SIMILARITY METRIC AND NONMETRIC SCORES: A COMPARISON

SHANNON GALLAGHER AND AUSTIN RUSSELL

ABSTRACT. After presenting some theoretical background about metric and nonmetric similarity scores, we compare the Jaro-Winkler metric, Levenshtein Edit Distance Metric, the LCS metric, and the Dice Coefficient nonmetric on two different data sets. The Dice Coefficient outperforms the metrics with respect to logistic regression and classification trees but lags behind the others with respect to Fellegi-Sunter Expected Matching in one of the data sets.

INTRODUCTION

In this project, we aim to present some theory on metric spaces with respect to similarity scores for record linkage. We provide a concrete example of record linkage data using four similarity scores: the Jaro-Winkler distance score, the Levenshtein distance, the Longest Common Subsequence (LCS), and one nonmetric, the Dice Coefficient. We examine the difference, if any, amongst the metric similarity scores, and the non-metric similarity score with regards to our concrete example, computational runtimes, and type of data to be compared.

Our project is motivated mathematically by metric spaces, which are briefly described in Section 2. In mathematics, metric spaces have a nice set of properties, and we want to see if these nice properties have any carry over to record linkage similarity scores. Our aim is to find a reason, if one exists, to choose metric scores over nonmetric scores or the other way around. We wish to optimize which score should be used for differing types of data sets.

We present the results of our experiments, which are results from the two data sets of different record pairs. We examine logistic regression models for each of the similarity scores as well as trees and the Fellegi-Sunter Expected Matching method. Along with the results, we provide conclusions on which score we would choose for these particular data sets and provide some explanations why we select this score.

1. SIMILARITY SCORES

In the context of record linkage, similarity scores give us a context to measure how different records are from one another. According to Peter Christen in *Data Matching*, "It is vital for data matching to employ comparison functions that return some indication of how similar two attributes are." Such comparison functions need to be tailored to the type of data that are being compared [2]. For instance, to borrow an example from *Data Matching* [2], intuitively, the names Shackelford and Shackleford should be close, and both of those should be pretty far away from the name Haverforth.

This leads to a situation in which there is currently no all encompassing metric that works for all types of data. Some metrics work better for numbers, some for

single words, and some for strings, all depending on what the data set is. Therefore, it is important to take into consideration all the properties of a similarity score as well as the data set one is looking to analyze. Notably, scores provide different computational complexities which we describe later in this paper. We briefly analyze the theory behind metrics and nonmetrics.

2. Metrics vs. Nonmetrics

A metric space $\mathcal{M} = (X, d)$ is a set X along with a function $d : X \to \mathbb{R}$ such that $\mathcal{M}$ satisfies the following properties:

- $d(x, y) \geq 0 \ \forall x, y \in X$ (non-negative)
- $d(x, y) = d(y, x) \ \forall x, y \in X$ (symmetric)
- $d(x, y) = 0$ if and only if $x = y$
- $\forall x, y, z \in X, d(x, z) \leq d(x, y) + d(y, z)$ (triangle inequality)

Metrics are useful in that they allow consistent comparisons among all elements, or points, in a set, much like the real life concept of distance between two points. We aim to analyze whether this perceived usefulness carries over to record linkage and similarity scores.

Pavel Zezula, et. al in their book, *Similarity Search: The Metric Space Approach* describe that using distance functions can be very versatile for problems that are modeled within a metric space. They write that the use of metrics is "a kind of *sorting*, *ordering*, or *ranking* of objects, with respect to the query object, where the ranking criterion is the distance measure" [8].

On the other hand, nonmetric similarity scores are those functions $n : X \to \mathbb{R}$ that do not satisfy the properties of a metric. Metrics that do not satisfy the non-negative property are called *pseudo-metrics*. According to Zezula, pseudo-metrics can be analyzed in the same way as metrics because these functions can be transformed into metrics by considering all the pairs of objects with distance zero to be a single entity [8]. Perhaps a more interesting nonmetric is one which is not symmetric. In the context of our previous example, we let n be our assymmetric similarity score. Then possibly n(Shackleford, Shackelford) = 0.8 and n(Shackleford, Shackelford) = 0.6. From an intuitive perspective this seems very strange. However, there are scores which have applications in fields such as chemoinformatics where the assymetric Tversky index function is utilized [6]. We do not examine an assymetric similarity score in this paper.

Perhaps the most common nonmetric similarity score is one that violates the triangle inequality. Physically speaking, a metric that does not satisfy the triangle inequality indicates that the shortest distance between two points is not a straight line. The nonmetric we examine in this paper, the Dice Coefficient, is one such example of a nonmetric that violates the triangle inequality property.

3. Specific Similarity Scores and Methods

In this section, we examine at length the theory behind the four similarity scores we use to analyze our data. The scores are the Jaro-Winkler metric, the Levenshtein Edit Distance metric, the LCS metric, and the Dice Coefficient.

3.1. Jaro-Winkler.

Developed because of the need to quickly analyze large amounts of data in the US Census, the Jaro-Winkler similarity score was originally created

by Matthew Jaro in 1972 and later updated by Wiliam Winkler. According to Winkler, "This metric accounts for the lengths of two strings and partially accounts for the types of errors...human beings typically make when constructing alphanumeric strings" [5].

The Jaro Winkler function is defined in Equation 3.2 and employs the use of weights. The weighting scheme allows some flexibility within this score for transpositions of characters, and the ratio of common characters to length of word can be assigned different values.

Formally, the orignal Jaro distance $\Phi_J : S \to \mathbb{R}$ is as follows from [5]:

$$\Phi_J(s_1, s_2) = W_1 \cdot \frac{c}{L_1} + W_2 \cdot \frac{c}{L_2} + W_t \cdot \frac{(c-\tau)}{c} \tag{3.1}$$

where s_1 and s_2 are strings of characters; W_i, $i = 1, 2$ is the weight assigned to the i^{th} string; W_t is the weight assigned to the transpositions; c is the number of characters that the two strings have in common; L_i, $i = 1, 2$ is the length of the i^{th} string; and τ is the number of common characters that are transposed. Additionally, $W_1 + W_2 + W_t = 1$ and if $c = 0$ then $\Phi_J(s_1, s_2) = 0$. The metric has continually been tweaked to allow more weighting for agreement in initial characters in strings . According to Winkler, this adjustment works better for high quality data than low quality data [5]. Computational runtime for the Jaro-Winkler metric is $\mathcal{O}(|s_1| + |s_2|)$ where $|s_i|$ is the length of the i^{th} string [2]. As a result, the Jaro-Winkler metric works better for single words than it does for multiple word strings. The modified Jaro-Winkler function is given below:

$$\Phi_{JW}(s_1, s_2) = \Phi_J(s_1, s_2) + (lp(1 - \Phi_J(s_1, s_2))) \tag{3.2}$$

Above, l denotes the length of the common prefix, and p is a scaling factor which determines how much weight should be given to shared characters at the beginning of words. We utilize the *jarowinkler* function supplied in the `RecordLinkage` package.

3.2. **Levenshtein Edit Distance.** Another metric that is often used for single word type data is the Levenshtein Edit Distance. This distance counts the number of edits it takes to "move" from string $x = x_1x_2...x_n$ to another string $y = y_1y_2...y_m$ where x_i and y_i are characters. The operations allowed according to Zuzela are the following:

- insertion: $insert(i, c, x) = x_1 \dots x_{i-1} c x_i \dots x_n$
- deletion: $delete(i, x) = x_1 \dots x_{i-1} x_{i+1} \dots x_n$
- subsitution: $sub(i, c, x) = x_1 \dots x_{i-1} c x_{i+1} \dots x_n$

The operations are given different weights and then combined to create a metric [8].

The formula for the Levenshtein Edit distance is the following as described in [2]. The variables s_1 and s_2 stand for different strings 1 and 2, $s_i[j]$ refers to the j^{th} character in the string i, and d counts the number of edits or operations.

- If $s_1[i] = s_2[j]$, then

$$d[i, j] = d[i - 1, j - 1]$$

- If $s_1[i] \neq s_2[j]$, then

$$d[i,j] = \begin{cases} d[i-1,j]+1 & \text{a deletion} \\ d[i,j-1]+1 & \text{an insertion, or} \\ d[i-1,j-1]+1 & \text{a subsitution} \end{cases}$$

- Finally, $dist_{lev}(s_1, s_2) = d[|s_1|, |s_2|]$

There are some instances where Levenshtein Edit Distance can be modified to allow for transpositions. Levenshtein Edit Distance can work for the same type of data that the Jaro-Winkler metric is often used on, single words. The Levenshtein Edit Distance is not restricted to words however and can work for strings of any type. Levenshtein Edit Distance has computational runtime on the order of $\mathcal{O}(|s_1| \times |s_2|)$ because one string needs to be turned into the other [2]. As a result, the Levenshtein Edit Distance can take significantly longer to run than the Jaro-Winkler metric for large data sets. We used the *stringdist* function contained in the `stringdist` package, supplying the option *method* = "*lv*".

3.3. **Longest Common Subsequence.** This is the third metric we analyze in this paper. The LCS by itself does not satisfy the triangle inequality. The LCS, s_{LCS}, as its name indicates, measures the longest common subsequence of two strings. According to Skopal and and Bustos in [7], that "even if modified to dissimilarity (e.g., $\delta(x,y) = s_{max} - s_{LCS}(x,y)$ with s_{max} the maximum possible value returned by s_{LCS}), it still does not satisfy the triangle inequality."

Still, the LCS can be turned into a metric through the following function. According to Bakkelund in [1], the function in Equation 3.3 does satisfy the properties of a metric.

$$lcs.score(s_1, s_2) = 1 - \frac{|s_{lcs}(s_1, s_2)|}{\max\{|s_1|, |s_2|\}} \tag{3.3}$$

One advantage of this metric is that it works well on both strings and vectors of numbers. It is versatile in its application. One possible disadvantage to this score is that does not take into account transpositions and can indicate that probable matches are nonmatches. Referring back to the example of Shackleford and Shackelford, the longest common subsequence length is 4 out of the 11 total letters. This may not be recorded as a match although we think the two may be. Since the two strings are being compared to one another, the run time is $\mathcal{O}(|s_1| \times |s_2|)$. We again used the *stringdist* function from the `stringdist` package, supplying the option *method* = "*lcs*" and modified it to return the metric specified in Equation 3.3.

3.4. **Dice Coefficient.** According to Leach, the Dice Coefficient is dependent on the number of bits two strings have in common. This can be extended to characters in strings. An interesting note about the Dice Coefficient, is that it is used in chemoinformatics, and "The presence of common molecular features will therefore tend to increase the values of [the Dice Coefficient]" [6].

The Dice Coefficient function is given by the following formula:

$$dice.co(s_1, s_2) = 1 - \frac{2|s_1 \cap s_2|}{|s_1| + |s_2|}.$$

On the left-hand-side, s_1 and s_2 refer to strings of characters. On the right-hand-side, $|s_1|$ and $|s_2|$ denote the number of bigrams of the strings s_1 and s_2 respectively,

while $|s_1 \cap s_2|$ denotes the number of bigrams the two strings have in common. Here, we create our own function to compute Dice Coefficient similarity scores. First, we write a helper function that generates a vector containing all of the bigrams for a string of given length. Then, the number of bigrams in common between the two strings are computed simply by using the built-in R function *intersection* on the two sets of bigrams, and then calling the R function *length* on the resultant set.

The Dice Coefficient is the only nonmetric we analyze in this paper. This similarity score does not satisfy the triangle inequality. An example of this are the strings $s_1 = ca$, $s_2 = at$, $s_3 = cat$ Then the reader can verify that $dice.co(s_1, s_2) = 1$, $dice.co(s_1, s_3) = \frac{1}{3}$, and $dice.co(s_1, s_2) = \frac{1}{3}$. Thus

$$dice.co(s_1, s_2) \not\leq dice.co(s_1, s_3) + dice.co(s_3, s_2).$$

The run time is $\mathcal{O}(|s_1| \times |s_2|)$. The call to *intersection* in our scoring function is what causes this time complexity, as our bigram helper-function only runs in linear time.

3.5. **Methods for Data Matching.** We test our similarity scores using three data matching techniques: logistic regression, classification trees, and Fellegi-Sunter Expected Matching.

For logistic regression, we provide a model for a data set for a computer algorithm, which in turn outputs the log odds of being a match for each record pair. From these log odds, we assign record pairs as matches and nonmatches and then analyze our error rate as we know the true matches and nonmatches.

Classification trees work by again providing a model for a computer algorithm which subsequently splits the data via different cutoff values for the similarity scores in different fields. The data is split into at most two pieces at each node in the tree. At the final nodes, the leaves of the tree, the probability of being a match or nonmatch is calculated. We use these probabilities to assign matches and non matches and then calculate the error rate to assess the performance.

The method for the Fellegi-Sunter Record Linkage Model can be found in *Data Quality* in [5]. In this method, we use a set of training data to calculate two sets or probabilities, $PM = P(F_1 \cap F_2 \cap \cdots \cap F_k|M)$ and $PU = P(F_1 \cap F_2 \cap \cdots \cap F_k|U)$ where F_i represents a field in the comparison array and can take a value of 0 or 1 with 0 being a nonmatch in that field and 1 being a match in that field. The 0 or 1 is determined by cutoff scores for the similarity score used. For example, a Jaro-Winkler similarity score of 0.90 or greater is deemed a match. The probabilities are then the likelihood of seeing a certain sequence of 0s and 1s in the set of fields, given a match (M) or a nonmatch U). A ratio $R = \frac{PM}{PU}$ is then calculated using the above probabilities for each sequence of 0s and 1s in the fields and then lower and upper bounds are evaluated. Depending where R falls with respect to the bounds, each record pair is determined to be a match, nonmatch, or sent to clerical review. After each record pair is assigned, we analyze the error rate. A variant of this method is to calculate PM and PU given the assumption of conditional independence. In this project, the training data is our actual data. In practice, the lower and upper bounds obtained using training data with unique identifiers can then be used to assign values of match, nonmatch, or clerical review to record pairs that come from similar data sets that do not contain unique identifiers.

4. Record Comparison with Different Metrics: Data Set 1

We use the four aforementioned similarity scores and a variety of tests to compare the scores amongst themselves. The methods we apply are logistic regression, classification trees, and Fellegi-Sunter Expected Matching with different assumptions of conditional independnece.

4.1. **Data Set 1.** Data Set 1 is formed as a subset of the table `RLData500` contained in the `R` package `RecordLinkage`. The data set consists of two lists with the fields first name, last name, birth year, birth month, and birthday as well as unique identifying number. There are 100 records in each list, and therefore 10000 total record pairs. For each record pair, a similarity score is given to each of the corresponding non-ID fields and stored in a scoring matrix. As a result, there are four separate 10000×6 matrices, one for each similarity score. There are 75 matches between the two lists, as identified by the unique ID fields.

4.2. **Logistic Regression Models.** For each of the four matrices, we create a logistic regression model based on the following formula

$$\text{match}_i = \text{first name}_i + \text{last name}_i + \text{birth year}_i + \text{birth month}_i + \text{birthday}_i \tag{4.1}$$

where i is the i^{th} record pair in one of the matrices, match takes a value of 0 (nonmatch) or 1 (match) and the variables on the right hand side of Equation 4.1 represent the similarity scores of the fields compared for the i^{th} record pair. A table of the logistic regression results for each of the similarity score can be found below in Figure 9.

As one can see from the results of Figure 9, none of the logistic regression models converged, meaning we cannot be sure of the accuracy of the resulting models. Still, the logistic regression models all worked very well on the data. All four models perfectly predict the true matches and nonmatches. This implies that we likely overfitted to the data and cannot draw many conclusions from these logistic regression models.

Score	Converged	First Name	Last Name	BY	BM	BD
Jaro-Winkler	No	159	104	87.1	28.3	42.6
Levenshtein	No	-7.58	-10.87	-6.53	-7.18	-18.7
LCS	No	175	250	213	254	561
Dice Coeff.	No	44.5	41.8	47.4	16.3	38.5

Figure 1. Table displaying the results of the logistic regression models for the four similarity scores.

4.3. **Classification Trees.** Besides using logistic regression, we also predict matches and nonmatches with our similarity scores using classification trees with the underlying idea being to split the data and collect matches and nonmatches at different leaves of the tree. We run the classification tree algorithm on the same algorithm found in Equation 4.1.

The trees we found are not perfect in predicting, which makes them more interesting to analyze than the logistic regression models. The different trees can be found below in Figure 2.

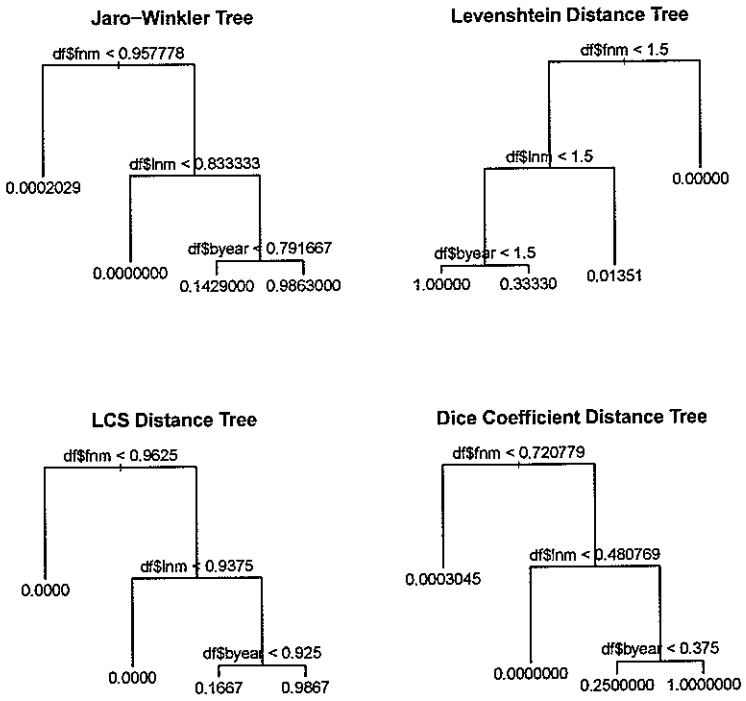

FIGURE 2. Classification trees of the four similarity scores. All split on first name, last name, and birth year.

Interestingly, the four similarity scores split on the same fields, in the same order: first name, last name, and then birth year. This implies that the four classification trees essentially "picked up" on the same model, just with different cut off values to accomodate the various similarity scores. This same tree structure implies that the similarity scores are finding essentially the same scores for each field, possibly up to a linear transformation. This means that the similarity scores are giving the same types of scores to the same types of words. One reason for this is that the data set may be too "nice" in that the data is too generic for the differences in the similarity scores to be observed.

Another observation is that in Figure 2, we see that the Levenshtein Distance Tree is flipped around its center, in comparison to the other trees. As we see from the tree, Levenshtein Distance splits on values greater than 1. Referring back to the formula for the Levenshtein distance, we see that there is no normalization and subtracting from 1 for the Levenshtein Distance tree, meaning that the scores can be greater than 1 and the closer they are to 0, the more similar the strings are. This is why the Levenshtein Distance tree is flipped, with respect to the other trees.

The ROC curves display $1 - sensitivity$ vs. $specificity$ or the ratio of number of false negatives out of the number of actual matches against the ratio of the number of true negatives out of the number of nonmatches using different cutoff values for

what constitutes a link for the similarity scores. A false negative is a pair that is assigned as a nonmatch but actually is a match, and a false positive is a pair that is assigned as a match and is actually not a match. Good similarity scores have close to area 1 under the curve, and bad similarity scores have area closer to $\frac{1}{2}$. Not shown on the graphs in Figure 3 are the points at (0,0) and (1,1) which belong to every ROC curve.

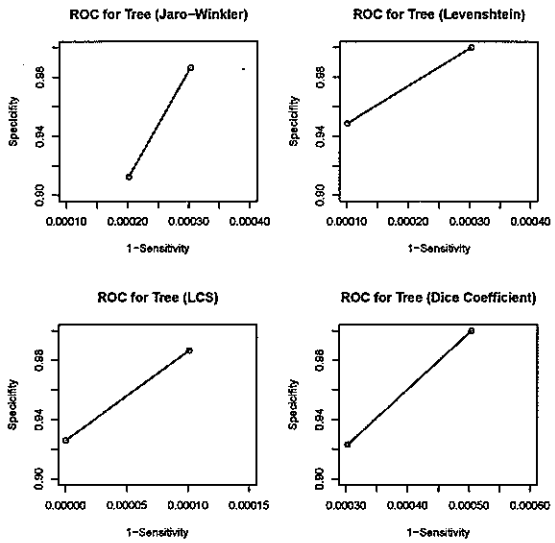

FIGURE 3. ROC curves for the classification trees for the four similarity scores.

We examine the ROC curves in order to analyze the tree predictions. The ROC curves for the logistic regression models do not show us anything new since the logistic regression models have no errors and are not shown here. The ROC curves in Figure 3 do very well in predicting matches, as they all have close to area 1 under the curve, which indicates that all four similarity scores predict matches and non matches well. Again, the extensions to (0,0) and (1,1) not show on the graphs. From the four ROC curves, we see that the Dice Coefficient actually has the most area under the curve, by a slim margin. This indicates that the Dice Coefficient performs the best of the four on this set of data using the classification tree method. We have a nonmetric performing three metrics.

4.4. **Fellegi-Sunter Expected Matching.** Felligi-Sunter Expected Matching is used to obtain a measurement of the accuracy of each of the similarity scores. The table contained in Figure 4 gives a summary of the results we obtained for Data Set 1. A description of the columns is as follows: in the method column, the suffix CI indicates that the particular row refers to a test in which conditional independence of events is assumed. The second and third columns give the numbers of Type 2 errors and Clerical Review cases obtained by running Expected Matching (with and without conditional independence) with each of our similarity scores on Data Set 1. A Type 2 error is the same as a false negative, or a pair that is assigned as

a nonmatch but is actually a match. There are no Type 1 errors, which are false positives, pairs that are assigned as a match but actually are not. The reason for this is that we assigned high cutoff values to each similarity score, meaning that designating a record pair as a match happens only if each of their components are very similar, which is highly unlikely if they are not really a match. An assignment of clerical review indicates that the record pair has differences which a particular metric is not capable of distinguishing satisfactorily, so the record is marked for human review.

The assumption of conditional independence is made when one believes that the individual components of a compound, conditional event occur independently of one another. In our case, this would mean that the characters contained in one data field for a particular person, i.e. the characters in a first name, have no statistical relation to the characters in any other data field, i.e. the person's last name. The assumption of conditional independence is not always valid, however it simplifies the computation of the threshold values needed to complete the Expected Matching method and assign values of Match, Non-Match, and Clerical Review appropriately.

Method	Type 2	Clerical Review
Jaro-Winkler	38	0
Jaro-Winkler-CI	38	91
Levenshtein	16	24
Levenshtein-CI	16	6
Dice Coefficient	58	35
Dice Coefficient-CI	73	103
LCS	35	5
LCS-CI	40	0

FIGURE 4. Table displaying the results of the Fellegi-Sunter Expected Matching tests conducted on Data Set 1.

There are several interesting features to note. First, the Levenshtein metric seems to have performed the best because of the low number of Type 2 errors and record pairs sent to clerical review. The second feature to note is that, in every case but one, namely the Levenshtein metric, the assumption of conditional independence caused a drop in accuracy. We do not think that this result can be generalized to all data sets of this kind. We see from Figure 4 that the Dice Coefficient performs the worst in this scenario, with the highest number of errors and record pairs sent to clerical review.

4.5. **Conclusions about Data Set 1.** Data Set 1 is a fairly nice data set in that it is small and our models fit to it very well. In the case of the logistic regression models, we likely overfit the data and cannot compare the results of our similarity scores against one another. For classification trees, the Dice Coefficient performs the best for Data Set 1, but performs the worst for Fellegi-Sunter Expected Matching. This large difference in performance is interesting and something we wish to investigate in further work. We note that conditional independence for Eexpected Matching is not a good assumption for this data set.

Score	Converged	First Name	Last Name	Major	Hometown
Jaro-Winkler	Yes	16.10	59.6	13.59	10.75
Levenshtein	Yes	-1.28	-2.17	-0.57	-0.34
LCS	Yes	385	564	267	1.23
Dice Coeff.	Yes	2e15	4e15	5e15	7e15

FIGURE 5. Table displaying the results of the logistic regression models for the four similarity scores for data set 2.

5. Record Comparison with Different Metrics: Data Set 2

To verify the results of our tests, we run the same tests on another set of data. This allows us to analyze our similarity scores without the results being attached to a single data set.

5.1. **Data Set 2.** Data Set 2 is based on a club roster at Carnegie Mellon University. There are two lists, each with 5 fields. There are 90 records in each list. The fields are first name, last name, major, hometown, and a unique identifying number. The first list is the 90 records without typos. The second is a duplicate of the first, with typos inserted in some of the fields. These lists are interesting because there are a few sets of siblings, meaning that the records automatically match on last name and hometown. This implies that last name and hometown are certainly not independent from one another. Again, each record pair is compared at each field for each similarity score. The result is four separate 8100×5 arrays. Each array contains 90 matches.

5.2. **Logistic Regression Models.** We run the logistic regression models for each array. The model we use is

$$\text{match}_i = \text{first name}_i + \text{last name}_i + \text{major}_i + \text{hometown}_i. \tag{5.1}$$

In contrast to Data Set 1, all these models converge for logistic regression. The table of coefficients is shown above in Figure 5. The Dice Coefficient's logistic model has very large coefficients. The Jaro-Winkler and LCS models have 0 significant predictors. The Levenshtein distance model has first name and major as significant predictors. All of first name, last name, major, and hometown are significant predictors for the Dice Coefficient.

The Jaro-Winkler model has false positives or nonmatches that are linked as a match. These errors are sibling pairs. The Levenshtein Distance model has two false positives, a different sibling pair than the Jaro-Winkler sibling pairs. The LCS model predicts matches very well, only having one false negative where there is a very large typo in one person's first name which changed it from James to John. The Dice Coefficient has the same results as the LCS model with one false positive on the same record pair. The following ROC curves in Figure 6 reflect the above analysis. The LCS model and Dice Coefficient have close to area 1 under the curve (for the Dice Coefficient, there is just a dot), whereas the other two scores perform less well, but are still impressive.

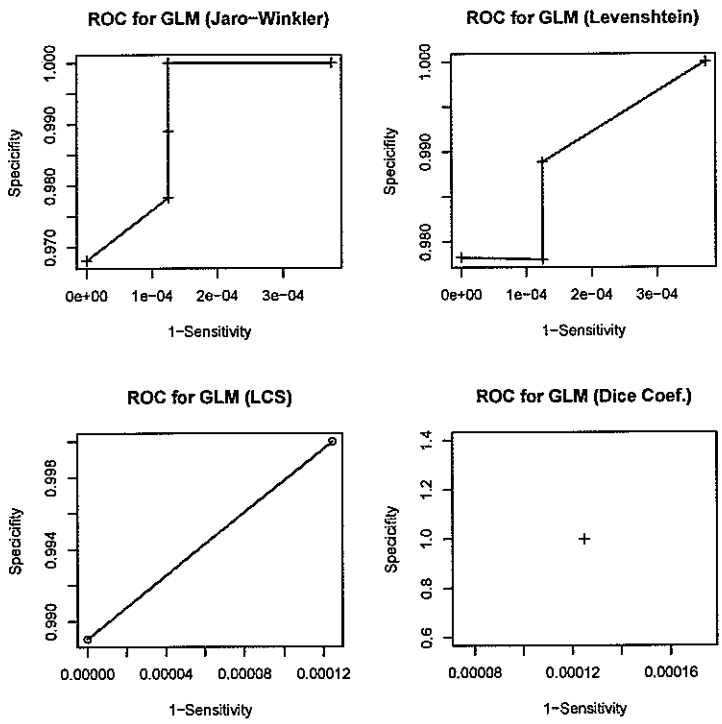

FIGURE 6. ROC curves for the logistic regression models for the four similarity scores for data set 2.

The LCS and Dice Coefficient Models perform the best with logistic regression and do a better job of identifying siblings as unique people. Siblings seem to be identified as the same person if there are a few letters in common within the majors for both the Jaro-Winkler and Levenshtein Edit Distance metrics.

5.3. **Classification Trees.** The classification trees for this data set for the different similarity scores are very similar to one another. We again run the same model for the classification trees algorithm as Equation 5.1. The trees are shown in Figure 7. The trees all split on last name and then first. Major is not used in the trees, whereas major does seem to have been an important factor in the regression models.

Again, the LCS and Dice Coefficient outperform the other two similarity scores. The ROC curves for the classification trees for data set 2 are displayed in Figure 8. In Figure 8, the ROC curves for the classification trees have only dots for the LCS and Dice Coefficient scores, meaning they perform well with low error rate.

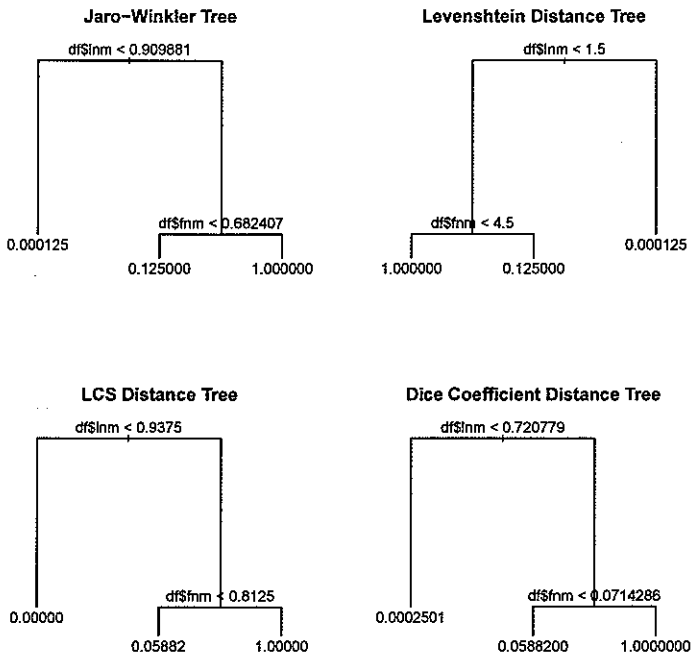

FIGURE 7. Classification Trees for the four similarity scores for data set 2.

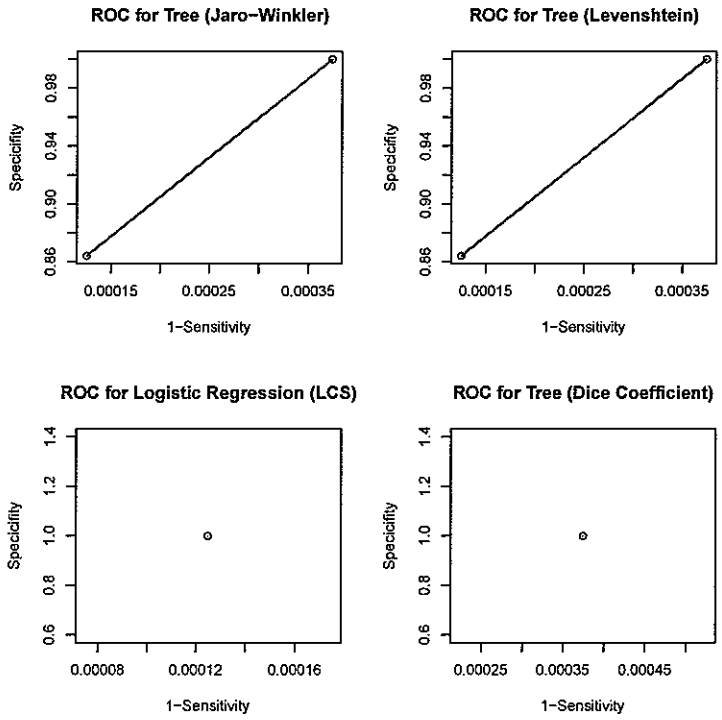

FIGURE 8. ROC Curves for the classification trees for the four similarity scores for data set 2.

5.4. **Expected Matching.** The summary results of our Expected Matching testing for Data Set 2 are presented below. Again, there are no Type 1 Errors.

Method	Type 2	Clerical Review
Jaro-Winkler	34	0
Jaro-Winkler-CI	47	52
Levenshtein	213	0
Levenshtein-CI	213	0
Dice Coefficient	32	0
Dice Coefficient-CI	50	47
LCS	25	0
LCS-CI	71	0

FIGURE 9. Table displaying the results of the Fellegi-Sunter Expected Matching tests conducted on Data Set 2.

Here we see a sharp contrast between the performance of the Levenshtein metric on Data Set 1 and on Data Set 2. This result is surprising, as the two data sets we use are of similar size and contain the same datatypes. Again, we see that the assumption of conditional independence is not beneficial. It should be noted that the similarity scores perform approximately equally as well on both data sets, taking into account the ratio of their sizes, with the exception being the Levenshtein metric, which performs drastically worse on the second data set.

5.5. **Conclusions about Data Set 2.** Data Set 2 turns out to be a more informative set than Data Set 1 in that the models clearly have differences amongst them. The logistic regression models perform better than the classification trees, but the LCS and Dice Coefficient both outperform the Jaro-Winkler and Levenshtein Edit Distance with respect to accuracy of matching. For Expected Matching, Levenshtein Edit Distance performs far worse than in Data Set 1, and Dice Coefficient has the second lowest error rate in contrast to Data Set 1 where it is by far the worst score. This seems to indicate that the Jaro-Winkler and Levenshtein Edit distance do not work as well on pairs of people who have the same last name. This is of note because the Jaro-Winkler function is used on Census data where there are presumably instances of different people with the same last name living in the same household.

One reason for the disparity in scores across the two data sets is that Data Set 1 has three numerical fields of short numbers whereas Data Set 2 has more text fields and 0 numerical fields. The Dice Coefficient and LCS are the best choices here for all the methods with Jaro-Winkler and Levenshtein Edit Distance lagging far behind.

6. RUN TIME

We briefly analyze the average run time of creating the arrays for the different similarity scores. We calculate the average time to create an array. The results are displayed in Figure 10.

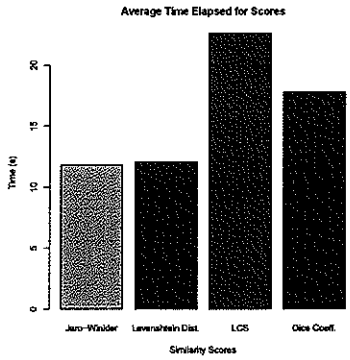

FIGURE 10. Average time to compute arrays for each similarity score.

The Jaro-Winkler score is the fastest, narrowly beating the Levenshtein Edit Distance. Slowest by a fair margin is the LCS, which is recursive in nature, and also significantly slower than the Jaro-Winkler and the Levenshtein Edit Distance is the Dice Coefficient. Surprisingly, the Levenshtein Edit Distance seems to have a similar run time to the Jaro-Winkler Edit distance, even though theoretically has a larger run time. Perhaps this indicates we are running a more optimized version of the Levenshtein Edit Distance than its base formulation. Not surprisingly, we completely implemented the Dice Coefficient, and it is not optimized and slower than Jaro-Winkler and Levenshtein Edit Distance. If running on large data sets, then running the LCS takes more than half the time it takes to run Jaro-Winkler on the data. This can be a very significant factor. The Dice Coefficient takes about 70% more time to run than Jaro-Winkler, meaning it also does not scale very well to large data sets.

7. SUMMARY, CONCLUSIONS, AND FURTHER WORK

We provide an introduction and theory behind metric and nonmetric functions. Specifically we go into detail about four similarity scores: the Jaro-Winkler metric, the Levenshtein Edit Distance metric, and LCS metric, and the Dice Coefficient nonmetric. We provide theoretical run times for each of the similarity scores and discuss the implementation of the LCS and Dice Coefficient, which are not initially provided in R. We then perform tests on two different data sets using different techniques of record linkage, including logistic regression, classification trees, and Fellegi-Sunter Expected Matching.

We notice that the logistic regression models differ between the two data sets, and that the LCS and Dice Coefficient are better at identifying siblings as unique individuals than the Jaro-Winkler and Levenshtein Edit Distance are. The classification trees all seem to have the same structure, except the Levenshtein Edit Distance is flipped around the center. This indicates that the similarity scores have the same type of scores for the same type of words. Again, the LCS and especially the Dice Coefficient perform very well in both data sets. Overall, the Dice Coefficient score seems to outperform the metrics because it has the lowest error rate. Although we have no indication that a nonmetric always outperforms a nonmetric.

We note that mathematically, we do not have a reason to use a metric score over a nonmetric score or the other way around. We would like to further examine the mathematical basis behind these similarity scores.

We observe that the behavior of the Levenshtein metric is sensitive to the data set. We also note that the assumption of conditional independence, although it eases computational aspects involved in our project, does not provide an increase in accuracy. All of the similarity scores exhibit similar behavior on both data sets, with the exception of the Levenshtein metric, which performs significantly worse on Data Set 2. This similar behavior on the data sets is not surprising as the data sets have similar structure in fields.

Examining the run time, we notice that Jaro-Winkler and Levenshtein Edit Distance are significantly faster than the LCS and Jaro-Winkler. In fact, if working with very large data, it may be a bad idea to run the LCS or Dice Coefficient even though they seem to perform better as the time cost may not be worth the gain in matching performance.

For further work, we would like to analyze combining metrics and nonmetrics as done by Cohen, Ravikumar, and Fienberg in [3]. We would also like to run the model on more complicated and different type data sets in order to be able to tests different sorts of metrics. Because of the significant difference in scoring for the Dice Coefficient in Data Set 1 amongst the classification trees, regression models, and Expected Matching, we would like to investigate the reason for the disparity. Looking into optimizing the run time of Dice Coefficient and LCS are of interest since they perform well but are slow to compute large data sets. With respect to Expected Matching, it would be beneficial to optimize the match cutoff values used for each test in order to maximize the performance of the similarity scores.

References

[1] D. Bakkelund. *An LCS-based string metric.* University of Oslo. September 23, 2009.

[2] P. Christen *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection.* Springer. (2012).

[3] W. Cohen, P. Ravikumar, and S. Fienberg. *A Comparison of String Distance Metrics for Name-Matching Tasks.* Proceedings of IJCAI-03 Workshop on Information Integration, page 73–78. (August 2003)

[4] A. Dholakia. *Introduction to Convolutional Codes with Applications.* Kluwer Academic Publishers. (1994).

[5] T. Herzog, F. Scheuren, and W. Winkler. *Data Quality and Record Linkage Techniques.* Springer. (2007).

[6] A. Leach and V.J. Gillet. *An Introduction to Chemoinformatics.* Springer. (2007).

[7] T. Skopal and B. Bustos. *On Nonmetric Similarity Search Problems in Complex Domains.* ACM Computing Surveys (2010).

[8] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search: The Metric Space Approach.* Springer. (2006).

APPENDIX A. CODE

```
#FUNCTIONS ############################
 lcs <- function(str1, str2){
      s <- 1 - stringdist(str1, str2, method="lcs")/max(nchar(c(str, str2)))
      return(s)
  }

#OLD RECURSIVE VERSION

lcs2 <- function(xs, ys) {
    if (nchar(xs) > 0 && nchar(ys) > 0) {    ## check that
    #we aren't in the base case
        xbeg <- substring(xs,1,nchar(xs)-1) # everything but last letter of xs
        xend <- substring(xs,nchar(xs))  ## last letter of xs
        ybeg <- substring(ys,1,nchar(ys)-1)
        yend <- substring(ys,nchar(ys))

        if (xend == yend) {
            return(paste(lcs(xbeg, ybeg), xend, sep=""))  ## recursive call o
            #n xbeg & ybeg  #paste("a","b",sep="") would output "ab"
        }
        else {
            r1 <- lcs(xs,ybeg)
            ##in the event that our two strings
            #do not have the same last character, we make two
            r2 <- lcs(xbeg,ys)
            ##recursive calls. Both calls are made
            #with one of the original strings, and the beginning of
            if (nchar(r1) >= nchar(r2)){
            ##the other string (since we know its
            #last letter isn't in common. Then a check is made to
                return(r1)
                ##see which recursive call gave the longer substring.
            }
            else {  ## the base case
                return(r2)
            }
        }
    }
    else {
        return("")
    }
}

#for dice coeff
bigrams <- function(string, num.bigrams){
    vec <- vector(length=num.bigrams)
    for (i in 1:num.bigrams){
        vec[i] <- substring(string, i, i+1)
    }
    return(vec)
}
#dice co similarity score
dice.co <- function(u,v){
    x.bigrams <- nchar(u)
    y.bigrams <- nchar(v)
    x.vec <- bigrams(u, x.bigrams)
    y.vec <- bigrams(v, y.bigrams)
    s <- 2*length(intersect(x.vec, y.vec))/(x.bigrams + y.bigrams)
    return(s)
}
#function to do logistic regression for first data set
#second data set is similar
do.logreg<-function(df){
    reg<-glm(df$id~df$fnm+df$lnm+df$byear+df$bm+df$bd, family=binomial, data=df)
return(reg)
}

#creates tree model for first data set
#second data set is similar
do.tree<-function(df){
reg<-tree(df$id~df$fnm+df$lnm+df$byear+df$bm+df$bd, data=df)
return(reg)
}

#function to create ROC table
rocVecs<-function(thresh, df, prob)
{
match.model<-ifelse(prob>=thresh, 1,0)
match.table<-table(df$id, match.model)
a<-match.table[1,1]
```

```
b<-match.table[1,2]
c<-match.table[2,1]
d<-match.table[2,2]
sens<-a/(a+c)
spec<-d/(d+b)
return(c(1-sens, spec))
}

#makes matrix of ROC values for different thresholds
plotMatrix<-function(thresh,df, prob)
{
mat<-array(dim=c(length(thresh),2))

for (i in (1:length(thresh)))
{
mat[i,]<-rocVecs(thresh[i],df, prob)
}
return(mat)
}

#puts everything into an array, takes in different similarity scores
makeComparisons<-function(listA, listB, fxn, ...){

rows<-length(listA[,1])
cols<-length(listA[1,])
comparisons.array<-array(dim=c(rows^2, cols))

#non clever way to make comparison matrix
comparison.fullmat<-array(dim=c(rows,rows, cols))
for (k in 1:cols){

for (i in 1:rows){
for (j in 1:rows){
if (k!=cols){
comparison.fullmat[i,j,k]<-fxn(as.character(listA[i,k], ...),
as.character(listB[j,k]))
}
else{
comparison.fullmat[i,j,k]<-ifelse(listA[i,k]==listB[j,k], 1,0)
}
}
}
}
index=0
for (i in 1:rows){
for (j in 1:rows){
index=index+1
for (k in 1:cols){
comparisons.array[index,k]<-comparison.fullmat[i,j,k]
}
}
}
return(data.frame(comparisons.array))
}

#LOGISTIC REGRESSION AND TREES,
#ROC CURVES, TIME ELAPSED CODE
#FOR DATA SET 1; DATA Set 2 very similar, not included

library(RecordLinkage)
library(stringdist)
library(tree)

source("reportFunctions.r")
listA<-read.table("list1.txt")

listB<- read.table("list2.txt")

listA<-subset(listA, select=-c(fname_c2, lname_c2))
listB<-subset(listB, select=-c(fname_c2, lname_c2))

#JARO WINKLER
comparisons.jaro<-makeComparisons(listA, listB, jarowinkler)
colnames(comparisons.jaro)<-c("fnm", "lnm", "byear", "bm", "bd", "id")

#logistic regression
reg.jaro<-do.logreg(comparisons.jaro)
summary(reg.jaro) #DID NOT CONVERGE
tree.jaro<-do.tree(comparisons.jaro)


#LEVENSHTEIN DISTANCE
comparisons.lev<-makeComparisons(listA, listB, levenshteinDist)
```

```
colnames(comparisons.lev)<-c("fnm", "lnm", "byear", "bm", "bd", "id")

#logistic regression
reg.lev<-do.logreg(comparisons.lev)
summary(reg.lev) #DID NOT CONVERGE
tree.lev<-do.tree(comparisons.lev)

#Dice coefficient
comparisons.dice<-makeComparisons(listA, listB, dice.co)
colnames(comparisons.dice)<-c("fnm", "lnm", "byear", "bm", "bd", "id")

#logistic regression
reg.dice<-do.logreg(comparisons.dice) #DID NOT CONVERGE
summary(reg.dice)
tree.dice<-do.tree(comparisons.dice)

#LCS
comparisons.lcs<-makeComparisons(listA, listB, lcs)
colnames(comparisons.lcs)<-c("fnm", "lnm", "byear", "bm", "bd", "id")
reg.lcs<-do.logreg(comparisons.lcs)
summary(reg.lcs) #DID NOT CONVERGE
tree.lcs<-do.tree(comparisons.lcs)

#ROC CURVE

reg.jaro.p<-reg.jaro$fit
tree.jaro.p<-predict(tree.jaro, comparisons.jaro)
prob.thresh<-seq(.1,.9, by=.1)
plot.jaro.glm<-plotMatrix(prob.thresh, df=comparisons.jaro, prob=reg.jaro.p)
plot.jaro.tree<-plotMatrix(prob.thresh, df=comparisons.jaro, prob=tree.jaro.p)

#LEVENSHTEIN
reg.lev.p<-reg.lev$fit
tree.lev.p<-predict(tree.lev, comparisons.lev)
plot.lev.glm<-plotMatrix(prob.thresh, df=comparisons.lev, prob=reg.lev.p)
plot.lev.tree<-plotMatrix(prob.thresh, df=comparisons.lev, prob=tree.lev.p)

#DICE COEFF

reg.dice.p<-reg.dice$fit
tree.dice.p<-predict(tree.dice, comparisons.dice)
plot.dice.glm<-plotMatrix(prob.thresh, df=comparisons.dice, prob=reg.dice.p)
plot.dice.tree<-plotMatrix(prob.thresh, df=comparisons.dice, prob=tree.dice.p)

#LCS

reg.lcs.p<-reg.lcs$fit
tree.lcs.p<-predict(tree.lcs, comparisons.lcs)
plot.lcs.glm<-plotMatrix(prob.thresh, df=comparisons.lcs, prob=reg.lcs.p)
plot.lcs.tree<-plotMatrix(prob.thresh, df=comparisons.lcs, prob=tree.lcs.p)

#TIME RUN
system.time(comparisons.jaro<-makeComparisons(listA, listB, jarowinkler))
   # user  system elapsed
  # 11.98    0.02   12.12
 system.time(comparisons.lev<-makeComparisons(listA, listB, levenshteinDist))
    # user  system elapsed
   # 12.4     0.0    12.5
system.time(comparisons.dice<-makeComparisons(listA, listB, dice.co))
   # user  system elapsed
  # 17.69    0.02   17.83
system.time(comparisons.lcs<-makeComparisons(listA, listB, lcs))
 # user  system elapsed
  # 25.46    0.00   25.65


 times<-c(11.98, 12.4,25.46, 17.69 )
 names(times)<-c("Jaro-Winkler", "Levenshtein Dist.", "LCS", "Dice Coeff.")

 pdf("reportTime.pdf")
 barplot(times, main="Time Elapsed for Scores", xlab="Similarity Scores",
    ylab="Time (s)", col=c("green", "blue", "purple", "red"))
 dev.off()

  #TREE PLOTS
pdf("treesReport1.pdf")
par(mfrow=c(2,2))
#jaro
plot(tree.jaro, cex=0.5)
text(tree.jaro)
title("Jaro-Winkler Tree")
#lev
plot(tree.lev, cex=0.5)
```

```
text(tree.lev)
title("Levenshtein Distance Tree")
#lcs
plot(tree.lcs, cex=0.5)
text(tree.lcs)
title("LCS Distance Tree")
#dice
plot(tree.dice, cex=0.5)
text(tree.dice)
title("Dice Coefficient Distance Tree")

dev.off()

#ROC CURVES
#TREES
pdf("treesROCReport1.pdf")
par(mfrow=c(2,2))
#jaro
plot(plot.jaro.tree[,1], plot.jaro.tree[,2],  xlab="1-Sensitivity",
ylab="Specicifity", main="ROC for Tree (Jaro-Winkler)", xlim=c(.0001, .0004), ylim=c(.90, 1))
lines(plot.jaro.tree[,1], plot.jaro.tree[,2], col="red", lw=2)
#lev
plot(plot.lev.tree[,1], plot.lev.tree[,2],  xlab="1-Sensitivity",
ylab="Specicifity", main="ROC for Tree (Levenshtein)", xlim=c(.0001, .0004), ylim=c(.90, 1))
lines(plot.lev.tree[,1], plot.lev.tree[,2], col="red", lw=2)
#lcs
plot(plot.lcs.tree[,1], plot.lcs.tree[,2],  xlab="1-Sensitivity",
ylab="Specicifity", main="ROC for Tree (LCS)", xlim=c(0, .00015), ylim=c(.90, 1))
lines(plot.lcs.tree[,1], plot.lcs.tree[,2], col="red", lw=2)

#dice
plot(plot.dice.tree[,1], plot.dice.tree[,2],  xlab="1-Sensitivity",
ylab="Specicifity", main="ROC for Tree (Dice Coefficient)", xlim=c(.0003, .0006), ylim=c(.90, 1))
lines(plot.dice.tree[,1], plot.dice.tree[,2], col="red", lw=2)
dev.off()


 ###EM###########

lv.thresh.compare.2 <- function(a){
     if (a == 1 || a == 2 || a == 3 || a == 5 || a == 6){
          return(0)
      }
      else {
          return(1)
      }
 }

 > lcs.thresh.compare.2 <- function(a){
      if (a == 1 || a == 2 || a == 5 || a == 6){
          return(0)
      }
      else {
          return(1)
      }
  }
 > jw.thresh.compare.2 <- function(a){
      if (a == 1 || a == 2){
          return(0)
      }
      else if (a == 5){
          return(0.5)
      }
      else {
          return(1)
      }
  }
 > dc.thresh.compare.2 <- function(a){
      if (a == 1  || a == 2){
          return(0)
      }
      else if (a == 5){
          return(0.5)
      }
      else {
          return(1)
      }
  }

  ##Scripts for 4 scores
  ##Only one included below

  pl.count <- 0
```

```
p2.count <- 0
p3.count <- 0
p4.count <- 0
p5.count <- 0
p6.count <- 0
p7.count <- 0
p8.count <- 0
p9.count <- 0
p10.count <- 0
p11.count <- 0
p12.count <- 0
count <- 0

for (i in 1:90){
  for (j in 1:90){
    count <- count + 1
    if (is.match(i,j)) {
      ifelse(LV.Score.Matrix[count,1] < 0.90, p1.count <- p1.count + 1, p2.count <- p2.count + 1)
      ifelse(LV.Score.Matrix[count,2] < 0.90, p3.count <- p3.count + 1, p4.count <- p4.count + 1)
      ifelse(LV.Score.Matrix[count,3] < 0.90, p5.count <- p5.count + 1, p6.count <- p6.count + 1)
    }
    else {
      ifelse(LV.Score.Matrix[count,1] < 0.90, p7.count <- p7.count + 1, p8.count <- p8.count + 1)
      ifelse(LV.Score.Matrix[count,2] < 0.90, p9.count <- p9.count + 1, p10.count <- p10.count + 1)
      ifelse(LV.Score.Matrix[count,3] < 0.90, p11.count <- p11.count + 1, p12.count <- p12.count + 1)
    }
  }
}

LV.R1.2 <- ((p1.count * p3.count * p5.count)/(match.count^3))/((p7.count * p9.count * p11.count)/(nonmatch.count^3))
LV.R2.2 <- ((p1.count * p3.count * p6.count)/(match.count^3))/((p7.count * p9.count * p12.count)/(nonmatch.count^3))
LV.R3.2 <- ((p1.count * p4.count * p5.count)/(match.count^3))/((p7.count * p10.count * p11.count)/(nonmatch.count^3))
LV.R4.2 <- ((p1.count * p4.count * p6.count)/(match.count^3))/((p7.count * p10.count * p12.count)/(nonmatch.count^3))
LV.R5.2 <- ((p2.count * p3.count * p5.count)/(match.count^3))/((p8.count * p9.count * p11.count)/(nonmatch.count^3))
LV.R6.2 <- ((p2.count * p3.count * p6.count)/(match.count^3))/((p8.count * p9.count * p12.count)/(nonmatch.count^3))
LV.R7.2 <- ((p2.count * p4.count * p5.count)/(match.count^3))/((p8.count * p10.count * p11.count)/(nonmatch.count^3))
LV.R8.2 <- ((p2.count * p4.count * p6.count)/(match.count^3))/((p8.count * p10.count * p12.count)/(nonmatch.count^3))

LV.Clerical.Score.Vector.2 <- c()

for (count in 1:8100){
  x <- LV.Score.Matrix[count,1]
  y <- LV.Score.Matrix[count,2]
  z <- LV.Score.Matrix[count,3]
  LV.Clerical.Score.Vector.2[count] <- lv.thresh.compare.2(lv.pattern.discern(x, y, z))
}
lv.discrepancy.indices.2 <- c()
for (count in 1:8100){
  if (LV.Clerical.Score.Vector.2[count] != LV.Score.Matrix[count,4]){
    lv.discrepancy.indices.2 <- c(lv.discrepancy.indices.2, count)
  }
}
lv.error.type.vector.2 <- c()
for (i in lv.discrepancy.indices.2){
  if (LV.Score.Matrix[i,4] == 1 && LV.Clerical.Score.Vector.2[i] == 0){
    lv.error.type.vector.2 <- c(lv.error.type.vector.2, "Type 1")
  }
  else if (LV.Score.Matrix[i,4] == 0 && LV.Clerical.Score.Vector.2[i] == 1){
    lv.error.type.vector.2 <- c(lv.error.type.vector.2, "Type 2")
  }
  else if (LV.Clerical.Score.Vector.2[i] == 0.5){
    lv.error.type.vector.2 <- c(lv.error.type.vector.2, "Clerical Review")
  }
  else {
    lv.error.type.vector.2 <- c(lv.error.type.vector.2, "???")
  }
}

lv.t1.2.count <- 0
lv.t2.2.count <- 0
for (xs in lv.error.type.vector.2) {
  if (xs == "Type 1") {
    lv.t1.2.count <- lv.t1.2.count + 1
  }
  if (xs == "Type 2") {
    lv.t2.2.count <- lv.t2.2.count + 1
  }
}
lv.cr.2.count <- length(lv.error.type.vector.2) - lv.t1.2.count - lv.t2.2.count
```

CARNEGIE MELLON UNIVERSITY
E-mail address: sgallagh@andrew.cmu.edu

CARNEGIE MELLON UNIVERSITY
E-mail address: grussell@andrew.cmu.edu