Probabilistic Record Linkage in SAS®

Glenn Wright, M.P.A, Kaiser Permanente, Oakland, CA

ABSTRACT

Record linkage is the process of deciding which records from one or more databases refer to the same entity, even when those records do not match exactly on any combination of identifiers. For example, we may wish to determine that a record for “Glenn Thomas Wright” living at “1234 West Fifth Street” refers to the same person as a record for “Glen T Wright” living at “1234 Fifth Street,” but not the same person as a record for “Denis Hulett” living at “678 Ninth Street.” Probabilistic record linkage is a family of record linkage techniques that assigns similarity scores to pairs of records and treats all pairs that score above a certain threshold as matches. This paper describes a method for implementing probabilistic record linkage in SAS, using PROC SQL and other tools.

INTRODUCTION

SAS programmers often face the task of linking together records that do not share a common identifier, but nevertheless refer to the same entity. For example, case reports for infectious diseases often have identifying information such as patient names and dates of birth, but no patient ID. This problem is variously known as “record linkage” or “entity resolution” when it applies to records across databases; “deduplication” or “merge-purge” when it applies to records within one database.

One popular and effective approach is to calculate a similarity score for each pair of records, and to treat all records with scores above a certain threshold as matches. This method, which is called “probabilistic record linkage,” is the approach used by most record linkage software, including free programs such as Link Plus (provided by the CDC Division of Cancer Prevention) and Link King (a SAS-based tool developed by the Substance Abuse and Mental Health Services Administration.) However, those who desire more control over the details of matching may wish to implement a probabilistic record matching process directly in SAS. The problem is deceptively difficult, and this paper presents one method for solving it.

1. The process of probabilistic record linkage requires five steps:
2. Cleaning, standardizing, and otherwise preparing the data.
3. Comparing pairs of records.
4. Calculating similarity scores.
5. Choosing a threshold.
6. Encoding the results.

The example programs in this paper will consider a deduplication task on a single data set, which is slightly simpler to describe that record linkage involving multiple data sets.

CLEANING AND STANDARDIZING DATA

The first step of the process is common to many data analysis tasks: We must clean and standardize the data. This step might include making sure that names are capitalized consistently, that addresses use “ST.” as an abbreviation for street names, that missing values are coded consistently, and so on. This step usually involves basic DATA step functions, so this paper will not explore it in detail. However, there are two considerations that might not be obvious. First, it is often useful at this stage to attach a permanent observation number to each record, if they do not already have record ID numbers. Second, this step is generally **not** a good time to correct spelling errors; in a large database, the number of misspellings and variations will be too large to fix manually.

A typical data cleaning step may look something like this:

```
data mydata_clean;
    set mydata;
    obs_num = _n_;
    *remove punctuation and standardize capitalization;
    fname = upcase(compress(tranwrd(fname,'-',' '),,'p'));
    lname = upcase(compress(tranwrd(lname,'-',' '),,'p'));
    if zip in('00000','99999') then zip = ' ';
run;
```

COMPARING PAIRS OF RECORDS

Probabilistic record linkage requires that we compare every record in one data set against every record in another data set; deduplication requires that we compare every record in a data set to every other record in the same data set. This process is called a "Cartesian join", and it is an example of a "many-to-many merge" problem – at one time a source of great frustration for SAS programmers.

Fortunately, recent versions of SAS include the PROC SQL procedure, which supports many-to-many merges using the JOIN clause. Full documentation of the syntax for SQL JOINs is readily available online, so this paper will focus on how to use the technique for record linkage.

A PROC SQL step that joins every record in a data set to every other record in the same data (a "self-join") set might look like this:

```
proc sql;
    create table cartesian_join as
    select a.obs_num as obs_num_1, b.obs_num as obs_num_2,
           a.fname as fname_1, b.fname as fname_2,
           a.lname as lname_1, b.lname as lname_2,
           a.dob as dob_1, b.dob as dob_2
    from   mydata_clean as a INNER JOIN mydata_clean as b
    on     a.obs_num > b.obs_num
    ;
quit;
```

Note the unusual ON clause: Instead of testing for the equality of some identifier variable – the most common way of merging – we are testing whether the observation number in the left-hand data set is greater than the observation number in the right-hand data set. That means every observation will be compared to every other observation exactly once; there is no need to compare B to A once we have already compared A to B. Also note that we save the names, observation numbers, and dates of birth from both data sets, for later inspection.

CALCULATING SIMILARITY SCORES

The most involved part of the probabilistic record linkage process is calculating similarity scores. In the simplest case, we might use CASE / WHEN statements – the PROC SQL equivalent of IF / THEN / ELSE – to assign scores based on whether fields from the two records match exactly; then use the CALCULATED keyword to add together the scores for each field to get a total score for each pair of records:

```
proc sql;
    create table cartesian_join as
    select a.obs_num as obs_num_1, b.obs_num as obs_num_2,
           a.fname as fname_1, b.fname as fname_2,
           a.lname as lname_1, b.lname as lname_2,
           a.dob as dob_1, b.dob as dob_2

           case when a.fname = b.fname then 10 else 0 end as fname_score,
           case when a.lname = b.lname then 10 else 0 end as lname_score,
           case when a.dob = b.dob then 10 else 0 end as dob_score,
           case when a.sex = b.sex then 10 else 0 end as sex_score,
           case when a.race = b.race then 10 else 0 end as race_score,

           calculated fname_score + calculated lname_score +
                  calculated dob_score + calculated sex_score +
                  calculated race_score as total_score

    from   mydata_clean as a INNER JOIN mydata_clean as b
    on     a.obs_num > b.obs_num
    ;
quit;
```

In this example, a pairs of records that match on all five variables (first name, last name, date of birth, sex, and race) will get a total score of 50 points; a pair of records that share date of birth and sex but no other values will get 20 points, and a pair of records that disagree on all values with get no points.

But in most cases, we want to assign different scores for matches on different fields – for example, a match on a distinctive field such as first name should probably be worth more than a match on a field like sex, that has only a few possible values. A formal method for assigning scores was described by Fellegi and Sunter (1969). Most record linkage software relies on some variation of the Fellegi-Sunter Model, but the Model relies on complex proofs and iterative processes that are beyond the scope of this paper. Nevertheless, most of the findings of the formal model can be simplified into intuitive principles. Consider a field, such as first name or sex, and compare the values between two records:

1. If the two values **agree**, that provides evidence that the two records refer to the same entity.
2. If the two values **disagree**, that evidence that the two records **do not** refer the same entity.
3. If the value is **missing** for either of the two records, that provides **no** evidence either way.

Some statistically inspired rules of thumb for choosing scores:

1. If the field is missing for either record, assign the pair zero points for that field.
2. If the fields for the two records agree, assign the pair some number of positive points. Fields with many distinct values, such as names or dates of birth, should be worth more points than fields with few values, such as sex or race.
3. If the fields for the two records **disagree**, assign the pair some number of **negative** points. For disagreement, what matters is not how many distinct values exist for the field, but how accurate and clean the field is – for example, if we know that the sex field our database is extremely reliable, then two records that disagree on sex most likely do not refer to the same person, and we should give that pair a large number of negative points. On the other hand, if the field for race is very messy, two records that disagree on race should only get a small number of negative points.

Informed by these guidelines, we might write a PROC SQL step more like this:

```
proc sql;
    create table similarity_scores as
    select a.obs_num as obs_num_1, b.obs_num as obs_num_2,
           a.fname as fname_1, b.fname as fname_2,
           a.lname as lname_1, b.lname as lname_2,
           a.dob as dob_1, b.dob as dob_2,

    case   when a.fname = '' or b.fname = '' then 0
                  when a.fname = b.fname then 11
                  else -5 end as fname_score,

           case   when a.lname = '' or b.lname = '' then 0
                  when a.lname = b.lname then 10
                  else -6 end as lname_score,

           case   when a.dob = . or b.dob = . then 0
                  when a.dob = b.dob then 12
                  else -6 end as dob_score

           case   when a.sex = 'U' or b.sex = 'U' then 0
                  when a.sex = b.sex then 2
                  else -5 end as sex_score,

           case   when a.race = 'U' or b.race = 'U' then 0
                  when a.race = b.race then 3
                  else -3 end as race_score

           calculated fname_score + calculated lname_score +
                  calculated dob_score + calculated sex_score +
                  calculated race_score as total_score

    from   mydata_clean as a INNER JOIN mydata_clean as b
    on     a.obs_num > b.obs_num
    ;
quit;
```

The scores shown in this example resemble those formally estimated for a real database of disease case reports, and may be a reasonable starting point for other analyses.

WEIGHTING BY FREQUENCY (OPTIONAL)

If we are willing to make the analysis more complex, we could account for the fact that some field values are much more common than others. For example, the first name “MARIA” is much, much more common than the first name “CIRCE”. Intuitively, then, two records that share the name CIRCE should get more points than two records that share the name MARIA. Skipping this step simplifies the analysis, but do realize that the problem is a large one – the most common names in a typical database may occurs thousands of times as often as the least common names.

The Fellegi-Sunter Model has as formula for assigning points based on the frequency of the value. Specifically, it assigns a number of points equal to the negative base-two logarithm of the value’s frequency. This formula is simple to implement in SAS:

```
data _null_;
    name = 'MARIA';
    freq = 0.05;
    name_score = -log2(freq);
    put name= name_score=;
    name = 'CIRCE';
    freq = 0.00001;
    name_score = -log2(freq);
    put name= name_score=;
run;
```

Notice that a match on the rare name is worth almost four times as many points as a match on the common name. This is makes sense for a large database – in a database of disease case reports for the state of California, for example, there may be many records that share common names like MARIA GONZALEZ and yet do not belong to the same person. Only if other fields agree as well – dates of birth or addresses, perhaps – should the total number of points be large enough to count as a match.

Using these weighted scores in a PROC SQL step can be tricky. The frequencies themselves are easy to calculate using PROC FREQ with an OUTPUT statement:

```
proc freq data = mydata_clean;
    tables fname;
    output out = fname_freq;
run;
```

These values can be attached back onto the original, cleaned data set using a MERGE statement or a PROC SQL JOIN:

```
proc sort data = fname_freq;
    by fname;
run;
data mydata_freqs;
    merge mydata_clean fname_freq;
    by fname;
    fname_points = -log2(freq);
run;
```

Alternatively, a programmer comfortable with several advanced SAS techniques could store the frequency values as an INFORMAT, then use INPUT() as a lookup function to attach them to the original data. This technique is very useful, but it is beyond the scope of this paper. For help with this approach, see “Table Look-up: Techniques Beyond the Obvious”, by Croonen and Theuwissen.

Once the specific score is attached to the original data set, we can change the positive points assigned for a match from a fixed value, like so:

```
case       when a.fname = '' or b.fname = '' then 0
           when a.fname = b.fname then 11
           else -5 end as fname_score,
```

…to a variable value, like this:

```
case       when a.fname = '' or b.fname = '' then 0
           when a.fname = b.fname then a.fname_points
           else -5 end as fname_score,
```

Note that specific weighting is useful only when the frequency of values varies. For values that are distributed more-or-less uniformly, like social security numbers or dates of birth, specific weighting is not worth the effort. What about fields that have only a handful of values, distributed unevenly – for example, a database that is two-thirds female and one-third male? In that case, we can easily calculate the appropriate number of points…

```
data _null_;
    sex = 'F';
    freq = 2/3;
    score = -log2(freq);
    put sex= score=;
    sex = 'M';
    freq = 1/3;
    score = -log2(freq);
    put sex= score=;
run;
```

…and hand-code the results directly in the PROC SQL code:

```
case       when a.sex = 'U' or b.sex = 'U' then 0
           when a.sex = 'M' and b.sex = 'M' then 1.5849
           when a.sex = 'F' and b.sex = 'F' then 0.5849
           else -5 end as sex_score,
```

INEXACT MATCHING (OPTIONAL)

Another way to extend the method is to give points not only for exact matches, but also for inexact matches – misspelled names, nicknames, transposed days and months in birth dates, nearby locations, et cetera. PROC SQL's CASE / WHEN logic makes it convenient to set up a hierarchy that assigns full points for exact matches and partial points for inexact matches. However, telling SAS how to recognize inexact matches is more art than science.

Dates are some of the easiest fields for which we can code inexact matches. Certain data entry errors are especially common: A typo in the day, month, or year field, or switching the day and the year. Here is a CASE / WHEN clause that uses the DAY(), MONTH(), and YEAR() functions to check for such mistakes:

```
case       when a.dob = . or b.dob = . then 0
           /* full points for an exact match */
           when a.dob = b.dob then 12
           /* near-full points for same year, but transposed day and month */
           when year(a.dob) = year(b.dob) and month(a.dob) = day(b.dob) and
                        day(a.dob) = month(b.dob) then 11
           /* reduced points when only two of the three fields match month */
           when year(a.dob) = year(b.dob) and month(a.dob) = month(b.dob) then 9
           when year(a.dob) = year(b.dob) and day(a.dob) = day(b.dob) then 9
           when month(a.dob) = month(b.dob) and day(a.dob) = day(b.dob) then 9
           /* negative points for disagreement */
           else -7 end as dob_score
```

Another simple form of inexact matching applies to geographic data. If you have geocoded address data with X and Y coordinates, you can use the distance formula as a form of inexact matching (you may remember this formula from high school geometry class):

```
    case   when a.address = '' or b.address = '' then 0
           /* full points for matching addresses */
           when a.address = b.address then 20
           /* reduced points for addresses within 25 miles - watch the units! */
           when sqrt((a.x-b.x)**2+(a.y-b.y)**2) <= 25 then 10
           /* greatly reduced points for addresses within 50 miles */
           when sqrt((a.x-b.x)**2+(a.y-b.y)**2) <= 50 then 5
           /* negative points for distant addresses */
           else -4 end as address_score
```

Inexact matching may also be useful for categorical variables, when some of the categories are ambiguous or overlapping, or are not used consistently. For example, we may want to treat race values of "Other" or "Multiracial" specially – we give positive points if they match, but we do not penalize for non-matching:

```
case       when a.race= 'U' or b.race = 'U' then 0
           when a.race = b.race then 2
           when a.race in ('M','O') or b.race in ('M','O') then 0
           else -3 end as race_score
```

The trickiest variables for inexact matching, however, are names and other character fields. The variety of problems is discouraging: Nicknames, misspellings, alternate spellings, hyphenated maiden names, middle names or initials that spill into first name or last name fields, and so on. What follows is an incomplete set of solutions; fully exploring the possibilities is far beyond the scope of this paper.

First, standardize and clean your data carefully. I suggest capitalizing all letters, converting hyphens to spaces, and removing all other punctuation and numbers. These steps will not solve more complex problems, but they will save you many headaches.

Second, check to see whether one name in the pair is a substring of the other name in the pair. The INDEX, INDEXW, FIND, and FINDW functions can help with this task. However, watch out for 1- or 2-letter names (e.g. initials) – you probably do not want to give as many points to the last name "G" for matching "GONZALEZ" as to the first name "ANA" for matching "ANA LUIS."

Third, search online for a nickname database, and reformat it so you can merge a list of possible nicknames onto each record. For example, if the record has the value "PATRICIA" in the first name field, you could create a new field called NICKNAMES that has the value "PAT PATTY PATTIE TRICIA TRISHA TRISH". You can then use function like INDEXW or FINDW to determine whether a name is found within the nickname list. Alternately, you could try to simply standardize away all the nicknames, but this creates some dilemmas. Do you standardize PAT to PATRICK or PATRICIA? Or, do you standardize PATRICK and PATRICA to PAT? And if so, does that mean you treat the names PATRICK and PATRICIA as matches?

Fourth, use "edit-distance" metrics and related functions. Edit-distance metrics are a family of functions that measure the similarity of character values by comparing two strings of characters and calculate how many operations it takes to transform one string into the other. The most well-known edit-distance metric is the Levenshtein distance, which counts how characters one must add, delete, or replace to transform one string into another. For example, the Levenshtein distance between "KITTEN" and "SMITTEN" is 2: The first operation required is to replace the "K" with an "M"; the second is to add an "S" to the beginning of the word.

SAS implements the Levenshtein edit distance in the COMPLEV function:

```
data example;
    name1 = 'GONZALEZ';
    name2 = 'GONZAELZ';
    lev = complev(name1,name2);
    put name1= name2= lev=;
    name1 = 'ANA';
    name2 = 'ANT';
    lev = complev(name1,name2);
    put name1= name2= lev=;
run;
```

I have found it useful to "normalize" the edit distance by dividing it by the length of one of the character values. This corrects for the fact that longer strings, almost by definition, require more operations to transform:

```
data example;
    name1 = 'GONZALEZ';
    name2 = 'GONZAELZ';
    normlev = complev(name1,name2)/max(length(name1),length(name2));
    put name1= name2= normlev=;
    name1 = 'ANA';
    name2 = 'ANT';
    normlev = complev(name1,name2)/max(length(name1),length(name2));
    put name1= name2= normlev=;
run;
```

The lower the Levenshtein edit distance, the closer the match – perfect matches have a distance of 0. In my experience, the best cutoff for the length-normalized Levenshtein distance is somewhere between 0.25 and 0.3 (i.e. once somewhere between one-quarter and three-tenths of the letters are different, we no longer consider the two strings to be approximate matches.) Another possible metric is the Jaro-Winkler distance, developed at the United States Census Bureau (see Winkler, 1990). However, implementing this metric in SAS may be difficult.

Once we have decided which methods we are going to use to find inexactly matching names, we can assemble them into a hierarchical CASE / WHEN statement. But there is one last wrinkle: If we are using both inexact matching **and** specific weighting, the specific weights for the two field values will be different. I recommend using the smaller of the two values; we do not want to give the "rare" misspelled name MAREA a high score for matching the common name MARIA.

```
case       when a.fname = '' or b.fname = '' then 0
           when a.fname = b.fname then a.fname_points
           when find(a.nicknames,b.fname) or find(b.nicknames,a.fname)
                     then 0.9*min(a.fname_points,b.fname_points)
           when findw(a.fname,b.fname) or findw(b.fname,a.fname)
                  then 0.9*min(a.fname_points,b.fname_points)
           when complev(a.fname,b.fname)/max(length(a.fname),length(b.fname))
                     <=0.25 then 0.8*min(a.fname_points,b.fname_points)
           when complev(a.fname,b.fname)/max(length(a.fname),length(b.fname))
                  < 0.35 then 0.5*min(a.fname_points,b.fname_points)
           else -5 end as fname_score
```

CHOOSING A THRESHOLD

Once we have calculated similarity scores, we need some way to choose the threshold above which two records are considered a match. This can be done by trial and error, but there are a few shortcuts that can improve the process. For simplicity's sake, consider the relatively stripped-down PROC SQL code we used earlier in the paper:

```
proc sql;
    create table similarity_scores as
    select a.obs_num as obs_num_1, b.obs_num as obs_num_2,
           a.fname as fname_1, b.fname as fname_2,
           a.lname as lname_1, b.lname as lname_2,
           a.dob as dob_1, b.dob as dob_2,

    case   when a.fname = '' or b.fname = '' then 0
                  when a.fname = b.fname then 11
                  else -5 end as fname_score,

           case   when a.lname = '' or b.lname = '' then 0
                  when a.lname = b.lname then 10
                  else -6 end as lname_score,

           case   when a.dob = . or b.dob = . then 0
                  when a.dob = b.dob then 12
                  else -6 end as dob_score
```

```
            case   when a.sex = 'U' or b.sex = 'U' then 0
                   when a.sex = b.sex then 2
                   else -5 end as sex_score,


            case   when a.race = 'U' or b.race = 'U' then 0
                   when a.race = b.race then 3
                   else -3 end as race_score

            calculated fname_score + calculated lname_score +
                   calculated dob_score + calculated sex_score +
                   calculated race_score as total_score

    from   mydata_clean as a INNER JOIN mydata_clean as b
    on     a.obs_num > b.obs_num
    ;
quit;
```

Every observation in the output data set SIMILARITY_SCORES represents a pair of records from the original data set MYDATA_CLEAN and contains a value called TOTAL_SCORE. Observations with high TOTAL_SCOREs probably represent pairs that **do** represent the same person, and observations with low TOTAL_SCOREs probably represent pairs that do **not** represent the same person. But how do we know where to draw the line?

One helpful step is to plot a histogram of the values of TOTAL_SCORE, using PROC UNIVARIATE:

```
proc univariate data = similarity_scores;
    var total_score;
    histogram total_score;
run;
```

If all goes well, we see something like this:

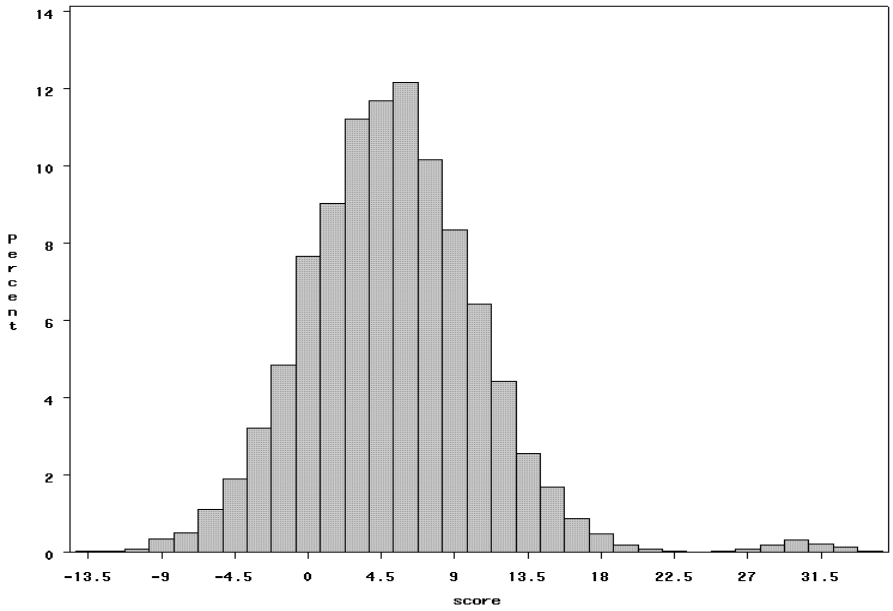

Figure 1. Histogram of Similarity Scores

This bimodal distribution is a sign that our method is working correctly – there should be a large peak in the lower range of scores and a much smaller peak in the higher range. Why is this so? Because the histogram shows every

pair of observations, both those that do refer to the same entity and those that do not – and the number of non-matching pairs is much greater than the number of matching pairs. In other words, two randomly selected records most likely do not refer to the same entity. If we have chosen our methods well, most of the true matches fall under the small, high-scoring peak of the histogram, whereas most of the false matches fall under the large, low-scoring peak.

In order to capture as many true matches and exclude as many false matches as possible, we ought to choose a threshold somewhere in the "trough" between the two peaks – around 25 or so in the example above. We might choose a relatively high or low threshold depending on the needs of our analysis – whether we want to be especially strict or lenient in how w infer matches.

Once we have chosen a threshold, we might want to double-check a small subset of the observations near that threshold visually, to see how many appear (to the naked eye and human judgment), to be true or false matches. This often reveals insights into how we might either refine our inexact matching algorithms or choose a better threshold:

```
proc print data = similarity_scores;
    where ranuni(0) <= 0.001 and 24 <= total_score <= 25;
run;
```

ENCODING RESULTS

Once we have determined which records are likely matches and which likely non-matches, we usually want to encode the results for use in future analyses. In some cases just finding pairs might be enough; for example, if we are testing for reinfection with a particular disease within a specific time frame. But in most cases, the best approach is to create a new "common identifier" that uniquely identifies which observations we think belong to which person. This step is surprisingly difficult in SAS – beyond the scope of this paper – but it is described in "Transitive Record Linkage in SAS using Hash Objects" (Wright and Hulett, 2010).

OPTIMIZING PERFORMANCE (DEFINITELY NOT OPTIONAL)

We have now discussed every phase of the record linkage process, but there is one more topic we must consider before we put the code together and run it: Performance.

For some SAS tasks, optimizing performance is optional. Often, if the process takes a long time, we can simply run it overnight and examine the results in the morning. But in the case of record linkage, optimization is usually unavoidable. Why? Because a the size of a Cartesian join grows quadratically with the size of the data sets involved. Comparing two data sets with a thousand records each yields a million pairs; comparing two data sets with ten thousand records each yields one hundred million pairs. If we do not take every possible step to conserve resources, our computer might take weeks to perform the calculations, and even then, our hard drive might run out of space and crash the PROC SQL step.

There are three main techniques we can use to conserve resources: Early thresholding, indexing, and blocking.

Early thresholding is simple: We add a WHERE clause to the PROC SQL step so that it immediately drops observations that do not meet some minimal similarity score. Generally we want to choose a score lower than the threshold we will later use to determine true matches, so that we still keep the "near misses" and examine them visually. For example, if we expect to treat record with a score of 35 or above as a match, we might use an early threshold of 25:

```
where calculated score >= 25
```

Early thresholding tells SAS to drop most of the pairs (especially non-matches) immediately rather than writing them to the hard disk or network drive, which means the process will take vastly less disk space.

The next technique is indexing. SAS indexes are like entries in a card catalog in a library – they take up some space and take some time to build, but once created they let SAS find observations much faster for purposes of a SORT or a JOIN. Indexes can be created in a DATA step or in PROC SQL; here is one syntax for adding them as part of the "data cleaning and standardization" step:

```
data mydata_clean (index = (obs_num fname lname dob zip));
    set mydata;
    obs_num = _n_;
    fname = upcase(compress(tranwrd(fname,'-',' '),,'p'));
    lname = upcase(compress(tranwrd(lname,'-',' '),,'p'));
run;
```

Once the indexes have been created, SAS determines automatically when it should use them. Guidelines for using SAS indexes can be found online, in papers such as "The Basics of Using SAS Indexes" by Michael A. Raithel.

The third and most complicated optimization technique is blocking. Blocking means that before SAS performs the complicated series of CASE / WHEN tests and processor-intensive Levenshtein functions, it checks to make sure there is at least **some** preliminary evidence that the two records refer to the same entity. The most common way of doing so is to check whether the observations match on **any** of a certain set of key variables, called "blocking variables" – these could be any important and distinctive variable, such as first name, last name, or date of birth.

Remember not to count missing variables as matches to other missing variables. The most intuitive way to code blocking is to add AND and OR conditions to the ON clause of the JOIN; the following code will compare only pairs of records that share at least one of three key variables: first name, last name, or date of birth.

```
from      mydata_clean as a INNER JOIN mydata_clean as b
on        a.obs_num > b.obs_num
and       (     a.fname = b.fname and a.fname is not missing
                or a.lname = b.lname and a.lname is not missing
                or a.dob = b.dob and a.dob is not missing
          )
```

However, there is a problem with this code – SAS cannot use indexes for a JOIN that contains an OR. For this reason, we are forced to rely instead on the somewhat clumsy syntax of the PROC SQL set operator UNION (see, for example, "SQL SET OPERATORS: SO HANDY VENN YOU NEED THEM", by Howard Schreier.) To make a long story short, the easiest way to code the UNIONs is to wrap the logic of the SELECT statement in a macro, and then invoke the macro once for each blocking variable, combining the multiple SELECT statements with UNIONs. And thus, the final code for our PROC SQL step could look like this:

```
%macro blocking_var(myvar);
    select case   when a.fname = '' or b.fname = '' then 0
                  when a.fname = b.fname then a.fname_points
                  when find(a.nicknames,b.fname) or find(b.nicknames,a.fname) then
                              0.9*min(a.fname_points,b.fname_points)
                  when findw(a.fname,b.fname) or findw(b.fname,a.fname) then
                                    0.9*min(a.fname_points,b.fname_points)
                  when complev(a.fname,b.fname)/
                              max(length(a.fname),length(b.fname)) <=0.25 then
                                    0.8*min(a.fname_points,b.fname_points)
                  when complev(a.fname,b.fname)/
                        max(length(a.fname),length(b.fname))<=0.35 then
                              0.5*min(a.fname_points,b.fname_points)
                  else -5 end as fname_score,

           case   when a.lname = '' or b.lname = '' then 0
                  when a.lname = b.lname then a.lname_points
                  when findw(a.lname,b.lname) or findw(b.lname,a.lname) then
                                    0.9*min(a.lname_points,b.lname_points)
                  when complev(a.lname,b.lname)/
                              max(length(a.lname),length(b.lname)) <=0.25 then
                                    0.8*min(a.lname_points,b.lname_points)
                  when complev(a.lname,b.lname)/
                        max(length(a.lname),length(b.lname))<=0.35 then
                              0.5*min(a.lname_points,b.lname_points)
                  else -5 end as lname_score,

           case   when a.dob = . or b.dob = . then 0
                  when a.dob = b.dob then 12
                  when year(a.dob) = year(b.dob) and month(a.dob) = day(b.dob)
                        and day(a.dob) = month(b.dob) then 11
                  when year(a.dob) = year(b.dob) and month(a.dob) = month(b.dob)
                              then 9
                  when year(a.dob) = year(b.dob) and day(a.dob) = day(b.dob) then 9
                  when month(a.dob) = month(b.dob) and day(a.dob) = day(b.dob)
                        then 9
                  else -6 end as dob_score,
```

```
            case   when a.race= 'U' or b.race = 'U' then 0
                   when a.race = b.race then 2
                   when a.race in ('M','O') or b.race in ('M','O') then 0
                   else -3 end as race_score

            calculated fname_score + calculated lname_score +
                   calculated dob_score + calculated race_score as score

    from   mydata_freqs as a INNER JOIN mydata_freqs as b

    on     a.obs_num > b.obs_num
    and    a.&myvar. = b.&myvar.

    where  a.&myvar. is not missing
    and    calculated score >= 25
%mend;
```

The above code creates a data set called SIMILARITY_SCORES that has scores for every pair of records whose core is 25 or greater (the “early threshold”.) Now we can apply our final threshold:

```
data threshold;
    set similarity_scores;
    where score >= 31;
run;
```

...and then feed the data set into the “transitive clustering” hash algorithm macro from the “transitive record linkage” paper:

```
%macro translink(in=,out=,key=key,keyfmt=);
    data &out. (keep = new_&key. old_&key.);
           length &key.1 &key.2 new_&key. old_&key. this_key dummy_key &keyfmt.;
           if _n_ = 1 then do;
                  declare hash h1(dataset: "&in.", multidata: 'yes');
                  h1.definekey ("&key.1");
                  h1.definedata(all: 'yes');
                  h1.definedone();
                  declare hash h2(dataset: "&in.", multidata: 'yes');
                  h2.definekey ("&key.2");
                  h2.definedata(all: 'yes');
                  h2.definedone();
                  declare hash buffer();
                  buffer.definekey('this_key');
                  buffer.definedata('this_key');
                  buffer.definedone();
                  declare hiter loop('buffer');
                  declare hash checklist();
                  checklist.definekey('dummy_key');
                  checklist.definedata('dummy_key');
                  checklist.definedone();
                  call missing(dummy_key);
           end;
```

```
            set &in.;
            if checklist.check(key: &key.1)=0 or checklist.check(key: &key.2)=0
                  then return;
            buffer.ref(key: &key.1, data: &key.1);
            buffer.ref(key: &key.2, data: &key.2);
            checklist.ref(key: &key.1, data: &key.1);
            checklist.ref(key: &key.2, data: &key.2);

            do until(buffer.num_items - starting_size = 0);
                  starting_size = buffer.num_items;
                  do while(loop.next()=0);
                        if(h1.find(key: this_key)=0) then do
                              until(h1.find_next(key: this_key)^=0);
                        buffer.ref(key: &key.1, data: &key.1);
                        buffer.ref(key: &key.2, data: &key.2);
                              checklist.ref(key: &key.1, data: &key.1);
                              checklist.ref(key: &key.2, data: &key.2);
                        end;
                        if(h2.find(key: this_key)=0) then do
                              until(h2.find_next(key: this_key)^=0);
                              buffer.ref(key: &key.1, data: &key.1);
                              buffer.ref(key: &key.2, data: &key.2);
                              checklist.ref(key: &key.1, data: &key.1);
                              checklist.ref(key: &key.2, data: &key.2);
                        end;
                  end;
            end;
            loop.first();
            new_&key. = this_key;
            do until(loop.next()^=0);
                  old_&key. = this_key;
                  output;
            end;
            buffer.clear();
      run;
%mend;
%translink(in=work.threshold, out=recoded, key=obs_num, keyfmt=$13.);
```

This macro creates a “lookup table” that shows which new, recoded identifier should be assigned to each record in the original data set – that is, which cluster or “real entity” (real person, in this case) each record should be assigned to. Finally, we can use the lookup table to merge the newly recoded identifiers back onto the original data:

```
proc sql;
    create table encode_linkage as
    select coalesce(recoded.new_id,original.zid) as dedup_id, original.*
    from   mydata_inexact as original LEFT JOIN recoded
    on     original.obs_num = recoded.new_obs_num
    ;
quit;
```

Now we have exactly what we originally wanted: A data set with a common identifier field that uniquely identifies patients – or at least, the closest thing that statistical guesswork can provide.

CONCLUSION

The record linkage method describe in this paper is complex, to say the least – the techniques used include PROC SQL, macros, indexes, hash objects, and obscure DATA step functions. However, this method produces high quality matches – it uses methods similar to those used in free and commercial record linkage software, and when customized to fit the needs of particular projects and data sets, it should produce better results than any off-the-shelf software package. For a sufficiently large, important record linkage task, the extra effort may well pay off.

REFERENCES

- Creunen, N.; Theuwissen, H. *Paper 011-27. Table Look-up: Techniques Beyond the Obvious.* www2.sas.com/proceedings/sugi27/p011-27.pdf, accessed 8/7/2011.
- Fellegi, I.; Sunter, A. (1969). "A Theory for Record Linkage." *Journal of the American Statistical Association 64 (328): pp. 1183-1210.*
- Raithel, M. *Paper 247-30. "The Basics of Using SAS Indexes." www2.**sas**.com/proceedings/sugi30/247-30.pdf, accessed 8/7/2011*
- Shreier, H. *Paper 242-21. "SQL Set Operators: So Handy Venn You Need Them." www2.sas.com/proceedings/sugi31/242-31.pdf*, accessed 8/7/11.
- Winkler, W.E. (1990). *"String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage." Proceedings of the Section on Survey Research Methods (American Statistical Association): 354-359.*
- Wright, G.; Hulett, D. "*Transitive Record Linkage in SAS using Hash Objects." www.wuss.org/proceedings10/databases/2997_2_DDI-Wright.PDF, accessed 8/7/2011.*

ACKNOWLEDGMENTS

Special thanks to John Kraemer for his suggestions regarding hash techniques, and to Denis Hulett for his guidance and research into the Fellegi-Sunter Model.

RECOMMENDED READING

Herzog, T. N.; Sheuren, F. J.; Winkler, W.E. (2007). *Data Quality and Record Linkage Techniques.* New York: Springer Science+Business Media, LCC.

CONTACT INFORMATION

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

Glenn Wright
Kaiser Permanente
1800 Harrison, 24th Floor
Oakland, CA 94612
510-625-4532
glenn.t.wright@kp.org