Internal Use Only (非公開)



TR-IT-0061

# Automatic Labeling of Prosodic Structure

Colin W. Wightman

Nick Campbell

1994.7

ABSTRACT

This paper describes a series of enhancements to the algorithm recently presented by Wightman and Ostendorf for labeling prosodic patterns in speech. Through a combination of improved methodology and enhanced feature extraction, significant reduction of the error rates associated with labeling prominences and major phrase boundaries are achieved. In particular, we show that joint labeling of the prominence and boundary phenomena can significantly improve the performance on both types of labels while eliminating inconsistencies which had been observed between the two label sets. Application of the improved algorithm is illustrated with experimental results for joint labeling of prosodic structure in a corpus of professionally read speech.

©ATR Interpreting Telecommunications
Research Laboratoriess.

©ATR 音声翻訳通信研究所

# Automatic Labeling of Prosodic Structure

Colin W. Wightman
Nick Campbell

ATR Interpreting Telecommunications Research Labs

Kyoto, Japan

# 1 Overview

This report describes work undertaken during the summer of 1994, while the first author was visiting ATR. The report is composed of three sections: The first presents a copy of a paper submitted to the IEEE Transactions on Speech and Audio Processing for publication, and describes a series of enhancements to the algorithm recently presented by Wightman and Ostendorf for labeling prosodic patterns in speech. Through a combination of improved methodology and enhanced feature extraction, significant reduction of the error rates associated with labeling prominences and major phrase boundaries has been achieved. In particular, the paper shows that joint labeling of prominence and boundary phenomena together can significantly improve the performance on both types of labels while eliminating inconsistencies which had previously been observed between the two label sets. Application of the improved algorithm is illustrated with experimental results for joint labeling of prosodic structure in a corpus of professionally read speech.

In the second section, we briefly identify the primary software components which have been developed as part of this project and document the shell scripts which implement all the primary functions. The third and final section contains source code listings for the two primary programs.

# 2 Technical Report

the following pages contain a copy of the paper submitted to the IEEE Transactions on Speech and Audio Processing for publication

# Improved Labeling of Prosodic Structure

Colin W. Wightman
Department of Electrical Engineering
New Mexico Institute of Mining and Technology
Socorro, NM 87801.

Nick Campbell
ATR Interpreting Telecommunications Research Labs
Kyoto 619-02, Japan

October 4, 1994

**Abstract**

This paper describes a series of enhancements to the algorithm recently presented by Wightman and Ostendorf for labeling prosodic patterns in speech. Through a combination of improved methodology and enhanced feature extraction, significant reduction of the error rates associated with labeling prominences and major phrase boundaries are achieved. In particular, we show that joint labeling of the prominence and boundary phenomena can significantly improve the performance on both types of labels while eliminating inconsistencies which had been observed between the two label sets. Experimental results are presented for joint labeling of prosodic structure in a corpus of professionally read speech.

## 1 Introduction

There is increasing interest in the use of prosody for the automated processing of spoken human language (c.f. [1, 2, 3, 4]). This interest is motivated largely by the growing availability of large speech corpora, combined with an awareness of the inadequacy of the features currently employed in the labeling thereof, and driven by the importance of prosody in human processing and understanding of speech. A central difficulty faces researchers in this area however: until recently, there had not been a consistent and effective means of automatically labeling the prosodic structures of interest. Without such a method, system designers were unable to detect prosodic features and were consequently unable to use them in later processing. Similarly, researchers attempting to apply corpus-based methods to the study of prosody have been unable to efficiently label the large corpora needed to make effective use of such methods.

In previous work, Wightman and Ostendorf have described an algorithm for labeling prosodic patterns in speech [5]. The method is novel in that it is designed as a post-recognition process,

utilizing the output of a speech recognizer as one source of information, and because it provides a single framework in which the full spectrum of cues that have been proposed for prosodic structures can be treated in an integrated manner. Nevertheless, the method they describe has several shortcomings which led to only modest labeling accuracy with relatively high false detection rates.

In this work, we continue to emphasize **phrasing** and **prominence** as prime determinants of prosodic structure. The former refers to the groupings of words in an utterance, and the latter describes the greater perceived strength or emphasis of certain syllables in each phrase. The roles of prominence and phrasing in spoken language, the cues which have been proposed for them, and previous attempts to detect them were recently reviewed in [5]. In this paper, we extend the Wightman and Ostendorf algorithm, introducing further acoustic measures and refinements of the prosodic information, and improving the algorithm to reduce the labeling process from a two-pass procedure to a single-pass one, which results in significant improvements in all error measures related to prominences and intonational phrase boundaries.

A distinction needs to be made between labeling prosodic *structure* and doing prosodic *transcription*. Prosodic structure describes the basic phrasal groupings and determines which syllables are prominent. In contrast, a complete prosodic transcription seeks to capture not only the presence and location of a prominence or boundary, but also its nature. Thus, when labeling structure, we are content to mark prominences and intonational boundaries with binary-valued flags, a prosodic transcription would require that we also identify the *types* of prominence and boundary events. For example, one of the more widely discussed prosodic transcription systems, ToBI [4], requires that the pitch accent or tone underlying the perception of a prominence be identified from a limited inventory. Likewise, the phrasal and boundary tones causing the perception of a boundary would need to be specified to distinguish, for example, the end of a yes/no question from the end of a declarative sentence. However, such a transcription system is strongly language dependent. Furthermore, while most theoreticians will agree on the presence and locations of prominences and boundaries in a spoken utterance, they differ sharply on the cues that trigger those perceptions and on how to represent them.

In order to be more theory independent, and for practical pragmatic reasons, we maintain that it is desirable to be able to label the prosodic *structure* of an utterance without necessarily producing a detailed transcription. Indeed, a representation of the prosodic structure is a *prerequisite* for producing a detailed transcription. Consequently, we have focussed our efforts on the labeling of prosodic structure, leaving the production of detailed transcriptions as future work.

The labeling task requires mapping a sequence of feature vectors (which we can derive from the acoustics) to a sequence of labels. The simplest approach to this problem is to treat each feature vector as an independent classification problem, assign a class to it, and move on to the next vector. This approach, however is too simplistic and fails to give very good performance when labeling prosody (see below and [6]). It is important to bear in mind that the goal is not simply to classify independent feature vectors, but to map a *sequence* of feature vectors $\mathrm{x}_1^n = \{\mathrm{x}_1, \ldots, \mathrm{x}_n\}$ to a *sequence* of labels $\alpha_1^n = \{\alpha_1, \ldots, \alpha_n\}$.

In order to represent the inter-relationships between the events being labeled (*i.e.*, to take account of the restrictions which govern the allowable sequences of the events) we model the problem

as a finite-state automaton in which there is exactly one state for each label class, and estimate the transition probabilities $p(\alpha_i|\alpha_1^{i-1}, \alpha_{i+1}^n)$, (the probability of being in state $i$ given the history of all the previous and future states) to find the sequence of states ($\alpha_1^n$) which maximizes the probability of the observed data.

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_n) = p(\mathbf{x}_1|\alpha_1)p(\alpha_1) \prod_{i=2}^{n} p(\mathbf{x}_i|\alpha_i)p(\alpha_i|\alpha_1^{i-1}, \alpha_{i+1}^n).$$

Where $p(\mathbf{x}_i|\alpha_i)$ is the conditional probability of observing feature vector $\mathbf{x}_i$ when the system is in state $\alpha$ at time index $i$.

This however, is a very complex operation, because the transition probabilities depend on the *entire* history of the labels. To find the best sequence of tags we would need to explore all possible label sequences leading to and from each feature vector, and the search space very quickly becomes too large to examine exhaustively. The problem can be overcome using stack-decode algorithms to efficiently explore the search space [7], or can be simplified as follows:

$$p(\alpha_i|\alpha_1^{i-1}, \alpha_{i+1}^n) = p(\alpha_i|\alpha_{i-1})$$

That is, we assume that the only thing we need to know about the history of the label sequence is the identity of the last label, under which assumption the problem reduces to a Hidden Markov Model for which powerful algorithms are well known and have been widely implemented (c.f. [8]).

The Wightman and Ostendorf algorithm thus contains three principal components. The feature extraction component transforms the various sources of information (the recognizer output, the actual waveform, pitch-tracking results, *etc.*) into a time-ordered sequence of feature vectors. The frequency with which these feature vectors are produced will depend on the units which are being labeled: one label will be produced for each feature vector. Thus, if we wish to label break indices on word boundaries, we will need to produce one feature vector for each word boundary. The feature vectors are then classified using a decision tree. Decision trees were chosen both because they provide a graceful means of handling extremely non-homogeneous features, and because by inspecting their internal structure we can gain some insight into which features are proving useful and why. Finally, the class likelihoods associated with the leaf nodes of the tree are treated as observation densities in a HMM, and Viterbi decoding is used to recover the most probable label sequence.

In the next section, we present the experimental conditions under which our work has been performed. We describe the corpus used and the fundamental algorithmic architecture. Generally, we use the same test data and conditions as reported in Wightman and Ostendorf [5] so that our results can be compared directly. In the following section, we present two changes in methodology which lead to improved performance and substantially reduced computational requirements. Then, in section 4, we modify and augment the original feature set to utilize better acoustic features. We conclude by evaluating the relative increases in performance attributable to the different modifications and consider the implication of these results for future improvements in prosodic labeling.

## 2 Experimental Conditions

The original labeling system consisted of two label sets: an intonational label set and one describing break indices. Intonational labeling was performed at the syllable level, with each syllable being marked as either prominent (P), carrying a boundary tone marking an intonational phrase boundary (BT), being both prominent and carrying a boundary tone (P-BT), or simple (neither prominent nor having a boundary tone) (S). In addition, all word boundaries were labeled with break indices, to show the degree of disjuncture between the adjoining words on a scale of 0–6. There is some coupling between the two labeling sets in that the boundary tones would normally be expected to coincide with boundaries having break indices of 4 or greater. This labeling system is similar to ToBI in its use of parallel labeling of intonation and break indices, but differs in that a ToBI transcription would specify the type of pitch accent that occurred for each prominence and the type of tone which occurred at the end each phrase. This is the distinction between labeling prosodic structure and prosodic transcription that was discussed in the introduction.

To facilitate direct comparison between our results and those reported previously, we have elected to use the same corpus and labeling task used in Wightman and Ostendorf [5]. The speech corpus consists of radio news stories read by a female professional news announcer. It is a subset of the Boston University Radio News Corpus [9] and contains 14095 syllables in 8568 words spoken in 116 utterances. The utterances are relatively long, roughly paragraph-sized, and there are audible breaths in the speech. The corpus has been hand-labeled with the seven-level break index and binary prominence and boundary tone labels as discussed above. In addition, the corpus has been automatically marked with time-aligned phonetic labels generated by a speech recognizer based on the stochastic segment model [10] and constrained to the word-level transcription. Because we have hand-labeled prominences and break indices for the entire corpus, we can estimate the performance of our algorithms by directly comparing automatically-generated labels with the hand-labels.

All of the results reported here were obtained using the same test procedure as used by Wightman and Ostendorf: the corpus was divided into thirds and the models trained on two-thirds and tested on the remaining third, using rotation, so that each test consisted of three sub-tests in which different thirds of the data were held out. Consequently, we report overall detection statistics as the percentage of the 4539 prominences which were correctly detected (correct detections) and the percentage of the 9556 non-prominent syllables which are mis-labeled (false detections). Likewise, we report the detection of intonational phrase boundaries in terms of the percentage of the 1852 hand-marked boundaries that were correctly detected and as the percentage of the 6716 boundaries which do not end intonational phrases and which are falsely labeled as boundaries. Note that the number of potential boundary sites is *less* than the number of potential prominence sites because prominences can occur on any syllable, whereas intonational phrases end only on word boundaries in fluent speech.

## 3 Improved Methodology

In this section, we address two aspects of the methodology used in the algorithm reported by Wightman and Ostendorf which made application of the algorithm difficult and computationally expensive.

### 3.1 Tree smoothing instead of pruning

The decision trees used for quantization were originally grown using a greedy growing algorithm with a minimum entropy criterion, and then pruned back to some pre-determined size. A difficulty with this approach, however, lies in determining what the appropriate size should be. The sizes reported in [5] were determined empirically by growing several trees to various different sizes and selecting the one which yielded the best performance. This is a costly, time-consuming process which must be repeated each time the feature set or corpus is changed. In addition, the pruning which gave the best results overall generally gave very poor results on rare classes. This occurs because the rare classes make up a relatively small percentage of the observations assigned to nodes high in the tree. When the tree is grown with a minimum entropy criterion, there would be only a small reduction in entropy associated with splitting such classes off: only after the tree has grown larger does splitting off the rarer classes produce significant entropy reductions. Pruning, which proceeds from the leaves towards the root, generally removes the lower nodes where important distinctions that split off the rare classes are made. Consequently, heavily pruned trees tend to be less reliable in identifying observations belonging to rare classes.

To address these problems, we have abandoned pruning and instead use smoothing to weight the estimates of the class probabilities in the subtree. Specifically, we utilize an algorithm which assigns a smoothing parameter $\lambda_i$ to every node $i$ in the tree (The original development of this algorithm was not published. The most coherent description of it can be found in [11]). These smoothing parameters may be regarded as an estimate of our confidence in the class probabilities obtained from the data assigned to a node. That is, if node $i$ has a large number of observations assigned to it, we can be fairly confident that the relative frequency of the classes represented by those observations, $q(\text{class} = c|\text{node} = i)$, are good estimates of the true distributions, $\text{p}(\text{class} = c|\text{node} = i)$. In this case, $\lambda_i$ would be close to 1. On the other hand, if only a few observations were assigned to the node, we should regard the estimated distribution as rather suspect, and assign a small value to $\lambda_i$. The smoothing is then done recursively, finding the smoothed distributions $\tilde{q}(c|i)$ from the local estimate at each node:

$$\tilde{q}(c|i) = \begin{cases} \lambda_i q(c|i) + \frac{(1-\lambda_i)}{\text{number of classes}} & \text{if } i = \text{the root node} \\ \lambda_i q(c|i) + (1-\lambda_i)\tilde{q}(c|\text{the parent node of } i) & \text{otherwise} \end{cases}$$

In other words, each node is smoothed with the one above it, and the root node is smoothed with uniform probabilities. In this regard, the method is very similar to the deleted interpolation smoothing often used in estimating models for speech recognition [12]. The tree smoothing problem can be recast as an HMM training problem and the smoothing parameters can consequently be found using Baum-Welch re-estimation [11]. One point of concern is that the objective of the

smoothing process is to maximize the probability of assigning the correct labels to a set of held-out smoothing data: this is not the same objective function used in growing the tree. It would be substantially more proper to smooth with a minimum entropy objective function but there is no simple algorithm to do this and the present smoothing algorithm yields trees that perform better than those obtained by pruning.

The use of smoothing addresses both of the concerns raised by pruning. By allowing continuous-valued weights, sections of the tree that contain important structure can be retained without overly penalizing performance as a result of over-trained sections. Further, the smoothing eliminates the need to determine the best tree size since that is effectively determined during the optimization of the smoothing parameters.

### 3.2 Joint labeling

The second change to the methodology presented by Wightman and Ostendorf, concerns the treatment of intonational labeling and the labeling of break indices as distinct processes. They used a two-pass procedure in which the complete labeling algorithm was used to label each *syllable* as unmarked, prominent, containing a boundary tone, or both. Then in a second pass, extracted a new set of feature vectors and generated a label for each *word* marking its break index (the break index assigned to the boundary following that word). This approach has several drawbacks: (1) it is computationally inelegant, requiring two passes of the complete algorithm, (2) some information used to label intonation was not made available to label break indices and *vice-versa*, and (3) the labels produced by the two passes were not coordinated.

The lack of coordination between the two labeling processes produced labels in which there were a substantial number of conflicts between the intonational labels and the break indices. Because the break indices of 4, 5, and 6 are generally regarded as corresponding to intonational phrase boundaries, we can expect that almost all boundaries with these break indices will also have a boundary tone marked on the last syllable preceding the boundary. Likewise, we expect that almost no boundaries with break indices of less than 4 would be preceded by boundary tones. However, although the estimated probabilities of prominence and boundary tone were included in the feature vectors used for break index labeling, the two sets of labels, intonation and break indices, were often inconsistent. The very weak coupling between the two passes in the two-pass approach could not take advantage of this powerful constraint.

A second consequence of the two-pass approach to labeling was the division that occurred between features used for labeling break indices and those used for labeling intonation. One consequence of this division is that while Wightman and Ostendorf report detecting 78.1% of the intonational phrase boundaries, they report detecting only 71.2% of the boundary tones. Given the comments made above with regard to the co-location of boundary tones and intonational phrase boundaries, it is clear that some of the information being used to label break indices might also be useful in labeling intonation. Similarly, although they had only a 4.3% false detection rate for boundary tones, they reported a 6.5% false detection rate for intonational phrases, suggesting that some of the information used in labeling boundary tones could be fruitfully applied to the labeling

of break indices. Note that this information is not necessarily the elements in the feature vectors: it may well be contained in the groupings which are formed in the decision trees themselves. Consequently, merely using feature vectors containing the union of the two original feature sets may not produce the desired results, and certainly would not address the problem of inconsistencies between the two label sets.

We have addressed both of these issues by changing to a joint labeling system. That is, rather than doing a first pass to generate 4-valued intonational labels for each syllable, followed by a second pass to generate 7-valued break indices for each word, we do a single pass to generate labels which mark both intonational events and break indices. To do this, it was necessary to combine both the *number* of labels (4-valued and 7-valued) and the *level* at which the labeling is to be done (syllable *versus* word). To preserve the resolution of our labeling we continue using syllable-level labels. This raises the question of what break index should be assigned to word-internal syllable boundaries, since they were previously defined only at word boundaries. Campbell [13] has proposed the use of a level 0 break index for word-internal syllables, and Wightman. *et al.* [14] have observed that it seems to be appropriate for boundaries within prosodic words. Consequently, it is an obvious generalization to include all word-internal syllable boundaries in this category and assign them break indices of 0.

Having decided to use syllable-level labeling, we needed to define a unified labeling system which combines the information carried in both the intonational labels and the break indices used previously. Initially, we used a simple Cartesian product, producing twenty-eight labels which corresponded to every possible combination of the four intonation labels and the seven break indices. After examining the confusion matrices produced when this label set was used, however, we concluded that fully half of the labels were never used. In particular, we noticed that boundary tones were *never* marked on syllables with break indices of 3 or less, and were *always* marked on syllables with break indices of 4 or greater. Consequently, we collapsed our joint labels to a set of fourteen, corresponding to the seven break indices with, and without, prominence. Notice that we are no longer explicitly labeling boundary tones: their presence is implicitly inferred from the presence of a break index of 4, 5, or 6.

### 3.3 Performance

In Table 1, we show differences in results between the algorithms. By replacing pruning with smoothing, we have obtained a considerable procedural advantage in that we no longer need to conduct a series of tests to determine the best tree size each time we alter the feature set or training corpus. Likewise, although the use of joint labeling appears to improve the overall performance measures only slightly, it eliminates the errors arising from inconsistencies between the break indices and the intonational labels, and dramatically simplifies the application of the algorithm by eliminating the need for a second pass. Additionally, by letting boundary tones be implied by the presence of a break index of 4 or greater, we have improved the detection rate for boundary tones from 71.2% to 78.6%.

| | Prominences | | Intonation Phrases | | Break Indices | |
|---|---|---|---|---|---|---|
| | correct | false | correct | false | correct | within ±1 |
| Baseline | 83.6 | 12.7 | 78.1 | 6.5 | 66.9 | 88.6 |
| Joint/smoothed | 83.1 | 12.2 | 78.6 | 5.6 | 64.6 | 87.7 |

Table 1: Comparing the overall performance measures: numbers are percentages. The Baseline results are those reported by Wightman and Ostendorf. Statistics for intonational phrase boundaries and break indices were computed using word-final syllables only.

# 4 Improved Features

Having improved our labeling methodology as described above, we were able to efficiently evaluate a large number of changes to the feature set used by Wightman and Ostendorf. We present these changes by grouping them into three major categories: (1) changes arising from a more principled representation of the F0-contour of the utterance, (2) new features designed to provide information from a larger neighborhood of surrounding syllables (*i.e.*, contextual cues), and (3) additional features based on acoustic correlates not previously considered. In each subsection, we describe the nature of the changes and discuss their motivation. Other than a few specific features discussed to illustrate key points, however, detailed enumeration of the features used is deferred to the final subsection on performance.

## 4.1 Representing the F0-contour

Prominence and intonational phrase boundaries are generally considered to be intonationally marked and the most readily available representation of a speaker's intonation is the F0-contour. However, extraction of the F0-contour is a notoriously difficult problem and even the best pitch trackers continue to make significant errors[15]. Moreover, many variations in the pitch track are due to segmental effects and so are very local, while the intonational events which mark prominences and boundaries are represented in the larger-scale, overall shape of the contour. We use the term "intonational events" here to avoid the controversy regarding the proper description of these markers: as mentioned above, researchers generally agree on the location of boundaries and prominences, but there are deep theoretical differences over what cues the listeners perception of these events. There is a considerable literature on this topic and readers are referred to [16] for a review. Of importance to the current work, however, is the consensus on the locations of boundaries and prominences, and on the general co-location of those events with significant features of the F0-contour (generally extrema: peaks, valleys, and inflection points).

Wightman and Ostendorf have noted that many of the features which they use for intonational labeling are *ad hoc* ratios of local averages of the output from a pitch tracker, rather than a principled representation of the overall shape of the F0-contour that can capture the prosodically-relevant features while suppressing the local, micro-prosodic effects and pitch tracking errors. Although a few methods have been proposed for generating such stylized representations of the overall contour (c.f. [17, 18]), these techniques produce essentially piecewise linear representations and do not

automatically produce parameters which are directly interpretable as features that we can tie to the syllables which we are trying to label. One method, however, is particularly well suited to our needs: the asymmetrical modal quadratic regression method developed by Hirst [19]. Hirst models the pitch track by using a quadratic function constrained by "target points" which he chooses so that a spline passing through any 3 adjacent targets will give the best approximation of the F0-contour in that region.

Hirst's use of target points is particularly useful for two reasons. First, the target points identify the extrema of the overall contour and thus coincide with linguistically interesting phenomena such as prominence and intonational boundaries. Indeed, Hirst has argued extensively that these target values form the basis of a valid phonetic description of the intonation of an utterance [20, 21, 16]. Second, he has published an efficient algorithm for automatically identifying the target points from the output of an automatic pitch tracker, thus making it possible for us to generate the target points as part of our automated labeling process.

Incorporating Hirst's target points to represent the overall structure of the F0-contour made it possible to modify several of the acoustic features to utilize this higher-level information. For example, consider the use of the ratio of the maximum pitch in a syllable to the mean pitch in that syllable to provide a crude estimate of the size of a peak in the F0-contour of the syllable (many prominent syllables contain such a peak). Unfortunately, this is a very noisy feature which lumps together both micro- and macro-prosodic effects, and it is susceptible to pitch tracking errors. The same values may result from local segmental perturbation of F0, from genuine peaks or maxima in the overall contour, or from a spurious error of the pitch tracker on a single frame. In contrast the target values computed from the output of the pitch tracker appear to be quite robust to errors [19], and they appear to reflect the macro-prosodic component of the contour. Consequently, we have altered several of the original features to use the maximum, minimum, and last *target values* in a syllable rather than F0 values, as discussed below.

One concern which is raised by the use of pitch targets is that they may not occur in the desired syllable. That is, a syllable may be perceived as prominent, with a peak in the F0-contour associated with that prominence, but the target value associated with that peak might not occur until after the syllable boundary. In this case, the target point would be part of the feature vector for the *next* syllable. To address this issue, we must make our feature vectors include information from a wider context than just the current syllable. This is discussed in more detail in the next subsection.

## 4.2 Using a larger context

Prominence and intonational boundaries can be somewhat dispersed in their effects and often cannot be isolated to a single segment or even, in some cases, to a single syllable. For this reason, if we are to achieve good labeling performance, we need to include information about a wider context than just a single syllable in our algorithm. To some extent, the transition probabilities used in in our model encode contextual information, but these only consider the *labels* of the adjacent syllables, rather than any of their acoustic features, and provide no information about syllables

other than those immediately adjacent to the syllable in question. To gain access to the richer acoustic information of other syllables and include more than just the immediate neighbors, we need to extract that information and include it in the feature vectors.

For cases like a peak in the F0-contour which occurs on one syllable but which cues a prominence perceived on the next or previous syllable, we have included binary-valued flags in the feature vectors which indicate whether the previous or following syllables have pitch targets assigned to them and whether those targets include a peak in the pitch contour.

Additionally, while the simple flags provide information about the surrounding syllables, they do not provide any information from larger contexts. One such context is the word. We typically only expect (at most) a single prominence within a word and, very often, that prominence is associated with the largest peak in the F0-contour. Thus we can expect that the ratio of the maximum pitch target in a syllable to the mean pitch of the *word* might provide more useful information. Likewise, the ratio of the mean pitch on one word to the mean pitch in the following word may help detect the pitch range resetting that often occurs at intonational phrase boundaries and thus improve the detection of these events.

Use of such higher-level information as the location of word boundaries and adjacent pitch targets is consistent with our approach to prosodic labeling and is *a priori* information which is accessible given that our input is produced by a speech recognizer. Indeed, the only information which we cannot include in our feature vectors is that which requires knowledge of the labels assigned to other syllables in the utterance. As described above, we have chosen to use only knowledge of the previous label, and that knowledge is already contained in the transition probabilities: including it in the feature vectors would be redundant.

### 4.3 New acoustic correlates

In addition to using an improved representation of the F0-contour and including more contextual information in our feature vectors, we have also introduced features to capture two acoustic correlates of prominence which have been proposed recently and which were not considered previously. Specifically, we refer to F0-intensity [22, 23, 24] and the harmonic ratio [25, 26].

Several recent developments in voice quality research (c.f. [25] and [26]) have shown that changes in spectral slope, produced by different configurations of the glottis, can be of relevance to the perception of prominence and can function independently of fundamental frequency change. Ananthapadmanabha, has referred to the peak of the negative derivative of the closing phase of the glottal pulse as an 'excitation amplitude', possibly correlated with prominence. This can be measured indirectly from the spectrum as the ratio of the first and second harmonics and is referred to as the *harmonic ratio*. In order to make use of acoustic correlates of vocal intensity, we have included this measure along with a measure of the energy at the fundamental frequency, as additional features in the model.

### 4.4 Performance

In Table 2 we present a complete list of the elements included in the feature vectors which yielded the best overall performance. The table provides a description of each feature and a brief statement of the motivation for its use. The entries of the table are organized in terms of the information provided by each. That is, in growing the tree, a record was kept of how much entropy reduction was due to splits on each feature. Thus, the end-of-word flag produced the largest change in entropy and is therefore in some sense the most useful feature. Notice that for features which can have many values (*i.e.*, non-flags) the entropy reduction for that feature is reported as the sum of the entropy reductions obtained by splitting on that feature with any threshold although, in terms of binary decision trees, splits with different thresholds are technically different questions. It must be recognized that the ranking and information content given for each feature is valid only in the context of all the other features. Many of the features contain partially redundant information (*i.e.*, their mutual information is non-zero) so removing one may significantly change the entropy reduction obtained from using the other in the tree design.

To explore this, and to identify the minimal set of features needed to achieve some modest level of performance, a series of experiments was conducted. In this series, the lowest ranked feature was removed and the entire labeling experiment was repeated to estimate the new performance level. This basic cycle was repeated until only a single feature was left. By always removing the feature providing the least information in the previous cycle, we hoped to identify the smallest set of the most useful features.

The results of this series of twenty-one experiments is presented in Figure 1. In this figure, we plot false detections *versus* correct detections (best performance is in the lower right-hand corner) parameterized by the number of features used. The points farthest to the left correspond to labeling using a single feature. The results are quite striking: while our best performance with the full feature set is significantly better than that reported by Wightman and Ostendorf, almost all of that performance can be accounted for by only six features (the top 6 entries in Table 2). Indeed, credible performance can be obtained with only three or four features (delete the pause duration, stressed nucleus flag, and the syllable mean pitch features). These results should not be interpreted as indicating that many of the acoustic correlates which have been identified are unimportant: we anticipate that many of the features which provided only minimal reductions in entropy in this study will be crucial to future studies on other corpora. For example, in radio news, there are very few low pitch accents. Consequently, it is not surprising that the ratio of the minimum pitch target to the word mean pitch was not very useful. In a corpus of conversational speech, however, we might expect this feature to become much more important.

## 5 Conclusions

In summary, we have described a series of modifications to the model proposed by Wightman and Ostendorf [5] which lead to significantly improved performance and greatly reduced complexity. To facilitate a direct comparison, we report our results on the same speaker-dependent task used in

| Feature | Motivation | Information |
|---|---|---|
| †End-of-word flag | All word-internal syllables have zero break indices | 0.913 |
| †Stressed nucleus flag | Prominences usually occur on stressed syllables | 0.314 |
| †Duration of following silent pause | Pause duration is correlated with break indices at and above intonational phrase boundaries [27] | 0.276 |
| †Normalized Duration of syllable rhyme | Correlated with break indices below the intonational phrase level [5] | 0.247 |
| Maximum pitch target / mean pitch in word | Prominences frequently occur on large peaks in F0-contour [28] | 0.172 |
| Syllable mean pitch (in ERB's [29]) | A broad measure of F0 which is well defined even when no targets occur in the syllable | 0.140 |
| Word mean pitch / next word mean pitch | F0 declination is reset at intonational phrase boundaries [30] | 0.104 |
| F0-intensity | See discussion above | 0.101 |
| Last target in syllable / global mean pitch | Can identify some types of boundary tones [31] | 0.100 |
| First syllable flag | Prominences more often occur on the first syllable of multi-syllabic words [32] | 0.091 |
| Harmonic ratio (HR) in syllable nucleus | See discussion above | 0.088 |
| Difference in HR between center and adjacent syllables | Estimates peaks in harmonic ratio | 0.076 |
| H flag | Syllable contains a pitch target corresponding to a peak in the overall F0-contour | 0.073 |
| Minimum pitch target / word mean pitch | Some prominences occur near minimums in the F0-contour [33]) | 0.070 |
| †Maximum pitch target / syllable mean pitch | Identifies local peaks in the F0-contour as discussed above | 0.047 |
| Previous syllable has pitch targets (flag) | Targets may not be assigned to the correct syllable, as discussed above. | 0.035 |
| Next syllable has pitch targets (flag) | Targets may not be assigned to the correct syllable, as discussed above. | 0.031 |
| Previous syllable has H pitch target (flag) | Targets may not be assigned to the correct syllable. Flag detects peak in previous syllable | 0.025 |
| †End-of-Utterance flag | Most utterances end at intonational phrase boundaries | 0.024 |
| Syllable contains exactly one pitch target | Some syllables are both prominent and contain a boundary tone: might produce multiple targets | 0.022 |
| †Breath flag | Speakers appear to breathe primarily at boundaries with break indices of 5 and 6 [5] | 0.021 |
| 3 or more syllables in word (flag) | Reduces tendency to mark more than one prominence in multi-syllabic words. | 0.018 |

Table 2: The features used for joint labeling of prominences and break indices. The unit of information used is bits. † indicates features used by Wightman and Ostendorf.

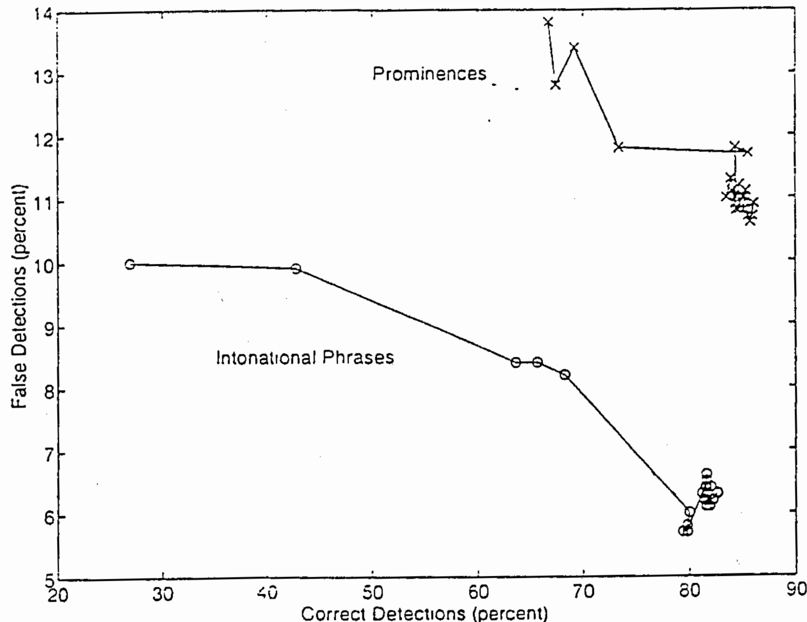


Figure 1: Labeling performance parameterized by number of features used. Left-most points correspond to use of a single feature.

| | Prominences | | Intonation Phrases | | Break Indices | |
|---|---|---|---|---|---|---|
| | correct | false | correct | false | correct | within $\pm 1$ |
| Baseline | 83.6 | 12.7 | 78.1 | 6.5 | 66.9 | 88.6 |
| This study | 85.9 | 10.7 | 82.6 | 6.3 | 64.8 | 88.4 |

Table 3: Comparing the overall change in performance resulting from this study. The Baseline results are those reported by Wightman and Ostendorf[5].

the original study, and compare the overall performance changes in Table 3. We have reduced by 26% the number of intonational phrase boundaries which are missed, while simultaneously reducing the number of false detections by 3%. For prominences, the missed detections have been reduced by 16% while the rate of false detections has been reduced by 19%.

In addition to the overall performance gains, we have eliminated the inconsistencies between the intonational labeling and the break index labeling through the introduction of a joint labeling system. This combined approach has the additional advantage of eliminating the need for a second pass to generate the full label set, thus reducing the computation required by almost half (the gain is not quite half because decoding the larger label set takes slightly longer). The complexity of the approach was further reduced by replacing the pruning method used by Wightman and Ostendorf to improve the robustness of their tree quantizers with a tree-smoothing approach. This change eliminated the need to specify a tree size *a priori* and thus eliminated the need for separate sets of experiments to determine the best tree size to use.

We have shown by a series of experiments, that virtually all of our performance can be achieved through the use of only six or seven of our twenty-two features. This observation, coupled with

the fact that the additional sixteen features improve the various performance measures by only 2.7% on average, suggests that either (1) there is a crucial acoustic feature which we have thus far failed to find in the literature or by experimentation, or (2) we have passed the point of diminished returns with regard to our modeling of the observation distributions and must look next to improved modeling of the transition probabilities.

This latter choice seems more likely as our assumption of a Markov model is clearly not well-supported by what we know about the higher-level organization of speech. For example, Wang and Hirschberg [34] have shown that a single feature, the distance from the last intonational phrase boundary compared to the length of the previous intonational phrase, can predict more than 80% of intonational phrase boundaries in a synthesis application. Stated simply, the Markov assumption that

$$p(\alpha_i|\alpha_1^{i-1}, \alpha_{i+1}^n) = p(\alpha_i|\alpha_{i-1})$$

needs to be relaxed. To do this will require a significant change to the labeling algorithm and is well beyond the scope of the present study. One way in which such a change could be made would be to utilize the tree quantizer as part of a stack decoder, rather than as a pre-processing step to a Viterbi decoder. This would be similar to the approach taken by Magerman's SPATTER parser for natural language syntactic labeling [11]. Indeed, Magerman's parser provides a natural environment in which prosodic labeling and syntactic parsing could be carried out jointly.

While substantially improved, the performance of the existing algorithm is still incomplete. In particular, boundary tones and prominences are not classified beyond simply marking their existence: it would be desirable to transcribe the accent and boundary types. This could be done by a simple *post-hoc* classifier applied to the sites identified by the current algorithm, but there is much evidence to suggest strong interactions between boundary effects and prominences in determining the overall intonation [35], and it is likely that better results would be obtained by incorporating the additional classes directly into the joint labeling system.

Finally, although great care has been taken to utilize features which are either known to be speaker-independent, or have been normalized so as to be robust to speaker variations, a systematic study of the source-dependence of our algorithm has not been completed. Wightman and Ostendorf's results for a limited speaker independent test were encouraging, but a larger test is clearly called for. To accomplish this, and to develop the additional classifications just mentioned, a significant increase in the amount of fully transcribed speech data is needed. The work described here should help provide one tool for obtaining this larger corpus.

# 6 Acknowledgments

The work described was performed at ATR. The authors wish to thank Drs. Yamazaki, Sagisaka, and Higuchi, for their continuous support of this effort. We also wish to thank Professor Osamu Fujimura for his kind advice, and to acknowledge the generosity of Mari Ostendorf and her students at Boston University in making portions of the BU Radio News Corpus available to us in advance of its official release. Finally, we must acknowledge our debt to Daniel Hirst who provided us with his computer software for extracting F0 target values.

## References

[1] P. Price, M. Ostendorf, S. Shattuck-Hufnagel, and C. Fong. "The use of prosody in syntactic disambiguation". *Journal of the Acoustical Society of America*, 90:2956–2970, 1991.

[2] J. Pierrehumbert and J. Hirschberg. "The meaning of intonational contours in the interpretation of discourse". In P. Cohen, J. Morgan, and M. Pollack, editors, *Intentions in Communication*. MIT Press, Cambridge, MA, 1990.

[3] M. Ostendorf, C. Wightman, and N. Veilleux. "Parse scoring with prosodic information: An analysis/synthesis approach". *Computer Speech and Language*, pages 193–210, July 1993.

[4] K. Silverman, J. Pitrelli, M. Beckman, J. Pierrehumbert, R. Ladd, C. Wightman, M. Ostendorf, and J. Hirschberg. "Tones and Break Indices: a standard fro prosodic transcription". In *Proceedings of the International Conf. Spoken Language Processing*, Banff, Canada, 1992.

[5] C. W. Wightman and M. Ostendorf. "Automatic labeling of prosodic patterns". *IEEE Trans. on Speech and Audio Processing*, October 1994.

[6] W. N. Campbell. "Combining the use of duration and f0 in an automatic analysis of dialogue prosody". In *Proc. International Conference on Spoken Language Processing*, Yokohama, Japan, September 1994.

[7] F. Jelinek. "A fast sequential decoding algorithm using a stack". *IBM Journal of Research Development*, 13, November 1969.

[8] L. R. Rabiner. "A tutorial on hidden markov models and selected applications in speech recognition". *Proceedings of the IEEE*, 77(2):257–286, February 1989.

[9] M. Ostendorf, P. Price, and S. Shattuck-Hufnagel. "The Boston University Radio News Corpus". manuscript.

[10] M. Ostendorf, A. Kannan, O. Kimball, and J. Rohlicek. "Continuous word recognition based on the stochastic segment model". In *Proceedings of the DARPA Workshop on Continuous Speech Recognition*, 1992.

[11] D. Magerman. *"Natural Language Parsing as Statistical Pattern Recognition"*. PhD thesis, Stanford University, February 1994.

[12] F. Jelinek and R. L. Mercer. "Interpolated estimation of Markov source parameters from sparse data. In *Proceedings, Workshop on Pattern Recognition in Practice*, pages 381–397, Amsterdam, 1980.

[13] W. N. Campbell. *"Multi-level Timing in Speech"*. PhD thesis, University of Sussex, U.K., 1992.

[14] C. Wightman, S. Shattuck-Hufnagel, M. Ostendorf, and P. Price. "Segmental durations in the vicinity of prosodic phrase boundaries". *Journal of the Acoustical Society of America*, March 1992.

[15] B. Secrest and G. Dodington. "An integrated pitch tracking algorithm for speech systems". In *Proc. International Conference on Acoustics, Speech and Signal Processing*, pages 1352–1355, 1983.

[16] D. Hirst. "Prediction of prosody: An overview. In G. Bailly, C. Benoit, and T. Sawallis, editors, *Talking Machines: Theories, Models, and Designs*, pages 199–204. Elsevier Science, 1992.

[17] M. T. M. Scheffers. "Automatic stylization of f0-contours". In *Proceedings of the Seventh FASE Symposiun: Speech 88*, Edinburgh, 1988.

[18] P. Taylor. "The rise/fall/connection model". Forthcoming.

[19] D. Hirst and R. Espresser. "Automatic modeling of fundamental frequency using a quadratic spline function. *Travauxde l'Institute de Phonetique d'Aix*, 15:71–85, 1993.

[20] D. Hirst. "Structures and categories in prosodic representations". *Travauxde l'Institute de Phonetique d'Aix*, 7:297–315, 1980.

[21] D. Hirst. "Structures and categories in prosodic representations". In A. Cutler and R. Ladd, editors, *Prosody: Models and Measurements*, pages 93–109. Springer-Verlag, Berlin, 1983.

[22] A. M. C. Slijter and V. J. van Heuven. "Perceptual cues of linguistic stress: Intensity revisited". In *Proceedings of the ESCA Workshop on Prosody*, pages 246–249, Lund, 1993.

[23] E. Nöth, A. Batliner, T. Kuhn, and G. Stallwitz. "Intensity as a predictor of focal accent". In *Proceedings of the XIIth International Congress of Phonetic Sciences*, pages 230–233, 1991.

[24] J. Pierrehumbert. "A preliminary study of the consequences of intonation for the voice source". *KTH Speech Transmission Laboratory Quarterly"*, pages 23–36, 1989.

[25] T. V. Ananthapadmanabha. "Spectral parameters of voice source pulse". In *Proceedings Speech Technology for Man-Machine Interaction*, pages 159–188, New Delhi, 1991.

[26] O Fujimura, A. Cimino, and M. Sawada. "Voice quality within a sentence: expressive effects of source spectral change". In *Proceedings Vocal Fold Physiology*, Karume, Japan, 1994. Forthcoming.

[27] C. Fong. "Duration modeling for speech synthesis and recognition". Master's thesis, Boston University, 1994.

[28] J. Terken. "Fundamental frequency and perceived prominence of accented syllables". *Journal of the Acoustical Society of America*, 89(4), April 1991.

[29] D. Hermes and J. van Gestel. "The frequency scale of speech intonation". *Journal of The Acoustical Society of America*, pages 97–102, July 1991.

[30] A. Cohen and J. 't Hart. "On the anatomy of intonation". *Lingua*, 19:177–179, 1967.

[31] N. Daly and V. Zue. "Acoustic, perceptual, and linguistic analyses of intonation contours in human/machine dialogues". In *Proceedings Int. Conf. Spoken Language Processing*, pages 497–500, Kobe, Japan, 1990.

[32] A. Cutler and D. Carter. "The predominance of strong initial syllables in the English vocabulary". *Computer Speech and Language*, 2:133–142, 1987.

[33] J. Pierrehumbert. *The Phonology and Phonetics of English Intonation.* PhD thesis, Massachusetts Institute of Technology, 1980.

[34] M. Wang and J. Hirschberg. "Predicting intonational boundaries automatically from text: the ATIS domain". In *Proc. Fourth DARPA Speech and Natural Language Workshop*, Asilomar, CA, 1991.

[35] M. Beckman and J. Pierrehumbert. "Intonational structure in Japanese and English". *Phonology Yearbook 3*, pages 255–309, 1986.

## 3 Software Overview

The software developed for this project is of three different types: (1) Two integrated programs which embody the entire labeling algorithm, (2) component programs which implement functionally distinct parts of the algorithm and which can be used to build new algorithms, and (3) shell scripts which illustrate how the software should be compiled, installed, and utilized.

### 3.1 Integrated programs

There are two integrated programs: autolabel.c and makemodel.c. Both can be found in the directory cww/labeling/final. Together they completely implement the joint labeling of prosodic structure structure described in the previous section.

**autolabel.c:** This program implements the entire labeling algorithm. It takes the aligner output (.words and .phones files) and an augmented F0 file (see go.preproc script, below) in addition to a model file and normalization file and produces the the .tones and .breaks files. Using command-line options, autolabel can be told to adapt duration, pitch and energy in any combination, update the parameter file used for normalizing phone duration, pitch and power, and its output can be controlled to generate several intermediate results: the raw data reformatted, the extracted feature vectors, the VQ codewords, or the final labels. These options are controlled by a 2-digit hexadecimal control word described in the header comment of the source file. Autolabel looks for and utilizes all of the environment variables used by the aligner with the addition of PITCH_PATH which allows the augmented F0 files to be located in a different directory than the waveforms.

**makemodel.c:** Trains the models used by autolabel. It expects two files to be specified on its command line: one contains feature vectors and the other contains the corresponding labels. There are several parameters set in the included header files (documented there) to control the model generation parameters. Makemodel produces a file called newmodel.hvq which should be renamed to something more meaningful and moved to the models directory.

**Adding a new speaker:** To add a new speaker, first align as much of their speech as possible. The use the go.estimate script (see below) to generate a file of normalization parameters for that speaker. The convention is to name the file with speaker identity and .norm as its extension. If prosody

labels are available for the speaker, autolabel should be run to extract feature vectors and then makemodels used to train a model (see go.trainmodel, below). The convention is to name model files using the speakers identity and .hvq as its extension. The .norm and .hvq files should then be placed in $ALIGNER_BASE/models, where the software tools will find them.

## 3.2 Component programs

The component programs are older, less structured, and not terribly pretty. They are included here for the benefit of those who really want to do something not supported by the integrated programs.

**features.c:** This is an early version of autolabel and can produce either a formated version of the input data or feature vectors. Additionally, it expects its input to be the .syl file format used in the subset of the BU radio News Corpus which is installed.

All of the following component programs are designed to handle multiple utterance files. That is, the input files may contain several utterances that have been concatenated, with a '***' to seperate utterance.

**smoothvq.c:** An early version of makemodels. It includes tree smoothing and everything, but only grows a tree: it does not estimate the HMM parameters. The trained tree is put into a file called newvq.tree and the class likelihoods for the leaves are put into a file called likeli.tmp. If the utterance boundaries ('***') don't line up in the features and labels files, smoothvq will complain.

**mle28.c:** Estimates the HMM parameters. makemodels.c is essentially a merged version of this program and smoothvq.c. The output is to stdout and needs to be prepended to the corresponding likeli.tmp file to produce a full HMM model.

**tclassr.c:** Uses a tree produced by smoothvq.c to quantize a set of feature vectors. Output is to stdout.

**fbviterbi.c** The ugliest, most non-intuitive implementation of HMM stuff you should ever hope to encounter. Given a full HMM model and the output from tclassr, fbviterbi will (1) do the viterbi decoding needed and dump the resulting labels (state sequence) to stdout. In addition, the intial and transition probabilities will be updated via Baum-Welch re-estimation and the new values placed in a file called pia.tmp

# 4 Shell scripts

In the directory cww/labeling/final, there are several shell scripts which illustrate the use of the integrated programs. They are illustrations in that they will need to be modified to handle different file names, directory structures, etc. Note that they are not executable directly, but must be invoked with the csh command.

**go.build_progs** Compiles and builds all the programs associated with the integrated programs.

**go.install_all** Copies the built programs and the modified aligner scripts to the appropriate aligner directories. NOTE: the scripts are currently based on the beta version of the aligner: when newer versions are installed, their scripts should be modified in parallel with the modified beta shells rather than being overwritten...DON'T USE THIS SCRIPT ONCE NEW ALIGNER VERSIONS HAVE BEEN INSTALLED!!

**go.estimate** Goes through the corpus and uses autolabel to build up a parameter file contining the values used to normalize duration, pitch, and power. These parameters are usually speaker-dependent so each new speaker should, ideally have a file of type .norm stored in the $ALIGNER_BASE/models directory.

**go.22 and go.get_fvectors** Goes through the database and extracts the feature vectors for each utterance. The 22 refers to the hex code passed to autolabel and causes it to adapt duration, but not pitch or energy, and to dump the feature vectors.

**go.trainmodel** Uses makemodel and the feature vectors generated by go.22 to train a model.

## 4.1 go.build progs

```
# Just a dumb command file to illustrate the compiler options

# Compile all the C programs
foreach prog (autolabel makemodels pitchproc reduc preproc)
    echo Compiling $prog
    gcc $prog.c -O2 -lm -static -o $prog
end

#build the autot widget (the xview app that lets users select models)
#Note that you must NOT use the gnu make program
echo Building autot widget
\rm -f autot
/usr/bin/make

echo The executables are ready...use go.install_all to update things
```

## 4.2 go.install_all

```
cp autolabel $ALIGNER_BASE/bin
cp autot $ALIGNER_BASE/bin
cp ../aligner/Align $ALIGNER_BASE/bin
cp ../aligner/align.BM $ALIGNER_BASE/menus
cp ../aligner/breaklabel.LM $ALIGNER_BASE/menus
cp ../aligner/doautolabel $ALIGNER_BASE/bin
cp ../aligner/tonelabel.LM $ALIGNER_BASE/menus
cp ../aligner/wave_pro $ALIGNER_BASE/files
```

## 4.3 go.estimate

```
# Estimate the phone normalization data

\rm f2b.norm
foreach file (`ls ~cww/data/radio/f2b/st*/*.words`)
set NAME=`basename $file .words`
set BASE=`dirname $file`/$NAME
echo $NAME
autolabel $BASE f2b.norm 11 > t
end
\rm t
```

## 4.4 go.22

```
foreach spkr (f2b)
  echo $spkr
  foreach file (`ls ~cww/data/radio/f2b/st*/*.syl`)
    set fi=`dirname $file`/`basename $file .syl`
    echo $fi
    features $fi $spkr.norm 22 > $fi.pafea
  end
end
```

## 4.5 go.get_fvectors

```
# Extract all the feature vectors and label files into the f2b directory
# in preparation for training a model.

foreach file (`ls ~cww/data/radio/f2b/st*/*.words`)
set NAME=`basename $file .words`
set BASE=`dirname $file`/$NAME
echo $BASE
autolabel $BASE f2b.norm 22 > f2b/$NAME.fea
cp $BASE.palabels f2b/$NAME.labels
end
```

### 4.6 go.trainmodel

```
# Before you can run this shell script, you need to create a file in
# the directory with all the files (./f2b is assumed here) called
# delete_these in which you list any files you DON'T want included in
# the training set. This may be because you want them saved out for
# testing, or (as in the radio news corpus) because they have syllable
#counts that don't match the handlabels...

foreach file (`cat ./f2b/delete_these`)
   \rm ./f2b/$file
end

# Now we put all the files into one pair of BIG files:
echo > all.fea
echo > all.labels
foreach file (`ls f2b/*.fea`)
cat $file >> all.fea
cat f2b/`basename $file .fea`.labels >> all.labels
end

makemodels all.fea all.labels

cp newmodel.hvq ./f2b/f2b.hvq
```

# 5 Source listings

the following pages contain a listing of the source code for the two programs

LINE # SOURCE TEXT

```
601            /* we put a pointer to it in the last syllable structure. If */
602            /* we've already found a nucleus for this syllable then */
603            /* there is a problem with the label files and we just alert */
604            /* the user and set the error flag which will abort the */
605            /* whole program. */
606            stress=stressvowel(label);
607            if (stress>0) {
608              if(last_syl->nucleus==NULL) {
609                last_syl->nucleus=last_phone;
610                if( stress==2) last_syl->stress=1;
611              }else {
612                fprintf(stderr,"Found second vowel in syllable at %d\n",cntr);
613                flag= -1;
614              }
615            }
616          }
617
618        }else if( (type==2) || (type==3) ) {
619
620          /* If we read a word or a syllable tag we'll end up here. The */
621          /* processing of word and syllable boundaries have a fair */
622          /* amount in common since a word boundary is also an implicit */
623          /* syllable boundary: we do the stuff in common first, and */
624          /* then do the word-unique processing if needed. We start out */
625          /* by allocating and checking a new syllable structure. Notice */
626          /* that we don't actually use the new structure just yet. */
627          syl_count++;
628          last_syl->next = new_syl();
629          if( last_syl->next == NULL) flag=0;
630          else {
631            /* Check for error conditions. */
632            if (last_syl->phones == NULL) {
633              fprintf(stderr,"Syllable with no phones detected at %d\n",cntr);
634              flag= -1;
635            }
636            if (last_syl->nucleus == NULL) {
637              fprintf(stderr,"Syllable with no nucleus detected at %d\n",cntr);
638              flag= -1;
639            }
640            /* set the flag in the last phone structure indicating that */
641            /* it is the last phone in its syllable, and set the flag */
642            /* indicating that we have started a new syllable. Also, if */
643            /* we just finished the first syllable in a word, put a */
644            /* pointer to it in the last word structure. */
645            last_phone->last=1;
646            last_syl->stop = last_phone->stop;
647            last_word->nsyl += 1;
648            syl_start=1;
649            if (word_start==1) {
650              last_word->syls = last_syl;
651              word_start=0;
652            }
653
654            /* Now we've finished the processing for a syllable */
655            /* boundary. If this was a word boundary, we still have some */
656            /* more work to do... */
657            if( type==3) {
658
659              /* Allocate and check a new word structure. We don't use */
660              /* it immediately: its for the next word, not the one we */
661              /* just finished. */
662              last_word->next = new_word();
663              if (last_word->next == NULL) flag=0;
664              else {
665                if (last_word->syls == NULL) {
666                  fprintf(stderr,"Word with no syllables detected at %d\n",cntr);
667                  flag= -1;
668                }
669                /* assuming there weren't any problems, copy the */
670                /* appropriate data into the last word structure. */
671                strcpy(last_word->label,label);
672                last_word->sil = pause;
673                last_word->brth = breath;
674                end_time = stop;
675                /* Having filled the last word structure, we advance to */
676                /* the new one we had allocated to get ready for the */
677                /* next word. We also set the flag in the last syllable */
678                /* indicating that it is the last syllable in its word, */
679                /* and set the flag indicating that we've started a new */
680                /* word. */
681                last_word=last_word->next;
682                last_syl->last=1;
683                word_start = 1;
684              }
685            }
686            /* Now that we're finaly done with the last syllable */
687            /* structure, we can advance to the next one. */
688            last_syl=last_syl->next;
689          }
690        }else{
691          fprintf(stderr,"Unexpected value returned by read_file().\n");
692          flag = -1;
693        }
694      }
695      /* We've read everything out of the file so we just need to set */
696      /* the flag in the last word structure to indicate that its the */
697      /* last word in the utterence. */
698      last_word->last = 1;
699
700      /* A couple of checks to make sure we don't have any word */
701      /* odd stuff indicating errors in the label files and we're all */
702      /* done. */
703      if (last_word->syls != NULL) {
704        fprintf(stderr,"Extra syllables at end of file\n");
705        flag= -1;
706      }
707      if (last_syl->phones != NULL) {
708        fprintf(stderr,"Extra phones at end of file\n");
709        flag= -1;
710      }
711    }
712    return(end_time*flag);
713  }
714
715
716  /* read_file actually reads data from files
717  /* This contains all the format-specific code for reading */
718  /* the files. .if you change file formats, this is the routine to hack */
719  /* at. A couple of basic ground rules govern its behaviour:
720
```

LINE # SOURCE TEXT

```
721
722     2) each time this routine is called it should return either a 1 (if
723     it read a phone), a 2 (if it read a word-internal syllable
724     boundary), a 3 (if it read a word boundary), or something negative
725     to indicate a EOF or error condition.
726
727     3) when a word boundary is encountered, all the information about
728     that boundary, including following pauses and breaths, should be
729     read and returned.
730  */
731  int read_file(basename,label,start,stop,dur,breath,pause)
732       char *basename,label[];
733       float *start,*stop,*dur,*breath,*pause;
734  {
735    char filename[128], *sp, *getenv();
736    int start_cnt,dur_cnt,flag,i,interword,type,color;
737    static char phone_buf[BUFFSIZE],word_buf[BUFFSIZE];
738    static char phone[PHONESIZE],word[WORDSIZE];
739    static float phone_time,word_time,last_time;
740    static int first=1, syl_flag=0, word_flag=0;
741    static FILE *fwords, *fphones;
742    static char phone_sep[4], word_sep[4];
743
744    /* Initialization. flag is used to propagate error status. first is */
745    /* a static flag which allows us to handle the startfile process */
746    /* correctly. When we first start a data file, we throw out all the */
747    /* utterence initial junk which isn't part of a word and keep doing */
748    /* so until we either hit the end of the file or we get a non-junk */
749    /* line. */
750
751    flag=1;
752    if (first==1) {
753      if( (sp=getenv("OBASE")) == NULL) sp=basename;
754      strcpy(filename,sp); strcat(filename, WORDS_EXT );
755      if( (fwords=fopen(filename,"r")) == NULL) {
756        fprintf(stderr,"Unable to open words file %s. Aborting...\n",filename);
757        exit(-1);
758      }
759      strcpy(filename,sp); strcat(filename, PHONES_EXT );
760      if( (fphones=fopen(filename,"r")) == NULL) {
761        fprintf(stderr,"Unable to open phones file %s. Aborting...\n",filename);
762        exit(-1);
763      }
764      strcpy(phone_sep,";");
765      fscanf(fphones,"%s",phone);
766      while( !feof(fphones) && strcmp(phone,"#") ){
767        if( !strcmp(phone,"separator") ) fscanf(fphones,"%s",phone_sep);
768        fscanf(fphones,"%s",phone);
769      }
770      strcpy(word_sep,";");
771      fscanf(fwords,"%s",word);
772      while( !feof(fwords) && strcmp(word,"#") ){
773        if( !strcmp(word,"separator") ) fscanf(fwords,"%s",word_sep);
774        fscanf(fwords,"%s",word);
775      }
776      if( (fgets(phone_buf,BUFFSIZE,fphones) == NULL)
777         || (fgets(word_buf,BUFFSIZE,fwords) == NULL) ){
778        fprintf(stderr,"No data in aligner files!\n");
779        flag= -1;
780      }
781
782      last_time=0.0;
783      while( (flag==1) && (first==1)) {
784        sscanf(word_buf,"%f %d %s",&word_time,&color,word);
785        if( (word[0] == '<') || (word[0] == '#') ) {
786          first=1;
787          last_time = word_time;
788          if( (fgets(phone_buf,BUFFSIZE,fphones) == NULL)
789             || (fgets(word_buf,BUFFSIZE,fwords) == NULL) )
790            flag = -1;
791        }else{
792          sscanf(phone_buf,"%f %d %s",&phone_time,&color,phone);
793          if( !strcmp(phone,"sil")
794             || !strcmp(phone,"SIL")
795             || !strcmp(phone,"BRTH")
796             || !strcmp(phone,"brth") ){
797            first=1;
798            last_time = phone_time;
799            if( fgets(phone_buf,BUFFSIZE,fphones) == NULL) flag = -1;
800          }
801          else first=0;
802        }
803      }
804    }
805
806    /* If first==0, then we read an actual label */
807    if( first == 0 ) {
808      if( (phone_time <= word_time) && (word_flag==0) ){ /* we're inside a word*/
809        if( syl_flag == 0) {
810          type=1;
811          strcpy(label,phone);
812          *start = last_time;
813          *stop = phone_time;
814          *dur = phone_time - last_time;
815          last_time = phone_time;
816          if( (sp=strchr(phone_buf,phone_sep[0])) != NULL){
817            if( (sp=strchr(label,phone_sep[0])) != NULL) *sp = '\0';
818            syl_flag=1;
819          }
820        }else{
821          type=2;
822          syl_flag=0;
823        }
824        if( syl_flag == 0){
825          if( fgets(phone_buf,BUFFSIZE,fphones) == NULL) {
826            if( phone_time == word_time) word_flag=1;
827            else flag = -1;
828          }
829          else sscanf(phone_buf,"%f %d %s",&phone_time,&color,phone);
830        }
831      }else{ /* We passed a word boundary */
832        type=3;
833        *breath=0.0;
834        *pause=0.0;
835        *stop = last_time;
836        strcpy(label,word);
837        while( (!strcmp(phone,"brth")
838               || !strcmp(phone,"sil")
839               || !strcmp(phone,"SIL")
840               || !strcmp(phone,"BRTH"))
```

LINE # SOURCE TEXT

```
841             && (flag==1) ){
842         *pause += phone_time - last_time;
843         if( !strcmp(phone,"brth") || !strcmp(phone,"BRTH"))
844           *breath += phone_time - last_time;
845         last_time = phone_time;
846         if( fgets(phone_buf,BUFFSIZE,fphones) == NULL) flag = -2;
847         else sscanf(phone_buf,"%f %d %s",&phone_time,&color,phone);
848       }
849       if( flag == 1){
850         if( fgets(word_buf,BUFFSIZE,fwords) == NULL) flag = -1;
851         else sscanf(word_buf,"%f %d %s",&word_time,&color,word);
852         while( (word[0] == '<') && (flag == 1) ){
853           if( fgets(word_buf,BUFFSIZE,fwords) == NULL) flag = -1;
854           else sscanf(word_buf,"%f %d %s",&word_time,&color,word);
855         }
856       }
857     }
858     if( flag < 0){
859       flag = 1;
860       fclose(fphones);
861       fclose(fwords);
862       first = -2;
863     }
864   }else{
865     if( first == -1){
866       type=3;
867       *breath=0.0;
868       *pause=0.0;
869       *stop = last_time;
870       strcpy(label,word);
871       first = -2;
872     }else flag = -1;
873   }
874   return(type*flag);
875 }
876 
877 /* Opens an ESPS file and checks some of the basic stuff in the */
878  /* preamble. Searches for the generic header items start_time and */
879  /* record_freq and returns their values (default to 0 and 100, */
880  /* respectively) along with the file stream pointer which is set to */
881  /* the start of the first data record. */
882 
883 FILE *fopen_esps(filename,mode,dstart,dfreq)
884      char *filename, *mode;
885      double *dstart, *dfreq;
886 {
887   FILE *infile;
888   struct {
889     long machine_code;
890     long check_code;
891     long data_offset;
892     long record_size;
893     long check;
894     long edr;
895     long align_pad_size;
896     long foreign_hd;
897   } preamble;
898   int i,j,start,freq;
899   char abc, start_time[16]="start_time",record_freq[16]="record_freq";
900 
901   start=0;
902   freq=0;
903   *dstart = 0;
904   *dfreq=100;
905   /* First, read the preamble, and make sure this is an ESPS file with */
906   /* the right size data records. */
907   if( (infile=fopen(filename,mode)) != NULL){
908     if( fread( (void *) &preamble, sizeof(preamble), 1, infile) != 1)
909       infile=NULL;
910     else{
911       if( (preamble.check != 27162) ){
912         fprintf(stderr,"Wrong Magic Number!!\n");
913         fclose(infile);
914         infile=NULL;
915       }
916       if( preamble.record_size != 48 ){
917         fprintf(stderr,"Wrong record size (%d) in .f0 file!\n",
918                 preamble.record_size);
919         fclose(infile);
920         infile=NULL;
921       }else{
922 
923         /* The basic file header looks okay, now sift through the */
924         /* header looking for the geneeri items we want. */
925         for(i=0;( (i<(preamble.data_offset - 32)) && (start+freq != 2) );i++){
926           abc = getc(infile);
927 
928           j=0;
929           while( (start==0) && (start_time[j++] == abc) && (j<11) ) abc=getc(infile);
930           if(j==11) {
931             for(j=0;j<7;j++) abc=getc(infile);
932             fread(dstart,sizeof(double),1,infile);
933             start=1;
934           }
935           i += j;
936 
937           j=0;
938           while( (freq==0) && (record_freq[j++] == abc) && (j<12) ) abc=getc(infile);
939           if(j==12) {
940             for(j=0;j<6;j++) abc=getc(infile);
941             fread(dfreq,sizeof(double),1,infile);
942             freq=1;
943           }
944           i += j;
945         }
946 
947         /* If we found the start time and record freq and can seek to */
948         /* the start of the data records, we're happy campers! */
949         if((start != 1) || (freq != 1) || (fseek(infile, preamble.data_offset, 0) !=0)) {
950           fclose(infile);
951           infile=NULL;
952         }
953       }
954     }
955   }
956   return(infile);
957 }
958 
959 
960 int get_pitch(basename,stop)
```

LINE # | SOURCE TEXT

```
 961        char *basename;
 962        float stop;
 963  {
 964    struct phone *ppntr;
 965    char filename[128],filebase[128],*sp,*getenv();
 966    float energy,pitch,peak_pitch,f0power,hr;
 967    int flag;
 968    double start,freq;
 969    FILE *fpitch, *ftarg;
 970
 971    if( (sp=getenv("PITCH_PATH")) == NULL) {
 972      strcpy(filename,basename);
 973    }else{
 974      strcpy(filename,sp);
 975      if( (sp=strrchr(basename,'/')) == NULL) {
 976        strcat(filename,"/");
 977        strcat(filename,basename);
 978      }else{
 979        strcat(filename,sp);
 980      }
 981    }
 982    strcpy(filebase,filename);
 983    strcat(filename, PITCH_EXT);
 984    if( (fpitch=fopen_esps(filename,"rb",&start,&freq)) == NULL) {
 985      fprintf(stderr,"Unable to open pitch file %s. Aborting...\n",filename);
 986      exit(-1);
 987    }
 988    pitch_offset = (int) start*freq;
 989    pitch_step = (float) 1.0/freq;
 990
 991    strcpy(filename,filebase); strcat(filename, TARGET_EXT );
 992    if( (ftarg=fopen(filename,"r")) == NULL) {
 993      fprintf(stderr,"Unable to open pitch targets file %s. Aborting...\n",
 994              filename);
 995      exit(-1);
 996    }
 997    ppntr=base_phone;;
 998    while( ppntr->next != NULL ){
 999      ppntr = ppntr->next;
1000      flag = read_pitch(fpitch,ppntr->start,ppntr->stop,
1001                        &energy,&pitch,&peak_pitch,&f0power,&hr);
1002      ppntr->energy = energy;
1003      ppntr->pitch = pitch;
1004      ppntr->f0_power = f0power;
1005      ppntr->hr = hr;
1006    }
1007
1008    num_targets=0;
1009    while( !feof(ftarg) && (num_targets<MAX_TARGETS)){
1010      fscanf(ftarg,"%f %f",
1011             &target_times[num_targets], &pitch_targets[num_targets]);
1012      num_targets++;
1013    }
1014    if( num_targets >= MAX_TARGETS){
1015      fprintf(stderr,"MAX_TARGETS exceeded! Increase and recompile.\n");
1016      flag= -1;
1017    }
1018
1019    fclose(fpitch);
1020    fclose(ftarg);
1021    return(flag);
1022  }
1023
1024  int read_pitch(fpitch,start,stop,energy,pitch,peak_pitch,f0power,hr)
1025        FILE *fpitch;
1026        float start,stop;
1027        float *energy,*pitch,*peak_pitch,*f0power,*hr;
1028  {
1029    double f0_data[8];
1030    float mean_energy,mean_pitch;
1031    int energy_cnt,pitch_cnt,flag;
1032    static int time;
1033    static int first=1,count=0;
1034
1035    flag=1;
1036    mean_energy=0.0;
1037    mean_pitch=0.0;
1038    energy_cnt=0;
1039    pitch_cnt=0;
1040    *pitch = 0.0;
1041    *energy = 0.0;
1042    *peak_pitch=0.0;
1043    *f0power= 0.0;
1044    *hr=0;
1045
1046    if (first==1) {
1047      time=pitch_offset;
1048      first=0;
1049    }
1050    while( (start > ((float)(time+0.01)*pitch_step)) && (flag==1) ) {
1051      if( fread(f0_data,sizeof(double),6,fpitch) != 6) flag= -1;
1052      else {
1053        time++; ;
1054        count++;
1055      }
1056    }
1057
1058    while( (((float)(time+0.01)*pitch_step)<stop) && (flag==1) ) {
1059      if( fread(f0_data,sizeof(double),6,fpitch) != 6) flag= -1;
1060      else {
1061        count++;
1062        mean_energy += (float) f0_data[2];
1063        energy_cnt++;
1064        if (f0_data[1] > 0.75) {
1065          *f0power += f0_data[4];
1066          *hr += f0_data[5];
1067          mean_pitch += (float) f0_data[0];
1068          if( f0_data[0] > *peak_pitch) *peak_pitch = f0_data[0];
1069          pitch_cnt++;
1070        }
1071        time++;
1072      }
1073    }
1074
1075    if( (flag != 1) && ((((float)time*pitch_step)-stop) > 0.01) ) {
1076      fprintf(stderr,"Unexpected end-of-file in pitch data!");
1077      fprintf(stderr,"(time/stop mismatch of %f seconds)\n",
1078              ((float)time*pitch_step)-stop);
1079    }
1080    else {
```

LINE # SOURCE TEXT

```
1081       flag = 1;
1082       if (energy_cnt >0) *energy = mean_energy/energy_cnt;
1083       else {
1084         fprintf(stderr,"Zero frames in phone starting at time %f %d!\n",
1085                start,time);
1086         flag = -1;
1087       }
1088     }
1089     if (pitch_cnt > 0) {
1090       *pitch = mean_pitch/pitch_cnt;
1091       *f0power /= pitch_cnt;
1092       *hr /= pitch_cnt;
1093     }
1094     return(flag);
1095 }
1096
1097
1098
1099 /* normalize_data() coordinates the process of getting (and possibly */
1100 /* updating) the normalization parameters and normalizing the data. */
1101 int normalize_data(filename,func)
1102      char filename[];
1103      int func;
1104 {
1105   struct phone *pntr;
1106   int flag,i;
1107   FILE *fnorm;
1108
1109   /* For starters, if the control code says not to update the */
1110   /* normalization file and we can't open it then we're stuck: */
1111   /* complain and quit. If we get the file open, read it. */
1112   flag=1;
1113   if( (func & 0x010) == 0) {
1114     if( (fnorm=fopen(filename,"r"))==NULL) {
1115       fprintf(stderr,"Unable to open normalization file %s\n",filename);
1116       flag = -1;
1117     }else flag = read_norm(fnorm);
1118
1119   }else{
1120     /* If we're supposed update the normalization parameters then, if */
1121     /* we can't open the file, we just initialize the our tables to */
1122     /* estimate the parameters from scratch. If we can open the file, */
1123     /* then we'll read it in. */
1124     if( (fnorm=fopen(filename,"r"))==NULL) init_norm();
1125     else {
1126       flag=read_norm(fnorm);
1127       fclose(fnorm);
1128     }
1129     /* Now that the estimation tables are initialized, (re)open the */
1130     /* file for writing, update the normalization parameters and write */
1131     /* them back into the file. */
1132     if( (fnorm=fopen(filename,"w"))==NULL) {
1133       fprintf(stderr,"Unable to open normalization file %s\n",filename);
1134       flag = -1;
1135     }else{
1136       update_norm();
1137       flag=write_norm(fnorm);
1138     }
1139   }
1140
1141   /* Now that we have the normalization parameters, we may wan't to */
1142   /* adapt the data. That is, we may want to shift it in some way to */
1143   /* improve the agreement between it and the data used in estimating */
1144   /* the normalization parameters. This is under user control via the */
1145   /* control code. */
1146   if( flag==1) {
1147     fclose(fnorm);
1148     if( (func & 0x020) != 0) adapt_dur();
1149     if( (func & 0x040) != 0) adapt_energy();
1150     if( (func & 0x080) != 0) adapt_pitch();
1151   }
1152
1153   /* Okay, now we can actually normalize the data! We do this by going */
1154   /* through each phone and computing a Z-score for duration, energy */
1155   /* and (maybe) pitch. This normalization is done simply by */
1156   /* subtracting off the phone-dependent mean and dividing by the */
1157   /* phone-dependent variance. Of course, we try to avoid dividing by */
1158   /* zero and, if we encounter any phones for which we don't have */
1159   /* parameters, we alert the user and assign nominal values to their */
1160   /* features. */
1161   pntr=base_phone;
1162   while( pntr->next != NULL){
1163     pntr=pntr->next;
1164     i=0;
1165     while( (i<nphones) && strcmp(pntr->label,ntable[i].phone) ) i++;
1166     if( i<nphones) {
1167       if( ntable[i].dur2 > 0.0)
1168         pntr->dur = (pntr->dur - ntable[i].dur)/ntable[i].dur2;
1169       else pntr->dur = 1.0;
1170       if( ntable[i].energy2 > 0.0)
1171         pntr->energy = (pntr->energy -
1172                     ntable[i].energy)/ntable[i].energy2;
1173       else pntr->energy = 1.0;
1174 /*      if( ntable[i].pitch2 > 0.0)
1175         pntr->pitch = (pntr->pitch -
1176                     ntable[i].pitch)/ntable[i].pitch2;
1177       else pntr->pitch = 1.0;   */
1178     }else {
1179       fprintf(stderr,"WARNING: phone %s not in normalization file\n",
1180               pntr->label);
1181       pntr->dur=1.0;
1182       pntr->energy=1.0;
1183       pntr->pitch=0.0;
1184     }
1185   }
1186   return(flag);
1187 }
1188
1189 void adapt_dur()
1190 {
1191   struct phone *pntr;
1192   float alpha;
1193   int cntr,i;
1194   float lambda[NPHONES],r[NPHONES],r_bar,sums[NPHONES];
1195   int counts[NPHONES];
1196
1197 /* The following code is for the maximimum-likelihood estimate of alpha. */
1198 /* cntr=0;
1199   alpha=0.0;
1200   for(i=0;i<nphones;i++){
```

LINE # SOURCE TEXT

```
1201      sums[i]=0.0;
1202      counts[i]=0;
1203      if(ntable[i].dur2 > 0.0)
1204        r[i] = (ntable[i].dur/ntable[i].dur2)*(ntable[i].dur/ntable[i].dur2);
1205      else r[i] = 0.0;
1206      lambda[i] = r[i]/ntable[i].dur;
1207    }
1208    pntr=base_phone;
1209    while( pntr->next != NULL) {
1210      pntr=pntr->next;
1211      i=0;
1212      while( (i<nphones) && strcmp(pntr->label,ntable[i].phone) ) i++;
1213      if( i<nphones) {
1214        sums[i] += pntr->dur;
1215        counts[i] += 1;
1216      }else fprintf(stderr,"WARNING: phone %s not in normalization file\n",
1217                pntr->label);
1218    }
1219    r_bar=0.0;
1220    for(i=0;i<nphones;i++) r_bar += counts[i] * r[i];
1221    for(i=0;i<nphones;i++) alpha += (r[i]/(r_bar*ntable[i].dur))*sums[i];
1222    fprintf(stderr,"Adapting duration: alpha = %f\n",alpha);
1223    for(i=0;i<nphones;i++){
1224      ntable[i].dur *= alpha;
1225      ntable[i].dur2 *= alpha;
1226    }
1227  */
1228    /* the following is code for the old way of adapting durations */
1229    cntr=0;
1230    alpha=0.0;
1231    pntr=base_phone;
1232    while( pntr->next != NULL) {
1233      pntr=pntr->next;
1234      i=0;
1235      while( (i<nphones) && strcmp(pntr->label,ntable[i].phone) ) i++;
1236      if( i<nphones) {
1237        alpha += pntr->dur/ntable[i].dur;
1238        cntr++;
1239      }else fprintf(stderr,"WARNING: phone %s not in normalization file\n",
1240                pntr->label);
1241    }
1242    if (cntr>0) alpha /= cntr;
1243    fprintf(stderr,"Adapting duration: alpha = %f\n",alpha);
1244    for(i=0;i<nphones;i++){
1245      ntable[i].dur *= alpha;
1246      ntable[i].dur2 *= alpha;
1247    }
1248  }
1249
1250  void adapt_energy()
1251  {
1252    struct phone *pntr;
1253    float alpha;
1254    int cntr,i;
1255
1256    cntr=0;
1257    alpha=0.0;
1258    pntr=base_phone;
1259    while( pntr->next != NULL) {
1260      pntr=pntr->next;
1261      i=0;
1262      while( (i<nphones) && strcmp(pntr->label,ntable[i].phone) ) i++;
1263      if( i<nphones) {
1264        alpha += pntr->energy - ntable[i].energy;
1265        cntr++;
1266      }else fprintf(stderr,"WARNING: phone %s not in normalization file\n",
1267                pntr->label);
1268    }
1269    if (cntr>0) alpha /= cntr;
1270    fprintf(stderr,"Adapting energy: offset = %f\n",alpha);
1271    for(i=0;i<nphones;i++){
1272      ntable[i].energy += alpha;
1273    }
1274  }
1275
1276  void adapt_pitch()
1277  {
1278    struct phone *pntr;
1279    float alpha;
1280    int cntr,i;
1281
1282    cntr=0;
1283    alpha=0.0;
1284    pntr=base_phone;
1285    while( pntr->next != NULL) {
1286      pntr=pntr->next;
1287      i=0;
1288      while( (i<nphones) && strcmp(pntr->label,ntable[i].phone) ) i++;
1289      if( i<nphones) {
1290        alpha += pntr->pitch - ntable[i].pitch;
1291        cntr++;
1292      }else fprintf(stderr,"WARNING: phone %s not in normalization file\n",
1293                pntr->label);
1294    }
1295    if (cntr>0) alpha /= cntr;
1296    fprintf(stderr,"Adapting pitch: offset = %f\n",alpha);
1297    for(i=0;i<nphones;i++){
1298      ntable[i].pitch += alpha;
1299    }
1300  }
1301
1302
1303  void update_norm()
1304  {
1305    struct phone *pntr;
1306    int i;
1307
1308    for(i=0;i<nphones;i++){
1309      ntable[i].dur2 = ntable[i].count *
1310        ( (ntable[i].dur2)*(ntable[i].dur2) +
1311         (ntable[i].dur)*(ntable[i].dur) );
1312      ntable[i].energy2 = ntable[i].count *
1313        ( (ntable[i].energy2)*(ntable[i].energy2) +
1314         (ntable[i].energy)*(ntable[i].energy) );
1315      ntable[i].pitch2 = ntable[i].count *
1316        ( (ntable[i].pitch2)*(ntable[i].pitch2) +
1317         (ntable[i].pitch)*(ntable[i].pitch) );
1318      ntable[i].dur *= ntable[i].count;
1319      ntable[i].energy *= ntable[i].count;
1320      ntable[i].pitch *= ntable[i].count;
```

LINE # | SOURCE TEXT

```
1321     }
1322
1323     pntr=base_phone;
1324     while( pntr->next != NULL) {
1325       pntr=pntr->next;
1326       i=0;
1327       while( (i<nphones) && strcmp(pntr->label,ntable[i].phone) ) i++;
1328       if( i >= NPHONES ) {
1329         i = NPHONES;
1330         fprintf(stderr,"More than %d phone labels found: increase NPHONES.\n",
1331                 NPHONES);
1332       }else if( i>= nphones) {
1333         strcpy(ntable[i].phone,pntr->label);
1334         nphones+=1;
1335       }
1336       ntable[i].count++;
1337       ntable[i].dur += pntr->dur;
1338       ntable[i].dur2 += (pntr->dur)*(pntr->dur);
1339       ntable[i].energy += pntr->energy;
1340       ntable[i].energy2 += (pntr->energy)*(pntr->energy);
1341       ntable[i].pitch += pntr->pitch;
1342       ntable[i].pitch2 += (pntr->pitch)*(pntr->pitch);
1343     }
1344
1345     for(i=0;i<nphones;i++){
1346       ntable[i].dur /= ntable[i].count;
1347       ntable[i].energy /= ntable[i].count;
1348       ntable[i].pitch /= ntable[i].count;
1349       ntable[i].dur2 /= ntable[i].count;
1350       ntable[i].energy2 /= ntable[i].count;
1351       ntable[i].pitch2 /= ntable[i].count;
1352
1353       ntable[i].dur2 = ntable[i].dur2 - (ntable[i].dur * ntable[i].dur);
1354       if (ntable[i].dur2 > 1e-6) ntable[i].dur2 = sqrt(ntable[i].dur2);
1355       else ntable[i].dur2 = 0.0;
1356
1357       ntable[i].energy2 = ntable[i].energy2-(ntable[i].energy*ntable[i].energy);
1358       if (ntable[i].energy2 > 1e-6) ntable[i].energy2 = sqrt(ntable[i].energy2);
1359       else ntable[i].energy2 = 0.0;
1360
1361       ntable[i].pitch2 = ntable[i].pitch2 - (ntable[i].pitch * ntable[i].pitch);
1362       if (ntable[i].pitch2 > 1e-6) ntable[i].pitch2 = sqrt(ntable[i].pitch2);
1363       else ntable[i].dur2 = 0.0;
1364     }
1365   }
1366
1367   void init_norm()
1368   {
1369     int i;
1370
1371     nphones = 0;
1372     for(i=0;i<NPHONES;i++){
1373       ntable[i].phone[0]='\0';
1374       ntable[i].count=0;
1375       ntable[i].dur=0.0;
1376       ntable[i].dur2=0.0;
1377       ntable[i].energy=0.0;
1378       ntable[i].energy2=0.0;
1379       ntable[i].pitch=0.0;
1380       ntable[i].pitch2=0.0;
1381     }
1382   }
1383
1384   int read_norm(fnorm)
1385        FILE *fnorm;
1386   {
1387     int flag,i;
1388
1389     flag=1;
1390     i=0;
1391     if( fscanf(fnorm,"%d",&nphones) == EOF) flag = -2;
1392     while( (i<nphones) && (flag==1) ){
1393       if( fscanf(fnorm,"%s %d %f %f %f %f %f %f\n",ntable[i].phone,
1394             &ntable[i].count,
1395             &ntable[i].dur,
1396             &ntable[i].dur2,
1397             &ntable[i].energy,
1398             &ntable[i].energy2,
1399             &ntable[i].pitch,
1400             &ntable[i].pitch2 ) == EOF) flag= -1;
1401       i++;
1402     }
1403     if( flag != 1) fprintf(stderr,"I/O error while reading normalization!\n");
1404     return(flag);
1405   }
1406
1407   int write_norm(fnorm)
1408        FILE *fnorm;
1409   {
1410     int flag,i,j,max_i,max_cnt,counts[NPHONES];
1411
1412     flag=1;
1413     if( fprintf(fnorm,"%d\n",nphones) == EOF) flag =-1;
1414
1415     for(i=0;i<nphones;i++) counts[i] = ntable[i].count;
1416     j=0;
1417     while( (j<nphones) && (flag==1) ){
1418       max_i=0;
1419       max_cnt=0;
1420       i=0;
1421       while( (i<nphones) && (flag==1) ){
1422         if( counts[i] > max_cnt ) {
1423           max_i=i;
1424           max_cnt=counts[i];
1425         }
1426         i++;
1427       }
1428       if( fprintf(fnorm,"%-6s %5d %10.8f %10.8f %10.5f %10.5f %10.5f %10.5f\n",
1429                   ntable[max_i].phone,
1430                   ntable[max_i].count,
1431                   ntable[max_i].dur,
1432                   ntable[max_i].dur2,
1433                   ntable[max_i].energy,
1434                   ntable[max_i].energy2,
1435                   ntable[max_i].pitch,
1436                   ntable[max_i].pitch2 ) == EOF) flag= -1;
1437       counts[max_i] = 0;
1438       j++;
1439     }
1440     if( flag != 1) fprintf(stderr,"I/O error while writing normalization!\n");
```

LINE # SOURCE TEXT

```
1441      return(flag);
1442  }
1443
1444
1445  int pass1_syl()
1446  {
1447    struct syl *spntr;
1448    struct phone *ppntr;
1449    struct word *wpntr;
1450    int dcnt,ecnt,end_syl,end,scnt;
1451    int tcnt,pcnt;
1452    float end_time;
1453
1454    tcnt=1; /* skip the first pitch target...makes the edge condition trivial */
1455    end=0;
1456    scnt=0;
1457    global_mean_pitch = 0.0;
1458    spntr=base_syl;
1459    wpntr=base_word;
1460
1461    while( end == 0) {
1462      ppntr=spntr->phones;
1463      spntr->max_pitch = 0.0;
1464      spntr->min_pitch = 10000.0;
1465      spntr->last_target = 0.0;
1466      end_syl=0;
1467      dcnt=0;
1468      ecnt=0;
1469      pcnt=0;
1470      spntr->ntargs = 0;
1471      while( target_times[tcnt] < spntr->start) tcnt++;
1472      end_time = spntr->stop;
1473      if( spntr->last == 1) {
1474        end_time += wpntr->sil;
1475        wpntr=wpntr->next;
1476      }
1477      while( (tcnt<num_targets) && (target_times[tcnt] <= spntr->stop) ){
1478        spntr->ntargs++;
1479        spntr->last_target = pitch_targets[tcnt];
1480        if( pitch_targets[tcnt] > spntr->max_pitch)
1481          spntr->max_pitch = pitch_targets[tcnt];
1482        if( pitch_targets[tcnt] < spntr->min_pitch)
1483          spntr->min_pitch = pitch_targets[tcnt];
1484        if( spntr->ntargs == 1) {
1485          if( pitch_targets[tcnt-1] < pitch_targets[tcnt] &&
1486             pitch_targets[tcnt] > pitch_targets[tcnt+1]) spntr->shape=1; /* H */
1487          else if( pitch_targets[tcnt-1] > pitch_targets[tcnt] &&
1488             pitch_targets[tcnt] < pitch_targets[tcnt+1]) spntr->shape=2; /* L */
1489          else if( pitch_targets[tcnt-1] < pitch_targets[tcnt] &&
1490             pitch_targets[tcnt] < pitch_targets[tcnt+1]) spntr->shape=3; /* U */
1491          else if( pitch_targets[tcnt-1] > pitch_targets[tcnt] &&
1492             pitch_targets[tcnt] > pitch_targets[tcnt+1]) spntr->shape=4; /* D */
1493        }else if(spntr->ntargs == 2){
1494          if( pitch_targets[tcnt-1] < pitch_targets[tcnt] &&
1495             pitch_targets[tcnt] > pitch_targets[tcnt+1])
1496            spntr->shape += 4; /* H */
1497          else if( pitch_targets[tcnt-1] > pitch_targets[tcnt] &&
1498                  pitch_targets[tcnt] < pitch_targets[tcnt+1])
1499            spntr->shape += 4; /* L */
1500          if( spntr->shape > 6) spntr->shape -= 6;
1501        }else spntr->shape=7;
1502        tcnt++;
1503      }
1504      if( spntr->min_pitch >500.0) spntr->min_pitch=0.0;
1505
1506      while( ppntr != spntr->nucleus) {
1507        spntr->mean_energy += ppntr->energy;
1508        if(ppntr->pitch>0.0){
1509          spntr->mean_pitch += ppntr->pitch;
1510          pcnt++;
1511        }
1512        spntr->mean_dur += ppntr->dur;
1513        ecnt++;
1514        spntr->onset += ppntr->dur;
1515        ppntr=ppntr->next;
1516      }
1517      if( ecnt > 0) spntr->onset /= ecnt;
1518      ppntr = spntr->nucleus;
1519      spntr->f0_power = ppntr->f0_power;
1520      spntr->hr = ppntr->hr;
1521      while( end_syl == 0){
1522        spntr->mean_energy += ppntr->energy;
1523        if(ppntr->pitch>0.0){
1524          spntr->mean_pitch += ppntr->pitch;
1525          pcnt++;
1526        }
1527        spntr->mean_dur += ppntr->dur;
1528        ecnt++;
1529        spntr->rhyme += ppntr->dur;
1530        dcnt++;
1531        if( ppntr->last == 1) end_syl=1;
1532        else ppntr=ppntr->next;
1533      }
1534      spntr->mean_energy /= ecnt;
1535      if(pcnt>0) spntr->mean_pitch /= pcnt;
1536      spntr->mean_dur /= ecnt;
1537      spntr->rhyme /= dcnt;
1538
1539      if(pcnt>0){
1540        global_mean_pitch += spntr->mean_pitch;
1541        scnt++;
1542      }
1543      spntr=spntr->next;
1544      if( spntr->next == NULL) end=1;
1545    }
1546    global_mean_pitch /= scnt;
1547    /* fprintf(stderr,"Global mean pitch = %f\n",global_mean_pitch); */
1548    return(1);
1549  }
1550
1551  int pass2_syl()
1552  {
1553    struct syl *pntr,*last,*next_syl;
1554    int end;
1555
1556    pntr = base_syl;
1557    end=0;
1558
1559    pntr->del_dur = 0.0;
1560    pntr->del_energy = 0.0;
```

LINE # SOURCE TEXT

```
1561     pntr->del_pitch = 0.0;
1562     last = pntr;
1563     pntr = pntr->next;
1564     if( pntr->next != NULL) next_syl = pntr->next;
1565     else next_syl = pntr;
1566     while( next_syl->next != NULL){
1567       if( last->ntargs > 0) pntr->ptarg=1;
1568       else pntr->ptarg=0;
1569       if( next_syl->ntargs > 0) pntr->ntarg=1;
1570       else pntr->ntarg=0;
1571       if(last->shape==1) pntr->ph=1;
1572       else pntr->ph=0;
1573       if(next_syl->shape==1) pntr->nh=1;
1574       else pntr->nh=0;
1575       pntr->del_dur = last->mean_dur - next_syl->mean_dur;
1576       pntr->del_energy = last->mean_energy - next_syl->mean_energy;
1577       pntr->del_hr = pntr->hr - 0.5*(last->hr + next_syl->hr);
1578 /* This block should be used if you aren't normalizing pitch */
1579       pntr->del_pitch = p_ratio(last->mean_pitch,next_syl->mean_pitch);
1580 /* This block should be used with normalized pitch */
1581 /*     pntr->del_pitch = last->mean_pitch - next_syl->mean_pitch; */
1582
1583
1584       last=pntr;
1585       pntr=next_syl;
1586       next_syl=pntr->next;
1587     }
1588     pntr->del_dur = 0.0;
1589     pntr->del_energy = 0.0;
1590     pntr->del_pitch = 0.0;
1591     pntr->del_hr = 0.0;
1592     return(1);
1593 }
1594
1595 int pass1_words()
1596 {
1597     struct word *wpntr;
1598     struct syl *spntr;
1599     int pcnt,end;
1600     float wmax,wmin;
1601
1602     wpntr = base_word;
1603     while( wpntr->last != 1){
1604       spntr = wpntr->syls;
1605       wmax=0.0;
1606       wmin=10000.0;
1607       end=0;
1608       pcnt=0;
1609       while( end==0){
1610         if( spntr->max_pitch > wmax) wmax=spntr->max_pitch;
1611         if( spntr->min_pitch < wmin) wmin=spntr->min_pitch;
1612         if( spntr->mean_pitch > 0.0){
1613           pcnt++;
1614           wpntr->mean_pitch += spntr->mean_pitch;
1615         }
1616         if( spntr->last == 1) end=1;
1617         else spntr = spntr->next;
1618       }
1619       wpntr->max_pitch = wmax;
1620       if(wmin <500.0) wpntr->min_pitch = wmin;
1621       else wpntr->min_pitch = 0.0;
1622       if(pcnt>0) wpntr->mean_pitch /= pcnt;
1623       wpntr = wpntr->next;
1624     }
1625
1626     return(1);
1627 }
1628
1629 int pass2_words()
1630 {
1631     return(1);
1632 }
1633
1634 struct phone *new_phone()
1635 {
1636     struct phone *pntr;
1637
1638     pntr = (struct phone *) malloc(sizeof(struct phone));
1639     if (pntr != NULL) {
1640       pntr->next      = NULL;
1641       pntr->last      = 0;
1642       pntr->label[0]  = '\0';
1643       pntr->start     = 0.0;
1644       pntr->stop      = 0.0;
1645       pntr->dur       = 0.0;
1646       pntr->energy    = 0.0;
1647       pntr->pitch     = 0.0;
1648     }
1649     return(pntr);
1650 }
1651
1652 struct syl *new_syl()
1653 {
1654     struct syl *pntr;
1655
1656     pntr = (struct syl *) malloc(sizeof(struct syl));
1657     if (pntr != NULL) {
1658       pntr->next      = NULL;
1659       pntr->last      = 0;
1660       pntr->phones    = NULL;
1661       pntr->nucleus   = NULL;
1662       pntr->stress    = 0;
1663       pntr->start     = 0.0;
1664       pntr->stop      = 0.0;
1665       pntr->dur       = 0.0;
1666       pntr->energy    = 0.0;
1667       pntr->max_pitch     = 0.0;
1668       pntr->rhyme     = 0.0;
1669       pntr->onset     = 0.0;
1670       pntr->mean_dur = 0.0;
1671       pntr->mean_pitch = 0.0;
1672       pntr->mean_energy = 0.0;
1673     }
1674     return(pntr);
1675 }
1676
1677 struct word *new_word()
1678 {
1679     struct word *pntr;
1680
```

LINE # SOURCE TEXT

```
1681    pntr = (struct word *) malloc(sizeof(struct word));
1682    if (pntr != NULL) {
1683      pntr->next     = NULL;
1684      pntr->last     = 0;
1685      pntr->nsyl     = 0;
1686      pntr->syls     = NULL;
1687      pntr->sil      = 0.0;
1688      pntr->brth     = 0.0;
1689    }
1690    return(pntr);
1691  }
1692
1693  int free_data()
1694  {
1695    struct phone *ppntr;
1696    struct syl *spntr;
1697    struct word *wpntr;
1698    int okay;
1699
1700    okay=1;
1701
1702    while( (okay==1) && (base_phone!=NULL) ) {
1703      ppntr = base_phone->next;
1704      okay = free(base_phone);
1705      base_phone = ppntr;
1706    }
1707
1708    while( (okay==1) && (base_syl!=NULL) ) {
1709      spntr = base_syl->next;
1710      okay = free(base_syl);
1711      base_syl = spntr;
1712    }
1713
1714    while( (okay==1) && (base_word!=NULL) ) {
1715      wpntr = base_word->next;
1716      okay = free(base_word);
1717      base_word = wpntr;
1718    }
1719
1720    if (okay!=1)
1721      fprintf(stderr,"Unable to free dynamic data structures!\n");
1722    return(okay);
1723  }
1724
1725  int stressvowel(phone)
1726       char phone[];
1727  {
1728    static  char vowels[5] = {'a','e','i','o','u'};
1729    char *index();
1730    int flag,i;
1731
1732    flag=0;
1733    for (i=0;i<5;i++) if (phone[0]==vowels[i]) flag=1;
1734    if ((flag==1)&&(index(phone,'+')!=NULL)) flag=2;
1735    return(flag);
1736  }
1737
1738  float p_ratio(arg1,arg2)
1739       float arg1,arg2;
1740  {
1741
1742    if( arg2 > 0.0) return(arg1/arg2);
1743    else return(1.0);
1744  }
1745
1746
```

LINE # HEADER TEXT

```
1  #include "models.h" /* has evrything shared between autolabel and Makemodels */
2
3  #define WORDS_EXT ".words"    /* name extension for aligner words file */
4  #define PHONES_EXT ".phones"     /* name extension for aligner phones file */
5  #define PITCH_EXT ".f0" /* name extension for output of get_f0 */
6  #define TARGET_EXT ".tgts" /* name extension for pitch targets file */
7  #define NPHONES 100
8  #define MAX_TARGETS 1024 /* Maximum number of pitch targets */
9  #define BUFFSIZE 128  /* Maximum number of chars in a line or filename */
10 #define WORDSIZE 64   /* Maximum number of characters in a word label */
11 #define PHONESIZE 8   /* Maximum number of characters in a phone label */
12
13 #include "autolabel_structs.h"
14
15
16 /* GLOBAL DATA STRUCTURES */
17 struct phone *base_phone;
18 struct syl *base_syl;
19 struct word *base_word;
20 struct pnorm ntable[NPHONES];
21 int nphones,syl_count;
22 float global_mean_pitch;
23 int num_targets;
24 int pitch_offset;
25 float pitch_step;
26 float pitch_targets[MAX_TARGETS], target_times[MAX_TARGETS];
27
28
29 /* FUNCTION PROTOTYPES */
30 int extract_features();
31 int label_data();
32 int tree_vq();
33 int read_data();
34       int read_file();
35       float read_aligner();
36       int get_pitch();
37       int read_pitch();
38 int normalize_data();
39       int read_norm();
40       void init_norm();
41       void update_norm();
42       int write_norm();
43       void adapt_dur();
44       void adapt_energy();
45       void adapt_pitch();
46 int pass1_syl();
47 int pass2_syl();
48 int pass1_words();
49 int pass2_words();
50 int write_data();
51 int free_data();
52 struct phone *new_phone();
53 struct syl *new_syl();
54 struct word *new_word();
55 float p_ratio();
56
```

LINE # SOURCE TEXT

```
1   #include <stdio.h>
2   #include <math.h>
3
4   #include "makemodels.h"  /* Contains structures and constant declarations */
5
6   /***********************************************************************/
7     /*  Main Program: */
8     /*        This is the main calling sequence for the tree quantizer */
9     /*        design program. It implements the design by using a greedy */
10    /*        growing algorithm to produce a large tree which is then */
11    /*        pruned using optimal pruning and the minimum entropy */
12    /*        criterion. Note that this produces a tree in which the */
13    /*        mutual information between the classes and the codewords */
14    /*        is maximized. */
15
16
17  main(argc,argv)
18       int argc;
19       char *argv[];
20  {
21    struct node *tree, *cut_here;
22    struct datum *growdata, *prunedata;
23    struct leafpntr *leaflist;
24    float impurity, change, lentrpy, logp, lastLogp;
25    int i,nnodes,npnts, nprune;
26
27    /* CALLING ARGUMENTS:  the program should be called with two */
28    /* arguments. These should be the names of the files containing the */
29    /* training observation vectors and the training labels. Note that */
30    /* it is not necessary to divide the data into two sets (for growing */
31    /* and pruning); the program will do that automatically. We start by */
32    /* checking to make sure the user provided the two arguments */
33
34    if (argc!=3) {
35      fprintf(stderr,"usage: %s training.observs training.labels\n",argv[0]);
36      exit(-1);
37    }
38
39    /* INITIALIZATION: we start by creating a one-node tree, and putting */
40    /* that node on the list of leaf nodes. */
41
42    tree = newnode();
43    leaflist = newleafpntr();
44    leaflist->nextleaf=NULL;
45    leaflist->leafnode=tree;
46    for(i=0;i<NQUESTIONS;i++) reductions[i]=0.0;
47
48    /* Next, we'll read the observations and labels. Note that the */
49    /* getdata routine breaks the data into two subsets and returns */
50    /* counts of the number of points in each subset, as well as */
51    /* pointers to each list of points. We attach the list of points for */
52    /* growing the tree to the one node we have, find the probability of */
53    /* each class at that node, and compute the entropy. Since the root */
54    /* is the only node a this point, its entropy is also the total */
55    /* impurity in the tree. */
56
57    if ( getdata(argv[1],&growdata,&npnts,argv[2],&prunedata,&nprune) == NULL ) {
58      fprintf(stderr,"Failed to initialize dataset, exiting...\n"); exit(-1); }
59    tree->data=growdata;
60    tree->ndata=npnts;
61    getpclasses(tree);
62    tree->entropy=entropy(tree);
63    impurity=tree->entropy;
64    fprintf(stderr,"%d items read: dataset entropy = %f (nats)\n"
65            ,npnts+nprune,impurity);
66    fprintf(stderr,"Assigned %d points for growing, %d points for pruning.\n"
67            ,npnts,nprune);
68
69    /* GROWING THE TREE: This is the basic training loop. We repeatedly */
70    /* pass the list of leaf nodes to the divide function which tries to */
71    /* split the node which will result in the largest change in */
72    /* impurity. This is the greedy growing algorithm, and when no more */
73    /* change is possible, we're done. */
74
75    float_buf = (float *) malloc( (npnts+nprune)*sizeof(float) );
76    class_buf = (int *) malloc( (npnts+nprune)*sizeof(int) );
77    change=1.0;
78    i=0;
79    while ( (change > PURE) && (i < (MAX_LEAVES - 1)) ) {
80      fprintf(stderr,"Division %d: ",++i);
81      change = divide(leaflist,npnts);
82      impurity -= change;
83      if (change!=0)
84        fprintf(stderr,"entropy = %f\n",impurity);
85    }
86    fprintf(stderr,"no further splits.\n");
87    /* treeprint(tree); */
88
89    /* This is clean up. We need to recover the data used to grow the */
90    /* tree so we strip it off the leaves and get back to one big list. */
91    /* Also, the pruning process will change which nodes are leaves so */
92    /* we can trash the list of leaves in anticipation of building a new */
93    /* one. */
94
95    growdata=strip_leaves(leaflist);
96    freeleaves(leaflist);
97    leaflist=NULL;
98    fprintf(stderr,"%d datums recovered\n\n",count(growdata));
99
100   /* SMOOTHING THE TREE */
101
102   fprintf(stderr,"Initializing smoothing parameters...");
103   make_buckets(tree);
104   fprintf(stderr,"Done.\n");
105   /* treeprint(tree); */
106   lastLogp = (float) exp( (double) re_estimate(tree,prunedata));
107   fprintf(stderr,"mean Pr{smoothing data} = %f\n",lastLogp);
108   while(
109         -(lastLogp - (logp=(float) exp((double)re_estimate(tree,prunedata))))
110         > 0.001) {
111     fprintf(stderr,"mean Pr{smoothing data} = %f\n",logp);
112     lastLogp=logp;
113   }
114   fprintf(stderr,"Smoothing probabilities...");
115   for(i=0;i<NCLASS;i++)
116     tree->pclass[i] = (lambda[tree->bucket]*tree->pclass[i])
117       + ((1-lambda[tree->bucket])*(1.0/((float)NCLASS)));
118   smooth_probs(tree);
119   fprintf(stderr,"Done.\n");
120   /* treeprint(tree); */
```

LINE # SOURCE TEXT

```
121    logp = (float) exp( (double) objective_function(tree,prunedata));
122    fprintf(stderr,"\nFinal mean Pr{smoothing data} = %f\n\n",logp);
123
124    /* treeprint(tree); */
125    dump_model(tree);
126    for(i=0;i<NQUESTIONS;i++)
127      fprintf(stderr,"Entropy reduction due to question %d:  %f\n"
128              ,i,reductions[i]);
129    fprintf(stderr,"Freeing data structures.\n");
130    freeleaves(leaflist);
131    leaflist=NULL;
132    freetree(tree);
133    free(class_buf);
134    free(float_buf);
135
136  }
137
138  float objective_function(nodep, datap)
139       struct node *nodep;
140       struct datum *datap;
141  {
142    struct datum *dpntr;
143    float prob,logp;
144    int i,cntr;
145
146    dpntr = datap;
147    logp = 0.0;
148    cntr=0;
149    while( dpntr != NULL) {
150      prob = classify(nodep,dpntr);
151      /* if(cntr<2) fprintf(stderr,"Pr{datum %d} = %f \n",cntr++,prob);  */
152      logp += (float) log( (double) prob );
153      dpntr = dpntr->link;
154      cntr++;
155    }
156    return(logp/((float) cntr));
157  }
158
159  float re_estimate(nodep, datap)
160       struct node *nodep;
161       struct datum *datap;
162  {
163    struct datum *dpntr;
164    struct node *npntr;
165    float logp,total,lam,maxl,minl;
166    double incrP[MAX_LEAVES],nodeProb[MAX_LEAVES],prob,prodincr;
167    int i,b,cntr,bucket[MAX_LEAVES],depth;
168
169    for( i=0;i<num_buckets;i++) {
170      c_b[i]=0.0;
171      d_b[i]=0.0;
172    }
173    logp=0.0;
174    total=0.0;
175
176    dpntr = datap;
177    cntr=1;
178    while( dpntr != NULL) {
179      npntr=nodep;
180      prob = 1.0/((float) NCLASS);
181      incrP[0] = prob;
182      /* if( cntr==1) fprintf(stderr,"depth= -1, prob=%f\n",(float)prob);
183         if( cntr==1) fprintf(stderr,"class=%d\n",dpntr->class); */
184      for(depth=0;npntr!=NULL;depth++){    /*forward-pass */
185        nodeProb[depth] = npntr->pclass[dpntr->class];
186        bucket[depth] = npntr->bucket;
187        lam = lambda[bucket[depth]];
188        incrP[depth+1] = (nodeProb[depth]*lam) + ( (1-lam)*prob);
189        prob = incrP[depth+1];
190        if( npntr->leaf != 1) {
191          if( TEST(dpntr->obsrv,npntr->qid,npntr->thresh)){
192            /* fprintf(stderr,"true\n"); */
193            npntr = npntr->tchild;
194          }
195          else {
196            /* fprintf(stderr,"false\n"); */
197            npntr = npntr->fchild;
198          }
199        }else npntr=NULL;
200        /* if( cntr==1)
201             fprintf(stderr,"depth=%d, prob=%f\n",depth,(float)prob); */
202      }
203      prodincr=1.0;
204      for(i=(depth-1);i>=0;i--){     /*backward-pass */
205        b=bucket[i];
206        lam=lambda[b];
207        c_b[b] += lam*nodeProb[i]*prodincr/prob;
208        prodincr *= (1.0 - lam);
209        d_b[b] += incrP[i]*prodincr/prob;
210      }
211      cntr++;
212      total += 1.0;
213      logp += (float) log(prob);
214      dpntr = dpntr->link;
215    }
216
217    minl=2.0;
218    maxl=0.0;
219    for(b=0;b<num_buckets;b++){
220      if( (d_b[b] + c_b[b]) > 0.0 ){
221        lambda[b] = (c_b[b] + 1.0)/(d_b[b]+c_b[b]+2.0);
222        if( lambda[b] > MAX_LAMBDA) lambda[b] = MAX_LAMBDA;
223        if( lambda[b] >maxl) maxl = lambda[b];
224        if( lambda[b] <minl) minl = lambda[b];
225      }
226    }
227    fprintf(stderr,"max lambda = %f, Minimum = %f\n",maxl,minl);
228    return(logp/total);
229  }
230
231  float classify(nodep, datap)
232       struct node *nodep;
233       struct datum *datap;
234  {
235    float prob;
236
237    if( nodep->leaf == 1)
238      prob = nodep->pclass[datap->class];
239    else {
240      if( TEST(datap->obsrv,nodep->qid,nodep->thresh))
```

LINE # SOURCE TEXT

```
241         prob = classify(nodep->tchild,datap);
242       else
243         prob = classify(nodep->fchild,datap);
244     }
245     return(prob);
246   }
247
248
249
250   /* smooth_probs smooths the output probabilities at the childeren of the */
251   /* the node. */
252   void smooth_probs(nodep)
253         struct node *nodep;
254   {
255     struct node *child;
256     float lam;
257     int i;
258
259
260     if( nodep->leaf != 1) {
261       child = nodep->tchild;
262       lam = lambda[child->bucket];
263       /* fprintf(stderr,"lambda[%d]=%f\n",child->bucket,lam); */
264       for( i=0; i<NCLASS; i++)
265         child->pclass[i] =
266           (lam * child->pclass[i]) + ( (1-lam)*nodep->pclass[i]);
267       smooth_probs(child);
268       child = nodep->fchild;
269       lam = lambda[child->bucket];
270       /* fprintf(stderr,"lambda[%d]=%f\n",child->bucket,lam); */
271       for( i=0; i<NCLASS; i++)
272         child->pclass[i] =
273           (lam * child->pclass[i]) + ( (1-lam)*nodep->pclass[i]);
274       smooth_probs(child);
275     } /* else{
276       for(i=0;i<NCLASS;i++) fprintf(stderr,"%f ",nodep->pclass[i]);
277       fprintf(stderr,"\n");
278     } */
279   }
280
281
282   void make_buckets(nodep)
283         struct node *nodep;
284   {
285     int *sort_buf,cntr,i,num_pnts;
286     struct leafpntr *node_list,*lpntr;
287
288     /* allocate the bucket buffers and initialize all the lambdas to 0.5 */
289     num_pnts=0;
290     fprintf(stderr,"\nMaking nodelist...");
291     node_list = make_nodelist(nodep,&num_pnts);
292     num_buckets = 1 + countnodes(nodep);
293     c_b = (float *) malloc( num_buckets * sizeof(float));
294     d_b = (float *) malloc( num_buckets * sizeof(float));
295     lambda = (float *) malloc( num_buckets * sizeof(float));
296     for(i=0;i<num_buckets;i++) lambda[i] = 0.5;
297     fprintf(stderr,"sorting...");
298     node_list = sort_nodelist(node_list);
299     fprintf(stderr,"Done.\n");
300
301     i=0;
302     cntr=0;
303     lpntr=node_list;
304     while( lpntr != NULL) {
305       /* fprintf(stderr,"adding %d points to bucket %d\n",
306           lpntr->leafnode->ndata,i); */
307       cntr += lpntr->leafnode->ndata;
308       lpntr->leafnode->bucket=i;
309       if( cntr >= BUCKET_SIZE ) {
310         cntr = 0;
311         i++;
312       }
313       lpntr = lpntr->nextleaf;
314     }
315     if( cntr == 0) i--; /* throw out last bucket if nothing got put into it */
316     if( (i+1) > num_buckets)
317       fprintf(stderr,"ERROR: made %d buckets, created %d.\n",(i+1),num_buckets);
318     num_buckets = i+1;
319     fprintf(stderr,"Initialized %d buckets.\n",num_buckets);
320   }
321
322   struct leafpntr *make_nodelist(nodep,num_pnts)
323         struct node *nodep;
324         int *num_pnts;
325   {
326     struct leafpntr *lpntr, *endp;
327
328     lpntr=newleafpntr();
329     lpntr->leafnode = nodep;
330     lpntr->nextleaf=NULL;
331     *num_pnts += nodep->ndata;
332
333     if( nodep->leaf != 1) {
334       lpntr->nextleaf = make_nodelist(nodep->tchild,num_pnts);
335       endp=lpntr;
336       while( endp->nextleaf != NULL) endp = endp->nextleaf;
337       endp->nextleaf = make_nodelist(nodep->fchild,num_pnts);
338     }
339
340     return(lpntr);
341   }
342
343   struct leafpntr *sort_nodelist(nlist)
344         struct leafpntr *nlist;
345   {
346     struct leafpntr *base, *temp, *best;
347     int swapped,max, count;
348
349     base=NULL;
350     while( nlist != NULL) {
351       max = 0;
352       count = 0;
353       temp=nlist;
354       best=NULL;
355       while(temp != NULL) {
356         if( temp->leafnode->ndata > max) {
357           best=temp;
358           max = temp->leafnode->ndata;
359         }
360         count++;
```

LINE # SOURCE TEXT

```
361       temp=temp->nextleaf;
362     }
363     if( best != NULL) {
364       /* fprintf(stderr,"found max=%d in list of %d nodes.\n",max,count); */
365       temp=nlist;
366       if( nlist == best) nlist = best->nextleaf;
367       else {
368         while(temp->nextleaf != best) temp = temp->nextleaf;
369         temp->nextleaf = best->nextleaf;
370       }
371       best->nextleaf = base;
372       base = best;
373     }
374   }
375
376   return(base);
377 }
378
379
380 /*************************************************************************/
381 /* append(data1,data2) appends two lists of data points and returns */
382 /*                      a pointer to the new list */
383
384 struct datum *append(datap1,datap2)
385       struct datum *datap1, *datap2;
386 {
387   struct datum *end;
388
389   /* We leave data2 alone. Go through data1 until you get to the last */
390   /* element (it has a NULL link). Make that element oint to the first */
391   /* element in data2. Data1 now points to the combined list. */
392
393   end=datap1;
394   while (end->link!=NULL) end=end->link;
395   end->link=datap2;
396   return(datap1);
397 }
398
399 /*************************************************************************/
400   /* find_cut(nodep,delta,lentrpy,size,npnts) */
401   /* */
402   /*      This function recursively descends the (sub)tree with base */
403   /*      at nodep and returns a pointer to the node within that */
404   /*      subtree which could be made a leaf (have its children cut) */
405   /*      with the smallest increase in impurity. The amount of that */
406   /*      increase is returned in delta, and the size and total entropy of */
407   /*      this (sub)tree's leaves (before the cut) are returned in */
408   /*      size and lentrpy, respectively. The total */
409   /*      number of data points mapped onto the tree (npnts) are */
410   /*      required to compute the necessary impurities. */
411
412 struct node *find_cut(nodep,delta,lentrpy,size,npnts)
413       struct node *nodep;
414       float *delta, *lentrpy;
415       int *size,npnts;
416 {
417   struct node *bestnode, *tbest, *fbest;
418   float tdelta,fdelta,heredelta,tentrpy,fentrpy;
419   int tsize, fsize;
420
421   if (nodep->leaf==1) {
422
423     /* LEAF NODE: if this is a leaf, life is simple: do not make a cut */
424     /* here! To prevent this from happening, return a huge change in */
425     /* entropy. Clearly the size of a leaf is 1. */
426
427     bestnode=nodep;
428     *lentrpy = entropy(nodep) * ((nodep->ndata)/(float)npnts);
429     *delta = 1.0e10;
430     *size=1;
431
432   }else{
433
434     /* INTERNAL NODE: first, find the best cut in each of the children */
435     /* nodes. The total leaf entopy below this node is just the sum of */
436     /* the leaf entropies of the children, and the leaf count of this */
437     /* subtree is tho leaf count of the children subtrees */
438
439     tbest=find_cut(nodep->tchild,&tdelta,&tentrpy,&tsize,npnts);
440     fbest=find_cut(nodep->fchild,&fdelta,&fentrpy,&fsize,npnts);
441     *lentrpy=tentrpy+fentrpy;
442     *size=tsize+fsize;
443
444     /* The change in impurity that will result from making cutting off */
445     /* the children subtrees is the change in entropy divided by the */
446     /* change in the size of the tree. If the cost of cutting off the */
447     /* children is less than the cost of the cheapest cut within the */
448     /* children, return a pointer to this node and report this cost. */
449     /* Otherwise, return a pointer to the node with the cheapest cut */
450     /* amoungst the children and that cost. */
451
452     heredelta=(nodep->entropy-tentrpy-fentrpy)/(tsize+fsize);
453     if ( (tdelta>heredelta) && (fdelta>heredelta) ) {
454       *delta = heredelta;
455       bestnode = nodep;
456     }else{
457       if (tdelta>fdelta) {
458         *delta = fdelta;
459         bestnode = fbest;
460       }else{
461         *delta = tdelta;
462         bestnode = tbest;
463       }
464     }
465   }
466   return(bestnode);
467 }
468
469 /*************************************************************************/
470 /* cut_tree(nodep)          */
471   /*       This function cuts off the children of the node pointed to */
472   /*       by nodep thus making it a leaf node. Nothing is returned. */
473   /*       */
474
475 void cut_tree(nodep)
476       struct node *nodep;
477 {
478
479   nodep->leaf=1;                          /* make it a leaf */
480   freetree(nodep->tchild);                /* release the two children */
```

| LINE # | SOURCE TEXT |
|---|---|

```
481    freetree(nodep->fchild);          /* subtrees, */
482    nodep->tchild=NULL;               /* and cancel the pointers */
483    nodep->fchild=NULL;
484  }
485
486  /**************************************************************************/
487   /* classify_list(nodep,leaflist,datap,npnts)          */
488   /*           This function recursively descends the (sub)tree whose */
489   /*           bass is pointed to by nodep, and maps the data in the list */
490   /*           pointed to by datap to its leaves. It returns a pointer */
491   /*           to a list of these leaves in leaflist. The total number */
492   /*           of data points in the dataset (npnts) is used to */
493   /*           calculate the entropy at each node. */
494
495  float classify_list(nodep,leaflist,datap,npnts)
496       struct node *nodep;
497       struct leafpntr **leaflist;
498       struct datum *datap;
499       int npnts;
500  {
501    struct datum *Tlist, *Flist, *nextp;
502    struct leafpntr *newleafp, *tleaves, *fleaves;
503    float entrpy, sub_entrpy;
504    int sizeofsubtree;
505
506    /* Process this node first: Attach the list of data to it, and fill */
507    /* its fields with the appropriate information. */
508
509    nodep->ndata=count(datap);
510    nodep->data=datap;
511    getpclasses(nodep);
512    nodep->entropy=nodep->ndata*entropy(nodep)/npnts;
513
514    if (nodep->leaf==1) {
515
516      /* LEAF NODE: if this is a leaf node, we're almost done. A pointer to*/
517      /* this node IS the list of leaves and its entropy IS the */
518      /* total. */
519
520      /* printf("cnt=%d\n",nodep->ndata); */
521      /* printdata(nodep->data); */
522      sub_entrpy=nodep->entropy;
523      newleafp=newleafpntr();
524      newleafp->leafnode=nodep;
525      newleafp->nextleaf=NULL;
526      *leaflist=newleafp;
527
528    }else{
529
530      /* INTERNAL NODE: if this is an internal node, then the list of */
531      /* data points should be divided between the children and mapped */
532      /* to their respective subtrees. */
533
534      Tlist=NULL;
535      Flist=NULL;
536      tleaves=NULL;
537      fleaves=NULL;
538      nodep->data=NULL;
539
540      /* Go through all the data points testing which child they belong */
541      /* to, and moving each point onto the appropriate list. */
542
543      while(datap!=NULL) {
544        nextp=datap->link;
545        if( TEST(datap->obsrv,nodep->qid,nodep->thresh)) {
546          datap->link=Tlist;
547          Tlist=datap;
548        }else{
549          datap->link=Flist;
550          Flist=datap;
551        }
552        datap=nextp;
553      }
554
555      /* If both children have non-empty lists, we just call */
556      /* classify_list again to map them onto their respective subtrees. */
557      /* When they return their leaf lists, we simply append them to */
558      /* form the leaflist for this node. */
559
560      if ( (Tlist!=NULL) && (Flist!=NULL) ) {
561
562        sub_entrpy = classify_list(nodep->fchild,&fleaves,Flist,npnts);
563        sub_entrpy += classify_list(nodep->tchild,&tleaves,Tlist,npnts);
564        *leaflist = addleaves(tleaves,fleaves);
565
566      }else{
567
568        /* This is a special case: it occurs when no pruning data got */
569        /* mapped to a node. When this occurs, we excise the node, and */
570        /* replace its parent node with its sibling */
571
572        if (Tlist==NULL) {
573          sub_entrpy =classify_list(nodep->fchild,&fleaves,Flist,npnts);
574          if(nodep->parent->tchild==nodep) (nodep->parent)->tchild=nodep->fchild;
575          if(nodep->parent->fchild==nodep) (nodep->parent)->fchild=nodep->fchild;
576          *leaflist=fleaves;
577          freetree(nodep->tchild);
578          free(nodep);
579        }else{
580          sub_entrpy =classify_list(nodep->tchild,&tleaves,Tlist,npnts);
581          if(nodep->parent->tchild==nodep) (nodep->parent)->tchild=nodep->tchild;
582          if(nodep->parent->fchild==nodep) (nodep->parent)->fchild=nodep->tchild;
583          *leaflist=tleaves;
584          freetree(nodep->fchild);
585          free(nodep);
586        }
587      }
588    }
589    return(sub_entrpy);
590  }
591
592  /**************************************************************************/
593   /*  addleaves(list1,list2)                          */
594   /*           Simply returns a pointer to a list formed by appending */
595   /*           list2 to list1.    */
596
597  struct leafpntr *addleaves(list1,list2)
598       struct leafpntr *list1, *list2;
599  {
600    struct leafpntr *end;
```

LINE # SOURCE TEXT

```
601
602   end=list1;
603   while(end->nextleaf!=NULL) end=end->nextleaf; /*get to last point in list1,*/
604   end->nextleaf=list2;                          /* make it point to list2.*/
605   return(list1);
606 }
607
608 /*************************************************************************/
609  /*  freeleaves(leaflist)    -- frees all the elements of a list of */
610  /*                        ponters to leaves. */
611
612 void freeleaves(leaflist)
613      struct leafpntr *leaflist;
614 {
615   struct leafpntr *nextp, *thisleaf;
616
617   thisleaf=leaflist;
618   while (thisleaf!=NULL) {
619     nextp=thisleaf->nextleaf;
620     free(thisleaf);
621     thisleaf=nextp;
622   }
623 }
624
625 /*************************************************************************/
626  /* strip_leaves(leaflist)      */
627  /*               This function goes through each leaf node pointed to */
628  /*               by leaflist and removes the data points on those */
629  /*               nodes. The data points are linked together into a */
630  /*               single list, a pointer to which is returned. */
631
632 struct datum *strip_leaves(leaflist)
633      struct leafpntr *leaflist;
634 {
635   struct leafpntr *leafp;
636   struct datum *start, *newstart, *end;
637
638   leafp=leaflist;
639   start=NULL;
640   end=NULL;
641   while (leafp!=NULL) {
642     /* if(leafp->leafnode->entropy>0.0) printdata(leafp->leafnode->data); */
643     if(leafp->leafnode->ndata==0) fprintf(stderr,"Empty leafnode found!!\n");
644     else {
645       newstart=leafp->leafnode->data;  /*get pntr to data for this node */
646       end=newstart;
647       while (end->link!=NULL) end=end->link;  /* go thru list and find end */
648       end->link=start;           /* make last point to list already stripped */
649       start=newstart;            /* make list stipped include this nodes list.*/
650     }
651     leafp->leafnode->data=NULL;  /* cancel this nodes data pntr */
652     leafp=leafp->nextleaf;       /* get the next leaf node */
653   }
654   return(start);
655 }
656
657 /*************************************************************************/
658  /*  count(datap) -- returns a count of the number of data points in a list.*/
659
660 int count(datap)
661      struct datum *datap;
662 {
663   int cntr=0;
664
665   while (datap!=NULL) {       /*    while there's still data in the list, */
666     cntr++;                   /* increment the counter,                  */
667     datap=datap->link;        /* and get the next point.                 */
668   }
669   return(cntr);
670 }
671
672 /*************************************************************************/
673  /*  divide(leaflist,npnts)        */
674  /*            This function will implements the main loop of the greedy */
675  /*            growing algorithm. It searches throught all the leafnodes */
676  /*            for the best leaf to split and the best way to split it. */
677  /*            When it finds the best split of the best leaf, it splits */
678  /*            that node, and updates the list of leaves. Npnts is used */
679  /*            to calculate the various entropies, and the function */
680  /*            returns the change in impurity that results. */
681
682 float divide(leaflist,npnts)
683      struct leafpntr *leaflist;
684      int npnts;
685 {
686   struct leafpntr *leafp;
687   struct node *nodep,*best_node;
688   int best_qid, try_qid, i;
689   float best_thresh, try_thresh, best_change, try_change;
690
691   /* First, we need to identify the best node to split and how to */
692   /* split it. We do that by going through the list of leaf nodes one */
693   /* at a time. If the leaf node has more than the minimum number of */
694   /* data points assigned to it, we'll see how much of a change in */
695   /* impurity we'd get if did the best job of dividing that node. If */
696   /* this expected change is the biggest we've seen so far, we'll save */
697   /* the node, question, and parameters for possible future use. When */
698   /* the loop exits, we will have examined all the leaves and the */
699   /* saved information will identify the best node to split and how to */
700   /* split it. */
701
702   best_change=0.0;
703   leafp=leaflist;
704   while (leafp!=NULL) {
705     if( (leafp->leafnode->pure==0) && (leafp->leafnode->ndata>MINPTS)) {
706       try_change=identify_question(leafp->leafnode,&try_qid,&try_thresh)/npnts;
707       if( try_change < PURE) leafp->leafnode->pure=1;
708       else if (try_change>best_change) {
709         best_change = try_change;
710         best_node = leafp->leafnode;
711         best_qid = try_qid;
712         best_thresh = try_thresh;
713       }
714     }
715     leafp = leafp->nextleaf;
716   }
717
718   /* If the best change in impurity (largest) is greater than zero, */
719   /* then there is a node to be split. Split the node using the */
720   /* question and parameters saved during the search above to do so. */
```

LINE # SOURCE TEXT

```
721   /* Finally, as a check, make sure that the change in entropy */
722   /* predicted during the search is actually what we got when the node */
723   /* was divided, and warn the user if it wasn't. */
724
725   try_change = 0.0;
726   if(best_change>=PURE){
727     fprintf(stderr,"(best split: question %d using %f) ",
728             best_qid,best_thresh);
729     reductions[best_qid] += best_change;
730     try_change=split(best_node,best_qid,best_thresh,leaflist,npnts);
731   } else best_change = 0.0;
732   if (try_change!=best_change) fprintf(stderr,"predicted and actual differ\n");
733   return(best_change);
734 }
735
736 /*********************************************************************/
737  /*  split(nodep,qid,thresh,leaflist,npnts)                          */
738  /*         The leaf node pointed to by nodep is split using question */
739  /*         qid with the threshold in thresh. The list of leaf nodes */
740  /*         (leaflist) is updated, and the change in impurity is */
741  /*         returned. The total number of data points mapped onto the */
742  /*         tree (npnts) is used to compute entropies.   */
743
744 float split(nodep,qid,thresh,leaflist,npnts)
745      struct node *nodep;
746      int qid, npnts;
747      float thresh;
748      struct leafpntr *leaflist;
749 {
750   struct leafpntr *leafp,*nextleaf,*tleaf,*fleaf;
751   struct datum *datap,*nextp;
752   struct node *Tnodep, *Fnodep;
753   int i,tcnt=0,fcnt=0;
754   float delta;
755
756   /* First, we allocate two new nodes which become the children of the */
757   /* node being split. We also identify the node being split as their */
758   /* parent node. */
759
760   Tnodep=newnode();
761   Tnodep->parent=nodep;
762   Fnodep=newnode();
763   Fnodep->parent=nodep;
764
765   /* Now we go through the list of data mapped onto this node and */
766   /* break it into two seperate lists: one for data points which */
767   /* satisfy the question (attached to the true-child node), and */
768   /* one for data points that do not (attached to the false-child node. */
769
770   datap=nodep->data;
771   while(datap!=NULL) {
772     nextp=datap->link;
773     if (TEST(datap->obsrv,qid,thresh)) { /* test is True*/
774       tcnt++;
775       datap->link=Tnodep->data;  /* make point the new head of the */
776       Tnodep->data=datap;        /* list of true-child points.    */
777     }else{
778       fcnt++;
779       datap->link=Fnodep->data;  /* make point the new head of the */
780       Fnodep->data=datap;        /* list of false-child points.   */
781     }
782     datap=nextp;
783   }
784   Tnodep->ndata=tcnt;   /* set true-child data count. */
785   Fnodep->ndata=fcnt;   /* set false-child data count. */
786
787   /* Now calculate the entropies of the children nodes, and convert */
788   /* this node to an internal node with the two new nodes as its */
789   /* children. Notice that we copy all the information about how we */
790   /* split this node into the structure associated with it. */
791
792   getpclasses(Tnodep);
793   Tnodep->entropy=Tnodep->ndata*entropy(Tnodep)/npnts;
794   getpclasses(Fnodep);
795   Fnodep->entropy=Fnodep->ndata*entropy(Fnodep)/npnts;
796   nodep->leaf=0;               /* make this an internal node */
797   nodep->tchild=Tnodep;        /* connect the true-child node */
798   nodep->fchild=Fnodep;        /* connect the false-child node */
799   nodep->data=NULL;            /* delete data pntr (data is now in children).*/
800   nodep->qid=qid;              /* save question used for making split */
801   nodep->thresh=thresh;        /* save params used.*/
802
803   /* Now we need to update the list of leaf nodes. This node needs to */
804   /* be removed from the list since it is no longer a leaf, and the */
805   /* two children nodes need to be added. */
806
807   leafp=leaflist;
808   while (leafp->leafnode!=nodep) leafp=leafp->nextleaf; /* find pntr */
809                                                          /* to this */
810                                                          /* node in the */
811                                                          /* leaflist. */
812   leafp->leafnode=Tnodep;           /* overwrite with pntr to true-child.*/
813   fleaf=newleafpntr();              /* Allocate another leaf pntr, */
814   fleaf->leafnode=Fnodep;           /* make it point to false-child, */
815   fleaf->nextleaf=leafp->nextleaf; /* make it link to rest of list  */
816   leafp->nextleaf=fleaf;            /* insert new leaf pntr in list  */
817
818   /* Finally, the change in impurity due to this split is computed as */
819   /* the difference in entropy between this node and its children. */
820
821   delta = (tcnt+fcnt)*entropy(nodep)-tcnt*entropy(Tnodep)-fcnt*entropy(Fnodep);
822   return(delta/npnts);
823 }
824
825 /*********************************************************************/
826 /*   identify_question(nodep,best_qid,best_thresh)                   */
827  /*              Very simply, this function determines the best way to */
828  /*              split the node pointed to by nodep. It does this by */
829  /*              sequentially trying each of the possible questions and, */
830  /*              for each one, idenifying the best set of parameters. The */
831  /*              best (largest) decrease in impurity identifies the best */
832  /*              question and parameters which are returned in best_qid */
833  /*              and best_param, respectively. The function returns the */
834  /*              change in impurity associated with making splitting this */
835  /*              node in the best way. */
836
837 float identify_question(nodep,best_qid,best_thresh)
838      struct node *nodep;
839      int *best_qid;
840      float *best_thresh;
```

```
841  {
842    int i, try_qid;
843    float best_change, try_change, try_thresh;
844
845    best_change=0.0;
846    *best_qid=0;
847
848    for (try_qid=0;try_qid<NQUESTIONS;try_qid++){
849      try_change=identify_split(nodep,try_qid,&try_thresh);
850      if (try_change>best_change) {
851        *best_qid=try_qid;
852        best_change=try_change;
853        *best_thresh=try_thresh;
854      }
855    }
856    return(best_change);
857  }
858
859  /**************************************************************************/
860   /*  identify_split(nodep,qid,best_thresh)                    */
861   /*             This function finds the best parameterization of */
862   /*             question number qid to split the data assigned to */
863   /*             nodep. This split maximizes the change in entropy */
864   /*             between this node and its prospective children. The */
865   /*             values of the parameterization found are passed back in */
866   /*             the array best_param, and the function returns the */
867   /*             change in entropy expected if the split is made. */
868
869  float identify_split(nodep,qid,best_thresh)
870       struct node *nodep;
871       int qid;
872       float *best_thresh;
873  {
874    struct datum *datap;
875    int i,cntr;
876    float tempval, best_change, try_change, try_thresh;
877
878    if( qid < NFLAGQ){
879      *best_thresh = 0.5;
880      best_change = slow_try_split(nodep,qid,*best_thresh);
881    }else{
882      cntr=0;
883      datap=nodep->data;
884      while (datap!=NULL) {
885        class_buf[cntr] = datap->class;
886        float_buf[cntr++] = datap->obsrv[qid];
887        datap = datap->link;
888      }
889      if( cntr != nodep->ndata)
890        fprintf(stderr,"Wrong number of observations found on a node!\n");
891      else{
892        best_change=0.0;
893        for(i=0;i<cntr;i++) {
894          try_thresh = float_buf[i];
895          try_change=try_split(nodep,qid,try_thresh);
896          if( try_change>best_change){
897           best_change=try_change;
898           *best_thresh=try_thresh;
899          }
900        }
901        if (find_neighbor(cntr,*best_thresh,qid,&tempval))
902          *best_thresh = (*best_thresh + tempval)/2.0;
903      }
904    }
905
906  /*    best_change=0.0;
907     for( try_thresh = thresh_start[qid];
908         try_thresh <= thresh_stop[qid];
909         try_thresh += thresh_incr[qid]){
910       try_change=try_split(nodep,qid,try_thresh);
911       if (try_change>best_change) {
912         best_change=try_change;
913         *best_thresh=try_thresh;
914       } */
915
916    return(best_change);
917  }
918
919  /**************************************************************************/
920  /* find_neighbor(count,val,index,neighbor)     */
921  /*                This function finds the nearest neighbor greater */
922  /*                of observ[index] greater than val and returns its */
923  /*                value in neighbor. The function returns a 1 if a */
924  /*                neighbor is found or a 0 if not. */
925
926  int find_neighbor(count,val,index,neighbor)
927       int count;
928       float val, *neighbor;
929       int index;
930  {
931    int i,result;
932    struct datum *nextp;
933    float temp;
934
935    result=0;              /* start by saying we didn't find neighbor */
936    *neighbor=1.0e200;     /* big, anything real should be smaller */
937
938    for(i=0;i<count;i++) {
939      temp = float_buf[i];
940      if ( (temp>val) && (temp<*neighbor) ) {
941        result=1;
942        *neighbor=temp;
943      }
944    }
945    return(result);
946  }
947
948
949  /**************************************************************************/
950  /**************************************************************************/
951  /* the following functions do not include any question-specific code. */
952   /* They are included in this file because the question-specific */
953   /* functions use them. You shouldn't need to edit anything below */
954   /* here. */
955
956
957  /**************************************************************************/
958   /* try_split(nodep,qid,thresh)          */
959   /*             Returns the change in entropy which would occur if the */
960   /*             data assigned to node nodep were split using question */
```

LINE # SOURCE TEXT

```
961  /*          gid with parameterization params. */
962
963  float try_split(nodep,qid,thresh)
964       struct node *nodep;
965       int qid;
966       float thresh;
967  {
968    struct datum *datap;
969    int i, N, Tcnt=0, Fcnt=0, cntrs[2][NCLASS];
970    float Tent=0.0, Fent=0.0, prob;
971
972    /* Basically, we're going to count how many data points of each */
973    /* class would get assigned to each child is this split were done. */
974    /* So start by setting the counters to zero. */
975
976    for (i=0;i<NCLASS;i++) {
977      cntrs[0][i]=0.0;
978      cntrs[1][i]=0.0;
979    }
980
981    /* Now go through all the data points in the list assigned to this */
982    /* node and test whether the question is true or false for each one */
983    /* Increment the appropriate counters. */
984
985    for(i=0;i<(nodep->ndata);i++) {
986      if( float_buf[i] > thresh) {
987        Tcnt++;
988        cntrs[1][class_buf[i]]++;
989      }else{
990        Fcnt++;
991        cntrs[0][class_buf[i]]++;
992      }
993    }
994
995    /* Now that the countings done, we can comput the entropy at for */
996    /* each child by finding the class probabilities. */
997
998    N=Tcnt+Fcnt;
999    for(i=0;i<NCLASS;i++) {
1000     prob=(float) cntrs [1][i]/(float)Tcnt;  /* P[class i in tchild] */
1001     if(prob>0.0) Tent -= prob*log(prob);
1002     prob=(float) cntrs [0][i]/(float)Fcnt;  /* P[class i in fchild] */
1003     if(prob>0.0) Fent -= prob*log(prob);
1004   }
1005
1006   /* Finally, we can compute the difference in entropy between this */
1007   /* node and its potential children. */
1008
1009   return(N*entropy(nodep)-Tcnt*Tent-Fcnt*Fent);
1010 }
1011
1012
1013 float slow_try_split(nodep,qid,thresh)
1014      struct node *nodep;
1015      int qid;
1016      float thresh;
1017 {
1018   struct datum *datap;
1019   int i, N, Tcnt=0, Fcnt=0, cntrs[2][NCLASS];
1020   float Tent=0.0, Fent=0.0, prob;
1021
1022   /* Basically, we're going to count how many data points of each */
1023   /* class would get assigned to each child is this split were done. */
1024   /* So start by setting the counters to zero. */
1025
1026   for (i=0;i<NCLASS;i++) {
1027     cntrs[0][i]=0.0;
1028     cntrs[1][i]=0.0;
1029   }
1030
1031   /* Now go through all the data points in the list assigned to this */
1032   /* node and test whether the question is true or false for each one. */
1033   /* Increment the appropriate counters. */
1034
1035   datap=nodep->data;
1036   while( datap != NULL) {
1037     if (TEST(datap->obsrv,qid,thresh)) {
1038       Tcnt++;
1039       cntrs[1][datap->class]++;
1040     }else{
1041       Fcnt++;
1042       cntrs[0][datap->class]++;
1043     }
1044     datap = datap->link;
1045   }
1046
1047   /* Now that the countings done, we can comput the entropy at for */
1048   /* each child by finding the class probabilities. */
1049
1050   N=Tcnt+Fcnt;
1051   for(i=0;i<NCLASS;i++) {
1052     prob=(float) cntrs [1][i]/(float)Tcnt;  /* P[class i in tchild] */
1053     if(prob>0.0) Tent -= prob*log(prob);
1054     prob=(float) cntrs [0][i]/(float)Fcnt;  /* P[class i in fchild] */
1055     if(prob>0.0) Fent -= prob*log(prob);
1056   }
1057
1058   /* Finally, we can compute the difference in entropy between this */
1059   /* node and its potential children. */
1060
1061   return(N*entropy(nodep)-Tcnt*Tent-Fcnt*Fent);
1062 }
1063
1064
1065 /*************************************************************************/
1066  /* entropy(nodep) -- returns the entropy associated with node nodep. */
1067  /* */
1068
1069 float entropy(nodep)
1070      struct node *nodep;
1071 {
1072   float entrpy;
1073   int i;
1074
1075   entrpy=0.0;
1076   for(i=0;i<NCLASS;i++)
1077     if (nodep->pclass[i]!=0.0)
1078       entrpy += nodep->pclass[i]*log(nodep->pclass[i]);
1079   return(-entrpy);
1080 }
```

LINE # SOURCE TEXT

```
1081
1082 /**************************************************************************/
1083 /**************************************************************************/
1084 /* The following functions provide utility services for initializing and
1085 manipulating the data structures */
1086
1087
1088 /**************************************************************************/
1089  /* newnode() -- allocates and initializes a new node. returns a */
1090  /*             pointer to it. */
1091
1092 struct node *newnode()
1093 {
1094   struct node *newp;
1095   int i;
1096
1097   newp = (struct node *) malloc(sizeof(struct node)); /* get the memory */
1098   newp->pure=0;
1099   newp->parent = NULL;
1100   newp->leaf   = 1;     /* default is to make this a leaf node */
1101   newp->tchild = NULL;
1102   newp->fchild = NULL;
1103   newp->qid    = -1;    /* this is an illegal question code  */
1104   newp->ndata = 0;      /* no data is assigned to this node. */
1105   newp->data   = NULL;
1106   for (i=0;i<NCLASS;i++) newp->  pclass[i] = 0.0; /* clear the p[class]*/
1107   newp->thresh = 0;
1108   return(newp);
1109 }
1110
1111 /**************************************************************************/
1112  /* freetree(nodep) -- Recursively descends the (sub)tree with base */
1113  /*                    at nodep, and releases all the nodes. */
1114  /*                    */
1115
1116 void freetree(nodep)
1117      struct node *nodep;
1118 {
1119   if (nodep->leaf!=1) {          /* if this is an internal node, release */
1120     freetree(nodep->tchild);     /* the children subtrees first. */
1121     freetree(nodep->fchild);
1122   }
1123   free(nodep);                   /* Then release this node. */
1124 }
1125
1126 /**************************************************************************/
1127  /* getpclasses(nodep) -- Computes the conditional probability of each */
1128  /*                       class, given this node, based on the data */
1129  /*                       assigned to this node. */
1130
1131 void getpclasses(nodep)
1132      struct node *nodep;
1133 {
1134   int cntrs[NCLASS],N,i,maxc,num_zeros;
1135   float maxp, pmin;
1136   struct datum *datump;
1137
1138   /* To get started, zero out all the counters and initialize a */
1139   /* pointer to the first data point on the list assigned to this */
1140   /* node. */
1141
1142   N=0;
1143   for (i=0;i<NCLASS;i++) cntrs[i]=0;
1144   datump=nodep->data;
1145
1146   /* Go through all the data points and increment the total counter */
1147   /* and the counter for the corresponding class. */
1148
1149   while (datump != NULL) {
1150     cntrs[datump->class]++;
1151     N++;
1152     datump=datump->link;
1153   }
1154
1155   /* Now compute the conditional probabilities by dividing the number */
1156   /* of observations in each class by the total number of */
1157   /* observations. Also, keep track of the largest probability; the */
1158   /* corresponding class is the one to be assigned to this node. */
1159   maxp=0.0;
1160   for (i=0;i<NCLASS;i++) {
1161     nodep->pclass[i]=(float) cntrs[i]/ (float)N;
1162     if (nodep->pclass[i]>maxp) {
1163       maxp=nodep->pclass[i];
1164       maxc=i;
1165     }
1166   }
1167   nodep->class=maxc; /* save the most likely class. */
1168 }
1169
1170 /**************************************************************************/
1171  /* getdata(filename1,headp,npnts,filename2,headp2,npnts2)       */
1172  /*           This function reads the training data. It is expected that */
1173  /*           the training data is in two files (filename1 and */
1174  /*           filename2) which contain the observation vectors and the */
1175  /*           corresponding class labels. As the training data is read */
1176  /*           in, it is divided into two subsets; one for growing the */
1177  /*           tree and one for pruning it. Pointers to these subsets are */
1178  /*           returned in headp and headp2, and counts of the number of */
1179  /*           points in each subset are returned in npnts and npnts2. */
1180  /*           It also generates the maximum-likelihood estimates of the */
1181  /*           initial and transition probabilities while reading the data. */
1182
1183 int getdata(filename1,headp,npnts,filename2,headp2,npnts2)
1184      char filename1[],filename2[];
1185      struct datum **headp,**headp2;
1186      int *npnts,*npnts2;
1187 {
1188   struct datum *lastd, *lastd2, *data, *data2, *tempd;
1189   int cntr1,cntr2,tclass,i,out,toggle,item;
1190   float tobs[DOBS];
1191   char dat1[32], dat2[32];
1192   float anorm[NCLASS],pnorm,e;
1193   int j,laststate,firstobs,numstarts,nvisits[NCLASS];
1194   FILE *fp1,*fp2;
1195
1196   /* INITIALIZATION: start by trying to open the two data files and */
1197   /* clearing the cntrs. Also set the two list pointers to NULL, which */
1198   /* indicates that they do not point to anything. */
1199
1200   out=1;
```

LINE # SOURCE TEXT

```
1201  if ( (fp1=fopen(filename1,"r"))==NULL) {
1202    fprintf(stderr,"Can't open file %s.\n",filename1);
1203    out=NULL;
1204  }
1205  if ( (fp2=fopen(filename2,"r"))==NULL) {
1206    fprintf(stderr,"Can't open file %s.\n",filename2);
1207    out=NULL;
1208  }
1209  cntr1=0;
1210  cntr2=0;
1211  data=NULL;
1212  data2=NULL;
1213
1214  /* initialize the finger-counting things for estmating the HMM params. */
1215  numstarts=0;
1216  firstobs=1;
1217  for(i=0;i<NCLASS;i++){
1218    initials[i]=0.0;
1219    nvisits[i]=0.0;
1220    anorm[i]=0.0;
1221    for(j=0;j<NCLASS;j++) transitions[i][j]=0.0;
1222  }
1223  laststate=LASTSTATE;
1224  nvisits[laststate]=1;
1225
1226  if (out=1) {
1227
1228    /* Everything went okay in initialization if we're here. Now we'll */
1229    /* read in the data. There are three complications. (1) We want to */
1230    /* split the data into two subsets. This is done via "toggle" */
1231    /* which is incremented modulo-two to select which subset to */
1232    /* assign each data point to. In this way, points are assigned */
1233    /* alternating between the two subsets so that they will be */
1234    /* roughly equal in size. The second complication (2) is that the */
1235    /* data files are expected to contain "***" to seperate tokens in */
1236    /* the input, and we want to ignore these. To do this, each file */
1237    /* is read into a string which is then checked for the presence of */
1238    /* "***". If this is not found, the string is converted to a */
1239    /* number and treated as data. Finally, (3) we need to keep track of how
1240       many times an utterence started in each state, and many times we saw
1241       each transition. */
1242
1243    srand(1234);
1244    toggle=0; /* old name...it just counts the number of pts read */
1245    item=1;
1246    while ((fscanf(fp1,"%s",dat1)!=EOF)&&(fscanf(fp2,"%s",dat2)!=EOF)) {
1247      if ( strcmp(dat1,"***") && strcmp(dat2,"***") ) {
1248
1249        /* Neither string contains "***", so convert them to data, */
1250        /* read in the rest of the observation vector, */
1251        /* and make sure that the class is legal. If not, we can't */
1252        /* really recover so give-up and the user to fix it. */
1253
1254        sscanf(dat1,"%f",&tobs[0]);
1255        sscanf(dat2,"%d",&tclass);
1256        for (i=1;i<DOBS;i++) fscanf(fp1,"%f",&tobs[i]);
1257        if ((tclass<0)||(tclass>=NCLASS)) {
1258          fprintf(stderr,"Illegal class (%d) in data item %d, exiting\n",
1259                  tclass,toggle+1);
1260          exit (-1);
1261        }
1262 /*if( (tobs[0]==0) && (tclass%7 != 0) )
1263   fprintf(stderr,"no EOW flag and break not 0!\n");
1264  if( (tobs[0]==1) && (tclass%7 == 0) )
1265   fprintf(stderr,"EOW flag and break is 0!\n"); */
1266
1267        /* The class is legal, so allocate a new data pointer, copy */
1268        /* the observation vector into the data point, and set the */
1269        /* class. Finally, assign the data point to whichever subset */
1270        /* is randomly selected, and update the appropriate counter. */
1271
1272        tempd=newdatum();
1273        tempd->link=NULL;
1274        for (i=0;i<DOBS;i++) tempd->obsrv[i]=tobs[i];
1275        tempd->class=tclass;
1276        if ( (((float) rand())/2147483648.0) <= TRAIN_FRAC) {
1277          if (data==NULL) {     /* if this is the first data point in */
1278            data=tempd;         /* this subset, just make it the list, */
1279            lastd=tempd;        /* and the cntr is 1 */
1280            cntr1=1;
1281          } else {
1282            lastd->link=tempd; /* If this isn't the first point in the list,*/
1283            lastd=tempd;        /* just add it to the end of the list which*/
1284            cntr1++;            /* is pointed to by lastd. */
1285          }
1286        } else {
1287          if (data2==NULL) {    /* this is a duplicate of the code for the */
1288            data2=tempd;        /* case where the data point is to be put */
1289            lastd2=tempd;       /* in the pruning data. */
1290            cntr2=1;
1291          } else {
1292            lastd2->link=tempd;
1293            lastd2=tempd;
1294            cntr2++;
1295          }
1296        }
1297        toggle++;
1298        /* Now update our HMM counters: if we just started a new utterence,
1299         then update the initial probability counter. Otherwise, update
1300         the transition count given the last state we saw and this one. */
1301        if( firstobs == 1) {
1302          initials[tclass]++;
1303          numstarts++;
1304          firstobs=0;
1305        }else{
1306          transitions[laststate][tclass]++;
1307          laststate = tclass;
1308          nvisits[tclass]++;
1309        }
1310
1311      } else {
1312
1313        /* one or both of the files produced a "***". If both did, */
1314        /* then we've reached the end of a token and everything is */
1315        /* fine. If only one produced the marker though, then the */
1316        /* files are misaligned and there's nothing we can do except */
1317        /* tell the hapless user and quit. */
1318
1319        if (strcmp(dat1,dat2)) {
1320          fprintf(stderr,
```

LINE # SOURCE TEXT

```
1321                  "Training observations and labels are misaligned in item %d !\n",item);
1322           exit(-1);
1323         }
1324         firstobs=1; /* starting a new utterence. */
1325         item++;
1326       }
1327     }
1328     *npnts=cntr1;      /* number of growing data points */
1329     *headp=data;       /* pointer to growing data list */
1330     *npnts2=cntr2;     /* number of pruning data points */
1331     *headp2=data2;     /* pointer to pruning data list */
1332 
1333     /* Finally, we need to convert our counts into probability estimates for
1334        the HMM. This is done in the usual way: dividing counts by the total.
1335        The odd part is dealing with zero counts...things we never saw. There
1336        are many things we could do which would be elegant, but we choose to
1337        live simply and just give these events a small proability of
1338        1/(100*Total number) and then renormalize everything so the probs all
1339        summ to one in the ways they should. The basic idea is that if I didn't
1340        see an event once in N trials, its probability is probably close
1341        to 1/100N. That is, if I looked at 100 times as much data I'd probably
1342        only see this event once. Crude and suboptimal...but simple. */
1343     pnorm=0.0;
1344     for(i=0;i<NCLASS;i++){
1345       if( initials[i]>0) initials[i] /= (float)numstarts;
1346       else initials[i] = 1.0/(100.0*(float)numstarts);
1347       pnorm += initials[i];
1348       for(j=0;j<NCLASS;j++){
1349         if( nvisits[i] >0){
1350           if( transitions[i][j] > 0) transitions[i][j] /= (float)nvisits[i];
1351           else transitions[i][j] = 1.0/(100.0*(float)nvisits[i]);
1352         }else transitions[i][j] = 1.0/(100.0*(float)nvisits[i]);
1353         anorm[i] += transitions[i][j];
1354       }
1355     }
1356     /* Now we just apply the normalizations */
1357     for(i=0;i<NCLASS;i++) initials[i] /= pnorm;
1358     for(i=0;i<NCLASS;i++)
1359       for(j=0;j<NCLASS;j++) transitions[i][j]/anorm[i];
1360   }
1361   return(out);
1362 }
1363 
1364 /**********************************************************************/
1365   /* newdatum()  -- allocates and initializes a new data point. */
1366   /*                Returns a pointer to the new point. */
1367 
1368 struct datum *newdatum()
1369 {
1370   struct datum *newp;
1371   int i;
1372 
1373   newp= (struct datum *) malloc( sizeof(struct datum) );
1374   newp->link  = NULL;
1375   for(i=0;i<DOBS;i++) newp->obsrv[i] = 0.0;   /* clear out observation vector */
1376   newp->class = -1;                            /* illegal class id */
1377   return(newp);
1378 }
1379 
1380 /**********************************************************************/
1381   /* newleafpntr()  -- allocates and initializes a new leafnode */
1382   /*                   pointer. Returns a pointer. */
1383 
1384 struct leafpntr *newleafpntr()
1385 {
1386   struct leafpntr *newp;
1387 
1388   newp = (struct leafpntr *) malloc( sizeof(struct leafpntr) );
1389   newp->nextleaf=NULL;
1390   newp->leafnode=NULL;
1391   return(newp);
1392 }
1393 
1394 /**********************************************************************/
1395   /* treeprint(nodep)  -- Recursively descends the (sub)tree with base */
1396   /*                      at nodep and prints each node (entropy) in */
1397   /*                      left-recursive order. Also prints the most */
1398   /*                      likely class at each leaf node. */
1399 
1400 void treeprint(nodep)
1401      struct node *nodep;
1402 {
1403   int i;
1404   /* If this is a leaf node, just print out the appropriate */
1405   /* information. Otherwise, print the complete left subtree, then */
1406   /* print the entropy of this internal node, then print the oomplete */
1407   /* right subtree. */
1408 
1409   if (nodep->leaf==1) {
1410     printf("leaf: lambda[%d]=%f,  ",nodep->bucket,lambda[nodep->bucket]);
1411     for(i=0;i<NCLASS;i++) printf("%6.4f  ",nodep->pclass[i]);
1412     printf("\n");
1413   }
1414   else {
1415     treeprint(nodep->tchild);
1416     printf("internal node: lambda[%d]=%f,  ",nodep->bucket,lambda[nodep->bucket]);
1417     for(i=0;i<NCLASS;i++) printf("%6.4f  ",nodep->pclass[i]);
1418     printf("\n");
1419     treeprint(nodep->fchild);
1420   }
1421 }
1422 
1423 /**********************************************************************/
1424   /* dump_model(rootp)                          */
1425   /*          This function is used to dump all the necessary information */
1426   /*          about the completed tree (with root at rootp, containing */
1427   /*          nnodes nodes) to a modelfile called newmodel.hvq. */
1428 
1429 void dump_model(rootp)
1430      struct node *rootp;
1431 {
1432   int i,j, nodenum, tchild, fchild, vqnum, total, num_leaves;
1433   float pclass_code[NCLASS][MAX_LEAVES], pcode[MAX_LEAVES],pstate[NCLASS];
1434   FILE *fp1;
1435 
1436   /* INITIALIZATION: try and open the two output files. Tell the user */
1437   /* what's going on in any case. Once the files are open, zero the */
1438   /* various counters we'll be using to compute probabilities. */
1439 
1440   if ((fp1=fopen("newmodel.hvq","w"))==NULL)
```

LINE # SOURCE TEXT

```
1441      fprintf(stderr,"Cannot open newmodel.hvq to write results\n");
1442    else fprintf(stderr,"Writing model to newmodel.hvq\n");
1443
1444    num_leaves = countleaves(rootp);
1445    for (i=0;i<NCLASS;i++) {
1446      pstate[i]=0.0;
1447      for (j=0;j<num_leaves;j++) pclass_code[i][j]=0.0;
1448    }
1449
1450    /* In order to assign numbers to all the nodes without any */
1451    /* conflicts, we use a single memory location (nodenum) and pass */
1452    /* pointers to it. Each time a node is dumped, it is assigned the */
1453    /* number currently in nodenum and the value of nodenum is */
1454    /* decremented. Since we always process the subtrees first, the */
1455    /* rootnode should always be numbered zero. Notice that if, after */
1456    /* all the growing and pruning, only one node remains, the user has */
1457    /* more problems than the fact that I was too lazy to write the code */
1458    /* for that special case: all they'll get is an error message. */
1459
1460    nodenum=countnodes(rootp)-1;
1461    vqnum=countleaves(rootp)-1;
1462    fprintf(fp1,"%d %d %d\n",nodenum+1,vqnum+1,NCLASS);
1463    if (rootp->leaf==0) {
1464      tchild=dump_subtree(rootp->tchild,&nodenum,&vqnum,fp1,pclass_code,pstate);
1465      fchild=dump_subtree(rootp->fchild,&nodenum,&vqnum,fp1,pclass_code,pstate);
1466    } else {
1467      fprintf(stderr,"ONE NODE TREE! This is bad, bad, bad...\n");
1468      exit(-1);
1469    }
1470
1471    /* Now that we've dumped the subtrees, dump the root node */
1472
1473    fprintf(fp1,"%d %d %d %d %d",nodenum,tchild,fchild,rootp->class,rootp->qid);
1474    fprintf(fp1," %f",rootp->thresh);
1475    fprintf(fp1,"\n\n");
1476
1477    /* Now we can write out the HMM parameters. We start with the initial
1478       probalities followed by the state transition matrix. */
1479    for(i=0;i<NCLASS;i++) fprintf(fp1,"%e ",initials[i]);
1480    fprintf(fp1,"\n\n");
1481    for(i=0;i<NCLASS;i++){
1482      for(j=0;j<NCLASS;j++) fprintf(fp1,"%e ",transitions[i][j]);
1483      fprintf(fp1,"\n");
1484    }
1485    fprintf(fp1,"\n");
1486
1487    /* Now we calculate the state likelihoods and dump them out */
1488    total=0;
1489    for (i=0;i<NCLASS;i++) {
1490      /* fprintf(stderr,"pstate[%d]=%f\n",i,pstate[i]); */
1491      total += pstate[i];
1492    }
1493    for (i=0;i<NCLASS;i++) pstate[i] /= (float)total;
1494
1495    for(i=0;i<num_leaves;i++) {
1496      pcode[i]=0.0;
1497      for (j=0;j<NCLASS;j++) {
1498        pcode[i] += pclass_code[j][i];
1499      }
1500      if (pcode[i]==0) {
1501        fprintf(stderr,
1502                "Zero observations of code %d !!\n THIS SHOULDN'T HAPPEN! (uniform likelihoods assumed)\n",
1503                i);
1504        for (j=0;j<NCLASS;j++) pclass_code[j][i]= 1.0/NCLASS;
1505      } /* else fprintf(stderr,"pcode[%d]=%f\n",i,pcode[i]); */
1506    }
1507
1508    /* Now that we've got the probabilities all computed, lets print */
1509    /* them out, and close the files */
1510
1511    for (i=0;i<num_leaves;i++) {
1512      for (j=0;j<NCLASS;j++)
1513        fprintf(fp1,"%f ",(pclass_code[j][i]/pstate[j]));
1514      fprintf(fp1,"\n");
1515    }
1516    fclose(fp1);
1517  }
1518
1519  /**************************************************************************/
1520  /* dump_subtree(nodep,nnodes,fp,pstate_class,pstate)                    */
1521  /*            Recursively descends subtree with base at nodep and */
1522  /*            dumps all its nodes to the file (previously) opened on */
1523  /*            fp. nnodes should point to a single integer larger than */
1524  /*            the number of nodes in the subtree. Upon return, nnodes */
1525  /*            will have been reduced by the number of subnodes in the */
1526  /*            tree. pstate_class will be increased by the leafnodes */
1527  /*            in the subtree. This function returns the number */
1528  /*            assigned to the subtree base node*/
1529
1530  int dump_subtree(nodep,nnodes,vqnum,fp,pclass_code,pstate)
1531       struct node *nodep;
1532       float pclass_code[NCLASS][MAX_LEAVES],pstate[NCLASS];
1533       int *nnodes, *vqnum;
1534       FILE *fp;
1535  {
1536    int i, nodenum, tchild, fchild, code;
1537
1538    /* Start by initializing nodenum. This location will be used to */
1539    /* coordinate numbering all the nodes in this subtree. */
1540
1541    nodenum = *nnodes;
1542    code = *vqnum;
1543
1544    if (nodep->leaf==1) {
1545
1546      /* If this is a leaf node, we just indicate that there are no */
1547      /* children, and update the array. The array contains a count of */
1548      /* the number of occurences of state i given the class assigned to */
1549      /* this node. We will assign the value in nodenum to this node. */
1550      nodep->class = code;
1551      code = code-1;
1552      tchild = -1;
1553      fchild = -1;
1554      for (i=0;i<NCLASS;i++) {
1555        pclass_code[i][nodep->class] = nodep->pclass[i];
1556        pstate[i] += (nodep->pclass[i]) * (nodep->ndata);
1557      }
1558
1559    }else{
1560
```

LINE # SOURCE TEXT

```
1561     /* If this is an internal node, we dump the children subtrees */
1562     /* first. This gives us the node numbers of the children nodes and */
1563     /* updates nodenum so it conatins the number we'll assign to this */
1564     /* node. */
1565
1566     tchild=dump_subtree(nodep->tchild,&nodenum,&code,fp,pclass_code,pstate);
1567     fchild=dump_subtree(nodep->fchild,&nodenum,&code,fp,pclass_code,pstate);
1568   }
1569
1570   /* At this point, nodenum contains the number we'll assign to this */
1571   /* node. To prevent conflicts, we'll update the source of nodenum to */
1572   /* prevent our number from being used again. */
1573
1574   *nnodes = nodenum-1;
1575   *vqnum = code;
1576
1577   /* Now we can just dump this node to the file. */
1578
1579   fprintf(fp,"%d %d %d %d %d",nodenum,tchild,fchild,nodep->class,nodep->qid);
1580   fprintf(fp," %f",nodep->thresh);
1581   fprintf(fp,"\n");
1582   return(nodenum);
1583 }
1584
1585 /***************************************************************************/
1586  /* countnodes(nodep) -- returns a count of the number of nodes in the */
1587  /*                      (sub)tree with base at nodep. */
1588
1589 int countnodes(nodep)
1590      struct node *nodep;
1591 {
1592   int nnodes;
1593
1594   /* If nodep is a leaf, the count is just 1. Otherwise, the count is */
1595   /* 1 plus the number of nodes in each child tree. */
1596
1597   if (nodep->leaf==1) nnodes=1;
1598   else
1599     nnodes = 1 + countnodes(nodep->tchild) + countnodes(nodep->fchild);
1600
1601   return(nnodes);
1602 }
1603
1604 /***************************************************************************/
1605  /* countleaves(nodep) -- returns a count of the number of leaves in the */
1606  /*                       (sub)tree with base at nodep. */
1607
1608 int countleaves(nodep)
1609      struct node *nodep;
1610 {
1611   int nnodes;
1612
1613   /* If nodep is a leaf, the count is just 1. Otherwise, the count is */
1614   /* the number of leaves in the children */
1615
1616   if (nodep->leaf==1) nnodes=1;
1617   else
1618     nnodes = countleaves(nodep->tchild) + countleaves(nodep->fchild);
1619
1620   return(nnodes);
1621 }
1622
1623 /***************************************************************************/
1624  /* printdata(datap) -- Prints the data pairs in the list datap. */
1625
1626 int printdata(datap)
1627      struct datum *datap;
1628 {
1629   struct datum *nextp, *thisp;
1630
1631   /* Go through the list on point at a time, and print each point as */
1632   /* an ordered pair (observation, class). */
1633
1634   thisp=datap;
1635   printf("leaf node data:");
1636   while (thisp!=NULL) {
1637     printf(" (%f,%d)",thisp->obsrv[0],thisp->class);
1638     thisp=thisp->link;
1639   }
1640   printf("\n");
1641 }
1642
1643
1644
```

LINE # HEADER TEXT

```
1   #include "models.h" /* All the stuff shared with autolabel... */
2
3   #define MINPTS 1       /* minimum number of pts in a leaf node to split it */
4   #define PURE 1e-6      /* don't divide nodes when the best change in */
5                          /* entropy that you'd get would be smaller than */
6                          /* this. */
7   #define TRAIN_FRAC 0.7 /* Fraction of data to use for growing the tree */
8   #define MAX_LAMBDA 0.999 /* Largest value of smoothing param ( <1) */
9   #define BUCKET_SIZE 100 /* minimum number of points in a smoothing bucket */
10
11  #define LASTSTATE 6 /* Label assumed for last (unseen) observation. Used in
12                    computing the transition probabilities. */
13
14
15  /* The following declarations set-up some global structure for representing the
16  tree and dataset. The tree will be represented using a doubly-linked list. The
17  (training) dataset will be structured into linked lists assigned to the leaf
18  nodes of the tree. Thus, each leaf node will contain a pointer to a list of the
19  dataset elements assigned to that leaf. Each internal (non-leaf) node will
20  identify the the question (and its parameters) to be used to select the true or
21  false child when descending the tree. A linked list of pointers to the leaf
22  nodes makes searching the leaves easy.*/
23
24  struct datum {
25    struct datum *link;       /* for linked-list implementation */
26    float obsrv[DOBS];        /* observation vector for this point */
27    int class;                /* classification of this point */
28  };
29
30  struct node {
31    struct node *parent;             /* pntr to parent node in tree */
32    int leaf;                        /* =1 if this is a leaf node, 0 otherwise */
33    struct node *tchild, *fchild;    /* pntrs to children nodes */
34    int class;                       /* class assigned to this node */
35    float pclass[NCLASS];            /* pr(class) for each class, at this node */
36    float pnode;                     /* pr(this node)  (sum pclass over classes */
37    int ndata;                       /* number of points assigned to this leaf */
38    struct datum *data;              /* pntr to data list assigned to this leaf */
39    int qid;                         /* id of question to be asked at this node */
40    float thresh;        /* parameters used to ask question */
41    float entropy;                   /* entropy of training points at this node */
42    int pure;                        /* flag to say we won't split the node again*/
43    int bucket;                      /* bucket number for smoothing parameter */
44  };
45
46  struct leafpntr {
47    struct leafpntr *nextleaf;  /* for linked-list implementation */
48    struct node *leafnode;      /* pntr to leaf node */
49  };
50
51  /* GLOBAL DATA STRUCTURES */
52  static float priors[NCLASS];
53  float *float_buf,*lambda,*c_b,*d_b;
54  int *class_buf,num_buckets;
55  float thresh_start[NQUESTIONS];
56  float thresh_incr[NQUESTIONS];
57  float thresh_stop[NQUESTIONS];
58  float reductions[NQUESTIONS]; /* global table used to accumulate entropy
59                                   reductions for final summary statistics */
60
61        /* These next two are used to estimate the HMM transition and initial
62           probabilities. There are global because I was lazy and didn't pass
63           them like a good boy...getdata() counts into them while reading
64           the training data, and dump_model() uses them to produce the
65           HMM parameters. Fix it when you're bored. */
66  float transitions[NCLASS][NCLASS], initials[NCLASS];
67
68
69  /* FUNCTION PROTOTYPES */
70  struct datum *append( struct datum *datap1, struct datum *datap2);
71  struct node *find_cut( struct node *nodep, float *delta, float *lentrpy,
72                         int *size, int npnts);
73  void cut_tree( struct node *nodep);
74  float classify_list( struct node *nodep, struct leafpntr **leaflist,
75                       struct datum *datap, int npnts);
76  struct leafpntr *addleaves( struct leafpntr *list1, struct leafpntr *list2);
77  void freeleaves( struct leafpntr *leaflist);
78  struct datum *strip_leaves( struct leafpntr *leaflist);
79  int count( struct datum *datap);
80  float divide( struct leafpntr *leaflist, int npnts);
81  float split(struct node *nodep, int qid, float thresh,
82              struct leafpntr *leaflist, int npnts);
83  float identify_question(struct node *nodep, int *best_qid, float *best_thresh);
84  struct node *newnode();
85  void freetree( struct node *nodep);
86  void getpclasses(struct node *nodep);
87  int getdata(char filename1[], struct datum **headp, int *npnts,
88              char filename2[], struct datum **headp2, int *npnts2);
89  struct datum *newdatum();
90  struct leafpntr *newleafpntr();
91  void treeprint( struct node *nodep);
92  void dump_model( struct node *nodep);
93  int dump_subtree( struct node *nodep, int *nnodes, int *vqnum, FILE *fp,
94                    float pclass_code[NCLASS][MAX_LEAVES], float pstate[NCLASS]);
95  int countnodes(struct node *nodep);
96  int printdata(struct datum *datap);
97
98  float identify_split( struct node *nodep, int qid, float *best_thresh);
99  int find_neighbor( int count, float val, int index,
100                   float *neighbor);
101 float try_split( struct node *nodep, int qid, float thresh);
102 float slow_try_split( struct node *nodep, int qid, float thresh);
103 float entropy( struct node *nodep);
104 void find_thresholds( struct datum *datap);
105
106 void make_buckets(struct node *nodep);
107 struct leafpntr *make_nodelist(struct node *nodep, int *npnts);
108 struct leafpntr *sort_nodelist(struct leafpntr *lpntr);
109 void smooth_probs(struct node *nodep);
110 float objective_function(struct node *nodep, struct datum *datap);
111 float re_estimate(struct node *nodep, struct datum *datap);
112 float classify(struct node *nodep, struct datum *datap);
```