'Tuning' an ASM Metric: A Case Study in Metric ASM Optimization

† Hal Berghel, † David Roach
‡ George Balogh, ‡ Carroll Hyatt

† University of Arkansas
‡ Acxiom Corporation

Abstract

We present an approximate string matching case study. An optimized version of the edit distance algorithm is described which has proven more accurate for a particular commercial application than the existing (benchmark) algorithm. The evolution and nature of the optimizations are detailed and test results are presented.

Introduction

String matching refers to the activity of associating strings of symbols with one another. *Exact String Matching* (ESM) refers to the set of procedures which associate identical string tokens with one another [1]. *Approximate String Matching* (ASM) refers to a set of procedures which associate non-identical strings with one another on the basis of some criterion (or set of criteria) of similarity [2]. *Classical Approximate String Matching* is the traditional ASM approach which requires that each symbol be non-decomposable. To illustrate, classical ASM on strings of text would not analyze the properties of the symbols (e.g., symbol set from which they are drawn, typeface and size, etc.). Historically, most Classical ASM has been *probabilistic* in the sense that the result of string comparison was an estimate of likelihood that the two strings were the same. In contrast to probabilistic methods, *axiomatic* methods [3][4] directly encode the relevant definitions of similarity and thereby have uniform match probabilities of 1. A third method, comprised of edit distance algorithms, determine the minimal number of editing steps which transpose one string into another.

Any similarity relation which may exist between strings may be conveniently described in terms of Faulk categories [5]: *positional similarity*, or the degree to which matching symbols are in the same position in their respective strings [6]; *ordinal similarity*, or the degree to which the matching symbols are in the same order [7]; and *material similarity*, or the degree to which two strings are made up of the same symbol tokens [8-13]. Algorithms may, and frequently do, rely on one or more of these categories. Those which use only one category of measurement are called *single-relation* measures, while others are *multiple-relation* measures.

The advantage of single-relation similarity measures for ASM applications is that the algorithms are usually straightforward and easy to implement. However, this advantage comes at a considerable expense: single-relation measures only look at one aspect of similarity and, as a consequence, are limited to relatively unsophisticated applications. Because of this, multi-relation measures have dominated ASM research since the mid-1970's.

Two major categories of multi-relation techniques exist: *metric* and *non-metric*. Metric techniques are real-valued difference functions which satisfy, among other things, the property of triangular inequality

(i.e., if string 1 and string 2 are different as are strings 2 and 3, then the difference between strings 1 and 3 is less than or equal to the sum of the other differences). One popular metric ASM technique is the Levenshtein method [14], which has been applied extensively (e.g., [15][16]; see also [2] and [17]). Non-metric techniques are also quite popular. One leading non-metric technique is *n-gram analysis* [4][7][18][19[20].

As we mentioned in the first paragraph, classical ASM has been based on approximations or estimates of similarity. In the mid-1980's, another automated approach was developed by Berghel, et al [3][4]. This method directly encodes the set-theoretical definition of similarity in use into the computer program within a logic programming paradigm. The advantage of this approach is that it is maximally effective as measured by standard information-theoretic measures [21][22]. That is, it does not err in adjudging similarity, however it is known that this accuracy may come at the expense of efficiency when implemented within a logic programming paradigm.

In this paper, we report on a case study in ASM optimization based upon the Levenshtein metric.

The Levenshtein Metric

An ASM similarity measure is a real-valued difference function, d, over character strings, which satisfies the following conditions

1) $d(s_1,s_2) \geq 0$
2) $d(s_1,s_2) = 0 <-> s_1=s_2$
3) $d(s_1,s_2) = d(s_2,s_1)$
4) $d(s_1,s_2) + d(s_2,s_3) \geq d(s_1,s_3)$

for arbitrary character strings s_1,s_2,s_3.

A popular similarity measure which is both multi-relation and metric is the so-called 'Levenshtein Metric', named after the pioneer in coding theory who first suggested its use [14]. This measure has been applied to spelling correction by [15][16]. We now describe this technique based upon the presentation in Hall and Dowling [2].

Let d(i,j) be an multi-relation measure of similarity with respect to strings $s_1=c_1c_2...c_i$ and $s_2=c'_1c_2...c'_j$. We define it recursively as follows:

$$d(0,0) = 0$$
$$d(i,j) = \min[d(i,j-1)+1,\ d(i-1,j)+1,\ d(i-1,j-1)+v(c_i,c'_j),\ d(i-2,j-2)+v((c_{i-1},c'_j)+v(c_i,c'_{j-1})+1]$$

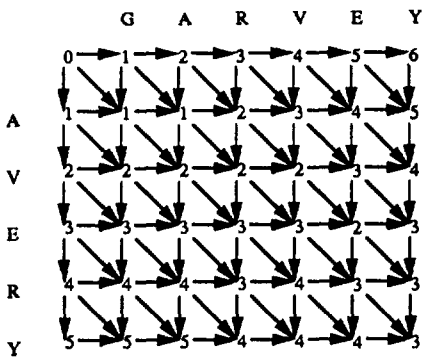
where,

$$v(c_i,c'_j)=0 <-> c_i = c'_j, \text{ and}$$
$$v(c_i,c'_j)=1 <-> c_i \neq c'_j.$$

In this case, we extract a measure of similarity between two strings by creating a directed graph for all nodes (i,j), with

horizontal and vertical edges weighted 1 (the penalty for a mismatch) and the weights of diagonal edges determined by v (see Figure 1).

Figure 1: Matrix Paths



Intuitively, since penalties are cumulative, the more dissimilar strings will have the longest paths. Since the difference measure, d, satisfies conditions 1)-4), above, it qualifies as a legitimate metric.

Note here that the terms of the minimization function approximate the Damerau conditions as set forth in [23] (i.e., string difference in terms of one missing character, one additional character, a substitution of a character and transposition of two characters). To illustrate, assume that the shortest path weight is 1 for two strings whose lengths differ by 1. Further, assume that this path weight resulted from the minimizing term d(i-1,j)+1. We would take this to mean that the shorter string is related to the longer by accidental omission of a character.

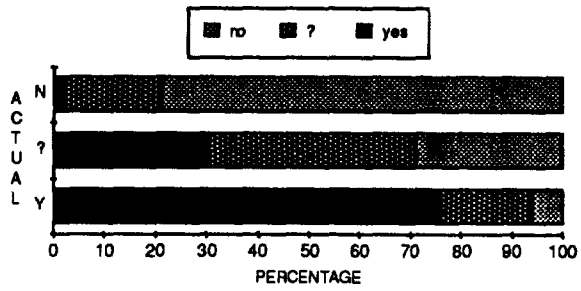
Objective

The objective of this research was to develop methodological and procedural guidelines which could regulate the 'tuning' of metric ASM algorithms so that they could equal or exceed the effectiveness of other generic ASM algorithms with respect to a test dataset.

Specific details of the test dataset and benchmark algorithm are irrelevant to this report since our objective is the development of guidelines which would apply of ASM algorithms as such and in general. The dataset of interest consisted of 2,500 pairs of last names. The names in the name-pair were slightly different from one another. For example, one might be 'Vareha' and the other 'Vahera'. For each pair, a (manual) determination was made whether the strings were sufficiently similar to justify the conclusion that one might likely be a corruption of the other. The determination was 3-valued: 'definitely a match', '?' and 'definitely not a match'. Then the performance of the benchmark algorithm was compared with these evaluations. The algorithm's output was also 3-valued: 'yes', '?' and 'no'. Obviously, for each name-pair nine situations could result. There was no variable weighting assigned to a mismatch, assigning a 'yes' with a 'no' was penalized the same as assigning a '?' to a 'yes'.

Figure 2 assesses the accuracy of the benchmark algorithm in terms of these nine possibilities as measured by the number of mismatches. The legend identifies the patterns that correspond to the conclusions of the algorithm. For instance, the blackened area represents the number of instances where the algorithm said there was a match. The other patterns correspond to the number of instances in which the algorithm said there was a mismatch or a questionable case. The x-axis represents percentages of the algorithm's conclusions that were Y's, N's, and ?'s. These are plotted against the three categories of manual responses on the y-axis. For example, the upper bar

Figure 2: Effectiveness of benchmark algorithm by category



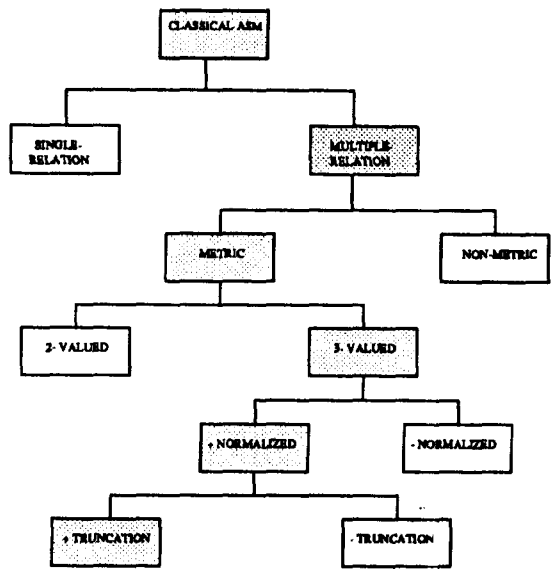
indicates that the algorithm: (1) says N when it should say N ≈ 79% of the time, (2) says ? when it should say N ≈ 19% of the time, and (3) says Y when it should say N ≈ 2% of the time. The other two bars compare the algorithm's output to the expected outputs of ? and Y, respectively.

We describe the general procedure which enabled us to 'tune' the Levenshtein metric so that the effectiveness exceeded the benchmark algorithm. We refer to the resulting algorithm as TM ('tuned metric').

Methodology

Figure 3 identifies the range of options considered in the search for a more effective ASM technique.

Figure 3: Major Decision Points



Only Classical ASM techniques were considered since axiomatic procedures have so far only been implemented as declarative (vs. procedural) programs, and thus have such different performance characteristics that comparison would be problematic.

Algorithms representing all three single-measure techniques (see introduction) were considered, although they are not shown in the figure. While single-relation measures may offer advantages, they must be used with caution. On the one extreme, material similarity tends to be too 'loose' (anagrams

are materially similar, for example), while on the other, positional tends to be too restrictive ('abcdef' and 'bcdef' are totally dissimilar, positionally). The compromise, ordinal similarity, has similar problems ('wilson' and 'wsn' are similar). For these reasons, single-relation measures are usually most appropriate in special-purpose applications for which the nature of the string differences is well understood. The application under discussion was not such an application.

The properties of a metric algorithm, particularly triangular inequality, were needed for further processing. As a component of a VLDB system, the ASM routines which we developed had to provide output scores which were amenable to further partitioning of the data. Metric ASM algorithms make it possible to partition the data without *a priori* knowledge about orthographical make-up (vs., e.g., n-gram analysis which derives its utility from the non-uniform distribution of n-grams in the data which must be known in advance). In the present context, all ASM activity must be done 'on the fly'.

Thus, we compared the Levenshtein algorithm (see above) to the benchmark algorithm. Implemented according to standard practice [2][15][16], the Levenshtein algorithm compared strings with respect to omission, insertion, substitution, and transposition of characters. Both algorithms were initially set up to return a binary match value (Yes or No). In this mode, it is trivial to optimize algorithms in terms of critical threshold: the cutoff point is adjusted to maximize agreement. On this basis, the Levenshtein algorithm had a slight edge in terms of accuracy (69% correct to 63%).

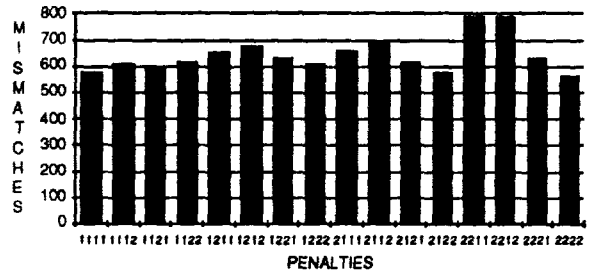
However, manual comparison of the results with the test dataset indicated a major problem with the two-valued approach: the accuracy could not appreciably increase unless both algorithms could better distinguish between degrees of similarity. On the two-valued account, 'a miss was as good as a mile'. However, the test dataset revealed many cases of very similar strings which differed in subtle ways which were not detectable by the algorithms. In order to catch such pairs it was decided to concentrate on only three-valued variations of the algorithms (Yes, No and Maybe). The advantage was that as long as a high percentage of the problematic string pairs were flagged as possible matches, further processing (depending upon the application and context) could be used to resolve the matter. The alternative would be to continuously tune the algorithms to accommodate each new subtlety, which would both be impractical and reduce the efficiency of the programs.

In their ternary versions, the accuracy advantage shifted (69% for the benchmark algorithm and 65% for Levenshtein), but did not increase overall. Once again, visual inspection was called for. Examination revealed that the nature of the remaining failures were such that a significant fraction could be overcome by means of two simple adjustments. It was found that by normalizing the match values with respect to the lengths of the strings (so that a one-character difference between strings of length two would be penalized more than a one-character difference between strings of length ten) and by handling truncations (e.g., WILL vs. WILLIS) independently of the ASM algorithms through pre-processing, the overall accuracy increased by 10% for both systems. It was now simply a matter of refinement.

The benchmark algorithm is also an multi-relation ASM technique which is three-valued and employs normalization and truncation according to the same algorithm. Although it was not a concern, we found that the general orientation of the benchmark algorithm was prudent: we were unable to find alternative orientations (i.e., with respect to Figure 3) which were more effective.

Having determined that within the range of our comparisons the Levenshtein algorithm was the most effective alternative to the benchmark algorithm, the next objective became optimization. Levenshtein metrics have two parameters which can be tuned. First, variations in the penalty set are possible. These values weigh the character differences according to the nature of the difference. In its standard form, the Levenshtein algorithm (cf. [2]) recognizes missing, extra, substituted and transposed characters (classical typing errors) and penalizes them equally with unity (i.e., the standard penalty set is $<1,1,1,1>$, corresponding to missing, extra, substituted and transposed characters, respectively). Figure 4 illustrates the effect that the penalty set may have on the accuracy of the algorithm, with the other parameter held constant.

Figure 4: Mismatch rate for varying penalties (optimized)

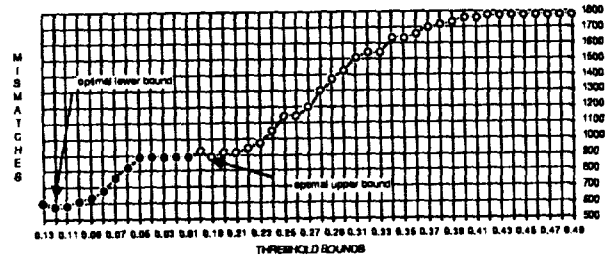


The second parameter which may be adjusted involves the thresholds at which distinctions between strings are made. In the three-valued framework, the distinction must be made between matches and questionable matches, and questionable matches and definite non-matches. Thus, there are two thresholds to contend with.

Obviously, with two equally important variables to consider, one will have to be held constant while the algorithm is optimized with respect to the other. The question of the order in which the parameters are optimized really comes to a question of an estimate of the coarseness or fineness of the parameter's changes. In the case under review, the coarser measure is the penalty set - we could effectively deactivate one quarter of the algorithm by substituting a penalty of 0 for 1 for any of the error types. As Figure 4 shows, the $<1,1,1,1>$ set of penalties for <omission, insertion, substitution, transposition>, respectively, performed comparatively well with respect to other combinations of values. $<2,2,2,2>$ yielded a few less mismatches but not enough to justify its use over the traditional penalty assignments.

Holding the penalties constant at $<1,1,1,1>$, an optimization algorithm was employed to adjust first the lower and then the upper thresholds until mismatches began to increase. Figure 5 shows that the optimal lower and upper bounds for our test dataset are 0.12 and 0.19 - the bounds at which the fewest mismatches occur.

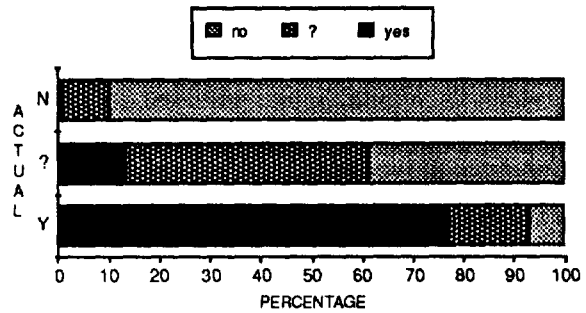
Figure 5: Effects of threshold changes on accuracy



Results

Figure 6 plots the accuracy of TM with respect to various combinations of matches and mismatches that occur as was done with the benchmark algorithm in Figure 2.

Figure 6: Effectiveness of TM algorithm by category



For example, the upper bar indicates that the algorithm: (1) says N when it should say N ≈ 90% of the time, (2) says ? when it should say N ≈ 9% of the time, and (3) says Y when it should say N ≈ 1% of the time. The other bars compare the cases in which the algorithm outputs ? or Y to what it should output. The actual results are tabulated below. The first Y, N, or ? represents the algorithm's output, and the second represents the desired output, i.e., the actual match value which was determined manually. 2,500 string pairs were tested. There are 1,257 non-matching pairs, 614 matching pairs, and 629 pairs that are border-line cases.

Benchmark Algorithm

Mismatches: 784

Match Type	Ratio	Percentage
Y-Y	458 / 614	74.59
Y-N	17 / 1257	1.35
Y-?	192 / 629	30.52
N-Y	38 / 614	6.19
N-N	998 / 1257	79.40
N-?	177 / 629	28.14
?-Y	118 / 614	19.22
?-N	242 / 1257	19.25
?-?	260 / 629	41.34

TM Algorithm

Mismatches: 579

Match Type	Ratio	Percentage
Y-Y	477 / 614	77.69
Y-N	3 / 1257	0.24
Y-?	79 / 629	12.56
N-Y	40 / 614	6.51
N-N	1139 / 1257	90.61
N-?	245 / 629	38.95
?-Y	97 / 614	15.80
?-N	115 / 1257	9.15
?-?	305 / 629	48.49

The difference in the number of mismatches (205) represents a 26% improvement in accuracy over the benchmark algorithm.

In addition to mismatch ratios, TM was evaluated using the standard information theoretic measures of precision, recall, fallout, and generality [21]. These measures are calculated for each of the three match categories Y, ?, and N. A match occurs when the algorithm's match value equals the actual match value. A mismatch and missed match occur when the algorithm's match value does not equal the actual match value. For instance, if the algorithm says Y when it should say ?, a mismatch has occurred with respect to the Y category and a missed match has occurred with respect to the ? category. Ideally, an algorithm would have 100% precision and recall and zero fallout. Generality represents an average that needs only remain constant when two algorithms are compared. These measures can be expressed in terms of matches, mismatches, and missed matches as follows:

Precision = matches / (matches + mismatches)
Recall = matches / (matches + missed matches)
Fallout = mismatches / (mismatches + total number that are neither matches, mismatches, nor missed matches)
Generality = (matches + missed matches) / (matches + missed matches + mismatches + the total number that are neither matches, mismatches, nor missed matches), i.e., the total number of string pairs compared

If we let [A-B] represent the ratio of the number of times the algorithm output A to the number of times it should have output B, then we can define the measures for each category (Y, N, and ?) as follows:

precision(YES) = [Y-Y] / ([Y-Y] + [Y-N] + [Y-?])
precision(?) = [?-?] / ([?-?] + [?-Y] + [?-N])
precision(NO) = [N-N] / ([N-N] + [N-Y] + [N-?])

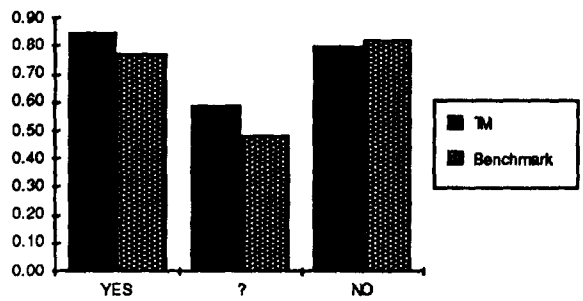
recall(YES) = [Y-Y] / ([Y-Y] + [?-Y] + [N-Y])
recall(?) = [?-?] / ([?-?] + [Y-?] + [N-?])
recall(NO) = [N-N] / ([N-N] + [Y-N] + [?-N])

fallout(YES) = ([Y-?] + [Y-N]) /
{([Y-?] + [Y-N]) + ([N-N] + [N-?] + [?-?] + [?-N])}
fallout(?) = ([?-Y] + [?-N]) /
{([?-Y] + [?-N]) + ([N-Y] + [N-N] + [Y-Y] + [Y-N])}
fallout(NO) = ([N-?] + [N-Y]) /
{([N-?] + [N-Y]) + ([Y-Y] + [Y-?] + [?-Y] + [?-?])}

generality(YES) = ([Y-Y] + {[?-Y] + [N-Y]}) /
([Y-Y] + {[?-Y] + [N-Y]}) +
([Y-?] + [Y-N]) +
([N-N] + [N-?] + [?-?] + [?-N])
generality(?) = ([?-?] + {[Y-?] + [N-?]}) /
([?-?] + {[Y-?] + [N-?]}) +
([?-Y] + [?-N]) +
([N-N] + [N-Y] + [Y-Y] + [Y-N])
generality(NO) = ([N-N] + {[?-N] + [Y-N]}) /
([N-N] + {[?-N] + [Y-N]}) +
([N-?] + [N-Y]) +
([Y-Y] + [Y-?] + [?-?] + [?-Y])

As indicated by Figures 7-9, precision increased for the Y and ? categories and decreased only slightly for the N category, recall increased for the Y and N categories and decreased slightly for the ? category, and fallout decreased for the Y and ? categories while increasing slightly for the N category.

Figure 7: Precision by category



The actual values are tabulated below:

Benchmark Algorithm

precision(Y): 0.77	precision(?): 0.48	precision(N):0.82
recall(Y): 0.73	recall(?): 0.52	recall(N): 0.80
fallout(Y): 0.07	fallout(?): 0.19	fallout(N): 0.18
generality(Y): 0.25	generality(?): 0.25	generality(N): 0.50

Figure 8: Recall by category

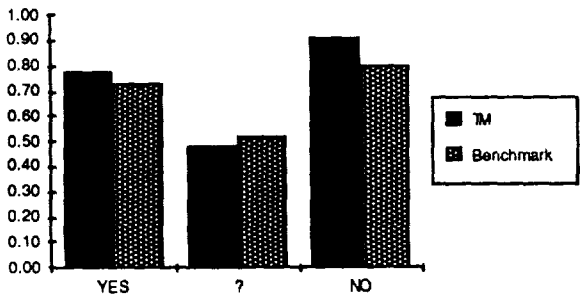
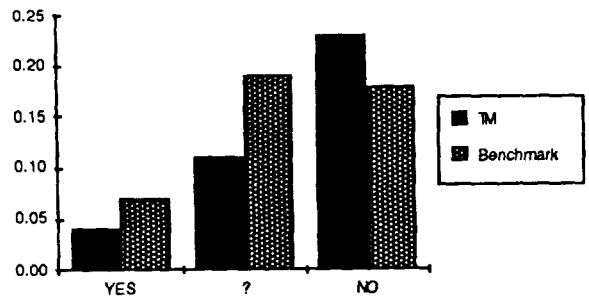


Figure 9: Fallout by category



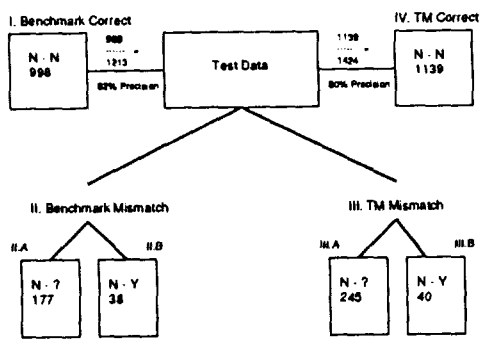
TM Algorithm

precision(Y): 0.85	precision(?): 0.59	precision(N):0.80
recall(Y): 0.78	recall(?): 0.48	recall(N): 0.91
fallout(Y): 0.04	fallout(?): 0.11	fallout(N): 0.23
generality(Y): 0.25	generality(?): 0.25	generality(N): 0.50

There was a net increase of 17% in precision and a net increase in recall of 12%. The fallout decreased by 6%.

The decrease in precision with respect to the N category deserves comment. Figure 10 depicts the breakdown of matches and mismatches within the category for both benchmark and tuned metric algorithms.

Figure 10. Matches and mismatches by algorithm (N category)



We note that while TM actually identified 141 more of the unmatched pairs than the benchmark, this advantage was offset by only half as many mismatches. This illustrates the sensitivity of the precision measure to mismatches.

In order to understand the underlying cause, it is helpful to determine the nature of the mismatches. TM mismatched 96 times in this category in situations where the benchmark algorithm made the correct assessment. The list in Figure 11 reveals typical differences.

Figure 11. Cases where TM mismatched with a value of 0.20 (N-Y & N-?) when the benchmark algorithm matched (Y-Y & ?-?)

N-Y	N-?
CORA CORREA	ALVARADO ALVAREZ
CORO CORRAO	ARIAS ARIZA
PAES PEASE	CAHILL HILL
PHILLIPS PHILP	CANALES CANALIZO
	CARREIRO CURRIER
	CHARETTE CHARTER
	CHRISTENSON CHRISTIAN
	COATES COTE
	FEELY FOLEY
	FERRARI FERREIRA
	FISCHER FISCHMAN
	GAGNE GAGON
	GENDREAU GENDRON
	GOULD GUILD
	HENDERSON HENDRICKSON
	JAENA JANES
	KEELY KILEY
	KLAMA KULMA
	LAFALAM LAFLAMME
	LANGILLE LANGLEY
	MARIO MAURO
	MATEO MATOS
	MCKENNA MCKINNEY
	QUEEN QUINE
	RAIMO RAMOS
	RASTANI RISTAINO
	REGAN RGEON
	SALWA SLIWA
	SANDREW SAUNDERS
	SILVERMAN SILVERSTEIN
	STEEN STONE
	SWEET SWETZ

In our opinion, further tuning was unwarranted for the results of TM more closely approximated our intuition than the test dataset. However, the tuning procedure would be the same as employed above.

Conclusion

The tunability of the Levenshtein metric makes it possible to tailor it to specific applications. Using a representative sample of the production data it can be adjusted to adapt to the types of corruptions that occur in that data. We have found that it significantly improves on an existing algorithm in a context where corrupted name pairs must be matched.

Acknowledgements

This research was supported in part by a grant from the ACXIOM corporation and grant # ASTA 91-A-02 from the Arkansas Science and Technology Authority.

References

[1] Knuth, D. (1973). Sorting and Searching. Reading: Addison-Wesley.

[2] Hall, P. & Dowling, G. (1980). Approximate String Matching. Computing Surveys, 12, 381-402.

[3] Berghel, H. (1987). A Logical Framework for the Correction of Spelling Errors in Electronic Documents. Info. Proc. and Mgmt., 23, 477-494.

[4] Berghel, H. & Andreu, C. (1988). TALISMAN: A Prototype Expert System for the Detection and Correction of Spelling Errors. Proc. 1988 ACM Symp. on Small Systems (pp. 107-113).

[5] Faulk, R. (1964). An Inductive Approach to Language Translation. Communications of the ACM, 7, 647-653.

[6] Glantz, H. (1957). On the Recognition of Information with a Digital Computer. Journal of the ACM, 4, 178-188.
[7] Angell, R., Freund, G., & Willett, P. (1983). Automatic Spelling Correction using a Trigram Similarity Measure. Information Processing and Management, 19, 255-261.
[8] Odell, M. & Russell, R. (1918, 1922). U.S Patents 1,261,167 (1918) and 1,435,663 (1922).
[9] Alberga, C. (1967). String Similarity and Misspellings. Communications of the ACM, 10, 302-313.
[10] Blair, C. (1960). A Program for Correcting Spelling Errors. Information and Control, 3, 60-67.
[11] Davidson, L. (1962). Retrieval of Misspelled Names in an Airlines Passenger Record System. Communications of the ACM, 5, 169-171.
[12] Muth, F. & Tharp, A. (1977). Correcting Human Error in Alphanumeric Terminal Input. Information Processing and Management, 13, 329-337.
[13] Pollock, J. & Zamora, A. (1984). Automatic Spelling Correction in Scientific and Scholarly Text. CACM, 27, 358-368.
[14] Levenshtein, V. (1966). Binary Codes Capable of Correcting Deletions, Insertions and Reversals. Sov. Phys. Dokl., 10, 707-710.
[15] Lowrance, R. & Wagner, R. (1975). An Extension of the String-to-String Correction Problem. Journal of the ACM, 22, 177-183.
[16] Wagner, R. & Fischer, M. (1974). The String-to-String Correction Problem. Journal of the ACM, 21, 168-178.
[17] Manber, U. (1989). Introduction to Algorithms: a Creative Approach. Reading: Addison-Wesley.
[18] Fu, K. (1976). Error-Correcting Parsing for Syntactic Pattern Recognition. In Klinger, et al (Eds.), Data Structures, Computer Graphics and Pattern Recognition. New York: Academic Press.
[19] Ullman, J. (1977). A Binary N-Gram Technique for Automatic Correction of Substitution, Deletion, Insertion and Reversal Errors in Words. The Computer Journal, 20, 141-147.
[20] Zamora, E., Pollock, J., & Zamora, A. (1981). The Use of Trigram Analysis for Spelling Error Detection. Information Processing and Management, 17, 305-316.
[21] Salton, G. & McGill, M. (1983). Introduction to Modern Information Retrieval. New York: McGraw Hill.
[22] Salton, G. (1990). Automatic Text Processing. Cambridge, MA: MIT Press.
[23] Damerau, F. J. (1964). A Technique for Computer Detection and Correction of Spelling Errors. Communications of the ACM, 7, 171-176.