The Importance of Being Parsed

Alexander M. Duda
University of Illinois at Urbana-Champaign
Department of Electrical and Computer Engineering
Beckman Institute for Advanced Science and Technology
amduda@illinois.edu

I. Introduction

In this paper[1], we discuss the process and results of completing a Monte Carlo Simulation of Probabilistic Parsing. Furthermore, in the conclusion we offer some reflections and future experimental ideas. The code for the work is included as an appendix. The grammar, $G\left(V_n, V_t, S, R\right)$, used was as follows:

$$V_n = \{S\}$$

$$V_t = \{a, b, c, d\}$$

$$R = \{S \to aSa, S \to bSb, S \to cSc, S \to dSd, S \to \epsilon\}$$

By using the grammar to generate a few sentences, one quickly sees that the language, $\mathcal{L}\left(G\right)$, consists of all even-length (length two or greater) palindromes; examples include aa, bb, $abba$, $abccba$, etc. In Chomsky Normal Form, G is as follows (where ϵ represents the null symbol):

$$\begin{array}{rcl}
S_0 & \longrightarrow & A_1A_2|A_1A_1|B_1B_2|B_1B_1|C_1C_2|C_1C_1|D_1D_2|D_1D_1|\epsilon \\
S & \longrightarrow & A_1A_2|A_1A_1|B_1B_2|B_1B_1|C_1C_2|C_1C_1|D_1D_2|D_1D_1 \\
A_1 & \longrightarrow & a \\
B_1 & \longrightarrow & b \\
C_1 & \longrightarrow & c \\
D_1 & \longrightarrow & d \\
A_2 & \longrightarrow & SA_1 \\
B_2 & \longrightarrow & SB_1 \\
C_2 & \longrightarrow & SC_1 \\
D_2 & \longrightarrow & SD_1
\end{array}$$

Using R we generated a large set $(n \gg 1000)$ of sentences $\{W_i | i = 1, 2, 3, \ldots, n\}$. For each sentence, $W_i = w_{i1}w_{i2}\ldots w_{ik}$, we constructed a cost matrix, $D = \left[d\left(w_i, v_j\right)\right]$, where the cost $d\left(w_i, v_j\right)$ was based on the pdf for a vocabulary word v_j which is assumed to be Gaussian $N\left(x, \mu_i, s\right)$. For illustration, consider the noisy sentence $a'b'b''a''$ and its distance matrix (where for the actual computation each of these words is represented by a four-dimensional vector in euclidean space):

[1]This work was completed as Project 2 for Dr. Stephen E. Levinson's ECE 594: Mathematical Models of Language, Spring 2013.

$$
\begin{array}{c} \\ a' \\ b' \\ b'' \\ a'' \end{array}
\begin{array}{c}
\begin{array}{cccc} a & b & c & d \end{array} \\
\begin{pmatrix}
d_{11} & d_{12} & d_{13} & d_{14} \\
d_{21} & d_{22} & d_{23} & d_{24} \\
d_{31} & d_{32} & d_{33} & d_{34} \\
d_{41} & d_{42} & d_{43} & d_{44}
\end{pmatrix}
\end{array}
$$

We then parsed each sentence and counted errors in both the acoustic ("naive") and parsed recognition of W_i. Furthermore, we accumulated the statistics over the entire set of sentences and examined how the error probability varied with s. Before getting to the results, we will offer a brief explanation of the process used along the way.

II. Process Explanation

Instead of providing a line-by-line explanation of how the code works (which is basically done via comments in the code itself), here we will highlight some of the main ideas, technical decisions, and conceptual interpretations that were required to make progress.

A. Representing a Vocabulary Word

In order to ensure symmetry among the distances between words, we let:

$$
\begin{aligned}
a &= [1,0,0,0]' \\
b &= [0,1,0,0]' \\
c &= [0,0,1,0]' \\
d &= [0,0,0,1]'
\end{aligned}
$$

Noise was introduced in a very natural way, with a Gaussian being centered around each of the coordinates, a value would be chosen to produce the new value for that coordinate. As the s increased, the euclidean distance of the noisy word to the original word would increase (and it could become closer to another of the four canonical words).

B. Sentence Generation

When we initially generated sentences from the grammar, we removed all duplicates. Thus, in order to generate a set of about 4,000 unique sentences, we needed to generate roughly 12,000 at the beginning.

C. Acoustic Decoding

For the "naive" decoding, for each word, its euclidean distance to one of the vocabulary words would be computed; the nearest one would be declared the originating word. For the decoding of each sentence, the number of word errors in decoding would be counted and an average word-error probability, p_e, for the particular sentence would be computed. The p_e was then averaged over all sentences. This provided a helpful metric for measuring the success of the acoustic decoding.

D. Parsed Decoding

For this process, we primarily referenced the following equations:

$$\Phi_{kk}(A) = \min_{\{A \rightarrow v_j\}} \{d_{kj}\} \quad (1)$$

$$\Phi_{k_1 k_2}(A) = \min_{\{A \rightarrow BC\}} \{ \min_{k_1 \leq \tau < k_2} \{\Phi_{k_1 \tau}(B) + \Phi_{\tau+1, k_2}(C)\}\} \quad (2)$$

For our purposes, eq.(1) was used with non-terminal symbols A_1, B_1, C_1, and D_1. Since the rules associated with these symbols all require transitioning to a single terminating symbol ($A \rightarrow v_j$), the *min* is effectively ignored. Thus, conceptually, it makes the (accurate) assertion that the "cost" of claiming that a given non-terminal symbol, A, produced the observed noisy terminal symbol is purely the euclidean distance (d_{kj}) of the kth word in the noisy sentence to the terminating symbol, v_j, that the non-terminating symbol can produce according to the grammar. This means that the $\Phi(A_1)$ will have along its diagonal the first column (that beneath the a) of the distance matrix; the other Φ matrices are defined similarly. The non-diagonal entries are filled with infinities, which makes sense, as this asserts that such rules cannot produce sentences of greater than length one.

For our purposes, eq.(2) was used with non-terminal symbols S, A_2, B_2, C_2, and D_2. Since there are eight different rules for S, each of which require transitioning to non-terminating symbols, we must consider taking the full minimum over the sliding window of values (represented by the τ) and over the different rules. However, since A_2, B_2, C_2, and D_2 each have only one specified rule, we can effectively ignore the first *min* and focus solely on the second. Conceptually, this computation ($\Phi_{k_1 k_2}(A)$) computes the "cost" of claiming that one of the non-terminating symbols was used to produce the sub-sentence starting from word k_1 to word k_2 of the observed noisy sentence; the particular combination that costs the least is selected. By completing this computation iteratively it eventually finds the sequence of rules that the grammar could have used to produce a sentence close to the noisy sentence with the least overall cost (when the cost is zero, it means the grammar could produce the noisy sentence).

It is important to note that in order for the equations to work, we must carefully specify the order of (k_1, k_2) values to use; we begin at the super diagonal (the diagonal just above the main diagonal) move along that diagonal for each of the Φ (always looking back at entries in previous diagonals for the current computation) and once they are all full, we move to the next diagonal. We proceed this way until we reach the top-right corner of the matrices. In the top-right of the $\Phi(S)$ will sit the overall "cost" of claiming that the grammar could have produced the noisy word.

Practically speaking, it is not enough just to compute the Φ matrices, one needs to be able to interpret the results to actually see which string was decoded. There are a number of ways to accomplish the task. One straightforward way is the following: create another set of matrices to store sub-sentences , one for each of the non-terminating symbols for a total of nine. We fill the diagonal with a for A_1, b for B_1, c for C_1, and d for D_1; on the off-diagonal we store nothing. In the sub-sentence matrices for the other non-terminating symbols we store nothing to begin. At each step, (as we move along the diagonals of the Φ matrices) when a particular pair of non-terminating symbols is chosen we look at the position in their respective sub-sentence matrices (in the same coordinates as the values just chosen for the Φ matrices), find the sub-sentences stored in each, and concatenate them in the order given by the sum above. This process is carried out along the way and eventually the decoded sentence will be stored in the top-right entry of the sub-sentence matrix for S, the same place where the overall cost for claiming S could produce such a string is stored in $\Phi(S)$.

III. Results

The naive approach assumed that the only way to determine the "correct" word would be to find the nearest word. As a result, a sentence was viewed as a sequence of determining nearest words. However, this completely ignored the structure of the language that generated the sentences! Therefore, in general, the decoded sentences were not even valid sentences in the language $\mathcal{L}(G)$.

However, by taking into account the structure, the grammar, the syntax, one realizes that certain combinations are more likely (not to mention well-formed). More specifically, by recognizing the hierarchy of a sentence (and recognizing the constraints imposed by the generating grammar), one can make a better decision when faced with noise. Considering the following equation from Levinson p.155 is helpful:

$$\frac{H(v|\tilde{v})}{log_2|V_T|} - \frac{1}{H(\mathcal{L}(G))} \leq p_e \tag{3}$$

Supposing v is the transmitted word and $\tilde{v}$ is the decoded word, the equivocation will be $H(v|\tilde{v})$. Using a naive (acoustic) decoder that does not take into account the grammar is equivalent to using an equivocation term $H_A(v|\tilde{v})$. Using the parsing decoder (which takes into account the grammar) is equivalent to using an equivocation term $H_P(v|\tilde{v})$. Clearly, $H_A(v|\tilde{v}) \geq H_P(v|\tilde{v})$ (since the acoustic estimate will have higher entropy than the parsed), which would mean that the acoustic decoder would have a higher lower-bound for the word-error probability, p_e. Therefore, for a fixed noise σ, the word-error probability will be larger for the acoustic method than the parsed. We observe this trend in our simulations, as shown in the figures below.

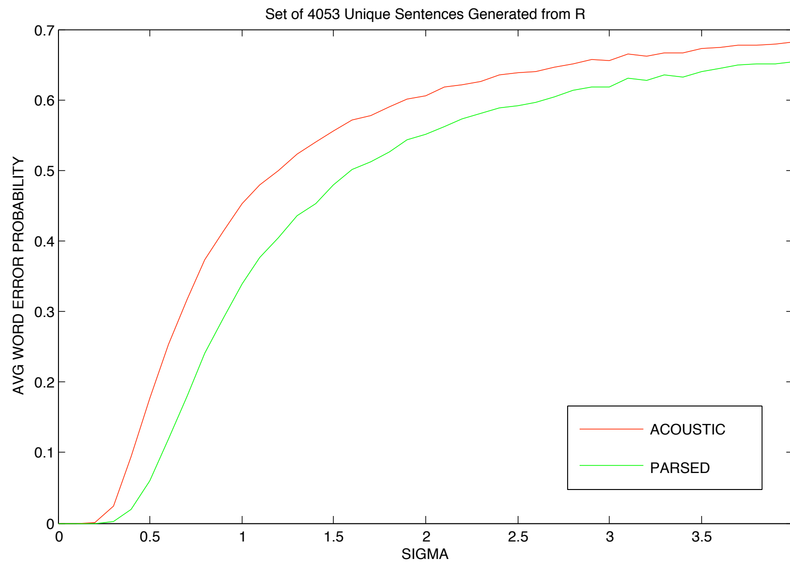

Fig. 1. The figure shows the average word-error probability (vertical axis) in a sentence as a function of the standard deviation ($0 \leq \sigma \leq 4$) with resolution 0.1 of the Gaussian noise that corrupts the original words (horizontal axis). 4053 unique sentence generated from R were used.

Figure 1 displays the results for an experiment run with a set of 4053 unique words generated by the grammar and for standard deviation of the noise $0 \leq \sigma \leq 4$, where steps were taken for σ with resolution of 0.1. Clearly, for $\sigma < 0.2$, the two methods perform identically–they both perfectly decode the noisy

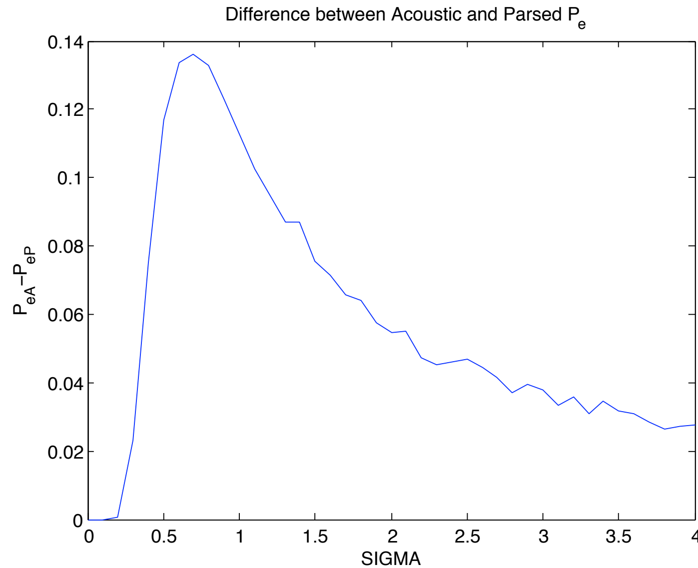

Fig. 2. The figure shows the difference in acoustic $p_{e,A}$ and parsed $p_{e,P}$ as a function of σ when 4053 unique sentences (generated from R) were used with $(0 \leq \sigma \leq 4)$ and resolution 0.1.

sentence. This is to be expected as the parser will never do *worse* than the acoustic. However, as the noise is turned up, we begin to see how the parsed outperforms the acoustic; the maximum outperformance occurs for $\sigma = 0.7$ and results in a difference $(p_{eA} - p_{eP})$ of 0.1362, as shown in Figure 2. As the noise is turned up more, the difference decreases until appearing to slow to a level of around .0277. Judging by the slope of the curves in both plots, it appears that if the noise was allowed to increase arbitrarily high, the acoustic and parsed curves could become arbitrarily close, seemingly approaching a value under 0.7 (the error probability for a uniform source with the same discrete alphabet would be 0.75; considering the entropy should be lower, in our case, it makes sense that the error probability is also lower). However, such results are not particularly relevant as noise beyond even a fairly small level results in such high error probabilities that it renders both methods useless. The main point to be taken from the plots is that the ordering of eq. (3) is preserved for a fixed σ, as explained in Levinson p.156 [2].

IV. Conclusion

The numerical results seemed to match well with previous experiments, intuition, and theory. However, before concluding this lesson on "the importance of being parsed", it is important to acknowledge one of the drawbacks. Unfortunately, the parsing method will likely not do very well (and will likely do worse than even the naive method) if presented with a sentence that was generated by another grammar (where the performance will scale with the degree to which the grammar differs). In this sense, the method seems fairly "over-trained" and brittle. Practically speaking, this may not really be a concern. For many applications, like those discussed in Levinson et. al, sometimes the best answer is to just have the user of a system provide input to help customize the constraints for the given user (and improve the parsing)[1].

The author would be enthusiastic about carrying out a set of experiments that could quantify this intuition and determine the extent to which a given parser's performance degrades as a function of the

grammar being progressively different.

Also, as another extension idea that integrates lessons from *both* projects, the author thinks it would be intriguing to: (1) generate text from a grammar; (2) feed said text to an HMM that could learn its structure; (3) somehow integrate the rules learned into a parser; (4) send the original text through noise into the parser; (5) see how the error rate will do as a function of the grammar complexity (which is known to the experimenter). Being able to quantify the probability of word-error as a function of noise and complexity of the inferred grammar would be a fascinating experiment.

REFERENCES

[1] S. Levinson, A. Rosenberg, and J.L. Flanagan. Evaluation of a word recognition system using syntax analysis. In *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP '77.*, volume 2, pages 483–486, 1977.
[2] Stephen E. Levinson. *Mathematical Models for Speech Technology*. Wiley, West Sussex, England, 2005.

0.1. **Primary Program.** Here we provide the code used in the top-level program.

```
%P2
%Alex Duda
clc;
clear all;
%First we generate about 10,000 sentences using R;
%Then we keep only the unique sentences, so
%we have a set of around 4,000 unique sentences.
time_start=rem(now,1)*24
tempN=30000;
i=1;
COUNTER=0;
while i<tempN
    tempW{i}=Sgen([]);
    if size(tempW{i},2)==0
    end
    if size(tempW{i},2)>0
        i=i+1;
    end
end
tempN=size(tempW,2);
W=unique(tempW(1:tempN));
N=size(W,2);
PROB_ERROR=zeros(1,N);
NAIVELY_DECODED_W=cell(1,N);
%Initialize a,b,c,d vectors.
a=[1 0 0 0]';
b=[0 1 0 0]';
c=[0 0 1 0]';
d=[0 0 0 1]';
%Send generated sentences to Snoisy to produce noisy versions.
%First set the standard deviation of the noise, s.
ITER=1;
sMIN=0;
sMAX=5;
sRES=0.1;
ITER_MAX=size(sMIN:sRES:sMAX,2);
AVG_PROB_ERROR_s=zeros(1,ITER_MAX);
NAIVE_AVG_PROB_ERROR_s=zeros(1,ITER_MAX);
for s=sMIN:sRES:sMAX
    NW=cell(1,N);
    for i=1:N
```

```
        NW{i}=Snoisy(W{i},s);
    end
    %Let us compute a naive decoding. Letter-by-letter, we will map to the nearest.
    for i=1:N
        LL=size(NW{i},2);
        if LL>0
            for j=1:LL
                NAIVELY_DECODED_W{i}(j)=NearestWord(NW{i}(:,j));
            end
            %2. Compare the corrected sentence with the actual (W{i}).
            NAIVE_ERRORS=0;
            for rr=1:LL
                if NAIVELY_DECODED_W{i}(rr)==W{i}(rr)

                else
                    NAIVE_ERRORS=NAIVE_ERRORS+1;
                end
            end

            NAIVE_PROB_ERROR(i)=NAIVE_ERRORS/LL;
        end
    end
    %Compute average error across words for specific s value AVG_ERROR(ITER)=sum(ERROR)/N.
    NAIVE_AVG_PROB_ERROR_s(ITER)=sum(NAIVE_PROB_ERROR)/N;
    %Compute the Cost Matrix.
    D=cell(1,N);
    for i=1:N
        D{i}=NWv(NW{i});
    end
    %Initialize the PHI Matrices for each D.
    PHI_init=cell(1,N);
    for i=1:N
        PHI_init{i}=PHI_INIT(D{i});
        PHI_initS{i}=Si(D{i});
    end
    %Set PHI (and PHI_S) to be equal to the initialized version.
    %This will be updated through the algorithm.
    PHI=PHI_init;
    PHI_S=PHI_initS;
    %For reference, the indices are as follows:
    %A1=1, B1=2, C1=3, D1=4, S=5, A2=6, B2=7, C2=8, D2=9
    %Must fill in entries of S,A2,B2,C2, D2 diagonal-by-diagonal.

    %Loop over the k1,k2 so we go across the superdiagonal of each, and move to
    %the next higher diagonal. Code below should fill up all the matrices.
```

```
for i=1:N
    L=size(W{i},2);
    if L>0
        %Determine which diagonal
        for j=2:L
            %Determine k1,k2 pairs.
            K1=1:1:L-j+1;
            K2=j:1:L;
            %Move along diagonal
            for d=1:L-j+1
                %%STORE THE STRINGS ALONG THE WAY.
                %%CONCATENATE THE BEST SUBSTRINGS AT EACH MINIMIZING ADDITION.

                %For a given k1 k2, compute all the PHIs:
                %Compute PHI_S for ith D
                SR=zeros(1,8);
                StC=cell(1,8);
                [StC{1},SR(1)]=min_sum_PHI_pair(K1(d), K2(d), 1, 6, PHI{i},PHI_S{i});
                [StC{2},SR(2)]=min_sum_PHI_pair(K1(d), K2(d), 1, 1, PHI{i},PHI_S{i});
                [StC{3},SR(3)]=min_sum_PHI_pair(K1(d), K2(d), 2, 7, PHI{i},PHI_S{i});
                [StC{4},SR(4)]=min_sum_PHI_pair(K1(d), K2(d), 2, 2, PHI{i},PHI_S{i});
                [StC{5},SR(5)]=min_sum_PHI_pair(K1(d), K2(d), 3, 8, PHI{i},PHI_S{i});
                [StC{6},SR(6)]=min_sum_PHI_pair(K1(d), K2(d), 3, 3, PHI{i},PHI_S{i});
                [StC{7},SR(7)]=min_sum_PHI_pair(K1(d), K2(d), 4, 9, PHI{i},PHI_S{i});
                [StC{8},SR(8)]=min_sum_PHI_pair(K1(d), K2(d), 4, 4, PHI{i},PHI_S{i});

                [PHI{i}{5}(K1(d),K2(d)),whichS]=min(SR);
                PHI_S{i}{5}{K1(d),K2(d)}= StC{whichS};

                %Compute PHI_A2 and PHI_S_A2 for ith D
                [PHI_S{i}{6}{K1(d),K2(d)},PHI{i}{6}(K1(d),K2(d))]=min_sum_PHI_pair(K1(d), K2(d), 5, 1, PHI{i},PHI_S{i});
                %Compute PHI_B2 for ith D
                [PHI_S{i}{7}{K1(d),K2(d)}, PHI{i}{7}(K1(d),K2(d))]=min_sum_PHI_pair(K1(d), K2(d), 5, 2, PHI{i},PHI_S{i});
                %Compute PHI_C2 for ith D
                [PHI_S{i}{8}{K1(d),K2(d)},PHI{i}{8}(K1(d),K2(d))]=min_sum_PHI_pair(K1(d), K2(d), 5, 3, PHI{i},PHI_S{i});
                %Compute PHI_D2 for ith D
                [PHI_S{i}{9}{K1(d),K2(d)},PHI{i}{9}(K1(d),K2(d))]=min_sum_PHI_pair(K1(d), K2(d), 5, 4, PHI{i},PHI_S{i});
            end
        end
        %1. Now you need to use the computations to say which sentence has been decoded.
        Decoded_W=PHI_S{i}{5}{1,L};
        % 2. Compare the corrected sentence with the actual (W{i}).
        ERRORS=0;
        for r=1:L
            if strcmp(Decoded_W(r),W{i}(r))==0
```

```
                    ERRORS=ERRORS+1;
                end
            end

            if (ERRORS>0)
                COUNTER=COUNTER+1;
                ERRORS;
                W{i};
                Decoded_W;
            end
            PROB_ERROR(i)=ERRORS/L;
            %3. Count the number of errors (0 if correct, 1 if incorrect); ERROR{i}
        end
        W{i};
        NW{i};
        Decoded_W;

        if mod(i,500)==0
            WORD_NUM=i
            TIME_i=rem(now,1)*24
        end
    end
    %Compute average error across words for specific s value AVG_ERROR(ITER)=sum(ERROR)/N.
    AVG_PROB_ERROR_s(ITER)=sum(PROB_ERROR)/N;
    ITER=ITER+1
end
time_end=rem(now,1)*24
plot([sMIN:sRES:sMAX], NAIVE_AVG_PROB_ERROR_s, 'r',[sMIN:sRES:sMAX], AVG_PROB_ERROR_s,'g')
```

0.2. **Sgen** (). .

```
function W = Sgen(x)
%When called,Sgen will run the grammar R and return a sentence.
r=rand(1);
w=x;
if  (r>=0) && (r<0.2)
    W=w;
elseif (r>=0.2)&&(r<0.4)
    w=['a' w 'a'];
    W=Sgen(w);
elseif (r>=0.4)&&(r<0.6)
    w=['b'  w 'b'];
    W=Sgen(w);
elseif (r>=0.6)&&(r<0.8)
```

```
    w=['c' w 'c'];
    W=Sgen(w);
elseif (r>=0.8)&&(r<=1)
    w=['d' w 'd'];
    W=Sgen(w);
end
end
```

0.3. **Snoisy ().** .

```
function [ NS ] = Snoisy( S, std )
%Snoisy accepts a string sentence and outputs a matrix of noisy 4D vecs.
mean=0;
L=size(S,2);
r=mean+std.*randn(16,L);
NS=zeros(4,L);
for i=1:L
    if strcmp(S(i),'a')
        NS(1:4,i)=[1+r(1,i) r(2,i) r(3,i) r(4,i)]';
    elseif strcmp(S(i),'b')
        NS(1:4,i)=[r(5,i) 1+r(6,i) r(7,i) r(8,i)]';
    elseif strcmp(S(i),'c')
        NS(1:4,i)=[r(9,i) r(10,i) 1+r(11,i) r(12,i)]';
    elseif strcmp(S(i),'d')
        NS(1:4,i)=[r(13,i) r(14,i) r(15,i) 1+r(16,i)]';
    end
end
```

0.4. **NearestWord ().** .

```
function [ near_w ] = NearestWord( w )
%NearestWord takes in a noisy letter and naively computes the nearest
%letter.
a=[1 0 0 0]';
b=[0 1 0 0]';
c=[0 0 1 0]';
d=[0 0 0 1]';
[~, I]=min([norm(w-a),norm(w-b),norm(w-c), norm(w-d)]);
if I==1
    near_w='a';
end
if I==2
    near_w='b';
end
```

```
if I==3
    near_w='c';
end
if I==4
    near_w='d';
end
end
```

0.5. **NWv** (). .

```
function [ P ] = NWv(S)
%NWv takes in a noisy sentence and outputs a distance matrix.
K=size(S,2);
P=zeros(K,4);
a=[1 0 0 0]';
b=[0 1 0 0]';
c=[0 0 1 0]';
d=[0 0 0 1]';
r=[a b c d];
for i=1:K
    L=1+(i-1)*4;
    H=i*4;
    for j=1:4
        P(i,j)=norm(S(L:H)'-r(1:4,j));
    end
end
end
```

0.6. **PHI_INIT** (). .

```
function [PHI_init] = PHI_INIT( D )
%PHI_1 accepts a distance matrix and outputs a cell consisting of
%the PHI_A1, PHI_B1, PHI_C1, PHI_D1
PHI_init=cell(1,9);
L=size(D,1);
%Fill each with appropriate INFs.
PHI_A1=Inf(L,L);
PHI_B1=Inf(L,L);
PHI_C1=Inf(L,L);
PHI_D1=Inf(L,L);
PHI_S=zeros(L,L);
PHI_A2=zeros(L,L);
PHI_B2=zeros(L,L);
PHI_C2=zeros(L,L);
```

```
PHI_D2=zeros(L,L);
for i=1:L
    PHI_S(i,i)=inf;
    PHI_A2(i,i)=inf;
    PHI_B2(i,i)=inf;
    PHI_C2(i,i)=inf;
    PHI_D2(i,i)=inf;
end
%Set main diagonal for A1, B1, C1, D1
for i=1:L
    PHI_A1(i,i)=D(i,1);
    PHI_B1(i,i)=D(i,2);
    PHI_C1(i,i)=D(i,3);
    PHI_D1(i,i)=D(i,4);
end
%Write the PHIs into the PHI_init
PHI_init{1}=PHI_A1;
PHI_init{2}=PHI_B1;
PHI_init{3}=PHI_C1;
PHI_init{4}=PHI_D1;
PHI_init{5}=PHI_S;
PHI_init{6}=PHI_A2;
PHI_init{7}=PHI_B2;
PHI_init{8}=PHI_C2;
PHI_init{9}=PHI_D2;
end
```

0.7. **Si** (). .

```
function [Si] = Si(D)
%Si accepts a distance matrix and outputs a cell consisting of
%the strings to be stored in PHI_A1, PHI_B1, PHI_C1, PHI_D1, etc.
Si=cell(1,9);
L=size(D,1);
[A1,B1,C1,D1,S,A2,B2,C2,D2] = deal(cell(L,L), cell(L,L),cell(L,L), cell(L,L),cell(L,L), cell(L,L),cell(L,L), cell(L,L),cell(L,L));
%Set main diagonal for A1, B1, C1, D1 with a,b,c,d, respectively.
for i=1:L
    [A1{i,i},B1{i,i},C1{i,i},D1{i,i}]=deal('a','b','c','d');
end
%Write the PHIs into the PHI_init
[Si{1},Si{2},Si{3},Si{4},Si{5},Si{6},Si{7},Si{8},Si{9}]=deal(A1,B1,C1,D1,S,A2,B2,C2,D2);
end
```

0.8. **min_sum_PHI_pair** (). .

```
function [ best_string, minimum ] = min_sum_PHI_pair( k1, k2, NT1, NT2, PHI,PHI_S)
%min_sum_PHI_pair accepts a set of indices in the PHI matrix (k1, k2) for a particular D,
%as well as a set of indices indicating which pair of Non-Terminal Symbols should be used (NT1, NT2).
%It then generates a set of numbers and outputs the minimum, as well as the best string,
%a concatenation of the best strings from the combination.
%For reference, the indices are as follows: A1=1, B1=2, C1=3, D1=4, S=5, A2=6, B2=7, C2=8, D2=9
for tau=k1:(k2-1)
    M(tau-k1+1)=PHI{NT1}(k1,tau)+PHI{NT2}(tau+1,k2);
end
[minimum, I]=min(M);
tau_best=I+k1-1;
best_string=[PHI_S{NT1}{k1,tau_best} PHI_S{NT2}{tau_best+1,k2}];
end
```