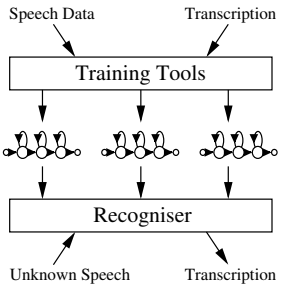
Chapter 1

The Fundamentals of HTK



HTK is a toolkit for building Hidden Markov Models (HMMs). HMMs can be used to model any time series and the core of HTK is similarly general-purpose. However, HTK is primarily designed for building HMM-based speech processing tools, in particular recognisers. Thus, much of the infrastructure support in HTK is dedicated to this task. As shown in the picture alongside, there are two major processing stages involved. Firstly, the HTK training tools are used to estimate the parameters of a set of HMMs using training utterances and their associated transcriptions. Secondly, unknown utterances are transcribed using the HTK recognition tools.

The main body of this book is mostly concerned with the mechanics of these two processes. However, before launching into detail it is necessary to understand some of the basic principles of HMMs. It is also helpful to have an overview of the toolkit and to have some appreciation of how training and recognition in HTK is organised.

This first part of the book attempts to provide this information. In this chapter, the basic ideas of HMMs and their use in speech recognition are introduced. The following chapter then presents a brief overview of HTK and, for users of older versions, it highlights the main differences in version 2.0 and later. Finally in this tutorial part of the book, chapter 3 describes how a HMM-based speech recogniser can be built using HTK. It does this by describing the construction of a simple small vocabulary continuous speech recogniser.

The second part of the book then revisits the topics skimmed over here and discusses each in detail. This can be read in conjunction with the third and final part of the book which provides a reference manual for HTK. This includes a description of each tool, summaries of the various parameters used to configure HTK and a list of the error messages that it generates when things go wrong.

Finally, note that this book is concerned only with HTK as a tool-kit. It does not provide information for using the HTK libraries as a programming environment.

1.1 General Principles of HMMs

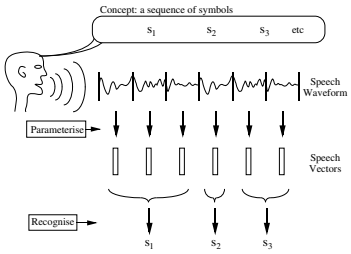


Fig. 1.1 Message Encoding/Decoding

Speech recognition systems generally assume that the speech signal is a realisation of some message encoded as a sequence of one or more symbols (see Fig. 1.1). To effect the reverse operation of recognising the underlying symbol sequence given a spoken utterance, the continuous speech waveform is first converted to a sequence of equally spaced discrete parameter vectors. This sequence of parameter vectors is assumed to form an exact representation of the speech waveform on the basis that for the duration covered by a single vector (typically 10ms or so), the speech waveform can be regarded as being stationary. Although this is not strictly true, it is a reasonable approximation. Typical parametric representations in common use are smoothed spectra or linear prediction coefficients plus various other representations derived from these.

The rôle of the recogniser is to effect a mapping between sequences of speech vectors and the wanted underlying symbol sequences. Two problems make this very difficult. Firstly, the mapping from symbols to speech is not one-to-one since different underlying symbols can give rise to similar speech sounds. Furthermore, there are large variations in the realised speech waveform due to speaker variability, mood, environment, etc. Secondly, the boundaries between symbols cannot be identified explicitly from the speech waveform. Hence, it is not possible to treat the speech waveform as a sequence of concatenated static patterns.

The second problem of not knowing the word boundary locations can be avoided by restricting the task to isolated word recognition. As shown in Fig. 1.2, this implies that the speech waveform corresponds to a single underlying symbol (e.g. word) chosen from a fixed vocabulary. Despite the fact that this simpler problem is somewhat artificial, it nevertheless has a wide range of practical applications. Furthermore, it serves as a good basis for introducing the basic ideas of HMM-based recognition before dealing with the more complex continuous speech case. Hence, isolated word recognition using HMMs will be dealt with first.

1.2 Isolated Word Recognition

Let each spoken word be represented by a sequence of speech vectors or *observations* $\boldsymbol{O}$, defined as

$$\boldsymbol{O} = \boldsymbol{o}_1, \boldsymbol{o}_2, \ldots, \boldsymbol{o}_T \tag{1.1}$$

where $\boldsymbol{o}_t$ is the speech vector observed at time t. The isolated word recognition problem can then be regarded as that of computing

$$\arg\max_i \{P(w_i|\boldsymbol{O})\} \tag{1.2}$$

where w_i is the i'th vocabulary word. This probability is not computable directly but using Bayes' Rule gives

$$P(w_i|\boldsymbol{O}) = \frac{P(\boldsymbol{O}|w_i)P(w_i)}{P(\boldsymbol{O})} \tag{1.3}$$

Thus, for a given set of prior probabilities $P(w_i)$, the most probable spoken word depends only on the likelihood $P(\boldsymbol{O}|w_i)$. Given the dimensionality of the observation sequence $\boldsymbol{O}$, the direct estimation of the joint conditional probability $P(\boldsymbol{o}_1, \boldsymbol{o}_2, \ldots|w_i)$ from examples of spoken words is not practicable. However, if a parametric model of word production such as a Markov model is

assumed, then estimation from data is possible since the problem of estimating the class conditional observation densities $P(\boldsymbol{O}|w_i)$ is replaced by the much simpler problem of estimating the Markov model parameters.

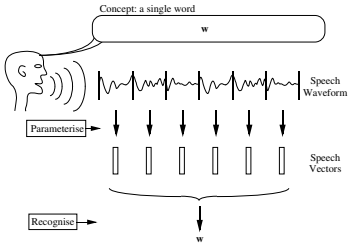


Fig. 1.2 Isolated Word Problem

I n HMM based speech recognition, it is assumed that the sequence of observed speech vectors corresponding to each word is generated by a Markov model as shown in Fig. 1.3. A Markov model is a finite state machine which changes state once every time unit and each time t that a state j is entered, a speech vector $\boldsymbol{o}_t$ is generated from the probability density $b_j(\boldsymbol{o}_t)$. Furthermore, the transition from state i to state j is also probabilistic and is governed by the discrete probability a_{ij}. Fig. 1.3 shows an example of this process where the six state model moves through the state sequence $X = 1, 2, 2, 3, 4, 4, 5, 6$ in order to generate the sequence $\boldsymbol{o}_1$ to $\boldsymbol{o}_6$. Notice that in HTK, the entry and exit states of a HMM are non-emitting. This is to facilitate the construction of composite models as explained in more detail later.

The joint probability that $\boldsymbol{O}$ is generated by the model M moving through the state sequence X is calculated simply as the product of the transition probabilities and the output probabilities. So for the state sequence X in Fig. 1.3

$$P(\boldsymbol{O}, X|M) = a_{12}b_2(\boldsymbol{o}_1)a_{22}b_2(\boldsymbol{o}_2)a_{23}b_3(\boldsymbol{o}_3)\ldots \tag{1.4}$$

However, in practice, only the observation sequence $\boldsymbol{O}$ is known and the underlying state sequence X is hidden. This is why it is called a *Hidden Markov Model*.

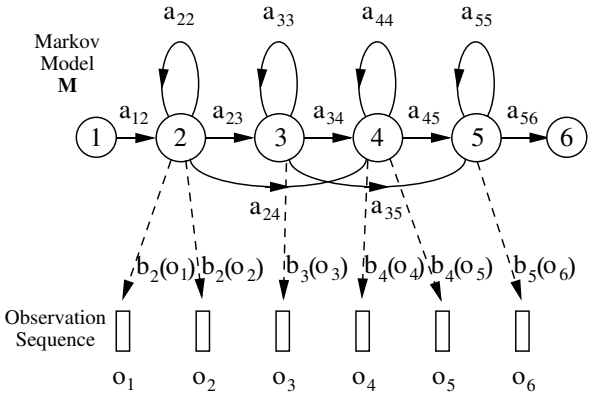


Fig. 1.3 The Markov Generation Model

Given that X is unknown, the required likelihood is computed by summing over all possible state sequences $X = x(1), x(2), x(3), \ldots, x(T)$, that is

$$P(\boldsymbol{O}|M) = \sum_X a_{x(0)x(1)} \prod_{t=1}^{T} b_{x(t)}(\boldsymbol{o}_t)a_{x(t)x(t+1)} \tag{1.5}$$

where $x(0)$ is constrained to be the model entry state and $x(T+1)$ is constrained to be the model exit state.

As an alternative to equation 1.5, the likelihood can be approximated by only considering the most likely state sequence, that is

$$\hat{P}(\boldsymbol{O}|M) = \max_{X}\left\{a_{x(0)x(1)}\prod_{t=1}^{T} b_{x(t)}(\boldsymbol{o}_t)a_{x(t)x(t+1)}\right\} \tag{1.6}$$

Although the direct computation of equations 1.5 and 1.6 is not tractable, simple recursive procedures exist which allow both quantities to be calculated very efficiently. Before going any further, however, notice that if equation 1.2 is computable then the recognition problem is solved. Given a set of models M_i corresponding to words w_i, equation 1.2 is solved by using 1.3 and assuming that

$$P(\boldsymbol{O}|w_i) = P(\boldsymbol{O}|M_i). \tag{1.7}$$

All this, of course, assumes that the parameters $\{a_{ij}\}$ and $\{b_j(\boldsymbol{o}_t)\}$ are known for each model M_i. Herein lies the elegance and power of the HMM framework. Given a set of training examples corresponding to a particular model, the parameters of that model can be determined automatically by a robust and efficient re-estimation procedure. Thus, provided that a sufficient number of representative examples of each word can be collected then a HMM can be constructed which implicitly models all of the many sources of variability inherent in real speech. Fig. 1.4 summarises the use of HMMs for isolated word recognition. Firstly, a HMM is trained for each vocabulary word using a number of examples of that word. In this case, the vocabulary consists of just three words: "one", "two" and "three". Secondly, to recognise some unknown word, the likelihood of each model generating that word is calculated and the most likely model identifies the word.

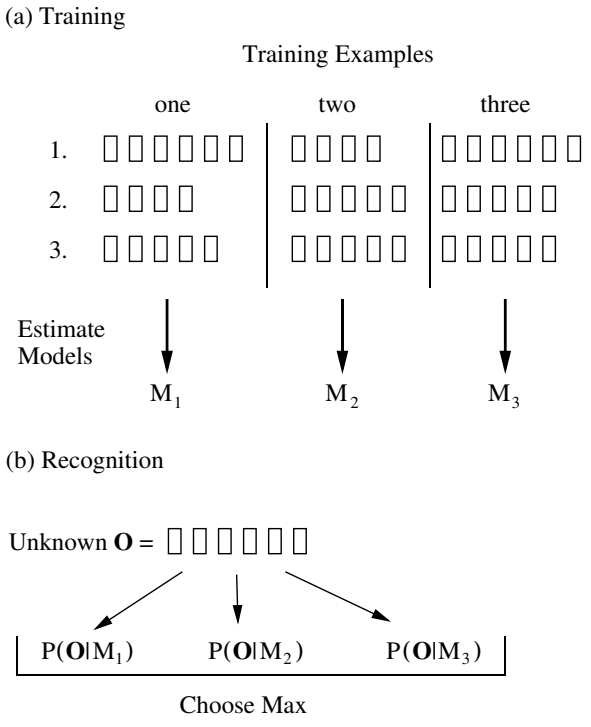


Fig. 1.4 Using HMMs for Isolated Word Recognition

1.3 Output Probability Specification

Before the problem of parameter estimation can be discussed in more detail, the form of the output distributions $\{b_j(\boldsymbol{o}_t)\}$ needs to be made explicit. HTK is designed primarily for modelling continuous parameters using continuous density multivariate output distributions. It can also handle observation sequences consisting of discrete symbols in which case, the output distributions are discrete probabilities. For simplicity, however, the presentation in this chapter will assume that continuous density distributions are being used. The minor differences that the use of discrete probabilities entail are noted in chapter 7 and discussed in more detail in chapter 11.

In common with most other continuous density HMM systems, HTK represents output distributions by Gaussian Mixture Densities. In HTK, however, a further generalisation is made. HTK allows each observation vector at time t to be split into a number of S independent data streams o_{st}. The formula for computing $b_j(\boldsymbol{o}_t)$ is then

$$b_j(\boldsymbol{o}_t) = \prod_{s=1}^{S} \left[\sum_{m=1}^{M_s} c_{jsm} \mathcal{N}(\boldsymbol{o}_{st}; \boldsymbol{\mu}_{jsm}, \boldsymbol{\Sigma}_{jsm}) \right]^{\gamma_s} \tag{1.8}$$

where M_s is the number of mixture components in stream s, c_{jsm} is the weight of the m'th component and $\mathcal{N}(\cdot; \boldsymbol{\mu}, \boldsymbol{\Sigma})$ is a multivariate Gaussian with mean vector $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$, that is

$$\mathcal{N}(\boldsymbol{o}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{\sqrt{(2\pi)^n |\boldsymbol{\Sigma}|}} e^{-\frac{1}{2}(\boldsymbol{o}-\boldsymbol{\mu})' \boldsymbol{\Sigma}^{-1} (\boldsymbol{o}-\boldsymbol{\mu})} \tag{1.9}$$

where n is the dimensionality of $\boldsymbol{o}$.

The exponent γ_s is a stream weight[1]. It can be used to give a particular stream more emphasis, however, it can only be set manually. No current HTK training tools can estimate values for it.

Multiple data streams are used to enable separate modelling of multiple information sources. In HTK, the processing of streams is completely general. However, the speech input modules assume that the source data is split into at most 4 streams. Chapter 5 discusses this in more detail but for now it is sufficient to remark that the default streams are the basic parameter vector, first (delta) and second (acceleration) difference coefficients and log energy.

1.4 Baum-Welch Re-Estimation

To determine the parameters of a HMM it is first necessary to make a rough guess at what they might be. Once this is done, more accurate (in the maximum likelihood sense) parameters can be found by applying the so-called Baum-Welch re-estimation formulae.

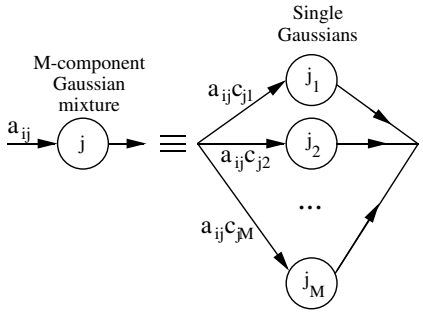


Fig. 1.5 Representing a Mixture

Chapter 8 gives the formulae used in HTK in full detail. Here the basis of the formulae will be presented in a very informal way. Firstly, it should be noted that the inclusion of multiple data streams does not alter matters significantly since each stream is considered to be statistically

[1] often referred to as a codebook exponent.

independent. Furthermore, mixture components can be considered to be a special form of sub-state in which the transition probabilities are the mixture weights (see Fig. 1.5).

Thus, the essential problem is to estimate the means and variances of a HMM in which each state output distribution is a single component Gaussian, that is

$$b_j(\boldsymbol{o}_t) = \frac{1}{\sqrt{(2\pi)^n|\boldsymbol{\Sigma_j}|}} e^{-\frac{1}{2}(\boldsymbol{o}_t-\boldsymbol{\mu}_j)'\boldsymbol{\Sigma}_j^{-1}(\boldsymbol{o}_t-\boldsymbol{\mu}_j)} \tag{1.10}$$

If there was just one state j in the HMM, this parameter estimation would be easy. The maximum likelihood estimates of $\boldsymbol{\mu}_j$ and $\boldsymbol{\Sigma}_j$ would be just the simple averages, that is

$$\hat{\boldsymbol{\mu}}_j = \frac{1}{T}\sum_{t=1}^{T} \boldsymbol{o}_t \tag{1.11}$$

and

$$\hat{\boldsymbol{\Sigma}}_j = \frac{1}{T}\sum_{t=1}^{T} (\boldsymbol{o}_t - \boldsymbol{\mu}_j)(\boldsymbol{o}_t - \boldsymbol{\mu}_j)' \tag{1.12}$$

In practice, of course, there are multiple states and there is no direct assignment of observation vectors to individual states because the underlying state sequence is unknown. Note, however, that if some approximate assignment of vectors to states could be made then equations 1.11 and 1.12 could be used to give the required initial values for the parameters. Indeed, this is exactly what is done in the HTK tool called HInit. HInit first divides the training observation vectors equally amongst the model states and then uses equations 1.11 and 1.12 to give initial values for the mean and variance of each state. It then finds the maximum likelihood state sequence using the Viterbi algorithm described below, reassigns the observation vectors to states and then uses equations 1.11 and 1.12 again to get better initial values. This process is repeated until the estimates do not change.

Since the full likelihood of each observation sequence is based on the summation of all possible state sequences, each observation vector $\boldsymbol{o}_t$ contributes to the computation of the maximum likelihood parameter values for each state j. In other words, instead of assigning each observation vector to a specific state as in the above approximation, each observation is assigned to every state in proportion to the probability of the model being in that state when the vector was observed. Thus, if $L_j(t)$ denotes the probability of being in state j at time t then the equations 1.11 and 1.12 given above become the following weighted averages

$$\hat{\boldsymbol{\mu}}_j = \frac{\sum_{t=1}^{T} L_j(t)\boldsymbol{o}_t}{\sum_{t=1}^{T} L_j(t)} \tag{1.13}$$

and

$$\hat{\boldsymbol{\Sigma}}_j = \frac{\sum_{t=1}^{T} L_j(t)(\boldsymbol{o}_t - \boldsymbol{\mu}_j)(\boldsymbol{o}_t - \boldsymbol{\mu}_j)'}{\sum_{t=1}^{T} L_j(t)} \tag{1.14}$$

where the summations in the denominators are included to give the required normalisation.

Equations 1.13 and 1.14 are the Baum-Welch re-estimation formulae for the means and covariances of a HMM. A similar but slightly more complex formula can be derived for the transition probabilities (see chapter 8).

Of course, to apply equations 1.13 and 1.14, the probability of state occupation $L_j(t)$ must be calculated. This is done efficiently using the so-called *Forward-Backward* algorithm. Let the forward probability[2] $\alpha_j(t)$ for some model M with N states be defined as

$$\alpha_j(t) = P(\boldsymbol{o}_1, \ldots, \boldsymbol{o}_t, x(t) = j|M). \tag{1.15}$$

That is, $\alpha_j(t)$ is the joint probability of observing the first t speech vectors and being in state j at time t. This forward probability can be efficiently calculated by the following recursion

$$\alpha_j(t) = \left[\sum_{i=2}^{N-1} \alpha_i(t-1)a_{ij}\right] b_j(\boldsymbol{o}_t). \tag{1.16}$$

[2] Since the output distributions are densities, these are not really probabilities but it is a convenient fiction.

This recursion depends on the fact that the probability of being in state j at time t and seeing observation $\boldsymbol{o}_t$ can be deduced by summing the forward probabilities for all possible predecessor states i weighted by the transition probability a_{ij}. The slightly odd limits are caused by the fact that states 1 and N are non-emitting[3]. The initial conditions for the above recursion are

$$\alpha_1(1) = 1 \tag{1.17}$$

$$\alpha_j(1) = a_{1j} b_j(\boldsymbol{o}_1) \tag{1.18}$$

for $1 < j < N$ and the final condition is given by

$$\alpha_N(T) = \sum_{i=2}^{N-1} \alpha_i(T) a_{iN}. \tag{1.19}$$

Notice here that from the definition of $\alpha_j(t)$,

$$P(\boldsymbol{O}|M) = \alpha_N(T). \tag{1.20}$$

Hence, the calculation of the forward probability also yields the total likelihood $P(\boldsymbol{O}|M)$.

The backward probability $\beta_j(t)$ is defined as

$$\beta_j(t) = P(\boldsymbol{o}_{t+1}, \ldots, \boldsymbol{o}_T | x(t) = j, M). \tag{1.21}$$

As in the forward case, this backward probability can be computed efficiently using the following recursion

$$\beta_i(t) = \sum_{j=2}^{N-1} a_{ij} b_j(\boldsymbol{o}_{t+1}) \beta_j(t+1) \tag{1.22}$$

with initial condition given by

$$\beta_i(T) = a_{iN} \tag{1.23}$$

for $1 < i < N$ and final condition given by

$$\beta_1(1) = \sum_{j=2}^{N-1} a_{1j} b_j(\boldsymbol{o}_1) \beta_j(1). \tag{1.24}$$

Notice that in the definitions above, the forward probability is a joint probability whereas the backward probability is a conditional probability. This somewhat asymmetric definition is deliberate since it allows the probability of state occupation to be determined by taking the product of the two probabilities. From the definitions,

$$\alpha_j(t)\beta_j(t) = P(\boldsymbol{O}, x(t) = j | M). \tag{1.25}$$

Hence,

$$\begin{aligned} L_j(t) &= P(x(t) = j | \boldsymbol{O}, M) \\ &= \frac{P(\boldsymbol{O}, x(t) = j | M)}{P(\boldsymbol{O}|M)} \\ &= \frac{1}{P} \alpha_j(t) \beta_j(t) \end{aligned} \tag{1.26}$$

where $P = P(\boldsymbol{O}|M)$.

All of the information needed to perform HMM parameter re-estimation using the Baum-Welch algorithm is now in place. The steps in this algorithm may be summarised as follows

1. For every parameter vector/matrix requiring re-estimation, allocate storage for the numerator and denominator summations of the form illustrated by equations 1.13 and 1.14. These storage locations are referred to as *accumulators*[4].

[3] To understand equations involving a non-emitting state at time t, the time should be thought of as being $t - \delta t$ if it is an entry state, and $t + \delta t$ if it is an exit state. This becomes important when HMMs are connected together in sequence so that transitions across non-emitting states take place *between frames*.

[4] Note that normally the summations in the denominators of the re-estimation formulae are identical across the parameter sets of a given state and therefore only a single common storage location for the denominators is required and it need only be calculated once. However, HTK supports a generalised parameter tying mechanism which can result in the denominator summations being different. Hence, in HTK the denominator summations are always stored and calculated individually for each distinct parameter vector or matrix.

2. Calculate the forward and backward probabilities for all states j and times t.
3. For each state j and time t, use the probability $L_j(t)$ and the current observation vector $\boldsymbol{o}_t$ to update the accumulators for that state.
4. Use the final accumulator values to calculate new parameter values.
5. If the value of $P = P(\boldsymbol{O}|M)$ for this iteration is not higher than the value at the previous iteration then stop, otherwise repeat the above steps using the new re-estimated parameter values.

All of the above assumes that the parameters for a HMM are re-estimated from a single observation sequence, that is a single example of the spoken word. In practice, many examples are needed to get good parameter estimates. However, the use of multiple observation sequences adds no additional complexity to the algorithm. Steps 2 and 3 above are simply repeated for each distinct training sequence.

One final point that should be mentioned is that the computation of the forward and backward probabilities involves taking the product of a large number of probabilities. In practice, this means that the actual numbers involved become very small. Hence, to avoid numerical problems, the forward-backward computation is computed in HTK using log arithmetic.

The HTK program which implements the above algorithm is called HRest. In combination with the tool HInit for estimating initial values mentioned earlier, HRest allows isolated word HMMs to be constructed from a set of training examples using Baum-Welch re-estimation.

1.5 Recognition and Viterbi Decoding

The previous section has described the basic ideas underlying HMM parameter re-estimation using the Baum-Welch algorithm. In passing, it was noted that the efficient recursive algorithm for computing the forward probability also yielded as a by-product the total likelihood $P(\boldsymbol{O}|M)$. Thus, this algorithm could also be used to find the model which yields the maximum value of $P(\boldsymbol{O}|M_i)$, and hence, it could be used for recognition.

In practice, however, it is preferable to base recognition on the maximum likelihood state sequence since this generalises easily to the continuous speech case whereas the use of the total probability does not. This likelihood is computed using essentially the same algorithm as the forward probability calculation except that the summation is replaced by a maximum operation. For a given model M, let $\phi_j(t)$ represent the maximum likelihood of observing speech vectors $\boldsymbol{o}_1$ to $\boldsymbol{o}_t$ and being in state j at time t. This partial likelihood can be computed efficiently using the following recursion (cf. equation 1.16)

$$\phi_j(t) = \max_i \{\phi_i(t-1)a_{ij}\}\, b_j(\boldsymbol{o}_t). \tag{1.27}$$

where

$$\phi_1(1) = 1 \tag{1.28}$$

$$\phi_j(1) = a_{1j} b_j(\boldsymbol{o}_1) \tag{1.29}$$

for $1 < j < N$. The maximum likelihood $\hat{P}(O|M)$ is then given by

$$\phi_N(T) = \max_i \{\phi_i(T)a_{iN}\} \tag{1.30}$$

As for the re-estimation case, the direct computation of likelihoods leads to underflow, hence, log likelihoods are used instead. The recursion of equation 1.27 then becomes

$$\psi_j(t) = \max_i \{\psi_i(t-1) + log(a_{ij})\} + log(b_j(\boldsymbol{o}_t)). \tag{1.31}$$

This recursion forms the basis of the so-called Viterbi algorithm. As shown in Fig. 1.6, this algorithm can be visualised as finding the best path through a matrix where the vertical dimension represents the states of the HMM and the horizontal dimension represents the frames of speech (i.e. time). Each large dot in the picture represents the log probability of observing that frame at that time and each arc between dots corresponds to a log transition probability. The log probability of any path is computed simply by summing the log transition probabilities and the log output probabilities along that path. The paths are grown from left-to-right column-by-column. At time t, each partial path $\psi_i(t-1)$ is known for all states i, hence equation 1.31 can be used to compute $\psi_j(t)$ thereby extending the partial paths by one time frame.

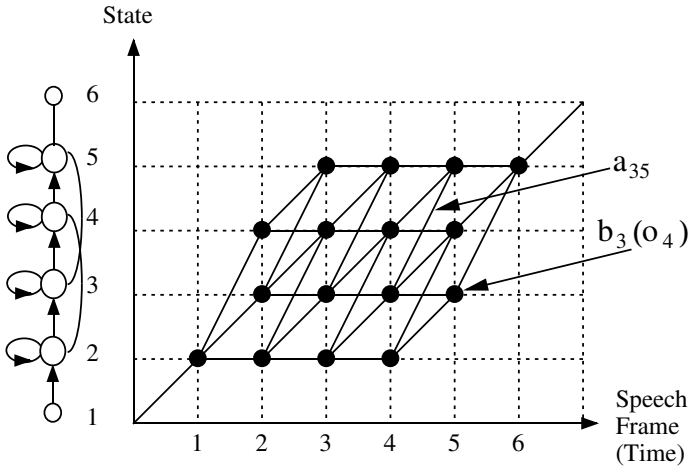


Fig. 1.6 The Viterbi Algorithm for Isolated Word Recognition

This concept of a path is extremely important and it is generalised below to deal with the continuous speech case.

This completes the discussion of isolated word recognition using HMMs. There is no HTK tool which implements the above Viterbi algorithm directly. Instead, a tool called HVite is provided which along with its supporting libraries, HNet and HRec, is designed to handle continuous speech. Since this recogniser is syntax directed, it can also perform isolated word recognition as a special case. This is discussed in more detail below.

1.6 Continuous Speech Recognition

Returning now to the conceptual model of speech production and recognition exemplified by Fig. 1.1, it should be clear that the extension to continuous speech simply involves connecting HMMs together in sequence. Each model in the sequence corresponds directly to the assumed underlying symbol. These could be either whole words for so-called *connected speech recognition* or sub-words such as phonemes for *continuous speech recognition*. The reason for including the non-emitting entry and exit states should now be evident, these states provide the *glue* needed to join models together.

There are, however, some practical difficulties to overcome. The training data for continuous speech must consist of continuous utterances and, in general, the boundaries dividing the segments of speech corresponding to each underlying sub-word model in the sequence will not be known. In practice, it is usually feasible to mark the boundaries of a small amount of data by hand. All of the segments corresponding to a given model can then be extracted and the *isolated word* style of training described above can be used. However, the amount of data obtainable in this way is usually very limited and the resultant models will be poor estimates. Furthermore, even if there was a large amount of data, the boundaries imposed by hand-marking may not be optimal as far as the HMMs are concerned. Hence, in HTK the use of HInit and HRest for initialising sub-word models is regarded as a *bootstrap* operation[5]. The main training phase involves the use of a tool called HERest which does *embedded training*.

Embedded training uses the same Baum-Welch procedure as for the isolated case but rather than training each model individually all models are trained in parallel. It works in the following steps:

1. Allocate and zero accumulators for all parameters of all HMMs.
2. Get the next training utterance.

[5] They can even be avoided altogether by using a *flat start* as described in section 8.3.

3. Construct a composite HMM by joining in sequence the HMMs corresponding to the symbol transcription of the training utterance.

4. Calculate the forward and backward probabilities for the composite HMM. The inclusion of intermediate non-emitting states in the composite model requires some changes to the computation of the forward and backward probabilities but these are only minor. The details are given in chapter 8.

5. Use the forward and backward probabilities to compute the probabilities of state occupation at each time frame and update the accumulators in the usual way.

6. Repeat from 2 until all training utterances have been processed.

7. Use the accumulators to calculate new parameter estimates for all of the HMMs.

These steps can then all be repeated as many times as is necessary to achieve the required convergence. Notice that although the location of symbol boundaries in the training data is not required (or wanted) for this procedure, the symbolic transcription of each training utterance is needed.

Whereas the extensions needed to the Baum-Welch procedure for training sub-word models are relatively minor[6], the corresponding extensions to the Viterbi algorithm are more substantial.

In HTK, an alternative formulation of the Viterbi algorithm is used called the *Token Passing Model* [7]. In brief, the token passing model makes the concept of a state alignment path explicit. Imagine each state j of a HMM at time t holds a single moveable token which contains, amongst other information, the partial log probability $\psi_j(t)$. This token then represents a partial match between the observation sequence $\boldsymbol{o}_1$ to $\boldsymbol{o}_t$ and the model subject to the constraint that the model is in state j at time t. The path extension algorithm represented by the recursion of equation 1.31 is then replaced by the equivalent *token passing algorithm* which is executed at each time frame t. The key steps in this algorithm are as follows

1. Pass a copy of every token in state i to all connecting states j, incrementing the log probability of the copy by $log[a_{ij}] + log[b_j(\boldsymbol{o}(t)]$.

2. Examine the tokens in every state and discard all but the token with the highest probability.

In practice, some modifications are needed to deal with the non-emitting states but these are straightforward if the tokens in entry states are assumed to represent paths extended to time $t - \delta t$ and tokens in exit states are assumed to represent paths extended to time $t + \delta t$.

The point of using the Token Passing Model is that it extends very simply to the continuous speech case. Suppose that the allowed sequence of HMMs is defined by a finite state network. For example, Fig. 1.7 shows a simple network in which each word is defined as a sequence of phoneme-based HMMs and all of the words are placed in a loop. In this network, the oval boxes denote HMM instances and the square boxes denote *word-end* nodes. This composite network is essentially just a single large HMM and the above Token Passing algorithm applies. The only difference now is that more information is needed beyond the log probability of the best token. When the best token reaches the end of the speech, the route it took through the network must be known in order to recover the recognised sequence of models.

[6] In practice, a good deal of extra work is needed to achieve efficient operation on large training databases. For example, the HERest tool includes facilities for pruning on both the forward and backward passes and parallel operation on a network of machines.

[7] See "Token Passing: a Conceptual Model for Connected Speech Recognition Systems", SJ Young, NH Russell and JHS Thornton, CUED Technical Report F_INFENG/TR38, Cambridge University, 1989. Available by anonymous ftp from `svr-ftp.eng.cam.ac.uk`.

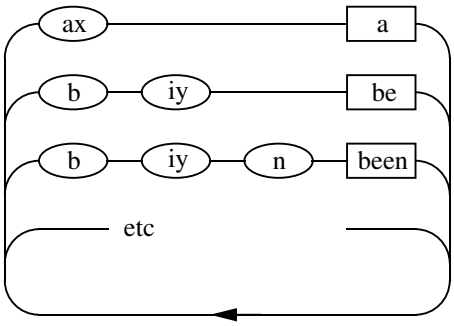


Fig. 1.7 Recognition Network for Continuously Spoken Word Recognition

The history of a token's route through the network may be recorded efficiently as follows. Every token carries a pointer called a *word end link*. When a token is propagated from the exit state of a word (indicated by passing through a word-end node) to the entry state of another, that transition represents a potential word boundary. Hence a record called a *Word Link Record* is generated in which is stored the identity of the word from which the token has just emerged and the current value of the token's link. The token's actual link is then replaced by a pointer to the newly created WLR. Fig. 1.8 illustrates this process.

Once all of the unknown speech has been processed, the WLRs attached to the link of the best matching token (i.e. the token with the highest log probability) can be traced back to give the best matching sequence of words. At the same time the positions of the word boundaries can also be extracted if required.

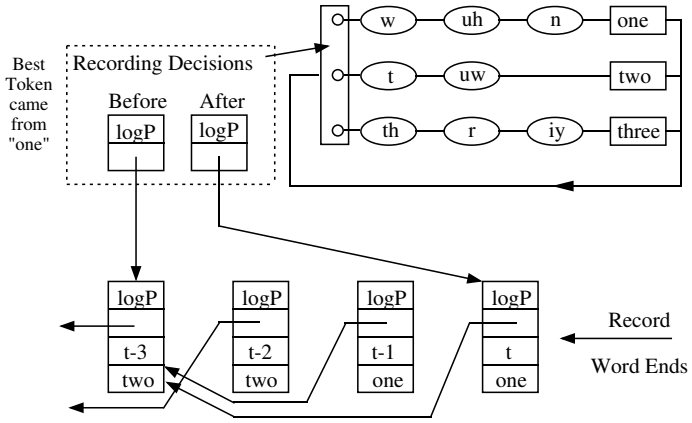


Fig. 1.8 Recording Word Boundary Decisions

The token passing algorithm for continuous speech has been described in terms of recording the word sequence only. If required, the same principle can be used to record decisions at the model and state level. Also, more than just the best token at each word boundary can be saved. This gives the potential for generating a lattice of hypotheses rather than just the single best hypothesis. Algorithms based on this idea are called *lattice N-best*. They are suboptimal because the use of a single token per state limits the number of different token histories that can be maintained. This limitation can be avoided by allowing each model state to hold multiple-tokens and regarding tokens as distinct if they come from different preceding words. This gives a class of algorithm called *word*

N-best which has been shown empirically to be comparable in performance to an optimal N-best algorithm.

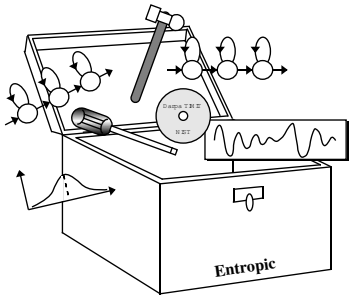
The above outlines the main idea of Token Passing as it is implemented within HTK. The algorithms are embedded in the library modules HNet and HRec and they may be invoked using the recogniser tool called HVite. They provide single and multiple-token passing recognition, single-best output, lattice output, N-best lists, support for cross-word context-dependency, lattice rescoring and forced alignment.

1.7 Speaker Adaptation

Although the training and recognition techniques described previously can produce high performance recognition systems, these systems can be improved upon by customising the HMMs to the characteristics of a particular speaker. HTK provides the tools HEAdapt and HVite to perform adaptation using a small amount of enrollment or adaptation data. The two tools differ in that HEAdapt performs offline supervised adaptation while HVite recognises the adaptation data and uses the generated transcriptions to perform the adaptation. Generally, more robust adaptation is performed in a supervised mode, as provided by HEAdapt, but given an initial well trained model set, HVite can still achieve noticeable improvements in performance. Full details of adaptation and how it is used in HTK can be found in Chapter 9.

Chapter 2

An Overview of the HTK Toolkit



The basic principles of HMM-based recognition were outlined in the previous chapter and a number of the key HTK tools have already been mentioned. This chapter describes the software architecture of a HTK tool. It then gives a brief outline of all the HTK tools and the way that they are used together to construct and test HMM-based recognisers. For the benefit of existing HTK users, the major changes in recent versions of HTK are listed. The following chapter will then illustrate the use of the HTK toolkit by working through a practical example of building a simple continuous speech recognition system.

2.1 HTK Software Architecture

Much of the functionality of HTK is built into the library modules. These modules ensure that every tool interfaces to the outside world in exactly the same way. They also provide a central resource of commonly used functions. Fig. 2.1 illustrates the software structure of a typical HTK tool and shows its input/output interfaces.

User input/output and interaction with the operating system is controlled by the library module HSHELL and all memory management is controlled by HMEM. Math support is provided by HMATH and the signal processing operations needed for speech analysis are in HSIGP. Each of the file types required by HTK has a dedicated interface module. HLABEL provides the interface for label files, HLM for language model files, HNET for networks and lattices, HDICT for dictionaries, HVQ for VQ codebooks and HMODEL for HMM definitions.

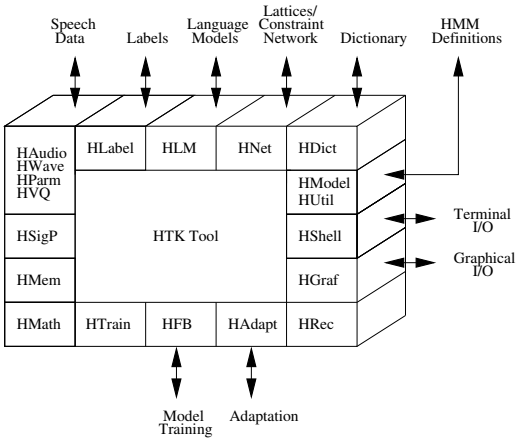


Fig. 2.1 Software Architecture

All speech input and output at the waveform level is via HWAVE and at the parameterised level via HPARM. As well as providing a consistent interface, HWAVE and HLABEL support multiple file formats allowing data to be imported from other systems. Direct audio input is supported by HAUDIO and simple interactive graphics is provided by HGRAF. HUTIL provides a number of utility routines for manipulating HMMs while HTRAIN and HFB contain support for the various HTK training tools. HADAPT provides support for the various HTK adaptation tools. Finally, HREC contains the main recognition processing functions.

As noted in the next section, fine control over the behaviour of these library modules is provided by setting configuration variables. Detailed descriptions of the functions provided by the library modules are given in the second part of this book and the relevant configuration variables are described as they arise. For reference purposes, a complete list is given in chapter 15.

2.2 Generic Properties of a HTK Tool

HTK tools are designed to run with a traditional command-line style interface. Each tool has a number of required arguments plus optional arguments. The latter are always prefixed by a minus sign. As an example, the following command would invoke the mythical HTK tool called HFOO

```
HFoo -T 1 -f 34.3 -a -s myfile file1 file2
```

This tool has two main arguments called `file1` and `file2` plus four optional arguments. Options are always introduced by a single letter option name followed where appropriate by the option value. The option value is always separated from the option name by a space. Thus, the value of the `-f` option is a real number, the value of the `-T` option is an integer number and the value of the `-s` option is a string. The `-a` option has no following value and it is used as a simple flag to enable or disable some feature of the tool. Options whose names are a capital letter have the same meaning across all tools. For example, the `-T` option is always used to control the trace output of a HTK tool.

In addition to command line arguments, the operation of a tool can be controlled by parameters stored in a configuration file. For example, if the command

```
HFoo -C config -f 34.3 -a -s myfile file1 file2
```

is executed, the tool HFOO will load the parameters stored in the configuration file `config` during its initialisation procedures. Configuration parameters can sometimes be used as an alternative to using command line arguments. For example, trace options can always be set within a configuration file. However, the main use of configuration files is to control the detailed behaviour of the library modules on which all HTK tools depend.

Although this style of command-line working may seem old-fashioned when compared to modern graphical user interfaces, it has many advantages. In particular, it makes it simple to write shell

scripts to control HTK tool execution. This is vital for performing large-scale system building and experimentation. Furthermore, defining all operations using text-based commands allows the details of system construction or experimental procedure to be recorded and documented.

Finally, note that a summary of the command line and options for any HTK tool can be obtained simply by executing the tool with no arguments.

2.3 The Toolkit

The HTK tools are best introduced by going through the processing steps involved in building a sub-word based continuous speech recogniser. As shown in Fig. 2.2, there are 4 main phases: data preparation, training, testing and analysis.

2.3.1 Data Preparation Tools

In order to build a set of HMMs, a set of speech data files and their associated transcriptions are required. Very often speech data will be obtained from database archives, typically on CD-ROMs. Before it can be used in training, it must be converted into the appropriate parametric form and any associated transcriptions must be converted to have the correct format and use the required phone or word labels. If the speech needs to be recorded, then the tool HSLab can be used both to record the speech and to manually annotate it with any required transcriptions.

Although all HTK tools can parameterise waveforms *on-the-fly*, in practice it is usually better to parameterise the data just once. The tool HCopy is used for this. As the name suggests, HCopy is used to copy one or more source files to an output file. Normally, HCopy copies the whole file, but a variety of mechanisms are provided for extracting segments of files and concatenating files. By setting the appropriate configuration variables, all input files can be converted to parametric form as they are read-in. Thus, simply copying each file in this manner performs the required encoding. The tool HList can be used to check the contents of any speech file and since it can also convert input on-the-fly, it can be used to check the results of any conversions before processing large quantities of data. Transcriptions will also need preparing. Typically the labels used in the original source transcriptions will not be exactly as required, for example, because of differences in the phone sets used. Also, HMM training might require the labels to be context-dependent. The tool HLEd is a script-driven label editor which is designed to make the required transformations to label files. HLEd can also output files to a single *Master Label File* MLF which is usually more convenient for subsequent processing. Finally on data preparation, HLStats can gather and display statistics on label files and where required, HQuant can be used to build a VQ codebook in preparation for building discrete probability HMM system.

2.3.2 Training Tools

The second step of system building is to define the topology required for each HMM by writing a prototype definition. HTK allows HMMs to be built with any desired topology. HMM definitions can be stored externally as simple text files and hence it is possible to edit them with any convenient text editor. Alternatively, the standard HTK distribution includes a number of example HMM prototypes and a script to generate the most common topologies automatically. With the exception of the transition probabilities, all of the HMM parameters given in the prototype definition are ignored. The purpose of the prototype definition is only to specify the overall characteristics and topology of the HMM. The actual parameters will be computed later by the training tools. Sensible values for the transition probabilities must be given but the training process is very insensitive to these. An acceptable and simple strategy for choosing these probabilities is to make all of the transitions out of any state equally likely.

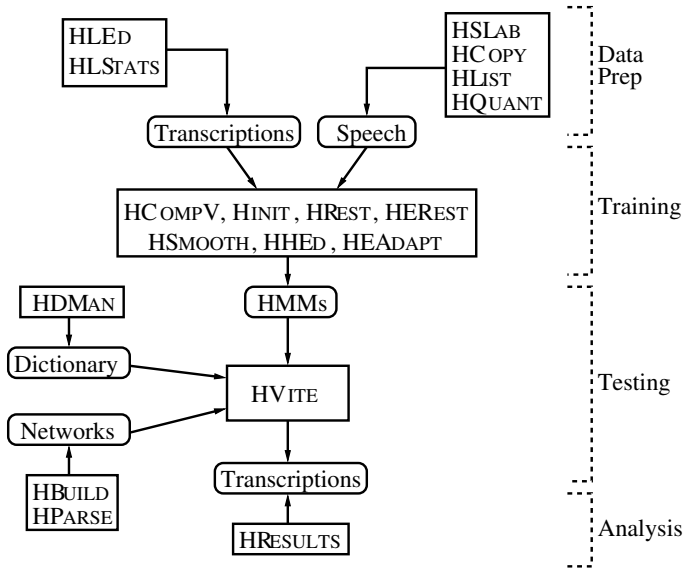


Fig. 2.2 HTK Processing Stages

The actual training process takes place in stages and it is illustrated in more detail in Fig. 2.3. Firstly, an initial set of models must be created. If there is some speech data available for which the location of the sub-word (i.e. phone) boundaries have been marked, then this can be used as *bootstrap data*. In this case, the tools HINIT and HREST provide *isolated word* style training using the fully labelled bootstrap data. Each of the required HMMs is generated individually. HINIT reads in all of the bootstrap training data and *cuts out* all of the examples of the required phone. It then iteratively computes an initial set of parameter values using a *segmental k-means* procedure. On the first cycle, the training data is uniformly segmented, each model state is matched with the corresponding data segments and then means and variances are estimated. If mixture Gaussian models are being trained, then a modified form of k-means clustering is used. On the second and successive cycles, the uniform segmentation is replaced by Viterbi alignment. The initial parameter values computed by HINIT are then further re-estimated by HREST. Again, the fully labelled bootstrap data is used but this time the segmental k-means procedure is replaced by the Baum-Welch re-estimation procedure described in the previous chapter. When no bootstrap data is available, a so-called *flat start* can be used. In this case all of the phone models are initialised to be identical and have state means and variances equal to the global speech mean and variance. The tool HCOMPV can be used for this.

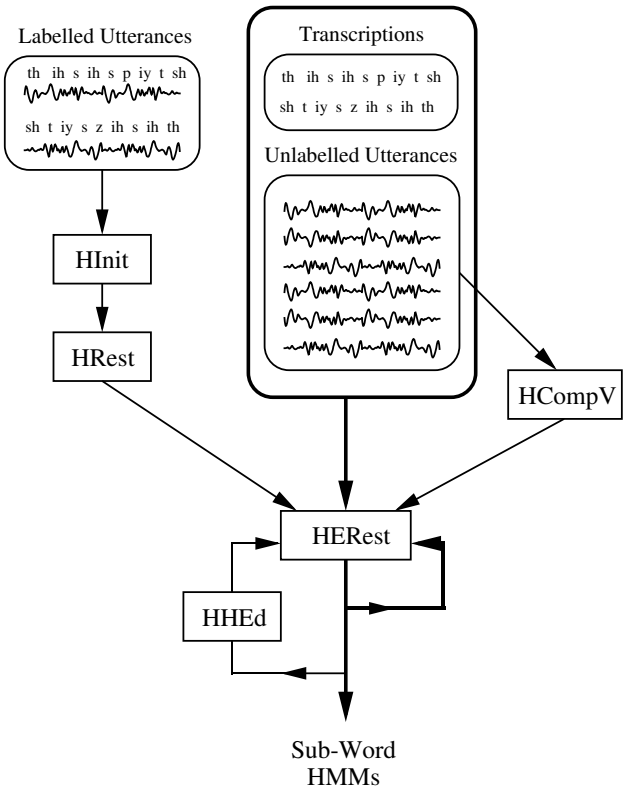


Fig. 2.3 Training Sub-word HMMs

Once an initial set of models has been created, the tool HERest is used to perform *embedded training* using the entire training set. HERest performs a single Baum-Welch re-estimation of the whole set of HMM phone models simultaneously. For each training utterance, the corresponding phone models are concatenated and then the forward-backward algorithm is used to accumulate the statistics of state occupation, means, variances, etc., for each HMM in the sequence. When all of the training data has been processed, the accumulated statistics are used to compute re-estimates of the HMM parameters. HERest is the core HTK training tool. It is designed to process large databases, it has facilities for pruning to reduce computation and it can be run in parallel across a network of machines.

The philosophy of system construction in HTK is that HMMs should be refined incrementally. Thus, a typical progression is to start with a simple set of single Gaussian context-independent phone models and then iteratively refine them by expanding them to include context-dependency and use multiple mixture component Gaussian distributions. The tool HHEd is a HMM definition editor which will clone models into context-dependent sets, apply a variety of parameter tyings and increment the number of mixture components in specified distributions. The usual process is to modify a set of HMMs in stages using HHEd and then re-estimate the parameters of the modified set using HERest after each stage. To improve performance for specific speakers the tools HEAdapt and HVite can be used to adapt HMMs to better model the characteristics of particular speakers using a small amount of training or adaptation data. The end result of which is a speaker adapted system.

The single biggest problem in building context-dependent HMM systems is always data insufficiency. The more complex the model set, the more data is needed to make robust estimates of its parameters, and since data is usually limited, a balance must be struck between complexity and the available data. For continuous density systems, this balance is achieved by tying parameters together as mentioned above. Parameter tying allows data to be pooled so that the shared parameters can be robustly estimated. In addition to continuous density systems, HTK also supports

fully tied mixture systems and discrete probability systems. In these cases, the data insufficiency problem is usually addressed by smoothing the distributions and the tool HSmooth is used for this.

2.3.3 Recognition Tools

HTK provides a single recognition tool called HVite which uses the token passing algorithm described in the previous chapter to perform Viterbi-based speech recognition. HVite takes as input a network describing the allowable word sequences, a dictionary defining how each word is pronounced and a set of HMMs. It operates by converting the word network to a phone network and then attaching the appropriate HMM definition to each phone instance. Recognition can then be performed on either a list of stored speech files or on direct audio input. As noted at the end of the last chapter, HVite can support cross-word triphones and it can run with multiple tokens to generate lattices containing multiple hypotheses. It can also be configured to rescore lattices and perform forced alignments.

The word networks needed to drive HVite are usually either simple word loops in which any word can follow any other word or they are directed graphs representing a finite-state task grammar. In the former case, bigram probabilities are normally attached to the word transitions. Word networks are stored using the HTK standard lattice format. This is a text-based format and hence word networks can be created directly using a text-editor. However, this is rather tedious and hence HTK provides two tools to assist in creating word networks. Firstly, HBuild allows sub-networks to be created and used within higher level networks. Hence, although the same low level notation is used, much duplication is avoided. Also, HBuild can be used to generate word loops and it can also read in a backed-off bigram language model and modify the word loop transitions to incorporate the bigram probabilities. Note that the label statistics tool HLStats mentioned earlier can be used to generate a backed-off bigram language model.

As an alternative to specifying a word network directly, a higher level grammar notation can be used. This notation is based on the Extended Backus Naur Form (EBNF) used in compiler specification and it is compatible with the grammar specification language used in earlier versions of HTK. The tool HParse is supplied to convert this notation into the equivalent word network.

Whichever method is chosen to generate a word network, it is useful to be able to see examples of the *language* that it defines. The tool HSGen is provided to do this. It takes as input a network and then randomly traverses the network outputting word strings. These strings can then be inspected to ensure that they correspond to what is required. HSGen can also compute the empirical perplexity of the task.

Finally, the construction of large dictionaries can involve merging several sources and performing a variety of transformations on each sources. The dictionary management tool HDMan is supplied to assist with this process.

2.3.4 Analysis Tool

Once the HMM-based recogniser has been built, it is necessary to evaluate its performance. This is usually done by using it to transcribe some pre-recorded test sentences and match the recogniser output with the correct reference transcriptions. This comparison is performed by a tool called HResults which uses dynamic programming to align the two transcriptions and then count substitution, deletion and insertion errors. Options are provided to ensure that the algorithms and output formats used by HResults are compatible with those used by the US National Institute of Standards and Technology (NIST). As well as global performance measures, HResults can also provide speaker-by-speaker breakdowns, confusion matrices and time-aligned transcriptions. For word spotting applications, it can also compute *Figure of Merit* (FOM) scores and *Receiver Operating Curve* (ROC) information.

2.4 Whats New In Version 2.2

This section lists the new features and refinements in HTK Version 2.2 compared to the preceding Version 2.1.

1. Speaker adaptation is now supported via the HEAdapt and HVite tools, which adapt a current set of models to a new speaker and/or environment.

- HEAdapt performs offline supervised adaptation using maximum likelihood linear regression (MLLR) and/or maximum a-posteriori (MAP) adaptation.
- HVite performs unsupervised adaptation using just MLLR.

Both tools can be used in a static mode, where all the data is presented prior to any adaptation, or in an incremental fashion.

2. Improved support for PC WAV files
   In addition to 16-bit PCM linear, HTK can now read
   - 8-bit CCITT mu-law
   - 8-bit CCITT a-law
   - 8-bit PCM linear

2.4.1 Features Added To Version 2.1

For the benefit of users of earlier versions of HTK this section lists the main changes in HTK Version 2.1 compared to the preceding Version 2.0.

1. The speech input handling has been partially re-designed and a new energy-based speech/silence detector has been incorporated into HParm. The detector is robust yet flexible and can be configured through a number of configuration variables. Speech/silence detection can now be performed on waveform files. The calibration of speech/silence detector parameters is now accomplished by asking the user to speak an arbitrary sentence.

2. HParm now allows random noise signal to be added to waveform data via the configuration parameter `ADDDITHER`. This prevents numerical overflows which can occur with artificially created waveform data under some coding schemes.

3. HNet has been optimised for more efficient operation when performing forced alignments of utterances using HVite. Further network optimisations tailored to biphone/triphone-based phone recognition have also been incorporated.

4. HVite can now produce partial recognition hypothesis even when no tokens survive to the end of the network. This is accomplished by setting the HRec configuration parameter `FORCEOUT` to true.

5. Dictionary support has been extended to allow pronunciation probabilities to be associated with different pronunciations of the same word. At the same time, HVite now allows the use of a pronunciation scale factor during recognition.

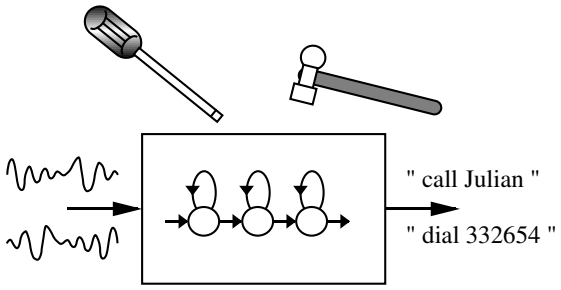
6. HTK now provides consistent support for reading and writing of HTK binary files (waveforms, binary MMFs, binary SLFs, HERest accumulators) across different machine architectures incorporating automatic byte swapping. By default, all binary data files handled by the tools are now written/read in big-endian (`NONVAX`) byte order. The default behavior can be changed via the configuration parameters `NATURALREADORDER` and `NATURALWRITEORDER`.

7. HWave supports the reading of waveforms in Microsoft WAVE file format.

8. HAudio allows key-press control of live audio input.

Chapter 3

A Tutorial Example of Using HTK



This final chapter of the tutorial part of the book will describe the construction of a recogniser for simple voice dialling applications. This recogniser will be designed to recognise continuously spoken digit strings and a limited set of names. It is sub-word based so that adding a new name to the vocabulary involves only modification to the pronouncing dictionary and task grammar. The HMMs will be continuous density mixture Gaussian tied-state triphones with clustering performed using phonetic decision trees. Although the voice dialling task itself is quite simple, the system design is general-purpose and would be useful for a range of applications.

The system will be built from scratch even to the extent of recording training and test data using the HTK tool HSLab. To make this tractable, the system will be speaker dependent[1], but the same design would be followed to build a speaker independent system. The only difference being that data would be required from a large number of speakers and there would be a consequential increase in model complexity.

Building a speech recogniser from scratch involves a number of inter-related subtasks and pedagogically it is not obvious what the best order is to present them. In the presentation here, the ordering is chronological so that in effect the text provides a recipe that could be followed to construct a similar system. The entire process is described in considerable detail in order give a clear view of the range of functions that HTK addresses and thereby to motivate the rest of the book.

The HTK software distribution also contains an example of constructing a recognition system for the 1000 word ARPA Naval Resource Management Task. This is contained in the directory `RMHTK` of the HTK distribution. Further demonstration of HTK's capabilities can be found in the directory `HTKDemo`. Some example scripts that may be of assistance during the tutorial are available in the `HTKTutorial` directory.

At each step of the tutorial presented in this chapter, the user is advised to thoroughly read the entire section before executing the commands, and also to consult the reference section for each HTK tool being introduced (chapter 14), so that all command line options and arguments are clearly understood.

[1] The final stage of the tutorial deals with adapting the speaker dependent models for new speakers

3.1 Data Preparation

The first stage of any recogniser development project is data preparation. Speech data is needed both for training and for testing. In the system to be built here, all of this speech will be recorded from scratch and to do this scripts are needed to prompt for each sentence. In the case of the test data, these prompt scripts will also provide the reference transcriptions against which the recogniser's performance can be measured and a convenient way to create them is to use the task grammar as a random generator. In the case of the training data, the prompt scripts will be used in conjunction with a pronunciation dictionary to provide the initial phone level transcriptions needed to start the HMM training process. Since the application requires that arbitrary names can be added to the recogniser, training data with good phonetic balance and coverage is needed. Here for convenience the prompt scripts needed for training are taken from the TIMIT acoustic-phonetic database.

It follows from the above that before the data can be recorded, a phone set must be defined, a dictionary must be constructed to cover both training and testing and a task grammar must be defined.

3.1.1 Step 1 - the Task Grammar

The goal of the system to be built here is to provide a voice-operated interface for phone dialling. Thus, the recogniser must handle digit strings and also personal name lists. Examples of typical inputs might be

> Dial three three two six five four
>
> Dial nine zero four one oh nine
>
> Phone Woodland
>
> Call Steve Young

HTK provides a grammar definition language for specifying simple task grammars such as this. It consists of a set of variable definitions followed by a regular expression describing the words to recognise. For the voice dialling application, a suitable grammar might be

```
$digit = ONE | TWO | THREE | FOUR | FIVE |
         SIX | SEVEN | EIGHT | NINE | OH | ZERO;
$name  = [ JOOP ] JANSEN |
         [ JULIAN ] ODELL |
         [ DAVE ] OLLASON |
         [ PHIL ] WOODLAND |
         [ STEVE ] YOUNG;
( SENT-START ( DIAL <$digit> | (PHONE|CALL) $name) SENT-END )
```

where the vertical bars denote alternatives, the square brackets denote optional items and the angle braces denote one or more repetitions. The complete grammar can be depicted as a network as shown in Fig. 3.1.

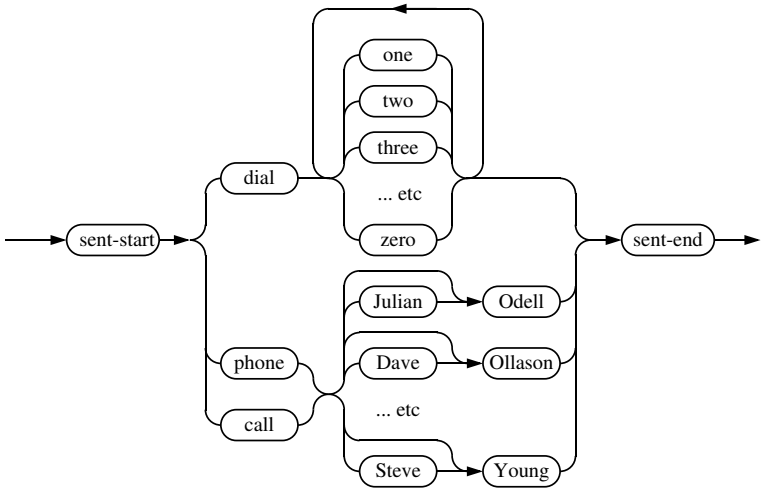


Fig. 3.1 Grammar for Voice Dialling

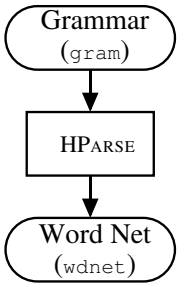


Fig. 3.2
Step 1

The above high level representation of a task grammar is provided for user convenience. The HTK recogniser actually requires a word network to be defined using a low level notation called HTK Standard Lattice Format (SLF) in which each word instance and each word-to-word transition is listed explicitly. This word network can be created automatically from the grammar above using the HPARSE tool, thus assuming that the file `gram` contains the above grammar, executing

```
HParse gram wdnet
```

will create an equivalent word network in the file `wdnet` (see Fig 3.2).

3.1.2 Step 2 - the Dictionary

The first step in building a dictionary is to create a sorted list of the required words. In the telephone dialling task pursued here, it is quite easy to create a list of required words by hand. However, if the task were more complex, it would be necessary to build a word list from the sample sentences present in the training data. Furthermore, to build robust acoustic models, it is necessary to train them on a large set of sentences containing many words and preferably phonetically balanced. For these reasons, the training data will consist of English sentences unrelated to the phone recognition task. Below, a short example of creating a word list from sentence prompts will be given. As noted above the training sentences given here are extracted from some prompts used with the TIMIT database and for convenience reasons they have been renumbered. For example, the first few items might be as follows

```
S0001 ONE VALIDATED ACTS OF SCHOOL DISTRICTS
S0002 TWO OTHER CASES ALSO WERE UNDER ADVISEMENT
S0003 BOTH FIGURES WOULD GO HIGHER IN LATER YEARS
S0004 THIS IS NOT A PROGRAM OF SOCIALIZED MEDICINE
etc
```

The desired training word list (`wlist`) could then be extracted automatically from these. Before using HTK, one would need to edit the text into a suitable format. For example, it would be necessary to change all white space to newlines and then to use the UNIX utilities `sort` and `uniq` to sort the words into a unique alphabetically ordered set, with one word per line. The script `prompts2wlist` from the `HTKTutorial` directory can be used for this purpose.

The dictionary itself can be built from a standard source using HDMan. For this example, the British English BEEP pronouncing dictionary will be used[2]. Its phone set will be adopted without modification except that the stress marks will be removed and a short-pause (`sp`) will be added to the end of every pronunciation. If the dictionary contains any silence markers then the `MP` command will merge the `sil` and `sp` phones into a single `sil`. These changes can be applied using HDMan and an edit script (stored in `global.ded`) containing the three commands

```
AS sp
RS cmu
MP sil sil sp
```

where `cmu` refers to a style of stress marking in which the lexical stress level is marked by a single digit appended to the phone name (e.g. `eh2` means the phone `eh` with level 2 stress).

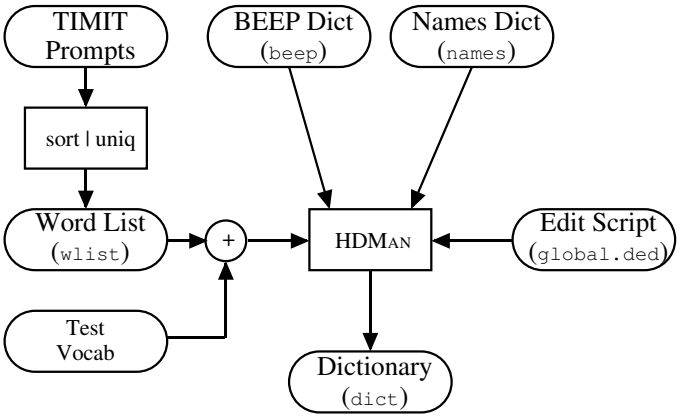


Fig. 3.3 Step 2

The command

```
HDMan -m -w wlist -n monophones1 -l dlog dict beep names
```

will create a new dictionary called `dict` by searching the source dictionaries `beep` and `names` to find pronunciations for each word in `wlist` (see Fig 3.3). Here, the `wlist` in question needs only to be a sorted list of the words appearing in the task grammar given above.

Note that `names` is a manually constructed file containing pronunciations for the proper names used in the task grammar. The option `-l` instructs HDMan to output a log file `dlog` which contains various statistics about the constructed dictionary. In particular, it indicates if there are words missing. HDMan can also output a list of the phones used, here called `monophones1`. Once training and test data has been recorded, an HMM will be estimated for each of these phones.

The general format of each dictionary entry is

```
WORD [outsym] p1 p2 p3 ....
```

[2]Available by anonymous ftp from `svr-ftp.eng.cam.ac.uk/pub/comp.speech/dictionaries/beep.tar.gz`. Note that items beginning with unmatched quotes, found at the start of the dictionary, should be removed.

which means that the word `WORD` is pronounced as the sequence of phones `p1 p2 p3 ...`. The string in square brackets specifies the string to output when that word is recognised. If it is omitted then the word itself is output. If it is included but empty, then nothing is output.

To see what the dictionary is like, here are a few entries.

```
A               ah sp
A               ax sp
A               ey sp
CALL            k ao l sp
DIAL            d ay ax l sp
EIGHT           ey t sp
PHONE           f ow n sp
SENT-END    []  sil
SENT-START  []  sil
SEVEN           s eh v n sp
TO              t ax sp
TO              t uw sp
ZERO            z ia r ow sp
```

Notice that function words such as `A` and `TO` have multiple pronunciations. The entries for `SENT-START` and `SENT-END` have a silence model `sil` as their pronunciations and null output symbols.

3.1.3 Step 3 - Recording the Data

The training and test data will be recorded using the HTK tool HSLab. This is a combined waveform recording and labelling tool. In this example HSLab will be used just for recording, as labels already exist. However, if you do not have pre-existing training sentences (such as those from the TIMIT database) you can create them either from pre-existing text (as described above) or by labelling your training utterances using HSLab. HSLab is invoked by typing

```
HSLab noname
```

This will cause a window to appear with a waveform display area in the upper half and a row of buttons, including a record button in the lower half. When the name of a normal file is given as argument, HSLab displays its contents. Here, the special file name `noname` indicates that new data is to be recorded. HSLab makes no special provision for prompting the user. However, each time the record button is pressed, it writes the subsequent recording alternately to a file called `noname_0.` and to a file called `noname_1.`. Thus, it is simple to write a shell script which for each successive line of a prompt file, outputs the prompt, waits for either `noname_0.` or `noname_1.` to appear, and then renames the file to the name prepending the prompt (see Fig. 3.4).

While the prompts for training sentences already were provided for above, the prompts for test sentences need to be generated before recording them. The tool HSGen can be used to do this by randomly traversing a word network and outputting each word encountered. For example, typing

```
HSGen -l -n 200 wdnet dict > testprompts
```

would generate 200 numbered test utterances, the first few of which would look something like:

```
1.  PHONE YOUNG
2.  DIAL OH SIX SEVEN SEVEN OH ZERO
3.  DIAL SEVEN NINE OH OH EIGHT SEVEN NINE NINE
4.  DIAL SIX NINE SIX TWO NINE FOUR ZERO NINE EIGHT
5.  CALL JULIAN ODELL
... etc
```

These can be piped to construct the prompt file `testprompts` for the required test data.

3.1.4 Step 4 - Creating the Transcription Files

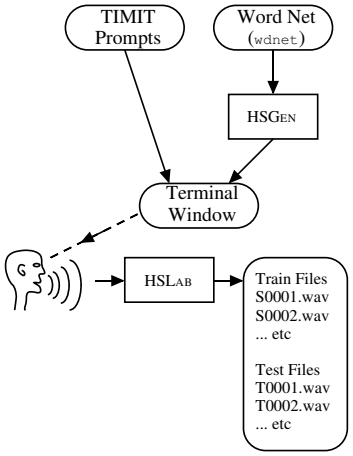


Fig. 3.4 Step 3

To train a set of HMMs, every file of training data must have an associated phone level transcription. Since there is no hand labelled data to bootstrap a set of models, a flat-start scheme will be used instead. To do this, two sets of phone transcriptions will be needed. The set used initially will have no short-pause (`sp`) models between words. Then once reasonable phone models have been generated, an `sp` model will be inserted between words to take care of any pauses introduced by the speaker.

The starting point for both sets of phone transcription is an orthographic transcription in HTK label format. This can be created fairly easily using a text editor or a scripting language. An example of this is found in the RM Demo at point 0.4. Alternatively, the script `prompts2mlf` has been provided in the `HTKTutorial` directory. The effect should be to convert the prompt utterances exampled above into the following form:

```
#!MLF!#
"*/S0001.lab"
ONE
VALIDATED
ACTS
OF
SCHOOL
DISTRICTS
.
"*/S0002.lab"
TWO
OTHER
CASES
ALSO
WERE
UNDER
ADVISEMENT
.
"*/S0003.lab"
BOTH
FIGURES
(etc.)
```

As can be seen, the prompt labels need to be converted into path names, each word should be written on a single line and each utterance should be terminated by a single period on its own.

The first line of the file just identifies the file as a *Master Label File* (MLF). This is a single file containing a complete set of transcriptions. HTK allows each individual transcription to be stored in its own file but it is more efficient to use an MLF.

The form of the path name used in the MLF deserves some explanation since it is really a *pattern* and not a name. When HTK processes speech files, it expects to find a transcription (or *label file*) with the same name but a different extension. Thus, if the file `/root/sjy/data/S0001.wav` was being processed, HTK would look for a label file called `/root/sjy/data/S0001.lab`. When MLF files are used, HTK scans the file for a pattern which matches the required label file name. However, an asterix will match any character string and hence the pattern used in the example is in effect path independent. It therefore allows the same transcriptions to be used with different versions of the speech data to be stored in different locations.

Once the word level MLF has been created, phone level MLFs can be generated using the label editor HLEd. For example, assuming that the above word level MLF is stored in the file `words.mlf`, the command

```
HLEd -l '*' -d dict -i phones0.mlf mkphones0.led words.mlf
```

will generate a phone level transcription of the following form where the `-l` option is needed to generate the path '`*`' in the output patterns.

```
#!MLF!#
"*/S0001.lab"
sil
w
ah
n
v
ae
l
ih
d
.. etc
```

This process is illustrated in Fig. 3.5.

The HLEd edit script `mkphones0.led` contains the following commands

```
EX
IS sil sil
DE sp
```

The expand `EX` command replaces each word in `words.mlf` by the corresponding pronunciation in the dictionary file `dict`. The `IS` command inserts a silence model `sil` at the start and end of every utterance. Finally, the delete `DE` command deletes all short-pause `sp` labels, which are not wanted in the transcription labels at this point.

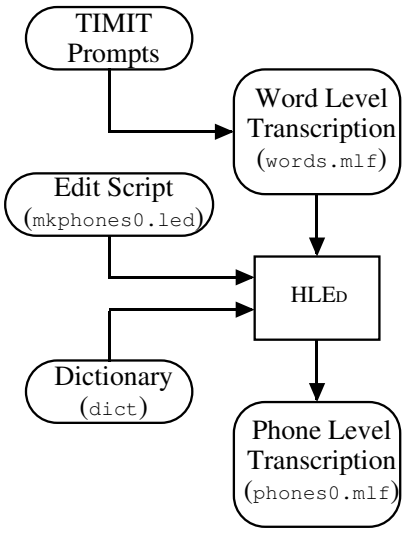


Fig. 3.5 Step 4

3.1.5 Step 5 - Coding the Data

The final stage of data preparation is to parameterise the raw speech waveforms into sequences of feature vectors. HTK support both FFT-based and LPC-based analysis. Here Mel Frequency Cepstral Coefficients (MFCCs), which are derived from FFT-based log spectra, will be used.

Coding can be performed using the tool HCOPY configured to automatically convert its input into MFCC vectors. To do this, a configuration file (`config`) is needed which specifies all of the conversion parameters. Reasonable settings for these are as follows

```
# Coding parameters
TARGETKIND = MFCC_0
TARGETRATE = 100000.0
SAVECOMPRESSED = T
SAVEWITHCRC = T
WINDOWSIZE = 250000.0
USEHAMMING = T
PREEMCOEF = 0.97
NUMCHANS = 26
CEPLIFTER = 22
NUMCEPS = 12
ENORMALISE = F
```

Some of these settings are in fact the default setting, but they are given explicitly here for completeness. In brief, they specify that the target parameters are to be MFCC using C_0 as the energy component, the frame period is 10msec (HTK uses units of 100ns), the output should be saved in compressed format, and a crc checksum should be added. The FFT should use a Hamming window and the signal should have first order preemphasis applied using a coefficient of 0.97. The filterbank should have 26 channels and 12 MFCC coefficients should be output. The variable `ENORMALISE` is by default true and performs energy normalisation on recorded audio files. It cannot be used with live audio and since the target system is for live audio, this variable should be set to false.

Note that explicitly creating coded data files is not necessary, as coding can be done "on-the-fly" from the original waveform files by specifying the appropriate configuration file (as above) with the relevant HTK tools. However, creating these files reduces the amount of preprocessing required during training, which itself can be a time-consuming process.

To run HCOPY, a list of each source file and its corresponding output file is needed. For example, the first few lines might look like

```
/root/sjy/waves/S0001.wav /root/sjy/train/S0001.mfc
/root/sjy/waves/S0002.wav /root/sjy/train/S0002.mfc
/root/sjy/waves/S0003.wav /root/sjy/train/S0003.mfc
/root/sjy/waves/S0004.wav /root/sjy/train/S0004.mfc
(etc.)
```

Files containing lists of files are referred to as script files[3] and by convention are given the extension `scp` (although HTK does not demand this). Script files are specified using the standard `-S` option and their contents are read simply as extensions to the command line. Thus, they avoid the need for command lines with several thousand arguments[4].

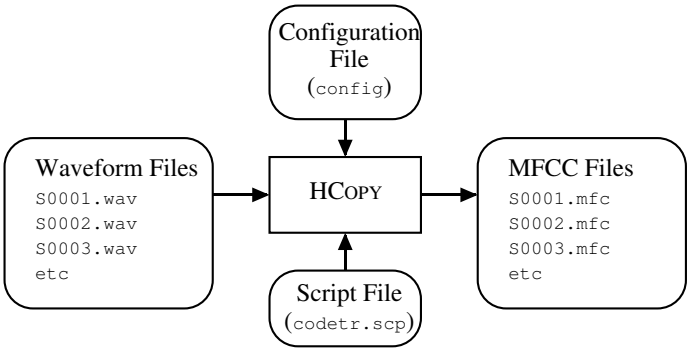


Fig. 3.6 Step 5

Assuming that the above script is stored in the file `codetr.scp`, the training data would be coded by executing

```
HCopy -T 1 -C config -S codetr.scp
```

This is illustrated in Fig. 3.6. A similar procedure is used to code the test data (using `TARGETKIND = MFCC_0_D_A` in config) after which all of the pieces are in place to start training the HMMs.

3.2 Creating Monophone HMMs

In this section, the creation of a well-trained set of single-Gaussian monophone HMMs will be described. The starting point will be a set of identical monophone HMMs in which every mean and variance is identical. These are then retrained, short-pause models are added and the silence model is extended slightly. The monophones are then retrained.

Some of the dictionary entries have multiple pronunciations. However, when HLEd was used to expand the word level MLF to create the phone level MLFs, it arbitrarily selected the first pronunciation it found. Once reasonable monophone HMMs have been created, the recogniser tool HVite can be used to perform a *forced alignment* of the training data. By this means, a new phone level MLF is created in which the choice of pronunciations depends on the acoustic evidence. This new MLF can be used to perform a final re-estimation of the monophone HMMs.

3.2.1 Step 6 - Creating Flat Start Monophones

The first step in HMM training is to define a prototype model. The parameters of this model are not important, its purpose is to define the model topology. For phone-based systems, a good topology to use is 3-state left-right with no skips such as the following

```
~o <VecSize> 39 <MFCC_0_D_A>
~h "proto"
```

[3] Not to be confused with files containing *edit* scripts

[4] Most UNIX shells, especially the C shell, only allow a limited and quite small number of arguments.

```
<BeginHMM>
 <NumStates> 5
 <State> 2
    <Mean> 39
      0.0 0.0 0.0 ...
    <Variance> 39
      1.0 1.0 1.0 ...
 <State> 3
    <Mean> 39
      0.0 0.0 0.0 ...
    <Variance> 39
      1.0 1.0 1.0 ...
 <State> 4
    <Mean> 39
      0.0 0.0 0.0 ...
    <Variance> 39
      1.0 1.0 1.0 ...
 <TransP> 5
  0.0 1.0 0.0 0.0 0.0
  0.0 0.6 0.4 0.0 0.0
  0.0 0.0 0.6 0.4 0.0
  0.0 0.0 0.0 0.7 0.3
  0.0 0.0 0.0 0.0 0.0
<EndHMM>
```

where each ellipsed vector is of length 39. This number, 39, is computed from the length of the parameterised static vector (`MFCC_0` = 13) plus the delta coefficients (+13) plus the acceleration coefficients (+13).

The HTK tool HCompV will scan a set of data files, compute the global mean and variance and set all of the Gaussians in a given HMM to have the same mean and variance. Hence, assuming that a list of all the training files is stored in `train.scp`, the command

```
HCompV -C config -f 0.01 -m -S train.scp -M hmm0 proto
```

will create a new version of `proto` in the directory `hmm0` in which the zero means and unit variances above have been replaced by the global speech means and variances. Note that the prototype HMM defines the parameter kind as `MFCC_0_D_A` (Note: 'zero' not 'oh'). This means that delta and acceleration coefficients are to be computed and appended to the static MFCC coefficients computed and stored during the coding process described above. To ensure that these are computed during loading, the configuration file `config` should be modified to change the target kind, i.e. the configuration file entry for `TARGETKIND` should be changed to

```
TARGETKIND = MFCC_0_D_A
```

HCompV has a number of options specified for it. The `-f` option causes a variance floor macro (called `vFloors`) to be generated which is equal to 0.01 times the global variance. This is a vector of values which will be used to set a floor on the variances estimated in the subsequent steps. The `-m` option asks for means to be computed as well as variances. Given this new prototype model stored in the directory `hmm0`, a *Master Macro File* (MMF) called `hmmdefs` containing a copy for each of the required monophone HMMs is constructed by manually copying the prototype and relabeling it for each required monophone (including "sil"). The format of an MMF is similar to that of an MLF and it serves a similar purpose in that it avoids having a large number of individual HMM definition files (see Fig. 3.7).

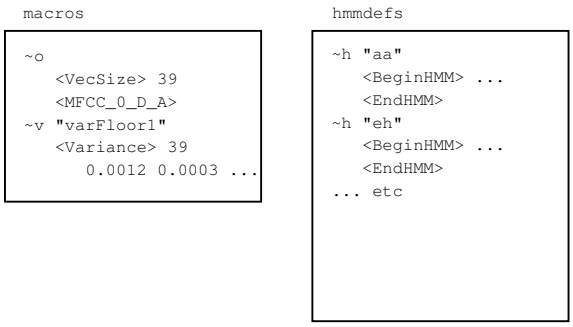


Fig. 3.7 Form of Master Macro Files

The flat start monophones stored in the directory `hmm0` are re-estimated using the embedded re-estimation tool HERest invoked as follows

```
HERest -C config -I phones0.mlf -t 250.0 150.0 1000.0 \
 -S train.scp -H hmm0/macros -H hmm0/hmmdefs -M hmm1 monophones0
```

The effect of this is to load all the models in `hmm0` which are listed in the model list `monophones0` (`monophones1` less the short pause (`sp`) model). These are then re-estimated them using the data listed in `train.scp` and the new model set is stored in the directory `hmm1`. Most of the files used in this invocation of HERest have already been described. The exception is the file `macros`. This should contain a so-called *global options* macro and the variance floor macro `vFloors` generated earlier. The global options macro simply defines the HMM parameter kind and the vector size i.e.

```
~o <MFCC_0_D_A> <VecSize> 39
```

See Fig. 3.7. This can be combined with `vFloors` into a text file called `macros`.

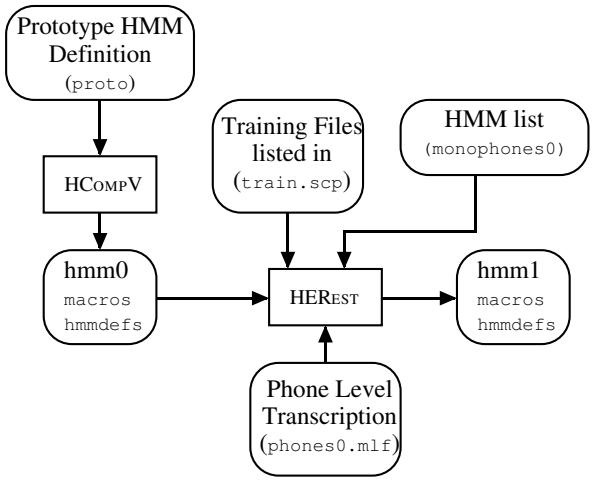


Fig. 3.8 Step 6

The `-t` option sets the pruning thresholds to be used during training. Pruning limits the range of state alignments that the forward-backward algorithm includes in its summation and it can reduce the amount of computation required by an order of magnitude. For most training files, a very tight pruning threshold can be set, however, some training files will provide poorer acoustic matching

and in consequence a wider pruning beam is needed. HERest deals with this by having an auto-incrementing pruning threshold. In the above example, pruning is normally 250.0. If re-estimation fails on any particular file, the threshold is increased by 150.0 and the file is reprocessed. This is repeated until either the file is successfully processed or the pruning limit of 1000.0 is exceeded. At this point it is safe to assume that there is a serious problem with the training file and hence the fault should be fixed (typically it will be an incorrect transcription) or the training file should be discarded. The process leading to the initial set of monophones in the directory `hmm0` is illustrated in Fig. 3.8.

Each time HERest is run it performs a single re-estimation. Each new HMM set is stored in a new directory. Execution of HERest should be repeated twice more, changing the name of the input and output directories (set with the options `-H` and `-M`) each time, until the directory `hmm3` contains the final set of initialised monophone HMMs.

3.2.2 Step 7 - Fixing the Silence Models

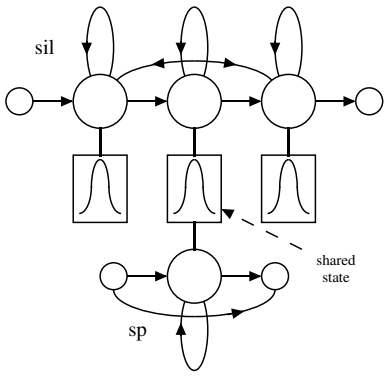


Fig. 3.9 Silence Models

The previous step has generated a 3 state left-to-right HMM for each phone and also a HMM for the silence model `sil`. The next step is to add extra transitions from states 2 to 4 and from states 4 to 2 in the silence model. The idea here is to make the model more robust by allowing individual states to absorb the various impulsive noises in the training data. The backward skip allows this to happen without committing the model to transit to the following word.

Also, at this point, a 1 state short pause `sp` model should be created. This should be a so-called *tee-model* which has a direct transition from entry to exit node. This `sp` has its emitting state tied to the centre state of the silence model. The required topology of the two silence models is shown in Fig. 3.9.

These silence models can be created in two stages

- Use a text editor on the file `hmm3/hmmdefs` to copy the centre state of the `sil` model to make a new `sp` model and store the resulting MMF `hmmdefs`, which includes the new `sp` model, in the new directory `hmm4`.
- Run the HMM editor HHEd to add the extra transitions required and tie the `sp` state to the centre `sil` state

HHEd works in a similar way to HLEd. It applies a set of commands in a script to modify a set of HMMs. In this case, it is executed as follows

```
HHEd -H hmm4/macros -H hmm4/hmmdefs -M hmm5 sil.hed monophones1
```

where `sil.hed` contains the following commands

```
AT 2 4 0.2 {sil.transP}
AT 4 2 0.2 {sil.transP}
AT 1 3 0.3 {sp.transP}
TI silst {sil.state[3],sp.state[2]}
```

The `AT` commands add transitions to the given transition matrices and the final `TI` command creates a tied-state called `silst`. The parameters of this tied-state are stored in the `hmmdefs` file and within each silence model, the original state parameters are replaced by the name of this macro. Macros are described in more detail below. For now it is sufficient to regard them simply as the mechanism by which HTK implements parameter sharing. Note that the phone list used here has been changed, because the original list `monophones0` has been extended by the new `sp` model. The new file is called `monophones1` and has been used in the above HHEd command.

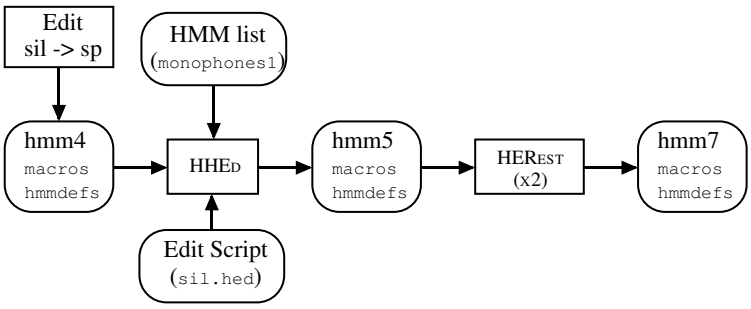


Fig. 3.10 Step 7

Finally, another two passes of HERest are applied using the phone transcriptions with `sp` models between words. This leaves the set of monophone HMMs created so far in the directory `hmm7`. This step is illustrated in Fig. 3.10

3.2.3 Step 8 - Realigning the Training Data

As noted earlier, the dictionary contains multiple pronunciations for some words, particularly function words. The phone models created so far can be used to *realign* the training data and create new transcriptions. This can be done with a single invocation of the HTK recognition tool HVite, viz

```
HVite -l '*' -o SWT -b silence -C config -a -H hmm7/macros \
      -H hmm7/hmmdefs -i aligned.mlf -m -t 250.0 -y lab \
      -I words.mlf -S train.scp  dict monophones1
```

This command uses the HMMs stored in `hmm7` to transform the input word level transcription `words.mlf` to the new phone level transcription `aligned.mlf` using the pronunciations stored in the dictionary `dict` (see Fig 3.11). The key difference between this operation and the original word-to-phone mapping performed by HLEd in step 4 is that the recogniser considers all pronunciations for each word and outputs the pronunciation that best matches the acoustic data.

In the above, the `-b` option is used to insert a silence model at the start and end of each utterance. The name `silence` is used on the assumption that the dictionary contains an entry

```
silence sil
```

Note that the dictionary should be sorted firstly by case (upper case first) and secondly alphabetically. The `-t` option sets a pruning level of 250.0 and the `-o` option is used to suppress the printing of scores, word names and time boundaries in the output MLF.

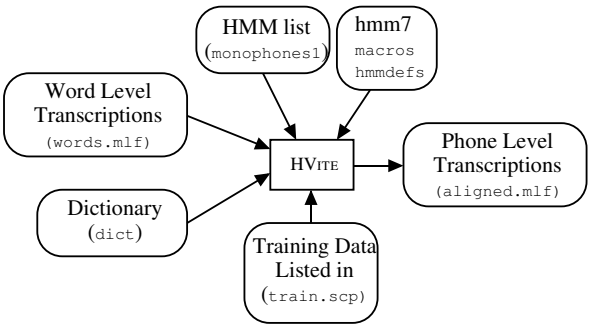


Fig. 3.11 Step 8

Once the new phone alignments have been created, another 2 passes of HERest can be applied to reestimate the HMM set parameters again. Assuming that this is done, the final monophone HMM set will be stored in directory `hmm9`.

3.3 Creating Tied-State Triphones

Given a set of monophone HMMs, the final stage of model building is to create context-dependent triphone HMMs. This is done in two steps. Firstly, the monophone transcriptions are converted to triphone transcriptions and a set of triphone models are created by copying the monophones and re-estimating. Secondly, similar acoustic states of these triphones are tied to ensure that all state distributions can be robustly estimated.

3.3.1 Step 9 - Making Triphones from Monophones

Context-dependent triphones can be made by simply cloning monophones and then re-estimating using triphone transcriptions. The latter should be created first using HLEd because a side-effect is to generate a list of all the triphones for which there is at least one example in the training data. That is, executing

```
HLEd -n triphones1 -l '*' -i wintri.mlf mktri.led aligned.mlf
```

will convert the monophone transcriptions in `aligned.mlf` to an equivalent set of triphone transcriptions in `wintri.mlf`. At the same time, a list of triphones is written to the file `triphones1`. The edit script `mktri.led` contains the commands

```
WB sp
WB sil
TC
```

The two `WB` commands define `sp` and `sil` as *word boundary symbols*. These then block the addition of context in the `TI` command, seen in the following script, which converts all phones (except word boundary symbols) to triphones . For example,

```
sil th ih s sp m ae n sp ...
```

becomes

```
sil th+ih th-ih+s ih-s sp m+ae m-ae+n ae-n sp ...
```

This style of triphone transcription is referred to as *word internal*. Note that some biphones will also be generated as contexts at word boundaries will sometimes only include two phones.

The cloning of models can be done efficiently using the HMM editor HHEd:

```
HHEd -B -H hmm9/macros -H hmm9/hmmdefs -M hmm10
     mktri.hed monophones1
```

where the edit script `mktri.hed` contains a clone command `CL` followed by `TI` commands to tie all of the transition matrices in each triphone set, that is:

```
CL triphones1
TI T_ah {(*-ah+*,ah+*,*-ah).transP}
TI T_ax {(*-ax+*,ax+*,*-ax).transP}
TI T_ey {(*-ey+*,ey+*,*-ey).transP}
TI T_b {(*-b+*,b+*,*-b).transP}
TI T_ay {(*-ay+*,ay+*,*-ay).transP}
...
```

The file `mktri.hed` can be generated using the *Perl* script `maketrihed` included in the `HTKTutorial` directory. When running the HHEd command you will get warnings about trying to tie transition matrices for the sil and sp models. Since neither model is context-dependent there aren't actually any matrices to tie.

The clone command `CL` takes as its argument the name of the file containing the list of triphones (and biphones) generated above. For each model of the form `a-b+c` in this list, it looks for the monophone `b` and makes a copy of it. Each `TI` command takes as its argument the name of a macro and a list of HMM components. The latter uses a notation which attempts to mimic the hierarchical structure of the HMM parameter set in which the transition matrix `transP` can be regarded as a sub-component of each HMM. The list of items within brackets are patterns designed to match the set of triphones, right biphones and left biphones for each phone.

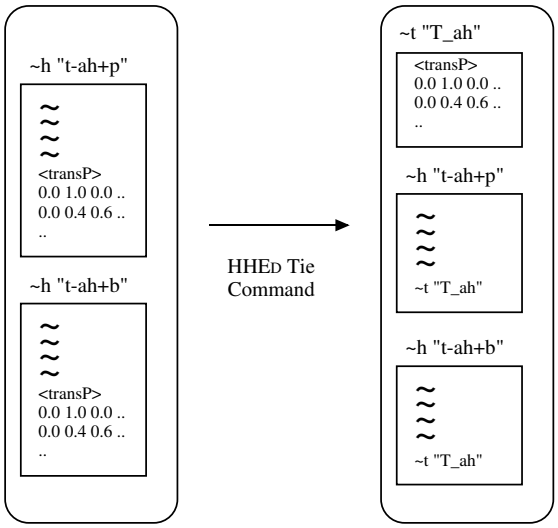


Fig. 3.12 Tying Transition Matrices

Up to now macros and tying have only been mentioned in passing. Although a full explanation must wait until chapter 7, a brief explanation is warranted here. Tying means that one or more HMMs share the same set of parameters. On the left side of Fig. 3.12, two HMM definitions are shown. Each HMM has its own individual transition matrix. On the right side, the effect of the first `TI` command in the edit script `mktri.hed` is shown. The individual transition matrices have been replaced by a reference to a *macro* called `T_ah` which contains a matrix shared by both models. When reestimating tied parameters, the data which would have been used for each of the original untied parameters is pooled so that a much more reliable estimate can be obtained.

Of course, tying could affect performance if performed indiscriminately. Hence, it is important to only tie parameters which have little effect on discrimination. This is the case here where the transition parameters do not vary significantly with acoustic context but nevertheless need to be estimated accurately. Some triphones will occur only once or twice and so very poor estimates would be obtained if tying was not done. These problems of data insufficiency will affect the output distributions too, but this will be dealt with in the next step.

Hitherto, all HMMs have been stored in text format and could be inspected like any text file. Now however, the model files will be getting larger and space and load/store times become an issue. For increased efficiency, HTK can store and load MMFs in binary format. Setting the standard `-B` option causes this to happen.

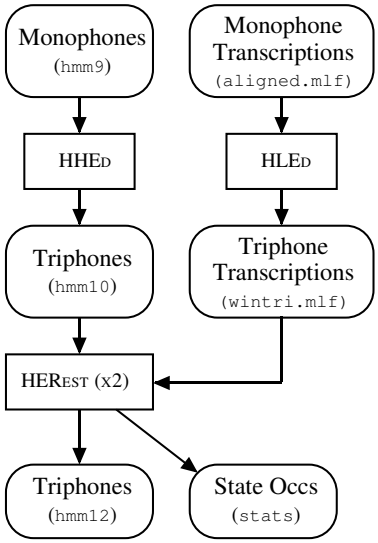


Fig. 3.13 Step 9

Once the context-dependent models have been cloned, the new triphone set can be re-estimated using HERest. This is done as previously except that the monophone model list is replaced by a triphone list and the triphone transcriptions are used in place of the monophone transcriptions.

For the final pass of HERest, the `-s` option should be used to generate a file of state occupation statistics called `stats`. In combination with the means and variances, these enable likelihoods to be calculated for clusters of states and are needed during the state-clustering process described below. Fig. 3.13 illustrates this step of the HMM construction procedure. Re-estimation should be again done twice, so that the resultant model sets will ultimately be saved in `hmm12`.

```
HERest -B -C config -I wintri.mlf -t 250.0 150.0 1000.0 -s stats \
 -S train.scp -H hmm11/macros -H hmm11/hmmdefs -M hmm12 triphones1
```

3.3.2 Step 10 - Making Tied-State Triphones

The outcome of the previous stage is a set of triphone HMMs with all triphones in a phone set sharing the same transition matrix. When estimating these models, many of the variances in the output distributions will have been floored since there will be insufficient data associated with many of the states. The last step in the model building process is to tie states within triphone sets in order to share data and thus be able to make robust parameter estimates.

In the previous step, the `TI` command was used to explicitly tie all members of a set of transition matrices together. However, the choice of which states to tie requires a bit more subtlety since the performance of the recogniser depends crucially on how accurate the state output distributions capture the statistics of the speech data.

HHEd provides two mechanisms which allow states to be clustered and then each cluster tied. The first is data-driven and uses a similarity measure between states. The second uses decision trees and is based on asking questions about the left and right contexts of each triphone. The decision tree attempts to find those contexts which make the largest difference to the acoustics and which should therefore distinguish clusters.

Decision tree state tying is performed by running HHEd in the normal way, i.e.

```
HHEd -B -H hmm12/macros -H hmm12/hmmdefs -M hmm13 \
     tree.hed triphones1 > log
```

Notice that the output is saved in a log file. This is important since some tuning of thresholds is usually needed.

The edit script `tree.hed`, which contains the instructions regarding which contexts to examine for possible clustering, can be rather long and complex. A script for automatically generating this file, `mkclscript`, is found in the RM Demo. A version of the `tree.hed` script, which can be used with this tutorial, is included in the `HTKTutorial` directory. Note that this script is only capable of creating the TB commands (decision tree clustering of states). The questions (QS) still need defining by the user. There is, however, an example list of questions which may be suitable to some tasks (or at least useful as an example) supplied with the RM demo (lib/quests.hed). The entire script appropriate for clustering English phone models is too long to show here in the text, however, its main components are given by the following fragments:

```
RO 100.0 stats
TR 0
QS "L_Class-Stop" {p-*,b-*,t-*,d-*,k-*,g-*}
QS "R_Class-Stop" {*+p,*+b,*+t,*+d,*+k,*+g}
QS "L_Nasal" {m-*,n-*,ng-*}
QS "R_Nasal" {*+m,*+n,*+ng}
QS "L_Glide" {y-*,w-*}
QS "R_Glide" {*+y,*+w}
....
QS "L_w" {w-*}
QS "R_w" {*+w}
QS "L_y" {y-*}
QS "R_y" {*+y}
QS "L_z" {z-*}
QS "R_z" {*+z}

TR 2

TB 350.0 "aa_s2" {(aa, *-aa, *-aa+*, aa+*).state[2]}
TB 350.0 "ae_s2" {(ae, *-ae, *-ae+*, ae+*).state[2]}
TB 350.0 "ah_s2" {(ah, *-ah, *-ah+*, ah+*).state[2]}
TB 350.0 "uh_s2" {(uh, *-uh, *-uh+*, uh+*).state[2]}
....
TB 350.0 "y_s4" {(y, *-y, *-y+*, y+*).state[4]}
TB 350.0 "z_s4" {(z, *-z, *-z+*, z+*).state[4]}
TB 350.0 "zh_s4" {(zh, *-zh, *-zh+*, zh+*).state[4]}

TR 1

AU "fulllist"
CO "tiedlist"

ST "trees"
```

Firstly, the `RO` command is used to set the outlier threshold to 100.0 and load the statistics file generated at the end of the previous step. The outlier threshold determines the minimum occupancy of any cluster and prevents a single outlier state forming a singleton cluster just because it is acoustically very different to all the other states. The `TR` command sets the trace level to zero in preparation for loading in the questions. Each `QS` command loads a single question and each question is defined by a set of contexts. For example, the first `QS` command defines a question called `L_Class-Stop` which is true if the left context is either of the stops `p`, `b`, `t`, `d`, `k` or `g`.

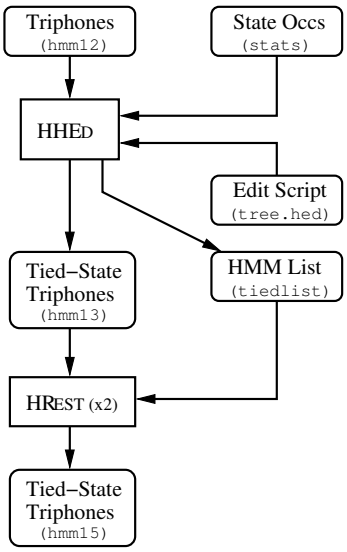


Fig. 3.14 Step 10

Notice that for a triphone system, it is necessary to include questions referring to both the right and left contexts of a phone. The questions should progress from wide, general classifications (such as consonant, vowel, nasal, diphthong, etc.) to specific instances of each phone. Ideally, the full set of questions loaded using the `QS` command would include every possible context which can influence the acoustic realisation of a phone, and can include any linguistic or phonetic classification which may be relevant. There is no harm in creating extra unnecessary questions, because those which are determined to be irrelevant to the data will be ignored.

The second `TR` command enables intermediate level progress reporting so that each of the following `TB` commands can be monitored. Each of these `TB` commands clusters one specific set of states. For example, the first `TB` command applies to the first emitting state of all context-dependent models for the phone `aa`.

Each `TB` command works as follows. Firstly, each set of states defined by the final argument is pooled to form a single cluster. Each question in the question set loaded by the `QS` commands is used to split the pool into two sets. The use of two sets rather than one, allows the log likelihood of the training data to be increased and the question which maximises this increase is selected for the first branch of the tree. The process is then repeated until the increase in log likelihood achievable by any question at any node is less than the threshold specified by the first argument (350.0 in this case).

Note that the values given in the `RO` and `TB` commands affect the degree of tying and therefore the number of states output in the clustered system. The values should be varied according to the amount of training data available. As a final step to the clustering, any pair of clusters which can be merged such that the decrease in log likelihood is below the threshold is merged. On completion, the states in each cluster i are tied to form a single shared state with macro name `xxx_i` where `xxx` is the name given by the second argument of the `TB` command.

The set of triphones used so far only includes those needed to cover the training data. The `AU` command takes as its argument a new list of triphones expanded to include all those needed for recognition. This list can be generated, for example, by using HDMAN on the entire dictionary (not just the training dictionary), converting it to triphones using the command `TC` and outputting a list of the distinct triphones to a file using the option `-n`

```
HDMan -b sp -n fulllist -g global.ded -l flog beep-tri beep
```

The -b sp option specifies that the sp phone is used as a word boundary, and so is excluded from triphones. The effect of the `AU` command is to use the decision trees to synthesise all of the new previously unseen triphones in the new list.

Once all state-tying has been completed and new models synthesised, some models may share exactly the same 3 states and transition matrices and are thus identical. The `CO` command is used to compact the model set by finding all identical models and tying them together[5], producing a new list of models called `tiedlist`.

One of the advantages of using decision tree clustering is that it allows previously unseen triphones to be synthesised. To do this, the trees must be saved and this is done by the `ST` command. Later if new previously unseen triphones are required, for example in the pronunciation of a new vocabulary item, the existing model set can be reloaded into HHEd, the trees reloaded using the `LT` command and then a new extended list of triphones created using the `AU` command.

After HHEd has completed, the effect of tying can be studied and the thresholds adjusted if necessary. The log file will include summary statistics which give the total number of physical states remaining and the number of models after compacting.

Finally, and for the last time, the models are re-estimated twice using HERest. Fig. 3.14 illustrates this last step in the HMM build process. The trained models are then contained in the file `hmm15/hmmdefs`.

3.4 Recogniser Evaluation

The recogniser is now complete and its performance can be evaluated. The recognition network and dictionary have already been constructed, and test data has been recorded. Thus, all that is necessary is to run the recogniser and then evaluate the results using the HTK analysis tool HResults

3.4.1 Step 11 - Recognising the Test Data

Assuming that `test.scp` holds a list of the coded test files, then each test file will be recognised and its transcription output to an MLF called `recout.mlf` by executing the following

```
HVite -H hmm15/macros -H hmm15/hmmdefs -S test.scp \
      -l '*' -i recout.mlf -w wdnet \
      -p 0.0 -s 5.0 dict tiedlist
```

The options `-p` and `-s` set the *word insertion penalty* and the *grammar scale factor*, respectively. The word insertion penalty is a fixed value added to each token when it transits from the end of one word to the start of the next. The grammar scale factor is the amount by which the language model probability is scaled before being added to each token as it transits from the end of one word to the start of the next. These parameters can have a significant effect on recognition performance and hence, some tuning on development test data is well worthwhile.

The dictionary contains monophone transcriptions whereas the supplied HMM list contains word internal triphones. HVite will make the necessary conversions when loading the word network `wdnet`. However, if the HMM list contained both monophones and context-dependent phones then HVite would become confused. The required form of word-internal network expansion can be forced by setting the configuration variable `FORCECXTEXP` to true and `ALLOWXWRDEXP` to false (see chapter 12 for details).

Assuming that the MLF `testref.mlf` contains word level transcriptions for each test file[6], the actual performance can be determined by running HResults as follows

```
HResults -I testref.mlf tiedlist recout.mlf
```

the result would be a print-out of the form

```
====================== HTK Results Analysis ==============
  Date: Sun Oct 22 16:14:45 1995
  Ref : testrefs.mlf
  Rec : recout.mlf
------------------------ Overall Results -----------------
SENT: %Correct=98.50 [H=197, S=3, N=200]
```

[5] Note that if the transition matrices had not been tied, the `CO` command would be ineffective since all models would be different by virtue of their unique transition matrices.

[6] The HLEd tool may have to be used to insert silences at the start and end of each transcription or alternatively HResults can be used to ignore silences (or any other symbols) using the `-e` option

```
WORD: %Corr=99.77, Acc=99.65 [H=853, D=1, S=1, I=1, N=855]
============================================================
```

The line starting with `SENT:` indicates that of the 200 test utterances, 197 (98.50%) were correctly recognised. The following line starting with `WORD:` gives the word level statistics and indicates that of the 855 words in total, 853 (99.77%) were recognised correctly. There was 1 deletion error (`D`), 1 substitution error (`S`) and 1 insertion error (`I`). The accuracy figure (`Acc`) of 99.65% is lower than the percentage correct (`Cor`) because it takes account of the insertion errors which the latter ignores.

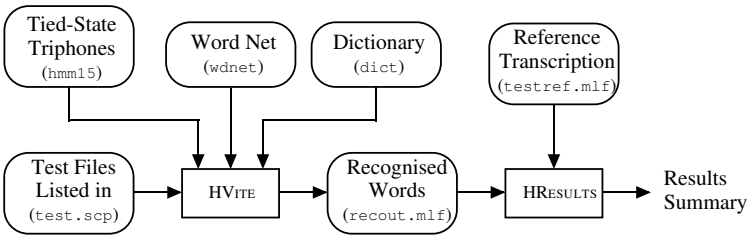


Fig. 3.15 Step 11

3.5 Running the Recogniser Live

The recogniser can also be run with live input. To do this it is only necessary to set the configuration variables needed to convert the input audio to the correct form of parameterisation. Specifically, the following needs to be appended to the configuration file `config` to create a new configuration file `config2`

```
# Waveform capture
SOURCERATE=625.0
SOURCEKIND=HAUDIO
SOURCEFORMAT=HTK
ENORMALISE=F
USESILDET=T
MEASURESIL=F
OUTSILWARN=T
```

These indicate that the source is direct audio with sample period 62.5 μsecs. The silence detector is enabled and a measurement of the background speech/silence levels should be made at start-up. The final line makes sure that a warning is printed when this silence measurement is being made.

Once the configuration file has been set-up for direct audio input, HVite can be run as in the previous step except that no files need be given as arguments

```
HVite -H hmm15/macros -H hmm15/hmmdefs -C config2 \
      -w wdnet -p 0.0 -s 5.0 dict tiedlist
```

On start-up, HVite will prompt the user to speak an arbitrary sentence (approx. 4 secs) in order to measure the speech and background silence levels. It will then repeatedly recognise and, if trace level bit 1 is set, it will output each utterance to the terminal. A typical session is as follows

```
Read 1648 physical / 4131 logical HMMs
Read lattice with 26 nodes / 52 arcs
Created network with 123 nodes / 151 links

READY[1]>
Please speak sentence - measuring levels
Level measurement completed
DIAL FOUR SIX FOUR TWO FOUR OH
```

```
      == [303 frames] -95.5773 [Ac=-28630.2 LM=-329.8] (Act=21.8)

READY[2]>
 DIAL ZERO EIGHT SIX TWO
      == [228 frames] -99.3758 [Ac=-22402.2 LM=-255.5] (Act=21.8)

READY[3]>
 etc
```

During loading, information will be printed out regarding the different recogniser components. The physical models are the distinct HMMs used by the system, while the logical models include all model names. The number of logical models is higher than the number of physical models because many logically distinct models have been determined to be physically identical and have been merged during the previous model building steps. The lattice information refers to the number of links and nodes in the recognition syntax. The network information refers to actual recognition network built by expanding the lattice using the current HMM set, dictionary and any context expansion rules specified. After each utterance, the numerical information gives the total number of frames, the average log likelihood per frame, the total acoustic score, the total language model score and the average number of models active.

Note that if it was required to recognise a new name, then the following two changes would be needed

1. the grammar would be altered to include the new name
2. a pronunciation for the new name would be added to the dictionary

If the new name required triphones which did not exist, then they could be created by loading the existing triphone set into HHEd, loading the decision trees using the `LT` command and then using the `AU` command to generate a new complete triphone set.

3.6 Adapting the HMMs

The previous sections have described the stages required to build a simple voice dialling system. To simplify this process, speaker dependent models were developed using training data from a single user. Consequently, recognition accuracy for any other users would be poor. To overcome this limitation, a set of speaker independent models could be constructed, but this would require large amounts of training data from a variety of speakers. An alternative is to adapt the current speaker dependent models to the characteristics of a new speaker using a small amount of training or adaptation data. In general, adaptation techniques are applied to well trained speaker independent model sets to enable them to better model the characteristics of particular speakers.

HTK supports both supervised adaptation, where the true transcription of the data is known and unsupervised adaptation where the transcription is hypothesised. In HTK supervised adaptation is performed offline by HEAdapt using maximum likelihood linear regression (MLLR) and/or maximum a-posteriori (MAP) techniques to estimate a series of transforms or a transformed model set, that reduces the mismatch between the current model set and the adaptation data. Unsupervised adaptation is provided by HVite (see section 13.6.2), using just MLLR.

The following sections describe offline supervised adaptation (using MLLR) with the use of HEAdapt.

3.6.1 Step 12 - Preparation of the Adaptation Data

As in normal recogniser development, the first stage in adaptation involves data preparation. Speech data from the new user is required for both adapting the models and testing the adapted system. The data can be obtained in a similar fashion to that taken to prepare the original test data. Initially, prompt lists for the adaptation and test data will be generated using HSGen. For example, typing

```
HSGen -l -n 20 wdnet dict > promptsAdapt
HSGen -l -n 20 wdnet dict > promptsTest
```

would produce two prompt files for the adaptation and test data. The amount of adaptation data required will normally be found empirically, but a performance improvement should be observable after just 30 seconds of speech. In this case, around 20 utterances should be sufficient. HSLAB can be used to record the associated speech.

Assuming that the script files `codeAdapt.scp` and `codeTest.scp` list the source and output files for the adaptation and test data respectively then both sets of speech can then be coded using the HCOPY commands given below.

```
HCopy -C config -S codeAdapt.scp
HCopy -C config -S codeTest.scp
```

The final stage of preparation involves generating context dependent phone transcriptions of the adaptation data and word level transcriptions of the test data for use in adapting the models and evaluating their performance. The transcriptions of the test data can be obtained using `prompts2mlf`. To minimize the problem of multiple pronunciations the phone level transcriptions of the adaptation data can be obtained by using HVITE to perform a *forced alignment* of the adaptation data. Assuming that word level transcriptions are listed in `adaptWords.mlf`, then the following command will place the phone transcriptions in `adaptPhones.mlf`.

```
HVite -l '*' -o SWT -b silence -C config -a -H hmm15/macros \
      -H hmm15/hmmdefs -i adaptPhones.mlf -m -t 250.0 \
      -I adaptWords.mlf -y lab -S adapt.scp dict tiedlist
```

3.6.2 Step 13 - Generating the Transforms

HEADAPT provides two forms of MLLR adaptation depending on the amount of adaptation data available. If only small amounts are available a global transform can be generated for every output distribution of every model. As more adaptation data becomes available more specific transforms can be generated for specific groups of Gaussians. To identify the number of transforms that can be estimated using the current adaptation data, HEADAPT uses a regression class tree to cluster together groups of output distributions that are to undergo the same transformation. The HTK tool HHED can be used to build a regression class tree and store it as part of the HMM set. For example,

```
HHEd -B -H hmm15/macros -H hmm15/hmmdefs -M hmm16 regtree.hed tiedlist
```

creates a regression class tree using the models stored in `hmm15`. The models are written out to the `hmm16` directory together with the regression class tree information. The HHED edit script `regtree.hed` contains the following commands

```
RN "models"
LS "stats"
RC 32 "rtree"
```

The `RN` command assigns an identifier to the HMM set. The `LS` command loads the state occupation statistics file `stats` generated by the last application of HEREST which created the models in `hmm15`. The `RC` command then attempts to build a regression class tree with 32 terminal or leaf nodes using these statistics.

HEADAPT can be used to perform either static adaptation, where all the adaptation data is processed in a single block or incremental adaptation, where adaptation is performed after a specified number of utterances and this is controlled by the `-i` option. In this tutorial the default setting of static adaptation will be used.

A typical use of HEADAPT involves two passes. On the first pass a global adaptation is performed. The second pass then uses the global transformation to transform the model set, producing better frame/state alignments which are then used to estimate a set of more specific transforms, using a regression class tree. After estimating the transforms, HEADAPT can output either the newly adapted model set or the transformations themselves in a transform model file (TMF). The latter can be advantageous if storage is an issue since the TMFs are significantly smaller than MMFs and the computational overhead incurred when transforming a model set using a TMF is negligible.

The two applications of HEADAPT below demonstrate a static two-pass adaptation approach where the global and regression class transformations are stored in the `global.tmf` and `rc.tmf` files respectively. The standard `-J` and `-K` options are used to load and save the TMFs respectively.

```
HEAdapt -C config -g -S adapt.scp -I adaptPhones.mlf -H hmm16/macros \
        -H hmm16/hmmdefs -K global.tmf tiedlist

HEAdapt -C config -S adapt.scp -I adaptPhones.mlf -H hmm16/macros \
        -H hmm16/hmmdefs -J global.tmf -K rc.tmf tiedlist
```

3.6.3 Step 14 - Evaluation of the Adapted System

To evaluate the performance of the adaptation, the test data previously recorded is recognised using HVite. Assuming that `testAdapt.scp` contains a list of all of the coded test files, then HVite can be invoked in much the same way as before but with the additional `-J` argument used to load the model transformation file `rc.tmf`.

```
HVite -H hmm16/macros -H hmm16/hmmdefs -S testAdapt.scp -l '*' \
      -J rc.tmf -i recoutAdapt.mlf -w wdnet \
      -p 0.0 -s 5.0 dict tiedlist
```

The results of the adapted model set can then be observed using HResults in the usual manner.

The RM Demo contains a section on speaker adaptation (point 5.6) and the recognition results obtained using an adapted model set are given below.

```
====================== HTK Results Analysis =======================
  Date: Wed Jan 06 21:09:23 1999
  Ref : usr/local/htk/RMHTK_V2.1/RMLib/wlabs/dms0_tst.mlf
  Rec : adapt/dms0_tst.mlf
------------------------ Overall Results --------------------------
SENT: %Correct=66.33 [H=65, S=33, N=98]
WORD: %Corr=94.25, Acc=93.10 [H=738, D=11, S=34, I=9, N=783]
===================================================================
```

The performance improvement gained by the adapted models can be evaluated by recognising the test data using the unadapted model set and comparing the two results. For the RM Demo task the following results were obtained with an unadapted model set.

```
====================== HTK Results Analysis =======================
  Date: Mon Dec 14 10:59:28 1998
  Ref : usr/local/htk/RMHTK_V2.1/RMLib/wlabs/dms0_tst.mlf
  Rec : unadapt/dms0_tst.mlf
------------------------ Overall Results --------------------------
SENT: %Correct=46.00 [H=46, S=54, N=100]
WORD: %Corr=89.04, Acc=86.43 [H=715, D=26, S=62, I=21, N=803]
===================================================================
```

3.7 Summary

This chapter has described the construction of a tied-state phone-based continuous speech recogniser and in so doing, it has touched on most of the main areas addressed by HTK: recording, data preparation, HMM definitions, training tools, adaptation tools, networks, decoding and evaluating. The rest of this book discusses each of these topics in detail.