Word Rejection for a Literacy Tutor

by

Michael K. McCandless

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Bachelor of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1992

Author .....
Department of Electrical Engineering and Computer Science
May 18, 1992

Certified by .....
Michael Phillips
Research Scientist
Thesis Supervisor

Accepted by .....
Leonard A. Gould
Chairman, Departmental Committee on Undergraduate Theses

Word Rejection for a Literacy Tutor

by

Michael K. McCandless

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 1992, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Computer Science and Engineering

Abstract

In this thesis, several strategies for word rejection in a literacy tutor (an application to help people learn how to read) are compared. Because we have no data collected of people speaking to such a device, data had to be adapted from an existing speech database to simulate the environment for rejection. Also tested are various criteria for rejection, specifically: direct word score, two normalized forms of the word score, and scores of the individual segments of each word. Each algorithm is tested and compared for false rejection and acceptance (of correct and incorrect tokens, respectively), and in addition for alignment errors of correct tokens.

Thesis Supervisor: Michael Phillips
Title: Research Scientist

Acknowledgments

I would like to thank Michael Phillips for his guidance and patience through the completion of this thesis, Dave Shoemaker for abundant video-game support, and Apple Computer for making this research possible. Most of all, I wish to thank all four of my parents, who have made this research possible in more ways than one.

Contents

List of Figures

Chapter 1

Introduction

1.1 Literacy Tutor

The Spoken Language Systems group at MIT's Laboratory for Computer Science has been developing a system that helps people learn to read — a literacy tutor. It presents text on a screen for the user to read, and then listens to what the user says. As the user reads the words, the computer decides if they were prounounced well enough, highlighting the words to indicate so. If the user has trouble with a word, the computer gives him or her guidance by pronouncing the word, either using a text-to-speech synthesizer, or by playing pre-recorded examples. Hopefully, through this interactive feedback users could quickly improve their reading skills.

Such a tool would have many advantages. Children could learn to read without taking parent's time. Illiterate adults may feel less ashamed and be more willing to work with a computer, than with a person. Also, it is likely that in the long run it would cost less to teach people to read with computers than with adults. It also seems that existing speech recognition technology, which is designed to recognize one of many word sequences that a user can say, should be readily adaptable to this task because it is known exactly what the user should say. The challenge is thus to verify that they said what they were supposed to, and that they said it well enough.

1.1.1 Interface Issues

A number of interface issues quickly surface when developing this tool. For example, a child who is learning to read will often pronounce a new word slowly, syllable by syllable. It would be excellent if the literacy tutor could then highlight each syllable the child pronounces, and allow words to be pronounced in this way, perhaps emphasizing the exact syllable where the user went wrong when pronouncing the word. Another concern is when to help the user — does the literacy tutor automatically detect he or she is having trouble by long pauses, repeated failed attempts at the word, or should there be some mechanism to ask for help, like a button to click or a keyword to say (eg, "help!")? It also seems reasonable to allow the user to pause, and then start again not where they left off, but a few words prior, or even back at the start of the sentence. The literacy tutor should expect this and handle it gracefully. Assuming the speech recognition system runs in real time, the literacy tutor should acknowledge the correct pronunciation of a word shortly after the word is pronounced; there should not be too much delay.

Thus, while the heart of the literacy tutor is the algorithm to accept or reject the words, there are many other important issues relating to the interface and operating in real time, which may impose constraints on the algorithms. This thesis focuses only on rejection algorithms and deals with real time constraints only when they are relevant and have an effect on the choice of rejection algorithm.

Chapter 2

Experiment Background

2.1 SUMMIT Speech Recognition System

The SUMMIT speech recognition system [4], currently being developed in MIT's Spoken Language Systems group, was used as the testbed for the rejection algorithms. Each of the algorithms analyzed is a small modification of the present SUMMIT system.

SUMMIT is a segment based speech recognition system, consisting of roughly three phases: acoustic analysis, classification, and lexical access. During acoustic analysis, three attributes are computed: the pitch of the speech, the output of an auditory model [3], and energy measurements in five bands. The auditory model produces both a mean rate response and a synchrony response, each consisting of vectors of 40 coefficients (the outputs of each of the channels), at a frame rate of 200 per second. Next, the speech is segmented, in a hierarchical manner, into likely phonetic units. This stage uses the auditory mean-rate response as input, and produces a dendrogram structure representing possible phonetic time boundaries and the segments between them, with associated probabilities [1].

For each segment a set of acoustic measurements is computed based on the pitch, energy, and auditory outputs of the frames near and within the segment, producing a 39 dimensional vector. Classification is done using a mixture diagonal gaussian model for each phonetic unit, yielding a vector for each segment representing the probability

that this segment is each of the phonetic units. Finally, a viterbi search finds the best alignment of this phonetic network with a lexical network [5].

The lexicon consists of nodes connected by arcs, where each arc represents a phonetic unit, and contains a weight representing the likelihood that this arc is used (unlikely pronunciations tend to a low weight during training). Each word has a set of initial nodes and final nodes; a path through this word must start at one of the initial nodes and end at one of the final nodes. Because words may have different pronunciations, there is in general more than one path through each word. Word-pair constraints dictate which word may follow another in a sentence, and during the viterbi search, these constraints are used to connect the final nodes of each word with the initial nodes of possible subsequent words. In addition to the constraints of the lexical network, the viterbi search allows insertions and deletions of segments, with associated penalties. A deletion is when a segment that was expected was not seen in the speech, and an insertion is an unnacounted for segment of speech in the acoustic network.

Training is done iteratively to optimize the lexical arc weights, models (from the mixture gaussian classifier), and various penalties (for insertion and deletions of speech segments during viterbi alignment). Deletion penalties are optimized for each model, while insertion penalties are optimized as a function of segment length.

2.2 Experimental Setup

2.2.1 Test Data

We know of no data that has been collected of children and adults speaking to a literacy tutor, so data had to be adapted from a different domain. To do this, a subset of the speech data and corresponding word transcriptions from the Airline Travel Information System database (ATIS) is used [2]. Each of the utterances is assumed to represent a user correctly pronouncing the text. To obtain an example of a user incorrectly pronouncing a word (ie, what should be rejected), one of the words in the transcription is replaced by a word chosen at random from the lexicon. Thus, the speech no longer matches the transcription, so the word where they differ ought to be rejected.

This data has a number of problems. First, it does not capture some of the qualities that people learning to read would exhibit. They would sound words out slowly, especially long words, would pause quite a bit between words, and probably speak quite a bit slower. When they have trouble (for example, maybe a child would pronounce 'cat' as 'kate'), what they say will be far more acoustically similar to what they should have said than the above technique is assuming. A random word from the lexicon is chosen as the incorrect word — no effort is made to choose a word that is similar in sound. Finally, it is assumed that the people who spoke these utterances pronounced all the words sufficiently well, or at least close enough to be accepted by the literacy tutor. But if they were not clear in their pronunciations, the rejection algorithm may fail on their words, making the algorithm look worse than it actually is.

The number of sentences used for testing is 483, with a total of 3857 words. The distribution of word lengths for the correct and incorrect tokens is shown in Figures 2-1 and 2-2. The correct words tend to be shorter than the incorrect words because the incorrect words are chosen at random, while the correct words occur according to their apriori distribution.

The version of SUMMIT used for these experiments was trained on 9711 utter-

ances, drawn from both the MADCOW database (released by NIST) and ATIS data collected at MIT [4]. For each of the algorithms, a subset lexicon is computed for each utterance, consisting of only the nodes, arcs, and legal word transitions corresponding to the words in the transcription. During the viterbi search insertions and deletions are not allowed, for two reasons. First, all speech should be accounted for. It does not seem reasonable, when creating a literacy tutor, to allow insertions of random speech, nor deletions of segments that are needed to complete a word. Second, leaving them out makes some of the computation and error analysis somewhat easier. This will affect performance, to some extent, as the lexicon was trained allowing insertions and deletions, but all algorithms should be affected approximately to the same extent.

2.2.2 Evaluation Strategy

The overall setup is shown in Figure 2-3. For the purposes of these experiments, a rejection algorithm is formally defined as taking as input a speech waveform and a word transcription, and producing a variable length vector of word-transition times. The output length is variable because the algorithm may not find a given word (ie, that word was rejected), and the search ends there (outputting the word times up to that word). For these evaluations, there is no attempt to recover — once an algorithm rejects a certain word in the utterance, it does not look for further words.

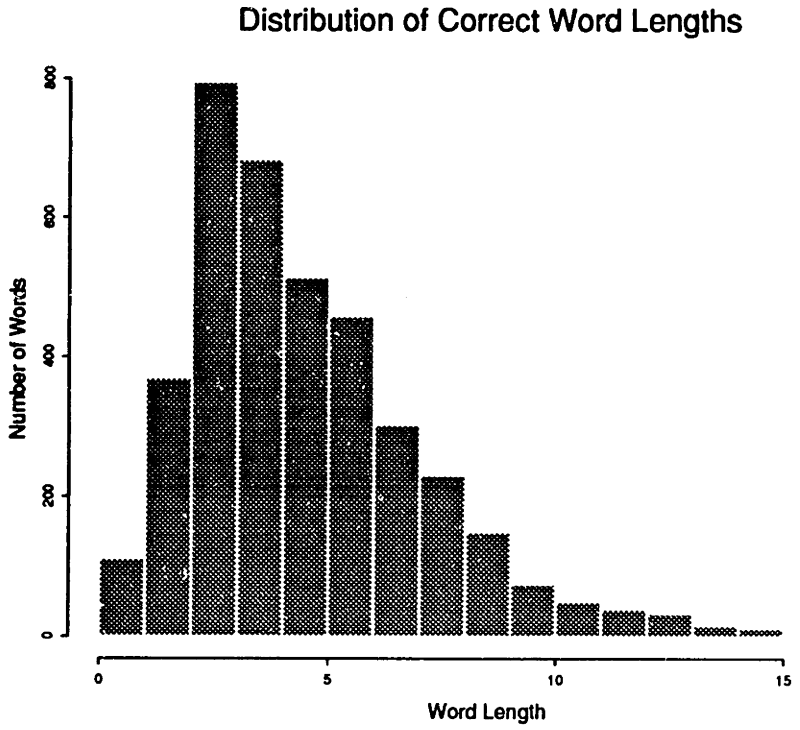

Figure 2-1: Segment lengths of correct words

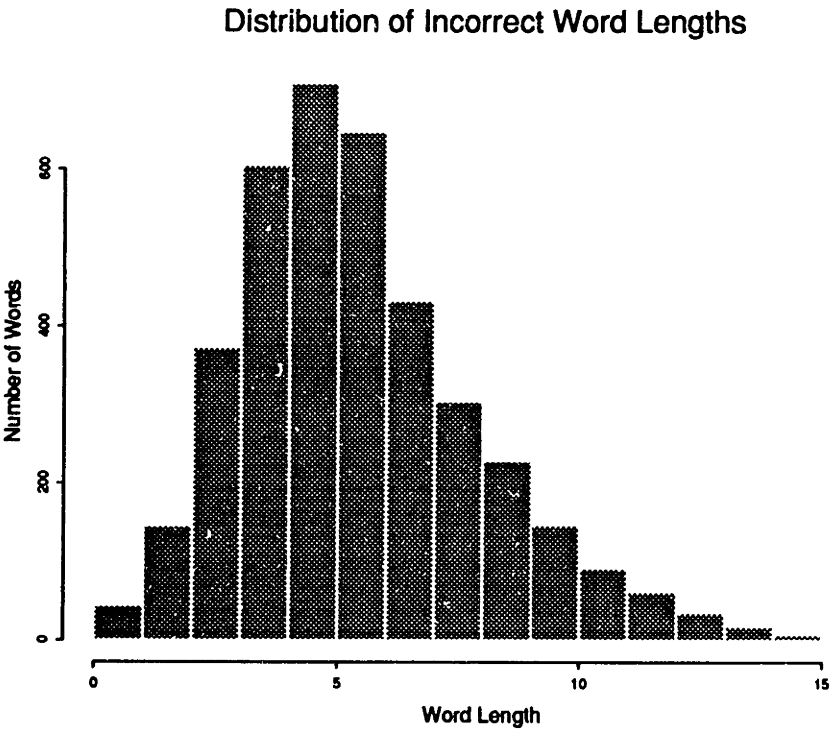

Figure 2-2: Segment lengths of incorrect words

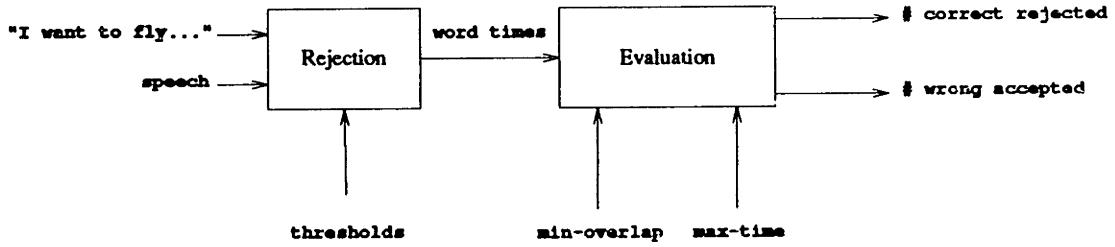

Figure 2-3: Stages of rejection algorithm evaluation

There is an inevitable tradeoff of how many correct tokens are rejected versus how many incorrect tokens are accepted. Where the algorithm falls on this tradeoff is a function of the thresholds chosen. With a low threshold, a high percentage of the incorrect tokens will be accepted, while a high threshold will tend to reject a high percentage of the correct tokens. The optimal threshold needs to be chosen according to the relative costs of each of the errors. While it is a good idea to minimize the number of incorrect tokens accepted, it is also important to avoid frustrating the user with too many rejections. Measuring these errors is a matter of keeping track of four counts for each algorithm:

- **num_right_rej** number of correct words rejected
- **num_right** total correct tokens
- **num_incorrect_acc** number of incorrect words accepted
- **num_incorrect** total incorrect tokens

Thus, first the correct word sequence is processed, consisting of N words. The output is then examined for one of three kinds of errors: 1) a word was rejected (rejection error), 2) the start or end time for a given word was too far from the correct times of that word (temporal error), and 3) the overlap of a word in time with the correct word is too little (overlap error). The correct word times are determined through a forced alignment of the speech with the known word transcription. Overlap errors may occur with very short words whose boundary times may be within the maximum allowed window, but overlap very little with the forced aligned word. Thus, for each word in the sentence, if that word was found to be an error, **num_right_rej**

is incremented by one, and the number of words up to and including the error is added to **num_right**. These two counts are summed across all utterances.

Once an error is found in the correct word sequence, the evaluation stops, increments the two counts, and goes on to the next sentence. For these evaluations, no attempt is made to continue looking for subsequent words, because some of the algorithms tested are not able to resume after a word is rejected. While other algorithms are more suited to resuming in a graceful manner, the lowest common denominator had to be chosen to maintain a consistent evaluation strategy. In the real time system, the user would repeat the rejected word until it is accepted, then go on to the rest of the sentence. Note that an unfortunate side effect of this choice is that the dataset size will shrink as thresholds become more stringent for each algorithm.

The worst errors are those where the word is simply rejected — in the real time system, these are the only errors for correct tokens. Temporal and overlap errors are measured during evaluation because of considerations for the interface of the real time system. The user can click on words he or she has already said, and listen to the portion of speech that the algorithm chose as that word. If the algorithm differs too much in time from what is correct, this will not sound right. Thus, improving temporal and overlap errors is worthwhile as well.

The next measure computed is the number of incorrect words accepted. For every utterance, and for every word in the utterance, a word is chosen at random from the lexicon to replace it. The speech and the modified transcription are processed, and the output is checked to see if the substituted word was accepted or rejected. If it was accepted, both **num_incorrect_acc** and **num_incorrect** are incremented. If a word before the incorrect word was not found (ie, one of the correct words is rejected), neither of the counts are incremented. Finally, if the incorrect word itself was not found (this is the correct behavior), only **num_incorrect** is incremented.

From these counts, the two error measures, false acceptance and false rejection, can be computed: false rejection is the ratio of **num_right_rej** and **num_right**, and false acceptance is the ratio of **num_incorrect_acc** and **num_incorrect**. These two measures provide a consistent means for evaluating and comparing each of the al-

gorithms. Each rejection algorithm has several parameters that control how strict the rejection is, and, in addition, evaluation of the algorithm takes two parameters: MAX_TME (to determine temporal errors) and MIN_OVLP (to determine overlap errors). During evaluation, MAX_TME was fixed at 0.10 seconds, and MIN_OVLP at 75%. All of the algorithms also have a lookahead threshold, which determines how far into the future it should wait until it is sure the current hypothesis for the word ending is correct. This was fixed at 0.20 seconds during evaluation. In Section 4.1, these parameters are changed for each of the algorithms, only to verify that changing them affects each algorithm roughly to the same extent.

In the algorithms that follow, one should be careful in separating three aspects. First, there is the unit level at which the algorithm operates — in these experiments, the word level or the model level. This determines which units to focus on, and which units are required to be "good enough" in some sense. The criterion for "good enough" is the next dimension of the algorithm: how is it decided whether the unit is accepted or rejected. In most of the algorithms, this criterion will simply be that the score is above some fixed threshold, but in some of the pruning experiments this threshold is a function of other variables. Finally, there is the algorithm itself: given the unit level, and given some criterion that each unit must satisfy, how are the word alignments computed?

Chapter 3

Algorithms

3.1 Pseudo Word Spotting (PWS)

The first approach is a pseudo-word-spotting algorithm, named so because it has the advantage of allowing the user to insert random speech before he or she actually pronounces the expected word (for example: 'Um', 'Uh', 'Oh man, I really don't know'). Real users would most likely tend to do this, so this sort of tolerance should be expected from a literacy tutor. The problem, however, is that this is a relaxation of the constraints that this task imposes, namely that words follow each other, exactly, in time. When word spotting is allowed, short words (ie, 'it' or 'the') tend to have a high false alarm rate because they are easily located in random speech.

The algorithm is implemented as follows. For a given sentence, a subset lexicon is computed that contains only the nodes, arcs, and legal word transitions corresponding to words and word transitions in the current sentence. Normal recognition is done, using the subset lexicon. Next, the word locations are computed according to the pseudo-code in Figure 3-1. The outer loop loops through each word in the sentence, and the next loop loops through all time boundaries. At each time boundary, the maximum word score is found by choosing a word-final node, tracing the corresponding path back to where it first entered the word (lexical node and time boundary), and subtracting the start score from the end score. This is done for all word final nodes for this word at each time boundary, and the maximum word score is chosen

as the score at the boundary (bound_max_score).

From bound_max_score, the algorithm derives the end time for a given word by searching for the boundary with the best word score above a certain threshold, starting with the time boundary that ended the previous word (zero at the beginning). Once a possible maximum is found (ie, one above the threshold, and one above any previous maxima), the algorithm continues looking for a certain lookahead into the future. If the maximum word score is not exceeded within this lookahead time, it stops looking and reports that boundary (max_bound) as the end of the word. Note that no consideration is given to how low the overall score for the partial path is -- all that matters is the partial score due to the portion of the path through the current word (hence the word-spotting nature of this algorithm).

The parameters of this algorithm are the word-score-threshold, and the lookahead. In general, the lookahead should be kept as short as possible to avoid unreasonable delay in accepting a word, but long enough to avoid triggering on false alarms. How long it needs to be will actually depend on the threshold: as the threshold decreases, there will be false alarms further from the true peak, so the lookahead will have to be longer.

Figure 3-2 shows the resulting receiver operating characteristic (ROC), computed by sweeping the word-score threshold across a wide range of values. There are two curves — one showing only rejection errors, and the other showing all errors (including temporal and overlap errors as well). The x-axis is the percentage of correct words rejected (false rejection), and the y-axis is the percentage of incorrect words accepted (false acceptance). When the threshold is too high (towards the bottom right of each curve), a high percentage of the correct words are rejected, while low thresholds allow too many incorrect words to be accepted. The graph tends to curve back at low thresholds because false peaks are being spotted far away (greater than the lookahead) from the true peaks, thus causing more temporal errors.

It seems somewhat disturbing that there are a very large number of temporal and overlap errors (around 25%), but one should keep in mind that the temporal parameters are somewhat strict (demanding the word end times be within .10 seconds

```
last_bound = 0
for word in sentence
    bound_max_score = −∞
    max_bound = −∞
    for bound from last_bound to num_bounds
        max_score = −∞
        for node in lexicon.word.final_nodes
            end_score = scores[bound][node]
            start_score = trace_to_word_initial(bound, node)
            if (end_score - start_score > max_score)
                max_score = score

        if (max_score > threshold and max_score > bound_max_score)
            max_bound = bound
            bound_max_score = max_score

        if (max_bound ≠ −∞ and ((times[bound] - times[max_bound]) > lookahead))
            word.end_boundary = max_bound
            last_bound = max_bound
            break
```

Figure 3-1: Pseudo-code to implement PWS

of the forced alignment, and overlap at least 75%). These errors can be explained by both the stringency of these parameters, and by the means used to derive the "correct" times (by forced recognition alignment). Also, most of the overlap errors are very short words, because a slight change in the boundary times of these words introduces a very large change in the overlap. Each of the algorithms examined in this thesis exhibit the same large number of errors, so a more detailed analysis will be postponed until Section 4.1.

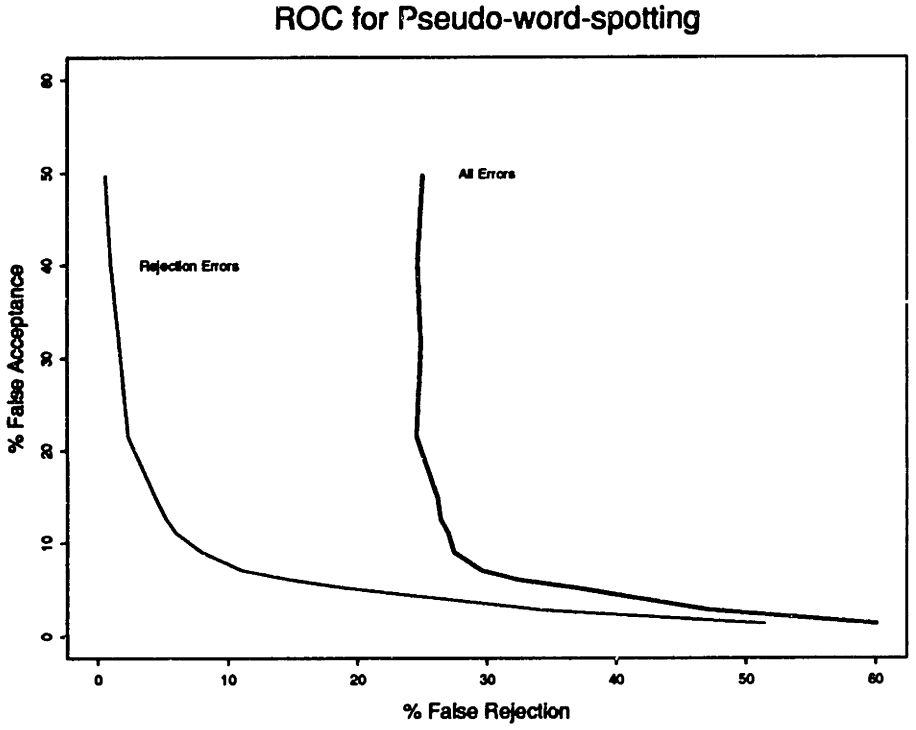

Figure 3-2: Performance of Pseudo-word-spotting

3.2 Max-sum-diff (MSD)

The second algorithm tested is called max-sum-diff (MSD). This was an initial attempt to create an algorithm that takes better advantage of the constraint that words must follow one another in time (ie, it does not allow random speech to separate the expected words). Allowing it to take advantage of this constraint increases the performance slightly over PWS, though the increase is quite a bit less than expected.

The algorithm is very similar to PWS, except for how the word score is derived for each time boundary (see Figure 3-3 for pseudo-code). Instead of looking at just the current word's score, the score of the entire path from the beginning of the sentence to the end of the current word is considered. Each word must contribute more than a certain threshold to the overall path score. Thus, if the best total score up to the end of word j is X, and the threshold is T, then a valid path ending with word $j+1$ must have a net score X+T or more to be accepted, and need not use the best subpath that ended at word j. The best score for the current word at the current boundary, bound_max_score, is then used in the same way as PWS to locate the end of the current word: look for a peak above a threshold, and look ahead to make sure there is no other peak soon thereafter.

The ROC for max-sum-diff is shown in Figure 3-4. Also shown, for comparison, is the ROC for PWS. Again, there are a very large number of temporal and overlap errors. The advantage of MSD is not as substantial as was expected, and there is certainly some question as to whether there is any advantage at all with regard to rejection errors. There does seem to be a slight improvement of alignment (temporal and overlap) errors, as words are forced to be closer to the end of the previous word.

This algorithm is unreasonable in that it does not allow much tolerance for words that cannot attach themselves exactly at the maximum ending time boundary and lexical node for the previous word. Specifically, if the current word attaches somewhere where the score is worse than the maximum for the previous word, it must make up that difference and also add the threshold, before it will be accepted. Thus, PWS was too lenient in its constraints, and MSD is too strict in its constraints.

```
last_bound = 0
max_sum_past = 0
for word in sentence
    bound_max_score = −∞
    max_bound = −∞
    for bound from last_bound to num_bounds
        max_score = −∞
        for node in lexicon.word.final_nodes
            end_score = scores[bound][node]
            if (end_score > max_score)
                max_score = end_score

        if (max_score - max_sum_past > threshold and max_score > bound_max_score)
            max_bound = bound
            bound_max_score = max_score

        if (max_bound ≠ −∞ and ((times[bound] - times[max_bound]) > lookahead))
            word.end_boundary = max_bound
            max_sum_past = bound_max_score
            last_bound = max_bound
            break
```

Figure 3-3: Pseudo-code to implement MSD

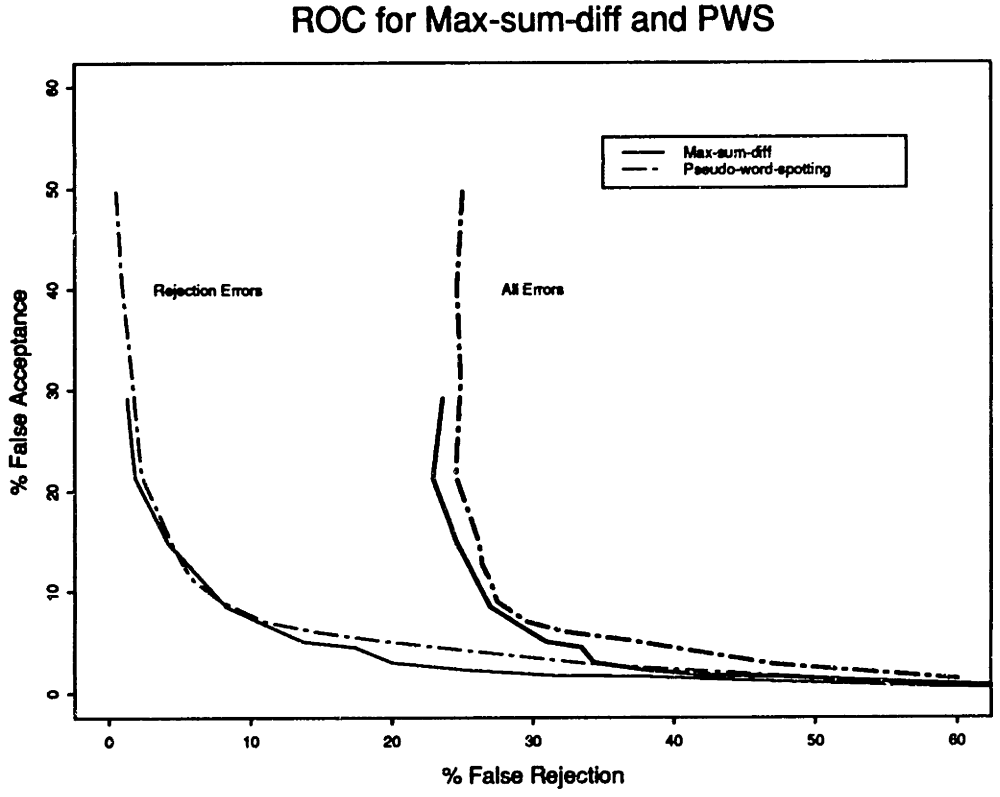

Figure 3-4: Performance of Max-sum-diff and PWS

3.3 Word Pruning (WP)

The third algorithm tested, and overall the one yielding the best performance, is word-pruning (WP). In this algorithm, at every time boundary, those word final lexical nodes that don't represent a sufficient match for the word ending there, according to some criterion, are pruned. Given any criterion for rejection at the word level, this algorithm is optimal in the sense that it will find a path (or partial path), if one exists, that includes only words that satisfy the criterion. The other two algorithms do not do this properly — PWS pays no attention to the path up to the current word, and MSD will fail to find some paths that do satisfy the criterion for each word. If there is such a path, or a partial path up to some word, where each word satisfies the rejection criterion, this algorithm is guaranteed to find it, and otherwise fail.

WP is a relaxation from MSD because MSD was very strict in demanding that the next word score be above a threshold better than the previous one, regardless of how it connected to the end of the current word. In this sense WP is better because it looks at the actual score each word contributes to the overall score, pruning those points where it does not pass the criterion.

It should be emphasized that the choice of this algorithm is orthogonal to the choice of the rejection criterion. Namely, no matter how it is decided whether a given match to a word, or even a model, is good enough, pruning can be used to then search for a path through the acoustic and lexical network such that every element (word or model) in the path satisfies the criterion. Thus, several experiments to test different criteria will be described. The first will use the same criterion as MSD and PWS: a constant threshold for the word score.

The pseudo-code for the algorithm is shown in Figure 3-5. This differs from the previous two algorithms because it alters the viterbi search during recognition. However, it finds the word boundaries in a similar lookahead manner. Within the viterbi search, pruning is done at each time boundary, before updating transitions to following words, thus preventing the next word from attaching to the end of the current word unless it passes the criterion. Using the simple word-score threshold

```
% This is called for each boundary during the viterbi search
prune_word(word, bound, scores, threshold)
    for node in lexicon.word.final_nodes
        end_score = scores[bound][node]
        start_score = trace_to_word_initial(bound, node)
        if (end_score - start_score < threshold)
            scores[bound][node] = −∞


% This is called after recognition to compute word locations
last_bound = 0
for word in sentence
    bound_max_score = −∞
    max_bound = −∞
    for bound from last_bound to num_bounds
        max_score = −∞
        for node in lexicon.word.final_nodes
            if (scores[bound][node] > max_score)
                max_score = score

        if (max_score > −∞ and max_score > bound_max_score)
            max_bound = bound
            bound_max_score = max_score

        if (max_bound ≠ −∞ and ((times[bound] - times[max_bound]) > lookahead))
            word.end_boundary = max_bound
            last_bound = max_bound
            break
```

Figure 3-5: Pseudo-code to implement WP

criterion for rejection, this pruning procedure, for every word final node, traces back to where this word was first entered. If the net gain in score from the initial to final lexical node of the word is above the threshold, the score is left as is, otherwise it is set to $-\infty$.

After the search is done, word start and end times are extracted in a manner very similar to the previous two algorithms: the boundary with the maximum score for the current word is sought, within a certain lookahead. One slight difference is that instead of choosing the word score to maximize, the partial path score, ending at the current word, is maximized. It is not necessary to trace back through the word

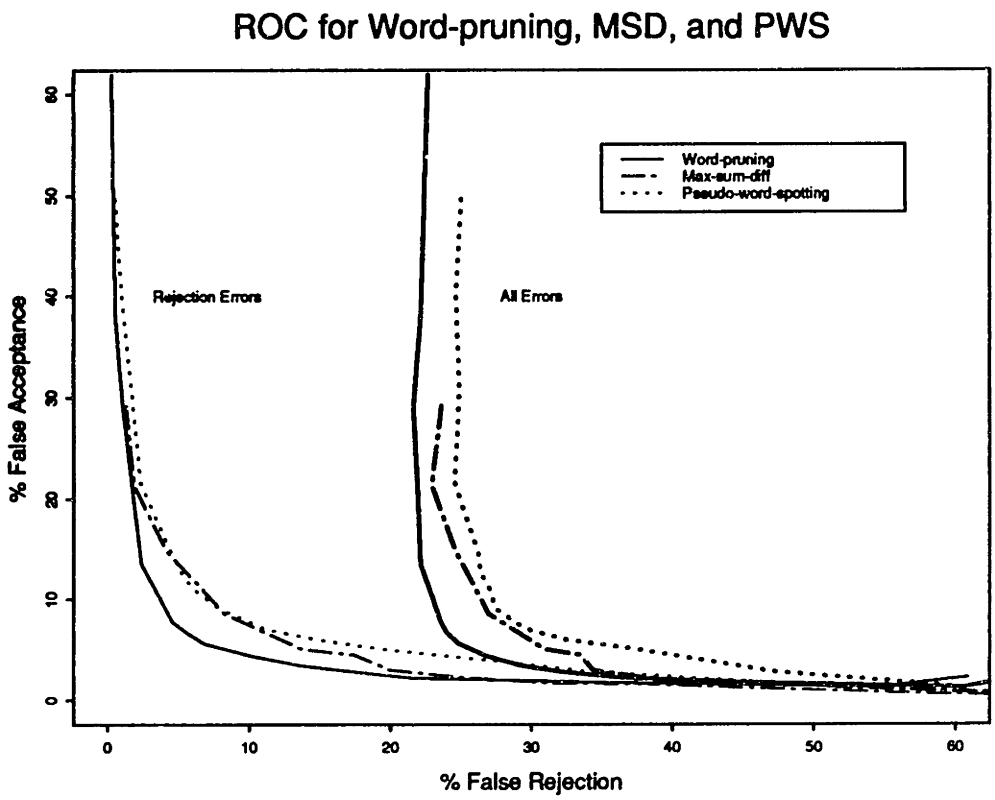

Figure 3-6: Performance of Word-pruning, MSD, and PWS

because if the partial path score ending at the current word is not $-\infty$, the current word's score must be above the threshold because it was not pruned. Also, all prior words' scores must also be above the threshold.

The ROC for this algorithm and criterion, along with those of PWS and MSD, are shown in Figure 3-6. Word pruning offers a substantial increase in performance for both rejection errors and temporal and overlap errors.

3.3.1 A Closer Look

In order to choose a reasonable criterion for rejection, it was necessary to examine the scores of correct and incorrect tokens in closer detail. Figures 3-7 and 3-8 show the distribution of these scores, as a function of word length (in segments). In these graphs, each of the vertical clusters represents words with the same number of segments; within each cluster, the tokens are spread out in the x direction (by $\pm$ 0.2 segments) to make the distribution more visible. The word-score on these graphs is

the pruning level required to reject each of the tokens. Thus, the rejection problem is a matter of separating the two sets of data.

The actual threshold to prune a given word really depends on how the previous word was pruned because this will change where (lexical node and time boundary) the word can start. Thus, deriving these graphs directly from the data would have been extremely time consuming. Instead, a reasonable shortcut was chosen: what really needs to be measured is the best score for the word in the vicinity of where that word was pronounced in the utterance (for the correct tokens). For the incorrect tokens, the best score for the incorrect word is computed, starting near where the previous word ended, and ending anywhere thereafter.

Each sentence with N words contributes N tokens to each of the graphs. First, a forced recognition path is computed to obtain the alignment of the correct words. Then, for each word, the best word-score starting and ending within a fixed time window of the forced alignment, is computed — this yields the points in Figure 3-7. The incorrect word scores are then computed by finding the best score for a word, chosen at random from the lexicon, beginning within a fixed time window of the end of the previous word, and ending anywhere. For the graphs, the allowed time window was 0.3 seconds.

The strategy used to derive the data stands out very clearly in these graphs: the separation for long words is very good. If real data were used instead, the incorrect tokens would, most likely, sound more like the correct word they were replacing, and their scores would be much closer to the correct scores.

3.4 Alternate Rejection Criteria

It is clear from the graph that simply choosing a constant threshold for the word score is not optimal — the scores of correct and incorrect tokens vary substantially with the number of segments in the word. As the number of segments increase, the mean score of correct tokens increases, but the deviation does as well. Thus, some sort of normalization is necessary so that longer words need a higher score to pass. This

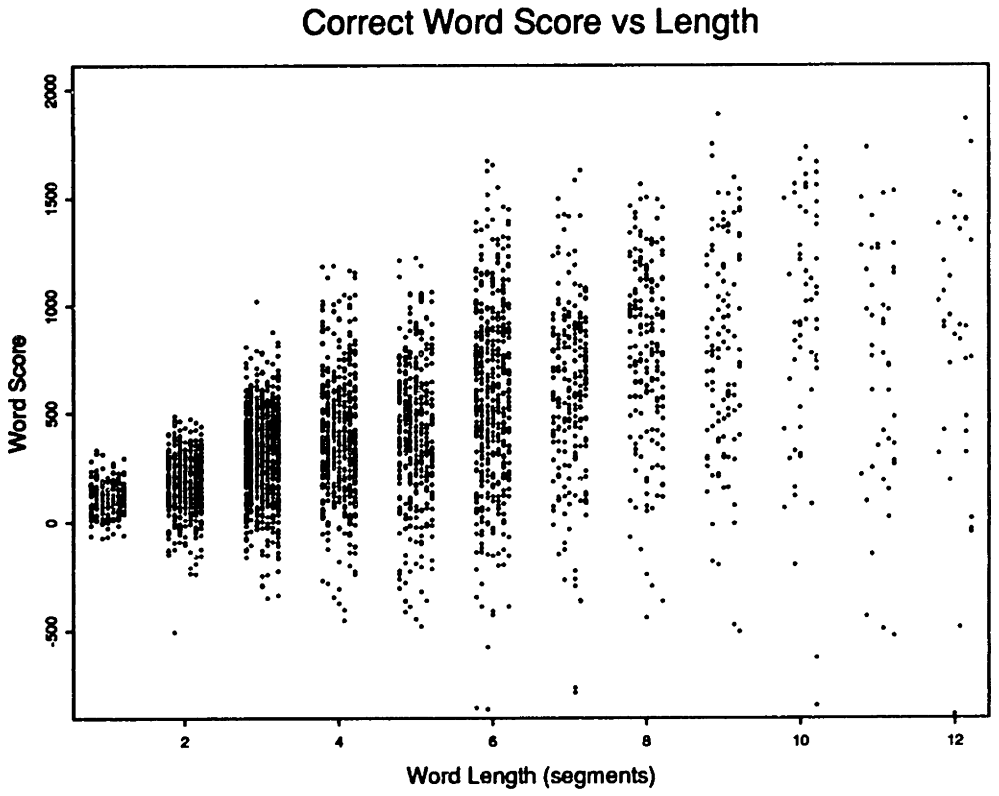

Figure 3-7: Unnormalized correct word scores

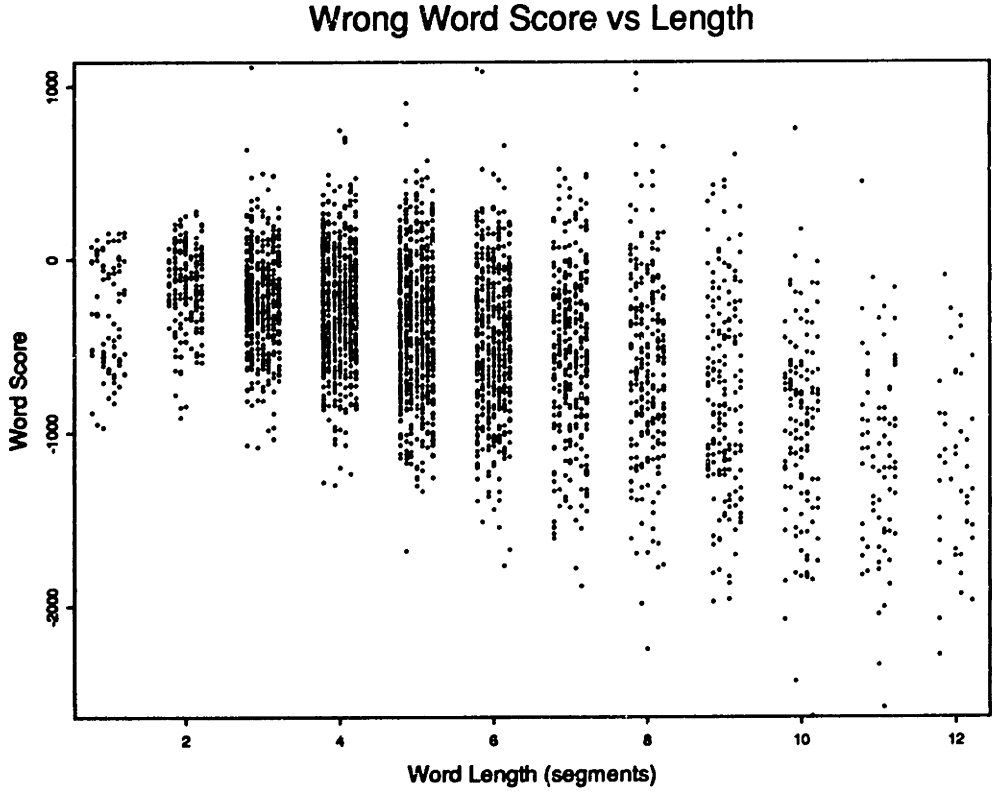

Figure 3-8: Unnormalized incorrect word scores

problem would be far more prevalent with real data because the separation between correct and incorrect tokens would be less.

There are a few effects that should occur due to a simple constant threshold. First, it is clear that the performance is not as good as it could be -- there are better curves to draw (thresholds optimized as a function of word-length) that could do the separation better. The second effect is one of word-spotting for the word following a long word when the threshold is low. A long word will tend to have a very high maximum score, and thus for a large number of time boundaries before and after the maximum-scoring time boundary, the score for the word will remain above the threshold. This will allow the next word to attach itself across a wide band of time boundaries, essentially allowing local word-spotting. This will affect both incorrect and correct tokens. A normalizing scheme should help both of these problems, and hopefully improve performance.

3.4.1 Model Specific Normalizing

The reason that longer words tend to have higher scores is that the models have average scores above zero. So, as an initial attempt to normalize word scores, the mean score for each model was measured by collecting a histogram of the model scores used during forced alignment. Then, during pruning, a word's score is normalized by subtracting off the means of all the models in the word, and then the word score is compared to a fixed threshold.

This did have the effect of moving the means of the correct words to approximately zero across all number of segments, while the deviation remained approximately the same as the unnormalized cases (see Figures 3-9 and 3-10). The incorrect word scores fall even faster, as the number of segments increases. The performance, when tested with the real pruning evaluation, was quite a bit worse. Figure 3-11 compares the unnormalized constant-threshold case with the model-specific normalizing. There is also a tremendous increase in temporal and overlap errors when model-specific normalizing is used.

There seem to be two reasons for the substantial performance loss. First, during

training of the models, the mean score of a given model is allowed to fluctuate to whatever maximizes performance. The means would have centered themselves at zero if that were optimal, but the fact that they did not means that their non-zero means are significant, and relevant to recognition performance. Models that are very sure of themselves, or very different from other models, would have high means to assert themselves during recognition. The second reason is apparent from inspection of the distribution of correct and incorrect word scores for the normalized and unnormalized cases. When the scores are unnormalized, the separation provided by a constant threshold looks far better than the normalized case because, although the mean score for correct tokens increases with word length, so does the deviation. A constant threshold criterion looks like it would be closer to optimal when the word scores are unnormalized than when they are normalized.

3.4.2 Retrospective Word Pruning

One of the problems with a constant threshold is that a very long word will have scores above the threshold across many time boundaries, thus allowing local word spotting for the word following it. Retrospective pruning is an attempt to minimize this effect. For a given sentence, this algorithm chooses a word specific pruning threshold such that the threshold for each word is either a fixed amount less than the maximum for that instance of the word, or a minimum threshold, whichever is greater. Thus, if a given word has a very high score, the threshold for that word will also be high (equal to the maximum word score minus the umbrella threshold), hopefully minimizing the number of time boundaries where the word is not pruned. This algorithm thus takes three thresholds: the cutoff threshold, an umbrella threshold, and the lookahead.

The viterbi search starts assuming the threshold for all words is the cutoff threshold, performing word pruning according to this threshold. Then, as the search progresses, if there is a word whose score is greater than the cutoff plus the umbrella, the guess of what the threshold for that particular word should be is changed to the maximum minus the umbrella, and the search backs up as far as necessary to fix whatever will change. Specifically, it backs up to where this word started having

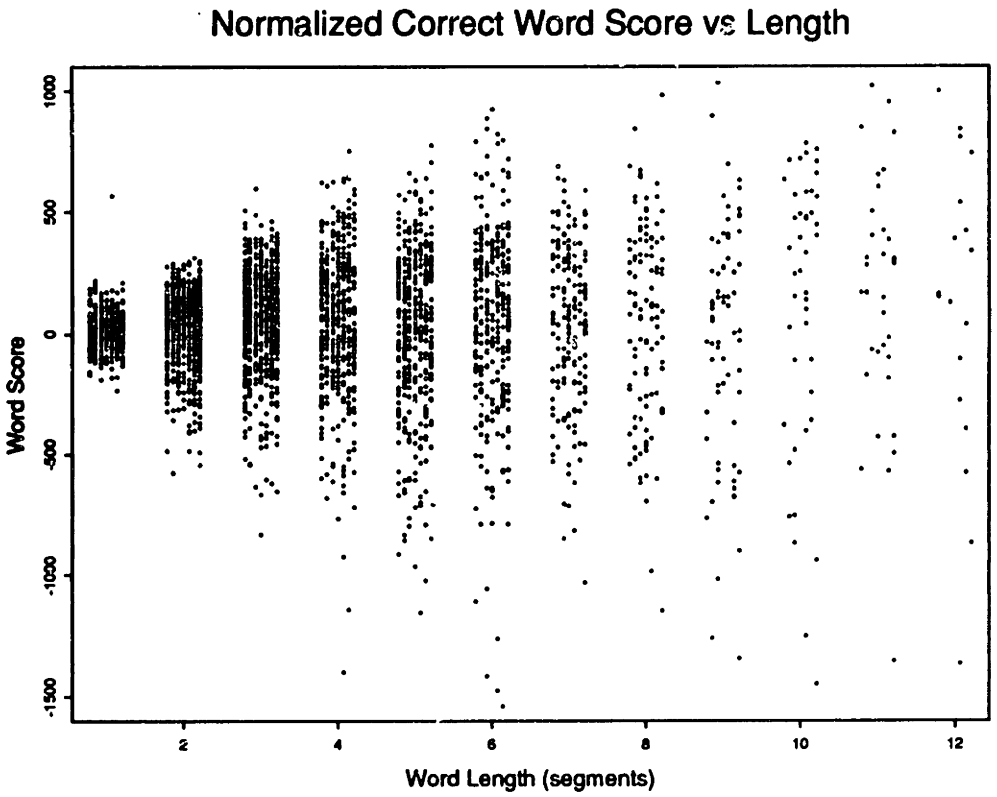

Figure 3-9: Normalized correct word scores

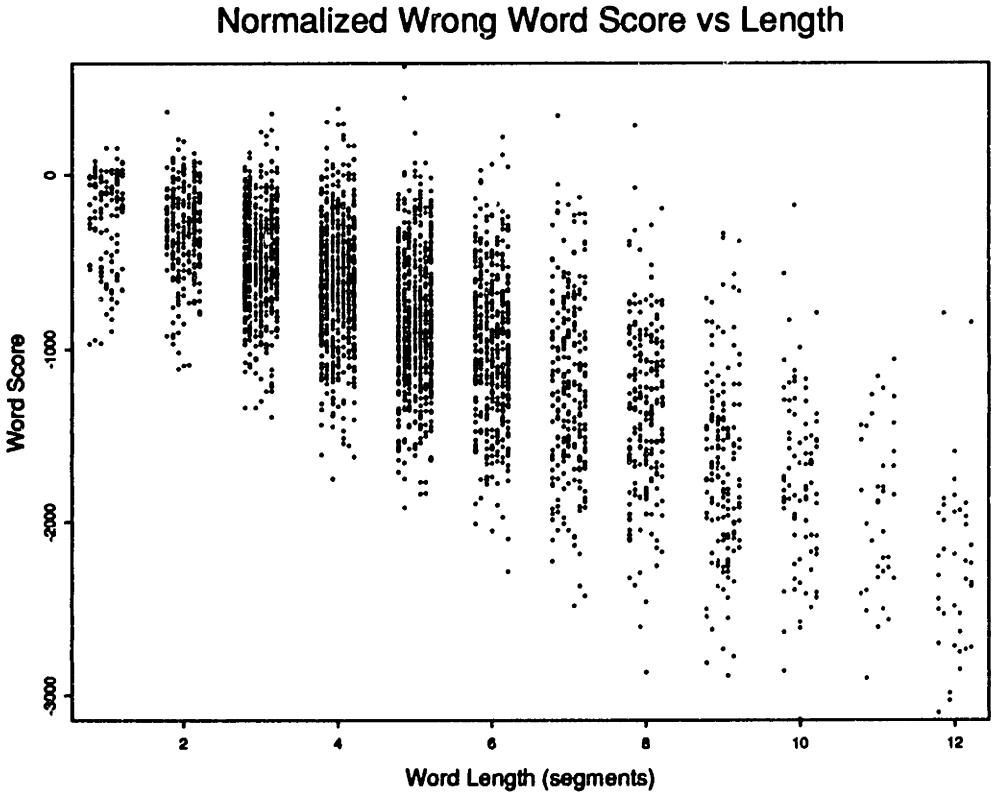

Figure 3-10: Normalized incorrect word scores

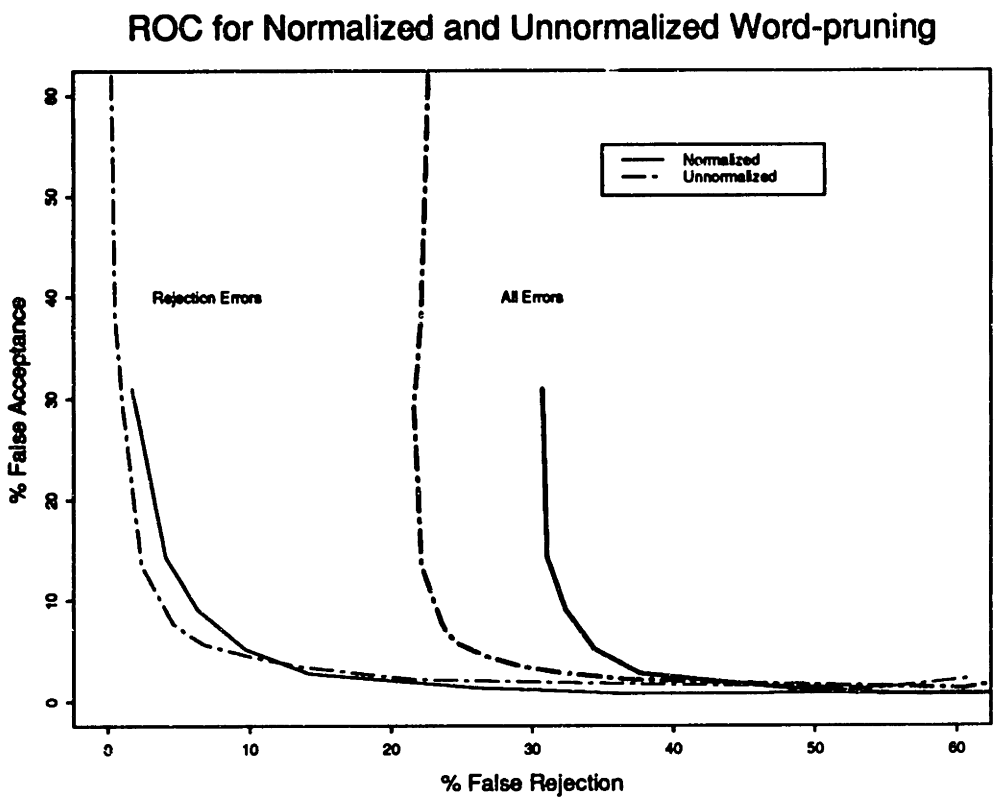

Figure 3-11: Performance of normalized and unnormalized word-pruning

non-pruned scores because some of these will now be pruned. It is often necessary to back up several times, each time changing a guess for the threshold for a given word to a higher value. This process is guaranteed to terminate because, upon each backup the cutoff threshold for some word increases, and word scores are necessarily finite, so eventually, the correct thresholds are determined, and the search stops.

The performance increase due to retro-pruning was slight. Retro-pruning reduced errors for the incorrect tokens, but at the same time hurt false rejections because the correct tokens also took advantage of the implicit word-spotting. When the umbrella threshold is too small, too many correct tokens are rejected; when it is too large, the algorithm approaches normal pruning as word-spotting becomes more prevalent. Figure 3-12 shows the ROC for simple word pruning, and then several additional contours where the pruning threshold remained constant, but the umbrella threshold varied.

Figure 3-13 compares the ROC for retrospective word pruning with an umbrella threshold of 300 with constant threshold word pruning. The two criteria perform

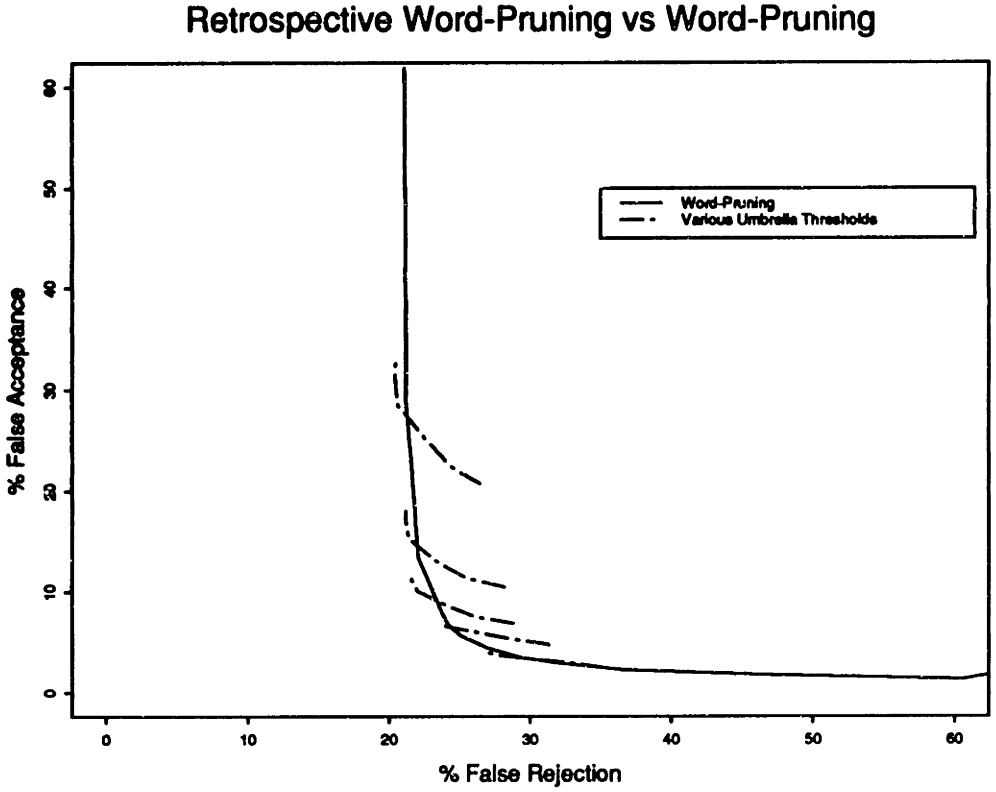

Figure 3-12: Performance of retro word pruning across many umbrella thresholds

roughly the same for rejection errors, but retrospective pruning reduces temporal and overlap errors substantially. This is expected because retrospective pruning reduces the local word spotting that would increase alignment errors.

3.5 Model Level Pruning (MLP)

The final criterion tested was at the model level, using a fixed threshold for each model score. Instead of demanding that each word score be good enough, each model within the word must now have a score above the cutoff threshold. Thus, if the viterbi search with model pruning finds a path through the acoustic network and lexicon, then that path consists of models that were all above the threshold. This would make sense because as a person is pronouncing a word, every sound in that word ought to be correct. By pruning at the word level, individual sounds within the word are allowed to have poor scores.

The graph in Figure 3-14 compares word-pruning and model-pruning, both using

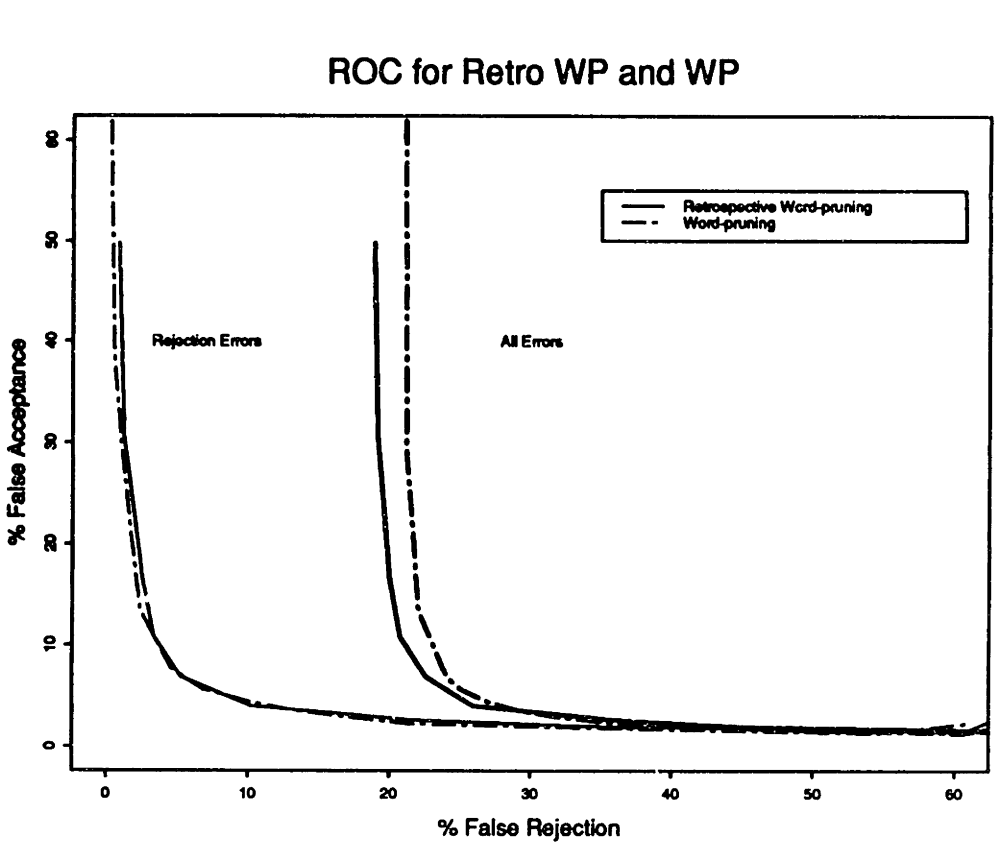

Figure 3-13: Performance of retro-word-pruning with umbrella of 300

the constant threshold criterion, counting rejection errors. Performance of model pruning is substantially worse than word pruning, presumably because the recognizer is not a good enough model to make such strict demands on model scores. By pruning at the word level, this inadequacy is glossed over to some extent.

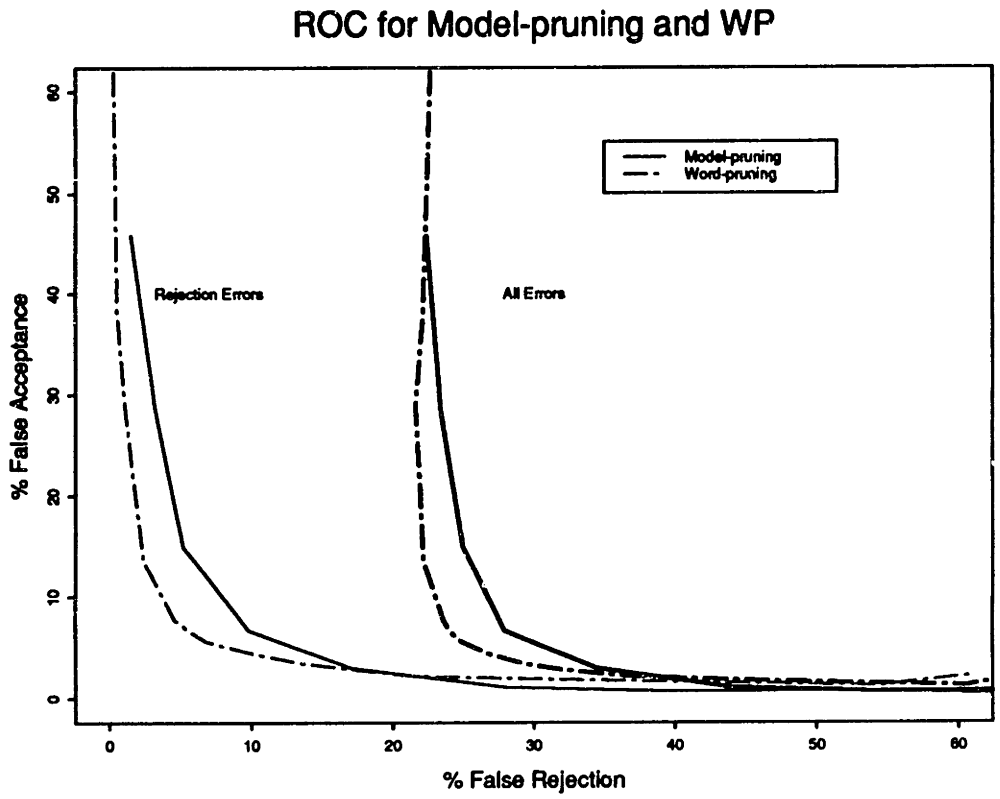

Figure 3-14: Performance of model-pruning and word-pruning

Chapter 4

Summary of Results

It is quite clear that word-pruning, using the retrospective criterion, overall performs better than the other algorithms. Model specific normalizing and model pruning both performed substantially worse. Table 4.1 summarizes the performance (% False Acceptance) of each of the algorithms at two points (counting only rejection errors): 5% False Rejection and 10% False Rejection.

At the outset, three of the thresholds were fixed for comparison of each of the algorithms, and for deriving the ROC in the graphs: lookahead was set at 0.20 seconds, MAX_TME was set at .10 seconds, and MIN_OVLP was set at 75%. To verify that these parameters affect each of the algorithms to approximately the same extent, several graphs were constructed to show ROC's for other operating points. As the lookahead changes, the primary effect is to reduce temporal and overlap errors, especially towards lower thresholds. Figures 4-1, 4-2, and 4-3 show the performance of pseudo-word-spotting, max-sum-diff, and word-pruning across four values of lookaheads (keeping MAX_TME and MIN_OVLP the same). It is quite clear from these graphs

Algorithm	False Acceptance at 5% False Rejection	False Acceptance at 10% False Rejection
Pseudo Word Spotting	13.2%	7.7%
Max Sum Diff	13.5%	7.4%
Word Pruning	7.3%	4.5%
Model Pruning	16.1%	6.6%

Table 4.1: Percent false acceptance of all algorithms (only rejection errors)

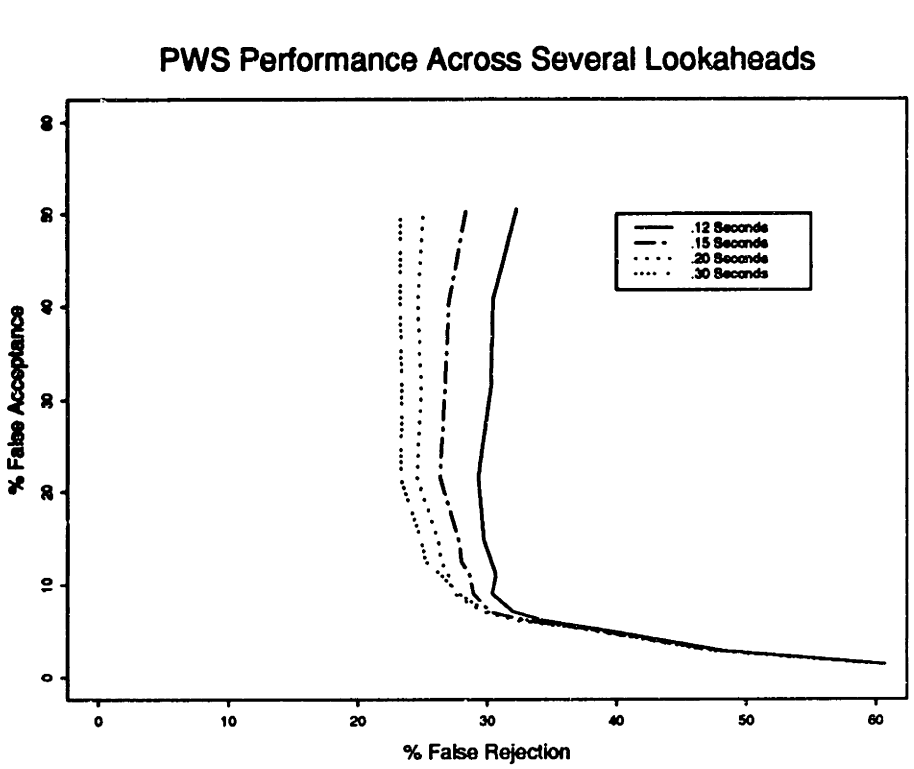

Figure 4-1: Pseudo-word-spotting performance as lookahead varies

that changing lookahead tends to change the performance of each in the same way (ie, longer lookahead shifts the curves left).

Variations in the temporal threshold, MAX_TME, were tested. A curious effect appeared as it varied from .10 to .30 seconds: while the number of temporal errors did fall substantially as MAX_TME rose, it was almost entirely offset by an increase in overlap errors. It seems that those tokens that were causing temporal errors were being swapped over almost entirely to overlap errors.

Finally, three different thresholds for MIN_OVLP were tested: 75%, 50%, and 0% (just ignore overlap errors entirely). These three curves, for each of the three algorithms, are shown in Figures 4-4, 4-5, 4-6; they were computed using a lookahead of 0.20 seconds, and MAX_TME of 0.20 seconds. It is clear from these graphs that at the chosen operating point for the two fixed thresholds, a large portion of the errors are overlap errors.

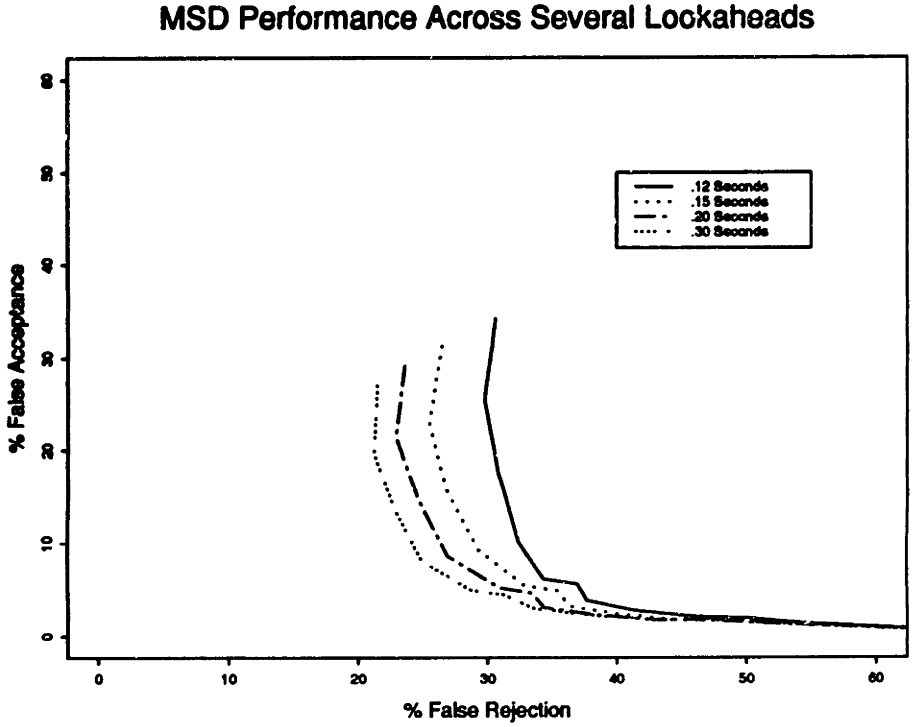

Figure 4-2: Max-sum-diff performance as lookahead varies

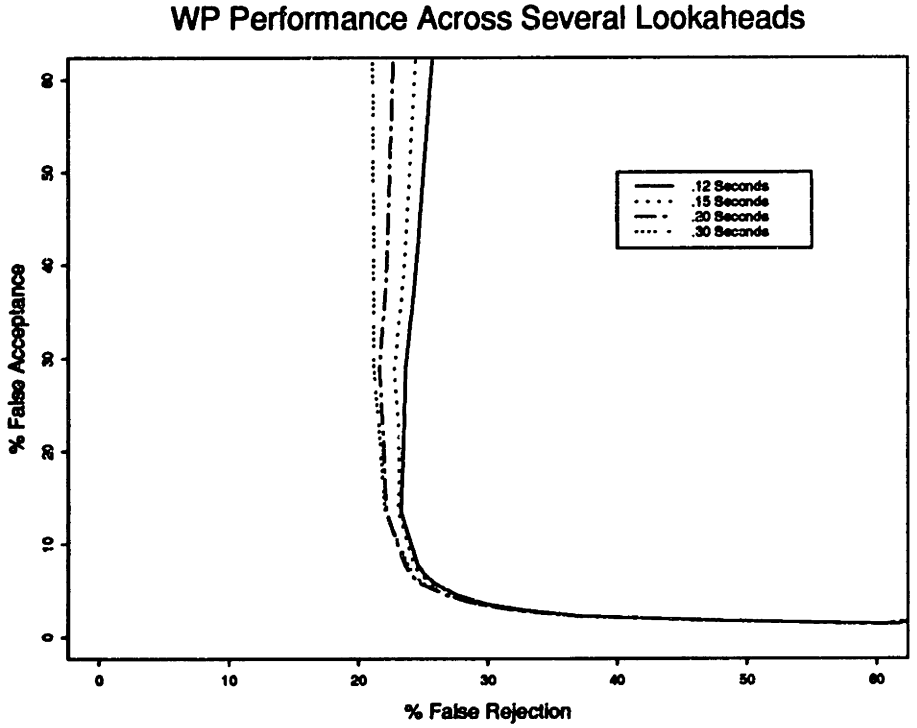

Figure 4-3: Word-pruning performance as lookahead varies

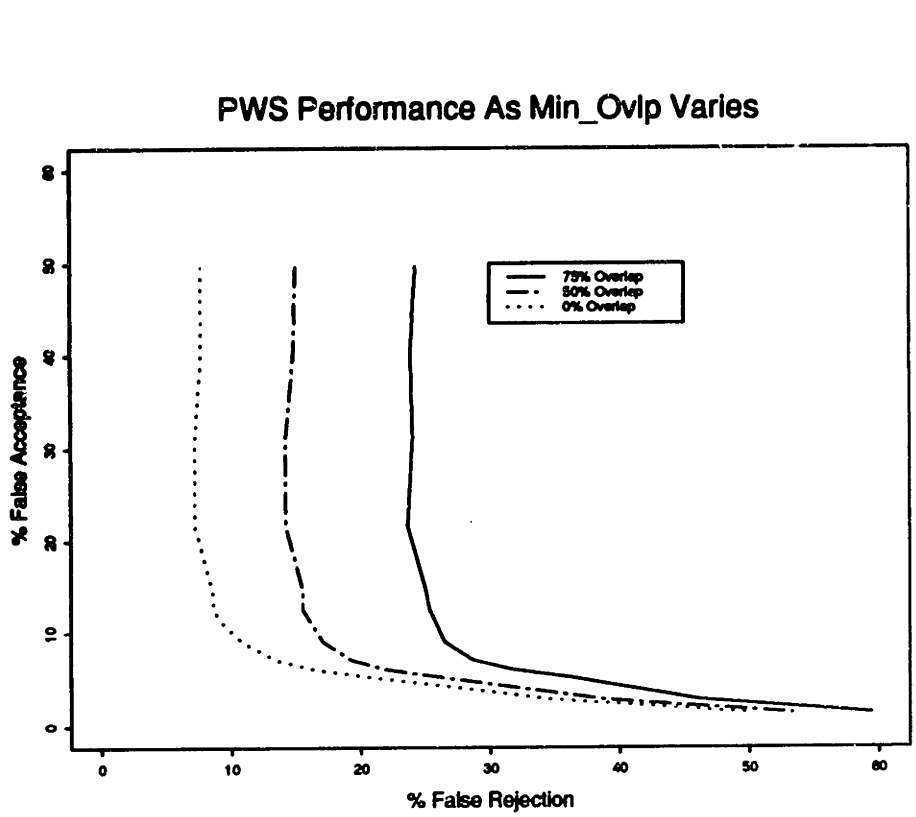

Figure 4-4: Pseudo-word-spotting performance as MIN_OVLP varies

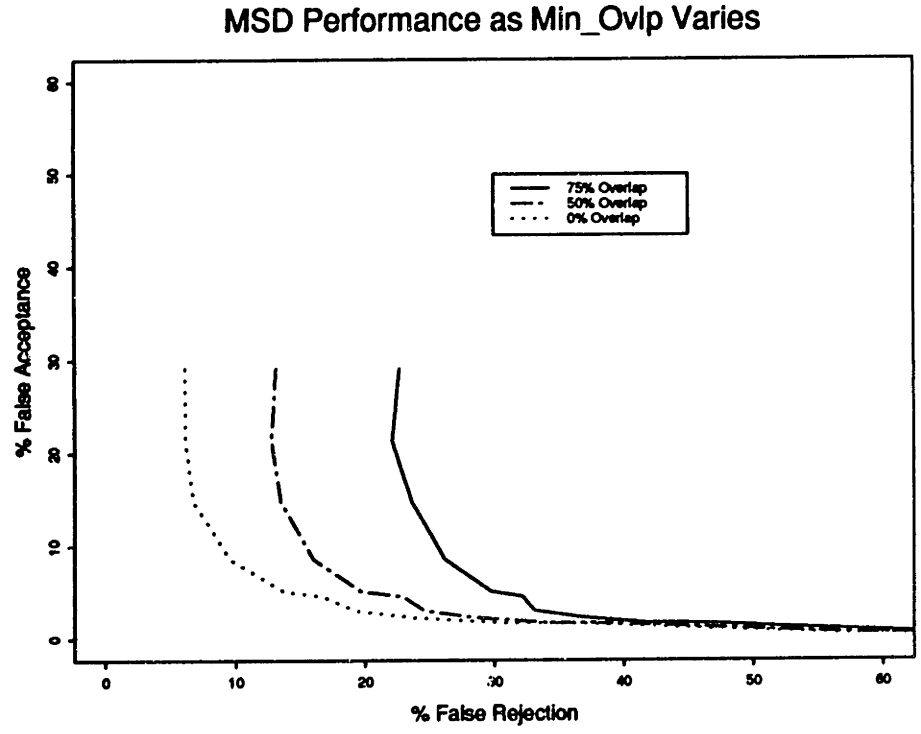

Figure 4-5: Max-sum-diff performance as MIN_OVLP varies

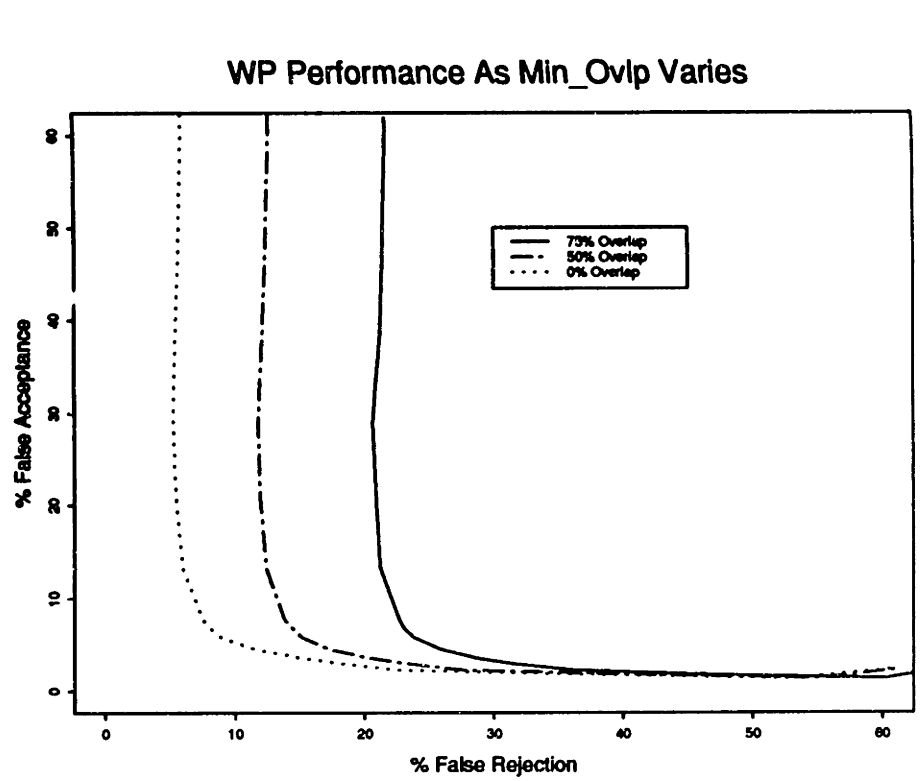

Figure 4-6: Word-pruning performance as MIN_OVLP varies

4.1 Temporal and Overlap Errors

Each of the algorithms seems to exhibit a very large number of temporal and overlap errors (somewhere between 20% and 30%), along with small variations between each of the algorithms. To understand just why this is happening, the temporal and overlap errors under constant threshold word-pruning, using a lookahead of 0.20 seconds and MAX_TME of .10 seconds, were examined in some detail using a tool that showed more closely what was occurring.

It quickly became clear that very short words account for almost all overlap errors (as expected), and that most temporal errors tend to be quite close (just not below 0.10 seconds). Often the difference of one time boundary accounted for more than that threshold. From the 10% false rejection point, three graphs were constructed showing the magnitude of the errors, and the lengths (in seconds) of the words that caused overlap errors. Figure 4-7 shows the percent overlap of the overlap errors: those that overlapped with the correct word less than 75%. A large portion of these errors fall above 50%. Figure 4-8 shows the word lengths of those words that caused

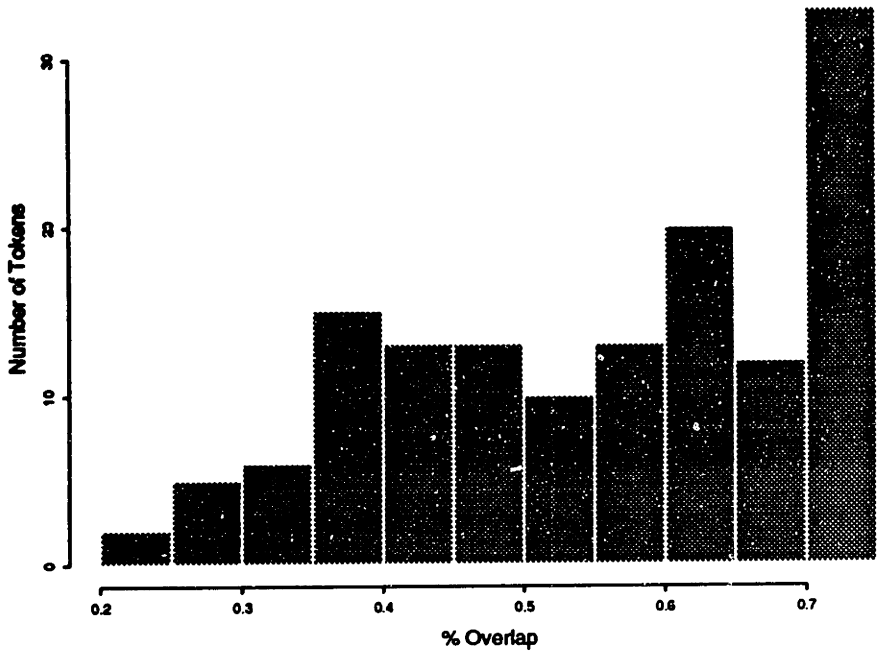

Figure 4-7: Percent overlap of overlap errors

overlap errors — they are very short. Finally, Figure 4-9 shows the distance, in time, of the aligned word from the correct word, when greater than the threshold of .10 seconds. Most of these temporal errors fall below .20 seconds.

Thus, it seems that most of these errors can be attributed to the rather demanding thresholds we set for temporal and overlap errors. The graphs of the previous section show the substantial performance increase when they are relaxed somewhat.

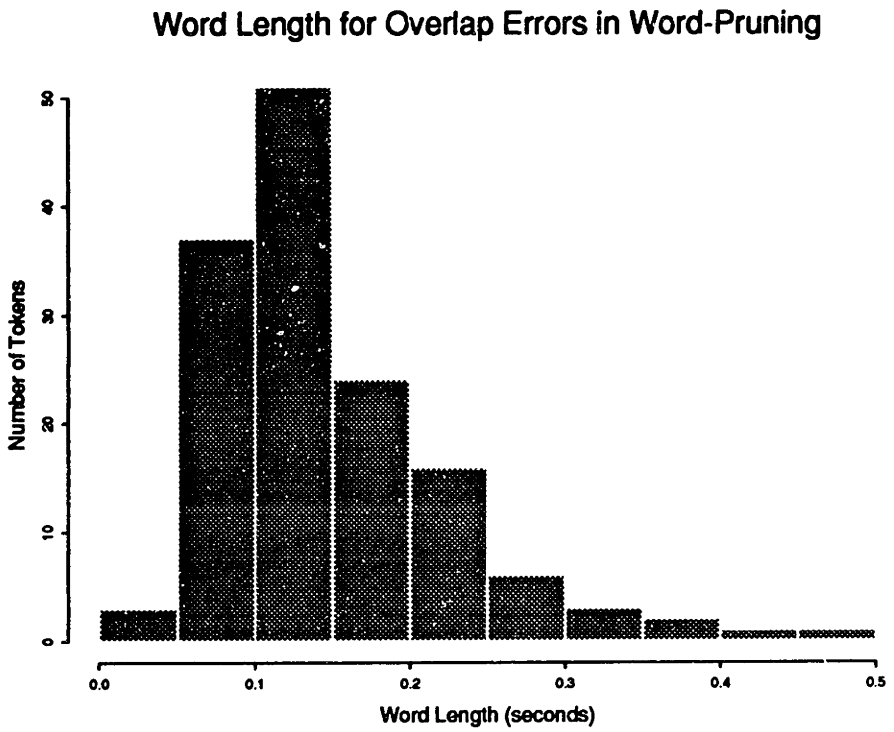

Figure 4-8: Word lengths of overlap errors

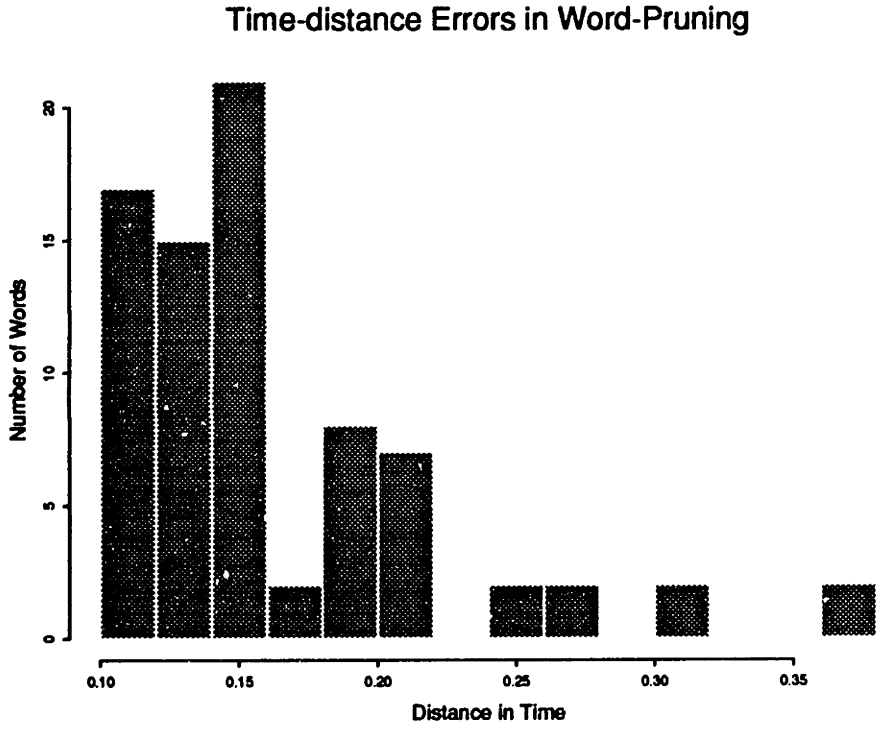

Figure 4-9: Time distance of temporal errors

Chapter 5

Conclusion

With existing speech recognition technology, a reasonable literacy tutor could be implemented today, using some modification of word-pruning with some form of normalization. The biggest impediment is the fact that no real data has been collected — before such a system can be implemented, real data must be collected to model the behavior of users. Once real data is collected, the run time issues can be addressed, and normalization schemes and thresholds can be established for the rejection algorithm.

It may also be necessary to develop new algorithms to, in general, train recognizers not for optimization of performance, but of rejection. When some of the algorithms tested were translated to a real time system, a number of difficulties arose. One of the most troubling was that while word scores tend to vary as a function of the length of the word, there is also substantial variation from word to word of the same length. Certain words would always receive low scores, presumably because their models had low means. Such variance may be optimal for performance, but make consistent rejection much more difficult. To deal with this directly, it may be necessary to implement word-specific rejection, which would require a substantially larger amount of data for training.

Much work must be done to deal with some of the interface and real time issues that were not considered in this thesis. Because the literacy tutor is a feedback system, the user will stop and repronounce a word if that word was rejected. The user may

also back up a few words, or start the sentence again. Robust mechanisms need to be developed to detect when the user is waiting for feedback, and deal with the resulting restarts accordingly. It is very important that the literacy tutor, even a prototype, operate in real time to provide this feedback quickly, and to avoid frustrating the user. It could certainly be worthwhile to cut a few corners during recognition, affecting accuracy somewhat, in order to operate closer to real time. The whole issue of when and how to help the user in pronouncing a word must also be examined closely.