Slice Distance

Petr Slavík
CEDAR, SUNY Buffalo
slavik@cedar.buffalo.edu

April 8, 2000

Abstract

We introduce a novel way of computing an image-independent recognizer-dependent distance between ASCII words. This distance is a natural generalization of the minimum edit distance in the context of handwriting recognition. It is based on confusion matrices for only parts of characters called character "slices". This "slice distance" naturally explains the confusions and misrecognitions encountered in recognition of handwritten words and phrases by a particular recognizer and could be used to exploit both the weak and the strong points of the recognizer.

Even though we describe the techniques for computing the slice distance using an example of a particular word recognizer, our methods can be easily generalized to almost any segmentation-based handwritten-word recognition system.

1 Introduction

Researchers in the area of word recognition naturally seek a measure that would allow them to predict in advance how difficult it would be for their word recognizer to distinguish among two (or more) words. In this paper, we introduce a distance function between ASCII words whose behavior closely follows the performance of a segmentation based word recognizer in that words that are "close" to each other with respect to this distance are also easy to be confused by the recognizer. The distance is computed using confusion matrices for character "slices" (parts of characters consisting of one or more elementary segments) and thus we call it the "slice distance".

Slice distance has many different application in the area of handwriting recognition, among them computing the lexicon density, evaluation of recognition results, dynamic combination of recognizers, etc.—see [3] for more details.

1.1 Related Results

Distance between two words is usually measured in terms of the total cost of edit operations that are needed to transform one word into the other one. Originally, computing of such a distance was motivated by correcting of typing and spelling errors and thus the only edit operations considered were *replacement* of one character by another character, *deletion* of a single character, and *insertion* of a single character.

In 1974, Wagner and Fisher [4] described a simple dynamic program that computes the minimum edit distance between two ASCII strings as the cost of a cheapest "trace" of the three elementary edit operations. Intuitively, a *trace* is a special sequence of elementary edit operations that transform first string into the second one while requiring at most one edit operation per character and a strict left-to-right order of elementary edit operations—see [4] for details. Under the assumption that the costs of elementary edit operations satisfy triangle inequality, Wagner and Fisher proved that the cost of the cheapest trace is indeed the same as the cost of the cheapest sequence of elementary edit operations.

Later on, other researchers have used minimum edit distance for different problems—correction of recognition results of printed-word recognizers, DNA matching, etc. with only slight modifications of the original edit operations.

Researchers in the area of handwriting recognition soon realized that the three edit operations of *replacement*, *deletion*, and *insertion* are not sufficient to account for errors in the recognition of handwritten words. Seni, Kripasundar, and Srihari [2] generalized the minimum edit distance by considering additional edit operations—*substitution* of a pair of characters, *merge* of two characters into one, and a *split* of one character into two. Similarly as Wagner and Fisher in [4], Seni et al. computed the generalized edit distance[1] as the cost of cheapest trace. The costs of elementary edit operations were determined empirically by observing samples of handwritten characters and examples of misrecognition errors of a handwritten-word recognizer. Based on these observations, Seni et al. decided whether each elementary transformation is VERY LIKELY, LIKELY, UNLIKELY, or IMPOSSIBLE and then assigned a (rather arbitrary) cost to each edit operation based on the type of operation and the level of likelihood. According to Seni et al., such weights of elementary transformations were independent of any particular recognizer and the generalized edit distance represented a *generic* measure of possible confusion between two handwritten words.

While adding some additional edit operations accounts for some errors in handwriting recognition (like confusing 'cl' and 'd'), it is still not general enough to explain errors like confusing 'rst' with 'on' in Figure 2. Our slice distance, on the other hand, accounts for *all* typical misrecognitions of handwritten words and phrases.

2 Recognizer-Dependent Distance

Let us first discuss the concept of a distance between two ASCII words with respect to a given recognizer. Having two words, $w1$ and $w2$, we would like to measure how far the word $w2$ is from the word $w1$, or better yet how difficult it would be for a given recognizer to confuse words $w1$ and $w2$. To determine such a distance for a holistic recognizer that treats the entire word as a single object is relatively easy—simply compute in the feature space of all recognizable words the feature distances between all prototypes of $w1$ and all prototypes of $w2$ and define the distance between $w1$ and $w2$ as minimum of all such feature distances.

The case of segmentation based recognizers, where each letter is considered a separate object, is more difficult. One way to determine a distance between two ASCII words with respect to a segmentation based recognizer is to use the minimum edit distance between $w1$ and $w2$—see for example [4]. This approach can be used with recognizers that are able to

[1] In this case, the cost of the cheapest trace is in general larger than the minimum distance even when the costs of elementary transformations satisfy the triangle inequality. Seni, Kripasundar,and Srihari in [2] incorrectly claim that the cost of the cheapest trace and the minimum generalized edit distance are the same *without* even assuming that triangle inequality holds among elementary distances.

correctly segment a word image into characters *without* recognizing the characters first, as is the case for recognizers of machine print. In such a case, one can use samples of training words and training characters to determine the cost of elementary edit operations (deletion, insertion, and substitution) with respect to a given recognizer.

In this paper, our focus is on recognizers of handwritten words and phrases (even though the ideas presented here could be modified for any word recognizer). Such recognizers typically combine character segmentation with recognition and hence the minimum edit distance as described above cannot be used. In what follows, we will use handwritten word recognizer called WMR (the Word Model Recognizer [1]) to describe an alternative way of computing recognizer-dependent image-independent distance between ASCII words.

2.1 WMR

Let us now describe the inner workings of the Word Model Recognizer. This will guide us later on in designing the recognizer-dependent distance function. Given an image of handwritten word, WMR first over-segments the image into sub-characters—see Figure 1.

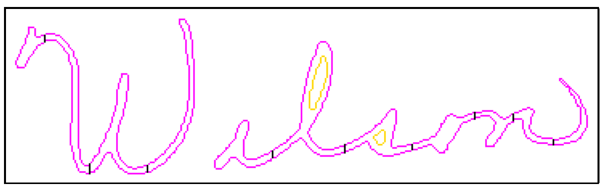

Figure 1: Segmented handwritten image in contour representation

Then, in the recognition phase, given an ASCII word from the lexicon, the segments are dynamically combined so as to obtain the best possible match between the word and the image; that is, to minimize the total distance between features of character templates corresponding to the letters in the ASCII word and the features of segment combinations from the image.

2.2 Motivation for Slice distance

Assume that WMR is presented with image in Figure 1 and the lexicon consists of two entries—'Wilson' and 'Amherst'. After dynamically checking all the possible segment combinations, WMR would correctly determine that the best way to match the image and the word 'Wilson' is to match segments 1-4 with `'W'`, segment 5 with `'i'`, segment 6 with `'l'`, etc. The best way to match the image against 'Amherst' would be to match segment 1 with `'A'`, segments 2-5 with `'m'`, segment 6 with `'h'`, segment 7 with `'e'`, segment 8-9 with `'r'`, segment 10 with `'s'`, and finally segment 11 with `'t'`—see Figure 2. Clearly the score of the second matching would be worse then the score of the first matching, hence the recognizer would correctly choose 'Wilson' as its first choice.

Figure 2 illustrates how confusions could possibly arise in determining the best possible answer. Letter `'A'` was matched with the same part of the image as the left part of letter `'W'`, left part of letter `'m'` was matched with the same "slice" of the image as the right part of `'W'`, right part of letter `'m'` was matched with the same slice of the image as letter `'i'`, etc. Hence to determine how difficult it would be to confuse 'Wilson' and 'Amherst'

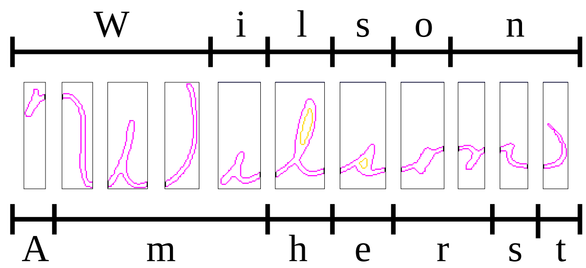

Figure 2: Matching of ASCII words with image

we have to first determine, how difficult it is to confuse `'A'` with the left part of `'W'`, left part of `'m'` with the right part of `'W'`, right part of `'m'` with a complete letter `'i'`, etc. Thus in general we must have information about how easy it is to confuse a slice of one character with a slice of another character. And not only that. Since we do not know the image before hand (and we want the distance to be image independent) we have to consider all the possible ways of confusing 'Wilson' with 'Amherst'; i.e. we have to consider all possible segmentation points of a hypothetical image and all possible ways of matching words 'Wilson' and 'Amherst' with the segments of such image. Then we choose the worst-case scenario (i.e. the smallest distance) among all possible combinations. This would be the measure of confusion between 'Wilson' and 'Amherst'. Since this measure depends ultimately on the distance between character "slices", we call it a "slice distance".

3 Computing the Slice Distance

As mentioned above, computation of the slice distance consists of two individual tasks—determining the elementary distances between all meaningful slices of characters, and then using these elementary distances as weights in the dynamic program that computes the slice distance between any two ASCII words.

3.1 Elementary Distances

Elementary distances between slices of different characters are computed during the training phase of WMR and stored in several 26 by 26 confusion matrices. These matrices are a natural generalization of confusion matrices between complete characters.

During training, WMR is presented with several thousand images of handwritten words. WMR over-segments each image, and the elementary segments are then *manually* combined into complete characters. These characters then serve as character templates[2].

It is easy to modify the training program to not only store the complete characters but also all their slices. Each such slice is stored together with the information about character class, number of segments in the slice, and which part of the complete character we are looking at (left, right, or middle).

To compute elementary distances, for each pair of characters and each elementary distance type, we computed the minimum over the distances between all possible slices. Thus

[2]To be precise, characters are first clustered based on the similarity of their features. Then for each cluster, the average of all characters in that cluster is used as template.

for example to compute the elementary distance between left part of `'A'` and right part of `'B'` such that both consist of exactly 2 elementary segments, distances between all 2-segment left slices of `'A'` and all 2-segment right slices of `'B'` were computed, and the elementary distance was set to their minimum.

Since WMR allows between 1 and 4 elementary segments per character, for each character, we have extracted left and right slices of 1 to 3 elementary segments, middle slices of 1 and 2 segments (from characters of size 3 and 4 respectively), and 1-segment middle-left and middle-right slices (from characters of size 4). Then we computed altogether 17 26x26 confusion matrices:

$SS1$, $SS2$, $SS3$, $SS4$ — elementary distances between complete characters of sizes 1, 2, 3, and 4, respectively;

$LR1$, $LR2$, $LR3$ — elementary distances between left slices and right slices of sizes 1, 2, and 3, respectively;

$LS1$, $LS2$, $LS3$ — elementary distances between left slices and complete characters of sizes 1, 2, and 3, respectively;

$RS1$, $RS2$, $RS3$ — elementary distances between right slices and complete characters of sizes 1, 2, and 3, respectively;

$MS1$, $MS2$ — elementary distances between middle slices and complete characters of sizes 1, and 2, respectively;

LMS, RMS — elementary distances between left and right middle slices and complete characters of size 1.

Note, that there is no point computing distances like $RM1$, $LL2$, etc. If, for example, right part of one character is being confused with a middle part of another character, both parts can be "extended" to the left, to get one of the following three cases already covered by the elementary distances—RLi, SMi, or SLi, where $i \geq 2$.

3.2 Dynamic Program

To compute the "slice distance" between two ASCII words $w1$ and $w2$, we consider all the possible meaningful (depending on words $w1$ and $w2$) numbers of segments, and all possible ways of combining the segments to match individual characters of each word. Character boundaries from each word determine the boundaries of slices. The slice distance between words $w1$ and $w2$ is then the minimum sum of elementary slice distances for each such combination. We denote the minimum slice distance between two ASCII words $w1$ and $w2$ by $msd(w1, w2)$. If $w1$ and $w2$ cannot be confused (due to significantly different lengths), we define $msd(w1, w2) = \infty$.

Checking *all* the possible ways of confusing words $w1$ and $w2$ one-by-one is not very efficient. Fortunately, there is a dynamic program that computes the slice distance in time $O(|w1| \cdot |w2| \cdot (|w1| + |w2|))$.

Let us first define several concepts needed to describe the dynamic program. Given words $w1$ and $w2$, we denote by $n1$ and $n2$ the number of characters in each word, i.e. $n1 = |w1|$ and $n2 = |w2|$. We denote by c_i the i-th character of word $w1$ and by d_j the j-th character of word $w2$. Hence $w1 = c_1 c_2 \dots c_{n1}$ and $w2 = d_1 d_2 \dots d_{n2}$. We denote by `max_K` the maximum number of segments of a (hypothetical) image matching both $w1$ and $w2$.

Let w be a word of n characters and let m be a matching between w and a (hypothetical) image of K segments. We say that i-th character of word w *ends* at segment k if k is the last segment of the super-segment that matches the i-th character. For a particular way of matching the word and the image, we denote this ending segment by $m(i)$, $1 \le i \le n$. Clearly $m(|w|) = K$. Additionally, we define $m(0) = 0$.

Since WMR allows a complete character to match between 1 and 4 consecutive segments of the image, $1 \le m(i) - m(i-1) \le 4$ and `max_K` $= 4 * \min(|w1|, |w2|)$.

We store the partial results of our computation in the following 3- or 4-dimensional matrices ($i = 1, \dots, n_1$, $j = 1, \dots, n_2$, $k = 1, \dots,$ `max_K` and $e = 1, \dots, 3$):

`fese`[i][j][k]: The smallest possible distance between first k segments of word $w1$ and first k segments of word $w2$ under the condition $k = m(i) = m(j)$. The name `fese` stands for "first-even, second-even" and corresponds to those matchings where the i-th and j-th characters of words $w1$ and $w2$ are aligned, i.e. their ending segments coincide. for example in Figure 2, the cost of matching 'Wi' with 'Am', would be stored as `fese[2][2][5]`.

`fesc`[i][j][k][e]: The smallest possible distance between first k segments of word $w1$ and first k segments of word $w2$ under the condition $k = m(i) = m(j) - e$. The name `fesc` stands for "first-even, second-cut" and corresponds to those matchings where i-th character of $w1$ ends within j-th character of $w2$, and there are exactly e segments of j-th character left "sticking out" beyond the last segment of i-th character. For example in Figure 2, the cost of (partially) matching 'Wilso' with 'Amher', would be stored as `fesc[5][5][8][1]`.

`fcse`[i][j][k][e]: The smallest possible distance between first k segments of word $w1$ and first k segments of word $w2$ under the condition $k = m(i) - e = m(j)$. The name `fcse` stands for "first-cut, second-even" and corresponds to those matchings where j-th character of $w2$ ends within i-th character of $w1$, and there are exactly e segments of i-th character left "sticking out" beyond the last segment of j-th character. For example in Figure 2, the cost of (partially) matching 'Wilson' with 'Amher', would be stored as `fcse[6][5][9][2]`.

Most of the elements of matrices `fese[][][]`, `fesc[][][][]`, and `fcse[][][][]` do not contain meaningful values, since there is no image and no matching that would satisfy the required conditions. There are two possibilities how to deal with this:

(a) Initialize all elements of all three matrices to ∞ (with the exception of `fese[0][0][0]`) and then compute all the elements one-by-one (starting with $k = 1$).

(b) Limit the ranges of k, i, j, and e to meaningful values only, thus avoiding unnecessary computations.

Since most of the elements of matrices `fese[][][]`, `fesc[][][][]`, and `fcse[][][][]` are meaningless, we have used the second approach.

The dynamic program now consists of the following three steps.

(A) Initialization:

$$\texttt{fese[0][0][0]} = 0$$

(B) Compute the (meaningful) values of matrices `fese[][][]`, `fesc[][][][]`, and `fcse[][][][]` for k segments (starting with $k = 1$, $k = 2$, up to $k =$ `max_K`) from the (meaningful) values for $k-1$, $k-2$, $k-3$, and $k-4$ segments using the following formulas:

$$\texttt{fese}[i][j][k] = \min \begin{cases} \min_{r=1,\dots,4} \texttt{fese}[i-1][j-1][k-r] + \texttt{SS}r[c_i][d_j] \\ \min_{r=1,\dots,3} \texttt{fesc}[i-1][j][k-r][r] + \texttt{SR}r[c_i][d_j] \\ \min_{r=1,\dots,3} \texttt{fcse}[i][j-1][k-r][r] + \texttt{RS}r[c_i][d_j] \end{cases}$$

$$\texttt{fesc}[i][j][k][e] = \min \begin{cases} \min_{r=1,\dots,4-e} \texttt{fese}[i-1][j-1][k-r] + \texttt{SL}r[c_i][d_j] \\ \min_{r=1,\dots,4-e} \texttt{fcse}[i][j-1][k-r][r] + \texttt{RL}r[c_i][d_j] \\ \min_{r=1,2} \texttt{fesc}[i-1][j][k-r][r] + \texttt{SM}r[c_i][d_j] \text{ for } e=1 \\ \texttt{fesc}[i-1][j][k-1][1] + \texttt{SMR}[c_i][d_j] \text{ for } e=1 \\ \texttt{fesc}[i-1][j][k-1][1] + \texttt{SML}[c_i][d_j] \text{ for } e=2 \end{cases}$$

$$\texttt{fcse}[i][j][k][e] = \min \begin{cases} \min_{r=1,\dots,4-e} \texttt{fese}[i-1][j-1][k-r] + \texttt{LS}r[c_i][d_j] \\ \min_{r=1,\dots,4-e} \texttt{fesc}[i-1][j][k-r][r] + \texttt{LR}r[c_i][d_j] \\ \min_{r=1,2} \texttt{fcse}[i][j-1][k-r][r] + \texttt{MS}r[c_i][d_j] \text{ for } e=1 \\ \texttt{fcse}[i][j-1][k-1][1] + \texttt{MRS}[c_i][d_j] \text{ for } e=1 \\ \texttt{fcse}[i][j-1][k-1][1] + \texttt{MLS}[c_i][d_j] \text{ for } e=2 \end{cases}$$

(C) Compute the minimum slice distance between $w1$ and $w2$. Assuming that we know all the values in matrices `fese[][][]`, `fesc[][][][]`, and `fcse[][][][]`, the slice distance between $w1$ and $w2$ is then

$$msd(w1, w2) = \min_k \texttt{fese}[n1][n2][k].$$

The formulas above are all straightforward and simply enumerate all the possible ways of matching two characters to the same parts of the image. It only remains to provide the reader with the details of determining meaningful ranges of i, j, k, and e.

Step 1: Determine the range for K—the total number of elementary segments

Since WMR allows between 1 to 4 elementary segments per complete character, two words $w1$ and $w2$ can be possibly confused on a hypothetical image that gets segmented by WMR into K segments, where

$$\texttt{min\_K} = \max(|w1|, |w2|) \le K \le \texttt{max\_K} = 4 * \min(|w1|, |w2|).$$

If no such K exists (that is, `min_K` $>$ `max_K`; that is, one word is more than 4 times longer than the other), we set $msd(w1, w2) = \infty$.

Step 2: Compute arrays '`min_dur[]`' and '`max_dur[]`' for both words

Given a word w with $|w| = n$, and the values `min_K` and `max_K` such that `min_K` $\geq n$ and `max_K` $\leq 4n$ we define the arrays `min_dur[]` and `max_dur[]` as

$$\texttt{min\_dur}[i] = \min\left\{ m(i) \;\middle|\; \begin{array}{l} m \text{ varies over all possible matchings between } w \\ \text{and } K \text{ segments with } \texttt{min\_K} \leq K \leq \texttt{max\_K} \end{array} \right\}$$

and

$$\texttt{max\_dur}[i] = \max\left\{ m(i) \;\middle|\; \begin{array}{l} m \text{ varies over all possible matchings between } w \\ \text{and } K \text{ segments with } \texttt{min\_K} \leq K \leq \texttt{max\_K} \end{array} \right\}$$

for $i = 0, \ldots, n$.

Given a word w and an image consisting of K elementary segments, the last segment of the first character can be segments 1, 2, 3, or 4; the last segment of the second character can be segments 2, 3, …, up to segment 8, etc. Hence `min_dur[1]` $= 1$, `max_dur[1]` $= 4$, `min_dur[2]` $= 2$, `max_dur[2]` $= 8$, etc. Similarly, last segment of the last character must be segment K, last segment of the previous character can be any of the segments $K-4$, $K-3$, $K-2$, or $K-1$, etc. Hence `min_dur[`n`]` $=$ `max_dur[`n`]` $= K$, `min_dur[`$n-1$`]` $= K-4$, `max_dur[`$n-1$`]` $= K-1$, etc. All this assuming that the word is not too long or too short compare with the number of elementary segments. If the word were too short, many characters would have to contain more than one segment, if the word were too long, many characters would have to contain less than 4 segments, hence the above might not hold.

The following are the precise formulas for arrays `min_dur[]` and `max_dur[]`, given that K is in the range `min_K` $\leq K \leq$ `max_K` and $|w| = n$, with `min_K` $\geq n$ and `max_K` $\leq 4n$.

$$\texttt{min\_dur}[i] = \max\{i, \texttt{min\_K} - 4 * (n - i)\}$$

and

$$\texttt{max\_dur}[i] = \min\{4 * i, \texttt{max\_K} - (n - i)\}.$$

Step 3: Compute arrays '`min_char[]`' and '`max_char[]`' for both words

Given a word w with $|w| = n$, variables `min_char[`k`]` and `max_char[`k`]` for $k = 0, \ldots,$ `max_K` contain the ID of the first and the last character of word w that can end at segment k. In other words,

$$\texttt{min\_char}[k] = \min\{i \mid m(i) = k \text{ and } m \text{ varies over all possible matchings}\}$$

and

$$\texttt{max\_char}[k] = \max\{i \mid m(i) = k \text{ and } m \text{ varies over all possible matchings}\}.$$

Arrays `min_char[]` and `max_char[]` can be easily computed using the formulas

$$\begin{array}{lcll} \texttt{min\_char}[k] & = & i & \text{for } \texttt{max\_dur}[i-1] < k \leq \texttt{max\_dur}[i], \\ \texttt{max\_char}[k] & = & i-1 & \text{for } \texttt{min\_dur}[i-1] \leq k < \texttt{min\_dur}[i]. \end{array}$$

Notice that $\texttt{min\_char[0]} = \texttt{max\_char[0]} = 0$.

It is now clear that the values of '$\texttt{fese[}i\texttt{][}j\texttt{][}k\texttt{]}$' are meaningful only when $0 \le k \le \texttt{max\_K}$, $\texttt{min\_char1[}k\texttt{]} \le i \le \texttt{max\_char1[}k\texttt{]}$, and $\texttt{min\_char2[}k\texttt{]} \le j \le \texttt{max\_char2[}k\texttt{]}$, the values of '$\texttt{fesc[}i\texttt{][}j\texttt{][}k\texttt{][}e\texttt{]}$', are meaningful only if $0 \le k$, $k{+}e \le \texttt{max\_K}$, $\texttt{min\_char1[}k\texttt{]} \le i \le \texttt{max\_char1[}k\texttt{]}$, and $\texttt{min\_char2[}k+e\texttt{]} \le j \le \texttt{max\_char2[}k+e\texttt{]}$, and similarly the values of '$\texttt{fcse[}i\texttt{][}j\texttt{][}k\texttt{][}e\texttt{]}$', are meaningful only if $0 \le k$, $k{+}e \le \texttt{max\_K}$, $\texttt{min\_char1[}k+e\texttt{]} \le i \le \texttt{max\_char1[}k+e\texttt{]}$, and $\texttt{min\_char2[}k\texttt{]} \le j \le \texttt{max\_char2[}k\texttt{]}$.

It is now straightforward to compute the meaningful elements of matrices '`fese[][][]`', '`fesc[][][][]`', and '`fcse[][][][]`' for a given k applying the formulas above only on the meaningful elements for $k-1$, $k-2$, $k-3$, and $k-4$. Notice in particular, that there was no need to initialize any values in matrices '`fesc[][][][]`' and '`fcse[][][][]`', since the first segments of the first characters are *always* aligned.

It should be noted that, in fact, WMR imposes variable limits on the number of segments per character (determined during training). For example '`j`' can have 1 or 2 segments, '`w`' can have between 1 and 4 segments, etc. Thus the "real" formulas for `max_K`, `min_dur[]`, $\texttt{max\_char[}k\texttt{]}$, etc. are slightly more complicated and different for each word. We have chosen not to consider this variability in order to simplify our exposition.

For our purposes, we did not enforce the triangle inequality upon the elementary confusion matrices. In fact, it would not be meaningful. Our slice distance is supposed to quantify how likely it is for two words to be confused by our recognizer—the larger the distance between two words, the less likely they could be confused. Thus it is completely normal to have small distance between words like 'i' and 'e', and 'e' and 'f' (since they could be easily confused by our recognizer) while having a large distance between words 'i' and 'f' (since they cannot be so easily confused).

Notice that, in general, the slice distance between words would not satisfy the triangle inequality even if the slice-confusion matrices did. The reason for this is that the segmentation of word $w1$ that gives the optimal slice distance between words $w1$ and $w2$ could be totally different from the segmentation that gives the optimal distance between words $w1$ and $w3$.

4 Slice Distance for Other Recognizers

We have described how to compute the slice distance between two ASCII words for a specific recognizer—the WMR. It is clear that this method can be successfully used for almost any segmentation-based recognizer. Some modifications may, however, be needed to account for different ways of computing "matching scores". For example, many recognizers instead of a distance between a character and an image segment use some kind of probability measure. Then the final matching score is the probability (confidence) of recognition. For such recognizers, our algorithm could easily be modified to output the probability of confusing two words—simply multiply the elementary probabilities of confusing character slices (instead of adding the elementary distances) and then maximize (instead of minimizing) the total probability over all possible slice-matchings.

5 Conclusion

We proposed a new method for computing recognizer-dependent distance between ASCII words. Such distances can be used in post-processing of recognition results, for selecting a recognizer that can best distinguish among words from a particular lexicon, or for a combination of recognition results. Several papers describing the use of the slice distance in the above tasks are currently in preparation.

References

[1] G. Kim and V. Govindaraju. A lexicon driven approach to handwritten word recognition for real-time applications. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(4):366–379, Apr. 1997.

[2] G. Seni, V. Kripasundar, and R. K. Srihari. Generalizing edit distance to incorporate domain information: Handwritten text recognition as a case study. *Pattern Recognition*, 29(3):405–414, 1996.

[3] P. Slavík and V. Govindaraju. Use of lexicon density in evaluating word recognizers. In *First International Workshop on Multiple Classifier Systems*, Cagliari, Italy, June 2000.

[4] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, Jan. 1974.