2
Kris	Johnson	K62
Chris	Jones	C62
Tracey	Smith	T62
Tracy	Besley	T62
Tracie	Smith-jones	T62
Chrys	Jones-Wright	C62
Jon	Wright	J5
John	Hall	J5
Timothy	Bones	T53
Jason	Jaones	J25
Tyler	ones	T46

Data Display 25: Illustration of SOUNDEX function

For the rows in blue, we see that both yield a value of 'C62'. This is because both started with 'C' and the 'h', 'y' and 'i' were discarded, leaving only 'r' and 's'. The 'r' was assigned a value of '6' and 's' was assigned a value of '2'. Notice that if the third row would have started with a 'C' instead of a 'K' it would have resulted in the same value. However, since it is started with a 'K', the result was 'K62'.

For the rows in pink, we see that the value was 'T62' and this was because we discarded all the vowels and 'y' after the first character, leaving only 'r' and 'c'.

For the rows in green, the 'o' and 'h' were discarded leaving only the 'n' after the first argument, resulting in a value of 'J5'.

SPEDIS Function

SOUNDEX determines how close in sound two character values are, while SPEDIS determines how close in spelling two character values are.

Syntax: **SPEDIS**(*query*, *keyword*)

The query identifies the word that will be used to search for a match. The keyword is the target word used in the query. SPEDIS removes trailing blanks in both the query and keyword.

Refer to the following SAS documentation for details on the cost of each operation between the query and the keyword
https://documentation.sas.com/?docsetId=lefunctionsref&docsetTarget=p0vmuxh8ljfn7on164nsgvmdrc5d.htm&docsetVersion=9.4&locale=en

The distance between two values is the sum of the operation costs (see link above) divided by the length of the query value. The value is rounded down to the nearest whole number if the ratio is greater than one.

In SAS Program 24 the FIRSTNAME is the query and FIRSTNAME2 is the target word that will be compared to see if there is a match with FIRSTNAME and determine the distance between the two values.

```
data roster_spedis;
   set newroster;
   FN_SPEDIS = spedis(FIRSTNAME, FIRSTNAME2);
   LN_SPEDIS = spedis(LASTNAME, LASTNAME2);
run;
```

SAS Program 24: Illustration of SPEDIS function

FIRSTNAME	LASTNAME	FIRSTNAME2	LASTNAME2	FN_SPEDIS	LN_SPEDIS
Jan	Write	Jan	Wright	66	40
Lucy	Smyth	Lucy	Smith	0	20
Kris	Johnson	Chris	Johnson	62	0
Chris	Jones	Chris	Jones	0	0
Tracey	Smith	Tracey	Smith	0	0
Tracy	Besley	Tracey	Besley	10	0
Tracie	Smith-jones	Tracie	Smith-Jones	0	9
Chrys	Jones-Wright	Chris	Jones-Wright	20	0
Jon	Wright	Jon	Wright	0	0
John	Hall	John	Hall	0	0
Timothy	Bones	Timothy	Jones	0	40
Jason	Jaones	Jason	Jones	0	16
Tyler	ones	Tyler	Jones	0	25

Data Display 26: Illustration of SPEDIS function

Using Data Display 26, we will trace through the calculation of the distance for the first record (i.e., Jan Write).

The cost to convert ‘ Jan’ to ‘Jan’ is 200. The deletion of the first character is 100. Since the first two characters were blank spaces both needed to be deleted. The length of the query, ‘Jan’, is 3. Thus, the distance is 200 / 3 = 66.6666. Recall from above that if the ratio is less than one then it is rounded down, so the distance is rounded down to 66.

The cost to convert ‘Wright’ to ‘Write’ is 200. We would need to delete a character (‘g’) from the middle which costs 50. We then need to replace one letter in the middle (‘h’ becomes ‘e’) which costs 100. The last operation needed would then be to swap the order of the last two letters which has an operation cost of 50. The length of the query is 5, thus the distance is 200 / 5 = 40.

To better illustrate the conversion of ‘Wright’ to ‘Write’, we walk through it step by step.

- Step 1: Delete ‘g’ – ‘Wright’ becomes ‘Wriht’
- Step 2: Replace ‘h’ with ‘e’ – ‘Wriht’ become ‘Wriet’
- Step 3: Swap order of ‘et’ – ‘Wriet’ becomes ‘Write’

NUMERIC FUZZY COMPARISONS

All the techniques illustrated thus far have been dealing with character strings, but what if we have a numeric value? There are various options available for determining if a numeric value is equivalent to another numeric value. In some cases, the values will be exactly equal, and no additional comparison is needed. However, there are some cases where the values are not quite equal but are equal ‘enough’ so

that if the values were rounded or truncated or a 'fuzz' factor is added then the values would be considered equal. Depending on the type of 'fuzz' factor you wish to consider will determine which function should be best utilized. Using the ROSTER data in Data Display 1, we will illustrate several numeric 'fuzzy' functions.

CEIL and CEILZ Functions

The CEIL function rounds **UP** to the nearest **smallest integer that is greater than or equal to the argument**, that is it will return an integer value that is greater than or equal to the argument. It uses fuzzing in order to avoid issues with floating points. If the result returned from the CEIL function is with 1E-12 of the argument, then the value is considered equal to the integer portion of the argument.

Syntax: **CEIL**(*argument*)

However, if you do not want to consider any fuzzing when rounding up to the nearest integer, then CEILZ is the function that should be used. CEILZ works the same as CEIL but it does not use fuzzing. Therefore, even if the return value is within 1E-12 of the argument it will round up to the nearest smallest integer instead of considering the value equal to the integer portion of the argument.

Syntax: **CEILZ**(*argument*)

SAS Program 25 and Data Display 27 illustrate the use of CEIL and CEILZ functions.

```
data fuzz_score;
   set roster;
   S_CEIL = ceil(SCORE);
   S_CEILZ = ceilz(SCORE);
run;
```

SAS Program 25: Illustration of CEIL and CEILZ functions

FIRSTNAME	LASTNAME	GENDER	SCORE	S_CEIL	S_CEILZ
Jan	Write	F	1.00000000000012	1	2
Lucy	Smyth	F	1.00000000000121	2	2
Kris	Johnson	F	1.00324325660746	2	2
Chris	Jones	M	0.00000000000121	1	1
Tracey	Smith	F	0.00000000000012	0	1
Tracy	Besley	M	0.00324325660746	1	1
Tracie	Smith-jones	F	-2.00000000000921	-2	-2
Chrys	Jones-Wright	F	-2.00000000000092	-2	-2
Jon	Wright	M	1.00000000000038	1	2
John	Hall	M	1.00000000000384	2	2
Timothy	Bones	M	-12.00000000000920	-12	-12
Jason	Jaones	M	-12.00000000000091	-12	-12
Tyler	ones	M	-5.99999999999999	-6	-5

Data Display 27: Illustration of CEIL and CEILZ functions

Notice that for the pink highlighted cells in Data Display 27 CEIL returns the integer portion of the SCORE since the return values were within 1E-12 of the original argument. However, for these same records CEILZ rounds up to the nearest smallest integer because there was zero fuzzing allowed. For the cells highlighted in blue since the argument was within 1E-12 of -6, then CEIL considers these equivalent and therefore returns the value of -6, but with CEILZ returned the smallest integer that was greater than the argument, which is -5.

FLOOR and FLOORZ Functions

FLOOR and FLOORZ functions operate in similar manner to CEIL and CEILZ with the difference being that FLOOR and FLOORZ round ***DOWN*** to the nearest **largest integer that is <u>less</u> than or equal to the argument**. FLOOR utilizes fuzzing while FLOORZ has zero fuzzing. Thus, return values within 1E-12 of the argument for FLOOR will return the integer portion.

Syntax: **FLOOR**(*argument*)

FLOORZ(*argument*)

SAS Program 26 and Data Display 28 illustrate the use of CEIL and CEILZ functions.

```
data fuzz_score;
   set roster;
   S_FLOOR = floor(SCORE);
   S_FLOORZ = floorz(SCORE);
run;
```

SAS Program 26: Illustration of FLOOR and FLOORZ functions

FIRSTNAME	LASTNAME	GENDER	SCORE	S_FLOOR	S_FLOORZ
Jan	Write	F	1.00000000000012	1	1
Lucy	Smyth	F	1.00000000000121	1	1
Kris	Johnson	F	1.00324325660746	1	1
Chris	Jones	M	0.00000000000121	0	0
Tracey	Smith	F	0.00000000000012	0	0
Tracy	Besley	M	0.00324325660746	0	0
Tracie	Smith-jones	F	-2.00000000000921	-3	-3
Chrys	Jones-Wright	F	-2.00000000000092	-2	-3
Jon	Wright	M	1.00000000000038	1	1
John	Hall	M	1.00000000000384	1	1
Timothy	Bones	M	-12.00000000000920	-13	-13
Jason	Jaones	M	-12.00000000000091	-12	-13
Tyler	ones	M	-5.99999999999999	-6	-6

Data Display 28: Illustration of FLOOR and FLOORZ functions

Data Display 28 shows that for the cells highlighted in pink that the return values are different. FLOOR returns the integer portion when the return value is within 1E-12 of the argument, while FLOORZ returns the largest integer value that is less than the argument.

FUZZ Function

The FUZZ function either returns the nearest integer (i.e., rounds up or down based on normal rounding rules) if the return value is within 1E-12 of the argument. If the return value is not within 1E-12 of the argument, then the FUZZ function returns the argument.

Syntax: **FUZZ**(*argument*)

SAS Program 27 and Data Display 29 illustrate the use of the FUZZ function. In order to demonstrate that the FUZZ function returned the argument if the return value was not within 1E-12 of the argument a format was applied to S_FUZZ.

```
data fuzz_score;
   set roster;
   format S_FUZZ 20.14;
   S_FUZZ = fuzz(SCORE);
run;
```

SAS Program 27: Illustration of FUZZ function

FIRSTNAME	LASTNAME	GENDER	SCORE	S_FUZZ
Jan	Write	F	1.00000000000012	1.00000000000000
Lucy	Smyth	F	1.00000000000121	1.00000000000121
Kris	Johnson	F	1.00324325660746	1.00324325660746
Chris	Jones	M	0.00000000000121	0.00000000000121
Tracey	Smith	F	0.00000000000012	0.00000000000000
Tracy	Besley	M	0.00324325660746	0.00324325660746
Tracie	Smith-jones	F	-2.00000000000921	-2.00000000000921
Chrys	Jones-Wright	F	-2.00000000000092	-2.00000000000000
Jon	Wright	M	1.00000000000038	1.00000000000000
John	Hall	M	1.00000000000384	1.00000000000384
Timothy	Bones	M	-12.00000000000920	-12.00000000000920
Jason	Jaones	M	-12.00000000000091	-12.00000000000000
Tyler	ones	M	-5.99999999999999	-6.00000000000000

Data Display 29: Illustration of FUZZ function

For the cells highlighted in pink in Data Display 29, note that the return value was within 1E-12 and therefore, the value was rounded accordingly.

ROUND and ROUNDZ Functions

The ROUND and ROUNDZ functions will round the first argument to the closest multiple of the optional second argument. If the second argument is not provided, then the functions round the only argument to the nearest integer using basic rounding rules. As with CEILZ and FLOORZ, ROUNDZ rounds using zero fuzzing.

Syntax: **ROUND**(*argument* <, *rounding-unit*>)

ROUNDZ(*argument* <, *rounding-unit*>)

Using the data in Data Display 1, we use the AGE values to illustrate the ROUND and ROUNDZ function. SAS Program 28 and Data Display 30 illustrate the use of both functions without the second argument. Therefore, for both ROUND and ROUNDZ, the functions will round to the nearest integer. However, ROUNDZ rounds with zero fuzzing.

```
data fuzz_age;
   set roster;
   A_ROUND1 = round(AGE);
   A_ROUNDZ1 = roundz(AGE);
run;
```

SAS Program 28: Illustration of ROUND and ROUNDZ functions without a second argument

FIRSTNAME	LASTNAME	GENDER	AGE	A_ROUND	A_ROUNDZ
Jan	Write	F	44.9999999999990	45	45
Lucy	Smyth	F	53.9383983572895	54	54
Kris	Johnson	F	39.3566050650240	39	39
Chris	Jones	M	48.8268309377139	49	49
Tracey	Smith	F	46.4499999999991	46	46
Tracy	Besley	M	47.4999999999990	48	47
Tracie	Smith-jones	F	35.6659822039699	36	36
Chrys	Jones-Wright	F	34.1546885694730	34	34
Jon	Wright	M	46.8145106091718	47	47
John	Hall	M	42.9949999999900	43	43
Timothy	Bones	M	48.3504449007529	48	48
Jason	Jaones	M	48.7118412046543	49	49
Tyler	ones	M	42.9499999999990	43	43

Data Display 30: Illustration of ROUND and ROUNDZ functions without a second argument

The cells highlighted in pink demonstrate the use of the fuzzing for the ROUND function. The ROUND function rounds the value to the nearest integer of 48 because using a fuzz factor SAS sees 47.4999999999990 as 47.5 as shown in Output 1 row 6. But ROUNDZ rounded to the nearest integer of 47 because zero fuzzing was used and therefore the entire value was used when rounding.

	FIRSTNAME	LASTNAME	GENDER	AGE
1	Jan	Write	F	45
2	Lucy	Smyth	F	53.938398357
3	Kris	Johnson	F	39.356605065
4	Chris	Jones	M	48.826830938
5	Tracey	Smith	F	46.45
6	Tracy	Besley	M	47.5
7	Tracie	Smith-jones	F	35.665982204
8	Chrys	Jones-Wright	F	34.154688569
9	Jon	Wright	M	46.814510609
10	John	Hall	M	42.995
11	Timothy	Bones	M	48.350444901
12	Jason	Jaones	M	48.711841205
13	Tyler	ones	M	42.95

Output 1: SAS data set representation of data display

If you want to round to something other than the nearest integer, then ROUND and ROUNDZ functions can also have a second argument that indicates to what degree the value should be rounded to, this is the rounding unit. SAS Program 29 and Data Display 31 show the results of when the rounding unit is 10.

```
data fuzz_age;
   set roster;
   A_ROUND2 = round(AGE, 10);
   A_ROUNDZ2 = roundz(AGE, 10);
run;
```

SAS Program 29: Illustration of ROUND and ROUNDZ functions with rounding unit = 10

FIRSTNAME	LASTNAME	GENDER	AGE	A_ROUND	A_ROUNDZ
Jan	Write	F	44.9999999999990	50	40
Lucy	Smyth	F	53.9383983572895	50	50
Kris	Johnson	F	39.3566050650240	40	40
Chris	Jones	M	48.8268309377139	50	50
Tracey	Smith	F	46.4499999999991	50	50
Tracy	Besley	M	47.4999999999990	50	50
Tracie	Smith-jones	F	35.6659822039699	40	40
Chrys	Jones-Wright	F	34.1546885694730	30	30
Jon	Wright	M	46.8145106091718	50	50
John	Hall	M	42.9949999999900	40	40
Timothy	Bones	M	48.3504449007529	50	50
Jason	Jaones	M	48.7118412046543	50	50
Tyler	ones	M	42.9499999999990	40	40

Data Display 31: Illustration of ROUND and ROUNDZ functions with rounding unit = 10

With the rounding unit of 10, the AGE is rounded to the nearest multiple of 10. Recall that how SAS sees the value will determine what result is returned when using ROUND. Row 1 in Output 1 indicates that SAS sees the value as 45 therefore rounding to the nearest multiple of 10 would mean it rounds up to 50. However, with ROUNDZ it is using the entire value as is and rounds down to 40.

The rounding unit does not have to be a multiple of 10 or even an integer. The next two examples demonstrate the outcome when the rounding unit is 2 (SAS Program 30 and Data Display 32) and 0.1 (SAS Program 31 and Data Display 33).

```
data fuzz_age;
   set roster;
   A_ROUND2 = round(AGE, 2);
   A_ROUNDZ2 = roundz(AGE, 2);
run;
```

SAS Program 30: Illustration of ROUND and ROUNDZ functions with rounding unit = 2

FIRSTNAME	LASTNAME	GENDER	AGE	A_ROUND	A_ROUNDZ
Jan	Write	F	45	46	44
Lucy	Smyth	F	53.93839836	54	54
Kris	Johnson	F	39.35660507	40	40
Chris	Jones	M	48.82683094	48	48
Tracey	Smith	F	46.45	46	46
Tracy	Besley	M	47.5	48	48
Tracie	Smith-jones	F	35.6659822	36	36
Chrys	Jones-Wright	F	34.15468857	34	34
Jon	Wright	M	46.81451061	46	46
John	Hall	M	42.995	42	42
Timothy	Bones	M	48.3504449	48	48
Jason	Jaones	M	48.71184121	48	48
Tyler	ones	M	42.95	42	42

Data Display 32: Illustration of ROUND and ROUNDZ functions with rounding unit = 2

```
data fuzz_age;
   set roster;
   A_ROUND3 = round(AGE, .1);
   A_ROUNDZ3 = roundz(AGE, .1);
run;
```

SAS Program 31: Illustration of ROUND and ROUNDZ functions with rounding unit = 0.1

FIRSTNAME	LASTNAME	GENDER	AGE	A_ROUND	A_ROUNDZ
Jan	Write	F	44.9999999999990	45	45
Lucy	Smyth	F	53.9383983572895	53.9	53.9
Kris	Johnson	F	39.3566050650240	39.4	39.4
Chris	Jones	M	48.8268309377139	48.8	48.8
Tracey	Smith	F	46.4499999999991	46.5	46.4
Tracy	Besley	M	47.4999999999990	47.5	47.5
Tracie	Smith-jones	F	35.6659822039699	35.7	35.7
Chrys	Jones-Wright	F	34.1546885694730	34.2	34.2
Jon	Wright	M	46.8145106091718	46.8	46.8
John	Hall	M	42.9949999999900	43	43
Timothy	Bones	M	48.3504449007529	48.4	48.4
Jason	Jaones	M	48.7118412046543	48.7	48.7
Tyler	ones	M	42.9499999999990	43	42.9

Data Display 33: Illustration of ROUND and ROUNDZ functions with rounding unit = 0.1

COMPFUZZ Function

SAS provides a comparison function to compare numeric variables that accounts for differences in floating point precision, COMPFUZZ. There may be differences in floating point precision between different operating systems and versions of SAS – and, there can be slight differences in sums relating to the order in which addends are introduced. In validation, for example, when there is difficulty achieving

the same results with different staff running independently, COMPFUZZ may be employed to assess whether floating point imprecision is to blame and offers a path to resolving issues.

The COMPFUZZ function returns a value (-1, 0, 1) that categorizes the comparison of two floating-point numbers. Optional arguments include FUZZ, which specifies the threshold for comparisons, expressed in multiples of machine precision; and SCALE, which specifies the scale factor.

Syntax: **COMPFUZZ**(*value-1, value-2* <, *fuzz* <, *scale*>>)

The calculation for the threshold is based on the value of the FUZZ argument. If FUZZ is greater than or equal to 0 but less than 1, then the threshold is the FUZZ times the absolute value of the SCALE. However, if the FUZZ is greater than 1, then the threshold is the FUZZ times the absolute value of the SCALE times the machine precision constant [CONSTANT('MACEPS')]. Note that the machine precision constant will vary based on the specific precision associated with the machine being used.

The example shown below demonstrates that in a relatively small data set, slight fuzz can be introduced simply by changing the order of addends in a sum – and whether or not that fuzz matters given metrics of scale. The differences, as demonstrated above with the numeric fuzzy functions, may not always be visible without the use of the round or other functions, but SAS assesses and flags slight differences. In this case, it is up to the user to make a decision regarding whether floating point precision is important in their particular situation and make an effort to standardize data if needed.

```
data compfuzz_score;
   set roster;

   /* ensure that sufficient decimal places are available */
   format SCOREP AGEP SUM1 SUM2 DIFF SCALE 22.16;
   SCOREP = SCORE + .00000000000009;
   AGEP = AGE +.00000000000009;

   /* add the numbers in two different orders */
   SUM1 = SCORE + SCOREP + AGE + AGEP;
   SUM2 = AGEP + AGE + SCOREP + SCORE;
   DIFF = abs(SUM1 - SUM2);

   put SUM1 = ;
   put SUM2 = ;
   put DIFF = ;

   /* use a fuzz factor and a scale value gives the correct result */
   SCALE = abs(SCORE) + abs(SCOREP);
   COMPFUZZ = compfuzz(SUM1, SUM2, 4, SCALE);
   put 'fuzz and scale: ' COMPFUZZ = ;
run;
```

SAS Program 32: Illustration of the COMPFUZZ function

OBS	SCOREP	AGEP	SUM1	SUM2
1	1.0000000000002100	44.9999999999990000	91.9999999999984000	91.9999999999984000
2	1.0000000000013000	53.9383983572896000	109.8767967145810000	109.8767967145810000
3	1.0032432566075500	39.3566050650240000	80.7196966432631000	80.7196966432631000
4	0.0000000000013000	48.8268309377140000	97.6536618754304000	97.6536618754304000
5	0.0000000000002100	46.4499999999991000	92.8999999999986000	92.8999999999986000
6	0.0032432566075500	47.4999999999990000	95.0064865132131000	95.0064865132131000
7	-2.0000000000091200	35.6659822039699000	67.3319644079215000	67.3319644079215000
8	-2.0000000000008300	34.1546885694731000	64.3093771389443000	64.3093771389443000
9	1.0000000000004700	46.8145106091718000	95.6290212183445000	95.6290212183445000
10	1.0000000000039300	42.9949999999900000	87.9899999999878000	87.9899999999878000
11	-12.0000000000091000	48.3504449007530000	72.7008898014875000	72.7008898014875000
12	-12.0000000000008000	48.7118412046544000	73.4236824093069000	73.4236824093069000
13	-5.9999999999999000	42.9499999999991000	73.8999999999982000	73.8999999999982000

OBS	DIFF	SCALE	COMPFUZZ
1	0.0000000000000142	2.0000000000003200	-1
2	0.0000000000000142	2.0000000000025100	1
3	0.0000000000000000	2.0064865132150100	0
4	0.0000000000000000	0.0000000000025100	0
5	0.0000000000000142	0.0000000000003300	-1
6	0.0000000000000000	0.0064865132150100	0
7	0.0000000000000000	4.0000000000183300	0
8	0.0000000000000142	4.0000000000017500	-1
9	0.0000000000000000	2.0000000000008500	0
10	0.0000000000000142	2.0000000000077700	1
11	0.0000000000000000	24.0000000000183000	0
12	0.0000000000000000	24.0000000000017000	0
13	0.0000000000000142	11.9999999999998000	-1

Data Display 34: Illustration of the COMPFUZZ function

Due to lack of space AGE and SCORE are not shown in Data Display 34. Refer to Data Display 1 for the values of AGE and SCORE.

Note that for observations 1, 2, 5, 8, 10 and 13, there are differences between SUM1 and SUM2. Although the difference is not evident in the variables themselves, if you took the difference of the two values you will see that there is a difference. Since FUZZ = 4 then we use FUZZ * SCALE * CONSTANT('MACEPS'). For the machine that was used for these examples, the precision constant is 0.0000000000000002. Therefore, looking at observation 1, we get threshold = 4 * 2.0000000000003200 * 0.0000000000000002 = 0.0000000000000018. So, the SUM1 < SUM2 – threshold which has a COMPFUZZ value of -1. For row 2, SUM1 > SUM2 + threshold which yields a COMPFUZZ value of 1. Due to how SAS displays the data, we can only see up to so many decimals and therefore it is difficult to see that SUM1 is less than SUM2 – threshold for row 1 and SUM1 is greater than SUM2 + threshold for row 2. Floating point precision can and does vary between machines, systems and even versions of SAS. As noted above, it can explain the inability to reproduce the same results in verification. Please consult the SAS documentation for more detail on the COMPFUZZ function and floating-point precision.

FUZZY ADDRESS CHECKING WITH SAS

Address matching is a task for which fuzzy matching techniques are frequently used. It is an example of "phrase matching", where there are multiple words in a phrase that need to match in order for two phrases to be considered equal. Consider the Data Display 35 below, a printout of selected street types from SASHELP.GCTYPE, in which there are a number of variations in street types from across the world.

NAME	TYPE	GROUP
AV	AVE	12
AVE	AVE	12
AVEN	AVE	12
AVENIDA	AVE	12
AVENU	AVE	12
AVENUE	AVE	12
AVN	AVE	12
AVNUE	AVE	12
BELT	BELT	16
BELTWAY	BELT	16
BL	BLVD	34
BLVD	BLVD	34
BOUL	BLVD	34
BOULEVARD	BLVD	34
BOULV	BLVD	34
BTWY	BELT	16
CIR	CIR	64
CIRC	CIR	64
CIRCL	CIR	64
CIRCLE	CIR	64
CIRCULO	CIR	64
CRCL	CIR	64
CRCLE	CIR	64
CÍR	CIR	64
CÍRCLE	CIR	64
HIGHWAY	HWY	250
HIGHWY	HWY	250
HIWAY	HWY	250
HIWY	HWY	250
HWAY	HWY	250
HWY	HWY	250
LA	LN	296
LANE	LN	296
LANES	LN	296
LN	LN	296
ST	ST	464
STR	ST	464
STREET	ST	464
STRT	ST	464
TER	TER	490
TERR	TER	490
TERRACE	TER	490
THOR	THOR	492
THOROUGHFARE	THOR	492
TLWY	TOLL	494
TOLL	TOLL	494
TOLLWAY	TOLL	494

Data Display 35: SASHELP.GCTYPE

As you can see, Avenue can be spelled a number of ways. SAS supplies this look-up table for PROC GEOCODE, discussed below. In a file or files of addresses, such variations in the street type spelling are just the tip of the iceberg in terms of the vast panoply of "dirty data". SAS uses this look-up table, and others, in performing fuzzy matches for street addresses (and other geographic entities such as county, congressional districts, etc.) and produces standardized addresses. In addition, we'll discuss another SAS tool for fuzzy address matching, creating our own fuzzy function to perform the normalizing of street types below.

As of SAS version 9.4 M5, PROC GEOCODE is available in BASE SAS to perform address matching, utilizing fuzzy matching techniques and normalizing geographic variables such as state, county, zip, and street address, as well as custom levels. It has been available in SAS/GRAPH since Version 8.2. The procedure produces X and Y coordinates for matches.

SAS provides city, zip and a very limited street level data set in the SASHELP folder and provides links to files required for full street level geocoding and more in SAS MAPSONLINE. Full street level matching data for the United States for this example was obtained through a link on SAS MAPS ONLINE, a SAS provided website dedicated to mapping with SAS, and used to replace the limited street level data set. This is referred to in the code in the LOOKUPSTREET statement as street.usm. This match data set is too large to show, but it is used just as a control file, format file, or fuzzy match with a file would be. Just as the comparison functions discussed above evaluate and score the results of fuzzy matching, PROC GEOCODE provides the match level and score for the fuzzy address matching performed.

The sample data set in Data Display 36 will be used to illustrate the use of PROC GEOCODE (SAS Program 33).

PROVNUM	ADDRESS	CITY	STATE	ZIP
105205	2121 E COMMERCIAL BLVD	FORT LAUDERDALE	FL	33308
106088	4650 STATE RD 16	SAINT AUGUSTINE	FL	32092
146035	2259 EAST 1100TH STREET	MENDON	IL	62351
175446	915 MCNAIR STREET	HALSTEAD	KS	67056
175549	12340 QUIVIRA ROAD	OVERLAND PARK	KS	66213
245395	965 MCMILLAN STREET	WORTHINGTON	MN	56187
385263	970 W JUNIPER AVENUE	HERMISTON	OR	97838
525362	719 E CATHERINE ST BOX 167	DARLINGTON	WI	53530
525462	245 SYCAMORE ST	SAUK CITY	WI	53583
676397	23450 PINE SHADOW LN	PORTER	TX	77365

Data Display 36: Sample list of addresses

```
PROC GEOCODE                  /* Invoke geocoding procedure        */
   method=STREET              /* Specify geocoding method          */
   data=prov                  /* Input data set of addresses       */
   out=dd.GEOCODED            /* Output data set with X/Y values   */
   lookupstreet=street.usm    /* Primary street lookup data set    */
   type=SASHELP.GCTYPE;       /* Lookup data set-added street type */
run;
```

SAS Program 33: Illustration of PROC GEOCODE

The results from running the data through PROC GEOCODE are found in Data Display 37. Notice that for the rows highlighted in pink that the variations for ‘street’ were all converted to ‘St’ and the rows highlighted in blue converted the variations for ‘road’ to ‘Rd’. For some addresses, there may be situations where there are two addresses tied to a particular location. This can occur when you have a physical location address and a mailing address as shown by the rows in green. GEOCODE will “normalize” the addresses to the actual physical addresses for that location found in the look up file which are used by the USPS.

OBS	ADDRESS	CITY	STATE	ZIP
1	2121 E COMMERCIAL BLVD	FORT LAUDERDALE	FL	33308
2	4650 STATE RD 16	SAINT AUGUSTINE	FL	32092
3	2259 EAST 1100TH STREET	MENDON	IL	62351
4	915 MCNAIR STREET	HALSTEAD	KS	67056
5	12340 QUIVIRA ROAD	OVERLAND PARK	KS	66213
6	965 MCMILLAN STREET	WORTHINGTON	MN	56187
7	970 W JUNIPER AVENUE	HERMISTON	OR	97838
8	719 E CATHERINE ST BOX 167	DARLINGTON	WI	53530
9	245 SYCAMORE ST	SAUK CITY	WI	53583
10	23450 PINE SHADOW LN	PORTER	TX	77365

OBS	M_ADDR	M_CITY	M_STATE	M_ZIP
1	2121 E Commercial Blvd	Fort Lauderdale	FL	33308
2	4650 State Rd 16	Green Cove Springs	FL	32092
3	2237 E 1100th St	Mendon	IL	62351
4	915 McNair St	Halstead	KS	67056
5	12340 Quivira Rd	Overland Park	KS	66213
6	965 McMillan St	Worthington	MN	56187
7	970 W Juniper Ave	Hermiston	OR	97838
8	8374 Co Rd E	Darlington	WI	53530
9	245 Sycamore St	Sauk City	WI	53583
10	24200 Pine Cir	Porter	TX	77365

Data Display 37: Result of sample address list run through PROC GEOCODE to normalize street address

CREATE YOUR OWN FUZZ FUNCTION

Many of the fuzzy matching techniques discussed above are case sensitive – so that frequently variables representing patterns need to be standardized with regard to case and punctuation. A full discussion of PROC FCMP is beyond the scope of this paper, but we will briefly discuss a user-defined function that can be helpful when performing fuzzy matching (and elsewhere). Use of a format library entry to identify non-standard street name terminology is a helpful tool – and the format can “learn” by including new variations as they are found. One method of using the results of the learned translations is to write a function incorporating the translations, as well as standardizing case, etc. A simplistic example follows below, in which street types are standardized prior to going into a fuzzy matching routine. As with the format, the function can be informed by new variations uncovered. In addition, the function performs such tasks as standardizing case, left justifying, and trimming.

```
proc fcmp outlib=work.funcs.address;
      function streets(addr $) $;
            length clean_address standardized_address $100;
            clean_address=upcase(addr);
            clean_address=left(trim(clean_address));
            clean_address=tranwrd(clean_address,' STREET ',' ST ');
            clean_address=tranwrd(clean_address,'ROAD','RD');
            clean_address=tranwrd(clean_address,'BOULEVARD','BLVD');
            clean_address=tranwrd(clean_address,'AVENUE','AVE');
            clean_address=tranwrd(clean_address,' DRIVE ',' DR ');
            clean_address=tranwrd(clean_address,'PLACE','PL');
            clean_address=tranwrd(clean_address,'LANE','LN');
            clean_address=tranwrd(clean_address,'CIRCLE','CIR');
            clean_address=tranwrd(clean_address,'COURT ','CT ');
            clean_address=tranwrd(clean_address,'PARKWAY','PKWY');
            standardized_address=clean_address;
            return(standardized_address);
      endsub;
quit;
```

SAS Program 34: Illustration of PROC FCMP

Data Display 38 represents the original roster used in previous examples, with addresses added to illustrate the use of the user-built function, ***streets***.

FIRSTNAME	LASTNAME	ADDRESS
Jan	Write	1234 Any Place, Anywhere, NC 12345
Lucy	Smyth	5673 MyBlock Drive, Myhome, TX 79732
Kris	Johnson	19752 Home Blvd, Home, MA 03321
Chris	Jones	98 NewTown Circle, Newtown, OK 31313
Tracey	Smith	1294-13 Johnson Lane, Nowhere, MN 23213

Data Display 38: Roster with addresses

Once a user-built function is created the options CMPLIB needs to point to the location of where the user-built function resides. SAS Program 35 illustrates the use of the option as well as implementing the function. After the execution of the program the addresses are cleaned so that there is consistency as shown in Data Display 39.

```
options cmplib=(work.funcs);

data roster3;
   set roster2;
   cleaned_address = streets(address);
run;
```

SAS Program 35: Illustration of usage of user-built function

FIRSTNAME	LASTNAME	CLEANED_ADDRESS
Jan	Write	1234 ANY PL, ANYWHERE, NC 12345
Lucy	Smyth	5673 MYBLOCK DRIVE, MYHOME, TX 79732
Kris	Johnson	19752 HOME BLVD, HOME, MA 03321
Chris	Jones	98 NEWTOWN CIR, NEWTOWN, OK 31313
Tracey	Smith	1294-13 JOHNSON LN, NOWHERE, MN 23213

Data Display 39: Illustration of usage of user-built function

CONCLUSION

SAS has provided a myriad of tools to utilize for "fuzzy" matching. Selection of records with where statements (conditions and special operators) and if statements (:_ operator and functions); standardizing of records using fuzzy matching techniques including user defined formats, functions, and PROC GEOCODE (address information); and PROC FCMP (a user-defined function to clean addresses) have all been discussed. We hope you've gained some appreciation for the "fuzz" and you'll get "fuzzy" along with us!

REFERENCES

Alabaster, Amy and Mary Anne Armstrong. 2018. "Interpreting electronic health data using SAS PRX functions." *Proceedings of WUSS 2018*. https://www.lexjansen.com/wuss/2018/92_Final_Paper_PDF.pdf

Warner, Christine. 2016. "Using Proc FCMP To Improve Fuzzy Matching." *Proceedings of the South East SAS Users Group 2016*. Bethesda, MD. https://analytics.ncsu.edu/sesug/2016/DM-126_Final_PDF.pdf

SAS 9.4 Functions and Call Routines: Reference, Fifth Edition. SAS Institute, Inc. https://documentation.sas.com/?docsetId=lefunctionsref&docsetTarget=n1b7dkf9vhuqczn16qfd41j6dd54.htm&docsetVersion=9.4&locale=en

RECOMMENDED READING

Cadieux, Richard and Daniel R. Bretheim. 2014. "Matching Rules: Too Loose, Too Tight or Just Right?" *Proceedings SAS Global Forum 2014.* Washington, DC: SAS Institute, Inc. http://support.sas.com/resources/papers/proceedings14/1674-2014.pdf

Carpenter, Art. 2018. "Using Arrays to Quickly Perform Fuzzy Merge Look-Ups: Case Studies in Efficiency." *Proceedings of the SAS Global Forum 2018*. Denver, CO: SAS Institute, Inc. https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2018/2396-2018.pdf

Carpenter, Art. 2018. "Using the FCMP Procedure to the Fullest: Getting Started and Doing More". *Proceedings of the SAS Global Forum 2018*, Denver, CO: SAS Institute, Inc. https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2018/2403-2018.pdf

Cody, Ron. 2017. "Cody's Data Cleaning Techniques Using SAS®, Third Edition", SAS Press, SAS Institute, Cary, NC, USA.

Dunham, Alan. 2016. "Fuzzy Name-Matching Applications." *Proceedings of the South East SAS Users Group 2016*. Bethesda, MD. https://analytics.ncsu.edu/sesug/2016/DM-109_Final_PDF.pdf

Dunn, Toby. 2014. "Getting the Warm and Fuzzy Feeling with Inexact Matching." *Proceedings of the SAS Global Forum 2014*. Washington, DC: SAS Institute, Inc. https://support.sas.com/resources/papers/proceedings14/1316-2014.pdf

Graebner, Robert W. 2011. "Fuzzy Merges - A Guide to Joining Data sets with Non-Exact Keys Using the SAS SQL Procedure." *Proceedings of the Midwest SAS Users Group 2011.* Kansas City, KS. https://www.lexjansen.com/mwsug/2011/sas101/MWSUG-2011-S110.pdf

Gupta, Sunil. 2018. "Fuzzy Joins with Proc SQL for Better Data Utilization." *Proceedings of the PhUSE-US 2018*. Raleigh, NC. https://www.lexjansen.com/phuse-us/2018/ct/CT01.pdf

Hadden, Louise. 2019. “Like, Learn to Love SAS® Like.” *Proceedings of WUSS 2019.* Renton, WA. https://proceedings.wuss.org/2019/163_Final_Paper_PDF.pdf

Horwitz, Lisa. 2017. “A Long-Time SAS® Programmer Learns New Tricks.” *Proceedings of SAS Global Forum 2017.* Orlando, FL: SAS Institute, Inc. https://support.sas.com/resources/papers/proceedings17/SAS0637-2017.pdf

Marinescu, Daniel. 2017. "Fuzzy Matching and Predictive Models for Acquisition of New Customers." *Proceedings of the SAS Global Forum 2017*. Orlando, FL: SAS Institute, Inc. https://support.sas.com/resources/papers/proceedings17/0881-2017.pdf

Martin, Kathryn. 2012. "Making Fuzzy Merges More Precise using the COMPARE Function in SAS®." *Proceedings of the Western Users of SAS Software 2019*. Long Beach, CA. https://www.lexjansen.com/wuss/2012/92.pdf

McCarthy, Michael. 2016. "Using Fuzzy Logic to Match a Street Address." *Proceedings of the South Central SAS Users Group 2016.* San Antonio, TX. https://www.lexjansen.com/scsug/2016/Using-Fuzzy-Logic.pdf

McNeill, Bill. 2018. "PROC FCMP: A Powerful SAS® Procedure You Should Be Using." *Proceedings of the SAS Global Forum 2018.* Denver, CO: SAS Institute, Inc. https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2018/2125-2018.pdf

Rineer, Brian. 2016. “Get Out of DATA Step Code and into Quality Knowledge Bases.” *Proceedings of SAS Global Forum 2016.* Las Vegas, NV: SAS Institute, Inc. https://support.sas.com/resources/papers/proceedings16/SAS5644-2016.pdf

Salas, S. Bianca, et. al. 2018. ”Fun with Address Matching: Use of the COMPGED Function and the SQL Procedure.” *Proceedings of SAS Global Forum 2018*. Denver, CO: SAS Institute, Inc. https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2018/2487-2018.pdf

Sloan, Stephen; and Dan Hoicowitz. 2016. "Fuzzy Matching: Where Is It Appropriate and How Is It Done? SAS® Can Help." *Proceedings of the South East SAS Users Group 2016.* Bethesda, MD. https://www.lexjansen.com/sesug/2016/BB-141_Final_PDF.pdf

Sloan, Stephen and Kirk Paul Lafler. 2018. “Fuzzy Matching Programming Techniques Using SAS® Software.” *Proceedings of SAS Global Forum 2018*. Denver, CO: SAS Institute, Inc. https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2018/2886-2018.pdf

Teres, Jedediah J. 2011. "Using SQL joins to Perform Fuzzy Matches on Multiple Identifiers." *Proceedings of the Northeast SAS Users Group 2011.* Portland, ME. https://www.lexjansen.com/nesug/nesug11/ps/ps07.pdf

Virkud, Arti. April 2015. "Fuzzy Matching." *Proceedings of the SAS Global Forum 2015*. Dallas, TX: SAS Institute, Inc. https://support.sas.com/resources/papers/proceedings15/3142-2015.pdf

Wood, Jefferson L and Grace Reynolds. 2011. "Macros for Managing Messy Data: Handling Duplicate Study Participants and Making Fuzzy Matches across Multiple Data Sets." *Proceedings of the SAS Global Forum 2011*. Las Vegas, NV: SAS Institute, Inc. https://support.sas.com/resources/papers/proceedings11/227-2011.pdf

CONTACT INFORMATION

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

Richann Watson
DataRich Consulting
richann.watson@datarichconsulting.com

Louise Hadden
Abt Associates, Inc.
louise_hadden@abtassoc.com