IDIAP COMMUNICATION

Correcting Confusion Matrices for Phone Recognizers

Andrew Lovitt [a] [b]

IDIAP–Com 07-03

March 14, 2007

[a] University of Illinois at Urbana-Champaign, Urbana, Il, USA
[b] IDIAP Research Institute, Martigny, Valais, Switzerland

IDIAP Research Institute **www.idiap.ch**
Rue du Simplon 4 P.O. Box 592 1920 Martigny – Switzerland
Tel: +41 27 721 77 11 Fax: +41 27 721 77 12 Email: info@idiap.ch

Labels		Hypothesis			
		ey	ao	ng	ix
	0	1	2	3	4
ey	1	0	1	2	3
b	2	1	1	2	3
el	3	2	2	2	3
ax	4	3	3	3	3
n	5	4	4	4	4
dh	6	5	5	5	5
ix	7	6	6	6	5

Tab. 1 – This table displays the Levenshtein distance matrix used to align the strings. This structure is inherent to the original Levenshtein algorithm. The red numbered path provides the alignment which is the first hypothesis in table 2. This happens to also be the path traveled by the HTK software. The blue numbers represent equally probable paths not taken.

1 Introduction

The use of the Levenshtein algorithm is used heavily in speech recognition to determine which hypothesis string is similar to the actual target. There is a problem with this algorithm because it only reports the distance and does not align the two strings. This prevents the output from containing information about the alignment which may shed light on the confusions and errors the recognizer is making.

HTK outputs the alignment in a very simple way by back tracing the Levenshtein matrix. While this provides one possible candidate hypothesis, there is unfortunately a large number of possible hypothesis given the minimum distance and the target string. Fortunately in speech recognition we have the ability to leverage the knowledge of the system to find the best possible candidate. It is possible to use this information because the algorithm or set of algorithms that arrive at the candidate string are systematic in the errors they make. Thus the errors and confusions are also to an extent predictable. In this report this predictability is used to create a Levenshtein algorithm which will provide the best candidate string based on this previous knowledge and the minimum distance. This work is primarily concerned with aligning the output of a phoneme recognizer however this algorithm will work for any string of symbols.

2 Basic Levenshtein algorithm

In order to properly align the errors in recognition and understand the confusions the phoneme recognizer is making, the Levenshtein distance calculator requires adjustment. The base line Levenshtein algorithm only outputs the distance between the two input strings, which is the sum of the insertions, deletions, and substitutions. This assumes a uniform insertion, deletion, and substitution probability for all possible phones. This assumption is false for speech recognition systems.

An adjusted Levenshtein algorithm is devised which relies on the original Levenshtein algorithm. The Levenshtein algorithm is adjusted so the matrix used to calculate the distance is back-traced to align the two strings. However a problem arises when multiple strings will have the same distance. In this case most implementations of an adjusted Levenshtein algorithm will assume some arbitrary alignment which is not representative of the actual confusions in the strings. An example of the Levenshtein matrix is seen in table 1.

This table shows the distance aligning the string in various ways. This data is slightly cryptic and the colored numbers are used as a guide. The red numbers show the path the HTK algorithm will backtrace to find the alignment of the strings. The blue numbers represent all the nodes in paths

Reference String	Hypothesis String
ey b el ax n dh ix	ey ao ng ix
	ey _ _ _ ao ng ix
	ey _ _ ao _ ng ix
	ey _ ao _ _ ng ix
	ey ao _ _ _ ng ix
	ey _ _ ao ng _ ix
	ey _ ao _ ng _ ix
	ey ao _ _ ng _ ix
	ey _ ao ng _ _ ix
	ey ao _ ng _ _ ix
	ey ao ng _ _ _ ix

Tab. 2 – Multiple alignment possibilities from the Levenshtein algorithm. The underscore is used to illustrate insertions and deletions. The second case is data from actual phone recognition on the TIMIT corpus. It turns out the 5th hypothesis is actually the best hypothesis due to the high confusability of 'ao' and 'ax' and 'ng' and 'n'.

that are not chosen by the algorithm and thus not considered. An example of these multiple possible hypothesis are seen in table 2. From this very simple example it is impossible to differentiate which hypothesis string is the actual hypothesis that the recognizer that it was trying to report, or in other words where did the phoneme recognizer cause the deletion, insertion, or substitution.

This causes no problems if the confusion probabilities for the recognizer are actually uniform probabilities for all possible phones for insertions, deletions, and substitutions. The phone recognizer's output on which this analysis is based has significant structure in the substitutions, insertions and deletions and this violates the assumption that the errors are random. In the second example above pulled from actual results we can see that one of the confusions makes more sense (the 5th one). The intuitive knowledge from human confusions leads us to the belief that 'ao' should be confused with 'ax' and not 'n'.

What follows is a discussion of this new algorithm which incorporates these substitution confusion into the alignment to produce 'less noisy' substitution, insertion, and deletion matrices for the experiment. If this is not done properly the consequence is that the substitution confusion matrix will have a lot of misrepresented hits confusions in the confusions matrix and this will cause the insertion and deletion numbers to be misrepresented as well. In the extereme case where a recognizer's only errors are where $\alpha\beta$ are reported as γ and α and γ are highly confused, the original backtracing will have α as an insertion 100% of the time and the substitution matrix will have β and γ has highly confused which is not the actual data.

3 Adjusted Levenshtein algorithm

The flow chart in fig. 1 shows the algorithm's structure and use. The algorithm starts with a list of the target and the recognized hypothesis for each sentence. The algorithm then computes the original Levenshtein distance and backtraces the alignment. Then the substitutions which must exist in any hypothesis are collected and a confusion matrix is made from the errors. This matrix is then used to align target and hypothesis in places where there is many different possible answers. The hypothesis with the most likely substitutions is chosen and then the substitution matrix is recomputed and the result is returned.

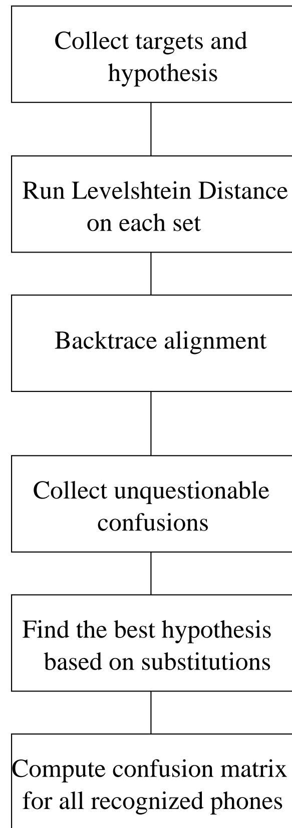

Fig. 1 – Overview of the algorithm used to align the target and the hypothesis in a more systematic way.

3.1 Alignment of substitutions

The first step is to take the back-traced strings from the Levenshtein algorithm and identify all the substitution confusions which are undeniable. This is done by looking for substrings where the beginning and end of the substring are reported as the same phone and the Levenshtein algorithm reports these ends are properly aligned when backtracked. Then if the number of phones between these two points in the hypothesis and the target are the same then all of these phones are substitution errors. (There is an issue with this assumption see sec. 5 for a discussion on this issue.)

This will provide a count matrix for the substitutions which is properly representative of all the substitutions in the training data. We can assert this because it is unreasonable to expect that the substitution errors will be different based on whether or not there is an insertion or deletion error near (or next to) a substitution. This count matrix is not symmeterized before row normalizing which results in the first confusion matrix. Symmeterization is not done because there is no proof that substitutions are symmetric. This substitution confusion matrix is then used to rescore the possible hypothesis from the Levenshtein algorithm.

3.2 Rescoring the Hypothesis

The Levenshtein algorithm is now rerun over the results. All possible hypothesis paths are ranked based the substitution confusion matrix to arrive at the best candidate. At all places where there are multiple possible hypothesis paths from the Levenshtein matrix the algorithm will investigate all the

possible paths and choose the algorithm which minimizes the following metric :

$$V = \sum_{i, i \in \text{Phones in hypothesis}} 1 - CM(T_i, H_i) \tag{1}$$

where CM is the substitution matrix previously computed. H_j and T_i are the ith phones in the hypothesis and target string respectively. In this case the hypothesis and target strings may contain blanks as holders for insertions and deletions. For this equation the diagonal elements of CM are set to 1 (in effect telling the algorithm that if the phone is the same in both strings we have the best placement for those phones). In this equation if the phone is an insertion or deletion the amount added is zero since one of the strings will not have a phone to compare that phone with. Thus this metric has the effect of reinforcing the substitution confusions that were found in the original pass through the alignment data.

After this is run the new alignment will show the most probable hypothesis for the given set of hypothesis strings, the target, and the substitution confusion matrix. From this hypothesis it is possible to more accurately identify the insertion, deletion, and substitution confusions in the results.

4 Results

For these results we will investigate the difference in the substitution confusion matrix and the insertion and deletion arrays from a TIMIT corpus phoneme recognizer. For both sets of analysis we will look at a network that uses a MRASTA and ANN with 800 hidden neurons. The data reported will be from the test section of the corpus which the networks were not trained on. The ordering of the data is done by grouping the phones based on the categories provided in the OGIbet document[1]. For reference the ordering is : iy, ih, eh, ae, ix, ux, ax, ax-h, uw, uh, ah, ao, aa, er, axr, ey, ay, oy, aw, ow, p, t, k, q, b, d, g, m, n, ng, nx, dx, f, th, s, sh, v, dh, z, zh, ch, jh, l, r, y, w, em, en, eng, el, hv, hh. These nets were trained using 54 phones. Long periods of silence was collapsed into one output neuron and short silences (like closures and pauses) are placed in another neuron. This left 52 phones from the TIMIT database and 2 silence classes.

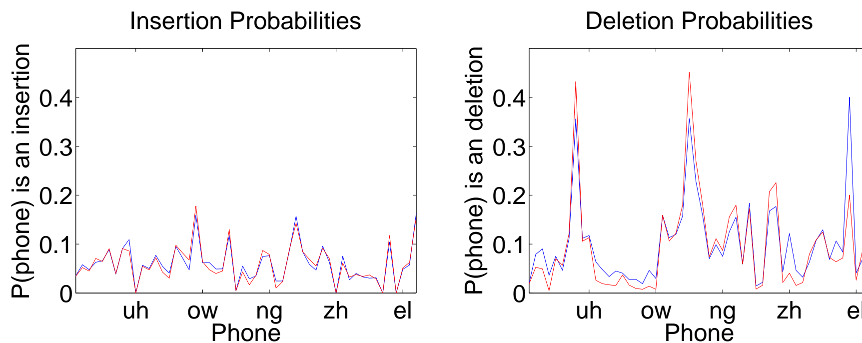

Fig. 2 – These two graphs show the insertion and deletion probabilities (respectively) for the TIMIT data recognized. The original alignment data is in blue and the corrected alignment is in red. The abscissa is in the order specified in the text. The insertion probabilities do not change as much as the deletion probabilities. We see a sharp increase in the aspired vowels (ax-h) and the plosives. These are completely in accordance with the expected results from this analysis.

Fig. 2 shows the difference in the probabilities of insertions and deletions between the original alignment and the corrected alignment. We can see a couple of trends from this data. The first trend

[1]See ftp ://speech.cse.ogi.edu/pub/cse553/ipa_ogi_worldbet_chart.ps.gz

is that the algorithm doesn't significantly improve the insertions and deletions in the data. There is a couple of reasons for this. Most notably there is very few of the errors in the phoneme recognizer are from insertion and deletions, only about 33% of the total. The majority of the errors come from substitutions. We also notice that in general the insertions are unchanged but the deletions show a couple of important differences. The vowels are less likely to be labeled as deleted in the hypothesis. This implies that the recognizer in general doesn't miss vowels which is backed up by observations. Also 'b', 'nx', 'dx', 'v', 'dh', and 'ax-h' show large number of deletion increases when we properly align the data. Also of note is that 'em' is less likely to be deleted. This is most likely due to the high confusions in 'm' and 'em' which means the errors are pushed to substitution confusions where they belong.

There is an inherent problem with displaying the large confusion matrices in reports and so that part will be neglected in favor of explaining the differences. One major difference is that 'b' and 'g' are significantly more confused in general and confusions with 'em' are eliminated. This is the most major difference. In general the major confusions are larger and the 'noise' in the confusion matrix is reduced.

In order to quantify the 'noise' reduction, the entropy is used to compare the row normalized data between the two algorithms. A very high entropy value will imply that the confusions are more uniform and small entropy value will imply that there is only a few very strong confusions. The max value of the entropy is 5.7 bits. Table 3 shows the percentage decrease in the entropy on a per phone basis.

The tangible benefits of the lower entropy is not completely encapsulated in the table however. We can see the entropy of 'er', 'r', and 'axr' all go down by a fair amount but what we don't see is that the phones in fact have much higher confusions with each other which reinforces confusions which are inherent to the data. While 5% sounds like a small amount of 'noise' clean-up, the affect on the confusion matrix is significant.

The final metric used to verify the noise floor is in fact lowered is found by estimating the standard deviation of the noise floor in the confusion matrix. The standard deviation is desired in relation to zero so all elements of the confusion matrix is assumed to have a negative partner of equal height in the negative direction. This will force the mean to be zero. Now we apply a simple EM algorithm to the data and estimate the noise in the confusion matrix. With a halting threshold of .01% the noise floor in from the original Levenshtein algorithm is .29% and the noise floor in the corrected algorithm is .15%. Again this does not completely take into account the importance of the use of this algorithm from a holistic perspective but it does show that that algorithm concentrates the probability in the major confusions while reducing the 'noise' in the confusion matrix. However the noise in the confusion matrix is reduced by 50%.

5 Issues

As previously discussed the results will only work well if the substitutions are in fact systematic to the data. If the confusions are not systematic any analysis of the confusion patterns will not be fruitful in any way. However there are more glaring problems if the results have a much higher phone error probability (PER). If the PER is very low the probability of insertion and deletion rates are very low and this prevents special cases in the data which are undetectable. Thus the lower the insertion and deletion rate is the better the algorithm will perform.

The cases which are undetectable in any case are the cases specified in table 4. These are impossible to detect situations because the Levenshtein distance is in fact larger than the minimal Levenshtein distance. This algorithm only works to find the most probable hypothesis which originates from the minimum Levenshtein distance.

In these cases the phone the recognizer actually reports was an insertion and a substitution. However in these cases it is possible that the 'not caught' cases are the actual results. In this example both the phones in the middle were not recognized. However both the original Levenshtein and the

Phone	H Original	H Adjusted	% Change
iy	0.57	0.56	0.02
ih	1.9	1.8	0.056
eh	2	1.9	0.041
ae	0.83	0.83	0.0014
ix	1.6	1.5	0.036
ux	1.8	1.7	0.064
ax	2	1.9	0.091
ax-h	2	1.9	0.026
uw	1.8	1.5	0.13
uh	2.7	2.5	0.043
ah	2	1.9	0.062
ao	1.4	1.2	0.11
aa	0.71	0.7	0.021
er	1	0.95	0.091
axr	1.6	1.5	0.076
ey	0.94	0.91	0.032
ay	0.75	0.69	0.083
oy	0.92	0.93	0.012
aw	1.7	1.6	0.067
ow	1.2	1.2	0.015
p	1.3	1.3	0.037
t	1.3	1.3	0.043
k	0.93	0.88	0.05
q	1.3	1.2	0.071
b	2.3	2.2	0.024
d	1.9	1.8	0.069
g	1.4	1.3	0.071
m	1.2	1.2	0.031
n	1.6	1.4	0.088
ng	1.3	1.2	0.072
nx	1.7	1.6	0.089
dx	0.96	0.93	0.03
f	1.1	1.1	0.038
th	2.1	2	0.065
s	0.4	0.37	0.081
sh	0.49	0.4	0.19
v	2.2	2.1	0.047
dh	1.7	1.6	0.049
z	0.98	0.88	0.11
zh	1.8	2	0.079
ch	1.2	1.2	0.025
jh	1.1	1	0.068
l	1.2	1.2	0.042
r	1.2	1.2	0.039
y	1.4	1.3	0.075
w	1	0.96	0.044
em	1.6	1.5	0.072
en	2.1	1.9	0.13
eng	0.68	0.68	0
el	1.8	1.6	0.1
hv	0.9	0.88	0.025
hh	1.8	1.8	0.016
Average	1.45	1.37	0.05

Tab. 3 – This table shows the decrease in entropy for each phone in the confusion matrix when the adjusted Levenshtein algorithm is used. The average decrease in entropy is about 5%.

	Reference String a b c d e f	Hypothesis String a b x e f
Caught	a b c d e f	a b _ x e f a b x _ e f
Not Caught		a b x _ _ e f a b _ _ x e f

Tab. 4 – This table shows an example of possible hypothesis results which are completely possible but not detectable by any algorithm operating on the phoneme recognizer output.

adjusted Levenshtein algorithms will not find this case because the minimum distance is when x is assumed to be a substitution no matter what. This is unavoidable. Thus the insertion and deletion rate of the system are critical to properly representing the substitution confusion matrix.

6 Conclusions

The new method proposed for utilizing important knowledge of the structure of the errors is presented and verified to create confusion matrices, insertion, and deletion maps which are more accurate and more properly represent the data. This method requires the substitution confusion matrix to have been pregenerated from the training data. The new algorithm permits less 'noise' in the confusion matrix than the original algorithm, which it does by concentrating the confusions in places where the confusions are most likely. This allows for a more correct confusion matrix which will improve conclusions and analysis which are based on the confusion patterns.

7 Code

For access to the matlab files in source code, email the primary author. The original code is seen in the following sections.

7.1 levenschtein_fixed.m

This code runs the actual realignment of the data. The input data is specifically structured and care must be taken to properly pass the information. See the help.

```
function [sub,actual,hypothesis] =
levenschtein_fixed(actual,hypothesis,max_value)
% this implements my new version of the DLD
% Input 1 is a data structure where actual(n).str = target
% input 2 has the same structure however (lenght(actual) == length(hypothesis).
% The strs must be an array of int [1:1:max_value]
% This will output the CM, where the row and col 1:1:max_value is the
% substitution CM and CM(:,max_value+1) is and deletion and
% CM(max_value+1,:) is an insertion.
%
% The original structures are sent back with the strings aligned.

CM = zeros(max_value+1,max_value+1);

for n = 1:1:length(actual)
  [out,ins,del,sub,ostr1,ostr2] =
```

```
adjustedDLD(actual(n).str,hypothesis(n).str,-1,CM);
  ostr1 = [-2 ostr1 -2];
  ostr2 = [-2 ostr2 -2];

  idx = find(ostr1-ostr2 == 0);
  for nn = 1:1:length(idx)-1
    if idx(nn) ~= idx(nn+1)-1
      one = ostr1(idx(nn):idx(nn+1));
      two = ostr2(idx(nn):idx(nn+1));
    end
  end
  bad = 0;
  for m = 2:1:length(one)-1
    if one(m) == -1 || two(m) == -1
      bad = 1;
    end
  end
  if isempty(find(one == -1)) && isempty(find(two == -1))
    for m = 2:1:length(one)-1
      CM(one(m),two(m)) = CM(one(m),two(m)) + 1;
    end
  end
end

for n = 1:1:max_value
  CM(n,:) = (CM(n,:)+.00000000000000001)/sum(CM(n,:)+.00000000001);
end

sub = zeros(size(CM));

for n = 1:1:length(actual)
  [out,ins,del,subx,ostr1,ostr2] =
adjustedDLD(actual(n).str,hypothesis(n).str,-1,CM);
  ostr1(find(ostr1 == -1)) = max_value+1;
  ostr2(find(ostr2 == -1)) = max_value+1;
  for m = 1:1:length(ostr1)
    sub(ostr1(m),ostr2(m)) = sub(ostr1(m),ostr2(m)) + 1;
  end
  actual(n).str = ostr1;
  hypothesis(n).str = ostr2;
end

function [out,ins,del,sub,ostr1,ostr2] = adjustedDLD(str1,str2,garb,CMsub)
% this will calculate the Damerau-Levenshtein-Distance

d = zeros(length(str1)+1,length(str2)+1);
% errors = zeros(length(str1)+1,length(str2)+1,3);

i = 0;
j = 0;
cost = 0;
```

```
for i = 1:1:length(str1)+1
  d(i,1) = i-1;
end

for j = 1:1:length(str2)+1
  d(1,j) = j-1;
end

for i = 1:1:length(str1)
  for j = 1:1:length(str2)
    if str1(i) == str2(j)
      cost = 0;
    else
      cost = 1;
    end
    d(i+1,j+1) = min([d(i,j+1)+1 d(i+1,j)+1 d(i,j)+cost]);
  end
end

out=d(end,end);

ins = 0;
del = 0;
sub = 0;

x = size(d,1);
y = size(d,2);

ostr1 = [];istrplace1=length(str1);
ostr2 = [];istrplace2=length(str2);

while (x > 1 && y > 1)
  if d(x-1,y-1) == d(x,y) && d(x-1,y) >= d(x,y) && d(x,y-1) >= d(x,y)
    x = x-1;
    y = y-1;
    ostr1 = [str1(istrplace1) ostr1];
    ostr2 = [str2(istrplace2) ostr2];
    istrplace1 = istrplace1 - 1;
    istrplace2 = istrplace2 - 1;
  elseif d(x-1,y-1) == d(x,y) - 1
    sub = sub+1;
    x = x-1;
    y = y-1;
    ostr1 = [str1(istrplace1) ostr1];
    ostr2 = [str2(istrplace2) ostr2];
    istrplace1 = istrplace1 - 1;
    istrplace2 = istrplace2 - 1;
  elseif d(x-1,y) == d(x,y) - 1
    del = del+1;
    x = x-1;
    ostr1 = [str1(istrplace1) ostr1];
    ostr2 = [garb ostr2];
```

```
    istrplace1 = istrplace1 - 1;
  elseif d(x,y-1) == d(x,y) - 1
    ins = ins+1;
    y = y-1;
    ostr1 = [garb ostr1];
    ostr2 = [str2(istrplace2) ostr2];
    istrplace2 = istrplace2 - 1;
  else
    disp('hhh');
  end
end

if x > 1
  while x > 1
    ostr2 = [garb ostr2];
    ostr1 = [str1(istrplace1) ostr1];
    istrplace1 = istrplace1 - 1;
    x = x - 1;
  end
elseif y > 1
  while y > 1
    ostr1 = [garb ostr1];
    ostr2 = [str2(istrplace2) ostr2];
    istrplace2 = istrplace2 - 1;
    y = y - 1;
  end
end

% up to here is the original code for the DLD

ostr2 = [-2 ostr2 -2];
ostr1 = [-2 ostr1 -2];

[zeroidx] = find(ostr2-ostr1 == 0);

for n = 1:1:length(zeroidx)-1
  if zeroidx(n+1)-zeroidx(n) >= 3
    one = find(ostr1(zeroidx(n)+1:zeroidx(n+1)-1) == -1);
    two = find(ostr2(zeroidx(n)+1:zeroidx(n+1)-1) == -1);
    if length(two) > length(one)
      move = ostr2(zeroidx(n)+1:zeroidx(n+1)-1);
      orig = ostr1(zeroidx(n)+1:zeroidx(n+1)-1);
    else
      move = ostr1(zeroidx(n)+1:zeroidx(n+1)-1);
      orig = ostr2(zeroidx(n)+1:zeroidx(n+1)-1);
    end
    move = move(find(move > -1));
    orig = orig(find(orig > -1));
    [a,move,orig]=brute_force(CMsub,move,orig);
    if length(two) > length(one)
      two = move;
      one = orig;
```

```
    else
      two = orig;
      one = move;
    end
    ostr2(zeroidx(n)+1:zeroidx(n+1)-1) = two;
    ostr1(zeroidx(n)+1:zeroidx(n+1)-1) = one;
  end
end

ostr2 = ostr2(2:end-1);
ostr1 = ostr1(2:end-1);
return

function [minval,rone,rtwo] = brute_force(CM,one,two)

if length(one) == 0
  minval = 0;
  rone = two*0-1;
  rtwo = two;
  return
end

minval = 1000000;
rone = one;
rtwo = two;
for n = 1:1:length(two)-length(one)+1
  thisval = (1-CM(one(1),two(n)))+(1-CM(two(n),one(1)));
  [a,b,c] = brute_force(CM,one(2:end),two(n+1:end));
  thisval = thisval + a;
  if minval > thisval
    minval = thisval;
    rone = [-1*ones(1,n-1) one(1) b];%
-1*ones(1,length(two)-(length(b)+1+(n-1)))
    rtwo = two;
  end
end
```

7.2 HTK_to_CM.m

This matlab function is provided to allow for quick analysis of HTK output files. It accepts the path to an HTK file and returns the confusion matrix with the phone mapping.

```
function [CM,phone,actual,hypo] = HTK_to_CM(filename)
% this function takes a file name for a HTK output with the hypothesis and
% the actual string in the file. This will then read the data and perform
% the corrected Levenshtein distance on the data. The algorithm then
% returns the confusion matrix and a structure which contains the mapping
% of the phones to the idx in the confusion matrix.
%
% The confusion matrix's rows are the phone IN THE ACTUAL string. The
% columns represent the phone aligned with it in the HYPOTHESIS.
```

```
%
% filename = path to the HTK output
%
% return:
% CM = confusion matrix
% phones = mapping from the phones to the CM. (Phone(i).phone is a string
% for row or column i.
%
% The CM row and CM col which are length(phones)+1 is the insertions (row)
% or deletions (col).
%
% This will also return the aligned hypothesis and label strings for
% review.

fid = fopen(filename,'r');

if fid == -1
  disp(sprintf('Could not open the file. Exiting.....'));
  return
end

phone(1).phone = 'grab';
line = fgetl(fid);
place = 1;
while ischar(line)
  [s,r] = strtok(line,': ');
  if strcmp(s,'LAB')
    str = [];
    while length(r)
      [s,r] = strtok(r,': ');
            if length(s) > 0
          idx = -1;
          for n = 1:1:length(phone)
            if strcmp(phone(n).phone,s)
              idx = n;
            end
           end
          if idx == -1
            phone(end+1).phone = s;
            idx = length(phone);
          end
          str = [str idx-1];
        end
        end
    actual(place).str = str;
    line = fgetl(fid);
    str = [];
    [s,r] = strtok(line,': ');
    while length(r)
      [s,r] = strtok(r,': ');
            if length(s) > 0
          idx = -1;
```

```
          for n = 1:1:length(phone)
            if strcmp(phone(n).phone,s)
              idx = n;
            end
          end
          if idx == -1
            phone(end+1).phone = s;
            idx = length(phone);
          end
          str = [str idx-1];
        end
        end
    hypo(place).str = str;
    place = place + 1;
  end
  line = fgetl(fid);
end

phone = phone(2:end);
[CM,actual,hypo] = levenschtein_fixed(actual,hypo,length(phone));

for n = 1:1:size(CM,1)
  CM(n,:) = CM(n,:)/sum(CM(n,:)+.00001);
end
```