Paper 1316-2014

Getting the Warm and Fuzzy Feeling with Inexact Matching

Toby Dunn, Dunn Consulting, San Antonio, Texas

Abstract

With the ever increasing proliferation of disparate complex data being collected and stored, it has never been more important that this information is accurate, clean, integrated, and often times in compliance with an expanding set of government regulations. This means that the data must be cleaned and standardized, duplicates must be identified and removed, and the individual data must be able to be joined or merged together in some way. However, it is often the case that this data does not have the same variables or values to make this possible with a simple Join or Merge. To that end, one has to employ a set of fuzzy logics or fuzzy matching. Simply put, fuzzy matching is the implementation of algorithmic processes (fuzzy logic) to determine the similarity between elements of data such as business names, people names, or address information. Fuzzy logic is used to predict the probability of data with non-exact matches to help in data cleansing, deduplication, or matching of disparate data sets. This paper shows the basics of using fuzzy logic by using SAS® functions, COMPLEV, multiple variables matches, and a modified Porter stemming algorithm.

Managing the proliferation of data today is becoming more and more complex. Businesses often have a multitude of disparate internal and external business systems collecting and storing information. These data will at some point need to be integrated while ensuring duplicates are identified and removed, and that new data is integrated without compromising existing data quality. One of the major problems that anyone who tries to integrate these data faces is lack of any common key or set of key variable in which to join them together. When this happens the programmer is left with two choices, throw up their hands in defeat or do their best to get a warm and fuzzy feeling with fuzzy matching.

Normally matching data between data sets consists of merging or joining two or more sets of data on a set of known and reliable key(s). However, fuzzy matching is the process by which data are joined together on one or more variables where the programmer can't be for certain that the values are always the same. SAS® does provide several functions to help with this uncertain match the one we will see later is the Compged (Generalized Edit Distance) function. Compged measures the dissimilarity between two strings by the number of deletions, insertions, or replacements of single characters that are required to transform one string into another.

A couple of important things to think about are: since Compged is doing a comparison of the strings and returning the number of deletions, insertions and/or replacement of characters the closer the two strings are to each other the lower the overall value return and thus the more certain we can be that the two strings match. Secondly, since there are no known key(s) to match on reliably, one is forced to try to match variables between data sources that may or may not be in a format that is suitable without some prep work on these values. One can conclude that the more variables we use in a match the better and closer the pair of values that Compged uses the better the match rate will be.

Therefore, Fuzzy Matching is nothing more than a bunch of hard core data cleaning with a ten second

compare process at the end. That might be a little over simplifying it but is the essences of Fuzzy Matching.
The order in which things must happen and the order in which I will present them in this paper are as follows: Determine variables that can be used to match or join on, parse/clean their values, and lastly compare the variables and determine the most likely matches with Compged.
Say you were asked to provide a report with a list of all current contractors working in your company that contained all their name, mailing information, current department they are working in and their listed job description. However, the information you need is in two different files and neither has SSN values or any other unique code(s) to match up the two files with.

1.Determine likely matching variables:
Consider the following two data sets: DataSet Contractor which contains information of individual contractors and their Job Descriptions and DataSet Company which contains data on the contractors currently working for the company and has the department that each contractor is working in.
The easiest way to start looking for possible variables to match on is to compare the contents of the two data files to see if the names of the variables are give any hint as to what is there to use.

Company			
#	**Variable**	**Type**	**Len**
3	Address	Char	100
4	CityStateZip	Char	100
2	Company	Char	100
5	Departments	Char	100
1	Name	Char	100

Contractor			
#	**Variable**	**Type**	**Len**
4	Address	Char	100
5	City	Char	100
3	Company	Char	100
1	FirstName	Char	100
2	LastName	Char	100
8	Profession	Char	100
6	State	Char	100
7	ZipCode	Char	100

If the idea is to get the best possible match then we will need as many variables as is reasonable to join on. The reason for this is the fewer variables you use the greater the possibility that values which aren't exactly the same will not return a match when in fact it should have been matched. Looking at the Proc Contents we can see that Company data set has a variable called CityStateZip. This automatically looks like an overloaded field containing three pieces of information. If the values are parsed out we then could match then to the Contractor data set which has these three values broken out into three separate variables. I can see Name is on the Company data set but the Contractor data set has variables for First and Last name. If Name is an overloaded field containing both first and last we will need to parse them out into two variables so we would have two more variables to match on. Lastly I see that both data sets have Address and Company variables which if the values are in the roughly

the same format we can use them. All total we have the possibility of using 7 different variables to match between the files when all is parsed and cleaned up.

2.) Clean Your Data Up

A visual inspection is needed to not only confirm that the values in the fields we want to join are have similar values that as is or after cleaning up will yield a quality match rate, but also to see what if any cleaning of these fields will need to take place.

Contractor							
First	Last	Company	Street	City	State	Zip	JobDescription
John	Smyth❶	United Data Machines ❷Co.	❸Twelve Main ❹St	Boston	Massachusetts	02108	Programming
Mary	Smith	eSolutions, Inc.	12 North Main St	Boston	Massachusetts	02108	❺Program
F❻	Jones	jones Consulting	4 Elm Ave	❼Fall River	Massachusetts	02108	Programs

Here we have a data set containing a list of contractors and their job descriptions. The first thing to look for is the overall layout, in this data sets case its relatively well formed data. That is to say there are no overloaded values, where there is more than one fact described in a variables value. An often seen example of this is name fields which contain first, last and sometimes middle name or initial such as (Toby Dunn or Toby A Dunn). This is good thing, as it means everything is broken down into the smallest usable pieces of information for us and we will not have to parse any of these fields.

❶ Here we see the last name Smyth, which is a unusual spelling of the name Smith. It is important to take note of these types of spellings as they may be legitimate spelling or possibly data entry errors.

❷ One often finds abbreviations like Co., Corp, Inc, etc… in company names. Since we can't be guaranteed that one data source inputs it in correctly within the data source much less between data sources these will need to be cleaned up.

❸ Often times one finds address with the leading digits spelled out as Twelve in this case and in other cases it is represented with all numerals 12. This poses a interesting problem that will have to be overcome as comparing Twelve to 12 will yield a poor match if at all.

❹ Just like in ❷ street address often times have abbreviations used such as St, Rt, Prkway, etc… like in ❷ we will need to deal with these to ensure that we eliminate as many possible false matches due to abbreviations.

❺ The job description field has many values that mean the same things such as Programming, Programmer, Programs. While these can be left alone it does make it hard to use these values. So we might want to start thing about a possibly cleaning these up as much as possible.

❻ Here we see that the person's name is represented by just his first initial. While there isn't much you can easily do with this it is important to note. If there are a great many of these and it effects your final match significantly you may want to do a fuzzy match of the data against or another known clean file to obtain these names. For our purposes today let us just take note of it.

❼ While the City value of Fall River belongs to the state of Massachusetts, it doesn't however belong to the 02108 zip code. In these cases there isn't any additional information to determine if the City is correct or if the zip code is correct. So you are left choosing one or the other as the correct value and look up the other variables value. I tend to choose the zip code as SAS has a built in Zip code file that can be used to verify and correct erroneous City or State values.

Company				
Name	Company	Address	CityStateZip	Team
❶Johnny Smith	United Data Machines	12 Main St	❷Boston, MA 02108	Marketing
John Smith	❸UDM	12 Main Street	Boston, MA 02108	Sales
Mary Smith	eSolutions	12 Main St	Boston, ❹RI 02108	Reporting
Fred Jones	Jones Consulting	4 Elm Avenue	Boston, MA 02108	Analysis

❶ The first thing that stands out is the fact that the Name variable is over loaded with both a first and last name rather than having a separate variable for each of them. Next the value Johnny is a valid name as well as a possible nickname for someone named John. Looking at the other values for the first and second record we can deduce that indeed they are the same person. This is an issue that will have to be dealt with in the cleaning process.

❷ A suspected this field is an overloaded value of City, state, and zip code values. Before we can use it to join with the individual values will need to be parsed out.

❸Here we see an abbreviation UDM for United Data Machines. There isn't much one can really do in these cases. It will affect the match rate but since we are going to use multiple variables hopefully it won't be too bad.

❹ Again we see RI for the state code that doesn't match the zipcode value. The value as is would be extremely if not impossible to check and correct this value, however, since the field will be parsed into its constituent values it can be done relatively easily.

```
Data TempB1 ( Drop = I ) ;
Set Department ;
Array CharZ  ( * ) $ 50 Name Company Address CityStateZip Departments  ;
Array CharZ2 ( * ) $ 50 _Name _Company _Address _CityStateZip _Departments  ;❶

ID2 + 1 ;

Do I = 1 By 1 To Dim( CharZ ) ;❷
  CharZ2( I ) = Compbl( UpCase( PrxChange( 's/[[:punct:]]//i' , -1 , Strip( CharZ( I ) ) ) ) ) ;
End ;


_FirstName = Scan( _Name , 1 , ' ' ) ;❸
_LastName  = Scan( _Name , 2 , ' ' ) ;

_City    = Scan( _CityStateZip , 1 , ' ' ) ;❹
_State   = Scan( _CityStateZip , 2 , ' ' ) ;
_ZipCode = Scan( _CityStateZip , 3 , ' ' ) ;

_Company = PrxChange(
's/\b(?:Corporation|Corp|Inc|Incorporated|Company|Co|LTD|LLC|PLLC)\b//i' , 1 ,
Strip( _Company ) ) ;❺

_Address = PrxChange(
's/(STREET|ST|AVENUE|AVE|LANE|LN|PARKWAY|PKWY|WAY|ROAD|RD|DRIVE|DR|PLA
CE|PL|CIRCLE|CIR|COURT|CT|PIKE|TERRACE|TER|TRAIL|TRL|TL|BOULEVARD|BLVD)//i'
, 1 , Strip( _Address ) ) ;❻

_FirstName = Put( _FirstName , $NickNames20. ) ;❼
Run ;


Proc SQL ;❽
Create Table TempB2 As
 Select A.* , UpCase( B.City ) As _City2 , UpCase( B.StateCode ) As _State2
   From TempB1 As A
              Left Join
              SASHELP.ZipCode As B
              On A._Zipcode = Put( B.Zip , Z5. ) ;
Quit ;
```

```
Proc Format ;
 Value $NickNames
   'JACK'    = 'JOHN'
   'JOCK'    = 'JOHN'
   'JOHNNY'  = 'JOHN'
   'JO'      = 'JOANNA'
   'JODY'    = 'JOANNA'
   'JOAN'    = 'JOANNA'
;
Run ;
```

One of the first things one has to do is properly clean up the data; however, by doing so we may or may not also need to keep the original values. To that end I normally create temporary variables where I can hold the cleaned up values at. In the case of this data step I create them with an ❶ Array and prefix each variable with a '_' and the original name of the variable, thus City's temp variable will be _City. This not only gives me a clear distinction between the original values and the cleaned up values but I can drop all of them with a simple Drop = _:.
Now that I have the temporary variables ready I need to copy the values from the old variables to the new variables❷. Since I am not one to waste a perfectly good line of code I make it multitask by doing some basic cleaning up of the values. The PrxChange simply removes everything that isn't a punctuation character, the Upcase the text field which will eliminate

mixed case fields and later on comparisons and finally the Compbl function makes ensures that there aren't multiple blanks in the text fields.
Next I parse the name into its constituent parts (First and Last)❸. I'll note here that if the name was more complicated I would revert to using a Regular Expression to parse the field. There are numerous examples of RegEx patterns online that will do this. At the lines marked with ❹, I parse the City, State, and Zip Code values. The next step is to take care of all those nasty abbreviations in the Company names and Street addresses. To do that, I have used PrxChange to find and eliminate them❺❻. The reason I use PrxChange is one, I just have to list out the values I want to search and eliminate or I could have a data set that contains a variable with all the values I want deleted. Then have a function that opens the data set and builds the RegEx Pattern from the values in the data set. This lends itself to a more generalized system of programs approach. The second reason I like using PrxChange is its ability to use case insensitivity when performing the match, hence the RegEx Patterns using the \i pattern modifiers.
Nicknames pose a significant hurdle to overcome when cleaning up a name field. If you decide to clean up nicknames then I would suggest creating a format of nicknames see code denoted by ❼. The important thing to note, is American names are relatively easy to clean up while foreign names are significantly more difficult to clean. Lastly I would like to mention that if you do decide to go down this rabbit hole, you are very careful as there can exist, depending on your proper to nickname list, a many to many pairing between proper and nicknames. Personally, I have developed a nickname list from many web listing of nickname lists and then culled them down to a one to many pairing. In the end it's a subjective call on your part the important thing is you are consistent with the pairings.
At ❽ is a SQL join with the SAS supplied Zipcode data set. When information on the address is incorrect one can't be 100% certain which part is correct or incorrect without going back to the source to validate the values. So pick one variable as your key and do your best to validate the rest. I tend to like Zip Codes as SAS supplies a data set to check the City and State values against or one can also use one of many zip code data sets supplied from Post Office or many other companies. Now that this data set is about as clean as we can get it lets turn our attention to the other one.

```
Data TempA1 ( Drop = I ) ;
Set Description ;
Array CharZ  ( * ) $ 50 FirstName LastName Company Address City State ZipCode Profession ;
Array CharZ2 ( * ) $ 50 _FirstName _LastName _Company _Address _City _State _ZipCode
_Profession ;

ID1 + 1 ;

Do I = 1 To Dim( CharZ2 ) ;
  CharZ2( I ) = Compbl( UpCase( PrxChange( 's/[[:punct:]]//i' , -1 , Strip( CharZ( I ) ) ) ) ) ;
End ;

_Company = PrxChange(
's/\b(?:Corporation|Corp|Inc|Incorporated|Company|Co|LTD|LLC|PLLC)\b//i' , 1 ,
 Strip( _Company ) ) ;

_Address = PrxChange(
's/(NORTH|SOUTH|EAST|WEST|STREET|ST|AVENUE|AVE|LANE|LN|PARKWAY|PKWY|WAY|
ROAD|RD|DRIVE|DR|PLACE|PL|CIRCLE|CIR|COURT|CT|PIKE|TERRACE|TER|TRAIL|TRL|TL|B
OULEVARD|BLVD)//i' , -1 , Strip( _Address ) ) ;



Addy1 = Scan( _Address , 1 , ' ' ) ;❶

If AnyDigit(Addy1) = 0  Then Do ;❷
  Pattern = CatS( 's/' , Addy1 , '/' , Put( Addy1 , $TextToNum. ) , '/i' ) ;
  _Address = PrxChange( Pattern , 1 , Strip( _Address ) ) ;
End ;



_Company = Compbl( _Company ) ;❸
_Address = Compbl( _Address ) ;

Run ;



Proc SQL ;
Create Table TempA2 As
  Select A.* , UpCase( B.City ) As _City2 , UpCase( B.StateCode ) As _State2
    From TempA1 As A
           Left Join
           SASHELP.ZipCode As B
          On A._Zipcode = Put( B.Zip , Z5. ) ;
Quit ;
```

```
Data NumToText ;
Length Start $ 50 ;
Retain Type 'C' FMTName '$TextToNum' ;

Do Label = 1 To 99 ;
  Start = Compress( UpCase( Put( Label , WordS50. )
) , '-' ) ;
  Output ;
End ;

Run ;


Proc Format
  Lib = Work CntlIn = NumToText ;
Run ;
```

Here we start with the same cleanup process as we did with the other data set. Create temporary variables to hold the cleaned up values, Compress out multiple blanks, UpCase the values, and remove any punctuation characters. However, at the spot marked with❶, where I create another variable called Addy1. This is variable gets the first word of the address field. The reason for this is if you remember, we had some address that started with '12' and some that started off with 'TWELVE'. Now we know If we compare then intuitively our brains will make a match as we know they are meaning the same thing. However, computers don't have that ability, so we need to make both of them either '12' or 'TWELVE'. To facilitate this I tend

to create a format containing the number as the label and use words format to create the written version. Then replace the written version with a numeric version. Which is what we see at ❷. I first check if there is a number if the word held in Addy1. If there isn't then there is a possibility that it is word version of a number, in our case 'Twelve'. I then build a pattern which in the case 'TWELVE would look like: 's/Twelve/12/i'. This then gets fed to the PrxChange function which will search and replace it only 1 time. The RegEx pattern gets recompiled with each observation. If the first word isn't in the Format then nothing happens and the original text is spit back out.

Now you may be thing that after all this data cleaning we can do a simple join and everything will just match perfectly. Well let's look at that scenario:

```
Proc SQL ;
Create Table Need As
 Select A.* , B._Departments
   From TempA2 ( Keep = _: ) As A ,
        TempB2 ( Keep = _: ) As B
    Where ( Strip( A._FirstName ) = Strip( B._FirstName ) )
          And
          ( Strip( A._LastName  ) = Strip( B._LastName  ) )
          And
          ( Strip( A._Company   ) = Strip( B._Company   ) )
          And
          ( Strip( A._City2     ) = Strip( B._City2     ) )
          And
          ( Strip( A._State2    ) = Strip( B._State2    ) )
          And
          ( Strip( A._Zipcode   ) = Strip( B._Zipcode   ) )
         And
         ( Strip( A._Address   ) = Strip( B._Address   ) )
;
Quit ;
```

Results in the following data set:

Obs	_FirstName	_LastName	_Company	_Address	_City	_State	_ZipCode	_Profession	_City2	_State2	_Departments
1	MARY	SMITH	ESOLUTIONS	12 MAIN	BOSTON	MASSACHUSETTS	02108	PROGRAMMER	BOSTON	MA	REPORTING

Hmmmm…. That didn't exactly work out very well. The reason is even though we have cleaned up the data fields they still aren't exactly matching. So what prey tell was the purpose to spending all this time and effort cleaning up these fields you may ask? Well, simply put we spent all this time and effort to get them as close as we could to one another for this next step.

3.) CompGed

So what exactly is CompGed? Well according to the SAS online documentation it states: "Generalized edit distance is a generalization of Levenshtein edit distance, which is a measure of dissimilarity between two strings. The Levenshtein edit distance is the number of deletions, insertions, or replacements of single characters that are required to transform *string-1* into *string-2*.". The Generalized Edit Distance is defined as "the minimum-cost sequence of

operations for constructing *string-1* from *string-2*". That's a whole lot of words to simply say it takes the first string and goes character by character attempting to make it into the second string. Each thing it has to do to make the first string into the second string has a cost associated with it. Think of it as a penalty, the fewer penalties you get the better off you are. A little later we will see how to not only change the amount of these penalties but also how to leverage them to our advantage.

So why not use Compare, Soundex, Spedis or Complev functions? Glad you asked. The long and short of it is Compare only returns the left most character that two strings are different, which wouldn't help us much in determining how close two strings match. It really is looking for when the first instance two strings are different from each other. Soundex works well for phonetic matches however that really only works well if we are talking about comparing English only words. Spedis, while the docs look similar to CompGed is much slower than either CompLev or CompGed. Plus you are stuck with the cost associations with each change to string one to make it into string two. Complev, is a limited version of CompGed, so it works faster, however this speed also means that it doesn't work well with text strings containing more than one word. No the best and one stop function for fuzzy string comparisons is CompGed. It suffers from none of the aforementioned issues.

Syntax:

COMPGED(*string-1*, *string-2* <,*cutoff*>< ,*modifiers*>)

Arguments:

string-1 (Required Argument) specifies a character constant, variable, or expression.

string-2 (Required Argument) specifies a character constant, variable, or expression.

cutoff (Optional Argument) is a numeric constant, variable, or expression. If the actual generalized edit distance is greater than the value of ***cutoff***, the value that is returned is equal to the value of ***cutoff***.

Modifiers (Optional Argument) specifies a character string that can modify the action of the COMPGED function. You can use one or more of the following characters as a valid modifier:

- i or I ignores the case in ***string-1*** and ***string-2***.
- l or L removes leading blanks in ***string-1*** and ***string-2*** before comparing the values.
- n or N removes quotation marks from any argument that is an n-literal and ignores the case of ***string-1*** and ***string-2***.
- : (colon) truncates the longer of ***string-1*** or ***string-2*** to the length of the shorter string, or to one, whichever is greater.

CompGed tries to convert the first string into the second string by moving left to right, character by character between the two strings performing the following operations: Append, Blank, Delete, Double, Fdelete, Finsert, Freplace, Insert, match, Punctuation, Replace, Single Swap, Truncate. Each operation that CompGed performs has a cost associated with performing

each procedure. Unlike Spedis with CompGed the programmer has the ability to change the cost of each operation by using the Call CompCost routine.
What each operation actually does and how it does wit with the associated cost of each operation is well documented in the SAS online documentations. So I will let you the reader go forth and explore. I will note one thing to bear in mind the order that some operations take place in the text strings is important. For example try having it add a letter to the beginning of a string vs the end of a string.

```
Data _Null_ ;
Test = CompGed( 'ABC' , 'AB' ) ;
Put Test= ;

Test = CompGed( 'ABC' , 'BC' ) ;
Put Test= ;
Run ;

Test=50
Test=200
```

The reason that the first CompGed yielded a score of 50 and the second a score of 200 is due to weights that CompGed uses. Adding a letter to the end as in the first case is simple an append worth 50 points. While adding one to the beginning is an FInsert and worth 200. The reason for this is that most of the time people do not screw up the first letter of a word. So odds are the words aren't the same and it penalizes changing the first letter more than the last letter.

Now that we have covered the basics of CompGed let us get back to our regularly scheduled example. At this point we have both files cleaned about as good as we can them without going to extreme measures and hit the law of marginal returns with regards to data cleaning. So now it's time to bring both data sources together. The most logical and easiest is through a SQL join that creates the Cartesian product of both data sets. Once that is done we then can compare each key variable between the two.

```
Proc SQL ; ❶
Create Table Need2 As
  Select ID1 , ID2 , A._FirstName As _FirstNameA , A._LastName As _LastNameA ,
         A._Company As _CompanyA , A._City2 As _City2A , A._State2 As _State2A ,
         A._ZipCode As _ZipCodeA , A._Address As _AddressA ,
         B._FirstName As _FirstNameB , B._LastName As _LastNameB ,
         B._Company As _CompanyB , B._City2 As _City2B , B._State2 As _State2B ,
         B._ZipCode As _ZipCodeB , B._Address As _AddressB
       From TempA2 As A ,
            TempB2 As B
;
Quit ;


Data Temp ; ❷
Set Need2 ;

FNameTest   = CompGed( _FirstNameA , _FirstNameB ) ;
LNameTest   = CompGed( _LastNameA  , _LastNameB  ) ;
CompanyTest = CompGed( _CompanyA   , _CompanyB   ) ;
CityTest    = CompGed( _City2A     , _City2B     ) ;
StateTest   = CompGed( _State2A    , _State2B    ) ;
ZipcodeTest = CompGed( _ZipcodeA   , _ZipcodeB   ) ;
AddressTest = CompGed( _AddressA   , _AddressB   ) ;

Test = Sum( FNameTest, LNameTest, CompanyTest, CityTest, StateTest, ZipCodeTest,
AddressTest ) ;
Run ;
```

```
Proc SQL;
Create Table Temp2 As❸
Select ID1 , ID2
  From Temp
   Group By ID1
    Having Min( Test ) = Test ;

Create Table Temp3 As ❹
 Select A.ID1 , A.ID2 , B.Profession
  From Temp2 As A
     Left Join
     TempB2 As B
     On ( A.ID2 = B.ID2 ) ;

Create Table Need As ❺
 Select A.* , B.*
  From Temp3 As A
     Left Join
     TempA1 ( Drop = _: ) As B
     On ( A.ID1 = B.ID1 ) ;

Quit ;
```

❶The first step in the actual fuzzy matching process we need to join the two data sets together in order to be able to compare the key variables. To do that the easiest method is with a SQL and performing a Cartesian product. A Cartesian product matches each row in one data set with every row in the second data set. Now obviously this has the potential to create

huge data sets. This can create issues with work space and processing time. One method that is commonly used is called Blocking. For more information see the section below on Blocking.

❷Once you have all the observations from the two data sets joined, the next step is to run the actual CompGed tests. Looking at the data step you can see that I ran CompGed on each pair of the Key variables. One of the obvious questions is why did I do it on each set of key variables and not once one a concatenated set of all the key variables. Well the answer is simple I did it for two reasons, first if you concatenate all the key variables and try using CompGed you will get a different score (more likely a higher score) than running CompGed on each set and then summing the final scores. The result is a poorer match rate between data sets because it will take more to transform string one into string two if you concatenated all the keys together than doing a CompGed on each variable and them summing them. Secondly, it allows me to weight the individual keys if I so desire. Just like you can use Cal CompCost to weight the individual operations of CompGed if you do a CompGed on each of the key variables you can weigh the individual scores for each pair of Key variables. So if you knew that say city wasn't very well populated or reliable, you could weight it lower than say first or last name.

❸Once all the scores are calculated for each key pair, you then, in the case of this example need to select the one with the lowest score. This is what the first SQL query is doing; it creates a data set with the key value pair that has the lowest score. If you are lucky you will have a bunch of 0 scores and the rest will be either very small or very large. However, the reality you will likely face is a bunch of scores all over the place. Don't fret, the first thing is to look at your scores and start figuring out a cutoff point at which the match rate starts failing. People have come up with multiple ways to figure out a cutoff point, averages, means, percentiles, etc… Find one that works with your data, hey you may want to go back and use CompCost to adjust CompGed's operation costs or possibly weight some of the variables if some of them are routinely causing your matches to have a high score. The last thing you will need to be paying attention for is ties, yes that is right ties. It is possible that you can have tied scores for two or more observations. When this happens you will need to look and see why this occurred, you may need to see if there is another variable you can add to your key variables to match on or as a last ditch attempt you can segregate these and do the matching by hand. I know hand matching sucks, but hey in the real world you gotta do what you gotta do, know what I mean.

❹Now that we have observations from each data set we want to join on, this step grabs the Profession variable from the Contractors dataset.

❺This final join grabs all the original variables from the Company data set and joins it with the correct Profession from the Contractor data set. You can use the cleaned up version or return the original variable values.

Wholla!!!! We have magically joined two data sources that have no distinct matching key(s). So we are done right? Well not exactly one last thing I'd like to show you is how to clean up the Profession values. You will notice that we have three different spellings for what is basically the same value.

```
Options CMPLIB = Work.Myfuncs ;

Data Need ;
Set Need ;
Profession = Porter( Profession ) ;
Run ;
```

Obs	Profession	New Profession
1	Programming	Program
2	Programming	Program
3	Program	Program
4	Programs	Program

The user defined function Porter is a modified Porter word stemmer. You can see the full code for the function in appendix A. It takes a word and attempts to reduce it down to its root. This becomes very handy when you need to remove suffixes from words. In the case above it reduces the words to a fairly good final word Program. Sometimes the root word is hard to understand, in these cases I find it usually has all the words stemmed correctly. So I end up sorting and then grabbing the first value of the unstemmed word and using that to replace all the words with. I will note that the stemming code does under and over stem some words. So it's not perfect. I am constantly working on improving it.

Blocking

While the example I used for this paper is rather simple odds are in real life your data will be larger and more complicated. So doing the SQL join like I used will result in a rather large if not huge Cartesian product. This is not only impracticable from a work space perspective but also from a processing time. So you will want to use what is known as blocking. Blocking is nothing more than using a key or set of keys that can limit the cross product such that only those observations from each data set that have a reasonable chance to match will be attempted. So for example if you have a variable like State or Country in your data sets you would want to add a Where statement that would say A.State = B.State. This way only observations where the State values are equal will be matched thus limiting the number of observations pairs that needs to be tested.

Fuzzy Matching Strings Containing Multiple Words

Matching text strings that contain multiple words isn't that much different than matching text strings with only one word. The one notable difference is the use of a Stop word list. Stop word list is a list of words that will be filtered out of the text before comparing the two strings. There is no one definitive list of words, I normally use one that I found online that contains the most common words in the English language. Then I can add or delete words from it as needed for each project. Implementation can be done via a PRXChange, user defined function, or TranWrd function.

```
Data _Null_ ;
Text1 = "I live in a forest" ;
Text2 = "I live in the forest" ;
Test = CompGed( Text1 , Text2 ) ;
Put Test= ;

Text1 = PrxChange( 's/I|in|a|the//i' , -1 , Strip( Text1 ) ) ;
Text2 = PrxChange( 's/I|in|a|the//i' , -1 , Strip( Text2 ) ) ;
Test = CompGed( Text1 , Text2 ) ;
Put Test= ;

Run ;

Test = 300
Test = 0
```

As you can see from the code and output above that the sentences say basically the same thing just in two different ways. By eliminating the words (I,in,a,the) very common words in the English language, the CompGed score goes from a 300 to 0 a perfect match. Obviously this is a trivial example to show how a Stop Word list works. However, by using A Stop word list in conjunction with something say like the Porter Stemmer function you have two very powerful tools to improve your match rate.

Acknowledgments

The author thank would like to thank the section chairs and volunteers for all their hard work and allowing me the present this paper! I would also like to thank all the wonderful and smart people on SAS-L listserve.

References

Staum, Paulette, (2007) "Fuzzy Matching using the COMPGED Function, NESUG 2007
Patridge, Charles (1998) "The Fuzzy Feeling SAS Provides: Electronic Matching of Records without Common Keys", SAS Observations
SAS Online Documentation

Trademark Citations

About the Authors

Toby Dunn

Toby Dunn has been programming in SAS for the last 13 years. He has applied his skills to multiple fields such as Banking, Education, Government, Healthcare, and Cloud Computing industries. He has numerous SGF, SUGI, and SUG papers covering a wide group of SAS programming topics.

Comments and suggestions can be sent to:

Toby Dunn
Dunn Consulting
E-mail: tobydunn@hotmail.com

Appendix A

```
Proc FCMP
 Outlib = DSN.MyFuncs.Utility ;
 Function VCCHeck( Word $ ) ;
  Length VC $ 32000 ;
  VC = LowCase( Word ) ;
  VC = PrxChange( 's/[aeiou]/V/'    , -1 , Strip( VC ) ) ;
  VC = PrxChange( 's/[^aeiouyV]/C/' , -1 , Strip( VC ) ) ;
  VC = PrxChange( 's/(?<=V)y/C/'    , -1 , Strip( VC ) ) ;
  Vc = PrxChange( 's/y/V/'          , -1 , Strip( VC ) ) ;
  VC = PrxChange( 's/VC/X/'         , -1 , Strip( VC ) ) ;

  Return( CountC( VC , 'X' ) ) ;

 EndSub ;


 Function CVC( Word $ ) ;
  Length VC $ 32000 ;
  VC = LowCase( Word ) ;
  VC = PrxChange( 's/[aeiou]/V/'    , -1 , Strip( VC ) ) ;
  VC = PrxChange( 's/[^aeiouyV]/C/' , -1 , Strip( VC ) ) ;
  VC = PrxChange( 's/(?<=V)y/C/'    , -1 , Strip( VC ) ) ;
  Vc = PrxChange( 's/y/V/'          , -1 , Strip( VC ) ) ;

  If ( PrxMatch( '/CVC/' , Strip( VC ) ) > 0 )
    And
    ( PrxMatch( '/[^wxy]$/' , Strip( Word ) ) ) Then Do ;

    Return( PrxMatch( '/CVC/' , Strip( VC ) ) > 0 ) ;
  End ;
  Else Do ;
    Return( 0 ) ;
  End ;

 EndSub ;


 Function Porter( Word $ ) $ ;
 Length Word2 Temp Temp3 Temp4 Temp1ab TEmp1bb I $ 100 ;
```

```
Word2 = LowCase( Word ) ;

If Length( Word ) < 3 Then Call Missing( Word2 ) ;

Word2 = PrxChange( 's/(?<=[aeiou])y/Y/' , 1 , Strip( Word2 ) ) ;


R1Pattern = PrxParse( '/[aeiou][^aeiouyY](.*)/' )  ;
R1Match   = PrxMatch( R1Pattern , Strip( Word2 ) ) ;
R1        = PrxPosN( R1Pattern , 1 , Word2 ) ;

R2Pattern = PrxParse( '/[aeiou][^aeiouyY](.*)/' ) ;
R2Match   = PrxMatch( R2Pattern , Strip( R1 )   ) ;
R2        = PrxPosN( R2Pattern , 1 , R1 ) ;




If Not ( PrxMatch( '/(?:u|s)s$/' , Strip( Word2 ) ) ) Then Do ;

  If PrxMatch( '/sses$/' , Strip( Word2 ) ) Then
  Word2 = PrxChange( 's/sses$/ss/'              , 1 , Strip( Word2 ) ) ;

  Else If PrxMatch( '/(?<=..)ie[ds]$/' , Strip( Word2 ) ) Then
  Word2 = PrxChange( 's/(?<=..)ie[ds]$/i/'      , 1 , Strip( Word2 ) ) ;

  Else If PrxMatch( '/ie[ds]$/' , Strip( Word2 ) ) Then
  Word2 = PrxChange( 's/ie[ds]$/ie/'            , 1 , Strip( Word2 ) ) ;

  Else If PrxMatch( '/s$/' , Strip( Word2 ) ) Then
  Word2 = PrxChange( 's/(?<!s)s$/$1/' , 1 , Strip( Word2 ) ) ;

End ;




If ( PrxMatch( '/(eed(ly)?)$/' , Strip( Word2 ) ) ) Then Do ;
   Temp1BA = PrxChange( 's/^(.*?)(eed(ly)?)$/$1/' , 1 , Strip( Word2 ) ) ;

   If ( VCCheck( Temp1BA ) > 0 ) Then
   Word2 = PrxChange( 's/(.*?)(eed(ly)?)$/$1ee/' , 1 , Strip( Word2 ) ) ;
End ;
```

```
If ( PrxMatch( '/(ed|ing)(ly)?$/' , Strip( Word2 ) ) ) Then Do ;

  Temp1BB = PrxChange( 's/^(.*?)((ed|ing)(ly)?)$/$1/' , 1 , Strip( Word2 ) ) ;


  If PrxMatch( '/[aeiouy]/' , Strip( Temp1BB ) ) > 0 Then

  Word2 = PrxChange( 's/^(.*?)((ed|ing)(ly)?)$/$1/' , 1 , Strip( Word2 ) ) ;

  If PrxMatch( '/(at|bl|iz)$/' , Strip( Word2 ) ) Then
  Word2 = CatS( Word2 , 'e' ) ;

  Word2 = PrxChange( 's/((.*?)([^aeiouylsz]))\1$/$1/' , 1 , Strip( Word2 ) ) ;

  If ( VCCheck( Word2 ) = 1 ) And ( CVC( Substr( Word2 , 1 , Length( Word2 ) - 1 ) ) ) Then
  Word2 = CatS( Word2 , 'e' ) ;

End ;




Word2 = PrxChange( 's/(?<=.[^aeiou])[y|Y]$/i/' , 1 , Strip( Word2 ) ) ;




Word2 = PrxChange( 's/([aeiouy][^aeiou].*?)tional$/$1tion/'          , 1 , Strip( Word2 ) ) ;
Word2 = PrxChange( 's/([aeiouy][^aeiou].*?)enci$/$1ence/'           , 1 , Strip( Word2 ) ) ;
Word2 = PrxChange( 's/([aeiouy][^aeiou].*?)anci$/$1ance/'           , 1 , Strip( Word2 ) ) ;
Word2 = PrxChange( 's/([aeiouy][^aeiou].*?)abli$/$1able/'           , 1 , Strip( Word2 ) ) ;
Word2 = PrxChange( 's/([aeiouy][^aeiou].*?)entli$/$1ent/'           , 1 , Strip( Word2 ) ) ;
Word2 = PrxChange( 's/([aeiouy][^aeiou].*?)izer|ization$/$1ize/'    , 1 , Strip( Word2 ) ) ;
Word2 = PrxChange( 's/([aeiouy][^aeiou].*?)(ational|ation|ator)$/$1ate/' , 1 , Strip( Word2 ) )
;
Word2 = PrxChange( 's/([aeiouy][^aeiou].*?)(alism|aliti|alli)$/$1al/'  , 1 , Strip( Word2 ) ) ;
Word2 = PrxChange( 's/([aeiouy][^aeiou].*?)fulness$/$1ful/'         , 1 , Strip( Word2 ) ) ;
Word2 = PrxChange( 's/([aeiouy][^aeiou].*?)(ousli|ousness)$/$1ous/'  , 1 , Strip( Word2 ) ) ;
Word2 = PrxChange( 's/([aeiouy][^aeiou].*?)(iveness|iviti)$/$1ive/'  , 1 , Strip( Word2 ) ) ;
```

```
Word2 = PrxChange( 's/([aeiouy][^aeiou].*?)(biliti|bli)$/$1ble/'        , 1 , Strip( Word2 ) ) ;
Word2 = PrxChange( 's/([aeiouy][^aeiou].*?)fulli$/$1ful/'               , 1 , Strip( Word2 ) ) ;
Word2 = PrxChange( 's/([aeiouy][^aeiou].*?)lessli$/$1less/'             , 1 , Strip( Word2 ) ) ;
Word2 = PrxChange( 's/({aeiouy][^aeiou].*?l)ogi$/$1og/'                 , 1 , Strip( Word2 ) ) ;
Word2 = PrxChange( 's/([aeiouy][^aeiou].*?[cdeghkmnrt])li$/$1/'         , 1 , Strip( Word2 ) ) ;



Word2 = PrxChange( 's/([aeiouy][^aeiou].*?)ational/$1ate/'          , 1 , Strip( Word2 ) ) ;
Word2 = PrxChange( 's/([aeiouy][^aeiou].*?)tional/$1tion/'          , 1 , Strip( Word2 ) ) ;
Word2 = PrxChange( 's/([aeiouy][^aeiou].*?)alize/$1al/'             , 1 , Strip( Word2 ) ) ;
Word2 = PrxChange( 's/([aeiouy][^aeiou].*?)(ic(ate|iti|cal))/$1ic/' , 1 , Strip( Word2 ) ) ;
Word2 = PrxChange( 's/([aeiouy][^aeiou].*?)(Full|ness)/$1/'         , 1 , Strip( Word2 ) ) ;
Word2 = PrxChange( 's/([aeiouy][^aeiou].*?)tional/$1tion/' , 1 , Strip( Word2 ) ) ;



If ( PrxMatch( '/ative$/' , Strip( Word2 ) ) ) Then Do ;

  Temp3 = PrxChange( 's/^(.*?)ative$/$1/' , 1 , Strip( Word2 ) ) ;

  If VCCheck( Temp3 ) > 1 Then
  Word2 = PrxChange( 's/ative$//' , 1 , Strip( Word2 ) ) ;

End ;



Temp = CatX( ',' , 'ement','ance','ence','able','ible','ment',
      'ant','ent','ism','ate','iti','ous','ive','ize',
      'ion','al','er','ic' );


Do K = 1 To COuntC( Temp , ',' ) + 1 ;

  I = Scan( Temp , K , ',' ) ;

  Temp4 = PrxChange( CatS( 's/^(.*?)' , I , '$/$1/') ,  1 , Strip( Word2 ) ) ;
```

```
   If ( VCCheck( Temp4 ) > 1 ) And ( I NE 'ion' ) And ( PrxMatch( CatS( '/' , I , '$/' ) , Strip( Word2 )
) ) Then Do ;
     Word2 = PrxChange(  CatS( 's/' , I , '$//' )     , 1 , Strip( Word2 ) ) ;
           End ;
   Else If ( VCCheck( Temp4 ) > 1 )
         And
         ( I EQ 'ion' )
         And
         ( PrxMatch( '/ion$/' , Strip( Word2 ) ) ) Then Do ;
     Word2 = PrxChange(  CatS( 's/(?<=[st])' , I , '$//' ) , 1 , Strip( Word2 ) ) ;
           End ;
  End ;



  If ( PrxMatch( '/e$/' , Strip( Word2 ) ) ) Then Do ;

   If VCCheck( Substr( Word2 , 1 , Length( Word2 ) - 1 ) )  > 1 Then
   Word2 = PrxChange( 's/e$//' , 1 , Strip( Word2 ) ) ;

   If VCCHeck( Substr( Word2 , 1 , Length( Word2 ) - 1 ) ) = 1
   And CVC( Substr( Word2 , 1 , Length( Word2 ) - 1 ) ) Then
   Word2 = PrxChange( 's/e$//' , 1 , Strip( Word2 ) ) ;
  End ;



  If ( PrxMatch( '/ll$/' , Strip( Word2 ) )
    And
    VCCHeck( Substr( Word2 , 1 , Length( Word2 ) - 2 ) ) > 1 ) Then
   Word2 = PrxChange( 's/ll$//' , 1 , Strip( Word2 ) ) ;


  If Length( Word ) < 3 Then Word2 = Word ;

  Return( UpCase( Word2 ) ) ;

EndSub ;

Run ;
```