The HTK book

Book · January 2002

CITATIONS
1,104

READS
5,796

12 authors, including:

Steve Young
University of Cambridge
323 PUBLICATIONS **20,736** CITATIONS
SEE PROFILE

M.J.F. Gales
University of Cambridge
406 PUBLICATIONS **13,821** CITATIONS
SEE PROFILE

Thomas Hain
The University of Sheffield
205 PUBLICATIONS **5,204** CITATIONS
SEE PROFILE

Xunying Liu
The Chinese University of Hong Kong
151 PUBLICATIONS **3,084** CITATIONS
SEE PROFILE

The HTK Book

Steve Young
Gunnar Evermann
Thomas Hain
Dan Kershaw
Gareth Moore
Julian Odell
Dave Ollason
Dan Povey
Valtcho Valtchev
Phil Woodland

The HTK Book (for HTK Version 3.2.1)

First published December 1995

Reprinted March 1996

Revised for HTK Version 2.1 March 1997

Revised for HTK Version 2.2 January 1999

Revised for HTK Version 3.0 July 2000

Revised for HTK Version 3.1 December 2001

Revised for HTK Version 3.2 December 2002

Contents

I Tutorial Overview 1

1 The Fundamentals of HTK 2
- 1.1 General Principles of HMMs . . . 3
- 1.2 Isolated Word Recognition . . . 3
- 1.3 Output Probability Specification . . . 6
- 1.4 Baum-Welch Re-Estimation . . . 6
- 1.5 Recognition and Viterbi Decoding . . . 9
- 1.6 Continuous Speech Recognition . . . 10
- 1.7 Speaker Adaptation . . . 13

2 An Overview of the HTK Toolkit 14
- 2.1 HTK Software Architecture . . . 14
- 2.2 Generic Properties of a HTK Tool . . . 15
- 2.3 The Toolkit . . . 16
  - 2.3.1 Data Preparation Tools . . . 16
  - 2.3.2 Training Tools . . . 16
  - 2.3.3 Recognition Tools . . . 19
  - 2.3.4 Analysis Tool . . . 19
- 2.4 Whats New In Version 3.2 . . . 19
  - 2.4.1 New In Version 3.1 . . . 20
  - 2.4.2 New In Version 2.2 . . . 20
  - 2.4.3 Features Added To Version 2.1 . . . 21

3 A Tutorial Example of Using HTK 22
- 3.1 Data Preparation . . . 23
  - 3.1.1 Step 1 - the Task Grammar . . . 23
  - 3.1.2 Step 2 - the Dictionary . . . 24
  - 3.1.3 Step 3 - Recording the Data . . . 26
  - 3.1.4 Step 4 - Creating the Transcription Files . . . 27
  - 3.1.5 Step 5 - Coding the Data . . . 29
- 3.2 Creating Monophone HMMs . . . 30
  - 3.2.1 Step 6 - Creating Flat Start Monophones . . . 30
  - 3.2.2 Step 7 - Fixing the Silence Models . . . 33
  - 3.2.3 Step 8 - Realigning the Training Data . . . 34
- 3.3 Creating Tied-State Triphones . . . 35
  - 3.3.1 Step 9 - Making Triphones from Monophones . . . 35
  - 3.3.2 Step 10 - Making Tied-State Triphones . . . 37
- 3.4 Recogniser Evaluation . . . 40
  - 3.4.1 Step 11 - Recognising the Test Data . . . 40
- 3.5 Running the Recogniser Live . . . 41
- 3.6 Adapting the HMMs . . . 42
  - 3.6.1 Step 12 - Preparation of the Adaptation Data . . . 42
  - 3.6.2 Step 13 - Generating the Transforms . . . 43
  - 3.6.3 Step 14 - Evaluation of the Adapted System . . . 44
- 3.7 Summary . . . 44

II HTK in Depth **45**

4 The Operating Environment **46**
4.1 The Command Line . . . 47
4.2 Script Files . . . 47
4.3 Configuration Files . . . 48
4.4 Standard Options . . . 50
4.5 Error Reporting . . . 50
4.6 Strings and Names . . . 51
4.7 Memory Management . . . 52
4.8 Input/Output via Pipes and Networks . . . 53
4.9 Byte-swapping of HTK data files . . . 53
4.10 Summary . . . 53

5 Speech Input/Output **56**
5.1 General Mechanism . . . 56
5.2 Speech Signal Processing . . . 58
5.3 Linear Prediction Analysis . . . 60
5.4 Filterbank Analysis . . . 61
5.5 Vocal Tract Length Normalisation . . . 62
5.6 Cepstral Features . . . 63
5.7 Perceptual Linear Prediction . . . 64
5.8 Energy Measures . . . 65
5.9 Delta, Acceleration and Third Differential Coefficients . . . 65
5.10 Storage of Parameter Files . . . 66
5.10.1 HTK Format Parameter Files . . . 66
5.10.2 Esignal Format Parameter Files . . . 68
5.11 Waveform File Formats . . . 68
5.11.1 HTK File Format . . . 69
5.11.2 Esignal File Format . . . 69
5.11.3 TIMIT File Format . . . 69
5.11.4 NIST File Format . . . 69
5.11.5 SCRIBE File Format . . . 70
5.11.6 SDES1 File Format . . . 70
5.11.7 AIFF File Format . . . 70
5.11.8 SUNAU8 File Format . . . 70
5.11.9 OGI File Format . . . 70
5.11.10 WAV File Format . . . 71
5.11.11 ALIEN and NOHEAD File Formats . . . 71
5.12 Direct Audio Input/Output . . . 71
5.13 Multiple Input Streams . . . 73
5.14 Vector Quantisation . . . 74
5.15 Viewing Speech with HLIST . . . 76
5.16 Copying and Coding using HCOPY . . . 78
5.17 Version 1.5 Compatibility . . . 79
5.18 Summary . . . 80

6 Transcriptions and Label Files **83**
6.1 Label File Structure . . . 83
6.2 Label File Formats . . . 84
6.2.1 HTK Label Files . . . 84
6.2.2 ESPS Label Files . . . 85
6.2.3 TIMIT Label Files . . . 85
6.2.4 SCRIBE Label Files . . . 85
6.3 Master Label Files . . . 86
6.3.1 General Principles of MLFs . . . 86
6.3.2 Syntax and Semantics . . . 87
6.3.3 MLF Search . . . 87
6.3.4 MLF Examples . . . 88

6.4 Editing Label Files . . . 90
6.5 Summary . . . 93

7 HMM Definition Files **94**
7.1 The HMM Parameters . . . 95
7.2 Basic HMM Definitions . . . 96
7.3 Macro Definitions . . . 99
7.4 HMM Sets . . . 103
7.5 Tied-Mixture Systems . . . 107
7.6 Discrete Probability HMMs . . . 107
7.7 Input Linear Transforms . . . 109
7.8 Tee Models . . . 109
7.9 Regression Class Trees for Adaptation . . . 110
7.10 Binary Storage Format . . . 110
7.11 The HMM Definition Language . . . 111

8 HMM Parameter Estimation **116**
8.1 Training Strategies . . . 116
8.2 Initialisation using HInit . . . 119
8.3 Flat Starting with HCompV . . . 122
8.4 Isolated Unit Re-Estimation using HRest . . . 124
8.5 Embedded Training using HERest . . . 125
8.6 Single-Pass Retraining . . . 128
8.7 Two-model Re-Estimation . . . 128
8.8 Parameter Re-Estimation Formulae . . . 129
8.8.1 Viterbi Training (HInit) . . . 129
8.8.2 Forward/Backward Probabilities . . . 130
8.8.3 Single Model Reestimation(HRest) . . . 132
8.8.4 Embedded Model Reestimation(HERest) . . . 132

9 HMM Adaptation **134**
9.1 Model Adaptation using MLLR . . . 135
9.1.1 Maximum Likelihood Linear Regression . . . 135
9.1.2 MLLR and Regression Classes . . . 135
9.1.3 Transform Model File Format . . . 137
9.2 Model Adaptation using MAP . . . 138
9.3 Using HEAdapt . . . 140
9.4 MLLR Formulae . . . 141
9.4.1 Estimation of the Mean Transformation Matrix . . . 142
9.4.2 Estimation of the Variance Transformation Matrix . . . 143

10 HMM System Refinement **144**
10.1 Using HHEd . . . 145
10.2 Constructing Context-Dependent Models . . . 145
10.3 Parameter Tying and Item Lists . . . 146
10.4 Data-Driven Clustering . . . 148
10.5 Tree-Based Clustering . . . 150
10.6 Mixture Incrementing . . . 152
10.7 Regression Class Tree Construction . . . 153
10.8 Miscellaneous Operations . . . 154

11 Discrete and Tied-Mixture Models **155**
11.1 Modelling Discrete Sequences . . . 155
11.2 Using Discrete Models with Speech . . . 156
11.3 Tied Mixture Systems . . . 158
11.4 Parameter Smoothing . . . 160

12 Networks, Dictionaries and Language Models **161**
12.1 How Networks are Used . . . 162
12.2 Word Networks and Standard Lattice Format . . . 163
12.3 Building a Word Network with HPARSE . . . 165
12.4 Bigram Language Models . . . 167
12.5 Building a Word Network with HBUILD . . . 169
12.6 Testing a Word Network using HSGEN . . . 170
12.7 Constructing a Dictionary . . . 171
12.8 Word Network Expansion . . . 173
12.9 Other Kinds of Recognition System . . . 176

13 Decoding **178**
13.1 Decoder Operation . . . 178
13.2 Decoder Organisation . . . 180
13.3 Recognition using Test Databases . . . 182
13.4 Evaluating Recognition Results . . . 183
13.5 Generating Forced Alignments . . . 186
13.6 Decoding and Adaptation . . . 187
13.6.1 Recognition with Adapted HMMs . . . 187
13.6.2 Unsupervised Adaptation . . . 187
13.7 Recognition using Direct Audio Input . . . 188
13.8 N-Best Lists and Lattices . . . 189

III Language Modelling **191**

14 Fundamentals of language modelling **192**
14.1 *n*-gram language models . . . 193
14.1.1 Word *n*-gram models . . . 193
14.1.2 Equivalence classes . . . 194
14.1.3 Class *n*-gram models . . . 194
14.2 Statistically-derived Class Maps . . . 195
14.2.1 Word exchange algorithm . . . 195
14.3 Robust model estimation . . . 197
14.3.1 Estimating probabilities . . . 197
14.3.2 Smoothing probabilities . . . 198
14.4 Perplexity . . . 199
14.5 Overview of *n*-Gram Construction Process . . . 200
14.6 Class-Based Language Models . . . 202

15 A Tutorial Example of Building Language Models **203**
15.1 Database preparation . . . 203
15.2 Mapping OOV words . . . 206
15.3 Language model generation . . . 207
15.4 Testing the LM perplexity . . . 208
15.5 Generating and using count-based models . . . 209
15.6 Model interpolation . . . 210
15.7 Class-based models . . . 211
15.8 Problem solving . . . 214
15.8.1 File format problems . . . 214
15.8.2 Command syntax problems . . . 214
15.8.3 Word maps . . . 215
15.8.4 Memory problems . . . 215
15.8.5 Unexpected perplexities . . . 215

16 Language Modelling Reference **216**
16.1 Words and Classes . . . 217
16.2 Data File Headers . . . 217
16.3 Word Map Files . . . 217
16.4 Class Map Files . . . 218
16.5 Gram Files . . . 220
16.6 Frequency-of-frequency (FoF) Files . . . 221
16.7 Word LM file formats . . . 221
16.7.1 The ARPA-MIT LM format . . . 222
16.7.2 The modified ARPA-MIT format . . . 223
16.7.3 The binary LM format . . . 223
16.8 Class LM file formats . . . 223
16.8.1 Class counts format . . . 224
16.8.2 The class probabilities format . . . 224
16.8.3 The class LM three file format . . . 224
16.8.4 The class LM single file format . . . 225
16.9 Language modelling tracing . . . 225
16.9.1 LCMap . . . 226
16.9.2 LGBase . . . 226
16.9.3 LModel . . . 226
16.9.4 LPCalc . . . 226
16.9.5 LPMerge . . . 226
16.9.6 LUtil . . . 226
16.9.7 LWMap . . . 226
16.10 Run-time configuration parameters . . . 227
16.10.1 USEINTID . . . 227
16.11 Compile-time configuration parameters . . . 228
16.11.1 LM_ID_SHORT . . . 228
16.11.2 LM_COMPACT . . . 228
16.11.3 LMPROB_SHORT . . . 228
16.11.4 INTERPOLATE_MAX . . . 228
16.11.5 SANITY . . . 228
16.11.6 INTEGRITY_CHECK . . . 228

IV Reference Section **230**

17 The HTK Tools **231**
17.1 Cluster . . . 232
17.1.1 Function . . . 232
17.1.2 Use . . . 233
17.1.3 Tracing . . . 234
17.2 HBuild . . . 235
17.2.1 Function . . . 235
17.2.2 Use . . . 235
17.2.3 Tracing . . . 236
17.3 HCompV . . . 237
17.3.1 Function . . . 237
17.3.2 Use . . . 237
17.3.3 Tracing . . . 238
17.4 HCopy . . . 239
17.4.1 Function . . . 239
17.4.2 Use . . . 239
17.4.3 Trace Output . . . 241
17.5 HDMan . . . 242
17.5.1 Function . . . 242
17.5.2 Use . . . 243
17.5.3 Tracing . . . 244
17.6 HEAdapt . . . 245

17.6.1 Function . . . 245
17.6.2 Use . . . 245
17.6.3 Tracing . . . 246
17.7 HERest . . . 248
17.7.1 Function . . . 248
17.7.2 Use . . . 249
17.7.3 Tracing . . . 250
17.8 HHEd . . . 251
17.8.1 Function . . . 251
17.8.2 Use . . . 258
17.8.3 Tracing . . . 259
17.9 HInit . . . 260
17.9.1 Function . . . 260
17.9.2 Use . . . 260
17.9.3 Tracing . . . 261
17.10 HLEd . . . 262
17.10.1 Function . . . 262
17.10.2 Use . . . 263
17.10.3 Tracing . . . 264
17.11 HList . . . 265
17.11.1 Function . . . 265
17.11.2 Use . . . 265
17.11.3 Tracing . . . 265
17.12 HLMCopy . . . 266
17.12.1 Function . . . 266
17.12.2 Use . . . 266
17.12.3 Tracing . . . 266
17.13 HLRescore . . . 267
17.13.1 Function . . . 267
17.13.2 Use . . . 267
17.13.3 Tracing . . . 268
17.14 HLStats . . . 269
17.14.1 Function . . . 269
17.14.2 Bigram Generation . . . 269
17.14.3 Use . . . 270
17.14.4 Tracing . . . 270
17.15 HParse . . . 271
17.15.1 Function . . . 271
17.15.2 Network Definition . . . 271
17.15.3 Compatibility Mode . . . 273
17.15.4 Use . . . 273
17.15.5 Tracing . . . 274
17.16 HQuant . . . 275
17.16.1 Function . . . 275
17.16.2 VQ Codebook Format . . . 275
17.16.3 Use . . . 275
17.16.4 Tracing . . . 276
17.17 HRest . . . 277
17.17.1 Function . . . 277
17.17.2 Use . . . 277
17.17.3 Tracing . . . 278
17.18 HResults . . . 279
17.18.1 Function . . . 279
17.18.2 Use . . . 280
17.18.3 Tracing . . . 282
17.19 HSGen . . . 283
17.19.1 Function . . . 283
17.19.2 Use . . . 283
17.19.3 Tracing . . . 283

17.20HSLab . . . 284
17.20.1 Function . . . 284
17.20.2 Use . . . 285
17.20.3 Tracing . . . 287
17.21HSmooth . . . 288
17.21.1 Function . . . 288
17.21.2 Use . . . 288
17.21.3 Tracing . . . 289
17.22HVite . . . 290
17.22.1 Function . . . 290
17.22.2 Use . . . 290
17.22.3 Tracing . . . 292
17.23LAdapt . . . 293
17.23.1 Function . . . 293
17.23.2 Use . . . 293
17.23.3 Tracing . . . 294
17.24LBuild . . . 295
17.24.1 Function . . . 295
17.24.2 Use . . . 295
17.24.3 Tracing . . . 295
17.25LFoF . . . 296
17.25.1 Function . . . 296
17.25.2 Use . . . 296
17.25.3 Tracing . . . 296
17.26LGCopy . . . 297
17.26.1 Function . . . 297
17.26.2 Use . . . 297
17.26.3 Tracing . . . 298
17.27LGList . . . 299
17.27.1 Function . . . 299
17.27.2 Use . . . 299
17.27.3 Tracing . . . 299
17.28LGPrep . . . 300
17.28.1 Function . . . 300
17.28.2 Use . . . 301
17.28.3 Tracing . . . 302
17.29LLink . . . 303
17.29.1 Function . . . 303
17.29.2 Use . . . 303
17.29.3 Tracing . . . 303
17.30LMerge . . . 304
17.30.1 Function . . . 304
17.30.2 Use . . . 304
17.30.3 Tracing . . . 304
17.31LNewMap . . . 305
17.31.1 Function . . . 305
17.31.2 Use . . . 305
17.31.3 Tracing . . . 305
17.32LNorm . . . 306
17.32.1 Function . . . 306
17.32.2 Use . . . 306
17.32.3 Tracing . . . 306
17.33LPlex . . . 307
17.33.1 Function . . . 307
17.33.2 Use . . . 307
17.33.3 Tracing . . . 308
17.34LSubset . . . 309
17.34.1 Function . . . 309
17.34.2 Use . . . 309

17.34.3 Tracing . . . 309

18 Configuration Variables **310**
18.1 Configuration Variables used in Library Modules . . . 310
18.2 Configuration Variables used in Tools . . . 314

19 Error and Warning Codes **315**
19.1 Generic Errors . . . 315
19.2 Summary of Errors by Tool and Module . . . 317

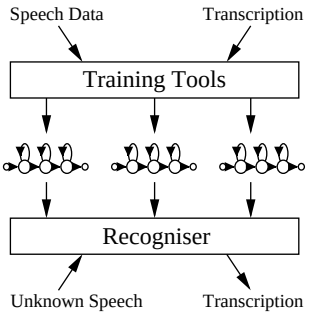
20 HTK Standard Lattice Format (SLF) **334**
20.1 SLF Files . . . 334
20.2 Format . . . 334
20.3 Syntax . . . 335
20.4 Field Types . . . 336
20.5 Example SLF file . . . 336

Part I

Tutorial Overview

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 above, 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

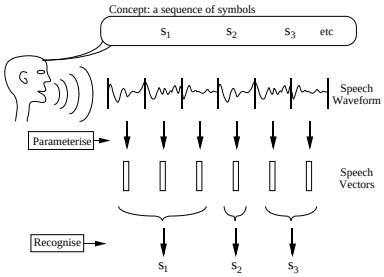


Fig. 1.1 Message Encoding/Decoding

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.

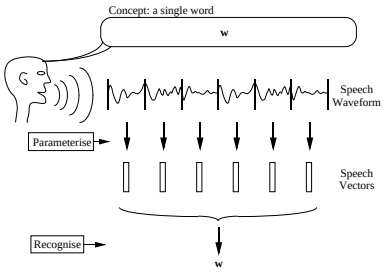


Fig. 1.2 Isolated Word Problem

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*.

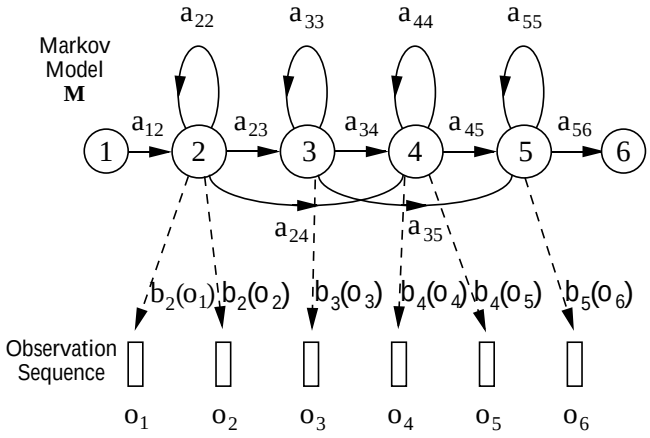


Fig. 1.3 The Markov Generation Model

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.

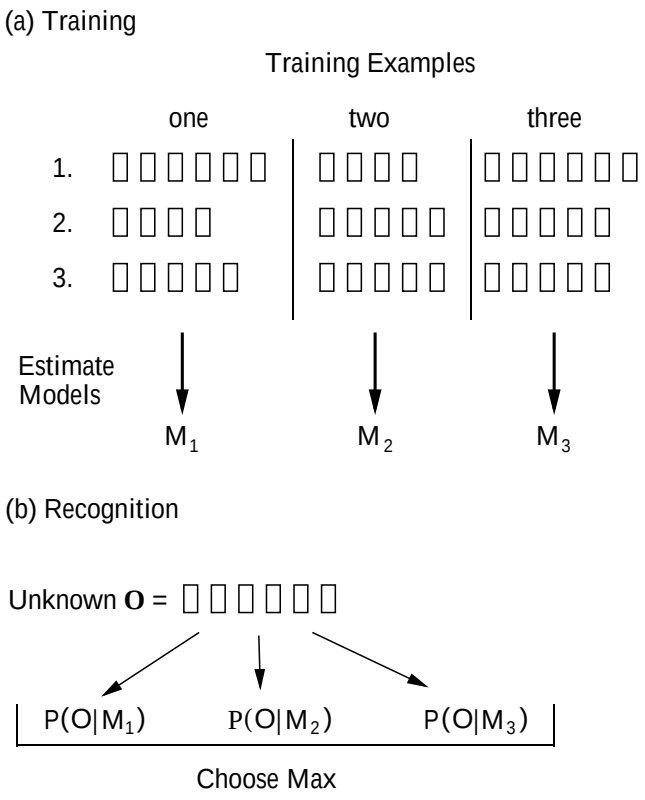


Fig. 1.4 Using HMMs for Isolated Word Recognition

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.

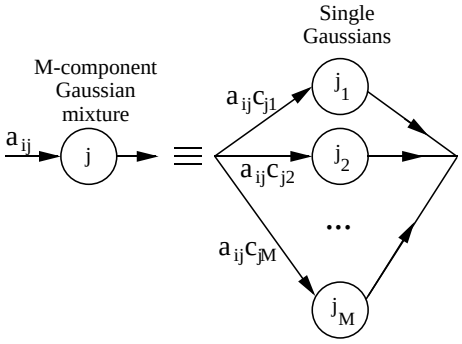


Fig. 1.5 Representing a Mixture

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.

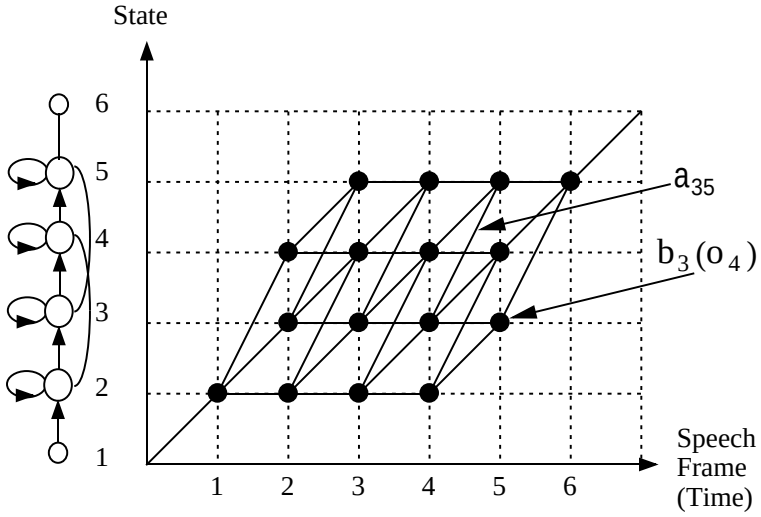


Fig. 1.6 The Viterbi Algorithm for Isolated Word Recognition

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:

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

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

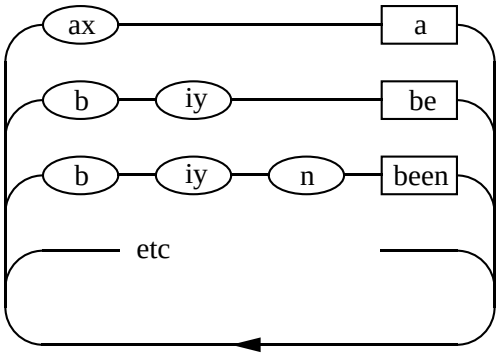


Fig. 1.7 Recognition Network for Continuously Spoken Word Recognition

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.

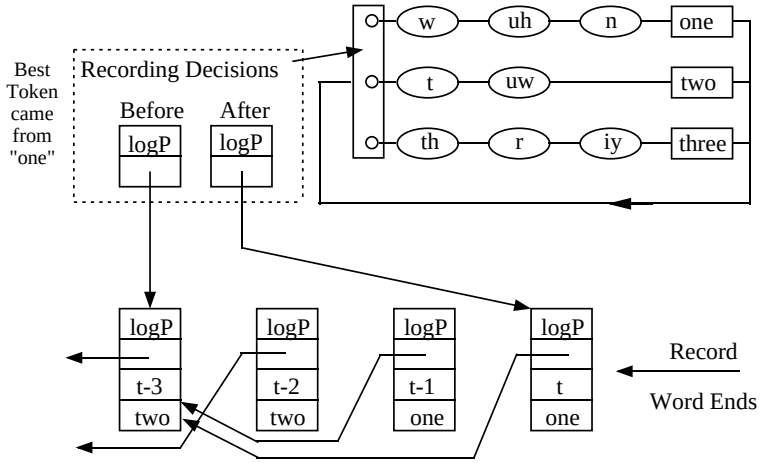


Fig. 1.8 Recording Word Boundary Decisions

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.

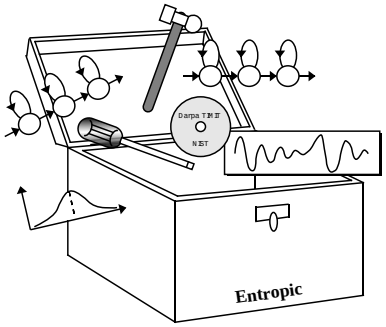
The above outlines the main idea of Token Passing as it is implemented within HTK. The algorithms are embedded in the library modules HNET and HREC and they may be invoked using the recogniser tool called HVITE. They provide single and multiple-token passing recognition, single-best output, lattice output, 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.

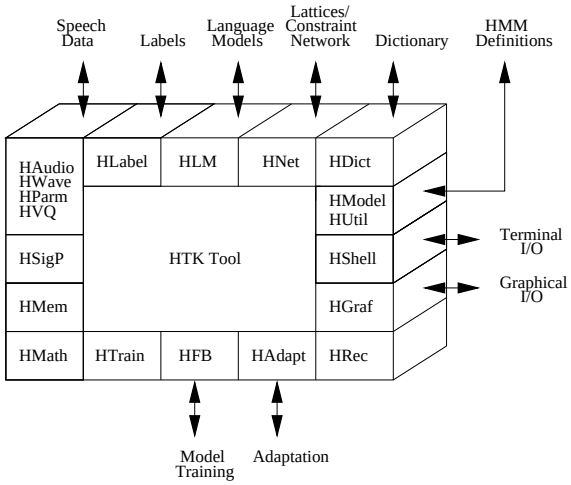


Fig. 2.1 Software Architecture

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 18.

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. Multiple configuration files can be specified by repeating the `-C` option, e.g.

```
HFoo -C config1 -C config2 -f 34.3 -a -s myfile file1 file2
```

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.

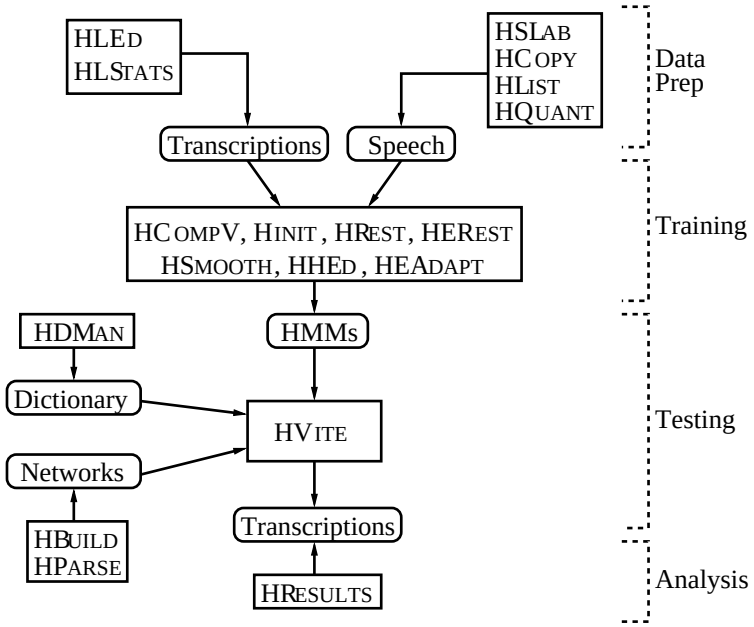


Fig. 2.2 HTK Processing Stages

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.

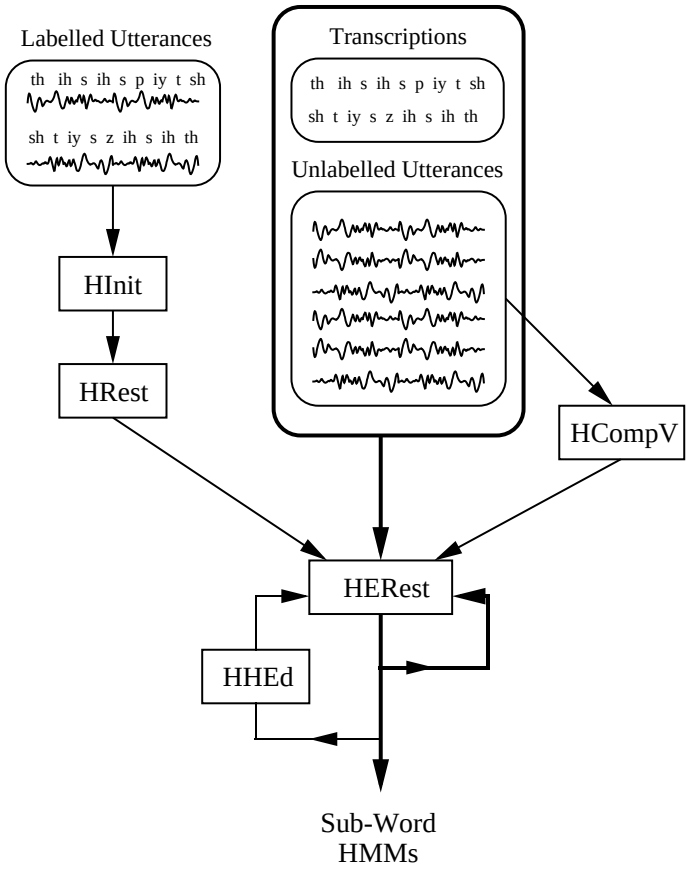


Fig. 2.3 Training Sub-word HMMs

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 3.2

This section lists the new features in HTK Version 3.2 compared to the preceding Version 3.1.

1. The HLM toolkit has been incorporated into HTK. It supports the training and testing of word or class-based n-gram language models.

2. HPARM supports global feature space transforms.

3. HPARM now supports third differentials ($\Delta\Delta\Delta$ parameters).
4. A new tool named HLRescore offers support for a number of lattice post-processing operations such as lattice pruning, finding the 1-best path in a lattice and language model expansion of lattices.
5. HERest supports 2-model re-estimation which allows the use of a separate alignment model set in the Baum-Welch re-estimation.
6. The initialisation of the decision-tree state clustering in HHEd has been improved.
7. HHEd supports a number of new commands related to variance flooring and decreasing the number of mixtures.
8. A major bug in the estimation of block-diagonal MLLR transforms has been fixed.
9. Many other smaller changes and bug fixes have been integrated.

2.4.1 New In Version 3.1

This section lists the new features in HTK Version 3.1 compared to the preceding Version 3.0 which was functionally equivalent to Version 2.2.

1. HPARM supports Perceptual Linear Prediction (PLP) feature extraction.
2. HPARM supports Vocal Tract Length Normalisation (VTLN) by warping the frequency axis in the filterbank analysis.
3. HPARM supports variance scaling.
4. HPARM supports cluster-based cepstral mean and variance normalisation.
5. All tools support an extended filename syntax that can be used to deal with unsegmented data more easily.

2.4.2 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

2.4.3 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.

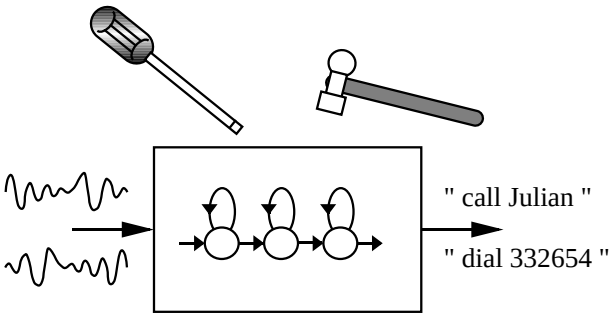
6. HTK now provides consistent support for reading and writing of HTK binary files (waveforms, binary MMFs, binary SLFs, HERest accumulators) across different machine architectures incorporating automatic byte swapping. By default, all binary data files handled by the tools are now written/read in big-endian (`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 17), 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.

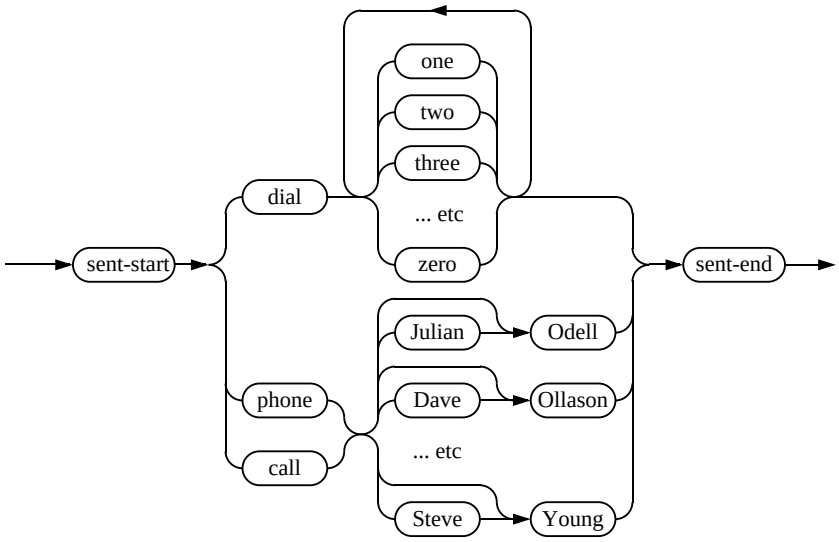


Fig. 3.1 Grammar for Voice Dialling

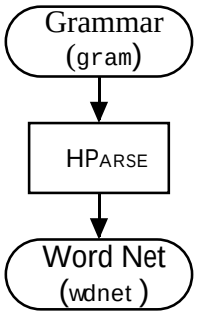


Fig. 3.2
Step 1

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).

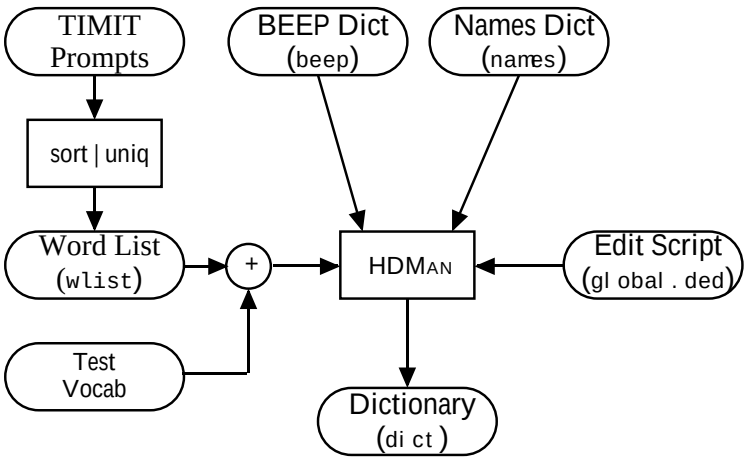


Fig. 3.3 Step 2

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

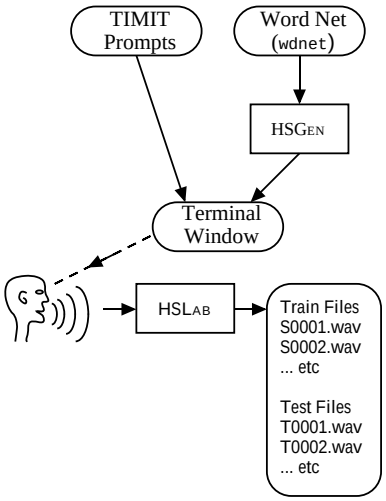


Fig. 3.4 Step 3

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.

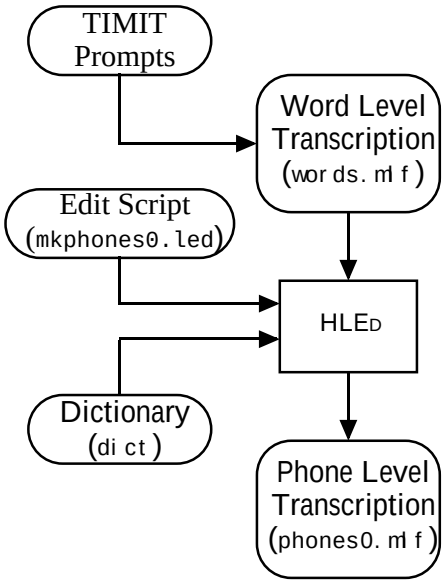


Fig. 3.5 Step 4

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].

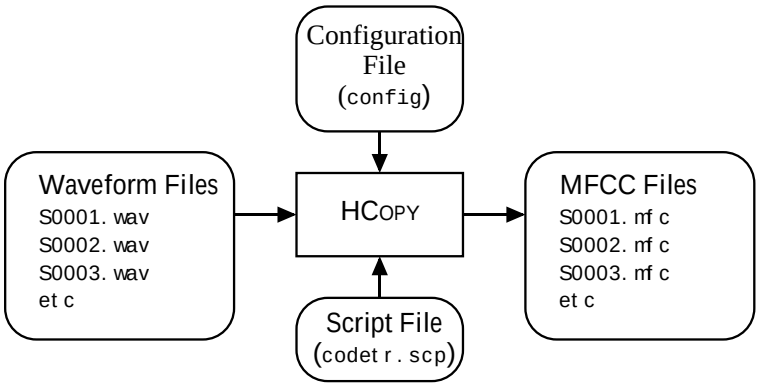


Fig. 3.6 Step 5

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).

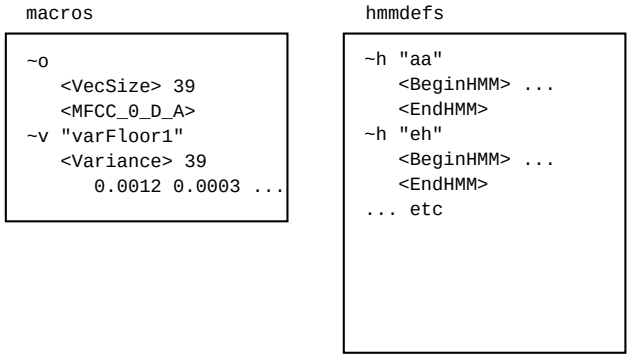


Fig. 3.7 Form of Master Macro Files

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`.

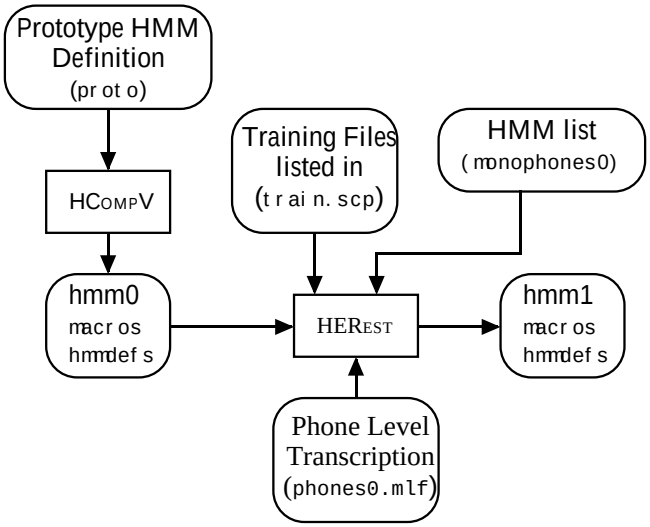


Fig. 3.8 Step 6

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

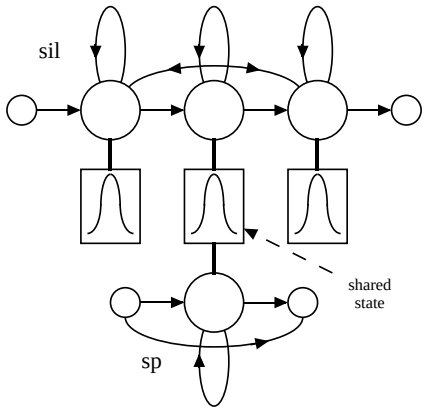


Fig. 3.9 Silence Models

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.

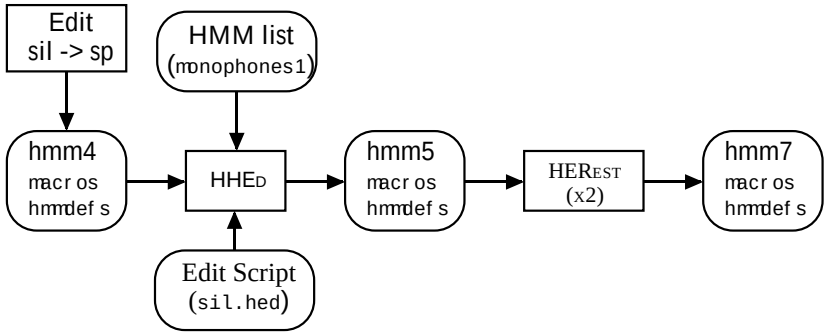


Fig. 3.10 Step 7

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.

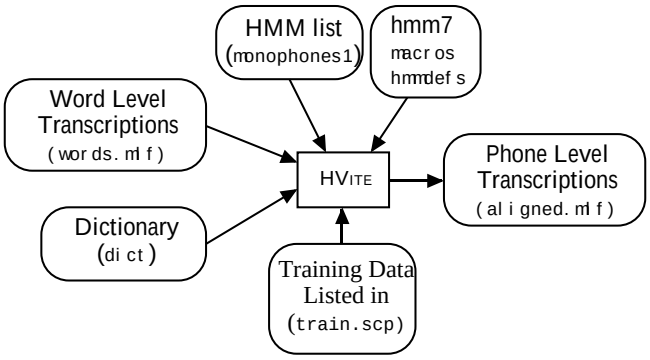


Fig. 3.11 Step 8

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.

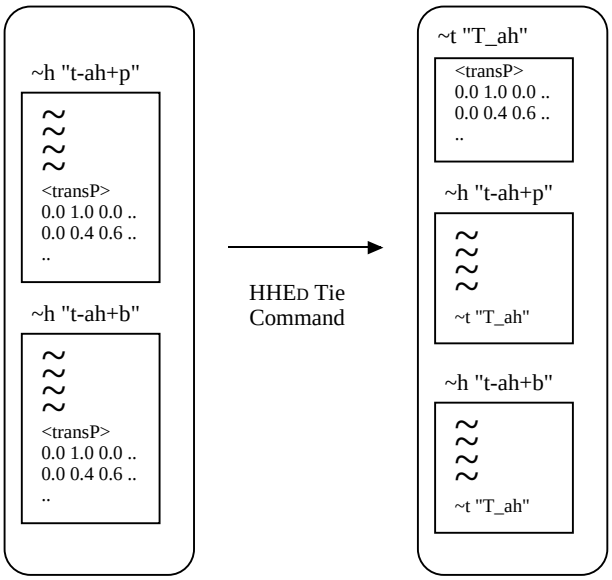


Fig. 3.12 Tying Transition Matrices

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.

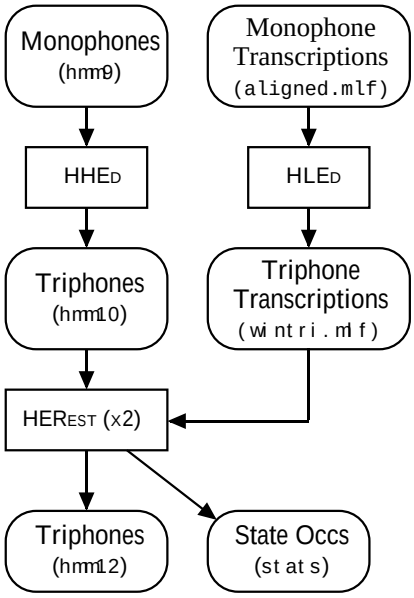


Fig. 3.13 Step 9

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`.

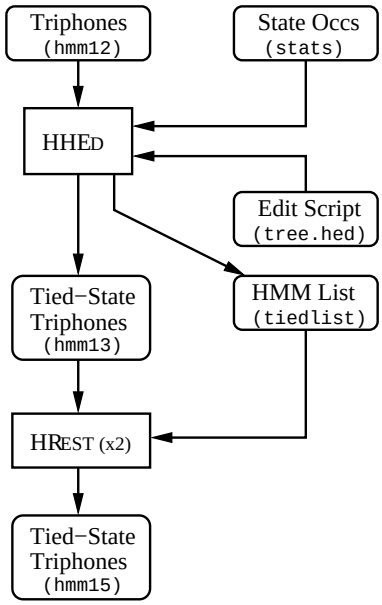


Fig. 3.14 Step 10

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.

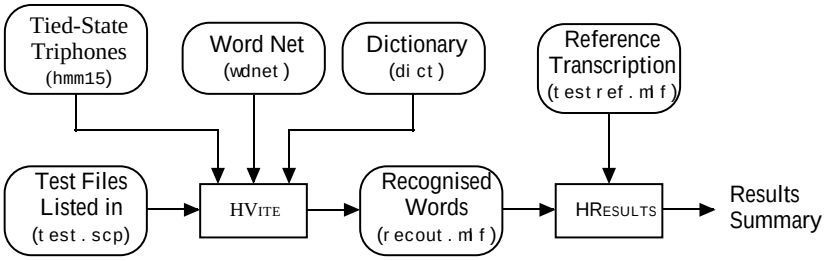


Fig. 3.15 Step 11

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

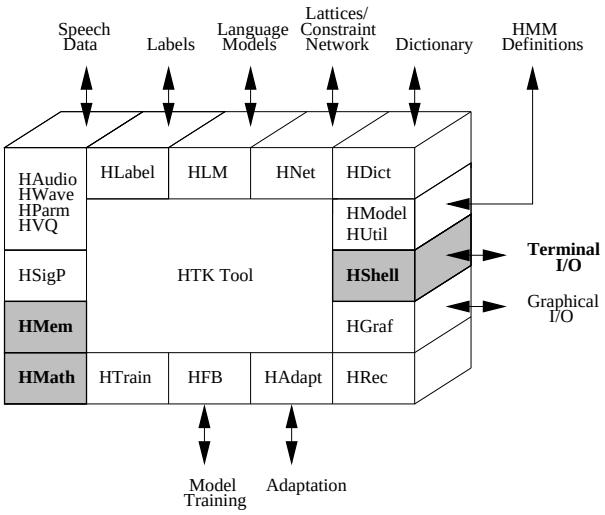
This chapter has described the construction of a tied-state phone-based continuous speech recogniser and in so doing, it has touched on most of the main areas addressed by HTK: recording, data preparation, HMM definitions, training tools, adaptation tools, networks, decoding and evaluating. The rest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

Part II

HTK in Depth

Chapter 4

The Operating Environment



This chapter discusses the various ways of controlling the operation of HTK tools along with related aspects of file system organisation, error reporting and memory management. All of the operating system and user interface functions are provided by the HTK module HSHELL. Memory management is a low level function which is largely invisible to the user, but it is useful to have a basic understanding of it in order to appreciate memory requirements and interpret diagnostic output from tools. Low level memory management in HTK is provided by HMEM and the management of higher level structures such as vectors and matrices is provided by HMATH.

The behaviour of a HTK tool depends on three sources of information. Firstly, all HTK tools are executed by issuing commands to the operating system shell. Each command typically contains the names of the various files that the tool needs to function and a number of optional arguments which control the detailed behaviour of the tool. Secondly, as noted in chapter 2 and shown in the adjacent figure, every HTK tool uses a set of standard library modules to interface to the various file types and to connect with the outside world. Many of these modules can be customised by setting parameters in a *configuration file*. Thirdly, a small number of parameters are specified using environment variables.

Terminal output mostly depends on the specific tool being used, however, there are some generic output functions which are provided by the library modules and which are therefore common across tools. These include version reporting, memory usage and error reporting.

Finally, HTK can read and write most data sources through pipes as an alternative to direct input and output from data files. This allows filters to be used, and in particular, it allows many of the external files used by HTK to be stored directly in compressed form and then decompressed *on-the-fly* when the data is read back in.

All of the above is discussed in more detail in the following sections.

4.1 The Command Line

The general form of command line for invoking a tool is[1]

```
tool [options] files ...
```

Options always consist of a dash followed by a single letter. Some options are followed by an argument as follows

```
-i              - a switch option
-t 3            - an integer valued option
-a 0.01         - a float valued option
-s hello        - a string valued option
```

Option names consisting of a capital letter are common across all tools (see section 4.4). Integer arguments may be given in any of the standard C formats, for example, `13`, `0xD` and `015` all represent the same number. Typing the name of a tool on its own always causes a short summary of the command line options to be printed in place of its normal operation. For example, typing

```
HERest
```

would result in the following output

```
USAGE: HERest [options] hmmList dataFiles...

 Option                                    Default

 -c f    Mixture pruning threshold          10.0
 -d s    dir to find hmm definitions       current
 -m N    set min examples needed per model   3
 -o s    extension for new hmm files        as src
 -p N    set parallel mode to N             off
 ...
```

The first line shows the names of the required files and the rest consists of a listing of each option, its meaning, and its default value.

The precise naming convention for specifying files depends on the operating system being used, but HTK always assumes the existence of a hierarchical file system and it maintains a distinction between directory paths and file names.

In general, a file will be located either in the current directory, some subdirectory of the current directory or some subdirectory of the root directory. For example, in the command

```
HList s1 dir/s2 /users/sjy/speech/s3
```

file `s1` must be in the current directory, `s2` must be in the directory `dir` within the current directory and `s3` must be in the directory `/users/sjy/speech`.

Some tools allow directories to be specified via configuration parameters and command line options. In all cases, the final path character (eg / in UNIX) need not (but may be) included. For example, both of the following are acceptable and have equivalent effect

```
HInit -L mymodels/new/  hmmfile data*
HInit -L mymodels/new   hmmfile data*
```

where the `-L` option specifies the directory in which to find the label files associated with the data files.

4.2 Script Files

Tools which require a potentially very long list of files (e.g. training tools) always allow the files to be specified in a script file via the `-S` option instead of via the command line. This is particularly useful when running under an OS with limited file name expansion capability. Thus, for example, HInit may be invoked by either

[1] All of the examples in this book assume the UNIX Operating System and the C Shell but the principles apply to any OS which supports hierarchical files and command line arguments

```
HInit hmmfile s1 s2 s3 s4 s5 ....
```

or

```
HInit -S filelist hmmfile
```

where `filelist` holds the list of files `s1`, `s2`, etc. Each file listed in a script should be separated by white space or a new line. Usually, files are listed on separate lines, however, when using HCopy which read pairs of files as its arguments, it is normal to write each pair on a single line. Script files should only be used for storing ellipsed file list arguments. Note that shell meta-characters should not be used in script files and will not be interpreted by the HTK tools.

Starting with HTK 3.1 the syntax of script fies has been extended. In addition to directly specifying the name of a physical file it is possible to define aliases and to select a segment from a file. The general syntax of an extended filename is

```
logfile=physfile[s,e]
```

where `logfile` is the logical filename used by the HTK tools and will appear in mlf files and similar. `physfile` is the physical name of the actual file on disk that will be accessed and `s` and `e` are indices that can be used to select only a segment of the file. One example of a use of this feature is the evaluation of different segmentations of the audio data. A new segmentation can be used by creating a new script file without having to create multiple copies of the data.

A typical script file might look like:

```
s23-0001-A_000143_000291.plp=/data/plp/complete/s23-0001-A.plp[143,291]
s23-0001-A_000291_000500.plp=/data/plp/complete/s23-0001-A.plp[291,500]
s23-0001-A_000500_000889.plp=/data/plp/complete/s23-0001-A.plp[500,889]
```

4.3 Configuration Files

Configuration files are used for customising the HTK working environment. They consist of a list of parameter-values pairs along with an optional prefix which limits the scope of the parameter to a specific module or tool.

The name of a configuration file can be specified explicitly on the command line using the `-C` command. For example, when executing

```
HERest ... -C myconfig  s1 s2 s3 s4 ...
```

The operation of HERest will depend on the parameter settings in the file `myconfig`.

When an explicit configuration file is specified, only those parameters mentioned in that file are actually changed and all other parameters retain their default values. These defaults are built-in. However, user-defined defaults can be set by assigning the name of a default configuration file to the environment variable `HCONFIG`. Thus, for example, using the UNIX C Shell, writing

```
setenv HCONFIG myconfig
HERest ... s1 s2 s3 s4 ...
```

would have an identical effect to the preceding example. However, in this case, a further refinement of the configuration values is possible since the opportunity to specify an explicit configuration file on the command line remains. For example, in

```
setenv HCONFIG myconfig
HERest ... -C xconfig  s1 s2 s3 s4 ...
```

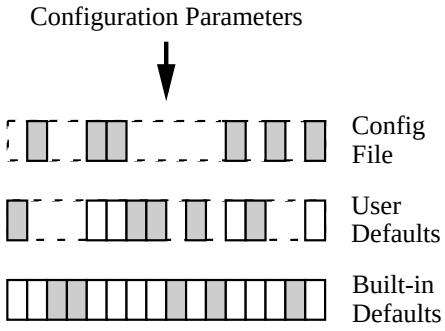


Fig. 4.1 Defining a Configuration

the parameter values in `xconfig` will over-ride those in `myconfig` which in turn will over-ride the built-in defaults. In practice, most HTK users will set general-purpose default configuration values using `HCONFIG` and will then over-ride these as required for specific tasks using the `-C` command line option. This is illustrated in Fig. 4.1 where the darkened rectangles indicate *active* parameter definitions. Viewed from above, all of the remaining parameter definitions can be seen to be masked by higher level over-rides.

The configuration file itself consists of a sequence of parameter definitions of the form

```
[MODULE:] PARAMETER = VALUE
```

One parameter definition is written per line and square brackets indicate that the module name is optional. Parameter definitions are not case sensitive but by convention they are written in upper case. A `#` character indicates that the rest of the line is a comment.

As an example, the following is a simple configuration file

```
# Example config file
        TARGETKIND = MFCC
        NUMCHANS   = 20
        WINDOWSIZE = 250000.0    # ie 25 msecs
        PREEMCOEF = 0.97
        ENORMALISE = T
HSHELL: TRACE = 02               # octal
HPARM:  TRACE = 0101
```

The first five lines contain no module name and hence they apply globally, that is, any library module or tool which is interested in the configuration parameter `NUMCHANS` will read the given parameter value. In practice, this is not a problem with library modules since nearly all configuration parameters have unique names. The final two lines show the same parameter name being given different values within different modules. This is an example of a parameter which every module responds to and hence does not have a unique name.

This example also shows each of the four possible types of value that can appear in a configuration file: string, integer, float and Boolean. The configuration parameter `TARGETKIND` requires a string value specifying the name of a speech parameter kind. Strings not starting with a letter should be enclosed in double quotes. `NUMCHANS` requires an integer value specifying the number of filter-bank channels to use in the analysis. `WINDOWSIZE` actually requires a floating-point value specifying the window size in units of 100ns. However, an integer can always be given wherever a float is required. `PREEMCOEF` also requires a floating-point value specifying the pre-emphasis coefficient to be used. Finally, `ENORMALISE` is a Boolean parameter which determines whether or not energy normalisation is to be performed, its value must be `T`, `TRUE` or `F`, `FALSE`. Notice also that, as in command line options, integer values can use the C conventions for writing in non-decimal bases. Thus, the trace value of 0101 is equal to decimal 65. This is particularly useful in this case because trace values are typically interpreted as bit-strings by HTK modules and tools.

If the name of a configuration variable is mis-typed, there will be no warning and the variable will simply be ignored. To help guard against this, the standard option `-D` can be used. This

displays all of the configuration variables before and after the tool runs. In the latter case, all configuration variables which are still unread are marked by a hash character. The initial display allows the configuration values to be checked before potentially wasting a large amount of cpu time through incorrectly set parameters. The final display shows which configuration variables were actually used during the execution of the tool. The form of the output is shown by the following example

```
HTK Configuration Parameters[3]
  Module/Tool     Parameter                  Value
#                 SAVEBINARY                  TRUE
  HPARM           TARGETRATE          256000.000000
                  TARGETKIND                 MFCC_0
```

Here three configuration parameters have been set but the hash (`#`) indicates that `SAVEBINARY` has not been used.

4.4 Standard Options

As noted in section 4.1, options consisting of a capital letter are common across all tools. Many are specific to particular file types and they will be introduced as they arise. However, there are six options that are standard across all tools. Three of these have been mentioned already. The option `-C` is used to specify a configuration file name and the option `-S` is used to specify a script file name, whilst the option `-D` is used to display configuration settings.

The two remaining standard options provided directly by HSHELL are `-A` and `-V`. The option `-A` causes the current command line arguments to be printed. When running experiments via scripts, it is a good idea to use this option to record in a log file the precise settings used for each tool. The option `-V` causes version information for the tool and each module used by that tool to be listed. These should always be quoted when making bug reports.

Finally, all tools implement the trace option `-T`. Trace values are typically bit strings and the meaning of each bit is described in the reference section for each tool. Setting a trace option via the command line overrides any setting for that same trace option in a configuration file. This is a general rule, command line options always override defaults set in configuration files.

All of the standard options are listed in the final summary section of this chapter. As a general rule, you should consider passing at least -A -D -V -T 1 to all tools, which will guarantee that sufficient information is available in the tool output.

4.5 Error Reporting

The HSHELL module provides a standard mechanism for reporting errors and warnings. A typical error message is as follows

```
HList: ERROR [+1110]
  IsWave: cannot open file speech.dat
```

This indicates that the tool HLIST is reporting an error number +1110. All errors have positive error numbers and always result in the tool terminating. Warnings have negative error numbers and the tool does not terminate. The first two digits of an error number indicate the module or tool in which the error is located (HLIST in this case) and the last two digits define the class of error. The second line of the error message names the actual routine in which the error occurred (here `IsWave`) and the actual error message. All errors and warnings are listed in the reference section at the end of this book indexed by error/warning number. This listing contains more details on each error or warning along with suggested causes.

Error messages are sent to the standard error stream but warnings are sent to the standard output stream. The reason for the latter is that most HTK tools are run with progress tracing enabled. Sending warnings to the standard output stream ensures that they are properly interleaved with the trace of progress so that it is easy to determine the point at which the warning was issued. Sending warnings to standard error would lose this information.

The default behaviour of a HTK tool on terminating due to an error is to exit normally returning the error number as exit status. If, however, the configuration variable `ABORTONERR` is set to true then the tool will core dump. This is a debugging facility which should not concern most users.

4.6 Strings and Names

Many HTK definition files include names of various types of objects: for example labels, model names, words, etc. In order to achieve some uniformity, HTK applies standard rules for reading strings which are names. These rules are not, however, necessary when using the language modelling tools – see below.

A name string consists of a single white space delimited word or a quoted string. Either the single quote ' or the double quote " can be used to quote strings but the start and end quotes must be matched. The backslash \ character can also be used to introduce otherwise reserved characters. The character following a backslash is inserted into the string without special processing unless that character is a digit in the range 0 to 7. In that case, the three characters following the backslash are read and interpreted as an octal character code. When the three characters are not octal digits the result is not well defined.

In summary the special processing is

Notation	Meaning
\\	\
_	represents a space that will not terminate a string
\'	' (and will not end a quoted string)
\"	" (and will not end a quoted string)
\nnn	the character with octal code \nnn

Note that the above allows the same effect to be achieved in a number of different ways. For example,

```
"\"QUOTE"
\"QUOTE
'"QUOTE'
\042QUOTE
```

all produce the string "QUOTE.

The only exceptions to the above general rules are:

- Where models are specified in HHEd scripts, commas (,), dots (.), and closing brackets ()) are all used as extra delimiters to allow HHEd scripts created for earlier versions of HTK to be used unchanged. Hence for example, (a,b,c,d) would be split into 4 distinct name strings a, b, c and d.
- When the configuration variable RAWMITFORMAT is set true, each word in a language model definition file consists of a white space delimited string with no special processing being performed.
- Source dictionaries read by HDMan are read using the standard HTK string conventions, however, the command IR can be used in a HDMan source edit script to switch to using this raw format.
- To ensure that the general definition of a name string works properly in HTK master label files, all MLFs must have the reserved . and /// terminators alone on a line with no surrounding white space. If this causes problems reading old MLF files, the configuration variable V1COMPAT should be set true in the module HLabel. In this case, HTK will attempt to simulate the behaviour of the older version 1.5.
- To force numbers to be interpreted as strings rather than times or scores in a label file, they must be quoted. If the configuration variable QUOTECHAR is set to ' or " then output labels will be quoted with the specified quote character. If QUOTECHAR is set to \, then output labels will be escaped. The default is to select the simplest quoting mechanism.

Note that under some versions of Unix HTK can support the 8-bit character sets used for the representation of various orthographies. In such cases the shell environment variable $LANG usually governs which ISO character set is in use.

Language modelling tools

Although these string conventions are unnecessary in HLM, to maintain compatibility with HTKthe same conventions are used. However, a number of options are provided to allow a mix of escaped and unescaped text files to be handled. Word maps allow the type of escaping (HTK or none) to be defined in their headers. When a degenerate form of word map is used (i.e. a map with no header), the LWMap configuration variable `INWMAPRAW` may be set to true to disable HTK escaping. By default, HLM tools output word lists and maps in HTK escaped form. However, this can be overridden by setting the configuration variable `OUTWMAPRAW` to true. Similar conventions apply to class maps. A degenerate class map can be read in raw mode by setting the LClass configuration variable `INCMAPRAW` to true, and a class map can be written in raw form by setting `OUTCMAPRAW` to true.

Input/output of N-gram language model files are handled by the HLM module `LModel`. Hence, by default input/output of LMs stored in the ARPA-MIT text format will assume HTK escaping conventions. This can be disabled for both input and output by setting `RAWMITFORMAT` to true.

4.7 Memory Management

Memory management is a very low level function and is mostly invisible to HTK users. However, some applications require very large amounts of memory. For example, building the models for a large vocabulary continuous speech dictation system might require 150MB or more. Clearly, when memory demands become this large, a proper understanding of the impact of system design decisions on memory usage is important. The first step in this is to have a basic understanding of memory allocation in HTK.

Many HTK tools dynamically construct large and complex data structures in memory. To keep strict control over this and to reduce memory allocation overheads to an absolute minimum, HTK performs its own memory management. Thus, every time that a module or tool wishes to allocate some memory, it does so by calling routines in HMem. At a slightly higher level, math objects such as vectors and matrices are allocated by HMath but using the primitives provided by HMem.

To make memory allocation and de-allocation very fast, tools create specific memory allocators for specific objects or groups of objects. These memory allocators are divided into a sequence of blocks, and they are organised as either Stacks, M-heaps or C-heaps. A Stack constrains the pattern of allocation and de-allocation requests to be made in a last-allocated first-deallocated order but allows objects of any size to be allocated. An M-heap allows an arbitrary pattern of allocation and de-allocation requests to be made but all allocated objects must be the same size. Both of these memory allocation disciplines are more restricted than the general mechanism supplied by the operating system, and as a result, such memory operations are faster and incur no storage overhead due to the need to maintain hidden housekeeping information in each allocated object. Finally, a C-heap uses the underlying operating system and allows arbitrary allocation patterns, and as a result incurs the associated time and space overheads. The use of C-heaps is avoided wherever possible.

Most tools provide one or more trace options which show how much memory has been allocated. The following shows the form of the output

```
---------------------- Heap Statistics ------------------------
nblk=1, siz= 100000*1, used= 32056, alloc= 100000 : Global Stack[S]
nblk=1, siz=   200*28, used=   100, alloc=   5600 : cellHeap[M]
nblk=1, siz=  10000*1, used=  3450, alloc=  10000 : mlfHeap[S]
nblk=2, siz=   7504*1, used=  9216, alloc=  10346 : nameHeap[S]
---------------------------------------------------------------
```

Each line describes the status of each memory allocator and gives the number of blocks allocated, the current block size (number of elements in block $\times$ the number of bytes in each element)[2], the total number of bytes in use by the tool and the total number of bytes currently allocated to that allocator. The end of each line gives the name of the allocator and its type: Stack[S], M-heap[M] or C-heap[M]. The element size for Stacks will always be 1 but will be variable in M-heaps. The documentation for the memory intensive HTK tools indicates what each of the main memory allocators are used for and this information allows the effects of various system design choices to be monitored.

[2] Block sizes typically grow as more blocks are allocated

4.8 Input/Output via Pipes and Networks

Most types of file in HTK can be input or output via a pipe instead of directly from or to disk. The mechanism for doing this is to assign the required input or output filter command to a configuration parameter or to an environment variable, either can be used. Within this command, any occurrence of the dollar symbol `$` will be replaced by the name of the required file. The output of the command will then be input to or output from the HTK tool via a pipe.

For example, the following command will normally list the contents of the speech waveform file `spfile`

```
HList spfile
```

However, if the value of the environment variable `HWAVEFILTER` is set as follows

```
setenv HWAVEFILTER 'gunzip -c $'
```

then the effect is to invoke the decompression filter `gunzip` with its input connected to the file `spfile` and its output connected to HLIST via a pipe. Each different type of file has a unique associated variable so that multiple input and/or filters can be used. The full list of these is given in the summary section at the end of this chapter.

HTK is often used to process large amounts of data and typically this data is distributed across a network. In many systems, an attempt to open a file can fail because of temporary network *glitches*. In the majority of cases, a second or third attempt to open the file a few seconds later will succeed and all will be well. To allow this to be done automatically, HTK tools can be configured to retry opening a file several times before giving up. This is done simply by setting the configuration parameter `MAXTRYOPEN` to the required number of retries[3].

4.9 Byte-swapping of HTK data files

Virtually all HTK tools can read and write data to and from binary files. The use of binary format as opposed to text can speed up the performance of the tools and at the same time reduce the file size when manipulating large quantities of data. Typical binary files used by the HTK tools are speech waveform/parameter files, binary master model files (MMF), binary accumulator files used in HMM parameter estimation and binary lattice files. However, the use of binary data format often introduces incompatibilities between different machine architectures due to the different byte ordering conventions used to represent numerical quantities. In such cases, byte swapping of the data is required. To avoid incompatibilities across different machine architectures, all HTK binary data files are written out using big-endian (`NONVAX`) representation of numerical values. Similarly, during loading HTK binary format files are assumed to be in `NONVAX` byte order. The default behavior can be altered using the configuration parameters `NATURALREADORDER` and `NATURALWRITEORDER`. Setting `NATURALREADORDER` to true will instruct the HTK tools to interpret the binary input data in the machine's natural byte order (byte swapping will never take place). Similarly, setting `NATURALWRITEORDER` to true will instruct the tools to write out data using the machine's natural byte order. The default value of these two configuration variables is false which is the appropriate setting when using HTK in a multiple machine architecture environment. In an environment comprising entirely of machines with `VAX` byte order both configuration parameters can be set true which will disable the byte swapping procedure during reading and writing of data.

4.10 Summary

This section summarises the globally-used environment variables and configuration parameters. It also provides a list of all the standard command line options used with HTK.

Table 4.1 lists all of the configuration parameters along with a brief description. A missing module name means that it is recognised by more than one module. Table 4.2 lists all of the environment parameters used by these modules. Finally, table 4.3 lists all of the standard options.

[3] This does not work if input filters are used.

Module	Name	Description
HSHELL	ABORTONERR	Core dump on error (for debugging)
HSHELL	HWAVEFILTER	Filter for waveform file input
HSHELL	HPARMFILTER	Filter for parameter file input
HSHELL	HLANGMODFILTER	Filter for language model file input
HSHELL	HMMLISTFILTER	Filter for HMM list file input
HSHELL	HMMDEFFILTER	Filter for HMM definition file input
HSHELL	HLABELFILTER	Filter for Label file input
HSHELL	HNETFILTER	Filter for Network file input
HSHELL	HDICTFILTER	Filter for Dictionary file input
HSHELL	LGRAMFILTER	Filter for gram file input
HSHELL	LWMAPFILTER	Filter for word map file input
HSHELL	LCMAPFILTER	Filter for class map file input
HSHELL	LMTEXTFILTER	Filter for text file input
HSHELL	HWAVEOFILTER	Filter for waveform file output
HSHELL	HPARMOFILTER	Filter for parameter file output
HSHELL	HLANGMODOFILTER	Filter for language model file output
HSHELL	HMMLISTOFILTER	Filter for HMM list file output
HSHELL	HMMDEFOFILTER	Filter for HMM definition file output
HSHELL	HLABELOFILTER	Filter for Label file output
HSHELL	HNETOFILTER	Filter for Network file output
HSHELL	HDICTOFILTER	Filter for Dictionary file output
HSHELL	LGRAMOFILTER	Filter for gram file output
HSHELL	LWMAPOFILTER	Filter for word map file output
HSHELL	LCMAPOFILTER	Filter for class map file output
HSHELL	MAXTRYOPEN	Number of file open retries
HSHELL	NONUMESCAPES	Prevent string output using \012 format
HSHELL	NATURALREADORDER	Enable natural read order for HTK binary files
HSHELL	NATURALWRITEORDER	Enable natural write order for HTK binary files
HMEM	PROTECTSTAKS	Warn if stack is cut-back (debugging)
	TRACE	Trace control (default=0)
	STARTWORD	Set sentence start symbol (<s>)
	ENDWORD	Set sentence end symbol (</s>)
	UNKNOWNNAME	Set OOV class symbol (!!UNK)
	RAWMITFORMAT	Disable HTK escaping for LM tools
LWMAP	INWMAPRAW	Disable HTK escaping for input word lists and maps
LWMAP	OUTWMAPRAW	Disable HTK escaping for output word lists and maps
LCMAP	INCMAPRAW	Disable HTK escaping for input class lists and maps
LCMAP	OUTCMAPRAW	Disable HTK escaping for output class lists and maps

Table. 4.1 Configuration Parameters used in Operating Environment

Env Variable	Meaning
HCONFIG	Name of default configuration file
HxxxFILTER	Input/Output filters as above

Table. 4.2 Environment Variables used in Operating Environment

Standard Option	Meaning
`-A`	Print command line arguments
`-B`	Store output HMM macro files in binary
`-C cf`	Configuration file is cf
`-D`	Display configuration variables
`-F fmt`	Set source data file format to fmt
`-G fmt`	Set source label file format to fmt
`-H mmf`	Load HMM macro file mmf
`-I mlf`	Load master label file mlf
`-J tmf`	Load transform model file tmf
`-K tmf`	Save transform model file tmf
`-L dir`	Look for label files in directory dir
`-M dir`	Store output HMM macro files in directory dir
`-O fmt`	Set output data file format to fmt
`-P fmt`	Set output label file format to fmt
`-Q`	Print command summary info
`-S scp`	Use command line script file scp
`-T N`	Set trace level to N
`-V`	Print version information
`-X ext`	Set label file extension to ext

Table. 4.3 Summary of Standard Options

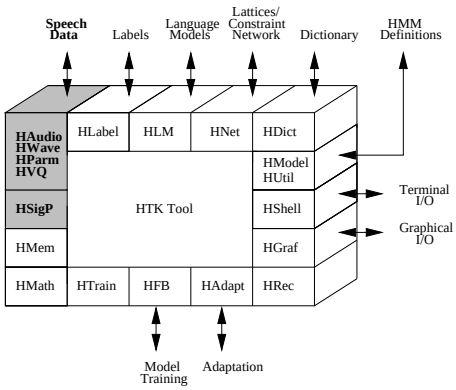
Chapter 5

Speech Input/Output

Many tools need to input parameterised speech data and HTK provides a number of different methods for doing this:

- input from a previously encoded speech parameter file
- input from a waveform file which is encoded as part of the input processing
- input from an audio device which is encoded as part of the input processing.

For input from a waveform file, a large number of different file formats are supported, including all of the commonly used CD-ROM formats. Input/output for parameter files is limited to the standard HTK file format and the new Entropic Esignal format.



All HTK speech input is controlled by configuration parameters which give details of what processing operations to apply to each input speech file or audio source. This chapter describes speech input/output in HTK. The general mechanisms are explained and the various configuration parameters are defined. The facilities for signal pre-processing, linear prediction-based processing, Fourier-based processing and vector quantisation are presented and the supported file formats are given. Also described are the facilities for augmenting the basic speech parameters with energy measures, delta coefficients and acceleration (delta-delta) coefficients and for splitting each parameter vector into multiple data streams to form *observations*. The chapter concludes with a brief description of the tools HLIST and HCOPY which are provided for viewing, manipulating and encoding speech files.

5.1 General Mechanism

The facilities for speech input and output in HTK are provided by five distinct modules: HAUDIO, HWAVE, HPARM, HVQ and HSIGP. The interconnections between these modules are shown in Fig. 5.1.

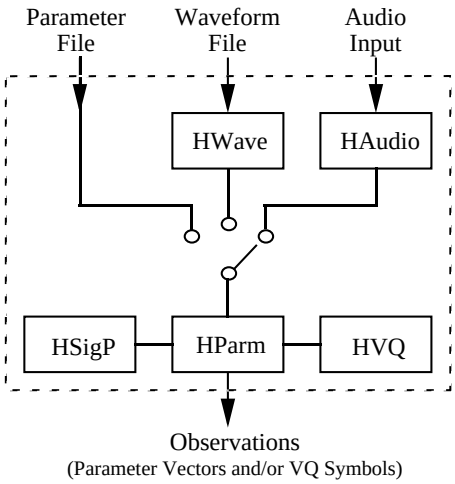


Fig. 5.1 Speech Input Subsystem

Waveforms are read from files using HWAVE, or are input direct from an audio device using HAUDIO. In a few rare cases, such as in the display tool HSLAB, only the speech waveform is needed. However, in most cases the waveform is wanted in parameterised form and the required encoding is performed by HPARM using the signal processing operations defined in HSIGP. The parameter vectors are output by HPARM in the form of observations which are the basic units of data processed by the HTK recognition and training tools. An observation contains all components of a raw parameter vector but it may be possibly split into a number of independent parts. Each such part is regarded by a HTK tool as a statistically independent data stream. Also, an observation may include VQ indices attached to each data stream. Alternatively, VQ indices can be read directly from a parameter file in which case the observation will contain only VQ indices.

Usually a HTK tool will require a number of speech data files to be specified on the command line. In the majority of cases, these files will be required in parameterised form. Thus, the following example invokes the HTK embedded training tool HEREST to re-estimate a set of models using the speech data files `s1`, `s2`, `s3`, ... These are input via the library module HPARM and they must be in exactly the form needed by the models.

```
HERest ...  s1 s2 s3 s4 ...
```

However, if the external form of the speech data files is not in the required form, it will often be possible to convert them automatically during the input process. To do this, configuration parameter values are specified whose function is to define exactly how the conversion should be done. The key idea is that there is a *source parameter kind* and *target parameter kind*. The source refers to the natural form of the data in the external medium and the target refers to the form of the data that is required internally by the HTK tool. The principle function of the speech input subsystem is to convert the source parameter kind into the required target parameter kind.

Parameter kinds consist of a base form to which one or more qualifiers may be attached where each qualifier consists of a single letter preceded by an underscore character. Some examples of parameter kinds are

`WAVEFORM`	simple waveform
`LPC`	linear prediction coefficients
`LPC_D_E`	LPC with energy and delta coefficients
`MFCC_C`	compressed mel-cepstral coefficients

The required source and target parameter kinds are specified using the configuration parameters `SOURCEKIND` and `TARGETKIND`. Thus, if the following configuration parameters were defined

```
SOURCEKIND = WAVEFORM
TARGETKIND = MFCC_E
```

then the speech input subsystem would expect each input file to contain a speech waveform and it would convert it to mel-frequency cepstral coefficients with log energy appended.

The source need not be a waveform. For example, the configuration parameters

```
SOURCEKIND = LPC
TARGETKIND = LPREFC
```

would be used to read in files containing linear prediction coefficients and convert them to reflection coefficients.

For convenience, a special parameter kind called ANON is provided. When the source is specified as ANON then the actual kind of the source is determined from the input file. When ANON is used in the target kind, then it is assumed to be identical to the source. For example, the effect of the following configuration parameters

```
SOURCEKIND = ANON
TARGETKIND = ANON_D
```

would simply be to add delta coefficients to whatever the source form happened to be. The source and target parameter kinds default to ANON to indicate that by default no input conversions are performed. Note, however, that where two or more files are listed on the command line, the meaning of ANON will not be re-interpreted from one file to the next. Thus, it is a general rule, that any tool reading multiple source speech files requires that all the files have the same parameter kind.

The conversions applied by HTK's input subsystem can be complex and may not always behave exactly as expected. There are two facilities that can be used to help check and debug the set-up of the speech i/o configuration parameters. Firstly, the tool HList simply displays speech data by listing it on the terminal. However, since HList uses the speech input subsystem like all HTK tools, if a value for TARGETKIND is set, then it will display the target form rather than the source form. This is the simplest way to check the form of the speech data that will actually be delivered to a HTK tool. HList is described in more detail in section 5.15 below.

Secondly, trace output can be generated from the HParm module by setting the TRACE configuration file parameter. This is a bit-string in which individual bits cover different parts of the conversion processing. The details are given in the reference section.

To summarise, speech input in HTK is controlled by configuration parameters. The key parameters are SOURCEKIND and TARGETKIND which specify the source and target parameter kinds. These determine the end-points of the required input conversion. However, to properly specify the detailed steps in between, more configuration parameters must be defined. These are described in subsequent sections.

5.2 Speech Signal Processing

In this section, the basic mechanisms involved in transforming a speech waveform into a sequence of parameter vectors will be described. Throughout this section, it is assumed that the SOURCEKIND is WAVEFORM and that data is being read from a HTK format file via HWave. Reading from different format files is described below in section 5.11. Much of the material in this section also applies to data read direct from an audio device, the additional features needed to deal with this latter case are described later in section 5.12.

The overall process is illustrated in Fig. 5.2 which shows the sampled waveform being converted into a sequence of parameter blocks. In general, HTK regards both waveform files and parameter files as being just sample sequences, the only difference being that in the former case the samples are 2-byte integers and in the latter they are multi-component vectors. The sample rate of the input waveform will normally be determined from the input file itself. However, it can be set explicitly using the configuration parameter SOURCERATE. The period between each parameter vector determines the output sample rate and it is set using the configuration parameter TARGETRATE. The segment of waveform used to determine each parameter vector is usually referred to as a window and its size is set by the configuration parameter WINDOWSIZE. Notice that the window size and frame rate are independent. Normally, the window size will be larger than the frame rate so that successive windows overlap as illustrated in Fig. 5.2.

For example, a waveform sampled at 16kHz would be converted into 100 parameter vectors per second using a 25 msec window by setting the following configuration parameters.

```
SOURCERATE = 625
TARGETRATE = 100000
WINDOWSIZE = 250000
```

Remember that all durations are specified in 100 nsec units[1].

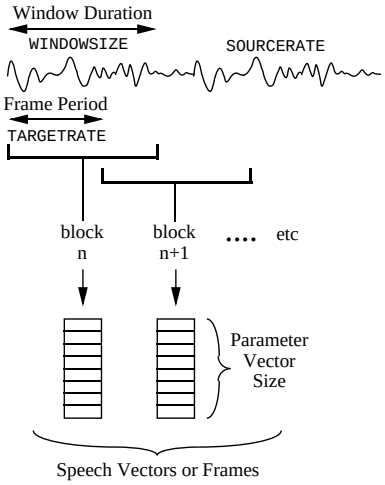


Fig. 5.2 Speech Encoding Process

Independent of what parameter kind is required, there are some simple pre-processing operations that can be applied prior to performing the actual signal analysis. Firstly, the DC mean can be removed from the source waveform by setting the Boolean configuration parameter `ZMEANSOURCE` to true (i.e. `T`). This is useful when the original analogue-digital conversion has added a DC offset to the signal. It is applied to each window individually so that it can be used both when reading from a file and when using direct audio input[2].

Secondly, it is common practice to pre-emphasise the signal by applying the first order difference equation

$$s'_n = s_n - k\, s_{n-1} \tag{5.1}$$

to the samples $\{s_n, n = 1, N\}$ in each window. Here k is the pre-emphasis coefficient which should be in the range $0 \leq k < 1$. It is specified using the configuration parameter `PREEMCOEF`. Finally, it is usually beneficial to taper the samples in each window so that discontinuities at the window edges are attenuated. This is done by setting the Boolean configuration parameter `USEHAMMING` to true. This applies the following transformation to the samples $\{s_n, n = 1, N\}$ in the window

$$s'_n = \left\{0.54 - 0.46 \cos\left(\frac{2\pi(n-1)}{N-1}\right)\right\} s_n \tag{5.2}$$

When both pre-emphasis and Hamming windowing are enabled, pre-emphasis is performed first.

In practice, all three of the above are usually applied. Hence, a configuration file will typically contain the following

```
ZMEANSOURCE = T
USEHAMMING = T
PREEMCOEF = 0.97
```

[1] The somewhat bizarre choice of 100nsec units originated in Version 1 of HTK when times were represented by integers and this unit was the best compromise between precision and range. Times are now represented by doubles and hence the constraints no longer apply. However, the need for backwards compatibility means that 100nsec units have been retained. The names `SOURCERATE` and `TARGETRATE` are also non-ideal, `SOURCEPERIOD` and `TARGETPERIOD` would be better.

[2] This method of applying a zero mean is different to HTK Version 1.5 where the mean was calculated and subtracted from the whole speech file in one operation. The configuration variable `V1COMPAT` can be set to revert to this older behaviour.

Certain types of artificially generated waveform data can cause numerical overflows with some coding schemes. In such cases adding a small amount of random noise to the waveform data solves the problem. The noise is added to the samples using

$$s'_n = s_n + qRND() \tag{5.3}$$

where $RND()$ is a uniformly distributed random value over the interval $[-1.0, +1.0)$ and q is the scaling factor. The amount of noise added to the data (q) is set with the configuration parameter `ADDDITHER` (default value 0.0). A positive value causes the noise signal added to be the same every time (ensuring that the same file always gives exactly the same results). With a negative value the noise is random and the same file may produce slightly different results in different trials.

One problem that can arise when processing speech waveform files obtained from external sources, such as databases on CD-ROM, is that the byte-order may be different to that used by the machine on which HTK is running. To deal with this problem, HWAVE can perform automatic byte-swapping in order to preserve proper byte order. HTK assumes by default that speech waveform data is encoded as a sequence of 2-byte integers as is the case for most current speech databases[3]. If the source format is known, then HWAVE will also make an assumption about the byte order used to create speech files in that format. It then checks the byte order of the machine that it is running on and automatically performs byte-swapping if the order is different. For unknown formats, proper byte order can be ensured by setting the configuration parameter `BYTEORDER` to `VAX` if the speech data was created on a little-endian machine such as a VAX or an IBM PC, and to anything else (e.g. `NONVAX`) if the speech data was created on a big-endian machine such as a SUN, HP or Macintosh machine.

The reading/writing of HTK format waveform files can be further controlled via the configuration parameters `NATURALREADORDER` and `NATURALWRITEORDER`. The effect and default settings of these parameters are described in section 4.9. Note that `BYTEORDER` should not be used when `NATURALREADORDER` is set to true. Finally, note that HTK can also byte-swap parameterised files in a similar way provided that only the byte-order of each 4 byte float requires inversion.

5.3 Linear Prediction Analysis

In linear prediction (LP) analysis, the vocal tract transfer function is modelled by an all-pole filter with transfer function[4]

$$H(z) = \frac{1}{\sum_{i=0}^{p} a_i z^{-i}} \tag{5.4}$$

where p is the number of poles and $a_0 \equiv 1$. The filter coefficients $\{a_i\}$ are chosen to minimise the mean square filter prediction error summed over the analysis window. The HTK module HSIGP uses the *autocorrelation method* to perform this optimisation as follows.

Given a window of speech samples $\{s_n, n = 1, N\}$, the first $p + 1$ terms of the autocorrelation sequence are calculated from

$$r_i = \sum_{j=1}^{N-i} s_j s_{j+i} \tag{5.5}$$

where $i = 0, p$. The filter coefficients are then computed recursively using a set of auxiliary coefficients $\{k_i\}$ which can be interpreted as the reflection coefficients of an equivalent acoustic tube and the prediction error E which is initially equal to r_0. Let $\{k_j^{(i-1)}\}$ and $\{a_j^{(i-1)}\}$ be the reflection and filter coefficients for a filter of order $i - 1$, then a filter of order i can be calculated in three steps. Firstly, a new set of reflection coefficients are calculated.

$$k_j^{(i)} = k_j^{(i-1)} \tag{5.6}$$

for $j = 1, i - 1$ and

$$k_i^{(i)} = \left\{ r_i + \sum_{j=1}^{i-1} a_j^{(i-1)} r_{i-j} \right\} / E^{(i-1)} \tag{5.7}$$

[3] Many of the more recent speech databases use compression. In these cases, the data may be regarded as being logically encoded as a sequence of 2-byte integers even if the actual storage uses a variable length encoding scheme.

[4] Note that some textbooks define the denominator of equation 5.4 as $1 - \sum_{i=1}^{p} a_i z^{-i}$ so that the filter coefficients are the negatives of those computed by HTK.

Secondly, the prediction energy is updated.

$$E^{(i)} = (1 - k_i^{(i)} k_i^{(i)}) E^{(i-1)} \tag{5.8}$$

Finally, new filter coefficients are computed

$$a_j^{(i)} = a_j^{(i-1)} - k_i^{(i)} a_{i-j}^{(i-1)} \tag{5.9}$$

for $j = 1, i-1$ and

$$a_i^{(i)} = -k_i^{(i)} \tag{5.10}$$

This process is repeated from $i = 1$ through to the required filter order $i = p$.

To effect the above transformation, the target parameter kind must be set to either `LPC` to obtain the LP filter parameters $\{a_i\}$ or `LPREFC` to obtain the reflection coefficients $\{k_i\}$. The required filter order must also be set using the configuration parameter `LPCORDER`. Thus, for example, the following configuration settings would produce a target parameterisation consisting of 12 reflection coefficients per vector.

```
TARGETKIND = LPREFC
LPCORDER = 12
```

An alternative LPC-based parameterisation is obtained by setting the target kind to `LPCEPSTRA` to generate linear prediction cepstra. The cepstrum of a signal is computed by taking a Fourier (or similar) transform of the log spectrum. In the case of linear prediction cepstra, the required spectrum is the linear prediction spectrum which can be obtained from the Fourier transform of the filter coefficients. However, it can be shown that the required cepstra can be more efficiently computed using a simple recursion

$$c_n = -a_n - \frac{1}{n} \sum_{i=1}^{n-1} (n-i) a_i c_{n-i} \tag{5.11}$$

The number of cepstra generated need not be the same as the number of filter coefficients, hence it is set by a separate configuration parameter called `NUMCEPS`.

The principal advantage of cepstral coefficients is that they are generally decorrelated and this allows diagonal covariances to be used in the HMMs. However, one minor problem with them is that the higher order cepstra are numerically quite small and this results in a very wide range of variances when going from the low to high cepstral coefficients. HTK does not have a problem with this but for pragmatic reasons such as displaying model parameters, flooring variances, etc., it is convenient to re-scale the cepstral coefficients to have similar magnitudes. This is done by setting the configuration parameter `CEPLIFTER` to some value L to *lifter* the cepstra according to the following formula

$$c'_n = \left(1 + \frac{L}{2} sin \frac{\pi n}{L}\right) c_n \tag{5.12}$$

As an example, the following configuration parameters would use a 14'th order linear prediction analysis to generate 12 liftered LP cepstra per target vector

```
TARGETKIND = LPCEPSTRA
LPCORDER = 14
NUMCEPS = 12
CEPLIFTER = 22
```

These are typical of the values needed to generate a good front-end parameterisation for a speech recogniser based on linear prediction.

Finally, note that the conversions supported by HTK are not limited to the case where the source is a waveform. HTK can convert any LP-based parameter into any other LP-based parameter.

5.4 Filterbank Analysis

The human ear resolves frequencies non-linearly across the audio spectrum and empirical evidence suggests that designing a front-end to operate in a similar non-linear manner improves recognition performance. A popular alternative to linear prediction based analysis is therefore filterbank

analysis since this provides a much more straightforward route to obtaining the desired non-linear frequency resolution. However, filterbank amplitudes are highly correlated and hence, the use of a cepstral transformation in this case is virtually mandatory if the data is to be used in a HMM based recogniser with diagonal covariances.

HTK provides a simple Fourier transform based filterbank designed to give approximately equal resolution on a mel-scale. Fig. 5.3 illustrates the general form of this filterbank. As can be seen, the filters used are triangular and they are equally spaced along the mel-scale which is defined by

$$\text{Mel}(f) = 2595 \log_{10}(1 + \frac{f}{700}) \tag{5.13}$$

To implement this filterbank, the window of speech data is transformed using a Fourier transform and the magnitude is taken. The magnitude coefficients are then *binned* by correlating them with each triangular filter. Here binning means that each FFT magnitude coefficient is multiplied by the corresponding filter gain and the results accumulated. Thus, each bin holds a weighted sum representing the spectral magnitude in that filterbank channel. As an alternative, the Boolean configuration parameter `USEPOWER` can be set true to use the power rather than the magnitude of the Fourier transform in the binning process.

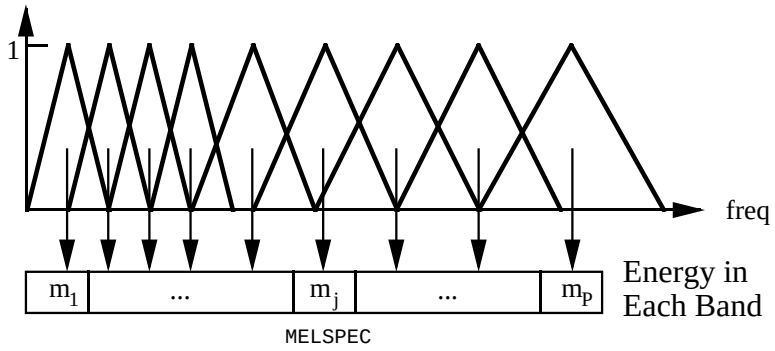


Fig. 5.3 Mel-Scale Filter Bank

Normally the triangular filters are spread over the whole frequency range from zero upto the Nyquist frequency. However, band-limiting is often useful to reject unwanted frequencies or avoid allocating filters to frequency regions in which there is no useful signal energy. For filterbank analysis only, lower and upper frequency cut-offs can be set using the configuration parameters `LOFREQ` and `HIFREQ`. For example,

```
LOFREQ = 300
HIFREQ = 3400
```

might be used for processing telephone speech. When low and high pass cut-offs are set in this way, the specified number of filterbank channels are distributed equally on the mel-scale across the resulting pass-band such that the lower cut-off of the first filter is at `LOFREQ` and the upper cut-off of the last filter is at `HIFREQ`.

If mel-scale filterbank parameters are required directly, then the target kind should be set to `MELSPEC`. Alternatively, log filterbank parameters can be generated by setting the target kind to `FBANK`.

5.5 Vocal Tract Length Normalisation

A simple speaker normalisation technique can be implemented by modifying the filterbank analysis described in the previous section. Vocal tract length normalisation (VTLN) aims to compensate for the fact that speakers have vocal tracts of different sizes. VTLN can be implemented by warping the frequency axis in the filterbank analysis. In HTK simple linear frequency warping is supported. The warping factor α is controlled by the configuration variable `WARPFREQ`. Here values of $\alpha < 1.0$ correspond to a compression of the frequency axis. As the warping would lead to some filters

being placed outside the analysis frequency range, the simple linear warping function is modified at the upper and lower boundaries. The result is that the lower boundary frequency of the analysis (`LOFREQ`) and the upper boundary frequency (`HIFREQ`) are always mapped to themselves. The regions in which the warping function deviates from the linear warping with factor α are controlled with the two configuration variables (`WARPLCUTOFF`) and (`WARPUCUTOFF`). Figure 5.4 shows the overall shape of the resulting piece-wise linear warping functions.

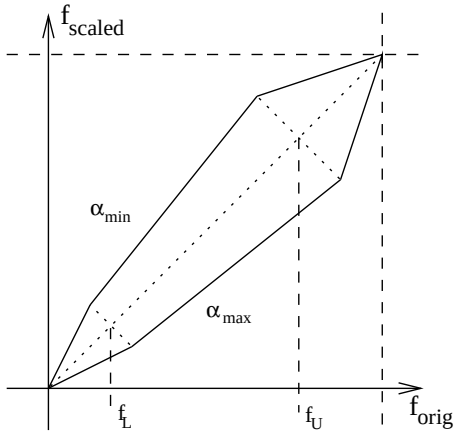


Fig. 5.4 Frequency Warping

The warping factor α can for example be found using a search procedure that compares likelihoods at different warping factors. A typical procedure would involve recognising an utterance with $\alpha = 1.0$ and then performing forced alignment of the hypothesis for all warping factors in the range $0.8 - 1.2$. The factor that gives the highest likelihood is selected as the final warping factor. Instead of estimating a separate warping factor for each utterance, large units can be used by for example estimating only one α per speaker.

Vocal tract length normalisation can be applied in testing as well as in training the acoustic models.

5.6 Cepstral Features

Most often, however, cepstral parameters are required and these are indicated by setting the target kind to `MFCC` standing for Mel-Frequency Cepstral Coefficients (MFCCs). These are calculated from the log filterbank amplitudes $\{m_j\}$ using the Discrete Cosine Transform

$$c_i = \sqrt{\frac{2}{N}} \sum_{j=1}^{N} m_j \cos\left(\frac{\pi i}{N}(j - 0.5)\right) \tag{5.14}$$

where N is the number of filterbank channels set by the configuration parameter `NUMCHANS`. The required number of cepstral coefficients is set by `NUMCEPS` as in the linear prediction case. Liftering can also be applied to MFCCs using the `CEPLIFTER` configuration parameter (see equation 5.12).

MFCCs are the parameterisation of choice for many speech recognition applications. They give good discrimination and lend themselves to a number of manipulations. In particular, the effect of inserting a transmission channel on the input speech is to multiply the speech spectrum by the channel transfer function. In the log cepstral domain, this multiplication becomes a simple addition which can be removed by subtracting the cepstral mean from all input vectors. In practice, of course, the mean has to be estimated over a limited amount of speech data so the subtraction will not be perfect. Nevertheless, this simple technique is very effective in practice where it compensates for long-term spectral effects such as those caused by different microphones and audio channels. To perform this so-called *Cepstral Mean Normalisation* (CMN) in HTK it is only necessary to add the `_Z` qualifier to the target parameter kind. The mean is estimated by computing the average of each cepstral parameter across each input speech file. Since this cannot be done with live audio, cepstral mean compensation is not supported for this case.

In addition to the mean normalisation the variance of the data can be normalised. For improved robustness both mean and variance of the data should be calculated on a larger units (e.g. on all the data from a speaker instead of just on a single utterance). To use speaker-/cluster-based normalisation the mean and variance estimates are computed offline before the actual recognition and stored in separate files (two files per cluster). The configuration variables `CMEANDIR` and `VARSCALEDIR` point to the directories where these files are stored. To find the actual filename a second set of variables (`CMEANMASK` and `VARSCALEMASK`) has to be specified. These masks are regular expressions in which you can use the special characters `?`, `*` and `%`. The appropriate mask is matched against the filename of the file to be recognised and the substring that was matched against the `%` characters is used as the filename of the normalisation file. An example config setting is:

```
CMEANDIR     = /data/eval01/plp/cmn
CMEANMASK    = %%%%%%%%%_*
VARSCALEDIR  = /data/eval01/plp/cvn
VARSCALEMASK = %%%%%%%%%_*
VARSCALEFN   = /data/eval01/plp/globvar
```

So, if the file `sw1-4930-B_4930Bx-sw1_000126_000439.plp` is to be recognised then the normalisation estimates would be loaded from the following files:

```
/data/eval01/plp/cmn/sw1-4930-B
/data/eval01/plp/cvn/sw1-4930-B
```

The file specified by `VARSCALEFN` contains the global target variance vector, i.e. the variance of the data is first normalised to 1.0 based on the estimate in the appropriate file in `VARSCALEDIR` and then scaled to the target variance given in `VARSCALEFN`.

The format of the files is very simple and each of them just contains one vector. Note that in the case of the cepstral mean only the static coefficients will be normalised. A cmn file could for example look like:

```
<CEPSNORM> <PLP_0>
<MEAN> 13
-10.285290 -9.484871 -6.454639 ...
```

The cepstral variance normalised always applies to the full observation vector after all qualifiers like delta and acceleration coefficients have been added, e.g.:

```
<CEPSNORM> <PLP_D_A_Z_0>
<VARIANCE> 39
33.543018 31.241779 36.076199 ...
```

The global variance vector will always have the same number of dimensions as the cvn vector, e.g.:

```
<VARSCALE> 39
 2.974308e+01 4.143743e+01 3.819999e+01 ...
```

These estimates can be generated using HCompV. See the reference section for details.

5.7 Perceptual Linear Prediction

An alternative to the Mel-Frequency Cepstral Coefficients is the use of Perceptual Linear Prediction (PLP) coefficients.

As implemented in HTK the PLP feature extraction is based on the standard mel-frequency filterbank (possibly warped). The mel filterbank coefficients are weighted by an equal-loudness curve and then compressed by taking the cubic root.[5] From the resulting auditory spectrum LP coefficents are estimated which are then converted to cepstral coefficents in the normal way (see above).

[5] the degree of compression can be controlled by setting the configuration parameter `COMPRESSFACT` which is the power to which the amplitudes are raised and defaults to 0.33)

5.8 Energy Measures

To augment the spectral parameters derived from linear prediction or mel-filterbank analysis, an energy term can be appended by including the qualifier _E in the target kind. The energy is computed as the log of the signal energy, that is, for speech samples $\{s_n, n = 1, N\}$

$$E = log \sum_{n=1}^{N} s_n^2 \tag{5.15}$$

This log energy measure can be normalised to the range $-E_{min}..1.0$ by setting the Boolean configuration parameter ENORMALISE to true (default setting). This normalisation is implemented by subtracting the maximum value of E in the utterance and adding 1.0. Note that energy normalisation is incompatible with live audio input and in such circumstances the configuration variable ENORMALISE should be explicitly set false. The lowest energy in the utterance can be clamped using the configuration parameter SILFLOOR which gives the ratio between the maximum and minimum energies in the utterance in dB. Its default value is 50dB. Finally, the overall log energy can be arbitrarily scaled by the value of the configuration parameter ESCALE whose default is 0.1.

When calculating energy for LPC-derived parameterisations, the default is to use the zeroth delay autocorrelation coefficient (r_0). However, this means that the energy is calculated after windowing and pre-emphasis. If the configuration parameter RAWENERGY is set true, however, then energy is calculated separately before any windowing or pre-emphasis regardless of the requested parameterisation[6].

In addition to, or in place of, the log energy, the qualifier _0 can be added to a target kind to indicate that the 0'th cepstral parameter C_0 is to be appended. This qualifier is only valid if the target kind is MFCC. Unlike earlier versions of HTK scaling factors set by the configuration variable ESCALE are not applied to C_0[7].

5.9 Delta, Acceleration and Third Differential Coefficients

The performance of a speech recognition system can be greatly enhanced by adding time derivatives to the basic static parameters. In HTK, these are indicated by attaching qualifiers to the basic parameter kind. The qualifier _D indicates that first order regression coefficients (referred to as delta coefficients) are appended, the qualifier _A indicates that second order regression coefficients (referred to as acceleration coefficients) and the qualifier _T indicates that third order regression coefficients (referred to as third differential coefficients) are appended. The _A qualifier cannot be used without also using the _D qualifier. Similarly the _T qualifier cannot be used without also using the _D and _A qualifiers.

The delta coefficients are computed using the following regression formula

$$d_t = \frac{\sum_{\theta=1}^{\Theta} \theta(c_{t+\theta} - c_{t-\theta})}{2\sum_{\theta=1}^{\Theta} \theta^2} \tag{5.16}$$

where d_t is a delta coefficient at time t computed in terms of the corresponding static coefficients $c_{t-\Theta}$ to $c_{t+\Theta}$. The value of Θ is set using the configuration parameter DELTAWINDOW. The same formula is applied to the delta coefficients to obtain acceleration coefficients except that in this case the window size is set by ACCWINDOW. Similarly the third differentials use THIRDWINDOW. Since equation 5.16 relies on past and future speech parameter values, some modification is needed at the beginning and end of the speech. The default behaviour is to replicate the first or last vector as needed to fill the regression window.

In older version 1.5 of HTK and earlier, this end-effect problem was solved by using simple first order differences at the start and end of the speech, that is

$$d_t = c_{t+1} - c_t, \quad t < \Theta \tag{5.17}$$

and

$$d_t = c_t - c_{t-1}, \quad t \geq T - \Theta \tag{5.18}$$

[6] In any event, setting the compatibility variable V1COMPAT to true in HPARM will ensure that the calculation of energy is compatible with that computed by the Version 1 tool HCode.

[7] Unless V1COMPAT is set to true.

where T is the length of the data file. If required, this older behaviour can be restored by setting the configuration variable `V1COMPAT` to true in HParm.

For some purposes, it is useful to use simple differences throughout. This can be achieved by setting the configuration variable `SIMPLEDIFFS` to true in HParm. In this case, just the end-points of the delta window are used, i.e.

$$d_t = \frac{(c_{t+\Theta} - c_{t-\Theta})}{2\Theta} \tag{5.19}$$

When delta and acceleration coefficients are requested, they are computed for all static parameters including energy if present. In some applications, the absolute energy is not useful but time derivatives of the energy may be. By including the `_E` qualifier together with the `_N` qualifier, the absolute energy is suppressed leaving just the delta and acceleration coefficients of the energy.

5.10 Storage of Parameter Files

Whereas HTK can handle waveform data in a variety of file formats, all parameterised speech data is stored externally in either native HTK format data files or Entropic Esignal format files. Entropic ESPS format is no longer supported directly, but input and output filters can be used to convert ESPS to Esignal format on input and Esignal to ESPS on output.

5.10.1 HTK Format Parameter Files

HTK format files consist of a contiguous sequence of *samples* preceded by a header. Each sample is a vector of either 2-byte integers or 4-byte floats. 2-byte integers are used for compressed forms as described below and for vector quantised data as described later in section 5.14. HTK format data files can also be used to store speech waveforms as described in section 5.11.

The HTK file format header is 12 bytes long and contains the following data

`nSamples`	– number of samples in file (4-byte integer)
`sampPeriod`	– sample period in 100ns units (4-byte integer)
`sampSize`	– number of bytes per sample (2-byte integer)
`parmKind`	– a code indicating the sample kind (2-byte integer)

The parameter kind consists of a 6 bit code representing the basic parameter kind plus additional bits for each of the possible qualifiers. The basic parameter kind codes are

0	`WAVEFORM`	sampled waveform
1	`LPC`	linear prediction filter coefficients
2	`LPREFC`	linear prediction reflection coefficients
3	`LPCEPSTRA`	LPC cepstral coefficients
4	`LPDELCEP`	LPC cepstra plus delta coefficients
5	`IREFC`	LPC reflection coef in 16 bit integer format
6	`MFCC`	mel-frequency cepstral coefficients
7	`FBANK`	log mel-filter bank channel outputs
8	`MELSPEC`	linear mel-filter bank channel outputs
9	`USER`	user defined sample kind
10	`DISCRETE`	vector quantised data

and the bit-encoding for the qualifiers (in octal) is

`_E`	000100	has energy
`_N`	000200	absolute energy suppressed
`_D`	000400	has delta coefficients
`_A`	001000	has acceleration coefficients
`_C`	002000	is compressed
`_Z`	004000	has zero mean static coef.
`_K`	010000	has CRC checksum
`_0`	020000	has 0'th cepstral coef.

The _A qualifier can only be specified when _D is also specified. The _N qualifier is only valid when both energy and delta coefficients are present. The sample kind LPDELCEP is identical to LPCEPSTRA_D and is retained for compatibility with older versions of HTK. The _C and _K only exist in external files. Compressed files are always decompressed on loading and any attached CRC is checked and removed. An external file can contain both an energy term and a 0'th order cepstral coefficient. These may be retained on loading but normally one or the other is discarded[8].

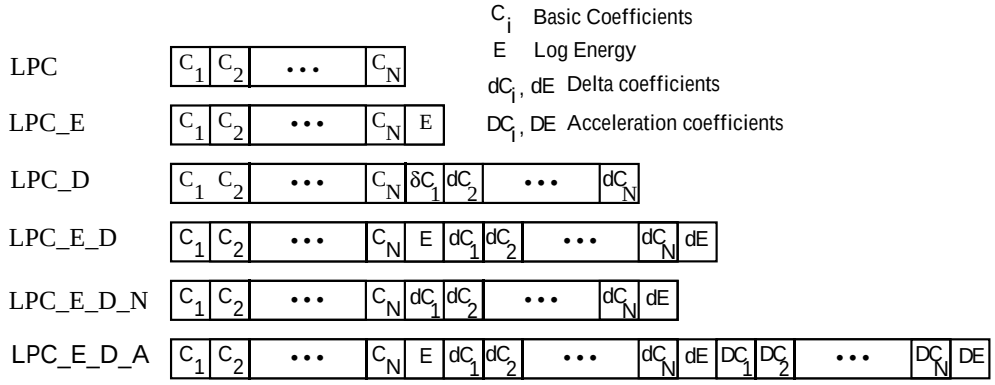


Fig. 5.5 Parameter Vector Layout in HTK Format Files

All parameterised forms of HTK data files consist of a sequence of vectors. Each vector is organised as shown by the examples in Fig 5.5 where various different qualified forms are listed. As can be seen, an energy value if present immediately follows the base coefficients. If delta coefficients are added, these follow the base coefficients and energy value. Note that the base form LPC is used in this figure only as an example, the same layout applies to all base sample kinds. If the 0'th order cepstral coefficient is included as well as energy then it is inserted immediately before the energy coefficient, otherwise it replaces it.

For external storage of speech parameter files, two compression methods are provided. For LP coding only, the IREFC parameter kind exploits the fact that the reflection coefficients are bounded by ± 1 and hence they can be stored as scaled integers such that $+1.0$ is stored as 32767 and -1.0 is stored as -32767. For other types of parameterisation, a more general compression facility indicated by the _C qualifier is used. HTK compressed parameter files consist of a set of compressed parameter vectors stored as shorts such that for parameter x

$$x_{short} = A * x_{float} - B$$

The coefficients A and B are defined as

$$\begin{aligned} A &= 2 * I/(x_{max} - x_{min}) \\ B &= (x_{max} + x_{min}) * I/(x_{max} - x_{min}) \end{aligned}$$

where x_{max} is the maximum value of parameter x in the whole file and x_{min} is the corresponding minimum. I is the maximum range of a 2-byte integer i.e. 32767. The values of A and B are stored as two floating point vectors prepended to the start of the file immediately after the header.

When a HTK tool writes out a speech file to external storage, no further signal conversions are performed. Thus, for most purposes, the target parameter kind specifies both the required internal representation and the form of the written output, if any. However, there is a distinction in the way that the external data is actually stored. Firstly, it can be compressed as described above by setting the configuration parameter SAVECOMPRESSED to true. If the target kind is LPREFC then this compression is implemented by converting to IREFC otherwise the general compression algorithm described above is used. Secondly, in order to avoid data corruption problems, externally stored HTK parameter files can have a cyclic redundancy checksum appended. This is indicated by the qualifier _K and it is generated by setting the configuration parameter SAVEWITHCRC to true. The principle tool which uses these output conversions is HCopy (see section 5.16).

[8] Some applications may require the 0'th order cepstral coefficient in order to recover the filterbank coefficients from the cepstral coefficients.

5.10.2 Esignal Format Parameter Files

The default for parameter files is native HTK format. However, HTK tools also support the Entropic Esignal format for both input and output. Esignal replaces the Entropic ESPS file format. To ensure compatibility Entropic provides conversion programs from ESPS to ESIG and vice versa.

To indicate that a source file is in Esignal format the configuration variable `SOURCEFORMAT` should be set to `ESIG`. Alternatively, `-F ESIG` can be specified as a command-line option. To generate Esignal format output files, the configuration variable `TARGETFORMAT` should be set to `ESIG` or the command line option `-O ESIG` should be set.

ESIG files consist of three parts: a preamble, a sequence of field specifications called the field list and a sequence of records. The preamble and the field list together constitute the header. The preamble is purely ASCII. Currently it consists of 6 information items that are all terminated by a new line. The information in the preamble is the following:

`line 1` – identification of the file format
`line 2` – version of the file format
`line 3` – architecture (ASCII, EDR1, EDR2, machine name)
`line 4` – preamble size (48 bytes)
`line 5` – total header size
`line 6` – record size

All ESIG files that are output by HTK programs contain the following global fields:

commandLine the command-line used to generate the file;

recordFreq a double value that indicates the sample frequency in Herz;

startTime a double value that indicates a time at which the first sample is presumed to be starting;

parmKind a character string that indicates the full type of parameters in the file, e.g: `MFCC_E_D`.

source_1 if the input file was an ESIG file this field includes the header items in the input file.

After that there are field specifiers for the records. The first specifier is for the basekind of the parameters, e.g: `MFCC`. Then for each available qualifier there are additional specifiers. Possible specifiers are:

```
zeroc
energy
delta
delta_zeroc
delta_energy
accs
accs_zeroc
accs_energy
```

The data segments of the ESIG files have exactly the same format as the the corresponding HTK files. This format was described in the previous section.

HTK can only input parameter files that have a valid parameter kind as value of the header field `parmKind`. If this field does not exist or if the value of this field does not contain a valid parameter kind, the file is rejected. After the header has been read the file is treated as an HTK file.

5.11 Waveform File Formats

For reading waveform data files, HTK can support a variety of different formats and these are all briefly described in this section. The default speech file format is HTK. If a different format is to be used, it can be specified by setting the configuration parameter `SOURCEFORMAT`. However, since file formats need to be changed often, they can also be set individually via the `-F` command-line option. This over-rides any setting of the `SOURCEFORMAT` configuration parameter.

Similarly for the output of waveforms, the format can be set using either the configuration parameter TARGETFORMAT or the -O command-line option. However, for output only native HTK format (HTK), Esignal format (ESIG) and headerless (NOHEAD) waveform files are supported.

The following sub-sections give a brief description of each of the waveform file formats supported by HTK.

5.11.1 HTK File Format

The HTK file format for waveforms is identical to that described in section 5.10 above. It consists of a 12 byte header followed by a sequence of 2 byte integer speech samples. For waveforms, the sampSize field will be 2 and the parmKind field will be 0. The sampPeriod field gives the sample period in 100ns units, hence for example, it will have the value 1000 for speech files sampled at 10kHz and 625 for speech files sampled at 16kHz.

5.11.2 Esignal File Format

The Esignal file format for waveforms is similar to that described in section 5.10 above with the following exceptions. When reading an ESIG waveform file the HTK programs only check whether the record length equals 2 and whether the datatype of the only field in the data records is SHORT. The data field that is created on output of a waveform is called WAVEFORM.

5.11.3 TIMIT File Format

The TIMIT format has the same structure as the HTK format except that the 12-byte header contains the following

hdrSize	– number of bytes in header ie 12 (2-byte integer)
version	– version number (2-byte integer)
numChannels	– number of channels (2-byte integer)
sampRate	– sample rate (2-byte integer)
nSamples	– number of samples in file (4-byte integer)

TIMIT format data is used only on the prototype TIMIT CD ROM.

5.11.4 NIST File Format

The NIST file format is also referred to as the Sphere file format. A NIST header consists of ASCII text. It begins with a label of the form NISTxx where xx is a version code followed by the number of bytes in the header. The remainder of the header consists of name value pairs of which HTK decodes the following

sample_rate	– sample rate in Hz
sample_n_bytes	– number of bytes in each sample
sample_count	– number of samples in file
sample_byte_format	– byte order
sample_coding	– speech coding eg pcm, μlaw, shortpack
channels_interleaved	– for 2 channel data only

The current NIST Sphere data format subsumes a variety of internal data organisations. HTK currently supports interleaved μlaw used in Switchboard, Shortpack compression used in the original version of WSJ0 and standard 16bit linear PCM as used in Resource Management, TIMIT, etc. It does not currently support the Shorten compression format as used in WSJ1 due to licensing restrictions. Hence, to read WSJ1, the files must be converted using the NIST supplied decompression routines into standard 16 bit linear PCM. This is most conveniently done under UNIX by using the decompression program as an input filter set via the environment variable HWAVEFILTER (see section 4.8).

For interleaved μlaw as used in Switchboard, the default is to add the two channels together. The left channel only can be obtained by setting the environment variable STEREOMODE to LEFT and the right channel only can be obtained by setting the environment variable STEREOMODE to RIGHT.

5.11.5 SCRIBE File Format

The SCRIBE format is a subset of the standard laid down by the European Esprit Programme SAM Project. SCRIBE data files are headerless and therefore consist of just a sequence of 16 bit sample values. HTK assumes by default that the sample rate is 20kHz. The configuration parameter `SOURCERATE` should be set to over-ride this. The byte ordering assumed for SCRIBE data files is `VAX` (little-endian).

5.11.6 SDES1 File Format

The SDES1 format refers to the "Sound Designer I" format defined by Digidesign Inc in 1985 for multimedia and general audo applications. It is used for storing short monoaural sound samples. The SDES1 header is complex (1336 bytes) since it allows for associated display window information to be stored in it as well as providing facilities for specifying repeat loops. The HTK input routine for this format just picks out the following information

`headerSize` – size of header ie 1336 (2 byte integer)
(182 byte filler)
`fileSize` – number of bytes of sampled data (4 byte integer)
(832 byte filler)
`sampRate` – sample rate in Hz (4 byte integer)
`sampPeriod` – sample period in microseconds (4 byte integer)
`sampSize` – number of bits per sample ie 16 (2 byte integer)

5.11.7 AIFF File Format

The AIFF format was defined by Apple Computer for storing monoaural and multichannel sampled sounds. An AIFF file consists of a number of *chunks*. A *Common* chunk contains the fundamental parameters of the sound (sample rate, number of channels, etc) and a *Sound Data* chunk contains sampled audio data. HTK only partially supports AIFF since some of the information in it is stored as floating point numbers. In particular, the sample rate is stored in this form and to avoid portability problems, HTK ignores the given sample rate and assumes that it is 16kHz. If this default rate is incorrect, then the true sample period should be specified by setting the `SOURCERATE` configuration parameter. Full details of the AIFF format are available from Apple Developer Technical Support.

5.11.8 SUNAU8 File Format

The SUNAU8 format defines a subset of the ".au" and ".snd" audio file format used by Sun and NeXT. An SUNAU8 speech data file consists of a header followed by 8 bit μlaw encoded speech samples. The header is 28 bytes and contains the following fields, each of which is 4 bytes

`magicNumber` – magic number 0x2e736e64
`dataLocation` – offset to start of data
`dataSize` – number of bytes of data
`dataFormat` – data format code which is 1 for 8 bit μlaw
`sampRate` – a sample rate code which is always 8012.821 Hz
`numChan` – the number of channels
`info` – arbitrary character string min length 4 bytes

No default byte ordering is assumed for this format. If the data source is known to be different to the machine being used, then the environment variable `BYTEORDER` must be set appropriately. Note that when used on Sun Sparc machines with 16 bit audio device the sampling rate of 8012.821Hz is not supported and playback will be peformed at 8KHz.

5.11.9 OGI File Format

The OGI format is similar to TIMIT. The header contains the following

`hdrSize` – number of bytes in header
`version` – version number (2-byte integer)

numChannels	– number of channels (2-byte integer)
sampRate	– sample rate (2-byte integer)
nSamples	– number of samples in file (4-byte integer)
lendian	– used to test for byte swapping (4-byte integer)

5.11.10 WAV File Format

The WAV file format is a subset of Microsoft's RIFF specification for the storage of multimedia files. A RIFF file starts out with a file header followed by a sequence of data "chunks". A WAV file is often just a RIFF file with a single "WAVE" chunk which consists of two sub-chunks - a "fmt" chunk specifying the data format and a "data" chunk containing the actual sample data. The WAV file header contains the following

'RIFF'	– RIFF file identification (4 bytes)
<length>	– length field (4 bytes)
'WAVE'	– WAVE chunk identification (4 bytes)
'fmt '	– format sub-chunk identification (4 bytes)
flength	– length of format sub-chunk (4 byte integer)
format	– format specifier (2 byte integer)
chans	– number of channels (2 byte integer)
sampsRate	– sample rate in Hz (4 byte integer)
bpsec	– bytes per second (4 byte integer)
bpsample	– bytes per sample (2 byte integer)
bpchan	– bits per channel (2 byte integer)
'data'	– data sub-chunk identification (4 bytes)
dlength	– length of data sub-chunk (4 byte integer)

Support is provided for 8-bit CCITT mu-law, 8-bit CCITT a-law, 8-bit PCM linear and 16-bit PCM linear - all in stereo or mono (use of STEREOMODE parameter as per NIST). The default byte ordering assumed for WAV data files is VAX (little-endian).

5.11.11 ALIEN and NOHEAD File Formats

HTK tools can read speech waveform files with alien formats provided that their overall structure is that of a header followed by data. This is done by setting the format to ALIEN and setting the environment variable HEADERSIZE to the number of bytes in the header. HTK will then attempt to infer the rest of the information it needs. However, if input is from a pipe, then the number of samples expected must be set using the environment variable NSAMPLES. The sample rate of the source file is defined by the configuration parameter SOURCERATE as described in section 5.2. If the file has no header then the format NOHEAD may be specified instead of ALIEN in which case HEADERSIZE is assumed to be zero.

5.12 Direct Audio Input/Output

Many HTK tools, particularly recognition tools, can input speech waveform data directly from an audio device. The basic mechanism for doing this is to simply specify the SOURCEKIND as being HAUDIO following which speech samples will be read directly from the host computer's audio input device.

Note that for live audio input, the configuration variable ENORMALISE should be set to false both during training and recognition. Energy normalisation cannot be used with live audio input, and the default setting for this variable is TRUE. When training models for live audio input, be sure to set ENORMALISE to false. If you have existing models trained with ENORMALISE set to true, you can retrain them using *single-pass retraining* (see section 8.6).

When using direct audio input, the input sampling rate may be set explicitly using the configuration parameter SOURCERATE, otherwise HTK will assume that it has been set by some external means such as an audio control panel. In the latter case, it must be possible for HAudio to obtain the sample rate from the audio driver otherwise an error message will be generated.

Although the detailed control of audio hardware is typically machine dependent, HTK provides a number of Boolean configuration variables to request specific input and output sources. These are indicated by the following table

Variable	Source/Sink
LINEIN	line input
MICIN	microphone input
LINEOUT	line output
PHONESOUT	headphones output
SPEAKEROUT	speaker output

The major complication in using direct audio is in starting and stopping the input device. The simplest approach to this is for HTK tools to take direct control and, for example, enable the audio input for a fixed period determined via a command line option. However, the HAudio/HParm modules provides two more powerful built-in facilities for audio input control.

The first method of audio input control involves the use of an automatic energy-based speech/silence detector which is enabled by setting the configuration parameter USESILDET to true. Note that the speech/silence detector can also operate on waveform input files.

The automatic speech/silence detector uses a two level algorithm which first classifies each frame of data as either speech or silence and then applies a heuristic to determine the start and end of each utterance. The detector classifies each frame as speech or silence based solely on the log energy of the signal. When the energy value exceeds a threshold the frame is marked as speech otherwise as silence. The threshold is made up of two components both of which can be set by configuration variables. The first component represents the mean energy level of silence and can be set explicitly via the configuration parameter SILENERGY. However, it is more usual to take a measurement from the environment directly. Setting the configuration parameter MEASURESIL to true will cause the detector to calibrate its parameters from the current acoustic environment just prior to sampling. The second threshold component is the level above which frames are classified as speech (SPEECHTHRESH) . Once each frame has been classified as speech or silence they are grouped into windows consisting of SPCSEQCOUNT consecutive frames. When the number of frames marked as silence within each window falls below a glitch count the whole window is classed as speech. Two separate glitch counts are used, SPCGLCHCOUNT before speech onset is detected and SILGLCHCOUNT whilst searching for the end of the utterance. This allows the algorithm to take account of the tendancy for the end of an utterance to be somewhat quieter than the beginning. Finally, a top level heuristic is used to determine the start and end of the utterance. The heuristic defines the start of speech as the beginning of the first window classified as speech. The actual start of the processed utterance is SILMARGIN frames before the detected start of speech to ensure that when the speech detector triggers slightly late the recognition accuracy is not affected. Once the start of the utterance has been found the detector searches for SILSEQCOUNT windows all classified as silence and sets the end of speech to be the end of the last window classified as speech. Once again the processed utterance is extended SILMARGIN frames to ensure that if the silence detector has triggered slightly early the whole of the speech is still available for further processing.

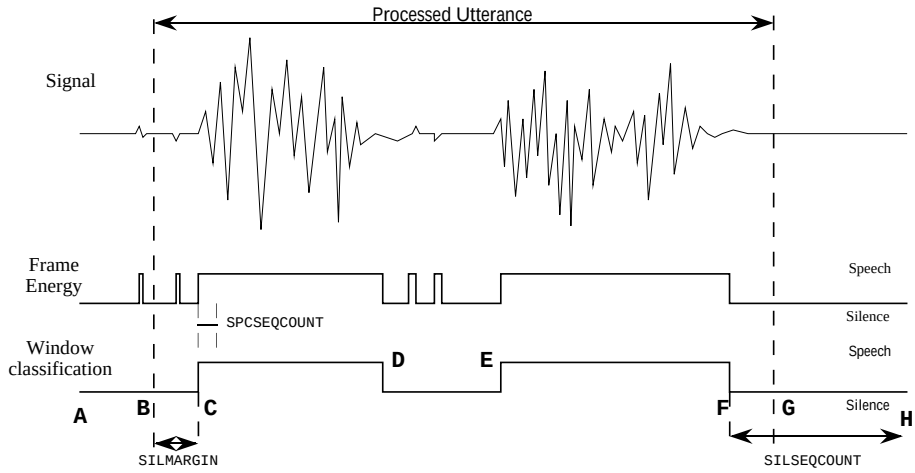


Fig. 5.6 Endpointer Parameters

Fig 5.6 shows an example of the speech/silence detection process. The waveform data is first

classified as speech or silence at frame and then at window level before finally the start and end of the utterance are marked. In the example, audio input starts at point A and is stopped automatically at point H. The start of speech, C, occurs when a window of SPCSEQCOUNT frames are classified as speech and the start of the utterance occurs SILMARGIN frames earlier at B. The period of silence from D to E is not marked as the end of the utterance because it is shorter than SILSEQCOUNT. However after point F no more windows are classified as speech (although a few frames are) and so this is marked as the end of speech with the end of the utterance extended to G.

The second built-in mechanism for controlling audio input is by arranging for a signal to be sent from some other process. Sending the signal for the first time starts the audio device. If the speech detector is not enabled then sampling starts immediately and is stopped by sending the signal a second time. If automatic speech/silence detection is enabled, then the first signal starts the detector. Sampling stops immediately when a second signal is received or when silence is detected. The signal number is set using the configuration parameter AUDIOSIG. Keypress control operates in a similar fashion and is enabled by setting the configuration parameter AUDIOSIG to a negative number. In this mode an initial keypress will be required to start sampling/speech detection and a second keypress will stop sampling immediately.

Audio output is also supported by HTK. There are no generic facilities for output and the precise behaviour will depend on the tool used. It should be noted, however, that the audio input facilities provided by HAudio include provision for attaching a *replay buffer* to an audio input channel. This is typically used to store the last few seconds of each input to a recognition tool in a circular buffer so that the last utterance input can be replayed on demand.

5.13 Multiple Input Streams

As noted in section 5.1, HTK tools regard the input observation sequence as being divided into a number of independent *data streams*. For building continuous density HMM systems, this facility is of limited use and by far the most common case is that of a single data stream. However, when building tied-mixture systems or when using vector quantisation, a more uniform coverage of the acoustic space is obtained by separating energy, deltas, etc., into separate streams.

This separation of parameter vectors into streams takes place at the point where the vectors are extracted from the converted input file or audio device and transformed into an observation. The tools for HMM construction and for recognition thus view the input data as a sequence of observations but note that this is entirely internal to HTK. Externally data is always stored as a single sequence of parameter vectors.

When multiple streams are required, the division of the parameter vectors is performed automatically based on the parameter kind. This works according to the following rules.

1 stream single parameter vector. This is the default case.

2 streams if the parameter vector contains energy terms, then they are extracted and placed in stream 2. Stream 1 contains the remaining static coefficients and their deltas and accelerations, if any. Otherwise, the parameter vector must have appended delta coefficients and no appended acceleration coefficients. The vector is then split so that the static coefficients form stream 1 and the corresponding delta coefficients form stream 2.

3 streams if the parameter vector has acceleration coefficients, then vector is split with static coefficients plus any energy in stream 1, delta coefficients plus any delta energy in stream 2 and acceleration coefficients plus any acceleration energy in stream 3. Otherwise, the parameter vector must include log energy and must have appended delta coefficients. The vector is then split into three parts so that the static coefficients form stream 1, the delta coefficients form stream 2, and the log energy and delta log energy are combined to form stream 3.

4 streams the parameter vector must include log energy and must have appended delta and acceleration coefficients. The vector is split into 4 parts so that the static coefficients form stream 1, the delta coefficients form stream 2, the acceleration coefficients form stream 3 and the log energy, delta energy and acceleration energy are combined to form stream 4.

In all cases, the static log energy can be suppressed (via the _N qualifier). If none of the above rules apply for some required number of streams, then the parameter vector is simply incompatible with that form of observation. For example, the parameter kind LPC_D_A cannot be split into 2 streams, instead 3 streams should be used.

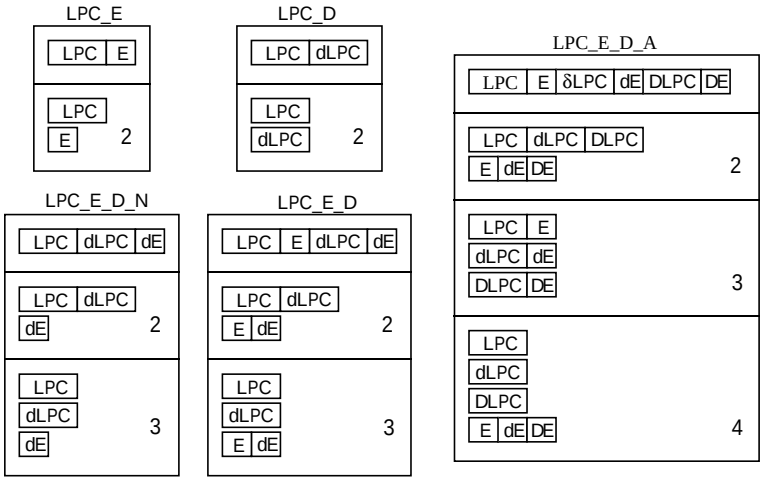


Fig. 5.7 Example Stream Construction

Fig. 5.7 illustrates the way that streams are constructed for a number of common cases. As earlier, the choice of `LPC` as the static coefficients is purely for illustration and the same mechanism applies to all base parameter kinds.

As discussed further in the next section, multiple data streams are often used with vector quantised data. In this case, each VQ symbol per input sample is placed in a separate data stream.

5.14 Vector Quantisation

Although HTK was designed primarily for building continuous density HMM systems, it also supports discrete density HMMs. Discrete HMMs are particularly useful for modelling data which is naturally symbolic. They can also be used with continuous signals such as speech by quantising each speech vector to give a unique VQ symbol for each input frame. The HTK module HVQ provides a basic facility for performing this vector quantisation. The VQ table (or codebook) can be constructed using the HTK tool HQuant.

When used with speech, the principle justification for using discrete HMMs is the much reduced computation. However, the use of vector quantisation introduces errors and it can lead to rather fragile systems. For this reason, the use of continuous density systems is generally preferred. To facilitate the use of continuous density systems when there are computational constraints, HTK also allows VQ to be used as the basis for pre-selecting a subset of Gaussian components for evaluation at each time frame.

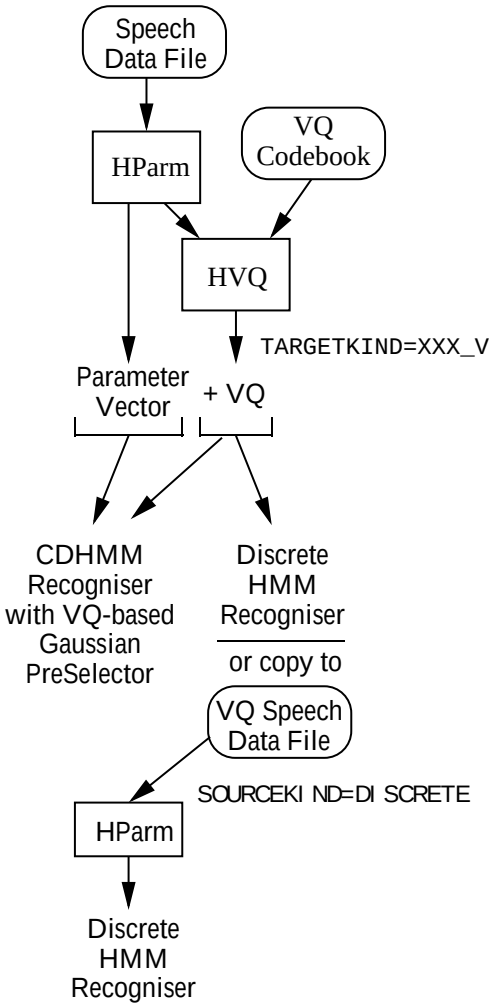


Fig. 5.8 Using Vector Quantisation

Fig. 5.8 illustrates the different ways that VQ can be used in HTK for a single data stream. For multiple streams, the same principles are applied to each stream individually. A converted speech waveform or file of parameter vectors can have VQ indices attached simply by specifying the name of a VQ table using the configuration parameter `VQTABLE` and by adding the `_V` qualifier to the target kind. The effect of this is that each *observation* passed to a recogniser can include both a conventional parameter vector and a VQ index. For continuous density HMM systems, a possible use of this might be to preselect Gaussians for evaluation (but note that HTK does not currently support this facility).

When used with a discrete HMM system, the continuous parameter vectors are ignored and only the VQ indices are used. For training and evaluating discrete HMMs, it is convenient to store speech data in vector quantised form. This is done using the tool HCopy to read in and vector quantise each speech file. Normally, HCopy copies the target form directly into the output file. However, if the configuration parameter `SAVEASVQ` is set, then it will store only the VQ indices and mark the kind of the newly created file as `DISCRETE`. Discrete files created in this way can be read directly by HParm and the VQ symbols passed directly to a tool as indicated by the lower part of Fig. 5.8.

HVQ supports three types of distance metric and two organisations of VQ codebook. Each codebook consists of a collection of nodes where each node has a mean vector and optionally a covariance matrix or diagonal variance vector. The corresponding distance metric used for each of these is simple Euclidean, full covariance Mahalanobis or diagonal covariance Mahalanobis. The codebook nodes are arranged in the form of a simple linear table or as a binary tree. In the linear case, the input vector is compared with every node in turn and the nearest determines the VQ

index. In the binary tree case, each non-terminal node has a left and a right daughter. Starting with the top-most root node, the input is compared with the left and right daughter node and the nearest is selected. This process is repeated until a terminal node is reached.

VQ Tables are stored externally in text files consisting of a header followed by a sequence of node entries. The header consists of the following information

magic	–	a magic number usually the original parameter kind
type	–	0 = linear tree, 1 = binary tree
mode	–	1 = diagonal covariance Mahalanobis 2 = full covariance Mahalanobis 5 = Euclidean
numNodes	–	total number of nodes in the codebook
numS	–	number of independent data streams
sw1,sw2,...	–	width of each data stream

Every node has a unique integer identifier and consists of the following

stream	–	stream number for this node
vqidx	–	VQ index for this node (0 if non-terminal)
nodeId	–	integer id of this node
leftId	–	integer id of left daughter node
rightId	–	integer id of right daughter node
mean	–	mean vector
cov	–	diagonal variance or full covariance

The inclusion of the optional variance vector or covariance matrix depends on the mode in the header. If present they are stored in inverse form. In a binary tree, the root id is always 1. In linear codebooks, the left and right daughter node id's are ignored.

5.15 Viewing Speech with HList

As mentioned in section 5.1, the tool HLIST provides a dual rôle in HTK. Firstly, it can be used for examining the contents of speech data files. In general, HLIST displays three types of information

1. *source header*: requested using the `-h` option
2. *target header*: requested using the `-t` option
3. *target data*: printed by default. The begin and end samples of the displayed data can be specified using the `-s` and `-e` options.

When the default configuration parameters are used, no conversions are applied and the target data is identical to the contents of the file.

As an example, suppose that the file called `timit.wav` holds speech waveform data using the TIMIT format. The command

```
HList -h -e 49 -F TIMIT timit.wav
```

would display the source header information and the first 50 samples of the file. The output would look something like the following

```
------------------------------ Source: timit.wav -------------------------------
  Sample Bytes:  2        Sample Kind:   WAVEFORM
  Num Comps:     1        Sample Period: 62.5 us
  Num Samples:   31437    File Format:   TIMIT
------------------------------ Samples: 0->49 ----------------------------------
    0:     8     -4     -1      0     -2     -1     -3     -2      0      0
   10:    -1      0     -1     -2     -1      1      0     -1     -2      1
   20:    -2      0      0      0      2      1     -2      2      1      0
   30:     1      0      0     -1      4      2      0     -1      4      0
   40:     2      2      1     -1     -1      1      1      2      1      1
---------------------------------- END -----------------------------------
```

The source information confirms that the file contains `WAVEFORM` data with 2 byte samples and 31437 samples in total. The sample period is 62.5μs which corresponds to a 16kHz sample rate. The displayed data is numerically small because it corresponds to leading silence. Any part of the file could be viewed by suitable choice of the begin and end sample indices. For example,

```
HList -s 5000 -e 5049 -F TIMIT timit.wav
```

would display samples 5000 through to 5049. The output might look like the following

```
---------------------------- Samples: 5000->5049 ----------------------------
 5000:    85   -116   -159   -252     23     99     69     92     79   -166
 5010:  -100   -123   -111     48    -19     15    111     41   -126   -304
 5020:  -189     91    162    255     80   -134   -174    -55     57    155
 5030:    90     -1     33    154     68   -149    -70     91    165    240
 5040:   297     50     13     72    187    189    193    244    198    128
-------------------------------------- END --------------------------------
```

The second use of HLIST is to check that input conversions are being performed properly. Suppose that the above TIMIT format file is part of a database to be used for training a recogniser and that mel-frequency cepstra are to be used along with energy and the first differential coefficients. Suitable configuration parameters needed to achieve this might be as follows

```
# Wave -> MFCC config file
SOURCEFORMAT = TIMIT    # same as -F TIMIT
TARGETKIND   = MFCC_E_D # MFCC + Energy + Deltas
TARGETRATE   = 100000   # 10ms frame rate
WINDOWSIZE   = 200000   # 20ms window
NUMCHANS     = 24       # num filterbank chans
NUMCEPS      = 8        # compute c1 to c8
```

HLIST can be used to check this. For example, typing

```
HList -C config -o -h -t -s 100 -e 104 -i 9  timit.wav
```

will cause the waveform file to be converted, then the source header, the target header and parameter vectors 100 through to 104 to be listed. A typical output would be as follows

```
------------------------------ Source: timit.wav ----------------------------
  Sample Bytes:  2        Sample Kind:   WAVEFORM
  Num Comps:     1        Sample Period: 62.5 us
  Num Samples:   31437    File Format:   TIMIT
---------------------------------- Target -------------------------------
  Sample Bytes:  72       Sample Kind:   MFCC_E_D
  Num Comps:     18       Sample Period: 10000.0 us
  Num Samples:   195      File Format:   HTK
--------------------------- Observation Structure --------------------------
x:    MFCC-1  MFCC-2  MFCC-3  MFCC-4  MFCC-5  MFCC-6  MFCC-7  MFCC-8       E
       Del-1   Del-2   Del-3   Del-4   Del-5   Del-6   Del-7   Del-8    DelE
------------------------------ Samples: 100->104 -------------------------
100:   3.573 -19.729  -1.256  -6.646  -8.293 -15.601 -23.404  10.988   0.834
       3.161  -1.913   0.573  -0.069  -4.935   2.309  -5.336   2.460   0.080
101:   3.372 -16.278  -4.683  -3.600 -11.030  -8.481 -21.210  10.472   0.777
       0.608  -1.850  -0.903  -0.665  -2.603  -0.194  -2.331   2.180   0.069
102:   2.823 -15.624  -5.367  -4.450 -12.045 -15.939 -22.082  14.794   0.830
      -0.051   0.633  -0.881  -0.067  -1.281  -0.410   1.312   1.021   0.005
103:   3.752 -17.135  -5.656  -6.114 -12.336 -15.115 -17.091  11.640   0.825
      -0.002  -0.204   0.015  -0.525  -1.237  -1.039   1.515   1.007   0.015
104:   3.127 -16.135  -5.176  -5.727 -14.044 -14.333 -18.905  15.506   0.833
      -0.034  -0.247   0.103  -0.223  -1.575   0.513   1.507   0.754   0.006
-------------------------------------- END --------------------------------
```

The target header information shows that the converted data consists of 195 parameter vectors, each vector having 18 components and being 72 bytes in size. The structure of each parameter vector

is displayed as a simple sequence of floating-point numbers. The layout information described in section 5.10 can be used to interpret the data. However, including the `-o` option, as in the example, causes HList to output a schematic of the observation structure. Thus, it can be seen that the first row of each sample contains the static coefficients and the second contains the delta coefficients. The energy is in the final column. The command line option `-i 9` controls the number of values displayed per line and can be used to aid in the visual interpretation of the data. Notice finally that the command line option `-F TIMIT` was not required in this case because the source format was specified in the configuration file.

It should be stressed that when HList displays parameterised data, it does so in exactly the form that *observations* are passed to a HTK tool. So, for example, if the above data was input to a system built using 3 data streams, then this can be simulated by using the command line option `-n` to set the number of streams. For example, typing

```
HList -C config -n 3 -o -s 100 -e 101 -i 9  timit.wav
```

would result in the following output

```
------------------------ Observation Structure ------------------------
nTotal=18 nStatic=8 nDel=16  eSep=T
x.1:    MFCC-1  MFCC-2  MFCC-3  MFCC-4  MFCC-5  MFCC-6  MFCC-7  MFCC-8
x.2:     Del-1   Del-2   Del-3   Del-4   Del-5   Del-6   Del-7   Del-8
x.3:         E    DelE
-------------------------- Samples: 100->101 -------------------------
100.1:   3.573 -19.729  -1.256  -6.646  -8.293 -15.601 -23.404  10.988
100.2:   3.161  -1.913   0.573  -0.069  -4.935   2.309  -5.336   2.460
100.3:   0.834   0.080
101.1:   3.372 -16.278  -4.683  -3.600 -11.030  -8.481 -21.210  10.472
101.2:   0.608  -1.850  -0.903  -0.665  -2.603  -0.194  -2.331   2.180
101.3:   0.777   0.069
--------------------------------- END ---------------------------------
```

Notice that the data is identical to the previous case, but it has been re-organised into separate streams.

5.16 Copying and Coding using HCopy

HCopy is a general-purpose tool for copying and manipulating speech files. The general form of invocation is

```
HCopy src tgt
```

which will make a new copy called `tgt` of the file called `src`. HCopy can also concatenate several sources together as in

```
HCopy src1 + src2 + src3 tgt
```

which concatenates the contents of `src1`, `src2` and `src3`, storing the results in the file `tgt`. As well as putting speech files together, HCopy can also take them apart. For example,

```
HCopy -s 100 -e -100 src tgt
```

will extract samples 100 through to N-100 of the file `src` to the file `tgt` where N is the total number of samples in the source file. The range of samples to be copied can also be specified with reference to a label file, and modifications made to the speech file can be tracked in a copy of the label file. All of the various options provided by HCopy are given in the reference section and in total they provide a powerful facility for manipulating speech data files.

However, the use of HCopy extends beyond that of copying, chopping and concatenating files. HCopy reads in all files using the speech input/output subsystem described in the preceding sections. Hence, by specifying an appropriate configuration file, HCopy is also a speech coding tool. For example, if the configuration file `config` was set-up to convert waveform data to MFCC coefficients, the command

```
HCopy -C config -s 100 -e -100 src.wav tgt.mfc
```

would parameterise the file waveform file `src.wav`, excluding the first and last 100 samples, and store the result in `tgt.mfc`.

HCopy will process its arguments in pairs, and as with all HTK tools, argument lists can be written in a script file specified via the `-S` option. When coding a large database, the separate invocation of HCopy for each file needing to be processed would incur a very large overhead. Hence, it is better to create a file, `flist` say, containing a list of all source and target files, as in for example,

```
src1.wav tgt1.mfc
src2.wav tgt2.mfc
src3.wav tgt3.mfc
src4.wav tgt4.mfc
etc
```

and then invoke HCopy by

```
HCopy -C config -s 100 -e -100 -S flist
```

which would encode each file listed in `flist` in a single invocation.

Normally HCopy makes a direct copy of the target speech data in the output file. However, if the configuration parameter `SAVECOMPRESSED` is set true then the output is saved in compressed form and if the configuration parameter `SAVEWITHCRC` is set true then a checksum is appended to the output (see section 5.10). If the configuration parameter `SAVEASVQ` is set true then only VQ indices are saved and the kind of the target file is changed to `DISCRETE`. For this to work, the target kind must have the qualifier `_V` attached (see section 5.14).

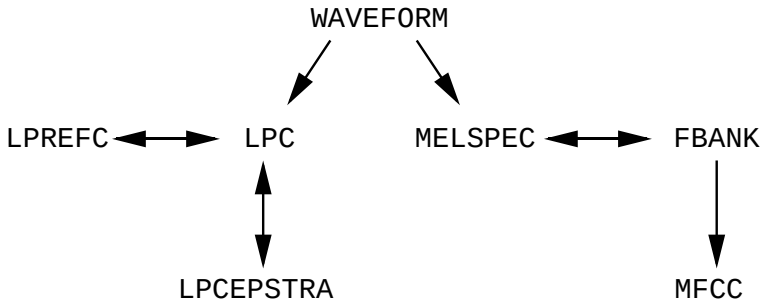


Fig. 5.9 Valid Parameter Kind Conversions

5.17 Version 1.5 Compatibility

The redesign of the HTK front-end in version 2 has introduced a number of differences in parameter encoding. The main changes are

1. Source waveform zero mean processing is now performed on a frame-by-frame basis.
2. Delta coefficients use a modified form of regression rather than simple differences at the start and end of the utterance.
3. Energy scaling is no longer applied to the zero'th MFCC coefficient.

If a parameter encoding is required which is as close as possible to the version 1.5 encoding, then the compatibility configuration variable `V1COMPAT` should be set to true.

Note also in this context that the default values for the various configuration values have been chosen to be consistent with the defaults or recommended practice for version 1.5.

5.18 Summary

This section summarises the various file formats, parameter kinds, qualifiers and configuration parameters used by HTK. Table 5.1 lists the audio speech file formats which can be read by the HWAVE module. Table 5.2 lists the basic parameter kinds supported by the HPARM module and Fig. 5.9 shows the various automatic conversions that can be performed by appropriate choice of source and target parameter kinds. Table 5.3 lists the available qualifiers for parameter kinds. The first 6 of these are used to describe the target kind. The source kind may already have some of these, HPARM adds the rest as needed. Note that HPARM can also delete qualifiers when converting from source to target. The final two qualifiers in Table 5.3 are only used in external files to indicate compression and an attached checksum. HPARM adds these qualifiers to the target form during output and only in response to setting the configuration parameters `SAVECOMPRESSED` and `SAVEWITHCRC`. Adding the `_C` or `_K` qualifiers to the target kind simply causes an error. Finally, Tables 5.4 and 5.5 lists all of the configuration parameters along with their meaning and default values.

Name	Description
`HTK`	The standard HTK file format
`TIMIT`	As used in the original prototype TIMIT CD-ROM
`NIST`	The standard SPHERE format used by the US NIST
`SCRIBE`	Subset of the European SAM standard used in the SCRIBE CD-ROM
`SDES1`	The Sound Designer 1 format defined by Digidesign Inc.
`AIFF`	Audio interchange file format
`SUNAU8`	Subset of 8bit ".au" and ".snd" formats used by Sun and NeXT
`OGI`	Format used by Oregan Graduate Institute similar to TIMIT
`WAV`	Microsoft WAVE files used on PCs
`ESIG`	Entropic Esignal file format
`AUDIO`	Pseudo format to indicate direct audio input
`ALIEN`	Pseudo format to indicate unsupported file, the alien header size must be set via the environment variable `HDSIZE`
`NOHEAD`	As for the ALIEN format but header size is zero

Table. 5.1 Supported File Formats

Kind	Meaning
`WAVEFORM`	scalar samples (usually raw speech data)
`LPC`	linear prediction coefficients
`LPREFC`	linear prediction reflection coefficients
`LPCEPSTRA`	LP derived cepstral coefficients
`LPDELCEP`	LP cepstra + delta coef (obsolete)
`IREFC`	LPREFC stored as 16bit (short) integers
`MFCC`	mel-frequency cepstral coefficients
`FBANK`	log filter-bank parameters
`MELSPEC`	linear filter-bank parameters
`USER`	user defined parameters
`DISCRETE`	vector quantised codebook symbols
`ANON`	matches actual parameter kind

Table. 5.2 Supported Parameter Kinds

Qualifier	Meaning
_A	Acceleration coefficients appended
_C	External form is compressed
_D	Delta coefficients appended
_E	Log energy appended
_K	External form has checksum appended
_N	Absolute log energy suppressed
_T	Third differential coefficients appended
_V	VQ index appended
_Z	Cepstral mean subtracted
_0	Cepstral C0 coefficient appended

Table. 5.3 Parameter Kind Qualifiers

Module	Name	Default	Description
HAUDIO	LINEIN	T	Select line input for audio
HAUDIO	MICIN	F	Select microphone input for audio
HAUDIO	LINEOUT	T	Select line output for audio
HAUDIO	SPEAKEROUT	F	Select speaker output for audio
HAUDIO	PHONESOUT	T	Select headphones output for audio
	SOURCEKIND	ANON	Parameter kind of source
	SOURCEFORMAT	HTK	File format of source
	SOURCERATE	0.0	Sample period of source in 100ns units
HWAVE	NSAMPLES		Num samples in alien file input via a pipe
HWAVE	HEADERSIZE		Size of header in an alien file
HWAVE	STEREOMODE		Select channel: RIGHT or LEFT
HWAVE	BYTEORDER		Define byte order VAX or other
	NATURALREADORDER	F	Enable natural read order for HTK files
	NATURALWRITEORDER	F	Enable natural write order for HTK files
	TARGETKIND	ANON	Parameter kind of target
	TARGETFORMAT	HTK	File format of target
	TARGETRATE	0.0	Sample period of target in 100ns units
HPARM	SAVECOMPRESSED	F	Save the output file in compressed form
HPARM	SAVEWITHCRC	T	Attach a checksum to output parameter file
HPARM	ADDDITHER	0.0	Level of noise added to input signal
HPARM	ZMEANSOURCE	F	Zero mean source waveform before analysis
HPARM	WINDOWSIZE	256000.0	Analysis window size in 100ns units
HPARM	USEHAMMING	T	Use a Hamming window
HPARM	PREEMCOEF	0.97	Set pre-emphasis coefficient
HPARM	LPCORDER	12	Order of LPC analysis
HPARM	NUMCHANS	20	Number of filterbank channels
HPARM	LOFREQ	-1.0	Low frequency cut-off in fbank analysis
HPARM	HIFREQ	-1.0	High frequency cut-off in fbank analysis
HPARM	USEPOWER	F	Use power not magnitude in fbank analysis
HPARM	NUMCEPS	12	Number of cepstral parameters
HPARM	CEPLIFTER	22	Cepstral liftering coefficient
HPARM	ENORMALISE	T	Normalise log energy
HPARM	ESCALE	0.1	Scale log energy
HPARM	SILFLOOR	50.0	Energy silence floor (dB)
HPARM	DELTAWINDOW	2	Delta window size
HPARM	ACCWINDOW	2	Acceleration window size
HPARM	VQTABLE	NULL	Name of VQ table
HPARM	SAVEASVQ	F	Save only the VQ indices
HPARM	AUDIOSIG	0	Audio signal number for remote control

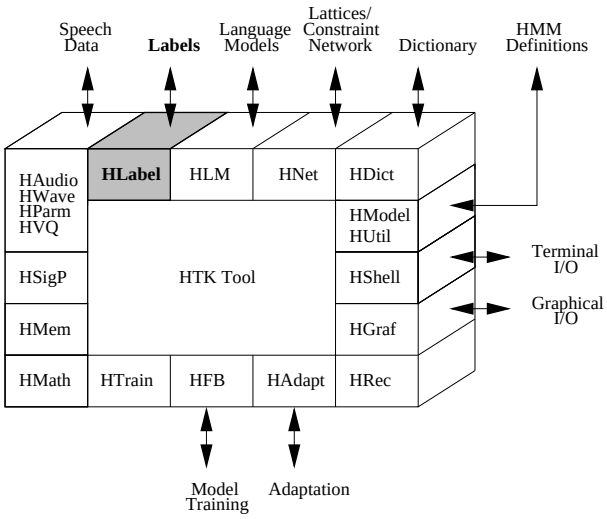
Table. 5.4 Configuration Parameters

Module	Name	Default	Description
HPARM	USESILDET	F	Enable speech/silence detector
HPARM	MEASURESIL	T	Measure background noise level prior to sampling
HPARM	OUTSILWARN	T	Print a warning message to `stdout` before measuring audio levels
HPARM	SPEECHTHRESH	9.0	Threshold for speech above silence level (dB)
HPARM	SILENERGY	0.0	Average background noise level (dB)
HPARM	SPCSEQCOUNT	10	Window over which speech/silence decision reached
HPARM	SPCGLCHCOUNT	0	Maximum number of frames marked as silence in window which is classified as speech whilst expecting start of speech
HPARM	SILSEQCOUNT	100	Number of frames classified as silence needed to mark end of utterance
HPARM	SILGLCHCOUNT	2	Maximum number of frames marked as silence in window which is classified as speech whilst expecting silence
HPARM	SILMARGIN	40	Number of extra frames included before and after start and end of speech marks from the speech/silence detector
HPARM	V1COMPAT	F	Set Version 1.5 compatibility mode
	TRACE	0	Trace setting

Table. 5.5 Configuration Parameters (cont)

Chapter 6

Transcriptions and Label Files



Many of the operations performed by HTK which involve speech data files assume that the speech is divided into segments and each segment has a name or *label*. The set of labels associated with a speech file constitute a *transcription* and each transcription is stored in a separate *label file*. Typically, the name of the label file will be the same as the corresponding speech file but with a different extension. For convenience, label files are often stored in a separate directory and all HTK tools have an option to specify this. When very large numbers of files are being processing, label file access can be greatly facilitated by using *Master Label Files (MLFs)*. MLFs may be regarded as index files holding pointers to the actual label files which can either be embedded in the same index file or stored anywhere else in the file system. Thus, MLFs allow large sets of files to be stored in a single file, they allow a single transcription to be shared by many logical label files and they allow arbitrary file redirection.

The HTK interface to label files is provided by the module HLABEL which implements the MLF facility and support for a number of external label file formats. All of the facilities supplied by HLABEL, including the supported label file formats, are described in this chapter. In addition, HTK provides a tool called HLED for simple batch editing of label files and this is also described. Before proceeding to the details, however, the general structure of label files will be reviewed.

6.1 Label File Structure

Most transcriptions are single-alternative and single-level, that is to say, the associated speech file is described by a single sequence of labelled segments. Most standard label formats are of this kind. Sometimes, however, it is useful to have several levels of labels associated with the same basic segment sequence. For example, in training a HMM system it is useful to have both the word level transcriptions and the phone level transcriptions *side-by-side*.

Orthogonal to the requirement for multiple levels of description, a transcription may also need to include multiple alternative descriptions of the same speech file. For example, the output of a speech recogniser may be in the form of an *N-best* list where each word sequence in the list represents one possible interpretation of the input.

As an example, Fig. 6.1 shows a speech file and three different ways in which it might be labelled. In part (a), just a simple orthography is given and this single-level single-alternative type of transcription is the commonest case. Part (b) shows a 2-level transcription where the basic level consists of a sequence of phones but a higher level of word labels are also provided. Notice that there is a distinction between the basic level and the higher levels, since only the basic level has explicit boundary locations marked for every segment. The higher levels do not have explicit boundary information since this can always be inferred from the basic level boundaries. Finally, part (c) shows the case where knowledge of the contents of the speech file is uncertain and three possible word sequences are given.

HTK label files support multiple-alternative and multiple-level transcriptions. In addition to start and end times on the basic level, a label at any level may also have a score associated with it. When a transcription is loaded, all but one specific alternative can be discarded by setting the configuration variable `TRANSALT` to the required alternative `N`, where the first (i.e. normal) alternative is numbered 1. Similarly, all but a specified level can be discarded by setting the configuration variable `TRANSLEV` to the required level number where again the first (i.e. normal) level is numbered 1.

All non-HTK formats are limited to single-level single-alternative transcriptions.

6.2 Label File Formats

As with speech data files, HTK not only defines its own format for label files but also supports a number of external formats. Defining an external format is similar to the case for speech data files except that the relevant configuration variables for specifying a format other than HTK are called `SOURCELABEL` and `TARGETLABEL`. The source label format can also be specified using the `-G` command line option. As with using the `-F` command line option for speech data files, the `-G` option overrides any setting of `SOURCELABEL`

6.2.1 HTK Label Files

The HTK label format is text based. As noted above, a single label file can contain multiple-alternatives and multiple-levels.

Each line of a HTK label file contains the actual label optionally preceded by start and end times, and optionally followed by a match score.

```
[start  [end] ] name [score] { auxname [auxscore] } [comment]
```

where `start` denotes the start time of the labelled segment in 100ns units, `end` denotes the end time in 100ns units, `name` is the name of the segment and `score` is a floating point confidence score. All fields except the name are optional. If `end` is omitted then it is set equal to -1 and ignored. This case would occur with data which had been labelled frame synchronously. If `start` and `end` are both missing then both are set to -1 and the label file is treated as a simple symbolic transcription. The optional score would typically be a log probability generated by a recognition tool. When omitted the score is set to 0.0.

The following example corresponds to the transcription shown in part (a) of Fig. 6.1

```
0000000 3600000 ice
3600000 8200000 cream
```

Multiple levels are described by adding further names alongside the basic name. The lowest level (shortest segments) should be given first since only the lowest level has start and end times. The label file corresponding to the transcription illustrated in part (b) of Fig. 6.1 would be as follows.

```
0000000 2200000 ay      ice
2200000 3600000 s
3600000 4300000 k       cream
4300000 5000000 r
5000000 7400000 iy
7400000 8200000 m
```

Finally, multiple alternatives are written as a sequence of separate label lists separated by three slashes (///). The label file corresponding to the transcription illustrated in part (c) of Fig. 6.1 would therefore be as follows.

```
0000000 2200000 I
2200000 8200000 scream
///
0000000 3600000 ice
3600000 8200000 cream
///
0000000 3600000 eyes
3600000 8200000 cream
```

Actual label names can be any sequence of characters. However, the `-` and `+` characters are reserved for identifying the left and right context, respectively, in a context-dependent phone label. For example, the label `N-aa+V` might be used to denote the phone `aa` when preceded by a nasal and followed by a vowel. These context-dependency conventions are used in the label editor HLEd, and are understood by all HTK tools.

6.2.2 ESPS Label Files

An *ESPS/waves+* label file is a text file with one label stored per line. Each label indicates a segment boundary. A complete description of the *ESPS/waves+* label format is given in the *ESPS/waves+* manual pages **xwaves (1-ESPS)** and **xlabel (1-ESPS)**. Only details required for use with HTK are given here.

The label data follows a header which ends with a line containing only a `#`. The header contents are generally ignored by HLabel. The labels follow the header in the form

```
time  ccode  name
```

where `time` is a floating point number which denotes the boundary location in seconds, `ccode` is an integer color map entry used by *ESPS/waves+* in drawing segment boundaries and `name` is the name of the segment boundary. A typical value for `ccode` is `121`.

While each HTK label can contain both a start and an end time which indicate the boundaries of a labeled segment, *ESPS/waves+* labels contain a single time in seconds which (by convention) refers to the end of the labeled segment. The starting time of the segment is taken to be the end of the previous segment and `0` initially.

ESPS/waves+ label files may have several boundary names per line. However, HLabel only reads *ESPS/waves+* label files with a single name per boundary. Multiple-alternative and/or multiple-level HTK label data structures cannot be saved using *ESPS/waves+* format label files.

6.2.3 TIMIT Label Files

TIMIT label files are identical to single-alternative single-level HTK label files without scores except that the start and end times are given as sample numbers rather than absolute times. TIMIT label files are used on both the prototype and final versions of the TIMIT CD ROM.

6.2.4 SCRIBE Label Files

The SCRIBE label file format is a subset of the European SAM label file format. SAM label files are text files and each line begins with a label identifying the type of information stored on that line. The HTK SCRIBE format recognises just three label types

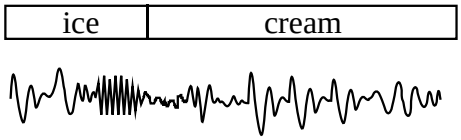
LBA	– acoustic label
LBB	– broad class label
UTS	– utterance

For each of these, the rest of the line is divided into comma separated fields. The LBA and LBB types have 4 fields: start sample, centre sample, end sample and label. HTK expects the centre sample to be blank. The UTS type has 3 fields: start sample, end sample and label. UTS labels may be multi-word since they can refer to a complete utterance. In order to make such labels usable within HTK tools, between word blanks are converted to underscore characters. The `EX` command in the HTK label editor HLEd can then be used to split such a compound label into individual word labels if required.

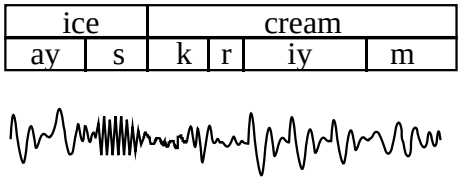
6.3 Master Label Files

6.3.1 General Principles of MLFs

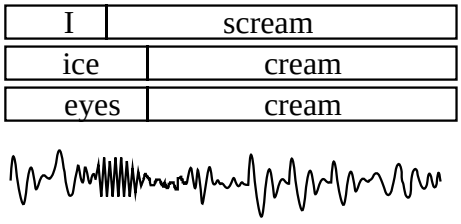
Logically, the organisation of data and label files is very simple. Every data file has a label file of the same name (but different extension) which is either stored in the same directory as the data file or in some other specified directory.



(a) 1-alternative, 1-level



(b) 1-alternative, 2-level



(c) 3-alternative, 1-level

Fig. 6.1 Example Transcriptions

This scheme is sufficient for most needs and commendably simple. However, there are many cases where either it makes unnecessarily inefficient use of the operating system or it seriously inconveniences the user. For example, to use a training tool with isolated word data may require the generation of hundreds or thousands of label files each having just one label entry. Even where individual label files are appropriate (as in the phonetically transcribed TIMIT database), each label file must be stored in the same directory as the data file it transcribes, or all label files must be stored in the same directory. One cannot, for example, have a different directory of label files for each TIMIT dialect region and then run the HTK training tool HEREST on the whole database.

All of these problems can be solved by the use of Master Label Files (MLFs). Every HTK tool which uses label files has a `-I` option which can be used to specify the name of an MLF file. When an MLF has been loaded, the normal rules for locating a label file apply except that the MLF is searched first. If the required label file `f` is found via the MLF then that is loaded, otherwise the file `f` is opened as normal. If `f` does not exist, then an error is reported. The `-I` option may be repeated on the command line to open several MLF files simultaneously. In this case, each is searched in turn before trying to open the required file.

MLFs can do two things. Firstly, they can contain embedded label definitions so that many or all of the needed label definitions can be stored in the same file. Secondly, they can contain the names of sub-directories to search for label files. In effect, they allow multiple search paths to be defined. Both of these two types of definition can be mixed in a single MLF.

MLFs are quite complex to understand and use. However, they add considerable power and flexibility to HTK which combined with the `-S` and `-L` options mean that virtually any organisation of data and label files can be accommodated.

6.3.2 Syntax and Semantics

An MLF consists of one or more individual definitions. Blank lines in an MLF are ignored but otherwise the line structure is significant. The first line must contain just `#!MLF!#` to identify it as an MLF file. This is not necessary for use with the `-I` option but some HTK tools need to be able to distinguish an MLF from a normal label file. The following syntax of MLF files is described using an extended BNF notation in which alternatives are separated by a vertical bar |, parentheses ( ) denote factoring, brackets [ ] denote options, and braces { } denote zero or more repetitions.

MLF = "#!MLF!#"
MLFDef { MLFDef }

Each definition is either a transcription for immediate loading or a subdirectory to search.

MLFDef = ImmediateTranscription | SubDirDef

An immediate transcription consists of a pattern on a line by itself immediately followed by a transcription which as far as the MLF is concerned is arbitrary text. It is read using whatever label file "driver" routines are installed in HLabel. It is terminated by a period written on a line of its own.

ImmediateTranscription =
Pattern
Transcription
"."

A subdirectory definition simply gives the name of a subdirectory to search. If the required label file is found in that subdirectory then the label file is loaded, otherwise the next matching subdirectory definition is checked.

SubDirDef = Pattern SearchMode String
SearchMode = "->" | "=>"

The two types of search mode are described below. A pattern is just a string

Pattern = String

except that the characters '`?`' and '`*`' embedded in the string act as wildcards such that '`?`' matches any single character and '`*`' matches 0 or more characters. A string is any sequence of characters enclosed in double quotes.

6.3.3 MLF Search

The names of label files in HTK are invariably reconstructed from an existing data file name and this means that the file names used to access label files can be partial or full path names in which the path has been constructed either from the path of the corresponding data file or by direct specification via the `-L` option. These path names are retained in the MLF search which proceeds as follows. The given label file specification `../d3/d2/d1/name` is matched against each pattern in the MLF. If a pattern matches, then either the named subdirectory is searched or an immediate definition is loaded. Pattern matching continues in this way until a definition is found. If no pattern matches then an attempt is made to open `../d3/d2/d1/name` directly. If this fails an error is reported.

The search of a sub-directory proceeds as follows. In simple search mode indicated by `->`, the file `name` must occur directly in the sub-directory. In full search mode indicated by `=>`, the files `name, d1/name, d2/d1/name`, etc. are searched for in that order. This full search allows a hierarchy of label files to be constructed which mirrors a hierarchy of data files (see Example 4 below).

Hashing is performed when the label file specification is either a full path name or in the form `*/file` so in these cases the search is very fast. Any other use of metacharacters invokes a linear

search with a full and relatively slow pattern match at each step. Note that all tools which generate label files have a `-l` option which is used to define the output directory in which to store individual label files. When outputting master label files, the `-l` option can be used to define the path in the output label file specifications. In particular, setting the option `-l '*'` causes the form `*/file` to be generated.

6.3.4 MLF Examples

1. Suppose a data set consisted of two training data files with corresponding label files:
   `a.lab` contains

   ```
    000000  590000  sil
    600000 2090000  a
   2100000 4500000  sil
   ```

   `b.lab` contains

   ```
    000000  990000  sil
   1000000 3090000  b
   3100000 4200000  sil
   ```

   Then the above two individual label files could be replaced by a single MLF

   ```
   #!MLF!#
   "*/a.lab"
    000000  590000  sil
    600000 2090000  a
   2100000 4500000  sil
   .
   "*/b.lab"
    000000  990000  sil
   1000000 3090000  b
   3100000 4200000  sil
   .
   ```

2. A digit data base contains training tokens `one.1.wav, one.2.wav, one.3.wav, ..., two.1.wav, two.2.wav, two.3.wav, ...`, etc. Label files are required containing just the name of the model so that HTK tools such as HERest can be used. If MLFs are not used, individual label files are needed. For example, the individual label files `one.1.lab, one.2.lab, one.3.lab, ....` would be needed to identifiy instances of "one" even though each file contains the same entry, just

   ```
   one
   ```

   Using an MLF containing

   ```
   #!MLF!#
   "*/one.*.lab"
   one
   .
   "*/two.*.lab"
   two
   .
   "*/three.*.lab"
   three
   .
   <etc.>
   ```

   avoids the need for many duplicate label files.

3. A training database `/db` contains directories `dr1, dr2, ..., dr8`. Each directory contains a subdirectory called `labs` holding the label files for the data files in that directory. The following MLF would allow them to be found

```
#!MLF!#
"*" -> "/db/dr1/labs"
"*" -> "/db/dr2/labs"
...
"*" -> "/db/dr7/labs"
"*" -> "/db/dr8/labs"
```

   Each attempt to open a label file will result in a linear search through `dr1` to `dr8` to find that file. If the sub-directory name is embedded into the label file name, then this searching can be avoided. For example, if the label files in directory `drx` had the form `drx_xxxx.lab`, then the MLF would be written as

```
#!MLF!#
"*/dr1_*" -> "/db/dr1/labs"
"*/dr2_*" -> "/db/dr2/labs"
...
"*/dr7_*" -> "/db/dr7/labs"
"*/dr8_*" -> "/db/dr8/labs"
```

4. A training database is organised as a hierarchy where `/disk1/db/dr1/sp2/u3.wav` is the data file for the third repetition from speaker 2 in dialect region `dr1` (see Figure 6.2).

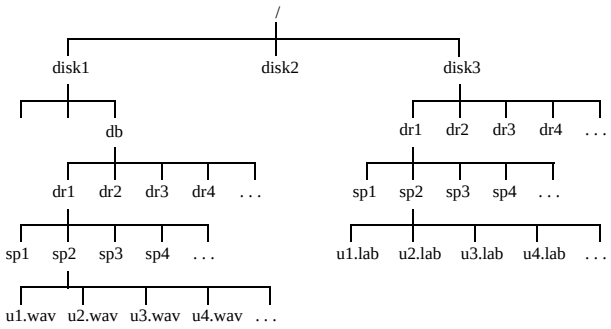


Fig. 6.2 Database Hierarchy: Data [Left]; Labels [Right].

Suppose that a similar hierarchy of label files was constructed on `disk3`. These label files could be found by any HTK tool by using an MLF containing just

```
#!MLF!#
"*" => "/disk3"
```

If for some reason all of the `drN` directories were renamed `ldrN` in the label hierarchy, then this could be handled by an MLF file containing

```
#!MLF!#
"*/dr1/*" => "/disk3/ldr1"
"*/dr2/*" => "/disk3/ldr2"
"*/dr3/*" => "/disk3/ldr3"
         etc.
```

These few examples should illustrate the flexibility and power of MLF files. It should noted, however, that when generating label names automatically from data file names, HTK sometimes discards path details. For example, during recognition, if the data files /disk1/dr2/sx43.wav and /disk2/dr4/sx43.wav are being recognised, and a single directory is specified for the output label files, then recognition results for both files will be written to a file called sx43.lab, and the latter occurrence will overwrite the former.

6.4 Editing Label Files

HTK training tools typically expect the labels used in transcription files to correspond directly to the names of the HMMs chosen to build an application. Hence, the label files supplied with a speech database will often need modifying. For example, the original transcriptions attached to a database might be at a fine level of acoustic detail. Groups of labels corresponding to a sequence of acoustic events (e.g. `pcl p'`) might need converting to some simpler form (e.g. `p`) which is more suitable for being represented by a HMM. As a second example, current high performance speech recognisers use a large number of context dependent models to allow more accurate acoustic modelling. For this case, the labels in the transcription must be converted to show the required contexts explicitly.

HTK supplies a tool called HLEd for rapidly and efficiently converting label files. The HLEd command invocation specifies the names of the files to be converted and the name of a script file holding the actual HLEd commands. For example, the command

```
HLEd edfile.led l1 l2 l3
```

would apply the edit commands stored in the file `edfile.led` to each of the label files `l1`, `l2` and `l3`. More commonly the new label files are stored in a new directory to avoid overwriting the originals. This is done by using the `-l` option. For example,

```
HLEd -l newlabs edfile.led l1 l2 l3
```

would have the same effect as previously except that the new label files would be stored in the directory `newlabs`.

Each edit command stored in an edit file is identified by a mnemonic consisting of two letters[1] and must be stored on a separate line. The supplied edit commands can be divided into two groups. The first group consist of commands which perform selective changes to specific labels and the second group contains commands which perform global transformations. The reference section defines all of these commands. Here a few examples will be given to illustrate the use of HLEd.

As a first example, when using the TIMIT database, the original 61 phoneme symbol set is often mapped into a simpler 48 phoneme symbol set. The aim of this mapping is to delete all glottal stops, replace all closures preceding a voiced stop by a generic voiced closure (`vcl`), all closures preceding an unvoiced stop by a generic unvoiced closure (`cl`) and the different types of silence to a single generic silence (`sil`). A HLEd script to do this might be

```
# Map 61 Phone Timit Set -> 48 Phones
SO
DE q
RE cl pcl tcl kcl qcl
RE vcl bcl dcl gcl
RE sil h# #h pau
```

The first line is a comment indicated by the initial hash character. The command on the second line is the *Sort* command `SO`. This is an example of a global command. Its effect is to sort all the labels into time order. Normally the labels in a transcription will already be in time order but some speech editors simply output labels in the order that the transcriber marked them. Since this would confuse the re-estimation tools, it is good practice to explicitly sort all label files in this way.

The command on the third line is the *Delete* command `DE`. This is a selective command. Its effect is to delete all of the labels listed on the rest of the command line, wherever they occur. In this case, there is just one label listed for deletion, the glottal stop `q`. Hence, the overall effect of this command will be to delete all occurrences of the `q` label in the edited label files.

The remaining commands in this example script are *Replace* commands `RE`. The effect of a Replace command is to substitute the first label following the `RE` for every occurrence of the remaining

[1] Some command names have single letter alternatives for compatibility with earlier versions of HTK.

labels on that line. Thus, for example, the command on the third line causes all occurrences of the labels `pcl`, `tcl`, `kcl` or `qcl` to be replaced by the label `cl`.

To illustrate the overall effect of the above HLEd command script on a complete label file, the following `TIMIT` format label file

```
 0000 2241 h#
 2241 2715 w
 2715 4360 ow
 4360 5478 bcl
 5478 5643 b
 5643 6360 iy
 6360 7269 tcl
 7269 8313 t
 8313 11400 ay
11400 12950 dcl
12950 14360 dh
14360 14640 h#
```

would be converted by the above script to the following

```
      0 1400625 sil
1400625 1696875 w
1696875 2725000 ow
2725000 3423750 vcl
3423750 3526875 b
3526875 3975000 iy
3975000 4543125 cl
4543125 5195625 t
5195625 7125000 ay
7125000 8093750 vcl
8093750 8975000 dh
8975000 9150000 sil
```

Notice that label boundaries in TIMIT format are given in terms of sample numbers (16kHz sample rate), whereas the edited output file is in HTK format in which all times are in absolute 100ns units.

As well as the Replace command, there is also a *Merge* command `ME`. This command is used to replace a sequence of labels by a single label. For example, the following commands would merge the closure and release labels in the previous TIMIT transcription into single labels

```
ME b bcl b
ME d dcl dh
ME t tcl t
```

As shown by this example, the label used for the merged sequence can be the same as occurs in the original but some care is needed since HLEd commands are normally applied in sequence. Thus, a command on line n is applied to the label sequence that remains after the commands on lines 1 to $n-1$ have been applied.

There is one exception to the above rule of sequential edit command application. The *Change* command `CH` provides for context sensitive replacement. However, when a sequence of Change commands occur in a script, the sequence is applied as a block so that the contexts which apply for each command are those that existed just prior to the block being executed. The Change command takes 4 arguments `X A Y B` such that every occurrence of label `Y` in the context of `A _ B` is changed to the label `X`. The contexts `A` and `B` refer to sets of labels and are defined by separate *Define Context* commands `DC`. The `CH` and `DC` commands are primarily used for creating context sensitive labels. For example, suppose that a set of context-dependent phoneme models are needed for TIMIT. Rather than treat all possible contexts separately and build separate triphones for each (see below), the possible contexts will be grouped into just 5 broad classes: C (consonant), V (vowel), N (nasal), L (liquid) and S (silence). The goal then is to translate a label sequence such as `sil b ah t iy n ...` into `sil+C S-b+V C-ah+C V-t+V C-iy+N V-n+ ...` where the `-` and `+` symbols within a label are recognised by HTK as defining the left and right context, respectively. To perform this transformation, it is necessary to firstly use `DC` commands to define the 5 contexts, that is

```
DC V iy ah ae eh ix ...
DC C t k d k g dh ...
DC L l r w j ...
DC N n m ng ...
DC S h# #h epi ...
```

Having defined the required contexts, a change command must be written for each context dependent triphone, that is

```
CH V-ah+V V ah V
CH V-ah+C V ah C
CH V-ah+N V ah N
CH V-ah+L V ah L
 ...
 etc
```

This script will, of course, be rather long (25 × number of phonemes) but it can easily be generated automaticaly by a simple program or shell script.

The previous example shows how to transform a set of phonemes into a context dependent set in which the contexts are user-defined. For convenience, HLEd provides a set of global transformation commands for converting phonemic transcriptions to conventional left or right biphones, or full triphones. For example, a script containing the single *Triphone Conversion* command `TC` will convert phoneme files to regular triphones. As an illustration, applying the `TC` command to a file containing the sequence `sil b ah t iy n ...` would give the transformed sequence `sil+b sil-b+ah b-ah+t ah-t+iy t-iy+n iy-n+ ....` Notice that the first and last phonemes in the sequence cannot be transformed in the normal way. Hence, the left-most and right-most contexts of these start and end phonemes can be specified explicitly as arguments to the `TC` commands if required. For example, the command `TC # #` would give the sequence `#-sil+b sil-b+ah b-ah+t ah-t+iy t-iy+n iy-n+ ... +#`. Also, the contexts at pauses and word boundaries can be blocked using the `WB` command. For example, if `WB sp` was executed, the effect of a subsequent `TC` command on the sequence `sil b ah t sp iy n ...` would be to give the sequence `sil+b sil-b+ah b-ah+t ah-t sp iy+n iy-n+ ...`, where `sp` represents a short pause. Conversely, the `NB` command can be used to ignore a label as far as context is concerned. For example, if `NB sp` was executed, the effect of a subsequent `TC` command on the sequence `sil b ah t sp iy n ...` would be to give the sequence `sil+b sil-b+ah b-ah+t ah-t+iy sp t-iy+n iy-n+ ....`

When processing HTK format label files with multiple levels, only the level 1 (i.e. left-most) labels are affected. To process a higher level, the *Move Level* command `ML` should be used. For example, in the script

```
ML 2
RE one 1
RE two 2
...
```

the Replace commands are applied to level 2 which is the first level above the basic level. The command `ML 1` returns to the base level. A complete level can be deleted by the *Delete Level* command `DL`. This command can be given a numeric argument to delete a specific level or with no argument, the current level is deleted. Multiple levels can also be split into single level alternatives by using the *Split Level* command `SL`.

When processing HTK format files with multiple alternatives, each alternative is processed as though it were a separate file.

Remember also that in addition to the explicit HLEd commands, levels and alternatives can be filtered on input by setting the configuration variables `TRANSLEV` and `TRANSALT` (see section 6.1).

Finally, it should be noted that most HTK tools require all HMMs used in a system to be defined in a *HMM List*. HLEd can be made to automatically generate such a list as a by-product of editing the label files by using the `-n` option. For example, the following command would apply the script `timit.led` to all files in the directory `tlabs`, write the converted files to the directory `hlabs` and also write out a list of all new labels in the edited files to `tlist`.

```
HLEd -n tlist -l hlabs -G TIMIT timit.led tlabs/*
```

Notice here that the `-G` option is used to inform HLEd that the format of the source files is `TIMIT`. This could also be indicated by setting the configuration variable `SOURCELABEL`.

6.5 Summary

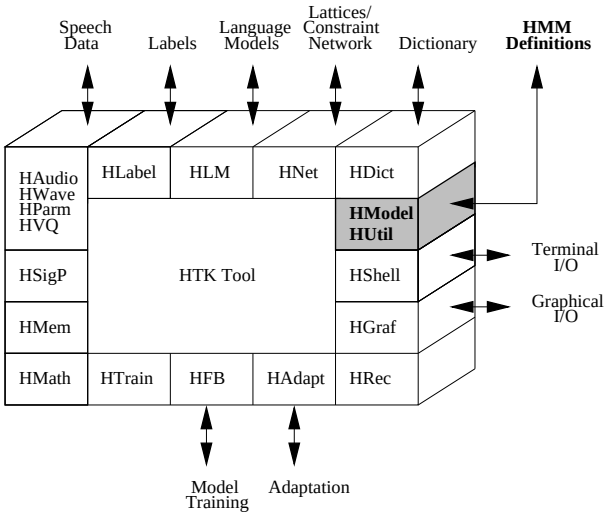
Table 6.1 lists all of the configuration parameters recognised by HLABEL along with a brief description. A missing module name means that it is recognised by more than one module.

Module	Name	Description
HLABEL	LABELSQUOTE	Specify label quote character
HLABEL	SOURCELABEL	Source label format
HLABEL	SOURCERATE	Sample period for SCRIBE format
HLABEL	STRIPTRIPHONES	Remove triphone contexts on input
HLABEL	TARGETLABEL	Target label format
HLABEL	TRANSALT	Filter alternatives on input
HLABEL	TRANSLEV	Filter levels on input
HLABEL	V1COMPAT	Version 1.5 compatibility mode
	TRACE	trace control (default=0)

Table. 6.1 Configuration Parameters used with Labels

Chapter 7

HMM Definition Files



The principle function of HTK is to manipulate sets of hidden Markov models (HMMs). The definition of a HMM must specify the model topology, the transition parameters and the output distribution parameters. The HMM observation vectors can be divided into multiple independent data streams and each stream can have its own weight. In addition, a HMM can have ancillary information such as duration parameters. HTK supports both continuous mixture densities and discrete distributions. HTK also provides a generalised tying mechanism which allows parameters to be shared within and between models.

In order to encompass this rich variety of HMM types within a single framework, HTK uses a formal language to define HMMs. The interpretation of this language is handled by the library module HMODEL which is responsible for converting between the external and internal representations of HMMs. In addition, it provides all the basic probability function calculations. A second module HUTIL provides various additional facilities for manipulating HMMs once they have been loaded into memory.

The purpose of this chapter is to describe the HMM definition language in some detail. The chapter begins by describing how to write individual HMM definitions. HTK macros are then explained and the mechanisms for defining a complete model set are presented. The various flavours of HMM are then described and the use of binary files discussed. Finally, a formal description of the HTK HMM definition language is given.

As will be seen, the definition of a large HMM system can involve considerable complexity. However, in practice, HMM systems are built incrementally. The usual starting point is a single HMM definition which is then repeatedly cloned and refined using the various HTK tools (in particular, HEREST and HHED). Hence, in practice, the HTK user rarely has to generate complex HMM definition files directly.

7.1 The HMM Parameters

A HMM consists of a number of states. Each state j has an associated observation probability distribution $b_j(\boldsymbol{o}_t)$ which determines the probability of generating observation $\boldsymbol{o}_t$ at time t and each pair of states i and j has an associated transition probability a_{ij}. In HTK the entry state 1 and the exit state N of an N state HMM are non-emitting.

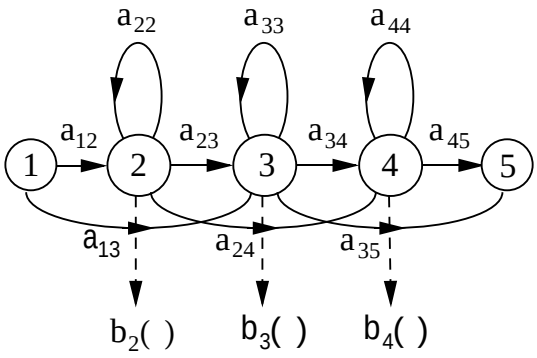


Fig. 7.1 Simple Left-Right HMM

Fig. 7.1 shows a simple left-right HMM with five states in total. Three of these are emitting states and have output probability distributions associated with them. The transition matrix for this model will have 5 rows and 5 columns. Each row will sum to one except for the final row which is always all zero since no transitions are allowed out of the final state.

HTK is principally concerned with continuous density models in which each observation probability distribution is represented by a mixture Gaussian density. In this case, for state j the probability $b_j(\boldsymbol{o}_t)$ of generating observation $\boldsymbol{o}_t$ is given by

$$b_j(\boldsymbol{o}_t) = \prod_{s=1}^{S} \left[\sum_{m=1}^{M_{js}} c_{jsm} \mathcal{N}(\boldsymbol{o}_{st}; \boldsymbol{\mu}_{jsm}, \boldsymbol{\Sigma}_{jsm}) \right]^{\gamma_s} \tag{7.1}$$

where M_{js} is the number of mixture components in state j for 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

$$\mathcal{N}(\boldsymbol{o}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{\sqrt{(2\pi)^n |\boldsymbol{\Sigma}|}} e^{-\frac{1}{2}(\boldsymbol{o}-\boldsymbol{\mu})' \boldsymbol{\Sigma}^{-1} (\boldsymbol{o}-\boldsymbol{\mu})} \tag{7.2}$$

where n is the dimensionality of $\boldsymbol{o}$. The exponent γ_s is a stream weight and its default value is one. Other values can be used to emphasise particular streams, however, none of the standard HTK tools manipulate it.

HTK also supports discrete probability distributions in which case

$$b_j(\boldsymbol{o}_t) = \prod_{s=1}^{S} \{P_{js}[v_s(\boldsymbol{o}_{st})]\}^{\gamma_s} \tag{7.3}$$

where $v_s(\boldsymbol{o}_{st})$ is the output of the vector quantiser for stream s given input vector $\boldsymbol{o}_{st}$ and $P_{js}[v]$ is the probability of state j generating symbol v in stream s.

In addition to the above, any model or state can have an associated vector of duration parameters $\{d_k\}$[1]. Also, it is necessary to specify the kind of the observation vectors, and the width of the observation vector in each stream. Thus, the total information needed to define a single HMM is as follows

- type of observation vector
- number and width of each data stream

[1] No current HTK tool can estimate or use these.

- optional model duration parameter vector
- number of states
- for each emitting state and each stream
  - mixture component weights or discrete probabilities
  - if continuous density, then means and covariances
  - optional stream weight vector
  - optional duration parameter vector
- transition matrix

The following sections explain how these are defined.

7.2 Basic HMM Definitions

Some HTK tools require a single HMM to be defined. For example, the isolated-unit re-estimation tool HREST would be invoked as

```
HRest hmmdef s1 s2 s3 ....
```

This would cause the model defined in the file `hmmdef` to be input and its parameters re-estimated using the speech data files `s1`, `s2`, etc.

```
~h "hmm1"
<BeginHMM>
    <VecSize> 4 <MFCC>
    <NumStates> 5
    <State> 2
       <Mean> 4
          0.2 0.1 0.1 0.9
       <Variance> 4
          1.0 1.0 1.0 1.0
    <State> 3
       <Mean> 4
          0.4 0.9 0.2 0.1
       <Variance> 4
          1.0 2.0 2.0 0.5
    <State> 4
       <Mean> 4
          1.2 3.1 0.5 0.9
       <Variance> 4
          5.0 5.0 5.0 5.0
    <TransP> 5
       0.0 0.5 0.5 0.0 0.0
       0.0 0.4 0.4 0.2 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>
```

Fig. 7.2 Definition for Simple L-R HMM

HMM definition files consist of a sequence of symbols representing the elements of a simple language. These symbols are mainly keywords written within angle brackets and integer and floating point numbers. The full HTK definition language is presented more formally later in section 7.11. For now, the main features of the language will be described by some examples.

Fig 7.2 shows a HMM definition corresponding to the simple left-right HMM illustrated in Fig 7.1. It is a continuous density HMM with 5 states in total, 3 of which are emitting. The first symbol in the file ~h indicates that the following string is the name of a macro of type h which means that it is a HMM definition (macros are explained in detail later). Thus, this definition describes a HMM called "hmm1". Note that HMM names should be composed of alphanumeric characters only and must not consist solely of numbers. The HMM definition itself is bracketed by the symbols <BeginHMM> and <EndHMM>.

The first line of the definition proper specifies the *global* features of the HMM. In any system consisting of many HMMs, these features will be the same for all of them. In this case, the global definitions indicate that the observation vectors have 4 components (<VecSize> 4) and that they denote MFCC coefficients (<MFCC>).

The next line specifies the number of states in the HMM. There then follows a definition for each emitting state j, each of which has a single mean vector $\boldsymbol{\mu}_j$ introduced by the keyword <Mean> and a diagonal variance vector $\boldsymbol{\Sigma}_j$ introduced by the keyword <Variance>. The definition ends with the transition matrix $\{a_{ij}\}$ introduced by the keyword <TransP>.

Notice that the dimension of each vector or matrix is specified explicitly before listing the component values. These dimensions must be consistent with the corresponding observation width (in the case of output distribution parameters) or number of states (in the case of transition matrices). Although in this example they could be inferred, HTK requires that they are included explicitly since, as will be described shortly, they can be detached from the HMM definition and stored elsewhere as a macro.

The definition for hmm1 makes use of many defaults. In particular, there is no definition for the number of input data streams or for the number of mixture components per output distribution. Hence, in both cases, a default of 1 is assumed.

Fig 7.3 shows a HMM definition in which the emitting states are 2 component mixture Gaussians. The number of mixture components in each state j is indicated by the keyword <NumMixes> and each mixture component is prefixed by the keyword <Mixture> followed by the component index m and component weight c_{jm}. Note that there is no requirement for the number of mixture components to be the same in each distribution.

State definitions and the mixture components within them may be listed in any order. When a HMM definition is loaded, a check is made that all the required components have been defined. In addition, checks are made that the mixture component weights and each row of the transition matrix sum to one. If very rapid loading is required, this consistency checking can be inhibited by setting the Boolean configuration variable `CHKHMMDEFS` to false.

As an alternative to diagonal variance vectors, a Gaussian distribution can have a full rank covariance matrix. An example of this is shown in the definition for hmm3 shown in Fig 7.4. Since covariance matrices are symmetric, they are stored in upper triangular form i.e. each row of the matrix starts at the diagonal element[2]. Also, covariance matrices are stored in their inverse form i.e. HMM definitions contain $\boldsymbol{\Sigma}^{-1}$ rather than $\boldsymbol{\Sigma}$. To reflect this, the keyword chosen to introduce a full covariance matrix is <InvCovar>.

[2] Covariance matrices are actually stored internally in lower triangular form

```
~h "hmm2"
<BeginHMM>
    <VecSize> 4 <MFCC>
    <NumStates> 4
    <State> 2 <NumMixes> 2
        <Mixture> 1 0.4
            <Mean> 4
                0.3 0.2 0.2 1.0
            <Variance> 4
                1.0 1.0 1.0 1.0
        <Mixture> 2 0.6
            <Mean> 4
                0.1 0.0 0.0 0.8
            <Variance> 4
                1.0 1.0 1.0 1.0
    <State> 3 <NumMixes> 2
        <Mixture> 1 0.7
            <Mean> 4
                0.1 0.2 0.6 1.4
            <Variance> 4
                1.0 1.0 1.0 1.0
        <Mixture> 2 0.3
            <Mean> 4
                2.1 0.0 1.0 1.8
            <Variance> 4
                1.0 1.0 1.0 1.0
    <TransP> 4
        0.0 1.0 0.0 0.0
        0.0 0.5 0.5 0.0
        0.0 0.0 0.6 0.4
        0.0 0.0 0.0 0.0
<EndHMM>
```

Fig. 7.3 Simple Mixture Gaussian HMM

Notice that only the second state has a full covariance Gaussian component. The first state has a mixture of two diagonal variance Gaussian components. Again, this illustrates the flexibility of HMM definition in HTK. If required the structure of every Gaussian can be individually configured.

Another possible way to store covariance information is in the form of the Choleski decomposition L of the inverse covariance matrix i.e. $\mathbf{\Sigma}^{-1} = LL'$. Again this is stored externally in upper triangular form so L' is actually stored. It is distinguished from the normal inverse covariance matrix by using the keyword <LLTCovar> in place of <InvCovar>[3].

The definition for hmm3 also illustrates another macro type, that is, ~o. This macro is used as an alternative way of specifying global options and, in fact, it is the format used by HTK tools when they write out a HMM definition. It is provided so that global options can be specifed ahead of any other HMM parameters. As will be seen later, this is useful when using many types of macro.

As noted earlier, the observation vectors used to represent the speech signal can be divided into two or more statistically independent data streams. This corresponds to the splitting-up of the input speech vectors as described in section 5.13. In HMM definitions, the use of multiple data streams must be indicated by specifying the number of streams and the width (i.e dimension) of each stream as a global option. This is done using the keyword <StreamInfo> followed by the number of streams, and then a sequence of numbers indicating the width of each stream. The sum of these stream widths must equal the original vector size as indicated by the <VecSize> keyword.

[3] The Choleski storage format is not used by default in HTK Version 2

```
~o <VecSize> 4 <MFCC>
~h "hmm3"
<BeginHMM>
    <NumStates> 4
    <State> 2 <NumMixes> 2
        <Mixture> 1 0.4
            <Mean> 4
                0.3 0.2 0.2 1.0
            <Variance> 4
                1.0 1.0 1.0 1.0
            <Mixture> 2 0.6
            <Mean> 4
                0.1 0.0 0.0 0.8
            <Variance> 4
                1.0 1.0 1.0 1.0
    <State> 3 <NumMixes> 1
        <Mean> 4
            0.10.20.61.4
        <InvCovar> 4
            1.00.10.00.0
                1.00.20.0
                    1.00.1
                        1.0
    <TransP> 4
        0.0 1.0 0.0 0.0
        0.0 0.5 0.5 0.0
        0.0 0.0 0.6 0.4
        0.0 0.0 0.0 0.0
<EndHMM>
```

Fig. 7.4 HMM with Full Covariance

An example of a HMM definition for multiple data streams is hmm4 shown in Fig 7.5. This HMM is intended to model 2 distinct streams, the first has 3 components and the second has 1. This is indicated by the global option <StreamInfo> 2 3 1. The definition of each state output distribution now includes means and variances for each individual stream.

Thus, in Fig 7.5, each state is subdivided into 2 streams using the <Stream> keyword followed by the stream number. Note also, that each individual stream can be weighted. In state 2 of hmm4, the vector following the <SWeights> keyword indicates that stream 1 has a weight of 0.9 whereas stream 2 has a weight of 1.1. There is no stream weight vector in state 3 and hence the default weight of 1.0 will be assigned to each stream.

No HTK tools are supplied for estimating optimal stream weight values. Hence, they must either be set manually or derived from some outside source. However, once set, they are used in the calculation of output probabilities as specified in equations 7.1 and 7.3, and hence they will affect the operation of both the training and recognition tools.

7.3 Macro Definitions

So far, basic model definitions have been described in which all of the information required to define a HMM has been given directly between the <BeginHMM> and <EndHMM> keywords. As an alternative, HTK allows the internal parts of a definition to be written as separate units, possibly

```
~o  <VecSize> 4 <MFCC>
    <StreamInfo> 2 3 1
~h "hmm4"
<BeginHMM>
    <NumStates> 4
    <State> 2
       <SWeights> 2 0.9 1.1
       <Stream> 1
          <Mean> 3
             0.2 0.1 0.1
          <Variance> 3
             1.0 1.0 1.0
       <Stream> 2
          <Mean> 1 0.0
          <Variance> 1 4.0
    <State> 3
       <Stream> 1
          <Mean> 3
             0.3 0.2 0.0
          <Variance> 3
             1.0 1.0 1.0
       <Stream> 2
          <Mean> 1 0.5
          <Variance> 1 3.0
    <TransP> 4
       0.0 1.0 0.0 0.0
       0.0 0.6 0.4 0.0
       0.0 0.0 0.4 0.6
       0.0 0.0 0.0 0.0
<EndHMM>
```

Fig. 7.5 HMM with 2 Data Streams

in several different files, and then referenced by name wherever they are needed. Such definitions are called *macros*.

```
~o <VecSize> 4 <MFCC>

~v "var"
    <Variance> 4
       1.0 1.0 1.0 1.0
```

Fig. 7.6 Simple Macro Definitions

HMM (~h) and global option macros (~o) have already been described. In fact, these are both rather special cases since neither is ever referenced explicitly by another definition. Indeed, the option macro is unusual in that since it is global and must be unique, it has no name. As an illustration of the use of macros, it may be observed that the variance vectors in the HMM definition hmm2 given in Fig 7.3 are all identical. If this was intentional, then the variance vector could be defined as a macro as illustrated in Fig 7.6.

A macro definition consists of a macro type indicator followed by a user-defined macro name. In this case, the indicator is ~v and the name is var. Notice that a global options macro is included before the definition for var. HTK must know these before it can process any other definitions thus the first macro file specified on the command line of any HTK tool must have the global options macro. Global options macro need not be repeated at the head of every definition file, but it does no harm to do so.

```
~h "hmm5"
<BeginHMM>
    <NumStates> 4
    <State> 2 <NumMixes> 2
       <Mixture> 1 0.4
          <Mean> 4
             0.3 0.2 0.2 1.0
          ~v "var"
       <Mixture> 2 0.6
          <Mean> 4
             0.1 0.0 0.0 0.8
          ~v "var"
    <State> 3 <NumMixes> 2
       <Mixture> 1 0.7
          <Mean> 4
             0.1 0.2 0.6 1.4
          ~v "var"
       <Mixture> 2 0.3
          <Mean> 4
             2.1 0.0 1.0 1.8
          ~v "var"
    <TransP> 4
       0.0 1.0 0.0 0.0
       0.0 0.5 0.5 0.0
       0.0 0.0 0.6 0.4
       0.0 0.0 0.0 0.0
<EndHMM>
```

Fig. 7.7 A Definition Using Macros

Once defined, a macro is used simply by writing the type indicator and name exactly as written in the definition. Thus, for example, Fig 7.7 defines a HMM called hmm5 which uses the variance macro var but is otherwise identical to the earlier HMM definition hmm2.

The definition for hmm5 can be understood by substituting the textual body of the var macro everywhere that it is referenced. Textually this would make the definition for hmm5 identical to that for hmm2, and indeed, if input to a recogniser, their effects would be similar. However, as will become clear in later chapters, the HMM definitions hmm2 and hmm5 differ in two ways. Firstly, if any attempt was made to re-estimate the parameters of hmm2, the values of the variance vectors would almost certainly diverge. However, the variance vectors of hmm5 are tied together and are guaranteed to remain identical, even after re-estimation. Thus, in general, the use of a macro enforces a *tying* which results in the corresponding parameters being shared amongst all the HMM structures which reference that macro. Secondly, when used in a recognition tool, the computation required to decode using HMMs with tied parameters will often be reduced. This is particularly true when higher level parts of a HMM definition are tied such as whole states.

There are many different macro types. Some have special meanings but the following correspond to the various distinct points in the hierarchy of HMM parameters which can be tied.

∼s	shared state distribution
∼m	shared Gaussian mixture component
∼u	shared mean vector
∼v	shared diagonal variance vector
∼i	shared inverse full covariance matrix
∼c	shared choleski L' matrix
∼x	shared arbitrary transform matrix[4]
∼t	shared transition matrix
∼d	shared duration parameters
∼w	shared stream weight vector

Fig 7.8 illustrates these potential tie points graphically for the case of continuous density HMMs. In this figure, each solid black circle represents a potential tie point, and the associated macro type is indicated alongside it.

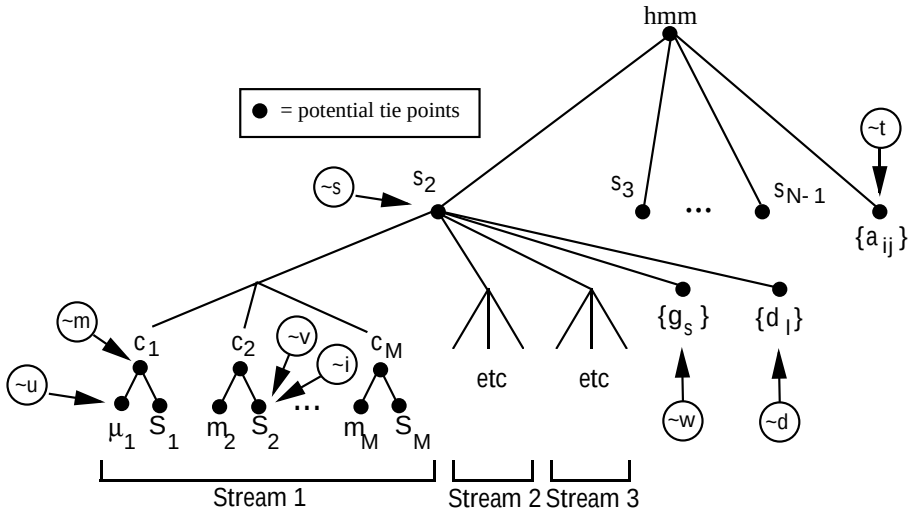


Fig. 7.8 HMM Hierarchy and Potential Tie Points

The tie points for discrete HMMs are identical except that the macro types ∼m, ∼v, ∼c, ∼i and ∼u are not relevant and are therefore excluded.

The macros with special meanings are as follows

∼l	logical HMM	∼h	physical HMM
∼o	global option values	∼p	tied mixture
∼r	regression class tree	∼j	linear transform

The distinction between logical and physical HMMs will be explained in the next section and option macros have already been described. The ∼p macro is used by the HMM editor HHEd for building tied mixture systems (see section 7.5). The ∼l or ∼p macros are special in the sense that they are created implicitly in order to represent specific kinds of parameter sharing and they never occur explicitly in HMM definitions.

7.4 HMM Sets

The previous sections have described how a single HMM definition can be specified. However, many HTK tools require complete model sets to be specified rather than just a single model. When this is the case, the individual HMMs which belong to the set are listed in a file rather than being enumerated explicitly on the command line. Thus, for example, a typical invocation of the tool HERest might be as follows

```
HERest ... -H mf1 -H mf2 ... hlist
```

where each `-H` option names a macro file and `hlist` contains a list of HMM names, one per line. For example, it might contain

```
ha
hb
hc
```

In a case such as this, the macro files would normally contain definitions for the models `ha`, `hb` and `hc`, along with any lower level macro definitions that they might require.

As an illustration, Fig 7.9 and Fig 7.10 give examples of what the macro files `mf1` and `mf2` might contain. The first file contains definitions for three states and a transition matrix. The second file contains definitions for the three HMMs. In this example, each HMM shares the three states and the common transition matrix. A HMM set such as this is called a *tied-state* system.

The order in which macro files are listed on the command line and the order of definition within each file must ensure that all macro definitions are defined before they are referenced. Thus, macro files are typically organised such that all low level structures come first followed by states and transition matrices, with the actual HMM definitions coming last.

When the HMM list contains the name of a HMM for which no corresponding macro has been defined, then an attempt is made to open a file with the same name. This file is expected to contain a single definition corresponding to the required HMM. Thus, the general mechanism for loading a set of HMMs is as shown in Fig 7.11. In this example, the HMM list `hlist` contains the names of five HMMs of which only three have been predefined via the macro files. Hence, the remaining definitions are found in individual HMM definition files `hd` and `he`.

When a large number of HMMs must be loaded from individual files, it is common to store them in a specific directory. Most HTK tools allow this directory to be specified explicitly using a command line option. For example, in the command

```
HERest -d hdir ... hlist ....
```

the definitions for the HMM listed in `hlist` will be searched for in the subdirectory `hdir`.

After loading each HMM set, HModel marks it as belonging to one of the following categories (called the *HSKind*

- `PLAINHS`
- `SHAREDHS`
- `TIEDHS`
- `DISCRETEHS`

Any HMM set containing discrete output distributions is assigned to the `DISCRETEHS` category (see section 7.6). If all mixture components are tied, then it is assigned to the `TIEDHS` category (see section 7.5). If it contains any shared states (∼s macros) or Gaussians (∼m macros) then it is `SHAREDHS`. Otherwise, it is `PLAINHS`. The category assigned to a HMM set determines which of several possible optimisations the various HTK tools can apply to it. As a check, the required kind

```
~o      <VecSize> 4 <MFCC>
~s "stateA"
     <Mean> 4
        0.2 0.1 0.1 0.9
     <Variance> 4
        1.0 1.0 1.0 1.0
~s "stateB"
     <Mean> 4
        0.4 0.9 0.2 0.1
     <Variance> 4
        1.0 2.0 2.0 0.5
~s "stateC"
     <Mean> 4
        1.2 3.1 0.5 0.9
     <Variance> 4
        5.0 5.0 5.0 5.0
~t "tran"
     <TransP> 5
        0.0 0.5 0.5 0.0 0.0
        0.0 0.4 0.4 0.2 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
```

Fig. 7.9 File mf1: shared state and transition matrix macros

of a HMM set can also be set via the configuration variable `HMMSETKIND`. For debugging purposes, this can also be used to re-categorise a `SHAREDHS` system as `PLAINHS`.

As shown in Figure 7.8, complete HMM definitions can be tied as well as their individual parameters. However, tying at the HMM level is defined in a different way. HMM lists have so far been described as simply a list of model names. In fact, every HMM has two names: a *logical* name and a *physical name*. The logical name reflects the rôle of the model and the physical name is used to identify the definition on disk. By default, the logical and physical names are identical. HMM tying is implemented by letting several logically distinct HMMs share the same physical definition. This is done by giving an explicit physical name immediately after the logical name in a HMM list.

```
~h "ha"
<BeginHMM>
    <NumStates> 5
    <State> 2
        ~s "stateA"
    <State> 3
        ~s "stateB"
    <State> 4
        ~s "stateB"
    ~t "tran"
<EndHMM>

~h "hb"
<BeginHMM>
    <NumStates> 5
    <State> 2
        ~s "stateB"
    <State> 3
        ~s "stateA"
    <State> 4
        ~s "stateC"
    ~t "tran"
<EndHMM>

~h "hc"
<BeginHMM>
    <NumStates> 5
    <State> 2
        ~s "stateC"
    <State> 3
        ~s "stateC"
    <State> 4
        ~s "stateB"
    ~t "tran"
<EndHMM>
```

Fig. 7.10 Simple Tied-State System

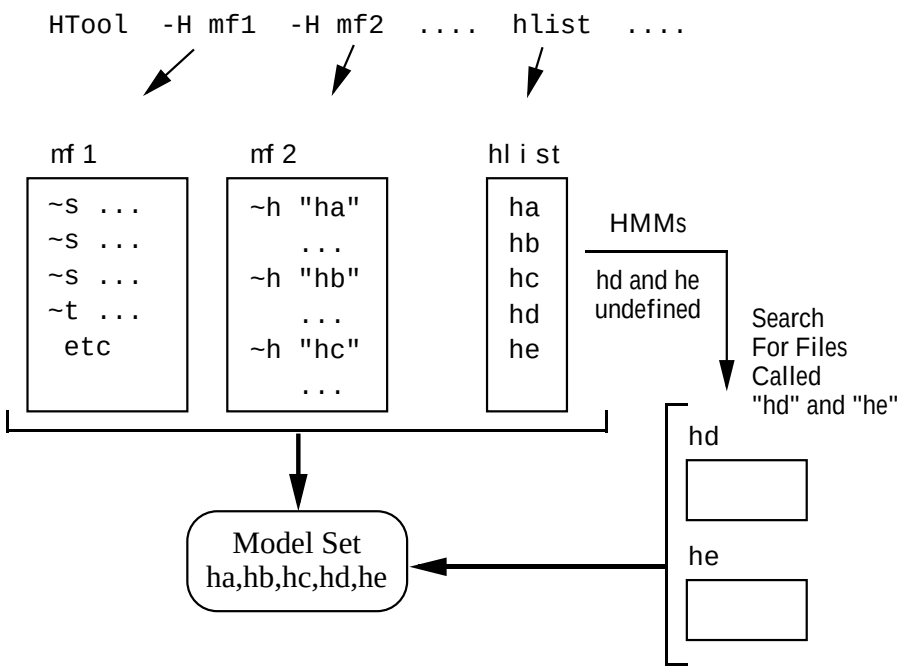


Fig. 7.11 Defining a Model Set

For example, in the HMM list shown in Fig 7.12, the logical HMMs `two`, `too` and `to` are tied and share the same physical HMM definition `tuw`. The HMMs `one` and `won` are also tied but in this case `won` shares `one`'s definition. There is, however, no subtle distinction here. The two different cases are given just to emphasise that the names used for the logical and physical HMMs can be the same or different, as is convenient. Finally, in this example, the models `three` and `four` are untied.

```
two     tuw
too     tuw
to      tuw
one
won     one
three
four
```

Fig. 7.12 HMM List with Tying

This mechanism is implemented internally by creating a ∼l macro definition for every HMM in the HMM list. If an explicit physical HMM is also given in the list, then the logical HMM is linked to that macro, otherwise a ∼h macro is created with the same name as the ∼l macro. Notice that this is one case where the "define before use" rule is relaxed. If an undefined ∼h is encountered then a dummy place-holder is created for it and, as explained above, HMODEL subsequently tries to find a HMM definition file of the same name.

Finally it should be noted that in earlier versions of HTK, there were no HMM macros. However, HMM definitions could be listed in a single *master macro file* or MMF. Each HMM definition began with its name written as a quoted string and ended with a period written on its own (just like master label files), and the first line of an MMF contained the string `#!MMF!#`. In HTK 3.2.1, the use of MMFs has been subsumed within the general macro definition facility using the ∼h type. However, for compatibility, the older MMF style of file can still be read by all HTK tools.

7.5 Tied-Mixture Systems

A Tied-Mixture System is one in which all Gaussian components are stored in a pool and all state output distributions share this pool. Fig 7.13 illustrates this for the case of single data stream.

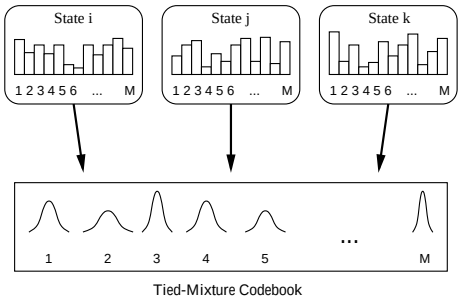


Fig. 7.13 Tied Mixture System

Each state output distribution is defined by M mixture component weights and since all states share the same components, all of the state-specific discrimination is encapsulated within these weights. The set of Gaussian components selected for the pool should be representative of the acoustic space covered by the feature vectors. To keep M manageable, multiple data streams are typically used with tied-mixture systems. For example, static parameters may be in one stream and delta parameters in another (see section 5.13). Each stream then has a separate pool of Gaussians which are often referred to as *codebooks*.

More formally, for S independent data streams, the output distribution for state j is defined as

$$b_j(\boldsymbol{o}_t) = \prod_{s=1}^{S} \left[\sum_{m=1}^{M_s} c_{jsm} \mathcal{N}(\boldsymbol{o}_{st}; \boldsymbol{\mu}_{sm}, \boldsymbol{\Sigma}_{sm}) \right]^{\gamma_s} \tag{7.4}$$

where the notation is identical to that used in equation 7.1. Note however that this equation differs from equation 7.1 in that the Gaussian component parameters and the number of mixture components per stream are state independent.

Tied-mixture systems lack the modelling accuracy of fully continuous density systems. However, they can often be implemented more efficiently since the total number of Gaussians which must be evaluated at each input frame is independent of the number of active HMM states and is typically much smaller.

A tied-mixture HMM system in HTK is defined by representing the pool of shared Gaussians as ~m macros with names "xxx1", "xxx2", ..., "xxxM" where "xxx" is an arbitrary name. Each HMM state definition is then specified by giving the name "xxx" followed by a list of the mixture weights. Multiple streams are identified using the <Stream> keyword as described previously.

As an example, Fig 7.14 shows a set of macro definitions which specify a 5 Gaussian component tied-mixture pool.

Fig 7.17 then shows a typical tied-mixture HMM definition which uses this pool. As can be seen, the mixture component weights are represented an array of real numbers as in the continuous density case.

The number of components in each tied-mixture codebook is typically of the order of 2 or 3 hundred. Hence, the list of mixture weights in each state is often long with many values being repeated, particularly floor values. To allow more efficient coding, successive identical values can be represented as a single value plus a repeat count in the form of an asterix followed by an integer multiplier. For example, Fig 7.15 shows the same HMM definition as above but using repeat counts. When HTK writes out a tied-mixture definition, it uses repeat counts wherever possible.

7.6 Discrete Probability HMMs

Discrete probability HMMs model observation sequences which consist of symbols drawn from a discrete and finite set of size M. As in the case of tied-mixture systems described above, this set is

```
~o <VecSize> 2 <MFCC>
~m "mix1"
    <Mean>  2 0.0 0.1
    <Variance> 2 1.0 1.0
~m "mix2"
    <Mean>  2 0.2 0.3
    <Variance> 2 2.0 1.0
~m "mix3"
    <Mean>  2 0.0 0.1
    <Variance> 2 1.0 2.0
~m "mix4"
    <Mean>  2 0.4 0.1
    <Variance> 2 1.0 1.5
~m "mix5"
    <Mean>  2 0.9 0.7
    <Variance> 2 1.5 1.0
```

Fig. 7.14 Tied-Mixture Codebook

```
~h "htm"
<BeginHMM>
    <NumStates> 4
    <State> 2 <NumMixes> 5
       <TMix> mix 0.2 0.1 0.3*2 0.1
    <State> 3 <NumMixes> 5
       <TMix> mix 0.4 0.3 0.1*3
    <TransP> 4
       ...
<EndHMM>
```

Fig. 7.15 HMM using Repeat Counts

often referred to as a *codebook*.

The form of the output distributions in a discrete HMM was given in equation 7.3. It consists of a table giving the probability of each possible observation symbol. Each symbol is identified by an index in the range 1 to M and hence the probability of any symbol can be determined by a simple table look-up operation.

For speech applications, the observation symbols are generated by a vector quantiser which typically associates a prototype speech vector with each codebook symbol. Each incoming speech vector is then represented by the symbol whose associated prototype is closest. The prototypes themselves are chosen to cover the acoustic space and they are usually calculated by clustering a representative sample of speech vectors.

In HTK, discrete HMMs are specified using a very similar notation to that used for tied-mixture HMMs. A discrete HMM can have multiple data streams but the width of each stream must be 1. The output probabilities are stored as logs in a scaled integer format such that if $d_{js}[v]$ is the stored discrete probability for symbol v in stream s of state j, the true probability is given by

$$P_{js}[v] = exp(-d_{js}[v]/2371.8) \tag{7.5}$$

Storage in the form of scaled logs allows discrete probability HMMs to be implemented very efficiently since HTK tools mostly use log arithmetic and direct storage in log form avoids the need for a run-time conversion. The range determined by the constant 2371.8 was selected to enable probabilities from 1.0 down to 0.000001 to be stored.

As an example, Fig 7.18 shows the definition of a discrete HMM called dhmm1. As can be seen, this has two streams. The codebook for stream 1 is size 10 and for stream 2, it is size 2. For

```
~j "lintran.mat"
<MMFIdMask> *
<MFCC>
<PreQual>
<LinXform>
    <VecSize> 2
    <BlockInfo> 1 2
    <Block> 1
        <Xform> 2 5
            1.0 0.1 0.2 0.1 0.4
            0.2 1.0 0.1 0.1 0.1
```

Fig. 7.16 Input Linear Transform

consistency with the representation used for continuous density HMMs, these sizes are encoded in the <NumMixes> specifier.

7.7 Input Linear Transforms

When reading feature vectors from files HTK will coerce them to the `TARGETKIND` specified in the config file. Often the `TARGETKIND` will contain certain qualifiers (specifying for example delta parameters). In addition to this parameter coercion it is possible to apply a linear transform before, or after, appending delta, acceleration and third derivative parameters.

Figure 7.16 shows an example linear transform. The <PreQual> keyword specifies that the linear transform is to be applied before the delta and delta-delta parameters specified in `TARGETKIND` are added. The default mode, no <PreQual> keyword, applies the linear transform after the addition of the qualifiers.

The linear transform fully supports projection from higher number of features to a smaller number of features. In the example, the parameterised data must consist of 5 `MFCC` parameters[5]. The model sets that are generated using this transform have a vector size of 2.

By default the linear transform is stored with the HMM. This is achieved by adding the <InputXform> keyword and specifying the transform or macroname. To allow compatibilty with tools only supporting the old format models it is possible to specify that no linear transform is to be stored with the model.

```
# Do not store linear transform
HMODEL: SAVEINPUTXFORM = FALSE
```

In addition it is possible to specify the linear transform as a HPARM configuration variable, `MATRTRANFN`.

```
# Specifying an input linear transform
HPARM: MATTRANFN = /home/test/lintran.mat
```

When a linear transform is specified in this form it is not necessary to have a macroname linked with it. In this case the filename will be used as the macroname (having stripped the directory name)

7.8 Tee Models

Normally, the transition probability from the non-emitting entry state to the non-emitting exit state of a HMM will be zero to ensure that the HMM aligns with at least one observation vector. Models which have a non-zero entry to exit transition probability are referred to as *tee-models*.

Tee-models are useful for modelling optional transient effects such as short pauses and noise bursts, particularly between words.

[5] If C0 or normalised log-energy are added these will be stripped prior to applying the linear transform

```
~h "htm"
<BeginHMM>
    <NumStates> 4
    <State> 2 <NumMixes> 5
       <TMix> mix 0.2 0.1 0.3 0.3 0.1
    <State> 3 <NumMixes> 5
       <TMix> mix 0.4 0.3 0.1 0.1 0.1
    <TransP> 4
       0.0 1.0 0.0 0.0
       0.0 0.5 0.5 0.0
       0.0 0.0 0.6 0.4
       0.0 0.0 0.0 0.0
<EndHMM>
```

Fig. 7.17 Tied-Mixture HMM

Although most HTK tools support tee-models, they are incompatible with those that work with isolated models such as HInit and HRest. When a tee-model is loaded into one of these tools, its entry to exit transition probability is reset to zero and the first row of its transition matrix is renormalised.

7.9 Regression Class Trees for Adaptation

In order to perform adaptation HTK generally requires the use of a binary regression tree. Its use in the adaptation process is explained in further detail in chapter 9. After construction (see section 10.7) the terminal nodes of the binary regression tree contain mixture component groupings or clusters. These clusters are referred to as regression base classes. Each mixture component in an HMM set belongs to a unique regression base class. The binary regression tree is stored as part of the HMM set, since its structure is necessary for the dynamic adaptation procedure described in section 9.1.2. Also each mixture component has a regression base class identifier (the terminal node indices) stored with it.

An example is shown in figure 7.19 and corresponds with the tree shown in figure 9.1. The example shows the use of the keyword <HMMSetId> used to store an identifier for this HMM set. This is important because the regression tree is built based on this HMM set and is hence specific to it. Many sets of transforms may be built that can be applied to this HMM set, but only one HMM set can be transformed by a transform set that utilises the regression tree. The regression tree is described by non-terminal nodes <Nodes> and terminal nodes <TNodes>. Each node contains its index followed by either the indices of its children (if it is a non-terminal) or the number of mixture components clustered at a terminal. Each mixture component as defined by the keyword <Mixture> has an <RClass> keyword followed by the regression base class index. When an HMM definition is loaded, a check is made to see that all the regression classes have been defined and that the total number of mixture components loaded for each regression class matches the number of mixture components defined in the regression tree.

The regression tree together with the mixture regression base class numbers can be constructed automatically with the use of the tool HHEd (see section 10.7).

7.10 Binary Storage Format

Throughout this chapter, a text-based representation has been used for the external storage of HMM definitions. For experimental work, text-based storage allows simple and direct access to HMM parameters and this can be invaluable. However, when using very large HMM sets, storage in text form is less practical since it is inefficient in its use of memory and the time taken to load can be excessive due to the large number of character to float conversions needed.

To solve these problems, HTK also provides a binary storage format. In binary mode, keywords are written as a single colon followed by an 8 bit code representing the actual keyword. Any

```
~o <DISCRETE> <StreamInfo> 2 1 1
~h "dhmm1"
<BeginHMM>
    <NumStates> 4
    <State> 2
        <NumMixes> 10 2
        <SWeights> 2 0.9 1.1
        <Stream> 1
            <DProb> 3288*4 32767*6
        <Stream> 2
            <DProb> 1644*2
    <State> 3
        <NumMixes> 10 2
        <SWeights> 2 0.9 1.1
        <Stream> 1
            <DProb> 5461*10
        <Stream> 2
            <DProb> 1644*2
    <TransP> 4
        0.0 1.0 0.0 0.0
        0.0 0.5 0.5 0.0
        0.0 0.0 0.6 0.4
        0.0 0.0 0.0 0.0
<EndHMM>
```

Fig. 7.18 Discrete Probability HMM

subsequent numerical information following the keyword is then in binary. Integers are written as 16-bit shorts and all floating-point numbers are written as 32-bit single precision floats. The repeat factor used in the run-length encoding scheme for tied-mixture and discrete HMMs is written as a single byte. Its presence immediately after a 16-bit discrete log probability is indicated by setting the top bit to 1 (this is the reason why the range of discrete log probabilities is limited to 0 to 32767 i.e. only 15 bits are used for the actual value). For tied-mixtures, the repeat count is signalled by subtracting 2.0 from the weight.

Binary storage format and text storage format can be mixed within and between input files. Each time a keyword is encountered, its coding is used to determine whether the subsequent numerical information should be input in text or binary form. This means, for example, that binary files can be manually patched by replacing a binary-format definition by a text format definition[6].

HTK tools provide a standard command line option (`-B`) to indicate that HMM definitions should be output in binary format. Alternatively, the Boolean configuration variable `SAVEBINARY` can be set to true to force binary format output.

7.11 The HMM Definition Language

To conclude this chapter, this section presents a formal description of the HMM definition language used by HTK. Syntax is described using an extended BNF notation in which alternatives are separated by a vertical bar |, parentheses () denote factoring, brackets [ ] denote options, and braces {} denote zero or more repetitions.

All keywords are enclosed in angle brackets[7] and the case of the keyword name is not significant. White space is not significant except within double-quoted strings.

The top level structure of a HMM definition is shown by the following rule.

[6] The fact that this is possible does not mean that it is recommended practice!

[7] This definition covers the textual version only. The syntax for the binary format is identical apart from the way that the lexical items are encoded.

```
~o <HMMSetId> ecrl_us_mono
    <VecSize> 4 <MFCC>
~r "ecrl_us_mono_tree_4"
    <RegTree> 4
    <Node> 1 2 3
    <Node> 2 4 5
    <Node> 3 6 7
    <TNode> 4 30
    <TNode> 5 25
    <TNode> 6 40
    <TNode> 7 39
~s "stateA" <NumMixes> 3
    <Mixture> 1 0.34
        <RClass> 4
        ~u "mean51"
        ~v "var65"
    <Mixture> 2 0.52
        <RClass> 7
        ~u "mean32"
        ~v "var65"
    <Mixture> 3 0.14
        <RClass> 5
        ~u "mean12"
        ~v "var3"
```

Fig. 7.19 MMF with a regression tree and classes

```
hmmdef =    [ ~h macro ]
            <BeginHMM>
                [ globalOpts ]
                <NumStates> short
                state { state }
                [ regTree ]
                transP
                [ duration ]
            <EndHMM>
```

A HMM definition consists of an optional set of global options followed by the <NumStates> keyword whose following argument specifies the number of states in the model inclusive of the non-emitting entry and exit states[8]. The information for each state is then given in turn, followed by the parameters of the transition matrix and the model duration parameters, if any. The name of the HMM is given by the ~h macro. If the HMM is the only definition within a file, the ~h macro name can be omitted and the HMM name is assumed to be the same as the file name.

The global options are common to all HMMs. They can be given separately using a ~o option macro

```
optmacro =    ~o globalOpts
```

or they can be included in one or more HMM definitions. Global options may be repeated but no definition can change a previous definition. All global options must be defined before any other macro definition is processed. In practice this means that any HMM system which uses parameter tying must have a ~o option macro at the head of the first macro file processed.

The full set of global options is given below. Every HMM set must define the vector size (via <VecSize>), the stream widths (via <StreamInfo>) and the observation parameter kind. However, if only the stream widths are given, then the vector size will be inferred. If only the vector size is given, then a single stream of identical width will be assumed. All other options default to null.

[8] Integer numbers are specified as either char or short. This has no effect on text-based definitions but for binary format it indicates the underlying C type used to represent the number.

```
globalOpts =  option { option }
option =      <HmmSetId> string |
              <StreamInfo> short { short } |
              <VecSize> short |
              <InputXform> inputXform |
              covkind |
              durkind |
              parmkind
```

The <HmmSetId> option allows the user to give the MMF an identifier. This is used as a sanity check to make sure that a TMF can be safely applied to this MMF. The arguments to the <StreamInfo> option are the number of streams (default 1) and then for each stream, the width of that stream. The <VecSize> option gives the total number of elements in each input vector. If both <VecSize> and <StreamInfo> are included then the sum of all the stream widths must equal the input vector size.

The covkind defines the kind of the covariance matrix

```
covkind =     <DiagC> | <InvDiagC> | <FullC> |
              <LLTC> | <XformC>
```

where <InvDiagC> is used internally. <LLTC> and <XformC> are not used in HTK Version 2.0. Setting the covariance kind as a global option forces all components to have this kind. In particular, it prevents mixing full and diagonal covariances within a HMM set.

The durkind denotes the type of duration model used according to the following rules

```
durkind =     <nullD> | <poissonD> | <gammaD> | <genD>
```

For anything other than <nullD>, a duration vector must be supplied for the model or each state as described below. Note that no current HTK tool can estimate or use such duration vectors.

The parameter kind is any legal parameter kind including qualified forms (see section 5.1)

```
parmkind =    <basekind{_D|_A|_T|_E|_N|_Z|_O|_V|_C|_K}>
basekind =    <discrete>|<lpc>|<lpcepstra>|<mfcc> | <fbank> |
              <melspec>| <lprefc>|<lpdelcep> | <user>
```

where the syntax rule for parmkind is non-standard in that no spaces are allowed between the base kind and any subsequent qualifiers. As noted in chapter 5, <lpdelcep> is provided only for compatibility with earlier versions of HTK and its further use should be avoided.

Each state of each HMM must have its own section defining the parameters associated with that state

```
state =       <State: Exp > short stateinfo
```

where the short following <State: Exp > is the state number. State information can be defined in any order. The syntax is as follows

```
stateinfo =   ~s macro |
              [ mixes ] [ weights ] stream { stream } [ duration ]
macro =       string
```

A stateinfo definition consists of an optional specification of the number of mixtures, an optional set of stream weights, followed by a block of information for each stream, optionally terminated with a duration vector. Alternatively, ~s macro can be written where macro is the name of a previously defined macro.

The optional mixes in a stateinfo definition specify the number of mixture components (or discrete codebook size) for each stream of that state

```
mixes =       <NumMixes> short {short}
```

where there should be one short for each stream. If this specification is omitted, it is assumed that all streams have just one mixture component.

The optional weights in a stateinfo definition define a set of exponent weights for each independent data stream. The syntax is

```
weights =    ~w macro | <SWeights> short vector
vector =     float { float }
```

where the short gives the number S of weights (which should match the value given in the <StreamInfo> option) and the vector contains the S stream weights γ_s (see section 7.1).

The definition of each stream depends on the kind of HMM set. In the normal case, it consists of a sequence of mixture component definitions optionally preceded by the stream number. If the stream number is omitted then it is assumed to be 1. For tied-mixture and discrete HMM sets, special forms are used.

```
stream =     [ <Stream> short ]
             (mixture { mixture } | tmixpdf | discpdf)
```

The definition of each mixture component consists of a Gaussian pdf optionally preceded by the mixture number and its weight

```
mixture =    [ <Mixture> short float ] mixpdf
```

If the <Mixture> part is missing then mixture 1 is assumed and the weight defaults to 1.0.

The tmixpdf option is used only for fully tied mixture sets. Since the mixpdf parts are all macros in a tied mixture system and since they are identical for every stream and state, it is only necessary to know the mixture weights. The tmixpdf syntax allows these to be specified in the following compact form

```
tmixpdf =    <TMix> macro weightList
weightList = repShort { repShort }
repShort =   short [ * char ]
```

where each short is a mixture component weight scaled so that a weight of 1.0 is represented by the integer 32767. The optional asterix followed by a char is used to indicate a repeat count. For example, 0*5 is equivalent to 5 zeroes. The Gaussians which make-up the pool of tied-mixtures are defined using ~m macros called macro1, macro2, macro3, etc.

Discrete probability HMMs are defined in a similar way

```
discpdf =    <DProb> weightList
```

The only difference is that the weights in the weightList are scaled log probabilities as defined in section 7.6.

The definition of a Gaussian pdf requires the mean vector to be given and one of the possible forms of covariance

```
mixpdf =     ~m macro | [ rclass ] mean cov [ <GConst> float ]
rclass =     <RClass> short
mean =       ~u macro | <Mean> short vector
cov =        var | inv | xform
var =        ~v macro | <Variance> short vector
inv =        ~i macro |
             (<InvCovar> | <LLTCovar>) short tmatrix
xform =      ~x macro | <Xform> short short matrix
matrix =     float {float}
tmatrix =    matrix
```

In mean and var, the short preceding the vector defines the length of the vector, in inv the short preceding the tmatrix gives the size of this square upper triangular matrix, and in xform the two short's preceding the matrix give the number of rows and columns. The optional <GConst>[9] gives that part of the log probability of a Gaussian that can be precomputed. If it is omitted, then it will be computed during load-in, including it simply saves some time. HTK tools which output HMM definitions always include this field. The optional <RClass> stores the regression base class index that this mixture component belongs to, as specified by the regression class tree (which is

[9] specifically, in equation 7.2 the GCONST value seen in HMM sets is calculated by multiplying the determinant of the covariance matrix by $(2\pi)^n$

also stored in the model set). HTK tools which output HMM definitions always include this field, and if there is no regression class tree then the regression identifier is set to zero.

In addition to defining the output distributions, a state can have a duration probability distribution defined for it. However, no current HTK tool can estimate or use these.

```
duration =    ~d macro | <Duration> short vector
```

Alternatively, as shown by the top level syntax for a hmmdef, duration parameters can be specified for a whole model.

A binary regression class tree (for the purposes of HMM adaptation as in chapter 9) may also exist for an HMM set. This is defined by

```
regTree =     ~r macro tree
tree =        <RegTree> short nodes
nodes =       (<Node> short short short | <TNode> short int) [ nodes ]
```

In tree the short preceding the nodes refers to the number of terminal nodes or leaves that the regression tree contains. Each node in nodes can either be a non-terminal <Node> or a terminal (leaf) <TNode>. For a <Node> the three following shorts refer to the node's index number and the index numbers of its children. For a <TNode>, the short refers to the leaf's index (which correspond to a regression base class index as stored at the component level in RClass, see above), while the int refers to the number of mixture components in this leaf cluster.

The transition matrix is defined by

```
transP =      ~t macro | <TransP> short matrix
```

where the short in this case should be equal to the number of states in the model.

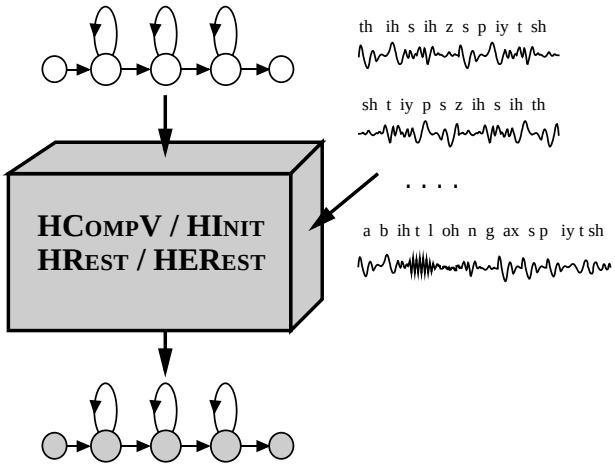
Finally the input transform is defined by

```
inputXform =  ~j macro | inhead inmatrix
inhead =      <MMFIdMask> string parmkind [<PreQual>]
inmatrix =    <LinXform> <VecSize> short <BlockInfo> short short {short} block {block}
block =       <Block> short xform
```

where the short following <VecSize> is the number of dimensions after applyingthe linear transform and must match the vector size of the HMM definition. The first short after <BlockInfo> is the number of block, this is followed by the number of output dimensions from each of the blocks.

Chapter 8

HMM Parameter Estimation



In chapter 7 the various types of HMM were described and the way in which they are represented within HTK was explained. Defining the structure and overall form of a set of HMMs is the first step towards building a recogniser. The second step is to estimate the parameters of the HMMs from examples of the data sequences that they are intended to model. This process of parameter estimation is usually called *training*. HTK supplies four basic tools for parameter estimation: HCompV, HInit, HRest and HERest. HCompV and HInit are used for initialisation. HCompV will set the mean and variance of every Gaussian component in a HMM definition to be equal to the global mean and variance of the speech training data. This is typically used as an initialisation stage for *flat-start* training. Alternatively, a more detailed initialisation is possible using HInit which will compute the parameters of a new HMM using a Viterbi style of estimation.

HRest and HERest are used to refine the parameters of existing HMMs using Baum-Welch Re-estimation. Like HInit, HRest performs *isolated-unit* training whereas HERest operates on complete model sets and performs *embedded-unit* training. In general, whole word HMMs are built using HInit and HRest, and continuous speech sub-word based systems are built using HERest initialised by either HCompV or HInit and HRest.

This chapter describes these training tools and their use for estimating the parameters of plain (i.e. untied) continuous density HMMs. The use of tying and special cases such as tied-mixture HMM sets and discrete probality HMMs are dealt with in later chapters. The first section of this chapter gives an overview of the various training strategies possible with HTK. This is then followed by sections covering initialisation, isolated-unit training, and embedded training. The chapter concludes with a section detailing the various formulae used by the training tools.

8.1 Training Strategies

As indicated in the introduction above, the basic operation of the HTK training tools involves reading in a set of one or more HMM definitions, and then using speech data to estimate the

parameters of these definitions. The speech data files are normally stored in parameterised form such as `LPC` or `MFCC` parameters. However, additional parameters such as delta coefficients are normally computed *on-the-fly* whilst loading each file.

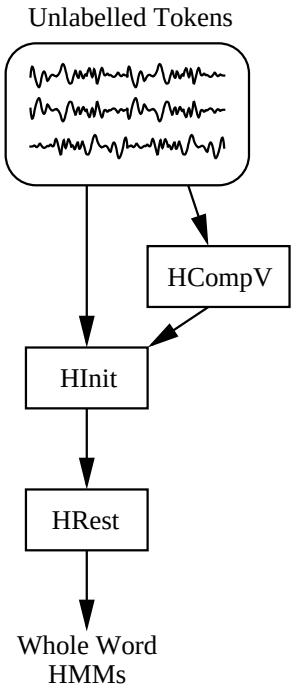


Fig. 8.1 Isolated Word Training

In fact, it is also possible to use waveform data directly by performing the full parameter conversion *on-the-fly*. Which approach is preferred depends on the available computing resources. The advantages of storing the data already encoded are that the data is more compact in parameterised form and pre-encoding avoids wasting compute time converting the data each time that it is read in. However, if the training data is derived from CD-ROMS and they can be accessed automatically on-line, then the extra compute may be worth the saving in magnetic disk storage.

The methods for configuring speech data input to HTK tools were described in detail in chapter 5. All of the various input mechanisms are supported by the HTK training tools except direct audio input.

The precise way in which the training tools are used depends on the type of HMM system to be built and the form of the available training data. Furthermore, HTK tools are designed to interface cleanly to each other, so a large number of configurations are possible. In practice, however, HMM-based speech recognisers are either whole-word or sub-word.

As the name suggests, whole word modelling refers to a technique whereby each individual word in the system vocabulary is modelled by a single HMM. As shown in Fig. 8.1, whole word HMMs are most commonly trained on examples of each word spoken in isolation. If these training examples, which are often called *tokens*, have had leading and trailing silence removed, then they can be input directly into the training tools without the need for any label information. The most common method of building whole word HMMs is to firstly use HINIT to calculate initial parameters for the model and then use HREST to refine the parameters using Baum-Welch re-estimation. Where there is limited training data and recognition in adverse noise environments is needed, so-called *fixed variance* models can offer improved robustness. These are models in which all the variances are set equal to the global speech variance and never subsequently re-estimated. The tool HCOMPV can be used to compute this global variance.

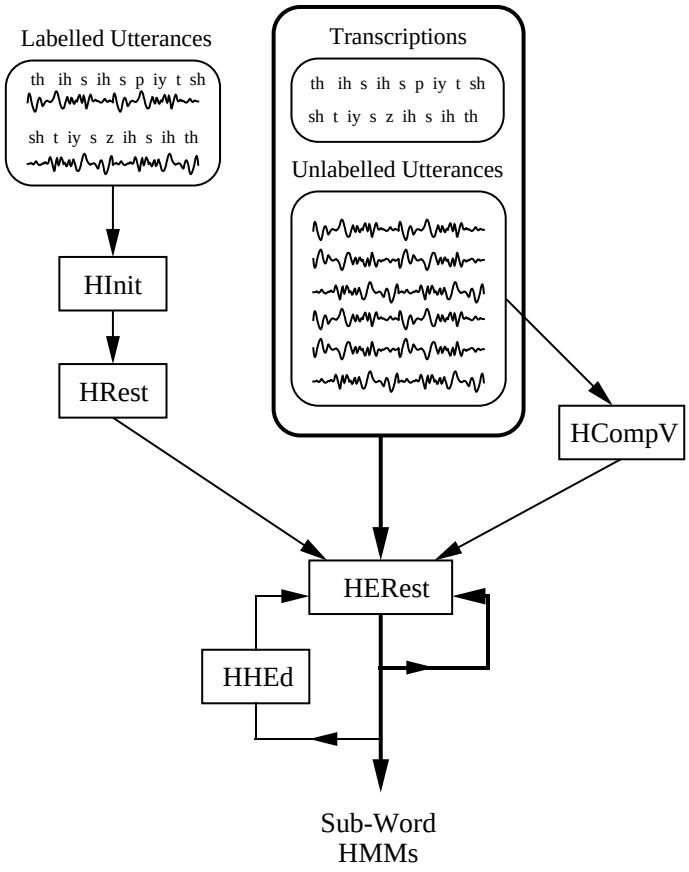


Fig. 8.2 Training Subword HMMs

Although HTK gives full support for building whole-word HMM systems, the bulk of its facilities are focussed on building sub-word systems in which the basic units are the individual sounds of the language called *phones*. One HMM is constructed for each such phone and continuous speech is recognised by joining the phones together to make any required vocabulary using a pronunciation dictionary.

The basic procedures involved in training a set of subword models are shown in Fig. 8.2. The core process involves the embedded training tool HEREST. HEREST uses continuously spoken utterances as its source of training data and simultaneously re-estimates the complete set of subword HMMs. For each input utterance, HEREST needs a transcription i.e. a list of the phones in that utterance. HEREST then joins together all of the subword HMMs corresponding to this phone list to make a single composite HMM. This composite HMM is used to collect the necessary statistics for the re-estimation. When all of the training utterances have been processed, the total set of accumulated statistics are used to re-estimate the parameters of all of the phone HMMs. It is important to emphasise that in the above process, the transcriptions are only needed to identify the sequence of phones in each utterance. No phone boundary information is needed.

The initialisation of a set of phone HMMs prior to embedded re-estimation using HEREST can be achieved in two different ways. As shown on the left of Fig. 8.2, a small set of hand-labelled *bootstrap* training data can be used along with the isolated training tools HINIT and HREST to initialise each phone HMM individually. When used in this way, both HINIT and HREST use the label information to extract all the segments of speech corresponding to the current phone HMM in order to perform isolated word training.

A simpler initialisation procedure uses HCOMPV to assign the global speech mean and variance to every Gaussian distribution in every phone HMM. This so-called *flat start* procedure implies that during the first cycle of embedded re-estimation, each training utterance will be uniformly segmented. The hope then is that enough of the phone models align with actual realisations of that phone so that on the second and subsequent iterations, the models align as intended.

One of the major problems to be faced in building any HMM-based system is that the amount of training data for each model will be variable and is rarely sufficient. To overcome this, HTK allows a variety of sharing mechanisms to be implemented whereby HMM parameters are tied together so that the training data is pooled and more robust estimates result. These tyings, along with a variety of other manipulations, are performed using the HTK HMM editor HHEd. The use of HHEd is described in a later chapter. Here it is sufficient to note that a phone-based HMM set typically goes through several refinement cycles of editing using HHEd followed by parameter re-estimation using HERest before the final model set is obtained.

Having described in outline the main training strategies, each of the above procedures will be described in more detail.

8.2 Initialisation using HInit

In order to create a HMM definition, it is first necessary to produce a prototype definition. As explained in Chapter 7, HMM definitions can be stored as a text file and hence the simplest way of creating a prototype is by using a text editor to manually produce a definition of the form shown in Fig 7.2, Fig 7.3 etc. The function of a prototype definition is to describe the form and topology of the HMM, the actual numbers used in the definition are not important. Hence, the vector size and parameter kind should be specified and the number of states chosen. The allowable transitions between states should be indicated by putting non-zero values in the corresponding elements of the transition matrix and zeros elsewhere. The rows of the transition matrix must sum to one except for the final row which should be all zero. Each state definition should show the required number of streams and mixture components in each stream. All mean values can be zero but diagonal variances should be positive and covariance matrices should have positive diagonal elements. All state definitions can be identical.

Having set up an appropriate prototype, a HMM can be initialised using the HTKtool HInit. The basic principle of HInit depends on the concept of a HMM as a generator of speech vectors. Every training example can be viewed as the output of the HMM whose parameters are to be estimated. Thus, if the state that generated each vector in the training data was known, then the unknown means and variances could be estimated by averaging all the vectors associated with each state. Similarly, the transition matrix could be estimated by simply counting the number of time slots that each state was occupied. This process is described more formally in section 8.8 below.

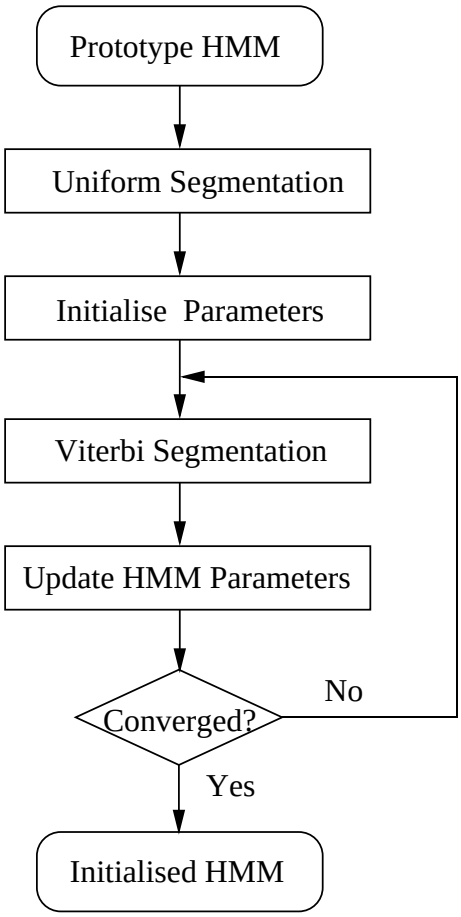


Fig. 8.3 HInit Operation

The above idea can be implemented by an iterative scheme as shown in Fig 8.3. Firstly, the Viterbi algorithm is used to find the most likely state sequence corresponding to each training example, then the HMM parameters are estimated. As a side-effect of finding the Viterbi state alignment, the log likelihood of the training data can be computed. Hence, the whole estimation process can be repeated until no further increase in likelihood is obtained.

This process requires some initial HMM parameters to get started. To circumvent this problem, HInit starts by uniformly segmenting the data and associating each successive segment with successive states. Of course, this only makes sense if the HMM is left-right. If the HMM is ergodic, then the uniform segmentation can be disabled and some other approach taken. For example, HCompV can be used as described below.

If any HMM state has multiple mixture components, then the training vectors are associated with the mixture component with the highest likelihood. The number of vectors associated with each component within a state can then be used to estimate the mixture weights. In the uniform segmentation stage, a K-means clustering algorithm is used to cluster the vectors within each state.

Turning now to the practical use of HInit, whole word models can be initialised by typing a command of the form

```
HInit hmm data1 data2 data3
```

where `hmm` is the name of the file holding the prototype HMM and `data1`, `data2`, etc. are the names of the speech files holding the training examples, each file holding a single example with no leading or trailing silence. The HMM definition can be distributed across a number of macro files loaded using the standard `-H` option. For example, in

```
HInit -H mac1 -H mac2 hmm data1 data2 data3 ...
```

then the macro files `mac1` and `mac2` would be loaded first. If these contained a definition for `hmm`, then no further HMM definition input would be attempted. If however, they did not contain a definition for `hmm`, then HINIT would attempt to open a file called `hmm` and would expect to find a definition for `hmm` within it. HINIT can in principle load a large set of HMM definitions, but it will only update the parameters of the single named HMM. On completion, HINIT will write out new versions of all HMM definitions loaded on start-up. The default behaviour is to write these to the current directory which has the usually undesirable effect of overwriting the prototype definition. This can be prevented by specifying a new directory for the output definitions using the `-M` option. Thus, typical usage of HINIT takes the form

```
HInit -H globals -M dir1 proto data1 data2 data3 ...
mv dir1/proto dir1/wordX
```

Here `globals` is assumed to hold a global options macro (and possibly others). The actual HMM definition is loaded from the file `proto` in the current directory and the newly initialised definition along with a copy of `globals` will be written to `dir1`. Since the newly created HMM will still be called `proto`, it is renamed as appropriate.

For most real tasks, the number of data files required will exceed the command line argument limit and a script file is used instead. Hence, if the names of the data files are stored in the file `trainlist` then typing

```
HInit -S trainlist -H globals -M dir1 proto
```

would have the same effect as previously.

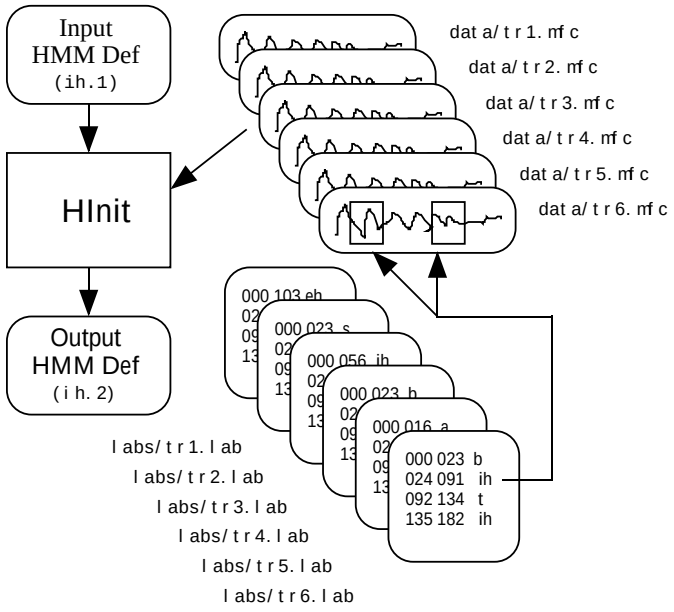


Fig. 8.4 File Processing in HInit

When building sub-word models, HINIT can be used in the same manner as above to initialise each individual sub-word HMM. However, in this case, the training data is typically continuous speech with associated label files identifying the speech segments corresponding to each sub-word. To illustrate this, the following command could be used to initialise a sub-word HMM for the phone `ih`

```
HInit -S trainlist -H globals -M dir1 -l ih -L labs proto
mv dir1/proto dir1/ih
```

where the option `-l` defines the name of the sub-word model, and the file `trainlist` is assumed to hold

```
data/tr1.mfc
data/tr2.mfc
data/tr3.mfc
data/tr4.mfc
data/tr5.mfc
data/tr6.mfc
```

In this case, HInit will first try to find label files corresponding to each data file. In the example here, the standard `-L` option indicates that they are stored in a directory called `labs`. As an alternative, they could be stored in a Master Label File (MLF) and loaded via the standard option `-I`. Once the label files have been loaded, each data file is scanned and all segments corresponding the label `ih` are loaded. Figure 8.4 illustrates this process.

All HTK tools support the `-T` trace option and although the details of tracing varies from tool to tool, setting the least signicant bit (e.g. by `-T 1`), causes all tools to output top level progress information. In the case of HInit, this information includes the log likelihood at each iteration and hence it is very useful for monitoring convergence. For example, enabling top level tracing in the previous example might result in the following being output

```
Initialising  HMM proto . . .
 States   :   2  3  4 (width)
 Mixes  s1:   1  1  1 ( 26  )
 Num Using:   0  0  0
 Parm Kind:  MFCC_E_D
 Number of owners = 1
 SegLab   :  ih
 maxIter  :  20
 epsilon  :  0.000100
 minSeg   :  3
 Updating :  Means Variances MixWeights/DProbs TransProbs
16 Observation Sequences Loaded
Starting Estimation Process
Iteration 1: Average LogP =  -898.24976
Iteration 2: Average LogP =  -884.05402  Change =    14.19574
Iteration 3: Average LogP =  -883.22119  Change =     0.83282
Iteration 4: Average LogP =  -882.84381  Change =     0.37738
Iteration 5: Average LogP =  -882.76526  Change =     0.07855
Iteration 6: Average LogP =  -882.76526  Change =     0.00000
Estimation converged at iteration 7
Output written to directory :dir1:
```

The first part summarises the structure of the HMM, in this case, the data is single stream MFCC coefficients with energy and deltas appended. The HMM has 3 emitting states, each single Gaussian and the stream width is 26. The current option settings are then given followed by the convergence information. In this example, convergence was reached after 6 iterations, however if the `maxIter` limit was reached, then the process would terminate regardless.

HInit provides a variety of command line options for controlling its detailed behaviour. The types of parameter estimated by HInit can be controlled using the `-u` option, for example, `-u mtw` would update the means, transition matrices and mixture component weights but would leave the variances untouched. A variance floor can be applied using the `-v` to prevent any variance getting too small. This option applies the same variance floor to all speech vector elements. More precise control can be obtained by specifying a variance macro (i.e. a `v` macro) called `varFloor1` for stream 1, `varFloor2` for stream 2, etc. Each element of these variance vectors then defines a floor for the corresponding HMM variance components.

The full list of options supported by HInit is described in the Reference Section.

8.3 Flat Starting with HCompV

One limitation of using HInit for the initialisation of sub-word models is that it requires labelled training data. For cases where this is not readily available, an alternative initialisation strategy is to make all models equal initially and move straight to embedded training using HERest. The

idea behind this so-called *flat start* training is similar to the uniform segmentation strategy adopted by HInit since by making all states of all models equal, the first iteration of embedded training will effectively rely on a uniform segmentation of the data.

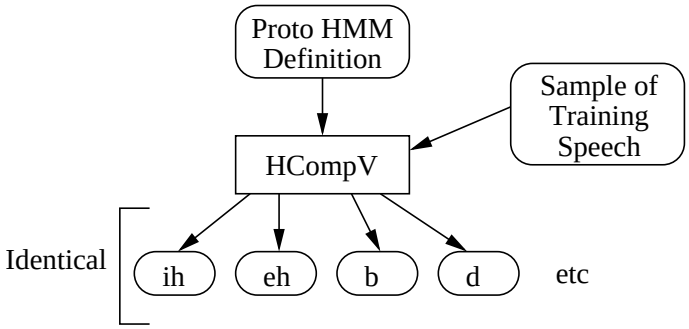


Fig. 8.5 Flat Start Initialisation

Flat start initialisation is provided by the HTK tool HCompV whose operation is illustrated by Fig 8.5. The input/output of HMM definition files and training files in HCompV works in exactly the same way as described above for HInit. It reads in a prototype HMM definition and some training data and outputs a new definition in which every mean and covariance is equal to the global speech mean and covariance. Thus, for example, the following command would read a prototype definition called `proto`, read in all speech vectors from `data1`, `data2`, `data3`, etc, compute the global mean and covariance and write out a new version of `proto` in `dir1` with this mean and covariance.

```
HCompV -m -H globals -M dir1 proto data1 data2 data3 ...
```

The default operation of HCompV is only to update the covariances of the HMM and leave the means unchanged. The use of the `-m` option above causes the means to be updated too. This apparently curious default behaviour arises because HCompV is also used to initialise the variances in so-called *Fixed-Variance* HMMs. These are HMMs initialised in the normal way except that all covariances are set equal to the global speech covariance and never subsequently changed.

Finally, it should be noted that HCompV can also be used to generate variance floor macros by using the `-f` option.

8.4 Isolated Unit Re-Estimation using HRest

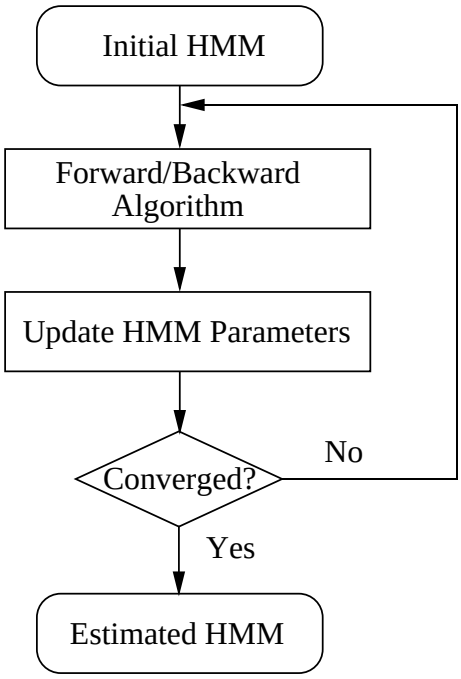


Fig. 8.6 HRest Operation

HREST is the final tool in the set designed to manipulate isolated unit HMMs. Its operation is very similar to HINIT except that, as shown in Fig 8.6, it expects the input HMM definition to have been initialised and it uses Baum-Welch re-estimation in place of Viterbi training. This involves finding the probability of being in each state at each time frame using the *Forward-Backward* algorithm. This probability is then used to form weighted averages for the HMM parameters. Thus, whereas Viterbi training makes a hard decision as to which state each training vector was "generated" by, Baum-Welch takes a soft decision. This can be helpful when estimating phone-based HMMs since there are no hard boundaries between phones in real speech and using a soft decision may give better results. The mathematical details of the Baum-Welch re-estimation process are given below in section 8.8.

HREST is usually applied directly to the models generated by HINIT. Hence for example, the generation of a sub-word model for the phone `ih` begun in section 8.2 would be continued by executing the following command

```
HRest -S trainlist -H dir1/globals -M dir2 -l ih -L labs dir1/ih
```

This will load the HMM definition for `ih` from `dir1`, re-estimate the parameters using the speech segments labelled with `ih` and write the new definition to directory `dir2`.

If HREST is used to build models with a large number of mixture components per state, a strategy must be chosen for dealing with *defunct mixture components*. These are mixture components which have very little associated training data and as a consequence either the variances or the corresponding mixture weight becomes very small. If either of these events happen, the mixture component is effectively deleted and provided that at least one component in that state is left, a warning is issued. If this behaviour is not desired then the variance can be floored as described previously using the `-v` option (or a variance floor macro) and/or the mixture weight can be floored using the `-w` option.

Finally, a problem which can arise when using HREST to initialise sub-word models is that of over-short training segments. By default, HREST ignores all training examples which have fewer frames than the model has emitting states. For example, suppose that a particular phone with 3 emitting states had only a few training examples with more than 2 frames of data. In this case, there would be two solutions. Firstly, the number of emitting states could be reduced. Since

HTK does not require all models to have the same number of states, this is perfectly feasible. Alternatively, some skip transitions could be added and the default reject mechanism disabled by setting the `-t` option. Note here that HInit has the same reject mechanism and suffers from the same problems. HInit, however, does not allow the reject mechanism to be suppressed since the uniform segmentation process would otherwise fail.

8.5 Embedded Training using HERest

Whereas isolated unit training is sufficient for building whole word models and initialisation of models using hand-labelled *bootstrap* data, the main HMM training procedures for building subword systems revolve around the concept of *embedded training*. Unlike the processes described so far, embedded training simultaneously updates all of the HMMs in a system using all of the training data. It is performed by HERest which, unlike HRest, performs just a single iteration.

In outline, HERest works as follows. On startup, HERest loads in a complete set of HMM definitions. Every training file must have an associated label file which gives a transcription for that file. Only the sequence of labels is used by HERest, however, and any boundary location information is ignored. Thus, these transcriptions can be generated automatically from the known orthography of what was said and a pronunciation dictionary.

HERest processes each training file in turn. After loading it into memory, it uses the associated transcription to construct a composite HMM which spans the whole utterance. This composite HMM is made by concatenating instances of the phone HMMs corresponding to each label in the transcription. The Forward-Backward algorithm is then applied and the sums needed to form the weighted averages accumulated in the normal way. When all of the training files have been processed, the new parameter estimates are formed from the weighted sums and the updated HMM set is output.

The mathematical details of embedded Baum-Welch re-estimation are given below in section 8.8.

In order to use HERest, it is first necessary to construct a file containing a list of all HMMs in the model set with each model name being written on a separate line. The names of the models in this list must correspond to the labels used in the transcriptions and there must be a corresponding model for every distinct transcription label. HERest is typically invoked by a command line of the form

```
HERest -S trainlist -I labs -H dir1/hmacs -M dir2 hmmlist
```

where `hmmlist` contains the list of HMMs. On startup, HERest will load the HMM master macro file (MMF) `hmacs` (there may be several of these). It then searches for a definition for each HMM listed in the `hmmlist`, if any HMM name is not found, it attempts to open a file of the same name in the current directory (or a directory designated by the `-d` option). Usually in large subword systems, however, all of the HMM definitions will be stored in MMFs. Similarly, all of the required transcriptions will be stored in one or more Master Label Files (MLFs), and in the example, they are stored in the single MLF called `labs`.

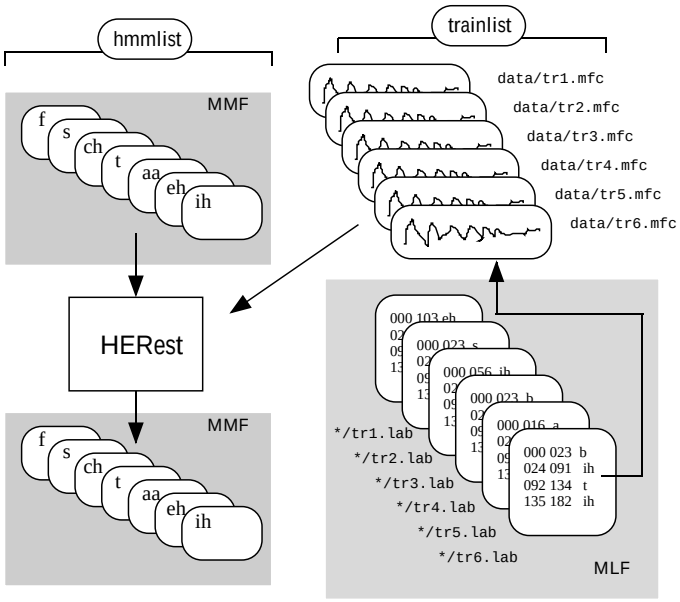


Fig. 8.7 File Processing in HERest

Once all MMFs and MLFs have been loaded, HEREST processes each file in the `trainlist`, and accumulates the required statistics as described above. On completion, an updated MMF is output to the directory `dir2`. If a second iteration is required, then HEREST is reinvoked reading in the MMF from `dir2` and outputing a new one to `dir3`, and so on. This process is illustrated by Fig 8.7.

When performing embedded training, it is good practice to monitor the performance of the models on unseen test data and stop training when no further improvement is obtained. Enabling top level tracing by setting `-T 1` will cause HEREST to output the overall log likelihood per frame of the training data. This measure could be used as a termination condition for repeated application of HEREST. However, repeated re-estimation to convergence may take an impossibly long time. Worse still, it can lead to over-training since the models can become too closely matched to the training data and fail to generalise well on unseen test data. Hence in practice around 2 to 5 cycles of embedded re-estimation are normally sufficient when training phone models.

In order to get accurate acoustic models, a large amount of training data is needed. Several hundred utterances are needed for speaker dependent recognition and several thousand are needed for speaker independent recognition. In the latter case, a single iteration of embedded training might take several hours to compute. There are two mechanisms for speeding up this computation. Firstly, HEREST has a pruning mechanism incorporated into its forward-backward computation. HEREST calculates the backward probabilities $\beta_j(t)$ first and then the forward probabilities $\alpha_j(t)$. The full computation of these probabilities for all values of state j and time t is unnecessary since many of these combinations will be highly improbable. On the forward pass, HEREST restricts the computation of the α values to just those for which the total log likelihood as determined by the product $\alpha_j(t)\beta_j(t)$ is within a fixed distance from the total likelihood $P(\boldsymbol{O}|M)$. This pruning is always enabled since it is completely safe and causes no loss of modelling accuracy.

Pruning on the backward pass is also possible. However, in this case, the likelihood product $\alpha_j(t)\beta_j(t)$ is unavailable since $\alpha_j(t)$ has yet to be computed, and hence a much broader *beam* must be set to avoid pruning errors. Pruning on the backward path is therefore under user control. It is set using the `-t` option which has two forms. In the simplest case, a fixed pruning beam is set. For example, using `-t 250.0` would set a fixed beam of 250.0. This method is adequate when there is sufficient compute time available to use a generously wide beam. When a narrower beam is used, HEREST will reject any utterance for which the beam proves to be too narrow. This can be avoided by using an incremental threshold. For example, executing

```
HERest -t 120.0 60.0 240.0 -S trainlist -I labs \
       -H dir1/hmacs -M dir2 hmmlist
```

would cause HERest to run normally at a beam width of 120.0. However, if a pruning error occurs, the beam is increased by 60.0 and HERest reprocesses the offending training utterance. Repeated errors cause the beam width to be increased again and this continues until either the utterance is successfully processed or the upper beam limit is reached, in this case 240.0. Note that errors which occur at very high beam widths are often caused by transcription errors, hence, it is best not to set the upper limit too high.

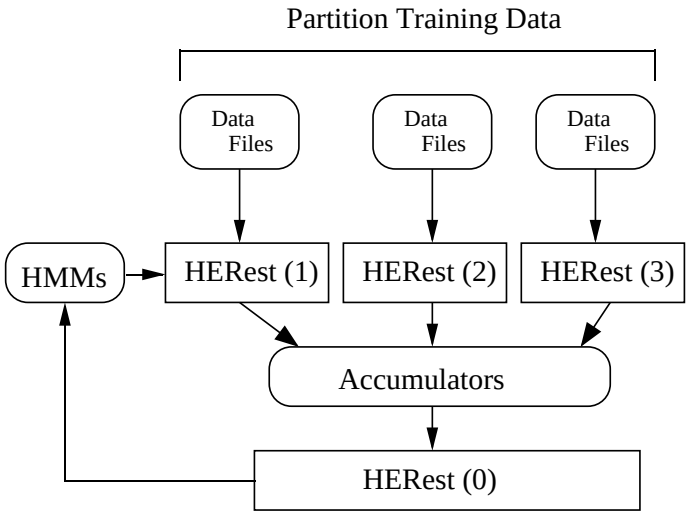


Fig. 8.8 HERest Parallel Operation

The second way of speeding-up the operation of HERest is to use more than one computer in parallel. The way that this is done is to divide the training data amongst the available machines and then to run HERest on each machine such that each invocation of HERest uses the same initial set of models but has its own private set of data. By setting the option `-p N` where `N` is an integer, HERest will dump the contents of all its accumulators into a file called `HERN.acc` rather than updating and outputing a new set of models. These dumped files are collected together and input to a new invocation of HERest with the option `-p 0` set. HERest then reloads the accumulators from all of the dump files and updates the models in the normal way. This process is illustrated in Figure 8.8.

To give a concrete example, suppose that four networked workstations were available to execute the HERest command given earlier. The training files listed previously in `trainlist` would be split into four equal sets and a list of the files in each set stored in `trlist1`, `trlist2`, `trlist3`, and `trlist4`. On the first workstation, the command

```
HERest -S trlist1 -I labs -H dir1/hmacs -M dir2 -p 1 hmmlist
```

would be executed. This will load in the HMM definitions in `dir1/hmacs`, process the files listed in `trlist1` and finally dump its accumulators into a file called `HER1.acc` in the output directory `dir2`. At the same time, the command

```
HERest -S trlist2 -I labs -H dir1/hmacs -M dir2 -p 2 hmmlist
```

would be executed on the second workstation, and so on. When HERest has finished on all four workstations, the following command will be executed on just one of them

```
HERest -H dir1/hmacs -M dir2 -p 0 hmmlist dir2/*.acc
```

where the list of training files has been replaced by the dumped accumulator files. This will cause the accumulated statistics to be reloaded and merged so that the model parameters can be reestimated and the new model set output to `dir2` The time to perform this last phase of the operation is very small, hence the whole process will be around four times quicker than for the straightforward sequential case.

8.6 Single-Pass Retraining

In addition to re-estimating the parameters of a HMM set, HERest also provides a mechanism for mapping a set of models trained using one parameterisation into another set based on a different parameterisation. This facility allows the front-end of a HMM-based recogniser to be modified without having to rebuild the models from scratch.

This facility is known as single-pass retraining. Given one set of well-trained models, a new set matching a different training data parameterisation can be generated in a single re-estimation pass. This is done by computing the forward and backward probabilities using the original models together with the original training data, but then switching to the new training data to compute the parameter estimates for the new set of models.

Single-pass retraining is enabled in HERest by setting the `-r` switch. This causes the input training files to be read in pairs. The first of each pair is used to compute the forward/backward probabilities and the second is used to estimate the parameters for the new models. Very often, of course, data input to HTK is modified by the HParm module in accordance with parameters set in a configuration file. In single-pass retraining mode, configuration parameters can be prefixed by the pseudo-module names `HPARM1` and `HPARM2`. Then when reading in the first file of each pair, only the `HPARM1` parameters are used and when reading the second file of each pair, only the `HPARM2` parameters are used.

As an example, suppose that a set of models has been trained on data with `MFCC_E_D` parameterisation and a new set of models using Cepstral Mean Normalisation (`_Z`) is required. These two data parameterisations are specified in a configuration file (`config`) as two separate instances of the configuration variable `TARGETKIND` i.e.

```
# Single pass retraining
HPARM1: TARGETKIND = MFCC_E_D
HPARM2: TARGETKIND = MFCC_E_D_Z
```

HERest would then be invoked with the `-r` option set to enable single-pass retraining. For example,

```
HERest -r -C config -S trainList -I labs -H dir1/hmacs -M dir2 hmmList
```

The script file `trainlist` contains a list of data file pairs. For each pair, the first file should match the parameterisation of the original model set and the second file should match that of the required new set. This will cause the model parameter estimates to be performed using the new set of training data and a new set of models matching this data will be output to `dir2`. This process of single-pass retraining is a significantly faster route to a new set of models than training a fresh set from scratch.

8.7 Two-model Re-Estimation

Another method for initialisation of model parameters implemented in HERest is two-model re-estimation. HMM sets often use the same basic units such as triphones but differ in the way the underlying HMM parameters are tied. In these cases two-model re-estimation can be used to obtain the state-level alignment using one model set which is used to update the parameters of a second model set. This is helpful when the model set to be updated is less well trained.

A typical use of two-model re-estimation is the initialisation of state clustered triphone models. In the standard case triphone models are obtained by cloning of monophone models and subsequent clustering of triphone states. However, the unclustered triphone models are considerably less powerful than state clustered triphone HMMs using mixtures of Gaussians. The consequence is poor state level alignment and thus poor parameter estimates, prior to clustering. This can be ameliorated by the use of well-trained *alignment models* for computing the forward-backward probabilities. In the maximisation stage of the Baum-Welch algorithm the state level posteriors are used to re-restimate the parameters of the *update model set*. Note that the corresponding models in the two sets must have the same number of states.

As an example, suppose that we would like to update a set of cloned single Gaussian monophone models in `dir1/hmacs` using the well trained state-clustered triphones in `dir2/hmacs` as alignment models. Associated with each model set are the model lists `hmmlist1` and `hmmlist2` respectively. In order to use the second model set for alignment a configuration file `config.2model` containing

```
# alignment model set for two-model re-estimation
ALIGNMODELMMF = dir2/hmacs
ALIGNHMMLIST  = hmmlist2
```

is necessary. HERest only needs to be invoked using that configuration file.

```
HERest -C config -C config.2model -S trainlist -I labs -H dir1/hmacs -M dir3 hmmlist1
```

The models in directory `dir1` are updated using the alignment models stored in directory `dir2` and the result is written to directory `dir3`. Note that `trainlist` is a standard HTK script and that the above command uses the capability of HERest to accept multiple configuration files on the command line. If each HMM is stored in a separate file, the configuration variables `ALIGNMODELDIR` and `ALIGNMODELEXT` can be used.

Only the state level alignment is obtained using the alignment models. In the exceptional case that the update model set contains mixtures of Gaussians, component level posterior probabilities are obtained from the update models themselves.

8.8 Parameter Re-Estimation Formulae

For reference purposes, this section lists the various formulae employed within the HTK parameter estimation tools. All are standard, however, the use of non-emitting states and multiple data streams leads to various special cases which are usually not covered fully in the literature.

The following notation is used in this section

N	number of states
S	number of streams
M_s	number of mixture components in stream s
T	number of observations
Q	number of models in an embedded training sequence
N_q	number of states in the q'th model in a training sequence
$\boldsymbol{O}$	a sequence of observations
$\boldsymbol{o}_t$	the observation at time t, $1 \leq t \leq T$
$\boldsymbol{o}_{st}$	the observation vector for stream s at time t
a_{ij}	the probability of a transition from state i to j
c_{jsm}	weight of mixture component m in state j stream s
$\boldsymbol{\mu}_{jsm}$	vector of means for the mixture component m of state j stream s
$\boldsymbol{\Sigma}_{jsm}$	covariance matrix for the mixture component m of state j stream s
λ	the set of all parameters defining a HMM

8.8.1 Viterbi Training (HInit)

In this style of model training, a set of training observations $\boldsymbol{O}^r$, $1 \leq r \leq R$ is used to estimate the parameters of a single HMM by iteratively computing Viterbi alignments. When used to initialise a new HMM, the Viterbi segmentation is replaced by a uniform segmentation (i.e. each training observation is divided into N equal segments) for the first iteration.

Apart from the first iteration on a new model, each training sequence $\boldsymbol{O}$ is segmented using a state alignment procedure which results from maximising

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

for $1 < i < N$ where

$$\phi_j(t) = \left[\max_i \phi_i(t-1) a_{ij}\right] b_j(\boldsymbol{o}_t)$$

with initial conditions given by

$$\phi_1(1) = 1$$

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

for $1 < j < N$. In this and all subsequent cases, the output probability $b_j(\cdot)$ is as defined in equations 7.1 and 7.2 in section 7.1.

If A_{ij} represents the total number of transitions from state i to state j in performing the above maximisations, then the transition probabilities can be estimated from the relative frequencies

$$\hat{a}_{ij} = \frac{A_{ij}}{\sum_{k=2}^{N} A_{ik}}$$

The sequence of states which maximises $\phi_N(T)$ implies an alignment of training data observations with states. Within each state, a further alignment of observations to mixture components is made. The tool HInit provides two mechanisms for this: for each state and each stream

1. use clustering to allocate each observation $\boldsymbol{o}_{st}$ to one of M_s clusters, or
2. associate each observation $\boldsymbol{o}_{st}$ with the mixture component with the highest probability

In either case, the net result is that every observation is associated with a single unique mixture component. This association can be represented by the indicator function $\psi^r_{jsm}(t)$ which is 1 if $\boldsymbol{o}^r_{st}$ is associated with mixture component m of stream s of state j and is zero otherwise.

The means and variances are then estimated via simple averages

$$\hat{\boldsymbol{\mu}}_{jsm} = \frac{\sum_{r=1}^{R}\sum_{t=1}^{T_r}\psi^r_{jsm}(t)\boldsymbol{o}^r_{st}}{\sum_{r=1}^{R}\sum_{t=1}^{T_r}\psi^r_{jsm}(t)}$$

$$\hat{\boldsymbol{\Sigma}}_{jsm} = \frac{\sum_{r=1}^{R}\sum_{t=1}^{T_r}\psi^r_{jsm}(t)(\boldsymbol{o}^r_{st}-\hat{\boldsymbol{\mu}}_{jsm})(\boldsymbol{o}^r_{st}-\hat{\boldsymbol{\mu}}_{jsm})'}{\sum_{r=1}^{R}\sum_{t=1}^{T_r}\psi^r_{jsm}(t)}$$

Finally, the mixture weights are based on the number of observations allocated to each component

$$\boldsymbol{c}_{jsm} = \frac{\sum_{r=1}^{R}\sum_{t=1}^{T_r}\psi^r_{jsm}(t)}{\sum_{r=1}^{R}\sum_{t=1}^{T_r}\sum_{l=1}^{M_s}\psi^r_{jsl}(t)}$$

8.8.2 Forward/Backward Probabilities

Baum-Welch training is similar to the Viterbi training described in the previous section except that the *hard* boundary implied by the ψ function is replaced by a *soft* boundary function L which represents the probability of an observation being associated any given Gaussian mixture component. This *occupation* probability is computed from the *forward* and *backward* probabilities.

For the isolated-unit style of training, the forward probability $\alpha_j(t)$ for $1 < j < N$ and $1 < t \leq T$ is calculated by the forward recursion

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

with initial conditions given by

$$\alpha_1(1) = 1$$

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

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

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

The backward probability $\beta_i(t)$ for $1 < i < N$ and $T > t \geq 1$ is calculated by the backward recursion

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

with initial conditions given by

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

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

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

In the case of embedded training where the HMM spanning the observations is a composite constructed by concatenating Q subword models, it is assumed that at time t, the α and β values corresponding to the entry state and exit states of a HMM represent the forward and backward probabilities at time $t-\Delta t$ and $t+\Delta t$, respectively, where Δt is small. The equations for calculating α and β are then as follows.

For the forward probability, the initial conditions are established at time $t = 1$ as follows

$$\alpha_1^{(q)}(1) = \begin{cases} 1 & \text{if } q = 1 \\ \alpha_1^{(q-1)}(1)a_{1N_{q-1}}^{(q-1)} & \text{otherwise} \end{cases}$$

$$\alpha_j^{(q)}(1) = a_{1j}^{(q)} b_j^{(q)}(\boldsymbol{o}_1)$$

$$\alpha_{N_q}^{(q)}(1) = \sum_{i=2}^{N_q-1} \alpha_i^{(q)}(1)a_{iN_q}^{(q)}$$

where the superscript in parentheses refers to the index of the model in the sequence of concatenated models. All unspecified values of α are zero. For time $t > 1$,

$$\alpha_1^{(q)}(t) = \begin{cases} 0 & \text{if } q = 1 \\ \alpha_{N_{q-1}}^{(q-1)}(t-1) + \alpha_1^{(q-1)}(t)a_{1N_{q-1}}^{(q-1)} & \text{otherwise} \end{cases}$$

$$\alpha_j^{(q)}(t) = \left[\alpha_1^{(q)}(t)a_{1j}^{(q)} + \sum_{i=2}^{N_q-1} \alpha_i^{(q)}(t-1)a_{ij}^{(q)}\right] b_j^{(q)}(\boldsymbol{o}_t)$$

$$\alpha_{N_q}^{(q)}(t) = \sum_{i=2}^{N_q-1} \alpha_i^{(q)}(t)a_{iN_q}^{(q)}$$

For the backward probability, the initial conditions are set at time $t = T$ as follows

$$\beta_{N_q}^{(q)}(T) = \begin{cases} 1 & \text{if } q = Q \\ \beta_{N_{q+1}}^{(q+1)}(T)a_{1N_{q+1}}^{(q+1)} & \text{otherwise} \end{cases}$$

$$\beta_i^{(q)}(T) = a_{iN_q}^{(q)}\beta_{N_q}^{(q)}(T)$$

$$\beta_1^{(q)}(T) = \sum_{j=2}^{N_q-1} a_{1j}^{(q)} b_j^{(q)}(\boldsymbol{o}_T)\beta_j^{(q)}(T)$$

where once again, all unspecified β values are zero. For time $t < T$,

$$\beta_{N_q}^{(q)}(t) = \begin{cases} 0 & \text{if } q = Q \\ \beta_1^{(q+1)}(t+1) + \beta_{N_{q+1}}^{(q+1)}(t)a_{1N_{q+1}}^{(q+1)} & \text{otherwise} \end{cases}$$

$$\beta_i^{(q)}(t) = a_{iN_q}^{(q)}\beta_{N_q}^{(q)}(t) + \sum_{j=2}^{N_q-1} a_{ij}^{(q)} b_j^{(q)}(\boldsymbol{o}_{t+1})\beta_j^{(q)}(t+1)$$

$$\beta_1^{(q)}(t) = \sum_{j=2}^{N_q-1} a_{1j}^{(q)} b_j^{(q)}(\boldsymbol{o}_t)\beta_j^{(q)}(t)$$

The total probability $P = \text{prob}(\boldsymbol{O}|\lambda)$ can be computed from either the forward or backward probabilities

$$P = \alpha_N(T) = \beta_1(1)$$

8.8.3 Single Model Reestimation(HRest)

In this style of model training, a set of training observations $\boldsymbol{O}^r,\ 1 \leq r \leq R$ is used to estimate the parameters of a single HMM. The basic formula for the reestimation of the transition probabilities is

$$\hat{a}_{ij} = \frac{\sum_{r=1}^{R} \frac{1}{P_r} \sum_{t=1}^{T_r-1} \alpha_i^r(t) a_{ij} b_j(\boldsymbol{o}_{t+1}^r) \beta_j^r(t+1)}{\sum_{r=1}^{R} \frac{1}{P_r} \sum_{t=1}^{T_r} \alpha_i^r(t) \beta_i^r(t)}$$

where $1 < i < N$ and $1 < j < N$ and P_r is the total probability $P = \text{prob}(\boldsymbol{O}^r|\lambda)$ of the r'th observation. The transitions from the non-emitting entry state are reestimated by

$$\hat{a}_{1j} = \frac{1}{R} \sum_{r=1}^{R} \frac{1}{P_r} \alpha_j^r(1) \beta_j^r(1)$$

where $1 < j < N$ and the transitions from the emitting states to the final non-emitting exit state are reestimated by

$$\hat{a}_{iN} = \frac{\sum_{r=1}^{R} \frac{1}{P_r} \alpha_i^r(T) \beta_i^r(T)}{\sum_{r=1}^{R} \frac{1}{P_r} \sum_{t=1}^{T_r} \alpha_i^r(t) \beta_i^r(t)}$$

where $1 < i < N$.

For a HMM with M_s mixture components in stream s, the means, covariances and mixture weights for that stream are reestimated as follows. Firstly, the probability of occupying the m'th mixture component in stream s at time t for the r'th observation is

$$L_{jsm}^r(t) = \frac{1}{P_r} U_j^r(t) c_{jsm} b_{jsm}(\boldsymbol{o}_{st}^r) \beta_j^r(t) b_{js}^*(\boldsymbol{o}_t^r)$$

where

$$U_j^r(t) = \begin{cases} a_{1j} & \text{if } t = 1 \\ \sum_{i=2}^{N-1} \alpha_i^r(t-1) a_{ij} & \text{otherwise} \end{cases} \tag{8.1}$$

and

$$b_{js}^*(\boldsymbol{o}_t^r) = \prod_{k \neq s} b_{jk}(\boldsymbol{o}_{kt}^r)$$

For single Gaussian streams, the probability of mixture component occupancy is equal to the probability of state occupancy and hence it is more efficient in this case to use

$$L_{jsm}^r(t) = L_j^r(t) = \frac{1}{P_r} \alpha_j(t) \beta_j(t)$$

Given the above definitions, the re-estimation formulae may now be expressed in terms of $L_{jsm}^r(t)$ as follows.

$$\begin{aligned} \hat{\boldsymbol{\mu}}_{jsm} &= \frac{\sum_{r=1}^{R} \sum_{t=1}^{T_r} L_{jsm}^r(t) \boldsymbol{o}_{st}^r}{\sum_{r=1}^{R} \sum_{t=1}^{T_r} L_{jsm}^r(t)} \\ \hat{\boldsymbol{\Sigma}}_{jsm} &= \frac{\sum_{r=1}^{R} \sum_{t=1}^{T_r} L_{jsm}^r(t) (\boldsymbol{o}_{st}^r - \hat{\boldsymbol{\mu}}_{jsm})(\boldsymbol{o}_{st}^r - \hat{\boldsymbol{\mu}}_{jsm})'}{\sum_{r=1}^{R} \sum_{t=1}^{T_r} L_{jsm}^r(t)} \\ \boldsymbol{c}_{jsm} &= \frac{\sum_{r=1}^{R} \sum_{t=1}^{T_r} L_{jsm}^r(t)}{\sum_{r=1}^{R} \sum_{t=1}^{T_r} L_j^r(t)} \end{aligned} \tag{8.2}$$

8.8.4 Embedded Model Reestimation(HERest)

The re-estimation formulae for the embedded model case have to be modified to take account of the fact that the entry states can be occupied at any time as a result of transitions out of the previous model. The basic formulae for the re-estimation of the transition probabilities is

$$\hat{a}_{ij}^{(q)} = \frac{\sum_{r=1}^{R} \frac{1}{P_r} \sum_{t=1}^{T_r-1} \alpha_i^{(q)r}(t) a_{ij}^{(q)} b_j^{(q)}(\boldsymbol{o}_{t+1}^r) \beta_j^{(q)r}(t+1)}{\sum_{r=1}^{R} \frac{1}{P_r} \sum_{t=1}^{T_r} \alpha_i^{(q)r}(t) \beta_i^{(q)r}(t)}$$

The transitions from the non-emitting entry states into the HMM are re-estimated by

$$\hat{a}_{1j}^{(q)} = \frac{\sum_{r=1}^{R} \frac{1}{P_r} \sum_{t=1}^{T_r-1} \alpha_1^{(q)r}(t) a_{1j}^{(q)} b_j^{(q)}(\boldsymbol{o}_t^r) \beta_j^{(q)r}(t)}{\sum_{r=1}^{R} \frac{1}{P_r} \sum_{t=1}^{T_r} \alpha_1^{(q)r}(t) \beta_1^{(q)r}(t) + \alpha_1^{(q)r}(t) a_{1N_q}^{(q)} \beta_1^{(q+1)r}(t)}$$

and the transitions out of the HMM into the non-emitting exit states are re-estimated by

$$\hat{a}_{iN_q}^{(q)} = \frac{\sum_{r=1}^{R} \frac{1}{P_r} \sum_{t=1}^{T_r-1} \alpha_i^{(q)r}(t) a_{iN_q}^{(q)} \beta_{N_q}^{(q)r}(t)}{\sum_{r=1}^{R} \frac{1}{P_r} \sum_{t=1}^{T_r} \alpha_i^{(q)r}(t) \beta_i^{(q)r}(t)}$$

Finally, the direct transitions from non-emitting entry to non-emitting exit states are re-estimated by

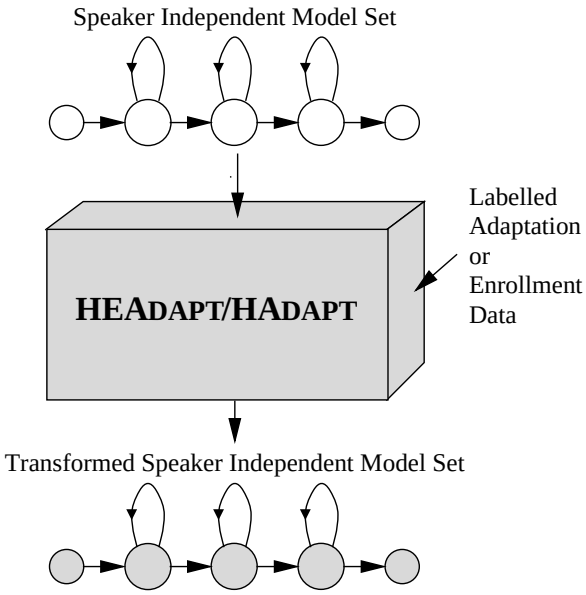
$$\hat{a}_{1N_q}^{(q)} = \frac{\sum_{r=1}^{R} \frac{1}{P_r} \sum_{t=1}^{T_r-1} \alpha_1^{(q)r}(t) a_{1N_q}^{(q)} \beta_1^{(q+1)r}(t)}{\sum_{r=1}^{R} \frac{1}{P_r} \sum_{t=1}^{T_r} \alpha_i^{(q)r}(t) \beta_i^{(q)r}(t) + \alpha_1^{(q)r}(t) a_{1N_q}^{(q)} \beta_1^{(q+1)r}(t)}$$

The re-estimation formulae for the output distributions are the same as for the single model case except for the obvious additional subscript for q. However, the probability calculations must now allow for transitions from the entry states by changing $U_j^r(t)$ in equation 8.1 to

$$U_j^{(q)r}(t) = \begin{cases} \alpha_1^{(q)r}(t) a_{1j}^{(q)} & \text{if } t = 1 \\ \alpha_1^{(q)r}(t) a_{1j}^{(q)} + \sum_{i=2}^{N_q-1} \alpha_i^{(q)r}(t-1) a_{ij}^{(q)} & \text{otherwise} \end{cases}$$

Chapter 9

HMM Adaptation



Chapter 8 described how the parameters are estimated for plain continuous density HMMs within HTK, primarily using the embedded training tool HEREST. Using the training strategy depicted in figure 8.2, together with other techniques can produce high performance speaker independent acoustic models for a large vocabulary recognition system. However it is possible to build improved acoustic models by tailoring a model set to a specific speaker. By collecting data from a speaker and training a model set on this speaker's data alone, the speaker's characteristics can be modelled more accurately. Such systems are commonly known as *speaker dependent* systems, and on a typical word recognition task, may have half the errors of a speaker independent system. The drawback of speaker dependent systems is that a large amount of data (typically hours) must be collected in order to obtain sufficient model accuracy.

Rather than training speaker dependent models, *adaptation* techniques can be applied. In this case, by using only a small amount of data from a new speaker, a good speaker independent system model set can be adapted to better fit the characteristics of this new speaker.

Speaker adaptation techniques can be used in various different modes. If the true transcription of the adaptation data is known then it is termed *supervised adaptation*, whereas if the adaptation data is unlabelled then it is termed *unsupervised adaptation*. In the case where all the adaptation data is available in one block, e.g. from a speaker enrollment session, then this termed *static adaptation*. Alternatively adaptation can proceed incrementally as adaptation data becomes available, and this is termed *incremental adaptation*.

HTK provides two tools to adapt continuous density HMMs. HEADAPT performs offline supervised adaptation using maximum likelihood linear regression (MLLR) and/or maximum a-posteriori (MAP) adaptation, while unsupervised adaptation is supported by HVITE (using only MLLR). In

this case HVite not only performs recognition, but simultaneously adapts the model set as the data becomes available through recognition. Currently, MLLR adaptation can be applied in both incremental and static modes while MAP supports only static adaptation. If MLLR and MAP adaptation is to be performed simultaneously using HEAdapt in the same pass, then the restriction is that the entire adaptation must be performed statically[1].

This chapter describes the supervised adaptation tool HEAdapt. The first sections of the chapter give an overview of MLLR and MAP adaptation and this is followed by a section describing the general usages of HEAdapt to build simple and more complex adapted systems. The chapter concludes with a section detailing the various formulae used by the adaptation tool. The use of HVite to perform unsupervised adaptation is discussed in section 13.6.2.

9.1 Model Adaptation using MLLR

9.1.1 Maximum Likelihood Linear Regression

Maximum likelihood linear regression or MLLR computes a set of transformations that will reduce the mismatch between an initial model set and the adaptation data[2]. More specifically MLLR is a model adaptation technique that estimates a set of linear transformations for the mean and variance parameters of a Gaussian mixture HMM system. The effect of these transformations is to shift the component means and alter the variances in the initial system so that each state in the HMM system is more likely to generate the adaptation data. Note that due to computational reasons, MLLR is only implemented within HTK for diagonal covariance, single stream, continuous density HMMs.

The transformation matrix used to give a new estimate of the adapted mean is given by

$$\hat{\boldsymbol{\mu}} = \boldsymbol{W}\boldsymbol{\xi}, \tag{9.1}$$

where $\boldsymbol{W}$ is the $n \times (n+1)$ transformation matrix (where n is the dimensionality of the data) and $\boldsymbol{\xi}$ is the extended mean vector,

$$\boldsymbol{\xi} = [\ w\ \mu_1\ \mu_2\ \ \ldots\ \ \mu_n\ ]^T$$

where w represents a bias offset whose value is fixed (within HTK) at 1.
Hence $\boldsymbol{W}$ can be decomposed into

$$\boldsymbol{W} = [\ \boldsymbol{b}\ \boldsymbol{A}\ ] \tag{9.2}$$

where $\boldsymbol{A}$ represents an $n \times n$ transformation matrix and $\boldsymbol{b}$ represents a bias vector.

The transformation matrix $\boldsymbol{W}$ is obtained by solving a maximisation problem using the *Expectation-Maximisation* (EM) technique. This technique is also used to compute the variance transformation matrix. Using EM results in the maximisation of a standard *auxiliary function*. (Full details are available in section 9.4.)

9.1.2 MLLR and Regression Classes

This adaptation method can be applied in a very flexible manner, depending on the amount of adaptation data that is available. If a small amount of data is available then a *global* adaptation transform can be generated. A global transform (as its name suggests) is applied to every Gaussian component in the model set. However as more adaptation data becomes available, improved adaptation is possible by increasing the number of transformations. Each transformation is now more specific and applied to certain groupings of Gaussian components. For instance the Gaussian components could be grouped into the broad phone classes: silence, vowels, stops, glides, nasals, fricatives, etc. The adaptation data could now be used to construct more specific broad class transforms to apply to these groupings.

Rather than specifying static component groupings or classes, a robust and dynamic method is used for the construction of further transformations as more adaptation data becomes available. MLLR makes use of a *regression class tree* to group the Gaussians in the model set, so that the set of transformations to be estimated can be chosen according to the amount and type of adaptation data that is available. The tying of each transformation across a number of mixture components

[1] By using two passes, one could perform incremental MLLR in the first pass (saving the new model or transform), followed by a second pass, this time using MAP adaptation.

[2] MLLR can also be used to perform environmental compensation by reducing the mismatch due to channel or additive noise effects.

makes it possible to adapt distributions for which there were no observations at all. With this process all models can be adapted and the adaptation process is dynamically refined when more adaptation data becomes available.

The regression class tree is constructed so as to cluster together components that are close in acoustic space, so that similar components can be transformed in a similar way. Note that the tree is built using the original speaker independent model set, and is thus independent of any new speaker. The tree is constructed with a centroid splitting algorithm, which uses a Euclidean distance measure. For more details see section 10.7. The terminal nodes or leaves of the tree specify the final component groupings, and are termed the *base (regression) classes*. Each Gaussian component of a model set belongs to one particular base class. The tool HHEd can be used to build a binary regression class tree, and to label each component with a base class number. Both the tree and component base class numbers are saved automatically as part of the MMF. Please refer to section 7.9 and section 10.7 for further details.

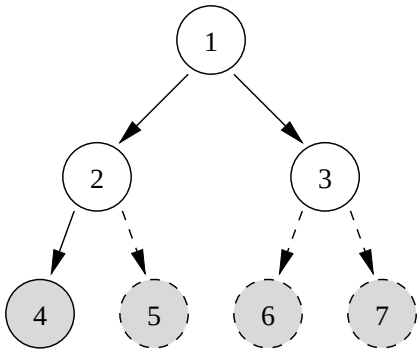


Fig. 9.1 A binary regression tree

Figure 9.1 shows a simple example of a binary regression tree with four base classes, denoted as $\{C_4, C_5, C_6, C_7\}$. During "dynamic" adaptation, the occupation counts are accumulated for each of the regression base classes. The diagram shows a solid arrow and circle (or node), indicating that there is sufficient data for a transformation matrix to be generated using the data associated with that class. A dotted line and circle indicates that there is insufficient data. For example neither node 6 or 7 has sufficient data; however when pooled at node 3, there is sufficient adaptation data. The amount of data that is "determined" as sufficient is set by the user as a command-line option to HEAdapt (see reference section 17.6).

HEAdapt uses a top-down approach to traverse the regression class tree. Here the search starts at the root node and progresses down the tree generating transforms only for those nodes which

1. have sufficient data **and**
2. are either terminal nodes (i.e. base classes) **or** have any children without sufficient data.

In the example shown in figure 9.1, transforms are constructed only for regression nodes 2, 3 and 4, which can be denoted as $\mathbf{W}_2$, $\mathbf{W}_3$ and $\mathbf{W}_4$. Hence when the transformed model set is required, the transformation matrices (mean and variance) are applied in the following fashion to the Gaussian components in each base class:-

$$\left\{ \begin{array}{lcl} \mathbf{W}_2 & \rightarrow & \{C_5\} \\ \mathbf{W}_3 & \rightarrow & \{C_6, C_7\} \\ \mathbf{W}_4 & \rightarrow & \{C_4\} \end{array} \right\}$$

At this point it is interesting to note that the global adaptation case is the same as a tree with just a root node, and is in fact treated as such.

9.1.3 Transform Model File Format

HEAdapt estimates the required transformation statistics and can either output a transformed MMF or a transform model file (TMF). The advantage in storing the transforms as opposed to an adapted MMF is that the TMFs are considerably smaller than MMFs (especially triphone MMFs). This section describes the format of the transform model file in detail.

The mean transformation matrix is stored as a block diagonal transformation matrix. The example block diagonal matrix **A** shown below contains three blocks. The first block represents the transformation for only the static components of the feature vector, while the second represents the deltas and the third the accelerations. This block diagonal matrix example makes the assumption that for the transformation, there is no correlation between the statics, deltas and delta deltas. In practice this assumption works quite well.

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_s & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_\Delta & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{A}_{\Delta^2} \end{pmatrix}$$

This format reduces the number of transformation parameters required to be learnt, making the adaptation process faster. It also reduces the adaptation data required per transform when compared with the full case. When comparing the storage requirements, the 3 block diagonal matrix requires much less storage capacity than the full transform matrix. Note that for convenience a full transformation matrix is also stored as a block diagonal matrix, only in this case there is a single block.

The variance transformation is a diagonal matrix and as such is simply stored as a vector. Figure 9.2 shows a simple example of a TMF. In this case the feature vector has nine dimensions, and the mean transform has three diagonal blocks. The TMF can be saved in ASCII or binary format. The user header is always output in ascii. The first two fields are speaker descriptor fields. The next field `<MMFID>`, the MMF identifier, is obtained from the global options macro in the MMF, while the regression class tree identifier `<RCID>` is obtained from the regression tree macro name in the MMF. If global adaptation is being performed, then the `<RCID>` will contain the identifier `global`, since a tree is unnecessary in the global case. Note that the MMF and regression class tree identifiers are set within the MMF using the tool HHEd. The final two fields are optional, but HEAdapt outputs these anyway for the user's convenience. These can be edited at any time (as can all the fields if desired, but editing `<MMFID>` and `<RCID>` fields should be avoided). The `<CHAN>` field should represent the adaptation data recording environment. Examples could be a particular microphone name, telephone channel or various background noise conditions. The `<DESC>` allow the user to enter any other information deemed useful. An example could be the speaker's dialect region.

```
<UID> djk
<NAME> Dan Kershaw
<MMFID> ECRL_UK_XWRD
<RCID> global
<CHAN> Standard
<DESC> None
<NBLOCKS> 3
<NODETHRESH> 700.0
<NODEOCC> 1 24881.8
<TRANSFORM> 1
   <MEAN_TR> 3
      <BLOCK> 1
          0.942 -0.032 -0.001
         -0.102  0.922 -0.015
         -0.016  0.045  0.910
      <BLOCK> 2
          1.021 -0.032 -0.011
         -0.017  1.074 -0.043
         -0.099  0.091  1.050
      <BLOCK> 3
          1.028  0.032  0.001
         -0.012  1.014 -0.011
         -0.091 -0.043  1.041
   <BIASOFFSET> 9
     -0.357  0.001 -0.002  0.132  0.072
      0.006  0.150  0.138  0.198
   <VARIANCE_TR> 9
      0.936  0.865  0.848  0.832  0.829
      0.786  0.947  0.869  0.912
```

Fig. 9.2 A Simple example of a TMF

Whenever a TMF is being used (in conjunction with an MMF), the MMF identifier in the MMF is checked against that in the TMF. These **must** match since the TMF is dependent on the model set it was constructed from. Also unless the `<RCID>` field is set to `global`, it is also checked for consistency against the regression tree identifier in the MMF.

The rest of the TMF contains a further information header, followed by all the transforms. The information header contains necessary transform set information such as the number of blocks used, node occupation threshold used, and the node occupation counts. Each transform has a regression class identifier number, the mean transformation matrix $\mathbf{A}$, an optional bias vector $\mathbf{b}$ (as in equation 9.2) and an optional variance transformation diagonal matrix $\mathbf{H}$ (stored as a vector). The example has both a bias offset and a variance transform.

9.2 Model Adaptation using MAP

Model adaptation can also be accomplished using a maximum a posteriori (MAP) approach. This adaptation process is sometimes referred to as Bayesian adaptation. MAP adaptation involves the use of prior knowledge about the model parameter distribution. Hence, if we know what the parameters of the model are likely to be (before observing any adaptation data) using the prior knowledge, we might well be able to make good use of the limited adaptation data, to obtain a decent MAP estimate. This type of prior is often termed an informative prior. Note that if the prior distribution indicates no preference as to what the model parameters are likely to be (a non-informative prior), then the MAP estimate obtained will be identical to that obtained using a maximum likelihood approach.

For MAP adaptation purposes, the informative priors that are generally used are the speaker independent model parameters. For mathematical tractability conjugate priors are used, which

results in a simple adaptation formula. The update formula for a single stream system for state j and mixture component m is

$$\hat{\boldsymbol{\mu}}_{jm} = \frac{N_{jm}}{N_{jm}+\tau}\bar{\boldsymbol{\mu}}_{jm} + \frac{\tau}{N_{jm}+\tau}\boldsymbol{\mu}_{jm} \tag{9.3}$$

where τ is a weighting of the a priori knowledge to the adaptation speech data and N is the occupation likelihood of the adaptation data, defined as,

$$N_{jm} = \sum_{r=1}^{R}\sum_{t=1}^{T_r} L_{jm}^{r}(t)$$

where $\boldsymbol{\mu}_{jm}$ is the speaker independent mean and $\bar{\boldsymbol{\mu}}_{jm}$ is the mean of the observed adaptation data and is defined as,

$$\bar{\boldsymbol{\mu}}_{jm} = \frac{\sum_{r=1}^{R}\sum_{t=1}^{T_r} L_{jm}^{r}(t)\boldsymbol{o}_t^r}{\sum_{r=1}^{R}\sum_{t=1}^{T_r} L_{jm}^{r}(t)}$$

As can be seen, if the occupation likelihood of a Gaussian component (N_{jm}) is small, then the mean MAP estimate will remain close to the speaker independent component mean. With MAP adaptation, every single mean component in the system is updated with a MAP estimate, based on the prior mean, the weighting and the adaptation data. Hence, MAP adaptation requires a new "speaker-dependent" model set to be saved.

One obvious drawback to MAP adaptation is that it requires more adaptation data to be effective when compared to MLLR, because MAP adaptation is specifically defined at the component level. When larger amounts of adaptation training data become available, MAP begins to perform better than MLLR, due to this detailed update of each component (rather than the pooled Gaussian transformation approach of MLLR). In fact the two adaptation processes can be combined to improve performance still further, by using the MLLR transformed means as the priors for MAP adaptation (by replacing $\boldsymbol{\mu}_{jm}$ in equation 9.3 with the transformed mean of equation 9.1). In this case components that have a low occupation likelihood in the adaptation data, (and hence would not change much using MAP alone) have been adapted using a regression class transform in MLLR. An example usage is shown in the following section.

9.3 Using HEAdapt

At the outset HEAdapt operates in a very similar fashion to HERest. Both use a frame/state alignment in order to accumulate various statistics about the data. In HERest these statistics are used to estimate new model parameters whilst in HEAdapt they are used to estimate the transformations for each regression base class, or new model parameters. HEAdapt will currently only produce transforms with single stream data and `PLAINHS` or `SHAREDHS` HMM systems (see section 7.4 on HMM set kinds).

In outline, HEAdapt works as follows. On startup, HEAdapt loads in a complete set of HMM definitions, including, the regression class tree and the base class number of each Gaussian component. Note that HEAdapt requires the MMF to contain a regression class tree. Every training file must have an associated label file which gives a transcription for that file. Only the sequence of labels is used by HEAdapt, and any boundary location information is ignored. Thus, these transcriptions can be generated automatically from the known orthography of what was said and a pronunciation dictionary.

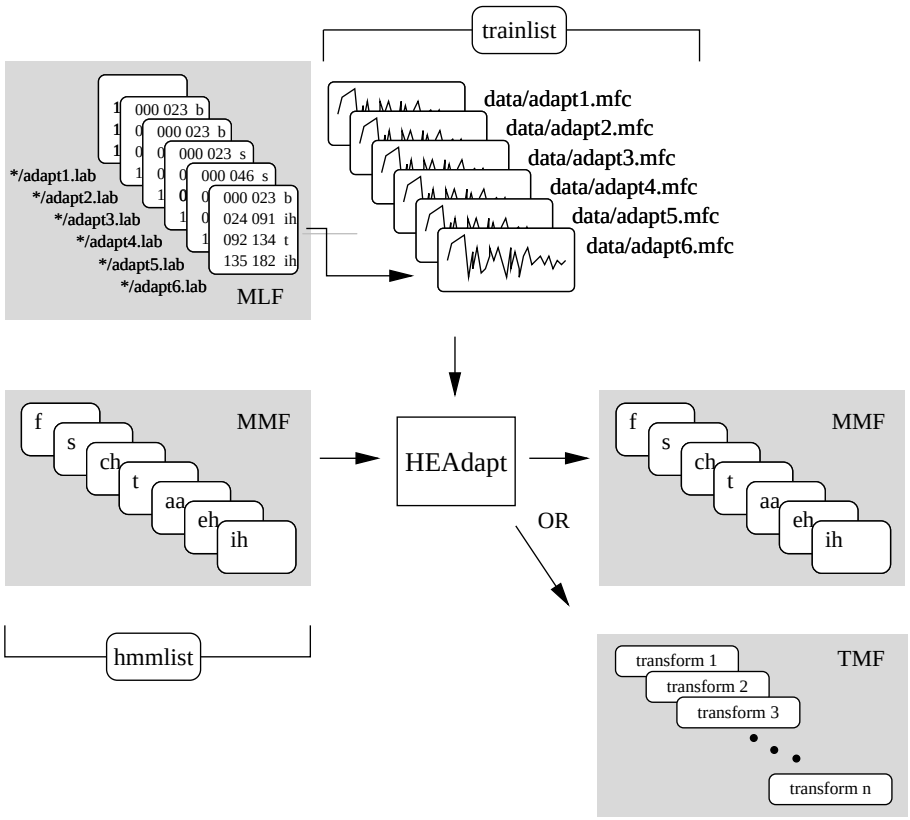


Fig. 9.3 File Processing in HEAdapt

HEAdapt processes each training file in turn. After loading it into memory, it uses the associated transcription to construct a composite HMM which spans the whole utterance. This composite HMM is made by concatenating instances of the phone HMMs corresponding to each label in the transcription. The Forward-Backward algorithm is then applied to obtain a frame/state alignment and the information necessary to form the standard auxiliary function is accumulated at the Gaussian component level. Note that this information is different from that required in HERest (see section 9.4). When all of the training files have been processed (within the static or incremental block), the regression base class statistics are accumulated using the component level statistics. Next the regression class tree is traversed and the new regression class transformations are calculated for those regression classes containing a sufficient occupation count at the lowest level in the tree, as described in section 9.1.2. Finally either the updated (i.e. adapted) HMM set or the transformations are output. Note that HEAdapt produces a transforms model file (TMF) that contains

transforms that are estimated to *transform from the input MMF* to a new environment/speaker based on the adaptation data presented.

The mathematical details of the Forward-Backward algorithm are given in section 8.8, while the mathematical details for the MLLR mean and variance transformation calculations can be found in section 9.4.

HEAdapt is typically invoked by a command line of the form

```
HEAdapt -S adaptlist -I labs -H dir1/hmacs -M dir2 hmmlist
```

where `hmmlist` contains the list of HMMs.

Once all MMFs and MLFs have been loaded, HEAdapt processes each file in the `adaptlist`, and accumulates the required statistics as described above. On completion, an updated MMF is output to the directory `dir2`.

If the following form of the command is used

```
HEAdapt -S adaptlist -I labs -H dir1/hmacs -K dir2/tmf hmmlist
```

then on completion a transform model file (TMF) `tmf` is output to the directory `dir2`. This process is illustrated by Fig 9.3. Section 9.1.3 describes the TMF format in more detail. The output `tmf` contains transforms that transform the MMF `hmacs`. Once this is saved, HVite can be used to perform recognition for the adapted speaker either using a transformed MMF or by using the speaker independent MMF together with a speaker specific TMF.

HEAdapt employs the same pruning mechanism as HERest during the forward-backward computation. As such the pruning on the backward path is under the user's control, and the beam is set using the `-t` option.

HEAdapt can also be run several times in block or static fashion. For instance a first pass might entail a global adaptation (forced using the `-g` option), producing the TMF `global.tmf` by invoking

```
HEAdapt -g -S adaptlist -I labs -H mmf -K tmfs/global.tmf \
        hmmlist
```

The second pass could load in the global transformations (and tranform the model set) using the `-J` option, performing a better frame/state alignment than the speaker independent model set, and output a set of regression class transformations,

```
HEAdapt -S adaptlist -I labs -H mmf -K tmfs/rc.tmf \
        -J tmfs/global.tmf hmmlist
```

Note again that the number of transformations is selected automatically and is dependent on the node occupation threshold setting and the amount of adaptation data available. Finally when producing a TMF, HEAdapt always generates a TMF to transform the input MMF in all cases. In the last example the input MMF is transformed by the global transform file `global.tmf` in order to obtain the frame/state alignment only. The final TMF that is output, `rc.tmf`, contains the set of transforms to transform the input MMF `mmf`, based on this frame/state alignment.

As an alternative, the second pass could entail MLLR together with MAP adaptation, outputing a new model set. Note that with MAP adaptation a transform can not be saved and a full HMM set must be output.

```
HEAdapt -S adaptlist -I labs -H mmf -M dir2 -k -j 12.0
        -J tmfs/global.tmf hmmlist
```

Note that MAP alone could be used by removing the `-k` option. The argument to the `-j` option represents the MAP adaptation scaling factor.

9.4 MLLR Formulae

For reference purposes, this section lists the various formulae employed within the HTK adaptation tool. It is assumed throughout that single stream data is used and that diagonal covariances are also used. All are standard and can be found in various literature.

The following notation is used in this section

$\mathcal{M}$	the model set
T	number of observations
m	a mixture component
$\boldsymbol{O}$	a sequence of observations
$\boldsymbol{o}(t)$	the observation at time t, $1 \leq t \leq T$
$\boldsymbol{\mu}_{m_r}$	mean vector for the mixture component m_r
$\boldsymbol{\xi}_{m_r}$	extended mean vector for the mixture component m_r
$\boldsymbol{\Sigma}_{m_r}$	covariance matrix for the mixture component m_r
$L_{m_r}(t)$	the occupancy probability for the mixture component m_r at time t

9.4.1 Estimation of the Mean Transformation Matrix

To enable robust transformations to be trained, the transform matrices are tied across a number of Gaussians. The set of Gaussians which share a transform is referred to as a regression class. For a particular transform case $\boldsymbol{W_m}$, the R Gaussian components $\{m_1, m_2, \ldots, m_R\}$ will be tied together, as determined by the regression class tree (see section 9.1.2). By formulating the standard auxiliary function, and then maximising it with respect to the transformed mean, and considering only these tied Gaussian components, the following is obtained,

$$\sum_{t=1}^{T}\sum_{r=1}^{R} L_{m_r}(t)\boldsymbol{\Sigma}_{m_r}^{-1}\boldsymbol{o}(t)\boldsymbol{\xi}_{m_r}^{T} = \sum_{t=1}^{T}\sum_{r=1}^{R} L_{m_r}(t)\boldsymbol{\Sigma}_{m_r}^{-1}\boldsymbol{W_m}\boldsymbol{\xi}_{m_r}\boldsymbol{\xi}_{m_r}^{T} \tag{9.4}$$

and $L_{m_r}(t)$, the occupation likelihood, is defined as,

$$L_{m_r}(t) = p(q_{m_r}(t) \mid \mathcal{M}, \boldsymbol{O}_T)$$

where $q_{m_r}(t)$ indicates the Gaussian component m_r at time t, and $\boldsymbol{O}_T = \{\boldsymbol{o}(1), \ldots, \boldsymbol{o}(T)\}$ is the adaptation data. The occupation likelihood is obtained from the forward-backward process described in section 8.8.

To solve for $\boldsymbol{W}_m$, two new terms are defined.

1. The left hand side of equation 9.4 is independent of the transformation matrix and is referred to as $\boldsymbol{Z}$, where
$$\boldsymbol{Z} = \sum_{t=1}^{T}\sum_{r=1}^{R} L_{m_r}(t)\boldsymbol{\Sigma}_{m_r}^{-1}\boldsymbol{o}(t)\boldsymbol{\xi}_{m_r}^{T}$$

2. A new variable $\boldsymbol{G}^{(i)}$ is defined with elements
$$g_{jq}^{(i)} = \sum_{r=1}^{R} v_{ii}^{(r)} d_{jq}^{(r)}$$
where
$$\boldsymbol{V}^{(r)} = \sum_{t=1}^{T} L_{m_r}(t)\boldsymbol{\Sigma}_{m_r}^{-1}$$
and
$$\boldsymbol{D}^{(r)} = \boldsymbol{\xi}_{m_r}\boldsymbol{\xi}_{m_r}^{T}$$

It can be seen that from these two new terms, $\boldsymbol{W}_m$ can be calculated from

$$\boldsymbol{w}_i^{T} = \boldsymbol{G}_i^{-1}\boldsymbol{z}_i^{T}$$

where $\boldsymbol{w}_i$ is the i^{th} vector of $\boldsymbol{W}_m$ and $\boldsymbol{z}_i$ is the i^{th} vector of $\boldsymbol{Z}$.

The regression class tree is used to generate the classes dynamically, so it is not known a-priori which regression classes will be used to estimate the transform. This does not present a problem, since $\boldsymbol{G}^{(i)}$ and $\boldsymbol{Z}$ for the chosen regression class may be obtained from its child classes (as defined by the tree). If the parent node R has children $\{R_1, \ldots, R_C\}$ then

$$\boldsymbol{Z} = \sum_{c=1}^{C} \boldsymbol{Z}^{(R_c)}$$

and

$$\boldsymbol{G}^{(i)} = \sum_{c=1}^{C} \boldsymbol{G}^{(iR_c)}$$

From this it is clear that it is only necessary to calculate $\boldsymbol{G}^{(i)}$ and $\boldsymbol{Z}$ for only the most specific regression classes possible – i.e. the base classes.

9.4.2 Estimation of the Variance Transformation Matrix

Estimation of the variance transformation matrices is only available for diagonal covariance Gaussian systems. The Gaussian covariance is transformed using,

$$\hat{\boldsymbol{\Sigma}}_m = \boldsymbol{B}_m^T \boldsymbol{H}_m \boldsymbol{B}_m$$

where $\boldsymbol{H}_m$ is the linear transformation to be estimated and $\boldsymbol{B}_m$ is the inverse of the Choleski factor of $\boldsymbol{\Sigma}_m^{-1}$, so

$$\boldsymbol{\Sigma}_m^{-1} = \boldsymbol{C}_m \boldsymbol{C}_m^T$$

and

$$\boldsymbol{B}_m = \boldsymbol{C}_m^{-1}$$

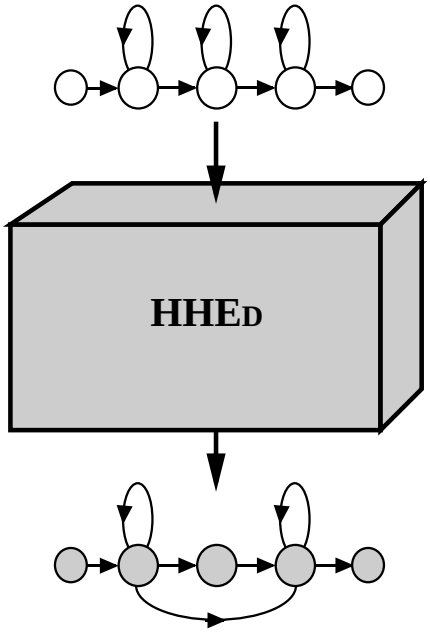
After rewriting the auxiliary function, the transform matrix $\boldsymbol{H}_m$ is estimated from,

$$\boldsymbol{H}_m = \frac{\sum_{r=1}^{R_c} \boldsymbol{C}_{m_r}^T \left[L_{m_r}(t)(\boldsymbol{o}(t) - \boldsymbol{\mu}_{m_r})(\boldsymbol{o}(t) - \boldsymbol{\mu}_{m_r})^T \right] \boldsymbol{C}_{m_r}}{L_{m_r}(t)}$$

Here, $\boldsymbol{H}_m$ is forced to be a diagonal transformation by setting the off-diagonal terms to zero, which ensures that $\hat{\boldsymbol{\Sigma}}_m$ is also diagonal.

Chapter 10

HMM System Refinement



In chapter 8, the basic processes involved in training a continuous density HMM system were explained and examples were given of building a set of HMM phone models. In the practical application of these techniques to building real systems, there are often a number of problems to overcome. Most of these arise from the conflicting desire to have a large number of model parameters in order to achieve high accuracy, whilst at the same time having limited and uneven training data.

As mentioned previously, the HTK philosophy is to build systems incrementally. Starting with a set of context-independent monophone HMMs, a system can be refined in a sequence of stages. Each refinement step typically uses the HTK HMM definition editor HHEd followed by re-estimation using HERest. These incremental manipulations of the HMM set often involve parameter tying, thus many of HHEd's operations involve generating new macro definitions.

The principle types of manipulation that can be performed by HHEd are

- HMM cloning to form context-dependent model sets
- Generalised parameter tying
- Data driven and decision tree based clustering.
- Mixture component splitting
- Adding/removing state transitions
- Stream splitting, resizing and recasting

This chapter describes how the HTK tool HHEd is used, its editing language and the main operations that can be performed.

10.1 Using HHEd

The HMM editor HHEd takes as input a set of HMM definitions and outputs a new modified set, usually to a new directory. It is invoked by a command line of the form

```
HHEd -H MMF1 -H MMF2 ... -M newdir cmds.hed hmmlist
```

where `cmds.hed` is an edit script containing a list of edit commands. Each command is written on a separate line and begins with a 2 letter command name.

The effect of executing the above command line would be to read in the HMMs listed in `hmmlist` and defined by files `MMF1`, `MMF2`, etc., apply the editing operations defined in `cmds.hed` and then write the resulting system out to the directory `newdir`. As with all tools, HTK will attempt to replicate the file structure of the input in the output directory. By default, any new macros generated by HHEd will be written to one or more of the existing MMFs. In doing this, HTK will attempt to ensure that the "definition before use" rule for macros is preserved, but it cannot always guarantee this. Hence, it is usually best to define explicit target file names for new macros. This can be done in two ways. Firstly, explicit target file names can be given in the edit script using the `UF` command. For example, if `cmds.hed` contained

```
....
UF smacs
# commands to generate state macros
....
UF vmacs
# commands to generate variance macros
....
```

then the output directory would contain an MMF called `smacs` containing a set of state macro definitions and an MMF called `vmacs` containing a set of variance macro definitions, these would be in addition to the existing MMF files `MMF1`, `MMF2`, etc.

Alternatively, the whole HMM system can be written to a single file using the `-w` option. For example,

```
HHEd -H MMF1 -H MMF2 ... -w newMMF cmds.hed hmmlist
```

would write the whole of the edited HMM set to the file `newMMF`.

As mentioned previously, each execution of HHEd is normally followed by re-estimation using HERest. Normally, all the information needed by HHEd is contained in the model set itself. However, some clustering operations require various statistics about the training data (see sections 10.4 and 10.5). These statistics are gathered by HERest and output to a *stats file*, which is then read in by HHEd. Note, however, that the statistics file generated by HERest refers to the input model set not the re-estimated set. Thus for example, in the following sequence, the HHEd edit script in `cmds.hed` contains a command (see the `RO` command in section 10.4) which references a statistics file (called `stats`) describing the HMM set defined by `hmm1/MMF`.

```
HERest -H hmm1/MMF -M hmmx -s stats hmmlist train1 train2 ....
HHEd -H hmm1/MMF -M hmm2 cmds.hed hmmlist
```

The required statistics file is generated by HERest but the re-estimated model set stored in `hmmx/MMF` is ignored and can be deleted.

10.2 Constructing Context-Dependent Models

The first stage of model refinement is usually to convert a set of initialised and trained context-independent monophone HMMs to a set of context dependent models. As explained in section 6.4, HTK uses the convention that a HMM name of the form `l-p+r` denotes the context-dependent version of the phone `p` which is to be used when the left neighbour is the phone `l` and the right neighbour is the phone `r`. To make a set of context dependent phone models, it is only necessary to construct a HMM list, called say `cdlist`, containing the required context-dependent models and then execute HHEd with a single command in its edit script

```
CL cdlist
```

The effect of this command is that for each model `l-p+r` in `cdlist` it makes a copy of the monophone `p`.

The set of context-dependent models output by the above must be reestimated using HERest. To do this, the training data transcriptions must be converted to use context-dependent labels and the original monophone hmm list must be replaced by `cdlist`. In fact, it is best to do this conversion before cloning the monophones because if the HLEd `TC` command is used then the `-n` option can be used to generate the required list of context dependent HMMs automatically.

Before building a set of context-dependent models, it is necessary to decide whether or not cross-word triphones are to be used. If they are, then word boundaries in the training data can be ignored and all monophone labels can be converted to triphones. If, however, word internal triphones are to be used, then word boundaries in the training transcriptions must be marked in some way (either by an explicit marker which is subsequently deleted or by using a short pause *tee-model*). This word boundary marker is then identified to HLEd using the `WB` command to make the `TC` command use biphones rather than triphones at word boundaries (see section 6.4).

All HTK tools can read and write HMM definitions in text or binary form. Text is good for seeing exactly what the tools are producing, but binary is much faster to load and store, and much more compact. Binary output is enabled either using the standard option `-B` or by setting the configuration variable `SAVEBINARY`. In the above example, the HMM set input to HHEd will contain a small set of monophones whereas the output will be a large set of triphones. In order, to save storage and computation, this is usually a good point to switch to binary storage of MMFs.

10.3 Parameter Tying and Item Lists

As explained in Chapter 7, HTK uses macros to support a generalised parameter tying facility. Referring again to Fig. 7.7.8, each of the solid black circles denotes a potential *tie-point* in the hierarchy of HMM parameters. When two or more parameter sets are tied, the same set of parameter values are shared by all the *owners* of the tied set. Externally, tied parameters are represented by macros and internally they are represented by structure sharing. The accumulators needed for the numerators and denominators of the Baum-Welch re-estimation formulae given in section 8.8 are attached directly to the parameters themselves. Hence, when the values of a tied parameter set are re-estimated, all of the data which would have been used to estimate each individual untied parameter are effectively pooled leading to more robust parameter estimation.

Note also that although parameter tying is implemented in a way which makes it transparent to the HTK re-estimation and recognition tools, in practice, these tools do notice when a system has been tied and try to take advantage of it by avoiding redundant computations.

Although macro definitions could be written by hand, in practice, tying is performed by executing HHEd commands and the resulting macros are thus generated automatically. The basic HHEd command for tying a set of parameters is the `TI` command which has the form

```
TI macroname itemlist
```

This causes all items in the given `itemlist` to be tied together and output as a macro called `macroname`. Macro names are written as a string of characters optionally enclosed in double quotes. The latter are necessary if the name contains one or more characters which are not letters or digits.

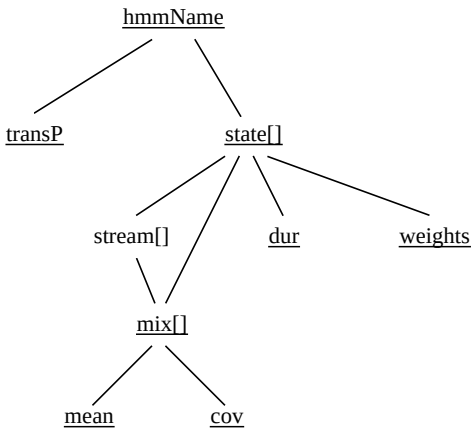


Fig. 10.1 Item List Construction

Item lists use a simple language to identify sets of points in the HMM parameter hierarchy illustrated in Fig. 7.7.8. This language is defined fully in the reference entry for HHEd. The essential idea is that item lists represent paths down the hierarchical parameter tree where the direction *down* should be regarded as travelling from the *root* of the tree to towards the *leaves*. A path can be unique, or more usually, it can be a pattern representing a set of paths down the tree. The point at which each path stops identifies one member of the set represented by the item list. Fig. 10.1 shows the possible paths down the tree. In text form the branches are replaced by dots and the underlined node names are possible terminating points. At the topmost level, an item list is a comma separated list of paths enclosed in braces.

Some examples, should make all this clearer. Firstly, the following is a legal but somewhat long-winded way of specifying the set of items comprising states 2, 3 and 4 of the HMM called `aa`

```
{ aa.state[2],aa.state[3],aa.state[4] }
```

however in practice this would be written much more compactly as

```
{ aa.state[2-4] }
```

It must be emphasised that indices in item lists are really *patterns*. The set represented by an item list consists of all those elements which match the patterns. Thus, if `aa` only had two emitting states, the above item list would not generate an error. It would simply only match two items. The reason for this is that the same pattern can be applied to many different objects. For example, the HMM name can be replaced by a list of names enclosed in brackets, furthermore each HMM name can include '?' characters which match any single character and '*' characters which match zero or more characters. Thus

```
{ (aa+*,iy+*,eh+*).state[2-4] }
```

represents states 2, 3 and 4 of all biphone models corresponding to the phonemes `aa`, `iy` and `eh`. If `aa` had just 2 emitting states and the others had 4 emitting states, then this item list would include 2 states from each of the `aa` models and 3 states from each of the others. Moving further down the tree, the item list

```
{ *.state[2-4].stream[1].mix[1,3].cov }
```

denotes the set of all covariance vectors (or matrices) of the first and third mixture components of stream 1, of states 2 to 4 of all HMMs. Since many HMM systems are single stream, the `stream` part of the path can be omitted if its value is 1. Thus, the above could have been written

```
{ *.state[2-4].mix[1,3].cov }
```

These last two examples also show that indices can be written as comma separated lists as well as ranges, for example, `[1,3,4-6,9]` is a valid index list representing states 1, 3, 4, 5, 6, and 9.

When item lists are used as the argument to a `TI` command, the kind of items represented by the list determines the macro type in a fairly obvious way. The only non-obvious cases are firstly that lists ending in `cov` generate ~v, ~i, ~c, or ~x macros as appropriate. If an explicit set of mixture components is defined as in

```
{ *.state[2].mix[1-5] }
```

then ~m macros are generated but omitting the indices altogether denotes a special case of mixture tying which is explained later in Chapter 11.

To illustrate the use of item lists, some example `TI` commands can now be given. Firstly, when a set of context-dependent models is created, it can be beneficial to share one transition matrix across all variants of a phone rather than having a distinct transition matrix for each. This could be achieved by adding `TI` commands immediately after the `CL` command described in the previous section, that is

```
CL cdlist
TI T_ah {*-ah+*.transP}
TI T_eh {*-eh+*.transP}
TI T_ae {*-ae+*.transP}
TI T_ih {*-ih+*.transP}
 ... etc
```

As a second example, a so-called Grand Variance HMM system can be generated very easily with the following HHEd command

```
TI "gvar" { *.state[2-4].mix[1].cov }
```

where it is assumed that the HMMs are 3-state single mixture component models. The effect of this command is to tie all state distributions to a single global variance vector. For applications, where there is limited training data, this technique can improve performance, particularly in noise.

Speech recognition systems will often have distinct models for silence and short pauses. A silence model `sil` may have the normal 3 state topology whereas a short pause model may have just a single state. To avoid the two models *competing* with each other, the `sp` model state can be tied to the centre state of the `sil` model thus

```
TI "silst" { sp.state[2], sil.state[3] }
```

So far nothing has been said about how the parameters are actually determined when a set of items is replaced by a single shared representative. When states are tied, the state with the broadest variances and as few as possible zero mixture component weights is selected from the pool and used as the representative. When mean vectors are tied, the average of all the mean vectors in the pool is used and when variances are tied, the largest variance in the the pool is used. In all other cases, the last item in the tie-list is arbitrarily chosen as representative. All of these selection criteria are *ad hoc*, but since the tie operations are always followed by explicit re-estimation using HERest, the precise choice of representative for a tied set is not critical.

Finally, tied parameters can be untied. For example, subsequent refinements of the context-dependent model set generated above with tied transition matrices might result in a much more compact set of models for which individual transition parameters could be robustly estimated. This can be done using the `UT` command whose effect is to untie all of the items in its argument list. For example, the command

```
UT {*-iy+*.transP}
```

would untie the transition parameters in all variants of the `iy` phoneme. This untying works by simply making unique copies of the tied parameters. These untied parameters can then subsequently be re-estimated.

10.4 Data-Driven Clustering

In section 10.2, a method of triphone construction was described which involved cloning all monophones and then re-estimating them using data for which monophone labels have been replaced by triphone labels. This will lead to a very large set of models, and relatively little training data for

each model. Applying the argument that context will not greatly affect the centre states of triphone models, one way to reduce the total number of parameters without significantly altering the models' ability to represent the different contextual effects might be to tie all of the centre states across all models derived from the same monophone. This tying could be done by writing an edit script of the form

```
TI "iyS3" {*-iy+*.state[3]}
TI "ihS3" {*-ih+*.state[3]}
TI "ehS3" {*-eh+*.state[3]}
 .... etc
```

Each `TI` command would tie all the centre states of all triphones in each phone group. Hence, if there were an average of 100 triphones per phone group then the total number of states per group would be reduced from 300 to 201.

Explicit tyings such as these can have some positive effect but overall they are not very satisfactory. Tying all centre states is too severe and worse still, the problem of undertraining for the left and right states remains. A much better approach is to use clustering to decide which states to tie. HHEd provides two mechanisms for this. In this section a data-driven clustering approach will be described and in the next section, an alternative decision tree-based approach is presented.

Data-driven clustering is performed by the `TC` and `NC` commands. These both invoke the same top-down hierarchical procedure. Initially all states are placed in individual clusters. The pair of clusters which when combined would form the smallest resultant cluster are merged. This process repeats until either the size of the largest cluster reaches the threshold set by the `TC` command or the total number of clusters has fallen to that specified by by the `NC` command. The size of cluster is defined as the greatest distance between any two states. The distance metric depends on the type of state distribution. For single Gaussians, a weighted Euclidean distance between the means is used and for tied-mixture systems a Euclidean distance between the mixture weights is used. For all other cases, the average probability of each component mean with respect to the other state is used. The details of the algorithm and these metrics are given in the reference section for HHEd.

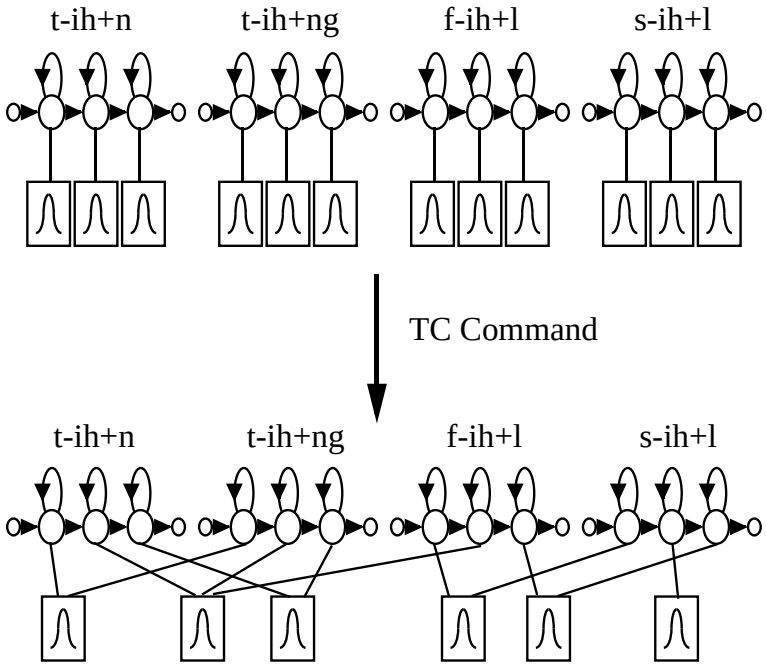


Fig. 10.2 Data-driven state tying

As an example, the following HHEd script would cluster and tie the corresponding states of the triphone group for the phone `ih`

```
TC 100.0 "ihS2" {*-ih+*.state[2]}
```

```
TC 100.0 "ihS3" {*-ih+*.state[3]}
TC 100.0 "ihS4" {*-ih+*.state[4]}
```

In this example, each TC command performs clustering on the specified set of states, each cluster is tied and output as a macro. The macro name is generated by appending the cluster index to the macro name given in the command. The effect of this command is illustrated in Fig. 10.2. Note that if a word-internal triphone system is being built, it is sensible to include biphones as well as triphones in the item list, for example, the first command above would be written as

```
TC 100.0 "ihS2" {(*-ih,ih+*,*-ih+*).state[2]}
```

If the above TC commands are repeated for all phones, the resulting set of tied-state models will have far fewer parameters in total than the original untied set. The numeric argument immediately following the TC command name is the cluster threshold. Increasing this value will allow larger and hence, fewer clusters. The aim, of course, is to strike the right balance between compactness and the acoustic accuracy of the individual models. In practice, the use of this command requires some experimentation to find a good threshold value. HHEd provides extensive trace output for monitoring clustering operations. Note in this respect that as well as setting tracing from the command line and the configuration file, tracing in HHEd can be set by the TR command. Thus, tracing can be controlled at the command level. Further trace information can be obtained by including the SH command at strategic points in the edit script. The effect of executing this command is to list out all of the parameter tyings currently in force.

A potential problem with the use of the TC and NC commands is that *outlier* states will tend to form their own singleton clusters for which there is then insufficient data to properly train. One solution to this is to use the RO command to remove outliers. This commmand has the form

```
RO thresh "statsfile"
```

where `statsfile` is the name of a statistics file output using the `-s` option of HERest. This statistics file holds the *occupation counts* for all states of the HMM set being trained. The term *occupation count* refers to the number of frames allocated to a particular state and can be used as a measure of how much training data is available for estimating the parameters of that state. The RO command must be executed *before* the TC or NC commands used to do the actual clustering. Its effect is to simply read in the statistics information from the given file and then to set a flag instructing the TC or NC commands to remove any outliers remaining at the conclusion of the normal clustering process. This is done by repeatedly finding the cluster with the smallest total occupation count and merging it with its nearest neighbour. This process is repeated until all clusters have a total occupation count which exceeds `thresh`, thereby ensuring that every cluster of states will be properly trained in the subsequent re-estimation performed by HERest.

On completion of the above clustering and tying procedures, many of the models may be effectively identical, since acoustically similar triphones may share common clusters for all their emitting states. They are then, in effect, so-called *generalised triphones*. State tying can be further exploited if the HMMs which are effectively equivalent are identified and then tied via the physical-logical mapping[1] facility provided by HMM lists (see section 7.4). The effect of this would be to reduce the total number of HMM definitions required. HHEd provides a compaction command to do all of this automatically. For example, the command

```
CO newList
```

will compact the currently loaded HMM set by identifying equivalent models and then tying them via the new HMM list output to the file `newList`. Note, however, that for two HMMs to be tied, they must be identical in all respects. This is one of the reasons why transition parameters are often tied across triphone groups otherwise HMMs with identical states would still be left distinct due to minor differences in their transition matrices.

10.5 Tree-Based Clustering

One limitation of the data-driven clustering procedure described above is that it does not deal with triphones for which there are no examples in the training data. When building word-internal triphone systems, this problem can often be avoided by careful design of the training database but when building large vocabulary cross-word triphone systems *unseen* triphones are unavoidable.

[1] The physical HMM which corresponding to several logical HMMs will be arbitrarily named after one of them.

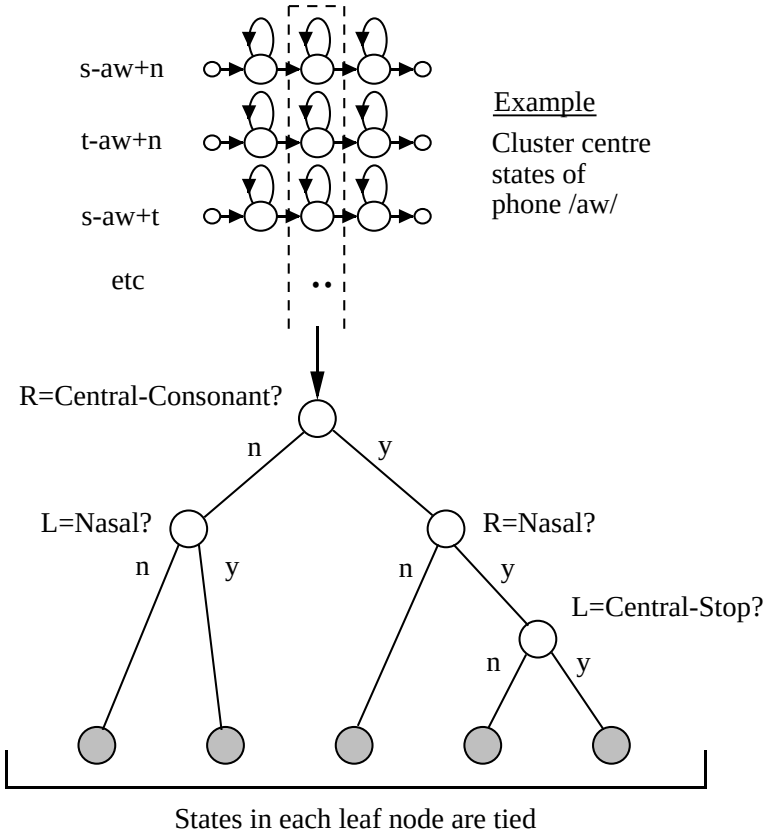


Fig. 10.3 Decision tree-based state tying

HHEd provides an alternative decision tree based clustering mechanism which provides a similar quality of clustering but offers a solution to the unseen triphone problem. Decision tree-based clustering is invoked by the command `TB` which is analogous to the `TC` command described above and has an identical form, that is

```
TB thresh macroname itemlist
```

Apart from the clustering mechanism, there are some other differences between `TC` and `TB`. Firstly, `TC` uses a distance metric between states whereas `TB` uses a log likelihood criterion. Thus, the threshold values are not directly comparable. Furthermore, `TC` supports any type of output distribution whereas `TB` only supports single-Gaussian continuous density output distributions. Secondly, although the following describes only state clustering, the `TB` command can also be used to cluster whole models.

A phonetic decision tree is a binary tree in which a yes/no phonetic question is attached to each node. Initially all states in a given item list (typically a specific phone state position) are placed at the root node of a tree. Depending on each answer, the pool of states is successively split and this continues until the states have trickled down to leaf-nodes. All states in the same leaf node are then tied. For example, Fig 10.3 illustrates the case of tying the centre states of all triphones of the phone /aw/ (as in "out"). All of the states trickle down the tree and depending on the answer to the questions, they end up at one of the shaded terminal nodes. For example, in the illustrated case, the centre state of `s-aw+n` would join the second leaf node from the right since its right context is a central consonant, and its right context is a nasal but its left context is not a central stop.

The question at each node is chosen to (locally) maximise the likelihood of the training data given the final set of state tyings. Before any tree building can take place, all of the possible phonetic questions must be loaded into HHEd using `QS` commands. Each question takes the form "Is the left or right context in the set P?" where the context is the model context as defined by its logical name. The set P is represented by an item list and for convenience every question is given a name. As an example, the following command

```
QS "L_Nasal" { ng-*,n-*,m-* }
```

defines the question "Is the left context a nasal?".

It is possible to calculate the log likelihood of the training data given any pool of states (or models). Furthermore, this can be done without reference to the training data itself since for single Gaussian distributions the means, variances and state occupation counts (input via a stats file) form sufficient statistics. Splitting any pool into two will increase the log likelihood since it provides twice as many parameters to model the same amount of data. The increase obtained when each possible question is used can thus be calculated and the question selected which gives the biggest improvement.

Trees are therefore built using a top-down sequential optimisation process. Initially all states (or models) are placed in a single cluster at the root of the tree. The question is then found which gives the best split of the root node. This process is repeated until the increase in log likelihood falls below the threshold specified in the `TB` command. As a final stage, the decrease in log likelihood is calculated for merging terminal nodes with differing parents. Any pair of nodes for which this decrease is less than the threshold used to stop splitting are then merged.

As with the `TC` command, it is useful to prevent the creation of clusters with very little associated training data. The `RO` command can therefore be used in tree clustering as well as in data-driven clustering. When used with trees, any split which would result in a total occupation count falling below the value specified is prohibited. Note that the `RO` command can also be used to load the required stats file. Alternatively, the stats file can be loaded using the `LS` command.

As with data-driven clustering, using the trace facilities provided by HHEd is recommended for monitoring and setting the appropriate thresholds. Basic tracing provides the following summary data for each tree

```
TB 350.00 aw_s3 {}
 Tree based clustering
  Start  aw[3] : 28  have  LogL=-86.899 occ=864.2
  Via    aw[3] : 5   gives LogL=-84.421 occ=864.2
  End    aw[3] : 5   gives LogL=-84.421 occ=864.2
TB: Stats 28->5 [17.9%]  { 4537->285 [6.3%] total }
```

This example corresponds to the case illustrated in Fig 10.3. The `TB` command has been invoked with a threshold of 350.0 to cluster the centre states of the triphones of the phone *aw*. At the start of clustering with all 28 states in a single pool, the average log likelihood per unit of occupation is -86.9 and on completion with 5 clusters this has increased to -84.4. The middle line labelled "via" gives the position after the tree has been built but before terminal nodes have been merged (none were merged in this case). The last line summarises the overall position. After building this tree, a total of 4537 states were reduced to 285 clusters.

As noted at the start of this section, an important advantage of tree-based clustering is that it allows triphone models which have no training data to be synthesised. This is done in HHEd using the `AU` command which has the form

```
AU hmmlist
```

Its effect is to scan the given `hmmlist` and any physical models listed which are not in the currently loaded set are synthesised. This is done by descending the previously constructed trees for that phone and answering the questions at each node based on the new unseen context. When each leaf node is reached, the state representing that cluster is used for the corresponding state in the unseen triphone.

The `AU` command can be used within the same edit script as the tree building commands. However, it will often be the case that a new set of triphones is needed at a later date, perhaps as a result of vocabulary changes. To make this possible, a complete set of trees can be saved using the `ST` command and then later reloaded using the `LT` command.

10.6 Mixture Incrementing

When building sub-word based continuous density systems, the final system will typically consist of multiple mixture component context-dependent HMMs. However, as indicated previously, the early stages of triphone construction, particularly state tying, are best done with single Gaussian models. Indeed, if tree-based clustering is to be used there is no option.

In HTK therefore, the conversion from single Gaussian HMMs to multiple mixture component HMMs is usually one of the final steps in building a system. The mechanism provided to do this is the HHEd `MU` command which will increase the number of components in a mixture by a process called *mixture splitting*. This approach to building a multiple mixture component system is extremely flexible since it allows the number of mixture components to be repeatedly increased until the desired level of performance is achieved.

The `MU` command has the form

```
MU n itemList
```

where `n` gives the new number of mixture components required and `itemList` defines the actual mixture distributions to modify. This command works by repeatedly splitting the mixture with the largest mixture weight until the required number of components is obtained. The actual split is performed by copying the mixture, dividing the weights of both copies by 2, and finally perturbing the means by plus or minus 0.2 standard deviations. For example, the command

```
MU 3 {aa.state[2].mix}
```

would increase the number of mixture components in the output distribution for state 2 of model `aa` to 3. Normally, however, the number of components in all mixture distributions will be increased at the same time. Hence, a command of the form is more usual

```
MU 3 {*.state[2-4].mix}
```

It is usually a good idea to increment mixture components in stages, for example, by incrementing by 1 or 2 then re-estimating, then incrementing by 1 or 2 again and re-estimating, and so on until the required number of components are obtained. This also allows recognition performance to be monitored to find the optimum.

One final point with regard to multiple mixture component distributions is that all HTK tools ignore mixture components whose weights fall below a threshold value called `MINMIX` (defined in `HModel.h`). Such mixture components are called *defunct*. Defunct mixture components can be prevented by setting the `-w` option in HERest so that all mixture weights are floored to some level above `MINMIX`. If mixture weights are allowed to fall below `MINMIX` then the corresponding Gaussian parameters will not be written out when the model containing that component is saved. It is possible to recover from this, however, since the `MU` command will replace defunct mixtures before performing any requested mixture component increment.

10.7 Regression Class Tree Construction

In order to perform most model adaptation tasks (see chapter 9), it will be neccesary to produce a binary regression class tree. This tree is stored in the MMF, along with a regression base class identifier for each mixture component. An example regression tree and how it may be used is shown in subsection 9.1.2. HHEd provides the means to construct a regression class tree for a given MMF, and is invoked using the `RC` command. It is also necessary to supply a statistics file, which is output using the `-s` option of `HERest`. The statistics file can be loaded by invoking the `LS` command.

A centroid-splitting algorithm using a Euclidean distance measure is used to grow the binary regression class tree to cluster the model set's mixture components. Each leaf node therefore specifies a particular mixture component cluster. This algorithm proceeds as follows until the requested number of terminals has been achieved.

- Select a terminal node that is to be split.
- Calculated the mean and variance from the mixture components clustered at this node.
- Create two children. Initialise their means to the parent mean perturbed in opposite directions (for each child) by a fraction of the variance.
- For each component at the parent node assign the component to one of the children by using a Euclidean distance measure to ascertain which child mean the component is closest to.
- Once all the components have been assigned, calculate the new means for the children, based on the component assignments.

- Keep re-assigning components to the children and re-estimating the child means until there is no change in assignments from one iteration to the next. Now finalise the split.

As an example, the following HHEd script would produce a regression class tree with 32 terminal nodes, or regression base classes:-

```
LS "statsfile"
RC 32 "rtree"
```

A further optional argument is possible with the `RC` command. This argument allows the user to specify the non-speech class mixture components using an `itemlist`, such as the silence mixture components.

```
LS "statsfile"
RC 32 "rtree" {sil.state[2-4].mix}
```

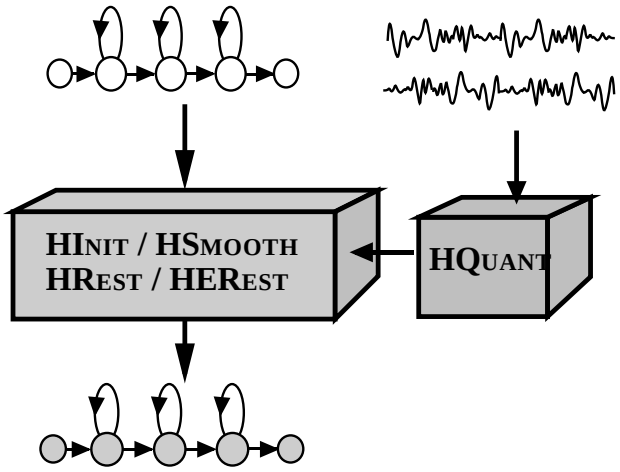
In this case the first split that will be made in the regression class tree will be to split the speech and non-speech sounds, after which the tree building continues as usual.

10.8 Miscellaneous Operations

The preceding sections have described the main HHEd commands used for building continuous density systems with tied parameters. A further group of commands (`JO`, `TI` and `HK`) are used to build tied-mixture systems and these are described in Chapter 11. Those remaining cover a miscellany of functions. They are documented in the reference entry for HHEd and include commands to add and remove state transitions (`AT`, `RT`); synthesise triphones from biphones (`MT`); change the parameter kind of a HMM (`SK`); modify stream dimensions (`SS`, `SU`, `SW`); change/add an identifier name to an MMF (`RN` command); and expand HMM sets by duplication, for example, as needed in making gender dependent models (`DP`).

Chapter 11

Discrete and Tied-Mixture Models



Most of the discussion so far has focussed on using HTK to model sequences of continuous-valued vectors. In contrast, this chapter is mainly concerned with using HTK to model sequences of discrete symbols. Discrete symbols arise naturally in modelling many types of data, for example, letters and words, bitmap images, and DNA sequences. Continuous signals can also be converted to discrete symbol sequences by using a quantiser and in particular, speech vectors can be *vector quantised* as described in section 5.14. In all cases, HTK expects a set of N discrete symbols to be represented by the contiguous sequence of integer numbers from 1 to N.

In HTK discrete probabilities are regarded as being closely analogous to the mixture weights of a continuous density system. As a consequence, the representation and processing of discrete HMMs shares a great deal with continuous density models. It follows from this that most of the principles and practice developed already are equally applicable to discrete systems. As a consequence, this chapter can be quite brief.

The first topic covered concerns building HMMs for discrete symbol sequences. The use of discrete HMMs with speech is then presented. The tool HQUANT is described and the method of converting continuous speech vectors to discrete symbols is reviewed. This is followed by a brief discussion of tied-mixture systems which can be regarded as a compromise between continuous and discrete density systems. Finally, the use of the HTK tool HSMOOTH for parameter smoothing by deleted interpolation is presented.

11.1 Modelling Discrete Sequences

Building HMMs for discrete symbol sequences is essentially the same as described previously for continuous density systems. Firstly, a prototype HMM definition must be specified in order to fix the model topology. For example, the following is a 3 state ergodic HMM in which the emitting states are fully connected.

```
~o <DISCRETE> <StreamInfo> 1 1
```

```
~h "dproto"
<BeginHMM>
   <NumStates> 5
   <State> 2 <NumMixes> 10
      <DProb> 5461*10
   <State> 3 <NumMixes> 10
      <DProb> 5461*10
   <State> 4 <NumMixes> 10
      <DProb> 5461*10
   <TransP> 5
       0.0 1.0 0.0 0.0 0.0
       0.0 0.3 0.3 0.3 0.1
       0.0 0.3 0.3 0.3 0.1
       0.0 0.3 0.3 0.3 0.1
       0.0 0.0 0.0 0.0 0.0
<EndHMM>
```

As described in chapter 7, the notation for discrete HMMs borrows heavily on that used for continuous density models by equating mixture components with symbol indices. Thus, this definition assumes that each training data sequence contains a single stream of symbols indexed from 1 to 10. In this example, all symbols in each state have been set to be equally likely[1]. If prior information is available then this can of course be used to set these initial values.

The training data needed to build a discrete HMM can take one of two forms. It can either be discrete (`SOURCEKIND=DISCRETE`) in which case it consists of a sequence of 2-byte integer symbol indices. Alternatively, it can consist of continuous parameter vectors with an associated VQ codebook. This latter case is dealt with in the next section. Here it will be assumed that the data is symbolic and that it is therefore stored in discrete form. Given a set of training files listed in the script file `train.scp`, an initial HMM could be estimated using

```
HInit -T 1 -w 1.0 -o dhmm -S train.scp -M hmm0 dproto
```

This use of HINIT is identical to that which would be used for building whole word HMMs where no associated label file is assumed and the whole of each training sequence is used to estimate the HMM parameters. Its effect is to read in the prototype stored in the file `dproto` and then use the training examples to estimate initial values for the output distributions and transition probabilities. This is done by firstly uniformly segmenting the data and for each segment counting the number of occurrences of each symbol. These counts are then normalised to provide output distributions for each state. HINIT then uses the Viterbi algorithm to resegment the data and recompute the parameters. This is repeated until convergence is achieved or an upper limit on the iteration count is reached. The transition probabilities at each step are estimated simply by counting the number of times that each transition is made in the Viterbi alignments and normalising. The final model is renamed `dhmm` and stored in the directory `hmm0`.

When building discrete HMMs, it is important to floor the discrete probabilites so that no symbol has a zero probability. This is achieved using the `-w` option which specifies a floor value as a multiple of a global constant called `MINMIX` whose value is 10^{-5}.

The initialised HMM created by HINIT can then be further refined if desired by using HREST to perform Baum-Welch re-estimation. It would be invoked in a similar way to the above except that there is now no need to rename the model. For example,

```
HRest -T 1 -w 1.0 -S train.scp -M hmm1 hmm0/dhmm
```

would read in the model stored in `hmm0/dhmm` and write out a new model of the same name to the directory `hmm1`.

11.2 Using Discrete Models with Speech

As noted in section 5.14, discrete HMMs can be used to model speech by using a vector quantiser to map continuous density vectors into discrete symbols. A vector quantiser depends on a so-called

[1] Remember that discrete probabilities are scaled such that 32767 is equivalent to a probability of 0.000001 and 0 is equivalent to a probability of 1.0

codebook which defines a set of partitions of the vector space. Each partition is represented by the mean value of the speech vectors belonging to that partition and optionally a variance representing the spread. Each incoming speech vector is then matched with each partition and assigned the index corresponding to the partition which is closest using a Mahanalobis distance metric.

In HTK such a codebook can be built using the tool HQuant. This tool takes as input a set of continuous speech vectors, clusters them and uses the centroid and optionally the variance of each cluster to define the partitions. HQuant can build both linear and tree structured codebooks. To build a linear codebook, all training vectors are initially placed in one cluster and the mean calculated. The mean is then perturbed to give two means and the training vectors are partitioned according to which mean is nearest to them. The means are then recalculated and the data is repartitioned. At each cycle, the total distortion (i.e. total distance between the cluster members and the mean) is recorded and repartitioning continues until there is no significant reduction in distortion. The whole process then repeats by perturbing the mean of the cluster with the highest distortion. This continues until the required number of clusters have been found.

Since all training vectors are reallocated at every cycle, this is an expensive algorithm to compute. The maximum number of iterations within any single cluster increment can be limited using the configuration variable `MAXCLUSTITER` and although this can speed-up the computation significantly, the overall training process is still computationally expensive. Once built, vector quantisation is performed by scanning all codebook entries and finding the nearest entry. Thus, if a large codebook is used, the run-time VQ look-up operation can also be expensive.

As an alternative to building a linear codebook, a tree-structured codebook can be used. The algorithm for this is essentially the same as above except that every cluster is split at each stage so that the first cluster is split into two, they are split into four and so on. At each stage, the means are recorded so that when using the codebook for vector quantising a fast binary search can be used to find the appropriate leaf cluster. Tree-structured codebooks are much faster to build since there is no repeated reallocation of vectors and much faster in use since only $O(\log_2 N)$ distance need to be computed where N is the size of the codebook. Unfortunately, however, tree-structured codebooks will normally incur higher VQ distortion for a given codebook size.

When delta and acceleration coefficients are used, it is usually best to split the data into multiple streams (see section 5.13. In this case, a separate codebook is built for each stream.

As an example, the following invocation of HQuant would generate a linear codebook in the file `linvq` using the data stored in the files listed in `vq.scp`.

```
HQuant -C config -s 4 -n 3 64 -n 4 16 -S vq.scp linvq
```

Here the configuration file `config` specifies the `TARGETKIND` as being `MFCC_E_D_A` i.e. static coefficients plus deltas plus accelerations plus energy. The `-s` options requests that this parameterisation be split into 4 separate streams. By default, each individual codebook has 256 entries, however, the `-n` option can be used to specify alternative sizes.

If a tree-structured codebook was wanted rather than a linear codebook, the `-t` option would be set. Also the default is to use Euclidean distances both for building the codebook and for subsequent coding. Setting the `-d` option causes a diagonal covariance Mahalanobis metric to be used and the `-f` option causes a full covariance Mahalanobis metric to be used.

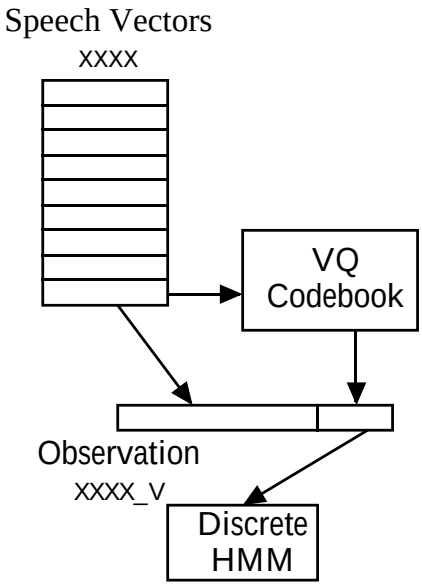


Fig. 11.1 VQ Processing

Once the codebook is built, normal speech vector files can be converted to discrete files using HCopy. This was explained previously in section 5.14. The basic mechanism is to add the qualifier `_V` to the `TARGETKIND`. This causes HParm to append a codebook index to each constructed observation vector. If the configuration variable `SAVEASVQ` is set true, then the output routines in HParm will discard the original vectors and just save the VQ indices in a `DISCRETE` file. Alternatively, HTK will regard any speech vector with `_V` set as being compatible with discrete HMMs. Thus, it is not necessary to explicitly create a database of discrete training files if a set of continuous speech vector parameter files already exists. Fig. 11.1 illustrates this process.

Once the training data has been configured for discrete HMMs, the rest of the training process is similar to that previously described. The normal sequence is to build a set of monophone models and then clone them to make triphones. As in continuous density systems, state tying can be used to improve the robustness of the parameter estimates. However, in the case of discrete HMMs, alternative methods based on interpolation are possible. These are discussed in section 11.4.

11.3 Tied Mixture Systems

Discrete systems have the advantage of low run-time computation. However, vector quantisation reduces accuracy and this can lead to poor performance. As a intermediate between discrete and continuous, a fully tied-mixture system can be used. Tied-mixtures are conceptually just another example of the general parameter tying mechanism built-in to HTK. However, to use them effectively in speech recognition systems a number of storage and computational optimisations must be made. Hence, they are given special treatment in HTK.

When specific mixtures are tied as in

```
TI "mix" {*.state[2].mix[1]}
```

then a Gaussian mixture component is shared across all of the owners of the tie. In this example, all models will share the same Gaussian for the first mixture component of state 2. However, if the mixture component index is missing, then all of the mixture components participating in the tie are *joined* rather than tied. More specifically, the commands

```
JO 128 2.0
TI "mix" {*.state[2-4].mix}
```

has the following effect. All of the mixture components in states 2 to 4 of all models are collected into a pool. If the number of components in the pool exceeds 128, as set by the preceding join command

JO, then components with the smallest weights are removed until the pool size is exactly 128. Similarly, if the size of the initial pool is less than 128, then mixture components are split using the same algorithm as for the Mix-Up MU command. All states then share all of the mixture components in this pool. The new mixture weights are chosen to be proportional to the log probability of the corresponding new mixture component mean with respect to the original distribution for that state. The log is used here to give a wider spread of mixture weights. All mixture weights are floored to the value of the second argument of the JO command times MINMIX.

The net effect of the above two commands is to create a set of tied-mixture HMMs[2] where the same set of mixture components is shared across all states of all models. However, the type of the HMM set so created will still be SHARED and the internal representation will be the same as for any other set of parameter tyings. To obtain the optimised representation of the tied-mixture weights described in section 7.5, the following HHEd HK command must be issued

```
HK TIEDHS
```

This will convert the internal representation to the special tied-mixture form in which all of the tied mixtures are stored in a global table and referenced implicitly instead of being referenced explicitly using pointers.

Tied-mixture HMMs work best if the information relating to different sources such as delta coefficients and energy are separated into distinct data streams. This can be done by setting up multiple data stream HMMs from the outset. However, it is simpler to use the SS command in HHEd to split the data streams of the currently loaded HMM set. Thus, for example, the command

```
SS 4
```

would convert the currently loaded HMMs to use four separate data streams rather than one. When used in the construction of tied-mixture HMMs this is analogous to the use of multiple codebooks in discrete density HMMs.

The procedure for building a set of tied-mixture HMMs may be summarised as follows

1. Choose a *codebook* size for each data stream and then decide how many Gaussian components will be needed from an initial set of monophones to approximately fill this codebook. For example, suppose that there are 48 three state monophones. If codebook sizes of 128 are chosen for streams 1 and 2, and a codebook size of 64 is chosen for stream 3 then single Gaussian monophones would provide enough mixtures in total to fill the codebooks.

2. Train the initial set of monophones.

3. Use HHEd to first split the HMMs into the required number of data streams, tie each individual stream and then convert the tied-mixture HMM set to have the kind TIEDHS. A typical script to do this for four streams would be

```
SS 4
JO 256 2.0
TI st1 {*.state[2-4].stream[1].mix}
JO 128 2.0
TI st2 {*.state[2-4].stream[2].mix}
JO 128 2.0
TI st3 {*.state[2-4].stream[3].mix}
JO 64 2.0
TI st4 {*.state[2-4].stream[4].mix}
HK TIEDHS
```

4. Re-estimate the models using HERest in the normal way.

Once the set of retrained tied-mixture models has been produced, context dependent models can be constructed using similar methods to those outlined previously.

When evaluating probabilities in tied-mixture systems, it is often sufficient to sum just the most likely mixture components since for any particular input vector, its probability with respect to many of the Gaussian components will be very low. HTK tools recognise TIEDHS HMM sets as being special in the sense that additional optimisations are possible. When full tied-mixtures are

[2] Also called *semi-continuous* HMMs in the the literature.

used, then an additional layer of pruning is applied. At each time frame, the log probability of the current observation is computed for each mixture component. Then only those components which lie within a threshold of the most likely component are retained. This pruning is controlled by the `-c` option in HREST, HEREST and HVITE.

11.4 Parameter Smoothing

When large sets of context-dependent triphones are built using discrete models or tied-mixture models, under-training can be a severe problem since each state has a large number of mixture weight parameters to estimate. The HTK tool HSMOOTH allows these discrete probabilities or mixture component weights to be smoothed with the monophone weights using a technique called deleted interpolation.

HSMOOTH is used in combination with HEREST working in parallel mode. The training data is split into blocks and each block is used separately to re-estimate the HMMs. However, since HEREST is in parallel mode, it outputs a dump file of accumulators instead of updating the models. HSMOOTH is then used in place of the second pass of HEREST. It reads in the accumulator information from each of the blocks, performs deleted interpolation smoothing on the accumulator values and then outputs the re-estimated HMMs in the normal way.

HSMOOTH implements a conventional deleted interpolation scheme. However, optimisation of the smoothing weights uses a fast binary chop scheme rather than the more usual Baum-Welch approach. The algorithm for finding the optimal interpolation weights for a given state and stream is as follows where the description is given in terms of tied-mixture weights but the same applies to discrete probabilities.

Assume that HEREST has been set-up to output N separate blocks of accumulators. Let $w_i^{(n)}$ be the i'th mixture weight based on the accumulator blocks 1 to N but excluding block n, and let $\bar{w}_i^{(n)}$ be the corresponding context independent weight. Let $x_i^{(n)}$ be the i'th mixture weight count for the deleted block n. The derivative of the log likelihood of the deleted block, given the probability distribution with weights $c_i = \lambda w_i + (1-\lambda)\bar{w}_i$ is given by

$$D(\lambda) = \sum_{n=1}^{N}\sum_{i=1}^{M} x_i^{(n)} \left[\frac{w_i^{(n)} - \bar{w}_i^{(n)}}{\lambda w_i^{(n)} + (1-\lambda)\bar{w}_i^{(n)}} \right] \tag{11.1}$$

Since the log likelihood is a convex function of λ, this derivative allows the optimal value of λ to be found by a simple binary chop algorithm, viz.

```
function FindLambdaOpt:
   if (D(0) <= 0) return 0;
   if (D(1) >= 0) return = 1;
   l=0; r=1;
   for (k=1; k<=maxStep; k++){
      m = (l+r)/2;
      if (D(m) == 0) return m;
      if (D(m) > 0) l=m; else r=m;
   }
   return m;
```

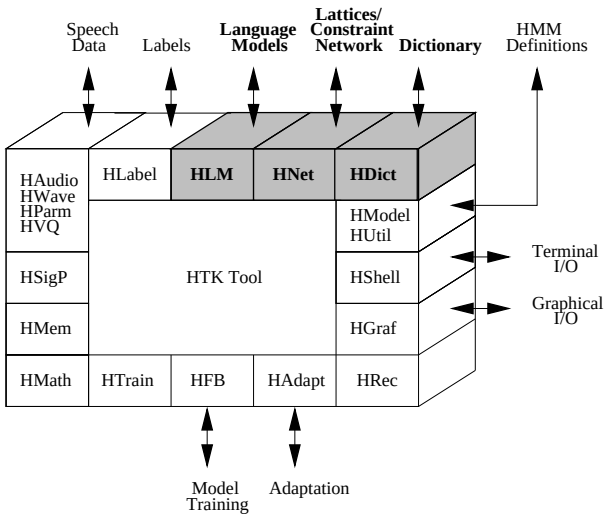
HSMOOTH is invoked in a similar way to HEREST. For example, suppose that the directory `hmm2` contains a set of accumulator files output by the first pass of HEREST running in parallel mode using as source the HMM definitions listed in `hlist` and stored in `hmm1/HMMDefs`. Then the command

```
HSmooth -c 4 -w 2.0 -H hmm1/HMMDefs -M hmm2 hlist hmm2/*.acc
```

would generate a new smoothed HMM set in `hmm2`. Here the `-w` option is used to set the minimum mixture component weight in any state to twice the value of `MINMIX`. The `-c` option sets the maximum number of iterations of the binary chop procedure to be 4.

Chapter 12

Networks, Dictionaries and Language Models



The preceding chapters have described how to process speech data and how to train various types of HMM. This and the following chapter are concerned with building a speech recogniser using HTK. This chapter focuses on the use of networks and dictionaries. A network describes the sequence of words that can be recognised and, for the case of sub-word systems, a dictionary describes the sequence of HMMs that constitute each word. A word level network will typically represent either a *Task Grammar* which defines all of the legal word sequences explicitly or a *Word Loop* which simply puts all words of the vocabulary in a loop and therefore allows any word to follow any other word. Word-loop networks are often augmented by a stochastic language model. Networks can also be used to define phone recognisers and various types of word-spotting systems.

Networks are specified using the HTK *Standard Lattice Format* (SLF) which is described in detail in Chapter 20. This is a general purpose text format which is used for representing multiple hypotheses in a recogniser output as well as word networks. Since SLF format is text-based, it can be written directly using any text editor. However, this can be rather tedious and HTK provides two tools which allow the application designer to use a higher-level representation. Firstly, the tool HPARSE allows networks to be generated from a source text containing extended BNF format grammar rules. This format was the only grammar definition language provided in earlier versions of HTK and hence HPARSE also provides backwards compatibility.

HPARSE task grammars are very easy to write, but they do not allow fine control over the actual network used by the recogniser. The tool HBUILD works directly at the SLF level to provide this detailed control. Its main function is to enable a large word network to be decomposed into a set of small self-contained sub-networks using as input an extended SLF format. This enhances the design process and avoids the need for unnecessary repetition.

HBuild can also be used to perform a number of special-purpose functions. Firstly, it can construct word-loop and word-pair grammars automatically. Secondly, it can incorporate a statistical bigram language model into a network. These can be generated from label transcriptions using HLStats. However, HTK supports the standard ARPA MIT-LL text format for backed-off N-gram language models, and hence, import from other sources is possible.

Whichever tool is used to generate a word network, it is important to ensure that the generated network represents the intended grammar. It is also helpful to have some measure of the difficulty of the recognition task. To assist with this, the tool HSGen is provided. This tool will generate example word sequences from an SLF network using random sampling. It will also estimate the perplexity of the network.

When a word network is loaded into a recogniser, a dictionary is consulted to convert each word in the network into a sequence of phone HMMs. The dictionary can have multiple pronunciations in which case several sequences may be joined in parallel to make a word. Options exist in this process to automatically convert the dictionary entries to context-dependent triphone models, either within a word or cross-word. Pronouncing dictionaries are a vital resource in building speech recognition systems and, in practice, word pronunciations can be derived from many different sources. The HTK tool HDMan enables a dictionary to be constructed automatically from different sources. Each source can be individually edited and translated and merged to form a uniform HTK format dictionary.

The various facilities for describing a word network and expanding into a HMM level network suitable for building a recogniser are implemented by the HTK library module HNet. The facilities for loading and manipulating dictionaries are implemented by the HTK library module HDict and for loading and manipulating language models are implemented by HLM. These facilities and those provided by HParse, HBuild, HSGen, HLStats and HDMan are the subject of this chapter.

12.1 How Networks are Used

Before delving into the details of word networks and dictionaries, it will be helpful to understand their rôle in building a speech recogniser using HTK. Fig 12.1 illustrates the overall recognition process. A word network is defined using HTK Standard Lattice Format (SLF). An SLF word network is just a text file and it can be written directly with a text editor or a tool can be used to build it. HTK provides two such tools, HBuild and HParse. These both take as input a textual description and output an SLF file. Whatever method is chosen, word network SLF generation is done *off-line* and is part of the system build process.

An SLF file contains a list of nodes representing words and a list of arcs representing the transitions between words. The transitions can have probabilities attached to them and these can be used to indicate *preferences* in a grammar network. They can also be used to represent bigram probabilities in a back-off bigram network and HBuild can generate such a bigram network automatically. In addition to an SLF file, a HTK recogniser requires a dictionary to supply pronunciations for each word in the network and a set of acoustic HMM phone models. Dictionaries are input via the HTK interface module HDict.

The dictionary, HMM set and word network are input to the HTK library module HNet whose function is to generate an equivalent network of HMMs. Each word in the dictionary may have several pronunciations and in this case there will be one branch in the network corresponding to each alternative pronunciation. Each pronunciation may consist either of a list of phones or a list of HMM names. In the former case, HNet can optionally expand the HMM network to use either word internal triphones or cross-word triphones. Once the HMM network has been constructed, it can be input to the decoder module HRec and used to recognise speech input. Note that HMM network construction is performed *on-line* at recognition time as part of the initialisation process.

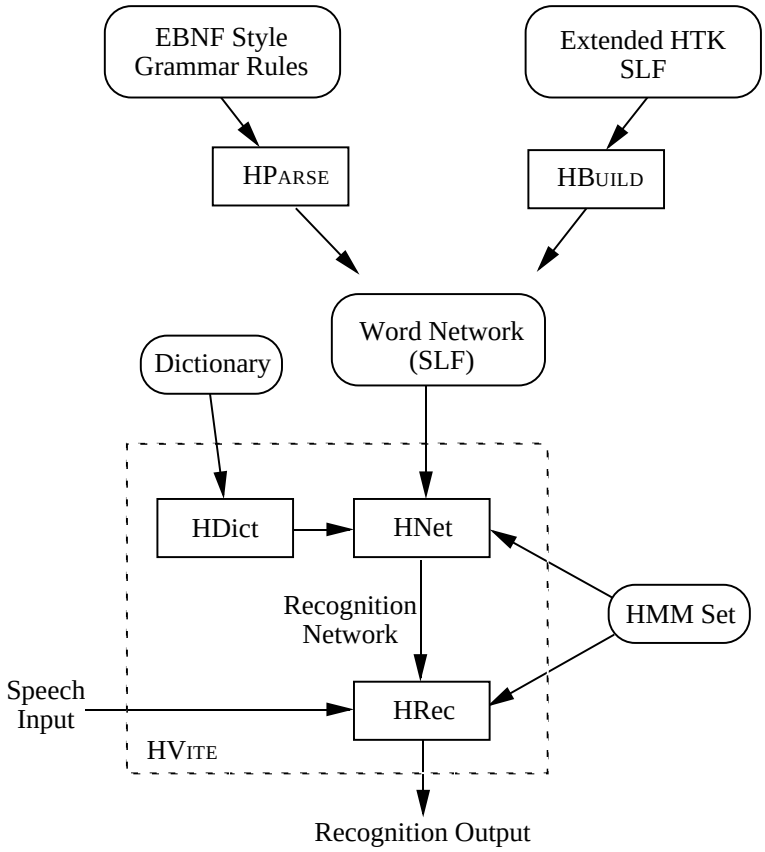


Fig. 12.1 Overview of the Recognition Process

For convenience, HTK provides a recognition tool called HVITE to allow the functions provided by HNET and HREC to be invoked from the command line. HVITE is particularly useful for running experimental evaluations on test speech stored in disk files and for basic testing using live audio input. However, application developers should note that HVITE is just a shell containing calls to load the word network, dictionary and models; generate the recognition network and then repeatedly recognise each input utterance. For embedded applications, it may well be appropriate to dispense with HVITE and call the functions in HNET and HREC directly from the application. The use of HVITE is explained in the next chapter.

12.2 Word Networks and Standard Lattice Format

This section provides a basic introduction to the HTK Standard Lattice Format (SLF). SLF files are used for a variety of functions some of which lie beyond the scope of the standard HTK package. The description here is limited to those features of SLF which are required to describe word networks suitable for input to HNET. The following Chapter describes the further features of SLF used for representing the output of a recogniser. For reference, a full description of SLF is given in Chapter 20.

A word network in SLF consists of a list of nodes and a list of arcs. The nodes represent words and the arcs represent the transition between words[1]. Each node and arc definition is written on a single line and consists of a number of fields. Each field specification consists of a "name=value" pair. Field names can be any length but all commonly used field names consist of a single letter. By convention, field names starting with a capital letter are mandatory whereas field names starting with a lower-case letter are optional. Any line beginning with a **#** is a comment and is ignored.

[1]More precisely, nodes represent the ends of words and arcs represent the transitions between word ends. This distinction becomes important when describing recognition output since acoustic scores are attached to arcs not nodes.

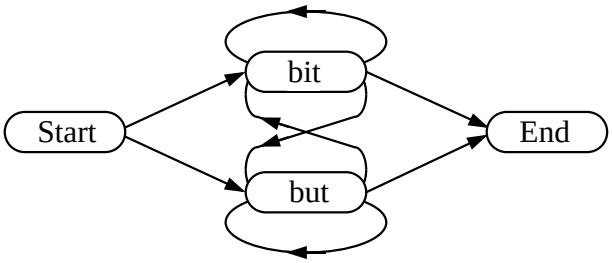


Fig. 12.2 A Simple Word Network

The following example should illustrate the basic format of an SLF word network file. It corresponds to the network illustrated in Fig 12.2 which represents all sequences consisting of the words "bit" and "but" starting with the word "start" and ending with the word "end". As will be seen later, the start and end words will be mapped to a silence model so this grammar allows speakers to say "bit but but bit bit ....etc".

```
# Define size of network: N=num nodes and L=num arcs
N=4 L=8
# List nodes: I=node-number, W=word
I=0 W=start
I=1 W=end
I=2 W=bit
I=3 W=but
# List arcs: J=arc-number, S=start-node, E=end-node
J=0 S=0 E=2
J=1 S=0 E=3
J=2 S=3 E=1
J=3 S=2 E=1
J=4 S=2 E=3
J=5 S=3 E=3
J=6 S=3 E=2
J=7 S=2 E=2
```

Notice that the first line which defines the size of the network must be given before any node or arc definitions. A node is a *network start node* if it has no predecessors, and a node is *network end node* if it has no successors. There must be one and only one network start node and one network end node. In the above, node 0 is a network start node and node 1 is a network end node. The choice of the names "start" and "end" for these nodes has no significance.

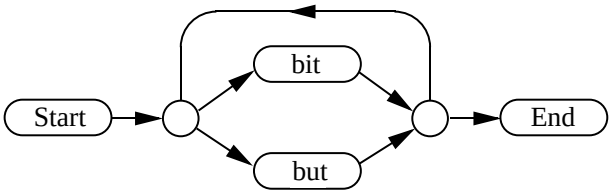


Fig. 12.3 A Word Network Using Null Nodes

A word network can have null nodes indicated by the special predefined word name `!NULL`. Null nodes are useful for reducing the number of arcs required. For example, the *Bit-But* network could be defined as follows

```
# Network using null nodes
```

```
N=6 L=7
I=0 W=start
I=1 W=end
I=2 W=bit
I=3 W=but
I=4 W=!NULL
I=5 W=!NULL
J=0 S=0 E=4
J=1 S=4 E=2
J=2 S=4 E=3
J=3 S=2 E=5
J=4 S=3 E=5
J=5 S=5 E=4
J=6 S=5 E=1
```

In this case, there is no significant saving, however, if there were many words in parallel, the total number of arcs would be much reduced by using null nodes to form common start and end points for the loop-back connections.

By default, all arcs are equally likely. However, the optional field `l=x` can be used to attach the log transition probability `x` to an arc. For example, if the word "but" was twice as likely as "bit", the arcs numbered 1 and 2 in the last example could be changed to

```
J=1 S=4 E=2 l=-1.1
J=2 S=4 E=3 l=-0.4
```

Here the probabilities have been normalised to sum to 1, however, this is not necessary. The recogniser simply adds the scaled log probability to the path score and hence it can be regarded as an additive word transition penalty.

12.3 Building a Word Network with HParse

Whilst the construction of a word level SLF network file by hand is not difficult, it can be somewhat tedious. In earlier versions of HTK, a high level grammar notation based on extended Backus-Naur Form (EBNF) was used to specify recognition grammars. This *HParse* format was read-in directly by the recogniser and compiled into a finite state recognition network at run-time.

In HTK 3.2.1, *HParse* format is still supported but in the form of an *off-line* compilation into an SLF word network which can subsequently be used to drive a recogniser.

A HParse format grammar consists of an extended form of regular expression enclosed within parentheses. Expressions are constructed from sequences of words and the metacharacters

`|` denotes alternatives

`[ ]` encloses options

{ } denotes zero or more repetitions

`< >` denotes one or more repetitions

`<< >>` denotes context-sensitive loop

The following examples will illustrate the use of all of these except the last which is a special-purpose facility provided for constructing context-sensitive loops as found in for example, context-dependent phone loops and word-pair grammars. It is described in the reference entry for HPARSE.

As a first example, suppose that a simple isolated word single digit recogniser was required. A suitable syntax would be

```
(
   one | two | three | four | five |
   six | seven | eight | nine | zero
)
```

This would translate into the network shown in part (a) of Fig. 12.4. If this HParse format syntax definition was stored in a file called `digitsyn`, the equivalent SLF word network would be generated in the file `digitnet` by typing

```
HParse digitsyn digitnet
```

The above digit syntax assumes that each input digit is properly end-pointed. This requirement can be removed by adding a silence model before and after the digit

```
(
   sil (one | two | three | four | five |
   six | seven | eight | nine | zero) sil
)
```

As shown by graph (b) in Fig. 12.4, the allowable sequence of models now consists of silence followed by a digit followed by silence. If a sequence of digits needed to be recognised then angle brackets can be used to indicate one or more repetitions, the HParse grammar

```
(
   sil < one | two | three | four | five |
   six | seven | eight | nine | zero > sil
)
```

would accomplish this. Part (c) of Fig. 12.4 shows the network that would result in this case.

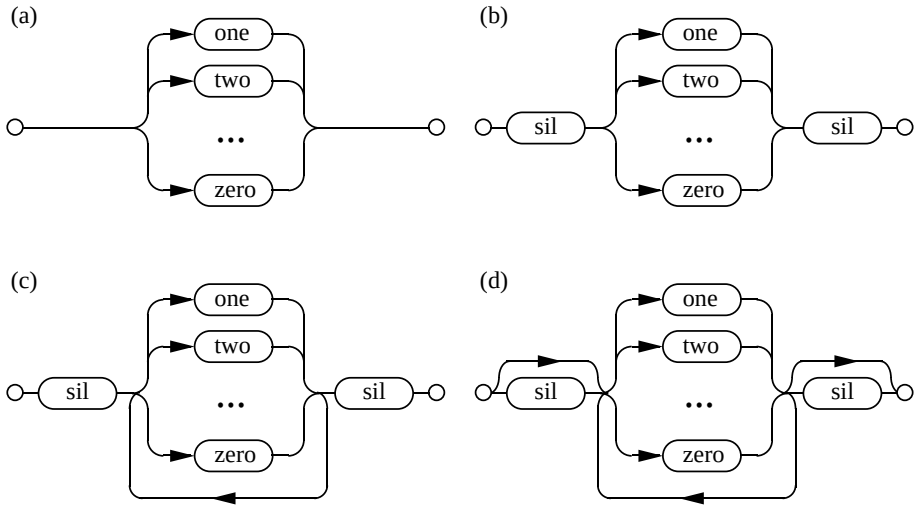


Fig. 12.4 Example Digit Recognition Networks

HParse grammars can define variables to represent sub-expressions. Variable names start with a dollar symbol and they are given values by definitions of the form

```
$var = expression ;
```

For example, the above connected digit grammar could be rewritten as

```
$digit = one | two | three | four | five |
         six | seven | eight | nine | zero;
(
   sil < $digit > sil
)
```

Here `$digit` is a variable whose value is the expression appearing on the right hand side of the assignment. Whenever the name of a variable appears within an expression, the corresponding expression is substituted. Note however that variables must be defined before use, hence, recursion is prohibited.

As a final refinement of the digit grammar, the start and end silence can be made optional by enclosing them within square brackets thus

```
$digit = one | two | three | four | five |
         six | seven | eight | nine | zero;
(
   [sil] < $digit > [sil]
)
```

Part (d) of Fig. 12.4 shows the network that would result in this last case.

HParse format grammars are a convenient way of specifying task grammars for interactive voice interfaces. As a final example, the following defines a simple grammar for the control of a telephone by voice.

```
$digit  = one | two | three | four | five |
          six | seven | eight | nine | zero;
$number = $digit { [pause] $digit};
$scode  = shortcode $digit $digit;
$telnum = $scode | $number;
$cmd    = dial $telnum |
          enter $scode for $number |
          redial | cancel;
$noise  = lipsmack | breath | background;
( < $cmd | $noise > )
```

The dictionary entries for `pause`, `lipsmack`, `breath` and `background` would reference HMMs trained to model these types of noise and the corresponding output symbols in the dictionary would be null.

Finally, it should be noted that when the HParse format was used in earlier versions of HTK, word grammars contained word pronunciations embedded within them. This was done by using the reserved node names `WD_BEGIN` and `WD_END` to delimit word boundaries. To provide backwards compatiblity, HPARSE can process these old format networks but when doing so it outputs a dictionary as well as a word network. This compatibility mode is defined fully in the reference section, to use it the configuration variable `V1COMPAT` must be set true or the `-c` option set.

Finally on the topic of word networks, it is important to note that any network containing an unbroken loop of one or more tee-models will generate an error. For example, if `sp` is a single state tee-model used to represent short pauses, then the following network would generate an error

```
( sil < sp | $digit > sil )
```

the intention here is to recognise a sequence of digits which may optionally be separated by short pauses. However, the syntax allows an endless sequence of `sp` models and hence, the recogniser could traverse this sequence without ever consuming any input. The solution to problems such as these is to rearrange the network. For example, the above could be written as

```
( sil < $digit sp > sil )
```

12.4 Bigram Language Models

Before continuing with the description of network generation and, in particular, the use of HBUILD, the use of bigram language models needs to be described. Support for statistical language models in HTK is provided by the library module HLM. Although the interface to HLM can support general N-grams, the facilities for constructing and using N-grams are limited to bigrams.

A bigram language model can be built using HLSTATS invoked as follows where it is a assumed that all of the label files used for training are stored in an MLF called `labs`

```
HLStats -b bigfn -o wordlist labs
```

All words used in the label files must be listed in the `wordlist`. This command will read all of the transcriptions in `labs`, build a table of bigram counts in memory, and then output a back-off bigram to the file `bigfn`. The formulae used for this are given in the reference entry for HLSTATS. However, the basic idea is encapsulated in the following formula

$$p(i,j) = \begin{cases} (N(i,j) - D)/N(i) & \text{if } N(i,j) > t \\ b(i)p(j) & \text{otherwise} \end{cases}$$

where $N(i,j)$ is the number of times word j follows word i and $N(i)$ is the number of times that word i appears. Essentially, a small part of the available probability mass is deducted from the higher bigram counts and distributed amongst the infrequent bigrams. This process is called *discounting*. The default value for the discount constant D is 0.5 but this can be altered using the configuration variable `DISCOUNT`. When a bigram count falls below the threshold t, the bigram is backed-off to the unigram probability suitably scaled by a back-off weight in order to ensure that all bigram probabilities for a given history sum to one.

Backed-off bigrams are stored in a text file using the standard ARPA MIT-LL format which as used in HTK is as follows

```
\data\
ngram 1=<num 1-grams>
ngram 2=<num 2-ngrams>

\1-grams:
P(!ENTER)       !ENTER  B(!ENTER)
P(W1)            W1     B(W1)
P(W2)            W2     B(W2)
...
P(!EXIT)        !EXIT   B(!EXIT)

\2-grams:
P(W1 | !ENTER)  !ENTER W1
P(W2 | !ENTER)  !ENTER W2
P(W1 | W1)      W1     W1
P(W2 | W1)      W1     W2
P(W1 | W2)      W2     W1
....
P(!EXIT | W1)   W1     !EXIT
P(!EXIT | W2)   W2     !EXIT
\end\
```

where all probabilities are stored as base-10 logs. The default start and end words, `!ENTER` and `!EXIT` can be changed using the HLSTATS `-s` option.

For some applications, a simple matrix style of bigram representation may be more appropriate. If the `-o` option is omitted in the above invocation of HLSTATS, then a simple full bigram matrix will be output using the format

```
!ENTER    0   P(W1 | !ENTER) P(W2 | !ENTER) .....
W1        0   P(W1 | W1)     P(W2 | W1)     .....
W2        0   P(W1 | W2)     P(W2 | W2)     .....
...
!EXIT     0   PN             PN             .....
```

where the probability $P(w_j|w_i)$ is given by row i,j of the matrix. If there are a total of N words in the vocabulary then `PN` in the above is set to $1/(N+1)$, this ensures that the last row sums to one. As a very crude form of smoothing, a floor can be set using the `-f minp` option to prevent any entry falling below `minp`. Note, however, that this does not affect the bigram entries in the first column which are zero by definition. Finally, as with the storage of tied-mixture and discrete probabilities, a run-length encoding scheme is used whereby any value can be followed by an asterisk and a repeat count (see section 7.5).

12.5 Building a Word Network with HBuild

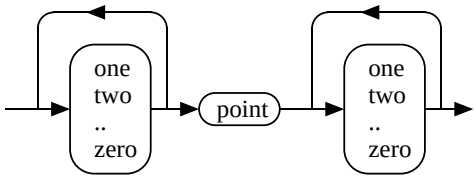


Fig. 12.5 Decimal Syntax

As mentioned in the introduction, the main function of HBUILD is allow a word-level network to be constructed from a main lattice and a set of sub-lattices. Any lattice can contain node definitions which refer to other lattices. This allows a word-level recognition network to be decomposed into a number of sub-networks which can be reused at different points in the network.

For example, suppose that decimal number input was required. A suitable network structure would be as shown in Fig. 12.5. However, to write this directly in an SLF file would require the digit loop to be written twice. This can be avoided by defining the digit loop as a sub-network and referencing it within the main *decimal* network as follows

```
# Digit network
SUBLAT=digits
N=14 L=21
# define digits
I=0  W=zero
I=1  W=one
I=2  W=two
...
I=9  W=nine
#  enter/exit & loop-back null nodes
I=10 W=!NULL
I=11 W=!NULL
I=12 W=!NULL
I=13 W=!NULL
# null->null->digits
J=0 S=10 E=11
J=1 S=11 E=0
J=2 S=11 E=1
...
J=10 S=11 E=9
# digits->null->null
J=11 S=0 E=12
...
J=19 S=9 E=12
J=20 S=12 E=13
# finally add loop back
J=21 S=12 E=11
.

# Decimal netork
N=5 L=4
# digits -> point -> digits
I=0 W=start
I=1 L=digits
I=2 W=pause
I=3 L=digits
I=4 W=end
```

```
# digits -> point -> digits
J=0 S=0 E=1
J=1 S=1 E=2
J=2 S=2 E=3
J=3 S=3 E=4
```

The sub-network is identified by the field `SUBLAT` in the header and it is terminated by a single period on a line by itself. The main body of the sub-network is written as normal. Once defined, a sub-network can be substituted into a higher level network using an `L` field in a node definition, as in nodes 1 and 3 of the decimal network above.

Of course, this process can be continued and a higher level network could reference the decimal network wherever it needed decimal number entry.

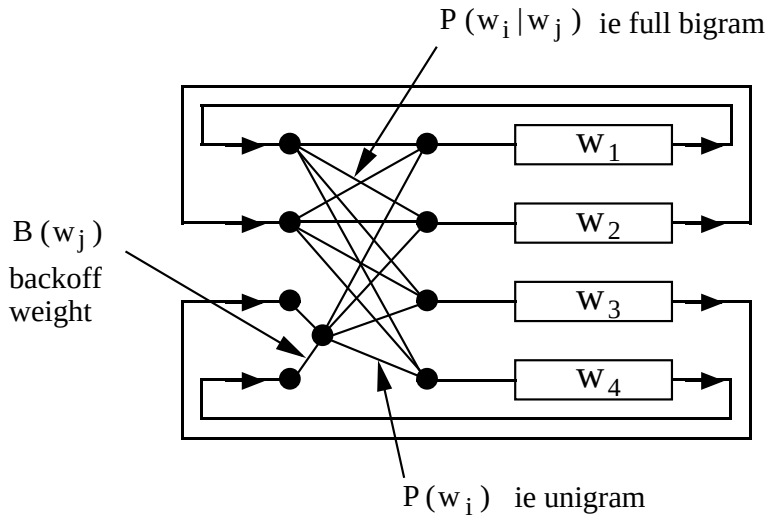


Fig. 12.6 Back-off Bigram Word-Loop Network

One of the commonest form of recognition network is the word-loop where all vocabulary items are placed in parallel with a loop-back to allow any word sequence to be recognised. This is the basic arrangement used in most dictation or transcription applications. HBuild can build such a loop automatically from a list of words. It can also read in a bigram in either ARPA MIT-LL format or HTK matrix format and attach a bigram probability to each word transition. Note, however, that using a full bigram language model means that every distinct pair of words must have its own unique loop-back transition. This increases the size of the network considerably and slows down the recogniser. When a back-off bigram is used, however, backed-off transitions can share a common loop-back transition. Fig. 12.6 illustrates this. When backed-off bigrams are input via an ARPA MIT-LL format file, HBuild will exploit this where possible.

Finally, HBuild can automatically construct a word-pair grammar as used in the ARPA Naval Resource Management task.

12.6 Testing a Word Network using HSGen

When designing task grammars, it is useful to be able to check that the language defined by the final word network is as envisaged. One simple way to check this is to use the network as a generator by randomly traversing it and outputting the name of each word node encountered. HTK provides a very simple tool called HSGen for doing this.

As an example if the file `bnet` contained the simple Bit-But network described above and the file `bdic` contained a corresponding dictionary then the command

```
HSGen bnet bdic
```

would generate a random list of examples of the language defined by `bnet`, for example,

```
start bit but bit bit bit end
start but bit but but end
start bit bit but but end
.... etc
```

This is perhaps not too informative in this case but for more complex grammars, this type of output can be quite illuminating.

HSGen will also estimate the empirical entropy by recording the probability of each sentence generated. To use this facility, it is best to suppress the sentence output and generate a large number of examples. For example, executing

```
HSGen -s -n 1000 -q bnet bdic
```

where the `-s` option requests statistics, the `-q` option suppresses the output and `-n 1000` asks for 1000 sentences would generate the following output

```
Number of Nodes = 4 [0 null], Vocab Size = 4
Entropy = 1.156462,  Perplexity = 2.229102
1000 Sentences: average len = 5.1, min=3, max=19
```

12.7 Constructing a Dictionary

As explained in section 12.1, the word level network is expanded by HNet to create the network of HMM instances needed by the recogniser. The way in which each word is expanded is determined from a dictionary.

A dictionary for use in HTK has a very simple format. Each line consists of a single word pronunciation with format

```
WORD [ '['OUTSYM']' ] [PRONPROB] P1 P2 P3 P4 ....
```

where `WORD` represents the word, followed by the optional parameters `OUTSYM` and `PRONPROB`, where `OUTSYM` is the symbol to output when that word is recognised (which must be enclosed in square brackets, `[` and `]`) and `PRONPROB` is the pronunciation probability (0.0 - 1.0). `P1`, `P2`, ... is the sequence of phones or HMMs to be used in recognising that word. The output symbol and the pronunciation probability are optional. If an output symbol is not specified, the name of the word itself is output. If a pronunciation probability is not specified then a default of 1.0 is assumed. Empty square brackets, `[]`, can be used to suppress any output when that word is recognised. For example, a dictionary might contain

```
bit           b  ih t
but           b  ah t
dog    [woof] d  ao g
cat    [meow] k  ae t
start  []     sil
end    []     sil
```

If any word has more than one pronunciation, then the word has a repeated entry, for example,

```
the           th iy
the           th ax
```

corresponding to the stressed and unstressed forms of the word "the".

The pronunciations in a dictionary are normally at the phone level as in the above examples. However, if context-dependent models are wanted, these can be included directly in the dictionary. For example, the Bit-But entries might be written as

```
bit           b+ih  b-ih+t  ih-t
but           b+ah  b-ah+t  ah-t
```

In principle, this is never necessary since HNet can perform context expansion automatically, however, it saves computation to do this off-line as part of the dictionary construction process. Of course, this is only possible for word-internal context dependencies. Cross-word dependencies can only be generated by HNet.

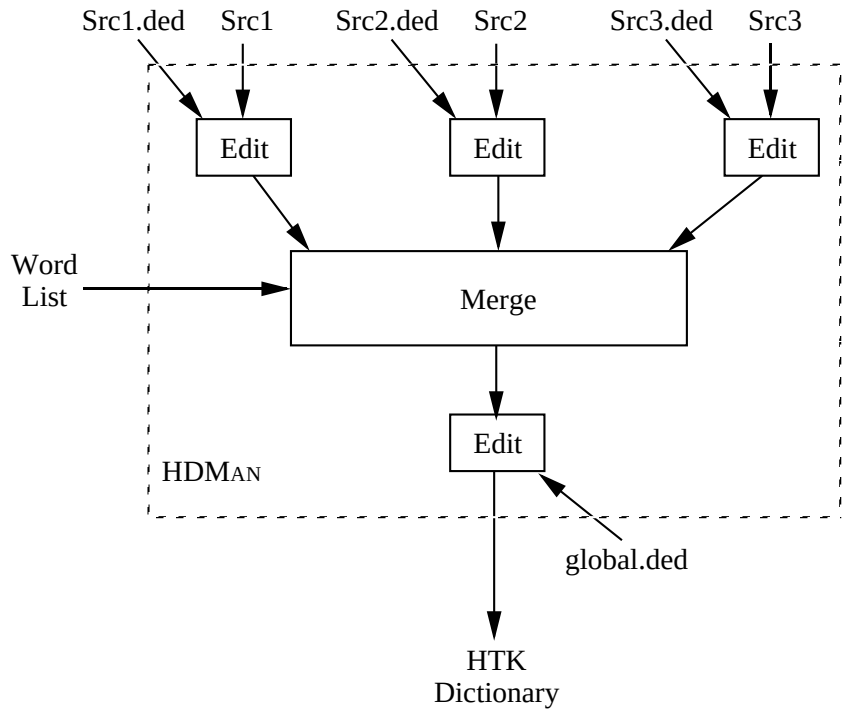


Fig. 12.7 Dictionary Construction using HDMan

Pronouncing dictionaries are a valuable resource and if produced manually, they can require considerable investment. There are a number of commercial and public domain dictionaries available, however, these will typically have differing formats and will use different phone sets. To assist in the process of dictionary construction, HTK provides a tool called HDMan which can be used to edit and merge differing source dictionaries to form a single uniform dictionary. The way that HDMan works is illustrated in Fig. 12.7.

Each source dictionary file must have one pronunciation per line and the words must be sorted into alphabetical order. The word entries must be valid HTK strings as defined in section 4.6. If an arbitrary character sequence is to be allowed, then the input edit script should have the command `IM RAW` as its first command.

The basic operation of HDMan is to scan the input streams and for each new word encountered, copy the entry to the output. In the figure, a word list is also shown. This is optional but if included HDMan only copies words in the list. Normally, HDMan copies just the first pronunciation that it finds for any word. Thus, the source dictionaries are usually arranged in order of *reliability*, possibly preceded by a small dictionary of special word pronunciations. For example, in Fig. 12.7, the main dictionary might be `Src2`. `Src1` might be a small dictionary containing correct pronunciations for words in `Src2` known to have errors in them. Finally, `Src3` might be a large poor quality dictionary (for example, it could be generated by a rule-based text-to-phone system) which is included as a last resort source of pronunciations for words not in the main dictionary.

As shown in the figure, HDMan can apply a set of editing commands to each source dictionary and it can also edit the output stream. The commands available are described in full in the reference section. They operate in a similar way to those in HLEd. Each set of commands is written in an edit script with one command per line. Each input edit script has the same name as the corresponding source dictionary but with the extension `.ded` added. The output edit script is stored in a file called `global.ded`. The commands provided include replace and delete at the word and phone level, context-sensitive replace and automatic conversions to left biphones, right biphones and word internal triphones.

When HDMan loads a dictionary it adds word boundary symbols to the start and end of each pronunciation and then deletes them when writing out the new dictionary. The default for these word boundary symbols is `#` but it can be redefined using the `-b` option. The reason for this is to allow context-dependent edit commands to take account of word-initial and word-final phone positions. The examples below will illustrate this.

Rather than go through each HDMan edit command in detail, some examples will illustrate the typical manipulations that can be performed by HDMan. Firstly, suppose that a dictionary transcribed unstressed "-ed" endings as `ih0 d` but the required dictionary does not mark stress but uses a schwa in such cases, that is, the transformations

```
ih0 d  #   ->   ax d
ih0       ->   ih  (otherwise)
```

are required. These could be achieved by the following 3 commands

```
MP axd0 ih0 d #
SP axd0 ax d #
RP ih ih0
```

The context sensitive replace is achieved by merging all sequences of `ih0 d #` and then splitting the result into the sequence `ax d #`. The final `RP` command then unconditionally replaces all occurrences of `ih0` by `ih`. As a second similar example, suppose that all examples of `ax l` (as in "bottle") are to be replaced by the single phone `el` provided that the immediately following phone is a non-vowel. This requires the use of the `DC` command to define a context consisting of all non-vowels, then a merge using `MP` as above followed by a context-sensitive replace

```
DC nonv l r w y .... m n ng #
MP axl ax l
CR el * axl nonv
SP axl ax l
```

the final step converts all non-transformed cases of `ax l` back to their original form.

As a final example, a typical output transformation applied via the edit script `global.ded` will convert all phones to context-dependent form and append a short pause model `sp` at the end of each pronunciation. The following two commands will do this

```
TC
AS sp
```

For example, these commands would convert the dictionary entry

```
BAT b ah t
```

into

```
BAT b+ah b-ah+t ah-t sp
```

Finally, if the `-l` option is set, HDMan will generate a log file containing a summary of the pronunciations used from each source and how many words, if any are missing. It is also possible to give HDMan a phone list using the `-n` option. In this case, HDMan will record how many times each phone was used and also, any phones that appeared in pronunciations but are not in the phone list. This is useful for detecting errors and unexpected phone symbols in the source dictionary.

12.8 Word Network Expansion

Now that word networks and dictionaries have been explained, the conversion of word level networks to model-based recognition networks will be described. Referring again to Fig 12.1, this expansion is performed automatically by the module HNet. By default, HNet attempts to infer the required expansion from the contents of the dictionary and the associated list of HMMs. However, 5 configurations parameters are supplied to apply more precise control where required: `ALLOWCXTEXP`, `ALLOWXWRDEXP`, `FORCECXTEXP`, `FORCELEFTBI` and `FORCERIGHTBI`.

The expansion proceeds in four stages.

1. *Context definition*
   The first step is to determine how model names are constructed from the dictionary entries and whether cross-word context expansion should be performed. The dictionary is scanned and each distinct phone is classified as either

(a) *Context Free*
In this case, the phone is skipped when determining context. An example is a model (`sp`) for short pauses. This will typically be inserted at the end of every word pronunciation but since it tends to cover a very short segment of speech it should not block context-dependent effects in a cross-word triphone system.

(b) *Context Independent*
The phone only exists in context-independent form. A typical example would be a silence model (`sil`). Note that the distinction that would be made by HNet between `sil` and `sp` is that whilst both would only appear in the HMM set in context-independent form, `sil` would appear in the contexts of other phones whereas `sp` would not.

(c) *Context Dependent*
This classification depends on whether a phone appears in the context part of the name and whether any context dependent versions of the phone exist in the HMMSet. Context Dependent phones will be subject to model name expansion.

2. *Determination of network type*
The default behaviour is to produce the simplest network possible. If the dictionary is closed (every phone name appears in the HMM list), then no expansion of phone names is performed. The resulting network is generated by straightforward substitution of each dictionary pronunciation for each word in the word network. If the dictionary is not closed, then if word internal context expansion would find each model in the HMM set then word internal context expansion is used. Otherwise, full cross-word context expansion is applied.

   The determination of the network type can be modified by using the configuration parameters mentioned earlier. By default `ALLOWCXTEXP` is set true. If `ALLOWCXTEXP` is set false, then no expansion of phone names is performed and each phone corresponds to the model of the same name. The default value of `ALLOWXWRDEXP` is false thus preventing context expansion across word boundaries. This also limits the expansion of the phone labels in the dictionary to word internal contexts only. If `FORCECXTEXP` is set true, then context expansion will be performed. For example, if the HMM set contained all monophones, all biphones and all triphones, then given a monophone dictionary, the default behaviour of HNet would be to generate a monophone recognition network since the dictionary would be closed. However, if `FORCECXTEXP` is set true and `ALLOWXWRDEXP` is set false then word internal context expansion will be performed. If `FORCECXTEXP` is set true and `ALLOWXWRDEXP` is set true then full cross-word context expansion will be performed.

3. *Network expansion*
Each word in the word network is transformed into a *word-end* node preceded by the sequence of model nodes corresponding to the word's pronunciation. For cross word context expansion, the initial and final context dependent phones (and any preceding/following context independent ones) are duplicated as many times as is necessary to cater for each different cross word context. Each duplicated word-final phone is followed by a similarly duplicated word-end node. Null words are simply transformed into word-end nodes with no preceding model nodes.

4. *Linking of models to network nodes*
Each model node is linked to the corresponding HMM definition. In each case, the required HMM model name is determined from the phone name and the surrounding context names. The algorithm used for this is

   (a) Construct the context-dependent name and see if the corresponding model exists.

   (b) Construct the context-independent name and see if the corresponding model exists.

   If the configuration variable `ALLOWCXTEXP` is false (a) is skipped and if the configuration variable `FORCECXTEXP` is true (b) is skipped. If no matching model is found, an error is generated. When the right context is a boundary or `FORCELEFTBI` is true, then the context-dependent name takes the form of a left biphone, that is, the phone `p` with left context `l` becomes `l-p`. When the left context is a boundary or `FORCERIGHTBI` is true, then the context-dependent name takes the form of a right biphone, that is, the phone `p` with right context `r` becomes `p+r`. Otherwise, the context-dependent name is a full triphone, that is, `l-p+r`. Context-free phones are skipped in this process so

```
sil aa r sp y uw sp sil
```

would be expanded as

```
sil sil-aa+r aa-r+y sp r-y+uw y-uw+sil sp sil
```

assuming that `sil` is context-independent and `sp` is context-free. For word-internal systems, the context expansion can be further controlled via the configuration variable `CFWORDBOUNDARY`. When set true (default setting) context-free phones will be treated as word boundaries so

```
aa r sp y uw sp
```

would be expanded to

```
aa+r aa-r sp y+uw y-uw sp
```

Setting `CFWORDBOUNDARY` false would produce

```
aa+r aa-r+y sp r-y+uw y-uw sp
```

Note that in practice, stages (3) and (4) above actually proceed concurrently so that for the first and last phone of context-dependent models, logical models which have the same underlying physical model can be merged.

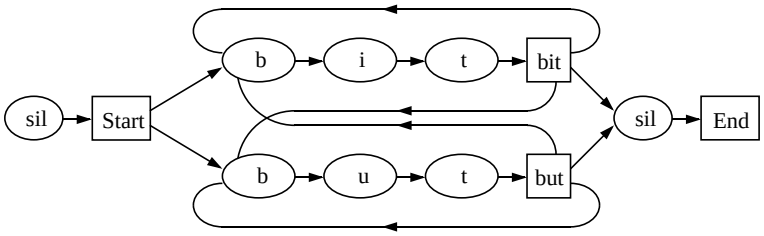


Fig. 12.8 Monophone Expansion of Bit-But Network

Having described the expansion process in some detail, some simple examples will help clarify the process. All of these are based on the Bit-But word network illustrated in Fig. 12.2. Firstly, assume that the dictionary contains simple monophone pronunciations, that is

```
bit         b  i  t
but         b  u  t
start       sil
end         sil
```

and the HMM set consists of just monophones

```
b  i  t  u  sil
```

In this case, HNET will find a closed dictionary. There will be no expansion and it will directly generate the network shown in Fig 12.8. In this figure, the rounded boxes represent model nodes and the square boxes represent word-end nodes.

Similarly, if the dictionary contained word-internal triphone pronunciations such as

```
bit         b+i  b-i+t  i-t
but         b+u  b-u+t  u-t
start       sil
end         sil
```

and the HMM set contains all the required models

```
b+i  b-i+t  i-t b+u  b-u+t  u-t  sil
```

then again HNET will find a closed dictionary and the network shown in Fig. 12.9 would be generated.

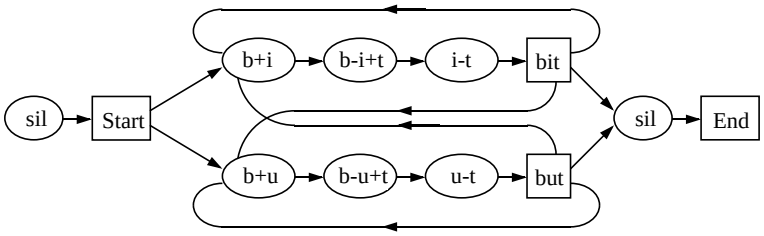


Fig. 12.9 Word Internal Triphone Expansion of Bit-But Network

If however the dictionary contained just the simple monophone pronunciations as in the first case above, but the HMM set contained just triphones, that is

```
sil-b+i  t-b+i  b-i+t  i-t+sil  i-t+b
sil-b+u  t-b+u  b-u+t  u-t+sil  u-t+b  sil
```

then HNET would perform full cross-word expansion and generate the network shown in Fig. 12.10.

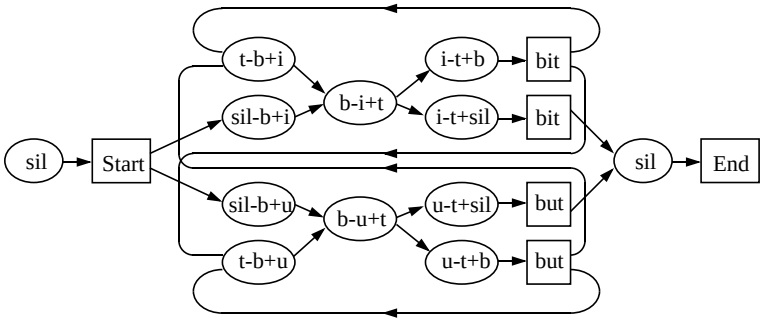


Fig. 12.10 Cross-Word Triphone Expansion of Bit-But Network

Now suppose that still using the simple monophone pronunciations, the HMM set contained all monophones, biphones and triphones. In this case, the default would be to generate the monophone network of Fig 12.8. If `FORCECXTEXP` is true but `ALLOWXWRDEXP` is set false then the word-internal network of Fig. 12.9 would be generated. Finally, if both `FORCECXTEXP` and `ALLOWXWRDEXP` are set true then the cross-word network of Fig. 12.10 would be generated.

12.9 Other Kinds of Recognition System

Although the recognition facilities of HTK are aimed primarily at sub-word based connected word recognition, it can nevertheless support a variety of other types of recognition system.

To build a phoneme recogniser, a word-level network is defined using an SLF file in the usual way except that each "word" in the network represents a single phone. The structure of the network will typically be a loop in which all phones loop back to each other.

The dictionary then contains an entry for each "word" such that the word and the pronunciation are the same, for example, the dictionary might contain

```
ih ih
eh eh
ah ah
... etc
```

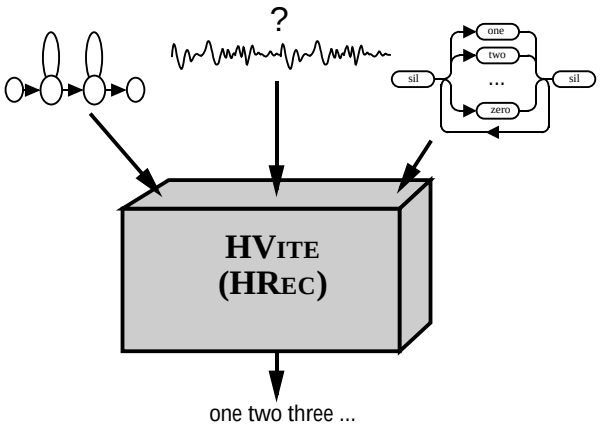
Phoneme recognisers often use biphones to provide some measure of context-dependency. Provided that the HMM set contains all the necessary biphones, then HNet will expand a simple phone loop into a context-sensitive biphone loop simply by setting the configuration variable `FORCELEFTBI` or `FORCERIGHTBI` to true, as appropriate.

Whole word recognisers can be set-up in a similar way. The word network is designed using the same considerations as for a sub-word based system but the dictionary gives the name of the whole-word HMM in place of each word pronunciation.

Finally, word spotting systems can be defined by placing each keyword in a word network in parallel with the appropriate filler models. The keywords can be whole-word models or subword based. Note in this case that word transition penalties placed on the transitions can be used to gain fine control over the false alarm rate.

Chapter 13

Decoding



The previous chapter has described how to construct a recognition network specifying what is allowed to be spoken and how each word is pronounced. Given such a network, its associated set of HMMs, and an unknown utterance, the probability of any path through the network can be computed. The task of a decoder is to find those paths which are the most likely.

As mentioned previously, decoding in HTK is performed by a library module called HREC. HREC uses the token passing paradigm to find the best path and, optionally, multiple alternative paths. In the latter case, it generates a lattice containing the multiple hypotheses which can if required be converted to an N-best list. To drive HREC from the command line, HTK provides a tool called HVITE. As well as providing basic recognition, HVITE can perform forced alignments, lattice rescoring and recognise direct audio input.

To assist in evaluating the performance of a recogniser using a test database and a set of reference transcriptions, HTK also provides a tool called HRESULTS to compute word accuracy and various related statistics. The principles and use of these recognition facilities are described in this chapter.

13.1 Decoder Operation

As described in Chapter 12 and illustrated by Fig. 12.1, decoding in HTK is controlled by a recognition network compiled from a word-level network, a dictionary and a set of HMMs. The recognition network consists of a set of nodes connected by arcs. Each node is either a HMM model instance or a word-end. Each model node is itself a network consisting of states connected by arcs. Thus, once fully compiled, a recognition network ultimately consists of HMM states connected by transitions. However, it can be viewed at three different levels: word, model and state. Fig. 13.1 illustrates this hierarchy.

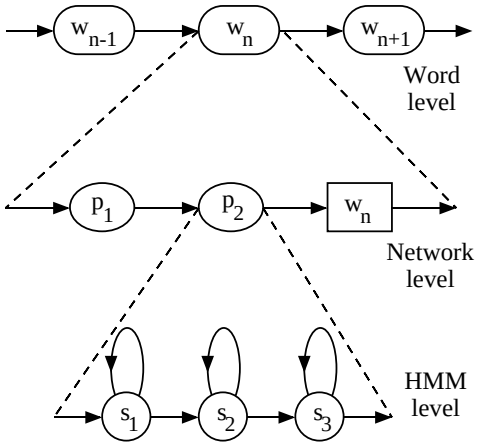


Fig. 13.1 Recognition Network Levels

For an unknown input utterance with T frames, every path from the start node to the exit node of the network which passes through exactly T emitting HMM states is a potential recognition hypothesis. Each of these paths has a log probability which is computed by summing the log probability of each individual transition in the path and the log probability of each emitting state generating the corresponding observation. Within-HMM transitions are determined from the HMM parameters, between-model transitions are constant and word-end transitions are determined by the language model likelihoods attached to the word level networks.

The job of the decoder is to find those paths through the network which have the highest log probability. These paths are found using a *Token Passing* algorithm. A token represents a partial path through the network extending from time 0 through to time t. At time 0, a token is placed in every possible start node.

Each time step, tokens are propagated along connecting transitions stopping whenever they reach an emitting HMM state. When there are multiple exits from a node, the token is copied so that all possible paths are explored in parallel. As the token passes across transitions and through nodes, its log probability is incremented by the corresponding transition and emission probabilities. A network node can hold at most N tokens. Hence, at the end of each time step, all but the N best tokens in any node are discarded.

As each token passes through the network it must maintain a history recording its route. The amount of detail in this history depends on the required recognition output. Normally, only word sequences are wanted and hence, only transitions out of word-end nodes need be recorded. However, for some purposes, it is useful to know the actual model sequence and the time of each model to model transition. Sometimes a description of each path down to the state level is required. All of this information, whatever level of detail is required, can conveniently be represented using a lattice structure.

Of course, the number of tokens allowed per node and the amount of history information requested will have a significant impact on the time and memory needed to compute the lattices. The most efficient configuration is $N = 1$ combined with just word level history information and this is sufficient for most purposes.

A large network will have many nodes and one way to make a significant reduction in the computation needed is to only propagate tokens which have some chance of being amongst the eventual winners. This process is called *pruning*. It is implemented at each time step by keeping a record of the best token overall and de-activating all tokens whose log probabilities fall more than a *beam-width* below the best. For efficiency reasons, it is best to implement primary pruning at the model rather than the state level. Thus, models are deactivated when they have no tokens in any state within the beam and they are reactivated whenever active tokens are propagated into them. State-level pruning is also implemented by replacing any token by a null (zero probability) token if it falls outside of the beam. If the pruning beam-width is set too small then the most likely path might be pruned before its token reaches the end of the utterance. This results in a *search error*. Setting the beam-width is thus a compromise between speed and avoiding search errors.

When using word loops with bigram probabilities, tokens emitted from word-end nodes will have a language model probability added to them before entering the following word. Since the range of language model probabilities is relatively small, a narrower beam can be applied to word-end nodes without incurring additional search errors. This beam is calculated relative to the best word-end token and it is called a *word-end beam*. In the case, of a recognition network with an arbitrary topology, word-end pruning may still be beneficial but this can only be justified empirically.

Finally, a third type of pruning control is provided. An upper-bound on the allowed use of compute resource can be applied by setting an upper-limit on the number of models in the network which can be active simultaneously. When this limit is reached, the pruning beam-width is reduced in order to prevent it being exceeded.

13.2 Decoder Organisation

The decoding process itself is performed by a set of core functions provided within the library module HREC. The process of recognising a sequence of utterances is illustrated in Fig. 13.2.

The first stage is to create a *recogniser-instance*. This is a data structure containing the compiled recognition network and storage for storing tokens. The point of encapsulating all of the information and storage needed for recognition into a single object is that HREC is re-entrant and can support multiple recognisers simultaneously. Thus, although this facility is not utilised in the supplied recogniser HVITE, it does provide applications developers with the capability to have multiple recognisers running with different networks.

Once a recogniser has been created, each unknown input is processed by first executing a *start recogniser* call, and then processing each observation one-by-one. When all input observations have been processed, recognition is completed by generating a lattice. This can be saved to disk as a standard lattice format (SLF) file or converted to a transcription.

The above decoder organisation is extremely flexible and this is demonstrated by the HTK tool HVITE which is a simple shell program designed to allow HREC to be driven from the command line.

Firstly, input control in the form of a recognition network allows three distinct modes of operation

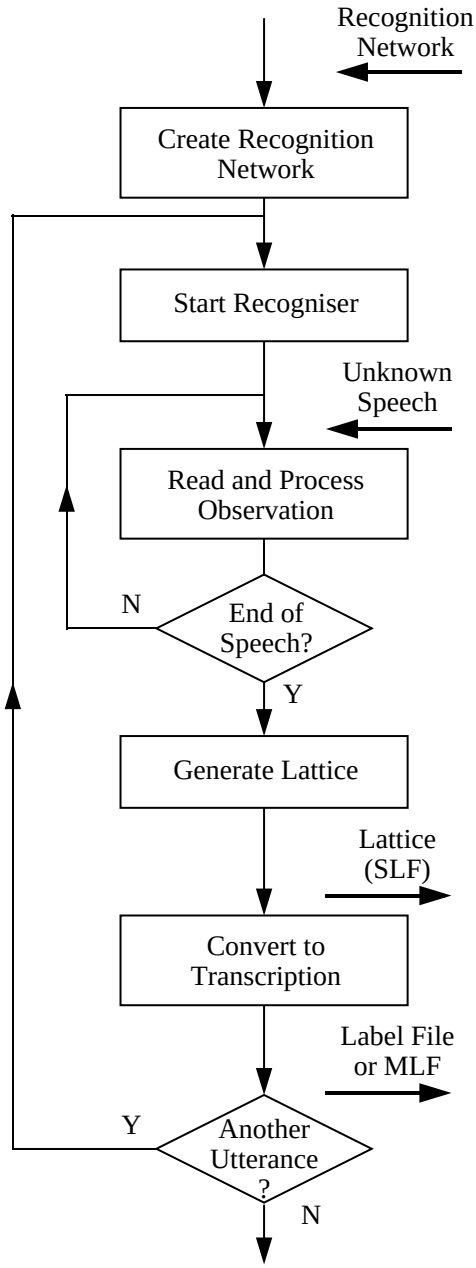


Fig. 13.2 Recognition Processing

1. *Recognition*
   This is the conventional case in which the recognition network is compiled from a task level word network.

2. *Forced Alignment*
   In this case, the recognition network is constructed from a word level transcription (i.e. orthography) and a dictionary. The compiled network may include optional silences between words and pronunciation variants. Forced alignment is often useful during training to automatically derive phone level transcriptions. It can also be used in automatic annotation systems.

3. *Lattice-based Rescoring*
   In this case, the input network is compiled from a lattice generated during an earlier recognition run. This mode of operation can be extremely useful for recogniser development since rescoring can be an order of magnitude faster than normal recognition. The required lattices are usually generated by a basic recogniser running with multiple tokens, the idea being to

generate a lattice containing both the correct transcription plus a representative number of confusions. Rescoring can then be used to quickly evaluate the performance of more advanced recognisers and the effectiveness of new recognition techniques.

The second source of flexibility lies in the provision of multiple tokens and recognition output in the form of a lattice. In addition to providing a mechanism for rescoring, lattice output can be used as a source of multiple hypotheses either for further recognition processing or input to a natural language processor. Where convenient, lattice output can easily be converted into N-best lists.

Finally, since HREC is explicitly driven step-by-step at the observation level, it allows fine control over the recognition process and a variety of traceback and on-the-fly output possibilities.

For application developers, HREC and the HTK library modules on which it depends can be linked directly into applications. It will also be available in the form of an industry standard API. However, as mentioned earlier the HTK toolkit also supplies a tool called HVITE which is a shell program designed to allow HREC to be driven from the command line. The remainder of this chapter will therefore explain the various facilities provided for recognition from the perspective of HVITE.

13.3 Recognition using Test Databases

When building a speech recognition system or investigating speech recognition algorithms, performance must be monitored by testing on databases of test utterances for which reference transcriptions are available. To use HVITE for this purpose it is invoked with a command line of the form

```
HVite -w wdnet dict hmmlist testf1 testf2 ....
```

where `wdnet` is an SLF file containing the word level network, `dict` is the pronouncing dictionary and hmmlist contains a list of the HMMs to use. The effect of this command is that HVITE will use HNET to compile the word level network and then use HREC to recognise each test file. The parameter kind of these test files must match exactly with that used to train the HMMs. For evaluation purposes, test files are normally stored in parameterised form but only the basic static coefficients are saved on disk. For example, delta parameters are normally computed during loading. As explained in Chapter 5, HTK can perform a range of parameter conversions on loading and these are controlled by configuration variables. Thus, when using HVITE, it is normal to include a configuration file via the `-C` option in which the required target parameter kind is specified. Section 13.7 below on processing direct audio input explains the use of configuration files in more detail.

In the simple default form of invocation given above, HVITE would expect to find each HMM definition in a separate file in the current directory and each output transcription would be written to a separate file in the current directory. Also, of course, there will typically be a large number of test files.

In practice, it is much more convenient to store HMMs in master macro files (MMFs), store transcriptions in master label files (MLFs) and list data files in a script file. Thus, a more common form of the above invocation would be

```
HVite -T 1 -S test.scp -H hmmset -i results -w wdnet dict hmmlist
```

where the file `test.scp` contains the list of test file names, `hmmset` is an MMF containing the HMM definitions[1], and `results` is the MLF for storing the recognition output.

As shown, it is usually a good idea to enable basic progress reporting by setting the trace option as shown. This will cause the recognised word string to be printed after processing each file. For example, in a digit recognition task the trace output might look like

```
File: testf1.mfc
SIL ONE NINE FOUR SIL
[178 frames] -96.1404 [Ac=-16931.8 LM=-181.2] (Act=75.0)
```

[1] Large HMM sets will often be distributed across a number of MMF files, in this case, the `-H` option will be repeated for each file.

where the information listed after the recognised string is the total number of frames in the utterance, the average log probability per frame, the total acoustic likelihood, the total language model likelihood and the average number of active models.

The corresponding transcription written to the output MLF form will contain an entry of the form

```
"testf1.rec"
       0  6200000 SIL  -6067.333008
 6200000  9200000 ONE  -3032.359131
 9200000 12300000 NINE -3020.820312
12300000 17600000 FOUR -4690.033203
17600000 17800000 SIL   -302.439148
.
```

This shows the start and end time of each word and the total log probability. The fields output by HVite can be controlled using the `-o`. For example, the option `-o ST` would suppress the scores and the times to give

```
"testf1.rec"
SIL
ONE
NINE
FOUR
SIL
.
```

In order to use HVite effectively and efficiently, it is important to set appropriate values for its pruning thresholds and the language model scaling parameters. The main pruning beam is set by the `-t` option. Some experimentation will be necessary to determine appropriate levels but around 250.0 is usually a reasonable starting point. Word-end pruning (`-v`) and the maximum model limit (`-u`) can also be set if required, but these are not mandatory and their effectiveness will depend greatly on the task.

The relative levels of insertion and deletion errors can be controlled by scaling the language model likelihoods using the `-s` option and adding a fixed *penalty* using the `-p` option. For example, setting `-s 10.0 -p -20.0` would mean that every language model log probability x would be converted to $10x - 20$ before being added to the tokens emitted from the corresponding word-end node. As an extreme example, setting `-p 100.0` caused the digit recogniser above to output

```
SIL OH OH ONE OH OH OH NINE FOUR OH OH OH OH SIL
```

where adding 100 to each word-end transition has resulted in a large number of insertion errors. The word inserted is "oh" primarily because it is the shortest in the vocabulary. Another problem which may occur during recognition is the inability to arrive at the final node in the recognition network after processing the whole utterance. The user is made aware of the problem by the message "No tokens survived to final node of network". The inability to match the data against the recognition network is usually caused by poorly trained acoustic models and/or very tight pruning beam-widths. In such cases, partial recognition results can still be obtained by setting the HRec configuration variable `FORCEOUT` true. The results will be based on the most likely partial hypothesis found in the network.

13.4 Evaluating Recognition Results

Once the test data has been processed by the recogniser, the next step is to analyse the results. The tool HResults is provided for this purpose. HResults compares the transcriptions output by HVite with the original reference transcriptions and then outputs various statistics. HResults matches each of the recognised and reference label sequences by performing an optimal string match using dynamic programming. Except when scoring word-spotter output as described later, it does not take any notice of any boundary timing information stored in the files being compared. The optimal string match works by calculating a score for the match with respect to the reference such that identical labels match with score 0, a label insertion carries a score of 7, a deletion carries a

score of 7 and a substitution carries a score of 10[2]. The optimal string match is the label alignment which has the lowest possible score.

Once the optimal alignment has been found, the number of substitution errors (S), deletion errors (D) and insertion errors (I) can be calculated. The percentage correct is then

$$\text{Percent Correct} = \frac{N - D - S}{N} \times 100\% \tag{13.1}$$

where N is the total number of labels in the reference transcriptions. Notice that this measure ignores insertion errors. For many purposes, the percentage accuracy defined as

$$\text{Percent Accuracy} = \frac{N - D - S - I}{N} \times 100\% \tag{13.2}$$

is a more representative figure of recogniser performance.

HResults outputs both of the above measures. As with all HTK tools it can process individual label files and files stored in MLFs. Here the examples will assume that both reference and test transcriptions are stored in MLFs.

As an example of use, suppose that the MLF `results` contains recogniser output transcriptions, `refs` contains the corresponding reference transcriptions and `wlist` contains a list of all labels appearing in these files. Then typing the command

```
HResults -I refs wlist results
```

would generate something like the following

```
====================== HTK Results Analysis =======================
  Date: Sat Sep  2 14:14:22 1995
  Ref : refs
  Rec : results
------------------------ Overall Results --------------------------
SENT: %Correct=98.50 [H=197, S=3, N=200]
WORD: %Corr=99.77, Acc=99.65 [H=853, D=1, S=1, I=1, N=855]
===================================================================
```

The first part shows the date and the names of the files being used. The line labelled `SENT` shows the total number of complete sentences which were recognised correctly. The second line labelled `WORD` gives the recognition statistics for the individual words[3].

It is often useful to visually inspect the recognition errors. Setting the `-t` option causes aligned test and reference transcriptions to be output for all sentences containing errors. For example, a typical output might be

```
Aligned transcription: testf9.lab vs testf9.rec
 LAB: FOUR    SEVEN NINE THREE
 REC: FOUR OH SEVEN FIVE THREE
```

here an "oh" has been inserted by the recogniser and "nine" has been recognised as "five"

If preferred, results output can be formatted in an identical manner to NIST scoring software by setting the `-h` option. For example, the results given above would appear as follows in NIST format

```
,-----------------------------------------------------------.
| HTK Results Analysis at Sat Sep  2 14:42:06 1995          |
| Ref: refs                                                 |
| Rec: results                                              |
|===========================================================|
|           # Snt |  Corr    Sub    Del    Ins    Err  S. Err |
|-----------------------------------------------------------|
| Sum/Avg |  200  |  99.77   0.12   0.12   0.12   0.35   1.50 |
`-----------------------------------------------------------'
```

[2]The default behaviour of HResults is slightly different to the widely used US NIST scoring software which uses weights of 3,3 and 4 and a slightly different alignment algorithm. Identical behaviour to NIST can be obtained by setting the -n option.

[3] All the examples here will assume that each label corresponds to a word but in general the labels could stand for any recognition unit such as phones, syllables, etc. HResults does not care what the labels mean but for human consumption, the labels `SENT` and `WORD` can be changed using the `-a` and `-b` options.

When computing recognition results it is sometimes inappropriate to distinguish certain labels. For example, to assess a digit recogniser used for voice dialing it might be required to treat the alternative vocabulary items "oh" and "zero" as being equivalent. This can be done by making them equivalent using the `-e` option, that is

```
HResults -e ZERO OH  .....
```

If a label is equated to the special label `???`, then it is ignored. Hence, for example, if the recognition output had silence marked by `SIL`, the setting the option `-e ??? SIL` would cause all the `SIL` labels to be ignored.

HRESULTS contains a number of other options. Recognition statistics can be generated for each file individually by setting the `-f` option and a confusion matrix can be generated by setting the `-p` option. When comparing phone recognition results, HRESULTS will strip any triphone contexts by setting the `-s` option. HRESULTS can also process N-best recognition output. Setting the option `-d N` causes HRESULTS to search the first `N` alternatives of each test output file to find the most accurate match with the reference labels.

When analysing the performance of a speaker independent recogniser it is often useful to obtain accuracy figures on a per speaker basis. This can be done using the option `-k mask` where `mask` is a pattern used to extract the speaker identifier from the test label file name. The pattern consists of a string of characters which can include the pattern matching metacharacters `*` and `?` to match zero or more characters and a single character, respectively. The pattern should also contain a string of one or more `%` characters which are used as a mask to identify the speaker identifier.

For example, suppose that the test filenames had the following structure

```
DIGITS_spkr_nnnn.rec
```

where `spkr` is a 4 character speaker id and `nnnn` is a 4 digit utterance id. Then executing HRESULTS by

```
HResults -h -k '*_%%%%_????.*' ....
```

would give output of the form

```
,-------------------------------------------------------------.
| HTK Results Analysis at Sat Sep  2 15:05:37 1995            |
| Ref: refs                                                   |
| Rec: results                                                |
|-------------------------------------------------------------|
|    SPKR | # Snt |  Corr    Sub    Del    Ins    Err  S. Err |
|-------------------------------------------------------------|
|    dgo1 |   20  | 100.00   0.00   0.00   0.00   0.00   0.00 |
|-------------------------------------------------------------|
|    pcw1 |   20  |  97.22   1.39   1.39   0.00   2.78  10.00 |
|-------------------------------------------------------------|
......
|=============================================================|
| Sum/Avg |  200  |  99.77   0.12   0.12   0.12   0.35   1.50 |
`-------------------------------------------------------------'
```

In addition to string matching, HRESULTS can also analyse the results of a recogniser configured for word-spotting. In this case, there is no DP alignment. Instead, each recogniser label w is compared with the reference transcriptions. If the start and end times of w lie either side of the mid-point of an identical label in the reference, then that recogniser label represents a *hit*, otherwise it is a *false-alarm* (FA).

The recogniser output must include the log likelihood scores as well as the word boundary information. These scores are used to compute the *Figure of Merit* (FOM) defined by NIST which is an upper-bound estimate on word spotting accuracy averaged over 1 to 10 false alarms per hour. The FOM is calculated as follows where it is assumed that the total duration of the test speech is T hours. For each word, all of the spots are ranked in score order. The percentage of true hits p_i found before the i'th false alarm is then calculated for $i = 1 \ldots N+1$ where N is the first integer $\geq 10T - 0.5$. The figure of merit is then defined as

$$\text{FOM} = \frac{1}{10T}(p_1 + p_2 + \ldots + p_N + ap_{N+1}) \tag{13.3}$$

where $a = 10T - N$ is a factor that interpolates to 10 false alarms per hour.

Word spotting analysis is enabled by setting the `-w` option and the resulting output has the form

```
------------------- Figures of Merit --------------------
    KeyWord:     #Hits      #FAs  #Actual       FOM
      BADGE:        92        83       102     73.56
     CAMERA:        20         2        22     89.86
     WINDOW:        84         8        92     86.98
      VIDEO:        72         6        72     99.81
    Overall:       268        99       188     87.55
---------------------------------------------------------
```

If required the standard time unit of 1 hour as used in the above definition of FOM can be changed using the `-u option`.

13.5 Generating Forced Alignments

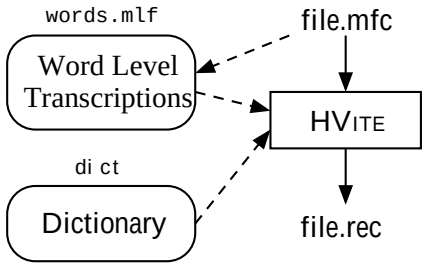


Fig. 13.3 Forced Alignment

HVITE can be made to compute forced alignments by not specifying a network with the `-w` option but by specifying the `-a` option instead. In this mode, HVITE computes a new network for each input utterance using the word level transcriptions and a dictionary. By default, the output transcription will just contain the words and their boundaries. One of the main uses of forced alignment, however, is to determine the actual pronunciations used in the utterances used to train the HMM system in this case, the `-m` option can be used to generate model level output transcriptions. This type of forced alignment is usually part of a *bootstrap* process, initially models are trained on the basis of one fixed pronunciation per word[4]. Then HVITE is used in forced alignment mode to select the best matching pronunciations. The new phone level transcriptions can then be used to retrain the HMMs. Since training data may have leading and trailing silence, it is usually necessary to insert a silence model at the start and end of the recognition network. The `-b` option can be used to do this.

As an illustration, executing

```
HVite -a -b sil -m -o SWT -I words.mlf \
   -H hmmset dict hmmlist file.mfc
```

would result in the following sequence of events (see Fig. 13.3). The input file name `file.mfc` would have its extension replaced by `lab` and then a label file of this name would be searched for. In this case, the MLF file `words.mlf` has been loaded. Assuming that this file contains a word level transcription called `file.lab`, this transcription along with the dictionary `dict` will be used to construct a network equivalent to `file.lab` but with alternative pronunciations included in parallel. Since `-b` option has been set, the specified `sil` model will be inserted at the start and end of the network. The decoder then finds the best matching path through the network and constructs a lattice which includes model alignment information. Finally, the lattice is converted to a transcription and output to the label file `file.rec`. As for testing on a database, alignments

[4] The HLED `EX` command can be used to compute phone level transcriptions when there is only one possible phone transcription per word

will normally be computed on a large number of input files so in practice the input files would be listed in a `.scp` file and the output transcriptions would be written to an MLF using the `-i` option.

When the `-m` option is used, the transcriptions output by HVite would by default contain both the model level and word level transcriptions . For example, a typical fragment of the output might be

```
 7500000  8700000 f  -1081.604736 FOUR 30.000000
 8700000  9800000 ao  -903.821350
 9800000 10400000 r   -665.931641
10400000 10400000 sp    -0.103585
10400000 11700000 s  -1266.470093 SEVEN 22.860001
11700000 12500000 eh  -765.568237
12500000 13000000 v   -476.323334
13000000 14400000 n  -1285.369629
14400000 14400000 sp    -0.103585
```

Here the score alongside each model name is the acoustic score for that segment. The score alongside the word is just the language model score.

Although the above information can be useful for some purposes, for example in bootstrap training, only the model names are required. The formatting option `-o SWT` in the above suppresses all output except the model names.

13.6 Decoding and Adaptation

Speaker adaptation techniques allow speaker independent model sets to be adapted to better fit the characteristics of individual speakers using a small amount of adaptation data. Chapter 9 described how the HEAdapt tool can be used to perform offline supervised adaptation (using the true transcription of the data).

This section describes how adapted model sets are used in the recognition process and also how HVite can be used to perform unsupervised adaptation on a model set (when no transcription is available).

13.6.1 Recognition with Adapted HMMs

As described in section 9.1.3, HEAdapt can produce either a MMF containing the newly adapted model set or a TMF containing just the adaptation transform. If a transformed MMF has been constructed, then HVite can be used in the usual way. If a TMF has been produced however, this needs to be passed to HVite (using the `-J` option) along with the model set from which the transform was estimated. HVite then transforms the model set using the TMF and recognises the input speech using the transformed model set. Thus, a common form of invocation would be

```
HVite -S test.scp -H hmmset -J trans.tmf -i results \
      -w wdnet dict hmmlist
```

13.6.2 Unsupervised Adaptation

Unsupervised adaptation occurs when no transcription of the adaptation data exists and one must be generated. In this case HVite can be used to create a transcription of the adaptation data and use this to estimate a transformation using MLLR. The transformation can then be saved to a TMF using the `-K` option.

Unsupervised adaptation is signalled by the use of the `-j` option and this also controls the mode of adaptation by specifying the number of utterances to be processed before a transform is estimated. Thus, the adaptation can be varied between static (adaptation only performed after recognition of all utterances) and incremental adaptation. As soon as a transform has been estimated during incremental adaptation, it is used to adapt the model set to improve performance for any subsequent utterances. Note however that only the final transformation is saved. To use HVite for this purpose it is invoked with a command line of the form

```
HVite -S adapt.scp -H hmmset -K trans.tmf -j 10 -i results \
      -w wdnet dict hmmlist
```

where `adapt.scp` contains a list of coded adaptation sentences, adaptation is being performed incrementally every 10 utterances and the final transform is stored in `trans.tmf`

13.7 Recognition using Direct Audio Input

In all of the preceding discussion, it has been assumed that input was from speech files stored on disk. These files would normally have been stored in parameterised form so that little or no conversion of the source speech data was required. When HVite is invoked with no files listed on the command line, it assumes that input is to be taken directly from the audio input. In this case, configuration variables must be used to specify firstly how the speech waveform is to be captured and secondly, how the captured waveform is to be converted to parameterised form.

Dealing with waveform capture first, as described in section 5.12, HTK provides two main forms of control over speech capture: signals/keypress and an automatic speech/silence detector. To use the speech/silence detector alone, the configuration file would contain the following

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

where the source sampling rate is being set to 16kHz. Notice that the `SOURCEKIND` must be set to `HAUDIO` and the `SOURCEFORMAT` must be set to `HTK`. Setting the Boolean variable `USESILDET` causes the speech/silence detector to be used, and the `MEASURESIL OUTSILWARN` variables result in a measurement being taken of the background silence level prior to capturing the first utterance. To make sure that each input utterance is being captured properly, the HVite option `-g` can be set to cause the captured wave to be output after each recognition attempt. Note that for a live audio input system, the configuration variable `ENORMALISE` should be explicitly set to `FALSE` both when training models and when performing recognition. Energy normalisation cannot be used with live audio input, and the default setting for this variable is `TRUE`.

As an alternative to using the speech/silence detector, a signal can be used to start and stop recording. For example,

```
# Waveform capture
SOURCERATE=625.0
SOURCEKIND=HAUDIO
SOURCEFORMAT=HTK
AUDIOSIG=2
```

would result in the Unix interrupt signal (usually the Control-C key) being used as a start and stop control[5]. Key-press control of the audio input can be obtained by setting `AUDIOSIG` to a negative number.

Both of the above can be used together, in this case, audio capture is disabled until the specified signal is received. From then on control is in the hands of the speech/silence detector.

The captured waveform must be converted to the required target parameter kind. Thus, the configuration file must define all of the parameters needed to control the conversion of the waveform to the required target kind. This process is described in detail in Chapter 5. As an example, the following parameters would allow conversion to Mel-frequency cepstral coefficients with delta and acceleration parameters.

```
# Waveform to MFCC parameters
TARGETKIND=MFCC_0_D_A
TARGETRATE=100000.0
WINDOWSIZE=250000.0
ZMEANSOURCE=T
```

[5] The underlying signal number must be given, HTK cannot interpret the standard Unix signal names such as `SIGINT`

```
USEHAMMING = T
PREEMCOEF = 0.97
USEPOWER = T
NUMCHANS = 26
CEPLIFTER = 22
NUMCEPS = 12
```

Many of these variable settings are the default settings and could be omitted, they are included explicitly here as a reminder of the main configuration options available.

When HVite is executed in direct audio input mode, it issues a prompt prior to each input and it is normal to enable basic tracing so that the recognition results can be seen. A typical terminal output might be

```
READY[1]>
Please speak sentence - measuring levels
Level measurement completed
DIAL ONE FOUR SEVEN
     ==  [258 frames] -97.8668 [Ac=-25031.3 LM=-218.4] (Act=22.3)

READY[2]>
CALL NINE TWO EIGHT
     ==  [233 frames] -97.0850 [Ac=-22402.5 LM=-218.4] (Act=21.8)

etc
```

If required, a transcription of each spoken input can be output to a label file or an MLF in the usual way by setting the `-e` option. However, to do this a file name must be synthesised. This is done by using a counter prefixed by the value of the HVite configuration variable `RECOUTPREFIX` and suffixed by the value of `RECOUTSUFFIX` . For example, with the settings

```
RECOUTPREFIX = sjy
RECOUTSUFFIX = .rec
```

then the output transcriptions would be stored as `sjy0001.rec`, `sjy0002.rec` etc.

13.8 N-Best Lists and Lattices

As noted in section 13.1, HVite can generate lattices and N-best outputs. To generate an N-best list, the `-n` option is used to specify the number of N-best tokens to store per state and the number of N-best hypotheses to generate. The result is that for each input utterance, a multiple alternative transcription is generated. For example, setting `-n 4 20` with a digit recogniser would generate an output of the form

```
"testf1.rec"
FOUR
SEVEN
NINE
OH
///
FOUR
SEVEN
NINE
OH
OH
///

etc
```

The lattices from which the N-best lists are generated can be output by setting the option `-z ext`. In this case, a lattice called `testf.ext` will be generated for each input test file `testf.xxx`. By default, these lattices will be stored in the same directory as the test files, but they can be redirected to another directory using the `-l` option.

The lattices generated by HVite have the following general form

```
VERSION=1.0
UTTERANCE=testf1.mfc
lmname=wdnet
lmscale=20.00  wdpenalty=-30.00
vocab=dict
N=31   L=56
I=0    t=0.00
I=1    t=0.36
I=2    t=0.75
I=3    t=0.81
... etc
I=30   t=2.48
J=0     S=0    E=1    W=SILENCE   v=0  a=-3239.01  l=0.00
J=1     S=1    E=2    W=FOUR      v=0  a=-3820.77  l=0.00
... etc
J=55    S=29   E=30   W=SILENCE   v=0  a=-246.99   l=-1.20
```

The first 5 lines comprise a header which records names of the files used to generate the lattice along with the settings of the language model scale and penalty factors. Each node in the lattice represents a point in time measured in seconds and each arc represents a word spanning the segment of the input starting at the time of its start node and ending at the time of its end node. For each such span, `v` gives the number of the pronunciation used, `a` gives the acoustic score and `l` gives the language model score.

The language model scores in output lattices do not include the scale factors and penalties. These are removed so that the lattice can be used as a constraint network for subsequent recogniser testing. When using HVite normally, the word level network file is specified using the `-w` option. When the `-w` option is included but no file name is included, HVite constructs the name of a lattice file from the name of the test file and inputs that. Hence, a new recognition network is created for each input file and recognition is very fast. For example, this is an efficient way of experimentally determining optimum values for the language model scale and penalty factors.

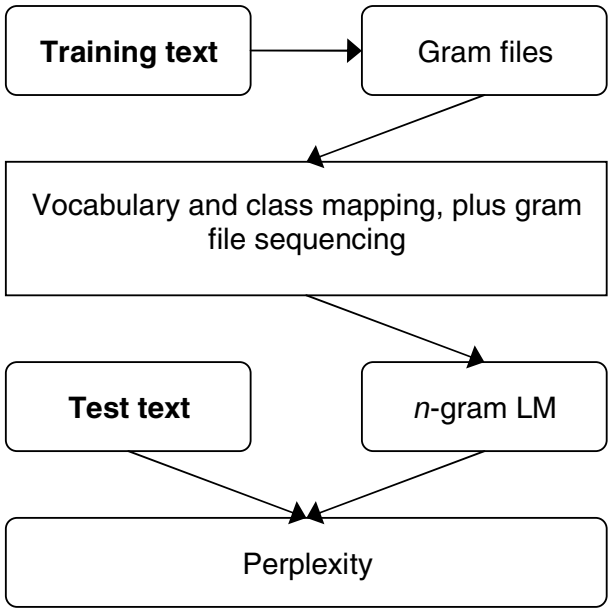
Part III

Language Modelling

Chapter 14

Fundamentals of language modelling

The HTK language modelling tools are designed for constructing and testing statistical n-gram language models. This chapter introduces language modelling and provides an overview of the supplied tools. It is strongly recommended that you read this chapter and then work through the tutorial in the following chapter – this will provide you with everything you need to know to get started building language models.



An n-gram is a sequence of n symbols (e.g. words, syntactic categories, etc) and an n-gram language model (LM) is used to predict each symbol in the sequence given its $n-1$ predecessors. It is built on the assumption that the probability of a specific n-gram occurring in some unknown test text can be estimated from the frequency of its occurrence in some given training text. Thus, as illustrated by the picture above, n-gram construction is a three stage process. Firstly, the training text is scanned and its n-grams are counted and stored in a database of *gram* files. In the second stage some words may be mapped to an out of vocabulary class or other class mapping may be applied, and then in the final stage the counts in the resulting gram files are used to compute n-gram probabalities which are stored in the *language model* file. Lastly, the *goodness* of a language model can be estimated by using it to compute a measure called *perplexity* on a previously unseen test set. In general, the better a language model then the lower its test-set perplexity.

Although the basic principle of an n-gram LM is very simple, in practice there are usually many more potential n-grams than can ever be collected in a training text in sufficient numbers to yield robust frequency estimates. Furthermore, for any real application such as speech recognition, the

use of an essentially static and finite training text makes it difficult to generate a single LM which is well-matched to varying test material. For example, an LM trained on newspaper text would be a good predictor for dictating news reports but the same LM would be a poor predictor for personal letters or a spoken interface to a flight reservation system. A final difficulty is that the *vocabulary* of an n-gram LM is finite and fixed at construction time. Thus, if the LM is word-based, it can only predict words within its vocabulary and furthermore new words cannot be added without rebuilding the LM.

The following four sections provide a thorough introduction to the theory behind n-gram models. It is well worth reading through this section because it will provide you with at least a basic understanding of what many of the tools and their parameters actually do – you can safely skip the equations if you choose because the text explains all the most important parts in plain English. The final section of this chapter then introduces the tools provided to implement the various aspects of n-gram language modelling that have been described.

14.1 n-gram language models

Language models estimate the probability of a word sequence, $\hat{P}(w_1, w_2, \ldots, w_m)$ – that is, they evaluate $P(w_i)$ as defined in equation 1.3 in chapter 1.[1]

The probability $\hat{P}(w_1, w_2, \ldots, w_m)$ can be decomposed as a product of conditional probabilities:

$$\hat{P}(w_1, w_2, \ldots, w_m) = \prod_{i=1}^{m} \hat{P}(w_i \mid w_1, \ldots, w_{i-1}) \tag{14.1}$$

14.1.1 Word n-gram models

Equation 14.1 presents an opportunity for approximating $\hat{P}(\mathcal{W})$ by limiting the context:

$$\hat{P}(w_1, w_2, \ldots, w_m) \simeq \prod_{i=1}^{m} \hat{P}(w_i \mid w_{i-n+1}, \ldots, w_{i-1}) \tag{14.2}$$

for some $n \geqslant 1$. If language is assumed to be ergodic – that is, it has the property that the probability of any state can be estimated from a large enough history independent of the starting conditions[2] – then for sufficiently high n equation 14.2 is exact. Due to reasons of data sparsity, however, values of n in the range of 1 to 4 inclusive are typically used, and there are also practicalities of storage space for these estimates to consider. Models using contiguous but limited context in this way are usually referred to as n-gram language models, and the conditional context component of the probability ("$w_{i-n+1}, \ldots, w_{i-1}$" in equation 14.2) is referred to as the *history*.

Estimates of probabilities in n-gram models are commonly based on maximum likelihood estimates – that is, by counting events in context on some given training text:

$$\hat{P}(w_i | w_{i-n+1}, \ldots, w_{i-1}) = \frac{C(w_{i-n+1}, \ldots, w_i)}{C(w_{i-n+1}, \ldots, w_{i-1})} \tag{14.3}$$

where $C(.)$ is the count of a given word sequence in the training text. Refinements to this maximum likelihood estimate are described later in this chapter.

The choice of n has a significant effect on the number of potential parameters that the model can have, which is maximally bounded by $|\mathbb{W}|^n$, where $\mathbb{W}$ is the set of words in the language model, also known as the *vocabulary*. A 4-gram model with a typically-sized 65,000 word vocabulary can therefore potentially have $65,000^4 \simeq 1.8 \times 10^{19}$ parameters. In practice, however, only a small subset of the possible parameter combinations represent likely word sequences, so the storage requirement is far less than this theoretical maximum – of the order of 10^{11} times less in fact.[3] Even given this significant reduction in coverage and a very large training text[4] there are still many plausible word sequences which will not be encountered in the training text, or will not be found a statistically significant number of times. It would not be sensible to assign all unseen sequences zero

[1] The theory components of this chapter – these first four sections – are condensed from portions of **"Adaptive Statistical Class-based Language Modelling"**, G.L. Moore; *Ph.D thesis, Cambridge University* 2001

[2] See section 5 of [Shannon 1948] for a more formal definition of ergodicity.

[3] Based on the analysis of 170 million words of newspaper and broadcast news text.

[4] A couple of hundred million words, for example.

probability, so methods of coping with low and zero occurrence word tuples have been developed. This is discussed later in section 14.3.

It is not only the storage space that must be considered, however – it is also necessary to be able to attach a reasonable degree of confidence to the derived estimates. Suitably large quantities of example training text are also therefore required to ensure statistical significance. Increasing the amount of training text not only gives greater confidence in model estimates, however, but also demands more storage space and longer analysis periods when estimating model parameters, which may place feasibility limits on how much data can be used in constructing the final model or how thoroughly it can be analysed. At the other end of the scale for restricted domain models there may be only a limited quantity of suitable in-domain text available, so local estimates may need smoothing with global priors. In addition, if language models are to be used for speech recognition then it is good to train them on *precise* acoustic transcriptions where possible – that is, text which features the hesitations, repetitions, word fragments, mistakes and all the other sources of deviation from purely grammatical language that characterise everyday speech. However, such acoustically accurate transcriptions are in limited supply since they must be specifically prepared; real-world transcripts as available for various other purposes almost ubiquitously correct any disfluencies or mistakes made by speakers.

14.1.2 Equivalence classes

The word n-gram model described in equation 14.2 uses an equivalence mapping on the word history which assumes that all contexts which have the same most recent $n-1$ words all have the same probability. This concept can be expressed more generally by defining an equivalence class function that acts on word histories, $\mathcal{E}(.)$, such that if $\mathcal{E}(x) = \mathcal{E}(y)$ then $\forall w : P(w|x) = P(w|y)$:

$$P(w_i \mid w_1, w_2, \ldots, w_{i-1}) = P(w_i \mid \mathcal{E}(w_1, w_2, \ldots, w_{i-1})) \tag{14.4}$$

A definition of $\mathcal{E}$ that describes a word n-gram is thus:

$$\mathcal{E}_{\text{word-}n\text{-gram}}(w_1, \ldots, w_i) = \mathcal{E}(w_{i-n+1}, \ldots, w_i) \tag{14.5}$$

In a good language model the choice of $\mathcal{E}$ should be such that it provides a reliable predictor of the next word, resulting in classes which occur frequently enough in the training text that they can be well modelled, and does not result in so many distinct history equivalence classes that it is infeasible to store or analyse all the resultant separate probabilities.

14.1.3 Class n-gram models

One method of reducing the number of word history equivalence classes to be modelled in the n-gram case is to consider some words as equivalent. This can be implemented by mapping a set of words to a word class $g \in \mathbb{G}$ by using a classification function $G(w) = g$. If any class contains more than one word then this mapping will result in less distinct word classes than there are words, $|\mathbb{G}| < |\mathbb{W}|$, thus reducing the number of separate contexts that must be considered. The equivalence classes can then be described as a sequence of these classes:

$$\mathcal{E}_{\text{class-}n\text{-gram}}(w_1, \ldots, w_i) = \mathcal{E}(G(w_{i-n+1}), \ldots, G(w_i)) \tag{14.6}$$

A deterministic word-to-class mapping like this has some advantages over a word n-gram model since the reduction in the number of distinct histories reduces the storage space and training data requirements whilst improving the robustness of the probability estimates for a given quantity of training data. Because multiple words can be mapped to the same class, the model has the ability to make more confident assumptions about infrequent words in a class based on other more frequent words in the same class[5] than is possible in the word n-gram case – and furthermore for the same reason it is able to make generalising assumptions about words used in contexts which are not explicitly encountered in the training text. These gains, however, clearly correspond with a loss in the ability to distinguish between different histories, although this might be offset by the ability to choose a higher value of n.

The most commonly used form of class n-gram model uses a single classification function, $G(.)$, as in equation 14.6, which is applied to each word in the n-gram, including the word which is being

[5] Since it is assumed that words are placed in the same class because they share certain properties.

predicted. Considering for clarity the bigram[6] case, then given $G(.)$ the language model has the terms w_i, w_{i-1}, $G(w_i)$ and $G(w_{i-1})$ available to it. The probability estimate can be decomposed as follows:

$$\begin{aligned} P_{\text{class}'}(w_i \mid w_{i-1}) &= P(w_i \mid G(w_i), G(w_{i-1}), w_{i-1}) \\ &\times \; P(G(w_i) \mid G(w_{i-1}), w_{i-1}) \end{aligned} \tag{14.7}$$

It is assumed that $P(w_i \mid G(w_i), G(w_{i-1}), w_{i-1})$ is independent of $G(w_{i-1})$ and w_{i-1} and that $P(G(w_i) \mid G(w_{i-1}), w_{i-1})$ is independent of w_{i-1}, resulting in the model:

$$P_{\text{class}}(w_i \mid w_{i-1}) = P(w_i \mid G(w_i)) \;\times\; P(G(w_i) \mid G(w_{i-1})) \tag{14.8}$$

Almost all reported class n-gram work using statistically-found classes is based on clustering algorithms which optimise $G(.)$ on the basis of bigram training set likelihood, even if the class map is to be used with longer-context models. It is interesting to note that this approximation appears to works well, however, suggesting that the class maps found are in some respects "general" and capture some features of natural language which apply irrespective of the context length used when finding these features.

14.2 Statistically-derived Class Maps

An obvious question that arises is how to compute or otherwise obtain a class map for use in a language model. This section discusses one strategy which has successfully been used.

Methods of statistical class map construction seek to maximise the likelihood of the training text given the class model by making iterative controlled changes to an initial class map – in order to make this problem more computationally feasible they typically use a deterministic map.

14.2.1 Word exchange algorithm

[Kneser and Ney 1993][7] describes an algorithm to build a class map by starting from some initial guess at a solution and then iteratively searching for changes to improve the existing class map. This is repeated until some minimum change threshold has been reached or a chosen number of iterations have been performed. The initial guess at a class map is typically chosen by a simple method such as randomly distributing words amongst classes or placing all words in the first class except for the most frequent words which are put into singleton classes. Potential moves are then evaluated and those which increase the likelihood of the training text most are applied to the class map. The algorithm is described in detail below, and is implemented in the HTK tool CLUSTER.

Let $\mathcal{W}$ be the training text list of words $(w_1, w_2, w_3, \ldots)$ and let $\mathbb{W}$ be the set of all words in $\mathcal{W}$. From equation 14.1 it follows that:

$$P_{\text{class}}(\mathcal{W}) \;=\; \prod_{x,y \in \mathbb{W}} P_{\text{class}}(x \mid y)^{C(x,y)} \tag{14.9}$$

where (x, y) is some word pair 'x' preceded by 'y' and $C(x, y)$ is the number of times that the word pair '$y\ x$' occurs in the list $\mathcal{W}$.

In general evaluating equation 14.9 will lead to problematically small values, so logarithms can be used:

$$\log P_{\text{class}}(\mathcal{W}) \;=\; \sum_{x,y \in \mathbb{W}} C(x, y).\log P_{\text{class}}(x \mid y) \tag{14.10}$$

Given the definition of a class n-gram model in equation 14.8, the maximum likelihood bigram probability estimate of a word is:

$$P_{\text{class}}(w_i \mid w_{i-1}) \;=\; \frac{C(w_i)}{C(G(w_i))} \times \frac{C\left(G(w_i), G(w_{i-1})\right)}{C(G(w_{i-1}))} \tag{14.11}$$

[6] By convention *unigram* refers to a 1-gram, *bigram* indicates a 2-gram and *trigram* is a 3-gram. There is no standard term for a 4-gram.

[7] R. Kneser and H. Ney, **"Improved Clustering Techniques for Class-Based Statistical Language Modelling"**; *Proceedings of the European Conference on Speech Communication and Technology* 1993, pp. 973-976

where $C(w)$ is the number of times that the word 'w' occurs in the list $\mathcal{W}$ and $C(G(w))$ is the number of times that the class $G(w)$ occurs in the list resulting from applying $G(.)$ to each entry of $\mathcal{W}$;[8] similarly $C(G(w_x), G(w_y))$ is the count of the class pair '$G(w_y)\ G(w_x)$' in that resultant list.

Substituting equation 14.11 into equation 14.10 and then rearranging gives:

$$\begin{aligned}
\log P_{\text{class}}(\mathcal{W}) &= \sum_{x,y\in\mathbb{W}} C(x,y).\log\left(\frac{C(x)}{C(G(x))}\times\frac{C(G(x),G(y))}{C(G(y))}\right)\\
&= \sum_{x,y\in\mathbb{W}} C(x,y).\log\left(\frac{C(x)}{C(G(x))}\right) + \sum_{x,y\in\mathbb{W}} C(x,y).\log\left(\frac{C(G(x),G(y))}{C(G(y))}\right)\\
&= \sum_{x\in\mathbb{W}} C(x).\log\left(\frac{C(x)}{C(G(x))}\right) + \sum_{g,h\in\mathbb{G}} C(g,h).\log\left(\frac{C(g,h)}{C(h)}\right)\\
&= \sum_{x\in\mathbb{W}} C(x).\log C(x) - \sum_{x\in\mathbb{W}} C(x).\log C(G(x))\\
&\quad + \sum_{g,h\in\mathbb{G}} C(g,h).\log C(g,h) - \sum_{g\in\mathbb{G}} C(g).\log C(g)\\
&= \sum_{x\in\mathbb{W}} C(x).\log C(x) + \sum_{g,h\in\mathbb{G}} C(g,h).\log C(g,h)\\
&\quad - 2\sum_{g\in\mathbb{G}} C(g).\log C(g) \qquad (14.12)
\end{aligned}$$

where (g,h) is some class sequence '$h\ g$'.

Note that the first of these three terms in the final stage of equation 14.12, "$\sum_{x\in\mathbb{W}} C(x)\ .\ \log(C(x))$", is independent of the class map function $G(.)$, therefore it is not necessary to consider it when optimising $G(.)$. The value a class map must seek to maximise, F_{M_C}, can now be defined:

$$F_{\text{M}_\text{C}} = \sum_{g,h\in\mathbb{G}} C(g,h).\log C(g,h) - 2\sum_{g\in\mathbb{G}} C(g).\log C(g) \qquad (14.13)$$

A fixed number of classes must be decided before running the algorithm, which can now be formally defined:

1. **Initialise**: $\forall w \in \mathbb{W} :\ G(w) = 1$
   Set up the class map so that all words are in the first class and all other classes are empty (*or* initialise using some other scheme)
2. **Iterate**: $\forall i \in \{1 \ldots n\}\ \wedge\ \neg s$
   For a given number of iterations $1 \ldots n$ or until some stop criterion s is fulfilled
   (a) **Iterate**: $\forall w \in \mathbb{W}$
   For each word w in the vocabulary
      i. **Iterate**: $\forall c \in \mathbb{G}$
      For each class c
         A. **Move** word w to class c, remembering its previous class
         B. **Calculate** the change in F_{M_C} for this move
         C. **Move** word w back to its previous class
      ii. **Move** word w to the class which increased F_{M_C} by the most, or do not move it if no move increased F_{M_C}

The initialisation scheme given here in step 1 represents a word unigram language model, making no assumptions about which words should belong in which class.[9] The algorithm is greedy and so

[8] That is, $C(G(w)) = \sum_{x:G(x)=G(w)} C(x)$.

[9] Given this initialisation, the first $(|\mathbb{G}| - 1)$ moves will be to place each word into an empty class, however, since the class map which maximises F_{M_C} is the one which places each word into a singleton class.

can get stuck in a local maximum and is therefore not guaranteed to find the optimal class map for the training text. The algorithm is rarely run until total convergence, however, and it is found in practice that an extra iteration can compensate for even a deliberately poor choice of initialisation.

The above algorithm requires the number of classes to be fixed before running. It should be noted that as the number of classes utilised increases so the overall likelihood of the training text will tend tend towards that of the word model.[10] This is why the algorithm does not itself modify the number of classes, otherwise it would naïvely converge on $|\mathbb{W}|$ classes.

14.3 Robust model estimation

Given a suitably large amount of training data, an extremely long n-gram could be trained to give a very good model of language, as per equation 14.1 – in practice, however, any actual extant model must be an approximation. Because it is an approximation, it will be detrimental to include within the model information which in fact was just noise introduced by the limits of the bounded sample set used to train the model – this information may not accurately represent text not contained within the training corpus. In the same way, word sequences which were not observed in the training text cannot be assumed to represent impossible sequences, so some probability mass must be reserved for these. The issue of how to redistribute the probability mass, as assigned by the maximum likelihood estimates derived from the raw statistics of a specific corpus, into a sensible estimate of the real world is addressed by various standard methods, all of which aim to create more robust language models.

14.3.1 Estimating probabilities

Language models seek to estimate the probability of each possible word sequence event occurring. In order to calculate maximum likelihood estimates this set of events must be finite so that the language model can ensure that the sum of the probabilities of all events is 1 given some context. In an n-gram model the combination of the finite vocabulary and fixed length history limits the number of unique events to $|\mathbb{W}|^n$. For any $n > 1$ it is highly unlikely that all word sequence events will be encountered in the training corpora, and many that do occur may only appear one or two times. A language model should not give any unseen event zero probability,[11] but without an infinite quantity of training text it is almost certain that there will be events it does not encounter during training, so various mechanisms have been developed to redistribute probability within the model such that these unseen events are given some non-zero probability.

As in equation 14.3, the maximum likelihood estimate of the probability of an event $\mathcal{A}$ occurring is defined by the number of times that event is observed, a, and the total number of samples in the training set of all observations, A, where $P(\mathcal{A}) = \frac{a}{A}$. With this definition, events that do not occur in the training data are assigned zero probability since it will be the case that $a = 0$. [Katz 1987][12] suggests multiplying each observed count by a discount coefficient factor, d_a, which is dependent upon the number of times the event is observed, a, such that $a' = d_a \,.\, a$. Using this discounted occurrence count, the probability of an event that occurs a times now becomes $P_{\text{discount}}(\mathcal{A}) = \frac{a'}{A}$. Different discounting schemes have been proposed that define the discount coefficient, d_a, in specific ways. The same discount coefficient is used for all events that occur the same number of times on the basis of the symmetry requirement that two events that occur with equal frequency, a, must have the same probability, p_a.

Defining c_a as the number of events that occur exactly a times such that $A = \sum_{a \geq 1} a \,.\, c_a$ it follows that the total amount of reserved mass, left over for distribution amongst the unseen events, is $\frac{1}{c_0}\left(1 - \frac{1}{A}\sum_{a \geq 1} d_a \,.\, c_a \,.\, a\right)$.

[10] Which will be higher, given maximum likelihood estimates.

[11] If it did then from equation 14.1 it follows that the probability of any piece of text containing that event would also be zero, and would have infinite perplexity.

[12] S.M. Katz, **"Estimation of Probabilities from Sparse Data for the Language Model Component of a Speech Recogniser"**; *IEEE Transactions on Acoustic, Speech and Signal Processing* 1987, vol. 35 no. 3 pp. 400-401

Discounting

In [Good 1953][13] a method of discounting maximum likelihood estimates was proposed whereby the count of an event occuring a times is discounted with

$$d_a = (a+1)\frac{c_{a+1}}{a \,.\, c_a} \tag{14.14}$$

A problem with this scheme, referred to as *Good-Turing* discounting, is that due to the count in the denominator it will fail if there is a case where $c_a = 0$ if there is any count $c_b > 0$ for $b > a$. Inevitably as a increases the count c_a will tend towards zero and for high a there are likely to be many such zero counts. A solution to this problem was proposed in [Katz 1987], which defines a cut-off value k at which counts a for $a > k$ are not discounted[14] – this is justified by considering these more frequently observed counts as reliable and therefore not needing to be discounted. Katz then describes a revised discount equation which preserves the same amount of mass for the unseen events:

$$d_a = \begin{cases} \frac{(a+1)\frac{c_{a+1}}{a \,.\, c_a} - (k+1)\frac{c_{k+1}}{c_1}}{1-(k+1)\frac{c_{k+1}}{c_1}} & : \quad 1 \leq a \leq k \\ 1 & : \quad a > k \end{cases} \tag{14.15}$$

This method is itself unstable, however – for example if $k.c_k > c_1$ then d_a will be negative for $1 \leq a \leq k$.

Absolute discounting

An alternative discounting method is *absolute* discounting,[15] in which a constant value m is substracted from each count. The effect of this is that the events with the lowest counts are discounted relatively more than those with higher counts. The discount coefficient is defined as

$$d_a = \frac{a - m}{a} \tag{14.16}$$

In order to discount the same amount of probability mass as the Good-Turing estimate, m must be set to:

$$m = \frac{c_1}{\sum_{a=1}^{A} a \,.\, c_a} \tag{14.17}$$

14.3.2 Smoothing probabilities

The above discounting schemes present various methods of redistributing probability mass from observed events to unseen events. Additionally, if events are infrequently observed then they can be smoothed with less precise but more frequently observed events.

In [Katz 1987] a *back off* scheme is proposed and used alongside Good-Turing discounting. In this method probabilities are redistributed via the recursive utilisation of lower level conditional distributions. Given the n-gram case, if the n-tuple is not observed frequently enough in the training text then a probability based on the occurrence count of a shorter-context $(n-1)$-tuple is used instead – using the shorter context estimate is referred to as *backing off*. In practice probabilities are typically considered badly-estimated if their corresponding word sequences are not explicitly stored in the language model, either because they did not occur in the training text or they have been discarded using some pruning mechanism.

Katz defines a function $\hat{\beta}(w_{i-n+1}, \ldots w_{i-1})$ which represents the total probability of all the unseen events in a particular context. The probability mass $\hat{\beta}$ is then distributed amongst all the unseen w_i and the language model probability estimate becomes:

$$\hat{P}(w_i \mid w_{i-n+1}, \ldots, w_{i-1}) =$$
$$\begin{cases} \alpha(w_{i-n+1}, \ldots, w_{i-1}) \,.\, \hat{P}(w_i | w_{i-n+2}, \ldots, w_{i-1}) & : \quad c(w_{i-n+1}, \ldots, w_i) = 0 \\ d_{c(w_{i-n+1}, \ldots, w_i)} \cdot \frac{c(w_{i-n+1}, \ldots, w_i)}{c(w_{i-n+1}, \ldots, w_{i-1})} & : \quad 1 \leq c(w_{i-n+1}, \ldots, w_i) \leq k \\ \frac{c(w_{i-n+1}, \ldots, w_i)}{c(w_{i-n+1}, \ldots, w_{i-1})} & : \quad c(w_{i-n+1}, \ldots, w_i) > k \end{cases} \tag{14.18}$$

[13] I.J. Good, **"The Population Frequencies of Species and the Estimation of Population Parameters"**; *Biometrika* 1953, vol. 40 (3,4) pp. 237-264

[14] It is suggested that "$k = 5$ or so is a good choice"

[15] H. Ney, U. Essen and R. Kneser, **"On Structuring Probabilistic Dependences in Stochastic Language Modelling"**; *Computer Speech and Language* 1994, vol.8 no.1 pp.1-38

where $c(.)$ is the count of an event and:

$$\alpha(w_{i-n+1},\ldots,w_{i-1}) = \frac{\hat{\beta}(w_{i-n+1},\ldots,w_{i-1})}{\sum_{w_i:c(w_{i-n+1},\ldots,w_i)=0}\hat{P}(w_i|w_{i-n+2},\ldots,w_{i-1})} \tag{14.19}$$

A back off scheme such as this can be implemented efficiently because all the back off weights α can be computed once and then stored as part of the language model, and through its recursive nature it is straightforward to incorporate within a language model. Through the use of pruning methods, contexts which occur 'too infrequently' are not stored in the model so in practice the test $c(w_1,\ldots,w_i)=0$ is implemented as referring to whether or not the context is in the model.

Cut-offs

With a back off scheme low count events can be discarded – *cut-off* – from the model and more frequently observed shorter-context estimates can be used instead. An additional advantage of discarding low occurrence events is that the model size can be substantially reduced, since in general as a decreases so the number of events c_a increases – in fact the Good-Turing discounting scheme depends upon this relationship.

14.4 Perplexity

A measure of language model performance based on average probability can be developed within the field of information theory [Shannon 1948][16]. A speaker emitting language can be considered to be a discrete information source which is generating a sequence of words $w_1, w_2, \ldots, w_m$ from a vocabulary set, $\mathbb{W}$. The probability of a symbol w_i is dependent upon the previous symbols $w_1,\ldots,w_{i-1}$. The information source's inherent per-word entropy H represents the amount of non-redundant information provided by each new word on average, defined in bits as:

$$H = -\lim_{m\to\infty}\frac{1}{m}\sum_{w_1,w_2,\ldots,w_m}\left(P(w_1,w_2,\ldots,w_m)\,\log_2 P(w_1,w_2,\ldots,w_m)\right) \tag{14.20}$$

This summation is over all possible sequences of words, but if the source is ergodic then the summation over all possible word sequences can be discarded and the equation becomes equivalent to:

$$H = -\lim_{m\to\infty}\frac{1}{m}\log_2 P(w_1,w_2,\ldots,w_m) \tag{14.21}$$

It is reasonable to assume ergodicity on the basis that we use language successfully without having been privy to all words that have ever been spoken or written, and we can disambiguate words on the basis of only the recent parts of a conversation or piece of text.

Having assumed this ergodic property, it follows that given a large enough value of m, H can be approximated with:

$$\hat{H} = -\frac{1}{m}\log_2 P(w_1,w_2,\ldots,w_m) \tag{14.22}$$

This last estimate is feasible to evaluate, thus providing the basis for a metric suitable for assessing the performance of a language model.

Considering a language model as an information source, it follows that a language model which took advantage of all possible features of language to predict words would also achieve a per-word entropy of H. It therefore makes sense to use a measure related to entropy to assess the actual performance of a language model. Perplexity, PP, is one such measure that is in standard use, defined such that:

$$PP = 2^{\hat{H}} \tag{14.23}$$

Substituting equation 14.22 into equation 14.23 and rearranging obtains:

$$PP = \hat{P}(w_1,w_2,\ldots,w_m)^{-\frac{1}{m}} \tag{14.24}$$

where $\hat{P}(w_1,w_2,\ldots,w_m)$ is the probability estimate assigned to the word sequence $(w_1, w_2, \ldots, w_m)$ by a language model.

[16] C.E. Shannon, **"A Mathematical Theory of Communication"**; *The Bell System Technical Journal* 1948, vol. 27 pp. 379-423, 623-656. Available online at `http://galaxy.ucsd.edu/new/external/shannon.pdf`

Perplexity can be considered to be a measure of on average how many different equally most probable words can follow any given word. Lower perplexities represent better language models, although this simply means that they 'model language better', rather than necessarily work better in speech recognition systems – perplexity is only loosely correlated with performance in a speech recognition system since it has no ability to note the relevance of acoustically similar or dissimilar words.

In order to calculate perplexity both a language model and some test text are required, so a meaningful comparison between two language models on the basis of perplexity requires the same test text and word vocabulary set to have been used in both cases. The size of the vocabulary can easily be seen to be relevant because as its cardinality is reduced so the number of possible words given any history must monotonically decrease, therefore the probability estimates must on average increase and so the perplexity will decrease.

14.5 Overview of n-Gram Construction Process

This section describes the overall process of building an n-gram language model using the HTK tools. As noted in the introduction, it is a three stage process. Firstly, the training text is scanned and the n-grams counts are stored in a set of *gram* files. Secondly, and optionally, the counts in the gram files are modified to perform vocabulary and class mapping. Finally the resulting gram files are used to build the LM. This separation into stages adds some complexity to the overall process but it makes it much more efficient to handle very large quantities of data since the gram files only need to be constructed once but they can be augmented, processed and used for constructing LMs many times.

The overall process involved in building an n-gram language model using the HTK tools is illustrated in Figure 14.1. The procedure begins with some training text, which first of all should be conditioned into a suitable format by performing operations such as converting numbers to a citation form, expanding common abbreviations and so on. The precise format of the training text depends on your requirements, however, and can vary enormously – therefore conditioning tools are not supplied with HTK.[17]

Given some input text, the tool LGPrep scans the input word sequence and counts n-grams.[18] These n-gram counts are stored in a buffer which fills as each new n-gram is encountered. When this buffer becomes full, the n-grams within it are sorted and stored in a *gram* file. All words (and symbols generally) are represented within HTK by a unique integer id. The mapping from words to ids is recorded in a word map. On startup, LGPrep loads in an existing word map, then each new word encountered in the input text is allocated a new id and added to the map. On completion, LGPrep outputs the new updated word map. If more text is input, this process is repeated and hence the word map will expand as more and more data is processed.

Although each of the gram files output by LGPrep is sorted, the range of n-grams within individual files will overlap. To build a language model, all n-gram counts must be input in sort order so that words with equivalent histories can be grouped. To accommodate this, all HTK language modelling tools can read multiple gram files and sort them on-the-fly. This can be inefficient, however, and it is therefore useful to first copy a newly generated set of gram files using the HLM tool LGCopy. This yields a set of gram files which are *sequenced*, i.e. the ranges of n-grams within each gram file do not overlap and can therefore be read in a single stream. Furthermore, the sequenced files will take less disc space since the counts for identical n-gram in different files will have been merged.

[17] In fact a very simple text conditioning Perl script is included in `LMTutorial/extras/LCond.pl` for demonstration purposes only – it converts text to uppercase (so that words are considered equivalent irrespective of case) and reads the input punctuation in order to tag sentences, stripping most other punctuation. See the script for more details.

[18] LGPrep can also perform text modification using supplied rules.

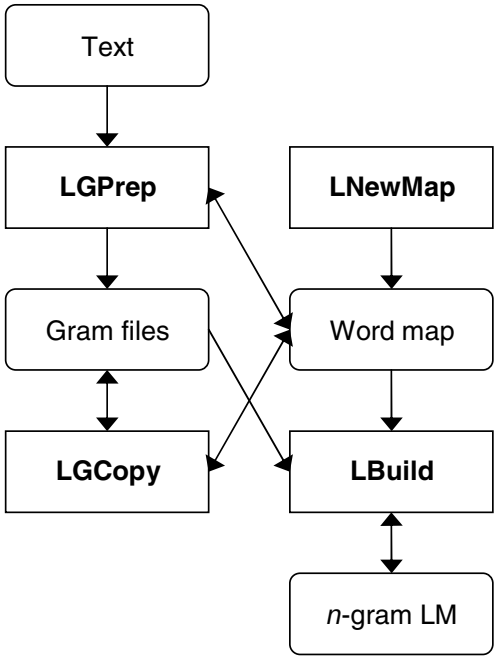


Fig. 14.1 The main stages in building an n-gram language model

The set of (possibly sequenced) gram files and their associated word map provide the raw data for building an n-gram LM. The next stage in the construction process is to define the vocabulary of the LM and convert all n-grams which contain OOV (out of vocabulary) words so that each OOV word is replaced by a single symbol representing the *unknown* class. For example, the n-gram `AN OLEAGINOUS AFFAIR` would be converted to `AN !!UNK AFFAIR` if the word "oleaginous" was not in the selected vocabulary and `!!UNK` is the name chosen for the unknown class.

This assignment of OOV words to a class of unknown words is a specific example of a more general mechanism. In HTK, any word can be associated with a named class by listing it in a *class map* file. Classes can be defined either by listing the class members or by listing all non-members. For defining the unknown class the latter is used, so a plain text list of all in-vocabulary words is supplied and all other words are mapped to the OOV class. The tool LGCopy can use a class map to make a copy of a set of gram files in which all words listed in the class map are replaced by the class name, and also output a word map which contains only the required vocabulary words and their ids plus any classes and their ids.

As shown in Figure 14.1, the LM itself is built using the tool LBuild. This takes as input the gram files and the word map and generates the required LM. The language model can be built in steps (first a unigram, then a bigram, then a trigram, etc.) or in a single pass if required.

14.6 Class-Based Language Models

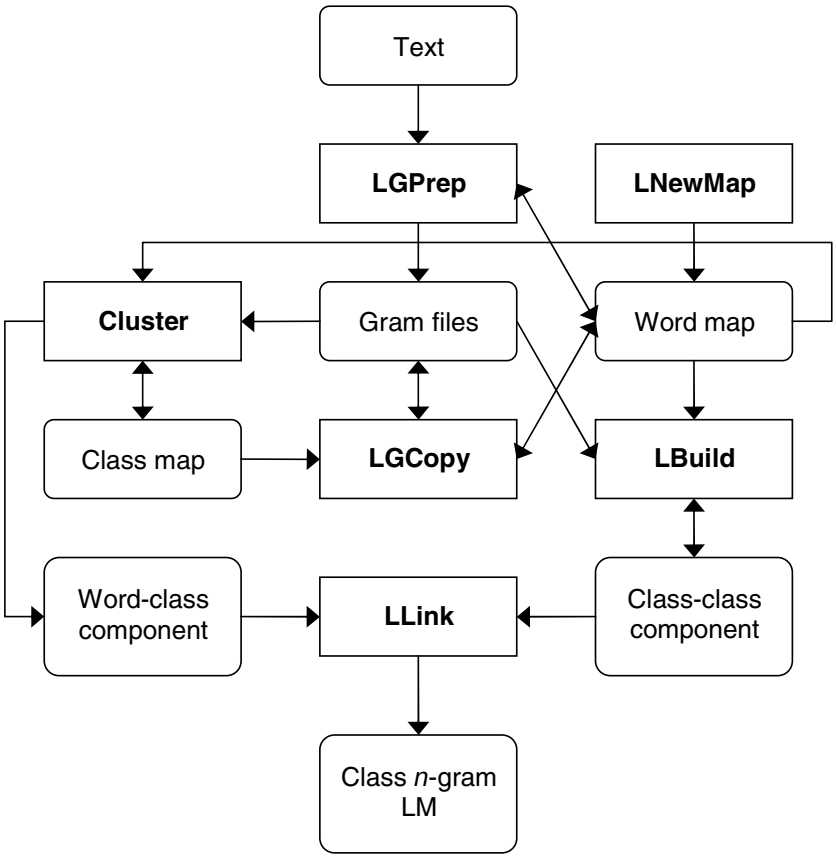


Fig. 14.2 The main stages in building a class-based language model

As described in section 14.1.3, a class-base language model consists of two separate components. The first is an n-gram which models the sequence of classes (i.e. $p(c_i, |c_{i-n+1}, \ldots, c_{n-1})$) and the second is a class map with associated word counts or probabilities within classes allowing the word-given-class probability bigram $p(w_k|c_k)$ to be evaluated. These files may then either be linked into a single composite file or a third 'linking' file is create to point to these two separate files – both of these operations can be performed using the LLINK tool.

Given a set of word classes defined in a class map file and a set of word level gram files, building a class-based model with the HTK tools requires only a few simple modifications to the basic procedure described above for building a word n-gram:

- CLUSTER is used with the word map and word level gram files derived from the source text to construct a class map which defines which class each word is in. The same tool is then used to generate the word-classes component file referred to above. Note that CLUSTER can also be used to generate this file from an existing or manually-generated class map.
- LGCOPY is used with the class map to convert the word level gram files derived from the source text into class gram files. LBUILD can then be used directly with the class level gram files to build the class sequence n-gram language model referred to above.
- LLINK is then run to create either a language model script pointing to the two separate language model files or a single composite file. The resulting language model is then ready for use.

The main steps of this procedure are illustrated in Figure 14.2.

The next chapter provides a more thorough introduction to the tools as well as a tutorial to work through explaining how to use them in practice.

Chapter 15

A Tutorial Example of Building Language Models

This chapter describes the construction and evaluation of language models using the HTK language modelling tools. The models will be built from scratch with the exception of the text conditioning stage necessary to transform the raw text into its most common and useful representation (e.g. number conversions, abbreviation expansion and punctuation filtering). All resources used in this tutorial can be found in the `LMTutorial` directory of the HTK distribution.

The text data used to build and test the language models are the copyright-free texts of 50 Sherlock Holmes stories by Arthur Conan Doyle. The texts have been partitioned into training and test material (49 stories for training and 1 story for testing) and reside in the `train` and `test` subdirectories respectively.

15.1 Database preparation

The first stage of any language model development project is data preparation. As mentioned in the introduction, the text data used in these example has already been conditioned. If you examine each file you will observe that they contains a sequence of tagged sentences. When training a language model you need to include sentence start and end labelling because the tools cannot otherwise infer this. Although there is only one sentence per line in these files, this is not a restriction of the HTK tools and is purely for clarity – you can have the entire input text on a single line if you want. Notice that the default sentence start and sentence end tokens of `<s>` and `</s>` are used – if you were to use different tokens for these you would need to pass suitable configuration parameters to the HTK tools.[1] An extremely simple text conditioning tool is supplied in the form of LCOND.PL in the `LMTutorial/extras` folder – this only segments text into sentences on the basis of punctuation, as well as converting to uppercase and stripping most punctuation symbols, and is not intended for serious use. In particular it does not convert numbers into words and will not expand abbreviations. Exactly what conditioning you perform on your source text is dependent on the task you are building a model for.

Once your text has been conditioned, the next step is to use the tool LGPREP to scan the input text and produce a preliminary set of sorted n-gram files. In this tutorial we will store all n-gram files created by LGPREP will be stored in the `holmes.0` directory, so create this directory now. In a Unix-type system, for example, the standard command is

```
$ mkdir holmes.0
```

The HTK tools maintain a cumulative word map to which every new word is added and assigned a unique id. This means that you can add future n-gram files without having to rebuild existing ones so long as you start from the same word map, thus ensuring that each id remains unique. The side effect of this ability is that LGPREP always expects to be given a word map, so to prepare the first n-gram file (also referred to elsewhere as a 'gram' file) you must pass an empty word map file.

You can prepare an initial, empty word map using the LNEWMAP tool. It needs to be passed the name to be used internally in the word map as well as a file name to write it to; optionally

[1] `STARTWORD` and `ENDWORD` to be precise.

you may also change the default character escaping mode and request additional fields. Type the following:

```
$ LNewMap -f WFC Holmes empty.wmap
```

and you'll see that an initial, empty word map file has been created for you in the file `empty.wmap`. Examine the file and you will see that it contains just a header and no words. It looks like this:

```
Name    = Holmes
SeqNo   = 0
Entries = 0
EscMode = RAW
Fields  = ID,WFC
\Words\
```

Pay particular attention to the `SeqNo` field since this represents the sequence number of the word map. Each time you add words to the word map the sequence number will increase – the tools will compare the sequence number in the word map with that in any data files they are passed, and if the word map is too old to contain all the necessary words then it will be rejected. The `Name` field must also match, although initially you can set this to whatever you like.[2] The other fields specify that no HTK character escaping will be used, and that we wish to store the (compulsory) word ID field as well as an optional count field, which will reveal how many times each word has been encountered to date. The `ID` field is always present which is why you did not need to pass it with the `-f` option to LNewMap.

To clarify, if we were to use the Sherlock Holmes texts together with other previously generated n-gram databases then the most recent word map available must be used instead of the prototype map file above. This would ensure that the map saved by LGPrep once the new texts have been processed would be suitable for decoding all available n-gram files.

We'll now process the text data with the following command:

```
$ LGPrep -T 1 -a 100000 -b 200000 -d holmes.0 -n 4
         -s "Sherlock Holmes" empty.wmap train/*.txt
```

The `-a` option sets the maximum number of new words that can be encountered in the texts to 100,000 (in fact, this is the default). If, during processing, this limit is exceeded then LGPrep will terminate with an error and the operation will have to be repeated by setting this limit to a larger value.

The `-b` option sets the internal n-gram buffer size to 200,000 n-gram entries. This setting has a direct effect on the overall process size. The memory requirent for the internal buffer can be calculated according to $mem_{bytes} = (n + 1) * 4 * b$ where n is the n-gram size (set with the `-n` option) and b is the buffer size. In the above example, the n-gram size is set to four which will enable us to generate bigram, trigram and four-gram language models. The smaller the buffer then in general the more separate files will be written out – each time the buffer fills a new n-gram file is generated in the output directory, specified by the `-d` option.

The `-T 1` option switches on tracing at the lowest level. In general you should probably aim to run each tool with at least `-T 1` since this will give you better feedback about the progress of the tool. Other useful options to pass are `-D` to check the state of configuration variables – very useful to check you have things set up correctly – and `-A` so that if you save the tool output you will be able to see what options it was run with. It's good practice to always pass `-T 1 -A -D` to every HTK tool in fact. You should also note that all HTK tools require the option switches to be passed *before* the compulsory tool parameters – trying to run `LGPrep train/*.txt -T 1` will result in an error, for example.

Once the operation has completed, the `holmes.0` directory should contain the following files:

```
gram.0  gram.1  gram.2  wmap
```

The saved word map file `wmap` has grown to include all newly encountered words and the identifiers that the tool has assigned them, and at the same time the map sequence count has been incremented by one.

[2] The exception to this is that differing text may follow a `%` character.

```
Name  = Holmes
SeqNo = 1
Entries = 18080
EscMode  = RAW
Fields  = ID,WFC
\Words\
<s>     65536   33669
IT      65537   8106
WAS     65538   7595
...
```

Remember that map sequence count together with the map's name field are used to verify the compatibility between the map and any n-gram files. The contents of the n-gram files can be inspected using the LGList tool: (if not using a Unix type system you may need to omit the `| more` and find some other way of viewing the output in a more manageable format; try `> file.txt` and viewing the resulting file if that works)

```
$ LGList holmes.0/wmap holmes.0/gram.2 | more

4-Gram File holmes.0/gram.2[165674 entries]:
 Text Source: Sherlock Holmes
'           IT          IS          NO          : 1
'CAUSE      I           SAVED       HER         : 1
'EM         </s>        <s>         WHO         : 1
</s>        <s>         '           IT          : 1
</s>        <s>         A           BAND        : 1
</s>        <s>         A           BEAUTIFUL   : 1
</s>        <s>         A           BIG         : 1
</s>        <s>         A           BIT         : 1
</s>        <s>         A           BROKEN      : 1
</s>        <s>         A           BROWN       : 2
</s>        <s>         A           BUZZ        : 1
</s>        <s>         A           CAMP        : 1
...
```

If you examine the other n-gram files you will notice that whilst the contents of each n-gram file are sorted, the files themselves are not sequenced – that is, one file does not carry on where the previous one left off; each is an independent set of n-grams. To derive a sequenced set of n-gram files, where no grams are repeated between files, the tool LGCopy must be used on these existing gram files. For the purposes of this tutorial the new set of files will be stored in the `holmes.1` directory, so create this and then run `LGCopy`:

```
$ mkdir holmes.1
$ LGCopy -T 1 -b 200000 -d holmes.1 holmes.0/wmap holmes.0/gram.*
Input file holmes.0/gram.0 added, weight=1.0000
Input file holmes.0/gram.1 added, weight=1.0000
Input file holmes.0/gram.2 added, weight=1.0000
Copying 3 input files to output files with 200000 entries
 saving 200000 ngrams to file holmes.1/data.0
 saving 200000 ngrams to file holmes.1/data.1
 saving 89516 ngrams to file holmes.1/data.2
489516 out of 489516 ngrams stored in 3 files
```

The resulting n-gram files, together with the word map, can now be used to generate language models for a specific vocabulary list. Note that it is not necessary to sequence the files in this way before building a language model, but if you have too many separate unsequenced n-gram files then you may encounter performance problems or reach the limit of your filing system to maintain open files – in practice, therefore, it is a good idea to always sequence them.

15.2 Mapping OOV words

An important step in building a language model is to decide on the system's vocabulary. For the purpose of this tutorial, we have supplied a word list in the file `5k.wlist` which contains the 5000 most common words found in the text. We'll build our language models and all intermediate files in the `lm_5k` directory, so create it with a suitable command:

```
$ mkdir lm_5k
```

Once the system's vocabulary has been specified, the tool LGCopy should be used to filter out all out-of-vocabulary (OOV) words. To achieve this, the 5K word list is used as a special case of a class map which maps all OOVs into members of the "unknown" word class. The unknown class symbol defaults to `!!UNK`, although this can be changed via the configuration parameter `UNKNOWNNAME`. Run LGCopy again:

```
$ LGCopy -T 1 -o -m lm_5k/5k.wmap -b 200000 -d lm_5k -w 5k.wlist
         holmes.0/wmap holmes.1/data.*
Input file holmes.1/data.0 added, weight=1.0000
Input file holmes.1/data.1 added, weight=1.0000
Input file holmes.1/data.2 added, weight=1.0000
Copying 3 input files to output files with 200000 entries
Class map = 5k.wlist [Class mappings only]
 saving 75400 ngrams to file lm_5k/data.0
92918 out of 489516 ngrams stored in 1 files
```

Because the `-o` option was passed, all n-grams containing OOVs will be extracted from the input files and the OOV words mapped to the unknown symbol with the results stored in the files `lm_5k/data.*`. A new word map containing the new class symbols (`!!UNK` in this case) and only words in the vocabulary will be saved to `lm_5k/5k.wmap`. Note how the newly produced OOV n-gram files can no longer be decoded by the original word map `holmes.0/wmap`:

```
$ LGList holmes.0/wmap lm_5k/data.0 |
  ERROR [+15330]  OpenNGramFile: Gram file map Holmes%%5k.wlist
inconsistent with Holmes
 FATAL ERROR - Terminating program LGList
```

The error is due to the mismatch between the original map's name ("Holmes") and the name of the map stored in the header of the n-gram file we attempted to list ("Holmes%%5k.wlist"). The latter name indicates that the word map was derived from the original map `Holmes` by resolving class membership using the class map `5k.wlist`. As a further consistency indicator, the original map has a sequence count of 1 whilst the class-resolved map has a sequence count of 2.

The correct command for listing the contents of the OOV n-gram file is:

```
$ LGList lm_5k/5k.wmap lm_5k/data.0 | more

4-Gram File lm_5k/data.0[75400 entries]:
 Text Source: LGCopy
!!UNK       !!UNK       !!UNK       !!UNK        : 50
!!UNK       !!UNK       !!UNK       </s>         : 20
!!UNK       !!UNK       !!UNK       A            : 2
!!UNK       !!UNK       !!UNK       ACCOUNTS     : 1
!!UNK       !!UNK       !!UNK       ACROSS       : 1
!!UNK       !!UNK       !!UNK       AND          : 17
...
```

At the same time the class resolved map `lm_5k/5k.wmap` can be used to list the contents of the n-gram, database files – the newer map can view the older grams, but not vice-versa.

```
$ LGList lm_5k/5k.wmap holmes.1/data.2 | more

4-Gram File holmes.1/data.2[89516 entries]:
 Text Source: LGCopy
```

```
THE         SUSSEX      MANOR       HOUSE       : 1
THE         SWARTHY     GIANT       GLARED      : 1
THE         SWEEP       OF          HIS         : 1
THE         SWEET       FACE        OF          : 1
THE         SWEET       PROMISE     OF          : 1
THE         SWINGING    DOOR        OF          : 1
...
```

However, any n-grams containing OOV words will be discarded since these are no longer in the word map.

Note that the required word map `lm_5k/5k.wmap` can also be produced also using the LSUBSET tool:

```
$ LSubset -T 1 holmes.0/wmap 5k.wlist lm_5k/5k.wmap
```

Note also that had the `-o` option not been passed to LGCOPY then the n-gram files built in `lm_5k` would have contained not only those with OOV entries but also all the remaining purely in-vocabulary words, the union of those shown by the two preceding `LGList` commands, in fact. The method that you choose to use depends on what experiments you are performing – the HTK tools allow you a degree of flexibility.

15.3 Language model generation

Language models are built using the LBUILD command. If you're constructing a class-based model you'll also need the CLUSTER tool, but for now we'll construct a standard word n-gram model.

You'll probably want to accept the default of using Turing-Good discounting for your n-gram model, so the first step in generating a language model is to produce a frequency of frequency (FoF) table for the chosen vocabulary list. This is performed automatically by LBUILD, but optionally you can generate this yourself using the LFOF tool and pass the result into LBUILD. This has only a negligable effect on computation time, but the result is interesting in itself because it provides useful information for setting cut-offs. Cut-offs are where you choose to discard low frequency events from the training text – you might wish to do this to decrease model size, or because you judge these infrequent events to be unimportant.

In this example, you can generate a suitable table from the language model databases and the newly generated OOV n-gram files:

```
$ LFoF -T 1 -n 4 -f 32 lm_5k/5k.wmap lm_5k/5k.fof
    holmes.1/data.* lm_5k/data.*
Input file holmes.1/data.0 added, weight=1.0000
Input file holmes.1/data.1 added, weight=1.0000
Input file holmes.1/data.2 added, weight=1.0000
Input file lm_5k/data.0 added, weight=1.0000
Calculating FoF table
```

After executing the command, the FoF table will be stored in `lm_5k/5k.fof`. It shows the number of times a word is found with a given frequency – if you recall the definition of Turing-Good discounting you will see that this needs to be known. See chapter 16 for further details of the FoF file format.

You can also pass a configuration parameter to LFOF to make it output a related table showing the number of n-grams that will be left if different cut-off rates are applied. Rerun LFOF and also pass it the existing configuration file `config`:

```
$ LFoF -C config -T 1 -n 4 -f 32 lm_5k/5k.wmap lm_5k/5k.fof
    holmes.1/data.* lm_5k/data.*
Input file holmes.1/data.0 added, weight=1.0000
Input file holmes.1/data.1 added, weight=1.0000
Input file holmes.1/data.2 added, weight=1.0000
Input file lm_5k/data.0 added, weight=1.0000
Calculating FoF table

cutoff  1-g     2-g     3-g     4-g
```

```
0       5001    128252  330433  471998
1       5001    49014   60314   40602
2       5001    30082   28646   15492
3       5001    21614   17945   8801
...
```

The information can be interpreted as follows. A bigram cut-off value of 1 will leave 49014 bigrams in the model, whilst a trigram cut-off of 3 will result in 17945 trigrams in the model. The configuration file `config` forces the tool to print out this extra information by setting `LPCALC: TRACE=3`. This is the trace level for one of the library modules, and is separate from the trace level for the tool itself (in this case we are passing `-T 1` to set trace level 1. The trace field consists of a series of bits, so setting trace 3 actually turns on two of those trace flags.

We'll now proceed to build our actual language model. In this the model will be generated in stages by executing the LBuild separately for each of the unigram, bigram and trigram sections of the model (we won't build a 4-gram model in this example, although the n-gram files we've build allow us to do so at a later date if we so wish), but you can build the final trigram in one go if you like. The following command will generate the unigram model:

```
$ LBuild -T 1 -n 1 lm_5k/5k.wmap lm_5k/ug
         holmes.1/data.* lm_5k/data.*
```

Look in the `lm_5k` directory and you'll discover the model `ug` which can now be used on its own as a complete ARPA format unigram language model.

We'll now build a bigram model with a cut-off of 1 and to save regenerating the unigram component we'll include our existing unigram model:

```
$ LBuild -C config -T 1 -t lm_5k/5k.fof -c 2 1 -n 2
         -l lm_5k/ug lm_5k/5k.wmap lm_5k/bg1
         holmes.1/data.* lm_5k/data.*
```

Passing the `config` file again means that we get given some discount coefficient information. Try rerunning the tool without the `-C config` to see the difference. We've also passed in the existing `lm_5k/5k.fof` file although this is not necessary – try omitting this and you'll find that the resulting file is identical. What will be different, however, is that the tool will print out the cut-off table seen when running LFoF with the `LPCALC: TRACE = 3` parameter set; if you don't want to see this then don't set `LPCALC: TRACE = 3` in the configuration file (try running the above command without `-t` and `-C`).

Note that this bigram model is created in HTKś own binary version of the ARPA format language model, with just the unigram component in text format by default. This makes the model more compact and faster to load. If you want to override this then simply add the `-f TEXT` parameter to the command.

Finally, the trigram model can be generated using the command:

```
$ LBuild -T 1 -c 3 1 -n 3 -l lm_5k/bg1
         lm_5k/5k.wmap lm_5k/tg1_1
         holmes.1/data.* lm_5k/data.*
```

Alternatively instead of the three stages above, you can also build the final trigram in one step:

```
$ LBuild -T 1 -c 2 1 -c 3 1 -n 3 lm_5k/5k.wmap
         lm_5k/tg2-1_1 holmes.1/data.* lm_5k/data.*
```

If you compare the two trigram models you'll see that they're the same size – there will probably be a few insignificant changes in probability due to more cumulative rounding errors incorporated in the three stage procedure.

15.4 Testing the LM perplexity

Once the language models have been generated, their "goodness" can be evaluated by computing the perplexity of previously unseen text data. This won't necessarily tell you how well the language model will perform in a speech recognition task because it takes no account of acoustic similarities or the vagaries of any particular system, but it will reveal how well a given piece of test text is

modelled by your language model. The directory `test` contains a single story which was withheld from the training text for testing purposes – if it had been included in the training text then it wouldn't be fair to test the perplexity on it since the model would have already 'seen' it.

Perplexity evaluation is carried out using LPlex. The tool accepts input text in one of two forms – either as an HTK style MLF (this is the default mode) or as a simple text stream. The text stream mode, specified with the `-t` option, will be used to evaluate the test material in this example.

```
$ LPlex -n 2 -n 3 -t lm_5k/tg1_1 test/red-headed_league.txt
LPlex test #0: 2-gram
perplexity 131.8723, var 7.8744, utterances 556, words predicted 8588
num tokens 10408, OOV 665, OOV rate 6.75% (excl. </s>)

Access statistics for lm_5k/tg1_1:
Lang model  requested  exact backed    n/a     mean    stdev
    bigram       8588  78.9%  20.6%   0.4%    -4.88     2.81
   trigram          0   0.0%   0.0%   0.0%     0.00     0.00
LPlex test #1: 3-gram
perplexity 113.2480, var 8.9254, utterances 556, words predicted 8127
num tokens 10408, OOV 665, OOV rate 6.75% (excl. </s>)

Access statistics for lm_5k/tg1_1:
Lang model  requested  exact backed    n/a     mean    stdev
    bigram       5357  68.2%  31.1%   0.6%    -5.66     2.93
   trigram       8127  34.1%  30.2%  35.7%    -4.73     2.99
```

The multiple `-n` options instruct LPlex to perform two separate tests on the data. The first test (`-n 2`) will use only the bigram part of the model (and unigram when backing off), whilst the second test (`-n 3`) will use the full trigram model. For each test, the first part of the result gives general information such as the number of utterances and tokens encountered, words predicted and OOV statistics. The second part of the results gives explicit access statistics for the back off model. For the trigram model test, the total number of words predicted is 8127. From this number, 34.1% were found as explicit trigrams in the model, 30.2% were computed by backing off to the respective bigrams and 35.7% were simply computed as bigrams by shortening the word context.

These perplexity tests do not include the prediction of words from context which includes OOVs. To include such n-grams in the calculation the `-u` option should be used.

```
$ LPlex -u -n 3 -t lm_5k/tg1_1 test/red-headed_league.txt
LPlex test #0: 3-gram
perplexity 117.4177, var 8.9075, utterances 556, words predicted 9187
num tokens 10408, OOV 665, OOV rate 6.75% (excl. </s>)

Access statistics for lm_5k/tg1_1:
Lang model  requested  exact backed    n/a     mean    stdev
    bigram       5911  68.5%  30.9%   0.6%    -5.75     2.94
   trigram       9187  35.7%  31.2%  33.2%    -4.77     2.98
```

The number of tokens predicted has now risen to 9187. For analysing OOV rates the tool provides the `-o` option which will print a list of unique OOVs encountered together with their occurrence counts. Further trace output is available with the `-T` option.

15.5 Generating and using count-based models

The language models generated in the previous section are static in terms of their size and vocabulary. For example, in order to evaluate a trigram model with cut-offs 2 (bigram) and 2 (trigram) the user would be required to rebuild the bigram and trigram stages of the model. When large amounts of text data are used this can be a very time consuming operation.

The HLM toolkit provides the capabilities to generate and manipulate a more generic type of model, called a count-based models, which can be dynamically adjusted in terms of its size and vocabulary. Count-based models are produced by specifying the `-x` option to LBuild. The user

may set cut-off parameters which control the initial size of the model, but if so then once the model is generated only higher cut-off values may be specified in the subsequent operations. The following command demonstrates how to generate a count-based model:

```
$ LBuild -C config -T 1 -t lm_5k/5k.fof -c 2 1 -c 3 1
         -x -n 3 lm_5k/5k.wmap lm_5k/tg1_1c
         holmes.1/data.* lm_5k/data.0
```

Note that in the above example the full trigram model is generated by a single invocation of the tool and no intermediate files are kept (i.e. the unigram and bigram models files).

The generated model can now be used in perplexity tests and different model sizes can be obtained by specifying new cut-off values via the `-c` option of LPlex. Thus, using a trigram model with cut-offs (2,2) gives

```
$ LPlex -c 2 2 -c 3 2 -T 1 -u -n 3 -t lm_5k/tg1_1c
        test/red-headed_league.txt
...
LPlex test #0: 3-gram
Processing text stream: test/red-headed_league.txt
perplexity 126.2665, var 9.0519, utterances 556, words predicted 9187
num tokens 10408, OOV 665, OOV rate 6.75% (excl. </s>)
...
```

and a model with cut-offs (3,3) gives

```
$ LPlex -c 2 3 -c 3 3 -T 1 -u -n 3 -t lm_5k/tg1_1c
        test/red-headed_league.txt
...
Processing text stream: test/red-headed_league.txt
perplexity 133.4451, var 9.0880, utterances 556, words predicted 9187
num tokens 10408, OOV 665, OOV rate 6.75% (excl. </s>)
...
```

However, the count model `tg1_1c` cannot be used directly in recognition tools such as HVite or HLvx. An ARPA style model of the required size suitable for recognition can be derived using the HLMCopy tool:

```
$ HLMCopy -T 1 lm_5k/tg1_1c lm_5k/rtg1_1
```

This will be the same as the original trigram model built above, with the exception of some insignificant rounding differences.

15.6 Model interpolation

The HTK language modelling tools also provide the capabilities to produce and evaluate interpolated language models. Interpolated models are generated by combining a number of existing models in a specified ratio to produce a new model using the tool LMerge. Furthermore, LPlex can also compute perplexities using linearly interpolated n-gram probabilities from a number of source models. The use of model interpolation will be demonstrated by combining the previously generated Sherlock Holmes model with an existing 60,000 word business news domain trigram model (`60bn_tg.lm`). The perplexity measure of the unseen Sherlock Holmes text using the business news model is 297 with an OOV rate of 1.5%. (`LPlex -t -u 60kbn_tg.lm test/*`). In the following example, the perplexity of the test date will be calculated by combining the two models in the ratio of 0.6 `60kbn_tg.lm` and 0.4 `tg1_1c`:

```
$ LPlex -T 1 -u -n 3 -t -i 0.6 ./60kbn_tg.lm
        lm_5k/tg1_1c test/red-headed_league.txt
Loading language model from lm_5k/tg1_1c
Loading language model from ./60kbn_tg.lm
Using language model(s):
  3-gram lm_5k/tg1_1c, weight 0.40
  3-gram ./60kbn_tg.lm, weight 0.60
```

```
Found 60275 unique words in 2 model(s)
LPlex test #0: 3-gram
Processing text stream: test/red-headed_league.txt
perplexity 188.0937, var 11.2408, utterances 556, words predicted 9721
num tokens 10408, OOV 131, OOV rate 1.33% (excl. </s>)

Access statistics for lm_5k/tg1_1c:
Lang model  requested  exact backed    n/a     mean    stdev
    bigram       5479  68.0%  31.3%   0.6%    -5.69     2.93
   trigram       8329  34.2%  30.6%  35.1%    -4.75     2.99

Access statistics for ./60kbn_tg.lm:
Lang model  requested  exact backed    n/a     mean    stdev
    bigram       5034  83.0%  17.0%   0.1%    -7.14     3.57
   trigram       9683  48.0%  26.9%  25.1%    -5.69     3.53
```

A single combined model can be generated using LMerge:

```
$ LMerge -T 1 -i 0.6 ./60kbn_tg.lm 5k_unk.wlist
       lm_5k/rtg1_1 5k_merged.lm
```

Note that LMerge cannot merge count-based models, hence the use of `lm_5k/rtg1_1` instead of its count-based equivalent `lm_5k/tg1_1c`. Furthermore, the word list supplied to the tool also includes the OOV symbol (`!!UNK`) in order to preserve OOV n-grams in the output model which in turn allows the use of the `-u` option in LPlex.

Note that the perplexity you will obtain with this combined model is much lower than that when interpolating the two together because the word list has been reduced from the union of the 60K and 5K ones down to a single 5K list. You can build a 5K version of the 60K model using HLMCopy and the `-w` option, but first you need to construct a suitable word list – if you pass it the `5k_unk.wlist` one it will complain about the words in it that weren't found in the language model. In the `extras` subdirectory you'll find a Perl script to rip the word list from the `60kbn_tg.lm` model, `getwordlist.pl`, and the result of running it in `60k.wlist` (the script will work with any ARPA type language model). The intersection of the 60K and 5K word lists is what is required, so if you then run the `extras/intersection.pl` Perl script, amended to use suitable filenames, you'll get the result in `60k-5k-int.wlist`. Then HLMCopy can be used to produce a 5K vocabulary version of the 60K model:

```
$ HLMCopy -T 1 -w 60k-5k-int.wlist 60kbn_tg.lm 5kbn_tg.lm
```

This can then be linearly interpolated with the previous 5K model to compare the perplexity result with that obtained from the LMerge-generated model. If you try this you will find that the perplexities are similar, but not exactly the same (a perplexity of 112 with the merged model and 114 with the two models linearly interpolated, in fact) – this is because using LMerge to combine two models and then using the result is not precisely the same as linearly interpolating two separate models; it is similar, however.

It is also possible to add to an existing language model using the LAdapt tool, which will construct a new model using supplied text and then merge it with the existing one in exactly the same way as LMerge. Effectively this tool allows you to short-cut the process by performing many operations with a single command – see the documentation in section 17.23 for full details.

15.7 Class-based models

A class-based n-gram model is similar to a word-based n-gram in that both store probabilities n-tuples of tokens – except in the class model case these tokens consist of word *classes* instead of words (although word models typically include at least one class for the unknown word). Thus building a class model involves constructing class n-grams. A second component of the model calculates the probability of a word given each class. The HTK tools only support deterministic class maps, so each word can only be in one class. Class language models use a separate file to store each of the two components – the word-given-class probabilities and the class n-grams – as well as a third file which points to the two component files. Alternatively, the two components can be combined

together into a standalone separate file. In this section we'll see how to build these files using the supplied tools.

Before a class model can be built it is necessary to construct a class map which defines which words are in each class. The supplied CLUSTER tool can derive a class map based on the bigram word statistics found in some text, although if you are constructing a large number of classes it can be rather slow (execution time measured in hours, typically). In many systems class models are combined with word models to give further gains, so we'll build a class model based on the Holmes training text and then interpolate it with our existing word model to see if we can get a better overall model.

Constructing a class map requires a decision to be made as to how many separate classes are required. A sensible number depends on what you are building the model for, and whether you intend it purely to interpolate with a word model. In the latter case, for example, a sensible number of classes is often around the 1000 mark when using a 64K word vocabulary. We only have 5000 words in our vocabulary so we'll choose to construct 200 classes in this case.

Create a directory called `holmes.2` and run CLUSTER with

```
$ Cluster -T 1 -c 150 -i 1 -k -o holmes.2/class lm_5k/5k.wmap
         holmes.1/data.0 lm_5k/data.0
Preparing input gram set
Input gram file holmes.1/data.0 added (weight=1.000000)
Input gram file lm_5k/data.0 added (weight=1.000000)
Beginning iteration 1
Iteration complete
Cluster completed successfully
```

The word map and gram files are passed as before – any OOV mapping should be made before building the class map. Passing the `-k` option told CLUSTER to keep the unknown word token `!!UNK` in its own singleton class, whilst the `-c 200` options specifies that we wish to create 150 classes. The `-i 1` performs only one iteration of the clusterer – performing further iterations is likely to give further small improvements in performance, but we won't wait for this here. Whilst CLUSTER is running you can look at the end of the `holmes.2/class.1.log` to see how far it has got. On a Unix-like system you could use a command like `tail holmes.2/class.1.log`, or if you wanted to monitor progress then `tail -f holmes.2/class.1.log` would do the trick. The `1` refers to the iteration, whilst the results are written to this filename because of the `-o holmes.2/class` option which sets the prefix for all output files.

In the `holmes.2` directory you will also see the files `class.recovery` and `class.recovery.cm` – these are a recovery status file and its associated class map which are exported at regular intervals because the CLUSTER tool can take so long to run. In this way you can kill the tool before it has finished and resume execution at a later date by using the `-x` option; in this case you would use `-x holmes.2/class.recovery` for example (making sure you pass the same word map and gram files – the tool does *not* currently check that you pass it the same files when restarting).

Once the tool finishes running you should see the file `holmes.2/class.1.cm` which is the resulting class map. It is in plain text format so feel free to examine it. Note, for example, how `CLASS23` consists almost totally of verb forms ending in `-ED`, whilst `CLASS41` lists various general words for a person or object. Had you created more classes then you would be likely to see more distinctive classes. We can now use this file to build the class n-gram component of our language model.

```
$ LGCopy -T 1 -d holmes.2 -m holmes.2/cmap -w holmes.2/class.1.cm
         lm_5k/5k.wmap holmes.1/data.0 lm_5k/data.0
Input file holmes.1/data.0 added, weight=1.0000
Input file lm_5k/data.0 added, weight=1.0000
Copying 2 input files to output files with 2000000 entries
Class map = holmes.2/class.1.cm
 saving 162397 ngrams to file holmes.2/data.0
330433 out of 330433 ngrams stored in 1 files
```

The `-w` option specifies an input class map which is applied when copying the gram files, so we now have a class gram file in `holmes.2/data.0`. It has an associated word map file `holmes.2/cmap` – although this only contains class names it is technically a word map since it is taken as input wherever a word map is required by the HTK language modelling tools; recall that word maps can contain classes as witnessed by `!!UNK` previously.

You can examine the class n-grams in a similar way to previously by using LGLIST

```
$ LGList holmes.2/cmap holmes.2/data.0 | more

3-Gram File holmes.2/data.0[162397 entries]:
 Text Source: LGCopy
CLASS1      CLASS10     CLASS103     : 1
CLASS1      CLASS10     CLASS11      : 2
CLASS1      CLASS10     CLASS118     : 1
CLASS1      CLASS10     CLASS12      : 1
CLASS1      CLASS10     CLASS126     : 2
CLASS1      CLASS10     CLASS140     : 2
CLASS1      CLASS10     CLASS147     : 1
...
```

And similarly the class n-gram component of the overall language model is built using LBUILD as previously with

```
$ LBuild -T 1 -c 2 1 -c 3 1 -n 3 holmes.2/cmap
     lm_5k/cl150-tg_1_1.cc holmes.2/data.*
Input file holmes.2/data.0 added, weight=1.0000
```

To build the word-given-class component of the model we must run CLUSTER again.

```
$ Cluster -l holmes.2/class.1.cm -i 0 -q lm_5k/cl150-counts.wc
     lm_5k/5k.wmap holmes.1/data.0 lm_5k/data.0
```

This is very similar to how we ran CLUSTER earlier, except that we now want to perform 0 iterations (`-i 0`) and we start by loading in the existing class map with `-l holmes.2/class.1.cm`. We don't need to pass `-k` because we aren't doing any further clustering and we don't need to specify the number of classes since this is read from the class map along with the class contents. The `-q lm_5k/cl150-counts.wc` option tells the tool to write word-given-class counts to the specified file. Alternatively we could have specified `-p` instead of `-q` and written probabilities as opposed to counts. The file is in a plain text format, and either the `-p` or `-q` version is sufficient for forming the word-given-class component of a class language model. Note that in fact we could have simply added either `-p` or `-q` the first time we ran CLUSTER and generated both the class map and language model component file in one go.

Given the two language model components we can now link them together to make our overall class n-gram language model.

```
$ LLink lm_5k/cl150-counts.wc lm_5k/cl150-tg_1_1.cc
     lm_5k/cl150-tg_1_1
```

The LLINK tool creates a simple text file pointing to the two necessary components, auto-detecting whether a count or probabilities file has been supplied. The resulting file, `lm_5k/cl150-tg_1_1` is the finished overall class n-gram model, which we can now assess the performance of with LPLEX.

```
$ LPlex -n 3 -t lm_5k/cl150-tg_1_1 test/red-headed_league.txt
LPlex test #0: 3-gram
perplexity 129.9225, var 7.5378, utterances 556, words predicted 9187
num tokens 10408, OOV 665, OOV rate 6.75% (excl. </s>)

Access statistics for lm_5k/cl150-tg_1_1:
Lang model  requested  exact backed    n/a     mean    stdev
    bigram       3104  95.5%   4.5%   0.0%    -4.67     1.62
   trigram       9187  66.2%  23.9%   9.9%    -4.87     2.75
```

The class trigram model performs worse than the word trigram (which had a perplexity of 117.4), but this is not a surprise since this is true of almost every reasonably-sized test set – the class model is less specific. Interpolating the two often leads to further improvements, however. We can find out if this will happen in this case by interpolating the models with LPLEX.

```
$ LPlex -u -n 3 -t -i 0.4 lm_5k/cl150-tg_1_1 lm_5k/tg1_1
      test/red-headed_league.txt
LPlex test #0: 3-gram
perplexity 102.6389, var 7.3924, utterances 556, words predicted 9187
num tokens 10408, OOV 665, OOV rate 6.75% (excl. </s>)

Access statistics for lm_5k/tg2-1_1:
Lang model  requested  exact backed    n/a     mean    stdev
    bigram       5911  68.5%  30.9%   0.6%    -5.75     2.94
   trigram       9187  35.7%  31.2%  33.2%    -4.77     2.98

Access statistics for lm_5k/cl150-tg_1_1:
Lang model  requested  exact backed    n/a     mean    stdev
    bigram       3104  95.5%   4.5%   0.0%    -4.67     1.62
   trigram       9187  66.2%  23.9%   9.9%    -4.87     2.75
```

So a further gain is obtained – the interpolated model performs significantly better. Further improvement might be possible by attempting to optimise the interpolation weight.

Note that we could also have used LLink to build a single class language model file instead of producing a third file which points to the two components. We can do this by using the `-s` single file option.

```
$ LLink -s lm_5k/cl150-counts.wc lm_5k/cl150-tg_1_1.cc
     lm_5k/cl150-tg_1_1.all
```

The file `lm_5k/cl150-tg_1_1.all` is now a standalone language model, identical in performance to `lm_5k/cl150-tg_1_1` created earlier.

15.8 Problem solving

Sometimes a tool returns an error message which doesn't seem to make sense when you check the files you've passed and the switches you've given. This section provides a few problem-solving hints.

15.8.1 File format problems

If a file which seems to be in the correct format is giving errors such as 'Bad header' then make sure that you are using the correct input filter. If the file is gzipped then ensure you are using a suitable configuration parameter to decompress it on input; similarly if it isn't compressed then check you're not trying to decompress it. Also check to see if you have two files, one with and one without a `.gz` extension – maybe you're picking up the wrong one and checking the other file.

You might be missing a switch or configuration file to tell the tool which format the file is in. In general none of the HTK language modelling tools can auto-detect file formats – unless you tell them otherwise they will expect the file type they are configured to default to and will give an error relevant to that type if it does not match. For example, if you omit to pass `-t` to LPlex then it will treat an input text file as a HTK label file and you will get a 'Too many columns' error if a line has more than 100 words on it or a ridiculously high perplexity otherwise. Check the command documentation in chapter 17.

15.8.2 Command syntax problems

If a tool is giving unexpected syntax errors then check that you have placed all the option switches *before* the compulsory parameters – the tools will not work if this rule is not followed. You must also place whitespace between switches and any options they expect. The ordering of switches is not important, but the order of compulsory parameters cannot be changed. Check the switch syntax – passing a redundant parameter to one will cause problems since it will be interpreted as the first compulsory parameter.

All HTK tools assume that a parameter which starts with a digit is a number of some kind – you cannot pass filenames which start with a digit, therefore. This is a limitation of the routines in HShell.

15.8.3 Word maps

If your word map and gram file combination is being rejected then make sure they match in terms of their sequence number. Although gram files are mainly stored in a binary format the header is in plain text and so if you look at the top of the file you can compare it manually with the word map. Note it is not a good idea to fiddle the values to match since they are bound to be different for a good reason! Word maps must have the same or a higher sequence id than a gram file in order to open that gram file – the names must match too.

The tools might not behave as you expect. For example, LGPrep will write its word map to the file `wmap` unless you tell it otherwise, irrespective of the input filename. It will also place it in the same directory as the gram files unless you changed its name from `wmap`(!) – check you are picking up the correct word map when building subsequent gram files.

The word ids start at 65536 in order to allow space for that many classes below them – anything lower is assumed to be a class. In turn the number of classes is limited to 65535.

15.8.4 Memory problems

Should you encounter memory problems then try altering the amount of space reserved by the tools using the relevant tool switches such as `-a` and `-b` for `LGPrep` and `LGCopy`. You could also try turning on memory tracing to see how much memory is used and for what (use the configuration `TRACE` parameters and the `-T` option as appropriate. Language models can become very large, however – hundreds of megabytes in size, for example – so it is important to apply cut-offs and/or discounting as appropriate to keep them to a suitable size for your system.

15.8.5 Unexpected perplexities

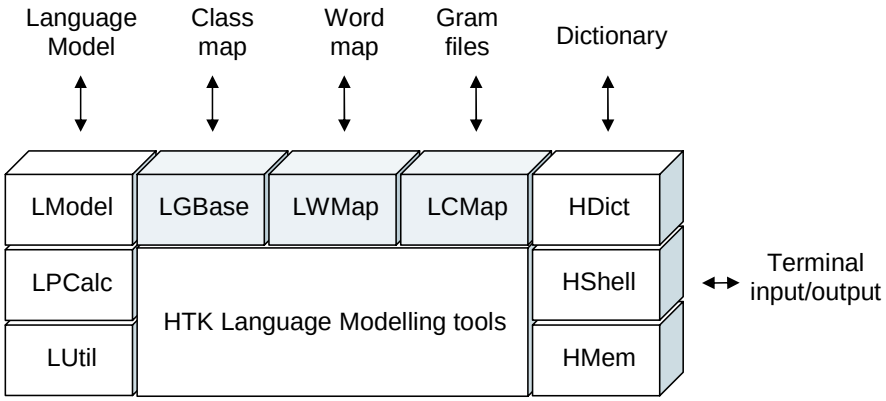
If perplexities are not what you expected, then there are many things that could have gone wrong – you may not have constructed a suitable model – but also some mistakes you might have made. Check that you passed all the switches you intended, and check that you have been consistent in your use of `*RAW*` configuration parameters – using escaped characters in the language model without them in your test text will lead to unexpected results. If you have not escaped words in your word map then check they're not escaped in any class map. When using a class model make sure you're passing the correct input file of the three separate components.

Check the switches to `LPlex` – did you set `-u` as you intended? If you passed a text file did you pass `-t`? Not doing so will lead either to a format error or to extremely bizarre perplexities!

Did you build the length of n-gram you meant to? Check the final language model by looking at the header of it, which is always stored in plain text format. You can easily see how many n-grams there are for each size of n.

Chapter 16

Language Modelling Reference



As noted in the introduction, building a language model with the HTK tools is a two stage process. In the first stage, the n-gram data is accumulated and in the second stage, language models are estimated from this data. The n-gram data consists of a set of gram files and their associated word map. Each gram file contains a list of n-grams and their counts. Each n-gram is represented by a sequence of n integer word indices and the word map relates each of these indices to the actual orthographic word form. As a special case, a word map containing just words and no indices acts as a simple *word list*.

In many cases, a class map is also required. Class maps give a name to a subset of words and associate an index with that name. Once defined, a class name can be used like a regular word and, in particular, it can be listed in a word map. In their most general form, class maps are used to build class-based language models. However, they are also needed for most word-based language models because they allow a class to be defined for *unknown* words.

This chapter includes descriptions of three basic types of data file: gram file, word map file, and class map file. As shown in the figure, each file type is supported by a specific HTK module which provides the required input and output and the other related operations.

Also described in this chapter is a fourth type of data file called a *frequency-of-frequency* or *FoF* file. A FoF file contains a count of the number of times an n-gram occurred just once, twice, three times, etc. It is used with Turing-Good discounting (although LBuild can generate FoF counts on the fly) but it can also be used to estimate the number of n-grams which will be included in the final language models.

The various language model formats, for both word and class-based models, are also described in detail.

Trace levels for each language modelling library modules are also described – see the tool documentation for details of tool level tracing options. Finally, run-time and compile-time settings are documented.

16.1 Words and Classes

In the HTK language modelling tools, words and classes are represented internally by integer indices in the range 0 to $2^{24} - 1$ (16777215). This range is chosen to fit exactly into 3 8-bit bytes thereby allowing efficient compaction of large lists of n-gram counts within gram files.

These integer indices will be referred to subsequently as *ids*. Class ids are limited to the range 0 to $2^{16} - 1$ and word ids fill the remaining range of 2^{16} to $2^{24} - 1$. Thus, any id with a zero most significant byte is a *class id* and all other ids are *word ids*.

In the context of word maps, the term *word* may refer to either an orthographic word or the name of a class. Thus, in its most general form, a word map can contain the ids of both orthographic words from a source text and class names defined in one or more class maps.

The mapping of orthographic words to ids is relatively permanent and normally takes place when building gram files (using LGPrep). Each time a new word is encountered, it is allocated a unique id. Once allocated, a word id should never be changed. Class ids, on the other hand, are more dynamic since their definition depends on the language model being built. Finally, composite word maps can be derived from a collection of word and class maps using the tool LSubset. These derived word maps are typically used to define a working subset of the name space and this subset can contain both word and class ids.

16.2 Data File Headers

All the data files have headers containing information about the file and the associated environment. The header is variable-size being terminated by a data symbol (e.g. `\Words\ \Grams\ \FoFs\`, etc) followed by the start of the actual data.

Each header field is written on a separate line in the form

```
<Field> = <value>
```

where `<Field>` is the name of the field and `<value>` is its value. The field name is case insensitive and zero or more spaces can surround the `=` sign. The `<value>` starts with the first printing character and ends at the last printing character on the line. HTK style escaping is never used in HLM headers.

Fields may be given in any order. Field names which are unrecognised by HTK are ignored. Further field names may be introduced in future, but these are guaranteed not to start with the letter "U".

(NB. The above format rules do not apply to the files described in section 16.8 – see that section for more details)

16.3 Word Map Files

A word map file is a text file consisting of a header and a list of word entries. The header contains the following

1. a name consisting of any printable character string (`Name=sss`).
2. the number of word entries (`Entries=nnn`)
3. a sequence number (`SeqNo=nnn`)
4. whether or not word ids `IDs` and word frequency counts `WFCs` are included (`Fields=ID` or `Fields=ID,WFC`). When the `Fields` field is missing, the word map contains only word names and it degenerates to the special case of a word list.
5. escaping mode (`EscMode=HTK` or `EscMode=RAW`). The default is `HTK`.
6. the language (`Language=xxx`)
7. word map source, a text string used with derived word maps to describe the source from which the subset was derived. (`Source=...`).

The first two of these fields must always be included, and for word maps, the `Fields` field must also be included. The remaining fields are optional. More header fields may be defined later and the user is free to insert others.

The word entries begin with the keyword `\Words\`. Each word is on a separate line with the format

```
word [id [count]]
```

where the id and count are optional. Proper word maps always have an `id`. When the `count` is included, it denotes the number of times that the word has been encountered during the processing of text data.

For example, a typical word map file might be

```
Name=US_Business_News
SeqNo=13
Entries=133986
Fields=ID,WFC
Language=American
EscMode=RAW
\Words\
<s>     65536 34850
CAN'T   65537 2087
THE     65538 12004
DOLLAR  65539 169
IS      65540 4593
....
```

In this example, the word map is called "US_Business_News" and it has been updated 13 times since it was originally created. It contains a total of 133986 entries and word frequency counts are included. The language is "American" and there is no escaping used (e.g. can't is written `CAN'T` rather than the standard HTK escaped form of `CAN\'T`).

As noted above, when the `Fields` field is missing, the word map contains only the words and serves the purpose of a simple word list. For example, a typical word list might be defined as follows

```
Name=US_Business_News
Entries=10000
\Words\
A
ABLE
ABOUT
...
ZOO
```

Word lists are used to define subsets of the words in a word map. Whenever a tool requires a word list, a simple list of words can be input instead of the above. For example, the previous list could be input as

```
A
ABLE
ABOUT
...
ZOO
```

In this case, the default is to assume that all input words are escaped. If raw mode input is required, the configuration variable `INWMAPRAW` should be set true (see section 4.6).

As explained in section 4.6, by default HTK tools output word maps in HTK escaped form. However, this can be overridden by setting the LWMap configuration variable `OUTWMAPRAW` to true.

16.4 Class Map Files

A class map file defines one or more word classes. It has a header similar to that of a word map file, containing values for `Name`, `Entries`, `EscMode` and `Language`. In this case, the number of entries refers to the number of classes defined.

The class definitions are introduced by the keyword `\Classes\`. Each class definition has a single line sub-header consisting of a name, an id number, the number of class members (or non-members) and a keyword which must be `IN` or `NOTIN`. In the latter case, the class consists of all words *except* those listed i.e. the class is defined by its complement.

The following is a simple example of a class map file.

```
Name=Simple_Classes
Entries=97
EscMode=HTK
Language=British
\Classes\
ARTICLES 1 3 IN
  A
  AN
  THE
COLOURS 2 4 IN
  RED
  BLUE
  GREEN
  YELLOW
SHAPES 3 6 IN
  SQUARE
  CIRCLE
  ...
etc
```

This class map file defines 97 distinct classes, the first of which is a class called `ARTICLES` (id=1) with 3 members: (a, an, the).

For simple word-based language models, the class map file is used to define the class of unknown words. This is usually just the complement of the vocabulary list. For example, a typical class map file defining the *unknown* class `!!UNKID` might be

```
Name=Vocab_65k_V2.3
Entries=1
Language=American
EscMode=NONE
\Classes\
!!UNKID 1 65426 NOTIN
  A
  ABATE
  ABLE
  ABORT
  ABOUND
  ...
```

Since this case is so common, the tools also allow a plain vocabulary list to be supplied in place of a proper class map file. For example, supplying a class map file containing just

```
A
ABATE
ABLE
ABORT
ABOUND
...
```

would have an equivalent effect to the previous example provided that the LCMap configuration variables `UNKNOWNID` and `UNKNOWNNAME` have been set in order to define the id and name to be used for the unknown class. In the example given, including the following two lines in the configuration file would have the desired effect

```
LCMAP: UNKNOWNID = 1
LCMAP: UNKNOWNNAME = !!UNKID
```

Notice that the similarity with the special case of word lists described in section 16.3. A plain word list can therefore be used to define both a vocabulary subset and the unknown class. In a conventional language model, these are, of course, the same thing.

In a similar fashion to word maps, the input of a headerless class map can be set to raw mode by setting the LCMap configuration variable `INCMAPRAW` and all class maps can be output in raw mode by setting the configuration variable `OUTCMAPRAW` to true.

16.5 Gram Files

Statistical language models are estimated by counting the number of events in a sample source text. These event counts are stored in *gram* files. Provided that they share a common word map, gram files can be grouped together in arbitrary ways to form the raw data pool from which a language model can be constructed. For example, a text source containing 100m words could be processed and stored as two gram files. A few months later, a 3rd gram file could be generated from a newly acquired text source. This new gram file could then be added to the original two files to build a new language model. The original source text is not needed and the gram files need not be changed.

A gram file consists of a header followed by a sorted list of n-gram counts. The header contains the following items, each written on a separate line

1. n-gram size ie 2 for bigrams, 3 for trigrams, etc. (`Ngram=N`)
2. Word map. Name of word map to be used with this gram file. (`WMap=wmapname`)
3. First gram. The first n-gram in the file (`gram1 = w1 w2 w3 ...`)
4. Sequence number. If given then the actual word map must have a sequence number which is greater than or equal to this. (`SeqNo=nnn)`
5. Last gram. The last n-gram in the file (`gramN = w1 w2 w3 ...`)
6. Number of distinct n-grams in file. (`Entries = N`)
7. Word map check. This is an optional field containing a word and its id. It can be included as a double check that the correct word map is being used to interpret this gram file. The given word is looked up in the word map and if the corresponding id does not match, an error is reported. (`WMCheck = word id`)
8. Text source. This is an optional text string describing the text source which was used to generate the gram file (`Source=...`).

For example, a typical gram file header might be

```
Ngram = 3
WMap = US_Business_News
Entries = 50345980
WMCheck = XEROX 340987
Gram1 = AN ABLE ART
GramN = ZEALOUS ZOO OWNERS
Source = WSJ Aug 94 to Dec 94
```

The n-grams themselves begin immediately following the line containing the keyword `\Grams\`[1]. They are listed in lexicographic sort order such that for the n-gram $\{w_1 w_2 \dots w_N\}$, w_1 varies the least rapidly and w_N varies the most rapidly. Each n-gram consists of a sequence of N 3-byte word ids followed by a single 1-byte count. If the n-gram occurred more than 255 times, then it is repeated with the counts being interpreted to the base 256. For example, if a gram file contains the sequence

```
w1 w2 ... wN c1
w1 w2 ... wN c2
w1 w2 ... wN c3
```

[1] That is, the first byte of the binary data immediately follows the newline character

corresponding to the n-gram $\{w_1 w_2 \dots w_N\}$, the corresponding count is

$$c_1 + c_2 * 256 + c_3 * 256^2$$

When a group of gram files are used as input to a tool, they must be organised so that the tool receives n-grams as a single stream in sort order i.e. as far as the tool is concerned, the net effect must be as if there is just a single gram file. Of course, a sufficient approach would be to open all input gram files in parallel and then scan them as needed to extract the required sorted n-gram sequence. However, if two n-gram files were organised such that the last n-gram in one file was ordered before the first n-gram of the second file, it would be much more efficient to open and read the files in sequence. Files such as these are said to be `sequenced` and in general, HTK tools are supplied with a mix of sequenced and non-sequenced files. To optimise input in this general case, all HTK tools which input gram files start by scanning the header fields `gram1` and `gramN`. This information allows a sequence table to be constructed which determines the order in which the constituent gram file must be opened and closed. This sequence table is designed to minimise the number of individual gram files which must be kept open in parallel.

This gram file sequencing is invisible to the HTK user, but it is important to be aware of it. When a large number of gram files are accumulated to form a frequently used database, it may be worth copying the gram files using LGCopy. This will have the effect of transforming the gram files into a fully sequenced set thus ensuring that subsequent reading of the data is maximally efficient.

16.6 Frequency-of-frequency (FoF) Files

A FoF file contains a list of the number of times that an n-gram occurred just once, twice, three times, ..., n times. Its format is similar to a word map file. The header contains the following information

1. n-gram size ie 2 for bigrams, 3 for trigrams, etc. (`Ngram=N`)
2. the number of frequencies counted (i.e. the number of rows in the FoF table (`Entries=nnn`)
3. Text source. This is an optional text string describing the text source which was used to generate the gram files used to compute this FoF file. (`Source=...`).

More header fields may be defined later and the user is free to insert others.

The data part starts with the keyword `\FoFs\`. Each contains a list of the unigrams, bigrams, ..., n-grams occurring exactly `k` times, where `k` is the number of the row of the table – the first row shows the number of n-grams occurring exactly 1 time, for example.

As an example, the following is a FoF file computed from a set of trigram gram files.

```
Ngram = 3
Entries = 100
Source = WSJ Aug 94 to Dec 94
\FoFs\
     1020  23458  78654
      904  19864  56089
...
```

FoF files are generated by the tool `LFoF`. This tool will also output a list containing an estimate of the number of n-grams that will occur in a language model for a given cut-off – set the configuration parameter `LPCALC: TRACE = 3`.

16.7 Word LM file formats

Language models can be stored on disk in three different file formats - *text*, *binary* and *ultra*. The text format is the standard ARPA-MIT formad used to distribute pre-computed language models. The binary format is a proprietary file format which is optimised for flexibility and memory usage. All tools will output models in this format unless instructed otherwise. The *ultra* LM format is a further development of the binary LM format optimised for fast loading times and small memory footprint. At the same time, models stored in this format cannot be pruned further in terms of size and vocabulary.

16.7.1 The ARPA-MIT LM format

This format for storing n-gram back-off langauge models is defined as follows

```
<LM_definition> = [ { <comment> } ]
                  \data\
                  <header>
                  <body>
                  \end\
      <comment> = { <word> }
```

An ARPA-style language model file comes in two parts - the header and the n-gram definitions. The header contains a description of the contents of the file.

```
<header> = { ngram <int>=<int> }
```

The first `<int>` gives the n-gram order and the second `<int>` gives the number of n-gram entries stored.

For example, a trigram language model consists of three sections - the unigram, bigram and trigram sections respectively. The corresponding entry in the header indicates the number of entries for that section. This can be used to aid the loading-in procedure. The body part contains all sections of the language model and is defined as follows:

```
  <body>  = { <lmpart1> } <lmpart2>
<lmpart1> = \<int>-grams:
            { <ngramdef1> }
<lmpart2> = \<int>-grams:
            { <ngramdef2> }
<ngramdef1> = <float> { <word> } <float>
<ngramdef2> = <float> { <word> }
```

Each n-gram definition starts with a probability value stored as $\log_{10}$ followed by a sequence of n words describing the actual n-gram. In all sections excepts the last one this is followed by a back-off weight which is also stored as $\log_{10}$. The following example shows an extract from a trigram language model stored in the ARPA-text format.

```
\data\
ngram 1=19979
ngram 2=4987955
ngram 3=6136155

\1-grams:
-1.6682  A      -2.2371
-5.5975  A'S    -0.2818
-2.8755  A.     -1.1409
-4.3297  A.'S   -0.5886
-5.1432  A.S    -0.4862
...

\2-grams:
-3.4627  A  BABY    -0.2884
-4.8091  A  BABY'S  -0.1659
-5.4763  A  BACH    -0.4722
-3.6622  A  BACK    -0.8814
...

\3-grams:
-4.3813  !SENT_START    A       CAMBRIDGE
-4.4782  !SENT_START    A       CAMEL
-4.0196  !SENT_START    A       CAMERA
-4.9004  !SENT_START    A       CAMP
-3.4319  !SENT_START    A       CAMPAIGN
...
\end\
```

Component	# with back-off weights	Total
unigram	65,467	65,467
bigram	2,074,422	6,224,660
trigram	4,485,738	9,745,297
fourgram	0	9,946,193

Table 16.1: Component statistics for a 65k word fourgram language model with cut-offs: bigram 1, trigram 2, fourgram 2.

16.7.2 The modified ARPA-MIT format

The efficient loading of the language model file requires prior information as to memory requirements. Such information is partially available from the header of the file which shows how many entries will be found in each section of the model. From the back-off nature of the language model it is clear that the back-off weight associated with an n-gram $(w_1, w_2, \ldots, w_{n-1})$ is only useful when $p(w_n|w_1, word_2, \ldots, w_{n-1})$ is an explicitly entry in the file or computed via backing-off to the corresponding $(n-1)$-grams. In other words, the presence of a back-off weight associated with the n-gram $w_1, w_2, \ldots, w_{n-1}$ can be used to indicate the existence of explicit n-grams $w_1, w_2, \ldots, w_n$. The use of such information can greatly reduce the storage requirements of the language model since the back-off weight requires extra storage. For example, considering the statistics shown in table 16.1, such selective memory allocation can result in dramatic savings. This information is accommodated by modifying the syntax and semantics of the rule

```
<ngramdef1> = <float> { <word> } [ <float> ]
```

whereby a back-off weight associated with n-gram $(w_1, w_2, \ldots, w_{n-1})$ indicates the existence of n-grams $(w_1, w_2, \ldots, w_n)$. This version will be referred to as the modified ARPA-text format.

16.7.3 The binary LM format

This format is the binary version of modified ARPA-text format. It was designed to be a compact, self-contained format which aids the fast loading of large language model files. The format is similar to the original ARPA-text format with the following modification

```
<header> = { (ngram <int>=<int>) | (ngram <int>~<int>) }
```

The first alternative in the rule describes a section stored as text, the second one describes a section stored in binary. The unigram section of a language model file is always stored as text.

```
<ngramdef> = <txtgram> | <bingram>
 <txtgram> = <float> { <word> } [ <float> ]
 <bingram> = <f_type> <f_size> <f_float> { <f_word> } [ <f_float> ]
```

In the above definition, `<f_type>` is a 1-byte flags field, `<f_size>` is a 1-byte unsigned number indicating the total size in bytes of the remaining fields, `<f_float>` is a 4-bytes field for the n-gram probability, `<f_word>` is a numeric word id, and the last `<f_float>` is the back-off weight. The numeric word identifier is an unsigned integer assigned to each word in the order of occurrence of the words in the unigram section. The minimum size of this field is 2-bytes as used in vocabulary lists with up to 65,5355 words. If this number is exceeded the field size is automatically extended to accommodate all words. The size of the fields used to store the probability and back-off weight are typically 4 bytes, however this may vary on different computer architectures. The least significant bit of the flags field indicates the presence/absence of a back-off weight with corresponding values 1/0. The remaining bits of the flags field are not used at present.

16.8 Class LM file formats

Class language models replace the word language model described in section 16.7 with an identical component which models class n-grams instead of word n-grams. They add to this a second component which includes the deterministic word-to-class mapping with associated word-given-class probabilities, expressed either as counts (which are normalised to probabilities on loading) or as explicit natural log probabilities. These two components are then either combined into a single file or are pointed to with a special link file.

16.8.1 Class counts format

The format of a word-given-class counts file, as generated using the `-q` option from CLUSTER, is as follows:
`Word|Class counts`
[blank line]
`Derived from: <file>`
`Number of classes: <int>`
`Number of words: <int>`
`Iterations: <int>`
[blank line]
`Word Class name Count`
followed by one line for each word in the model of the form:
`<word> CLASS<int> <int>`

The fields are mostly self-explanatory. The `Iterations:` header is for information only and records how many iterations had been performed to produce the classmap contained within the file, and the `Derived from:` header is similarly also for display purposes only. Any number of headers may be present; the header section is terminated by finding a line beginning with the four characters making up `Word`. The colon-terminated headers may be in any order.

`CLASS<int>` must be the name of a class in the classmap (technically actually the wordmap) used to build the class-given-class history n-gram component of the language model – the file built by LBUILD. In the current implementation these class names are restricted to being of the form `CLASS<int>`, although a modification to the code in LMODEL.C would allow this restriction to be removed. Each line after the header specifies the count of each word and the class it is in, so for example
`THE CLASS73 1859`
would specify that the word `THE` was in class `CLASS73` and occurred 1859 times.

16.8.2 The class probabilities format

The format of a word-given-class probabilities file, as generated using the `-p` option from CLUSTER, is very similar to that of the counts file described in the previous sub-section, and is as follows:
`Word|Class probabilities`
[blank line]
`Derived from: <file>`
`Number of classes: <int>`
`Number of words: <int>`
`Iterations: <int>`
[blank line]
`Word Class name Probability (log)`
followed by one line for each word in the model of the form:
`<word> CLASS<int> <float>`

As in the previous section, the fields are mostly self-explanatory. The `Iterations:` header is for information only and records how many iterations had been performed to produce the classmap contained within the file, and the `Derived from:` header is similarly also for display purposes only. Any number of headers may be present; the header section is terminated by finding a line beginning with the four characters making up `Word`. The colon-terminated headers may be in any order.

`CLASS<int>` must be the name of a class in the classmap (technically actually the wordmap) used to build the class-given-class history n-gram component of the language model – the file built by LBUILD. In the current implementation these class names are restricted to being of the form `CLASS<int>`, although a modification to the code in LMODEL.C would allow this restriction to be removed. Each `<float>` specifies the natural logarithm of the probability of the word given the class, or -99.9900 if the probability of the word is less than 1.0×10^{-20}.

16.8.3 The class LM three file format

A special class language model file, generated by LLINK, links together either the word-given-class probability or count files described above (either can be used to give the same results) with a class-

given-class history n-gram file constructed using LBuild. It is a simple text file which specifies the filename of the two relevant components:

```
Class-based LM
Word|Class counts:  <file> or Word|Class probabilities:  <file>
Class|Class grams:  <file>
```

The second line must state `counts` or `probabilities` as appropriate for the relevant file.

16.8.4 The class LM single file format

An alternative to the class language model file described in section 16.8.3 is the composite single-file class language model file, produced by LLink `-s` – this does *not* require the two component files to be present since it integrates them into a single file. The format of this resulting file is as follows:

```
CLASS MODEL
Word|Class <string: counts/probs>

Derived from: <file>
Number of classes: <int>
Number of words: <int>
Iterations: <int>

Class n-gram counts follow; word|class component is at end of file.
```

The second line must state either `counts` or `probabilities` as appropriate for the relevant component file used when constructing this composite file. The fields are mostly self-explanatory. The `Iterations:` header is for information only and records how many iterations had been performed to produce the classmap contained within the file, and the `Derived from:` header is similarly also for display purposes only. Any number of headers may be present; the header section is terminated by finding a line beginning with the five characters making up `Class`. The colon-terminated headers may be in any order.

The class-given-classes n-gram component of the model then follows immediately in any of the formats supported by word n-gram language models – ie. those described in section 16.7. No blank lines are expected between the header shown above and the included model, although they may be supported by the embedded model.

Immediately following the class-given-classes n-gram component follows the body of the word-given-class probabilities or counts file as described in sections 16.8.1 and 16.8.2 above. That is, the remainder of the file consists of lines of the form:

```
<word> CLASS<int> <float/int>
```

One line is expected for each word as specified in the header at the top of the file. Integer word counts should be provided in the final field for each word in the case of a counts file, or word-given-class probabilities if a probabilities file – as specified by the second line of the overall file. In the latter case each `<float>` specifies the natural logarithm of the probability of the word given the class, or -99.9900 if the probability of the word is less than 1.0×10^{-20}.

`CLASS<int>` must be the name of a class in the classmap (technically actually the wordmap) used to build the class-given-class history n-gram component of the language model – the file built by LBuild. In the current implementation these class names are restricted to being of the form `CLASS<int>`, although a modification to the code in LModel.c would allow this restriction to be removed.

16.9 Language modelling tracing

Each of the HTK language modelling tools provides its own trace facilities, as documented with the relevant tool in chapter 17. The standard libraries also provide their own trace settings, which can be set in a passed configuration file. Each of the supported trace levels is documented below with the octal value necessary to enable it.

16.9.1 LCMap

- 0001 Top level tracing
- 0002 Class map loading

16.9.2 LGBase

- 0001 Top level tracing
- 0002 Trace n-gram squashing
- 0004 Trace n-gram buffer sorting
- 0010 Display n-gram input set tree
- 0020 Display maximum parallel input streams
- 0040 Trace parallel input streaming
- 0100 Display information on FoF input/output

16.9.3 LModel

- 0001 Top level tracing
- 0002 Trace loading of language models
- 0004 Trace saving of language models
- 0010 Trace word mappings
- 0020 Trace n-gram lookup

16.9.4 LPCalc

- 0001 Top level tracing
- 0002 FoF table tracing

16.9.5 LPMerge

- 0001 Top level tracing

16.9.6 LUtil

- 0001 Top level tracing
- 0002 Show header processing
- 0004 Hash table tracing

16.9.7 LWMap

- 0001 Top level tracing
- 0002 Trace word map loading
- 0004 Trace word map sorting

16.10 Run-time configuration parameters

Section 4.10 lists the major standard HTK configuration parameter options whilst the rest of chapter 4 describes the general HTK environment and how to set those configuration parameters, whilst chapter 18 provides a comprehensive list. For ease of reference those parameters specifically relevant to the language modelling tools are reproduced in table 16.1.

Module	Name	Description
HShell	ABORTONERR	Core dump on error (for debugging)
HShell	HLANGMODFILTER	Filter for language model file input
HShell	HLABELFILTER	Filter for Label file input
HShell	HDICTFILTER	Filter for Dictionary file input
HShell	LGRAMFILTER	Filter for gram file input
HShell	LWMAPFILTER	Filter for word map file input
HShell	LCMAPFILTER	Filter for class map file input
HShell	HLANGMODOFILTER	Filter for language model file output
HShell	HLABELOFILTER	Filter for Label file output
HShell	HDICTOFILTER	Filter for Dictionary file output
HShell	LGRAMOFILTER	Filter for gram file output
HShell	LWMAPOFILTER	Filter for word map file output
HShell	LCMAPOFILTER	Filter for class map file output
HShell	MAXTRYOPEN	Number of file open retries
HShell	NONUMESCAPES	Prevent string output using \012 format
HShell	NATURALREADORDER	Enable natural read order for HTK binary files
HShell	NATURALWRITEORDER	Enable natural write order for HTK binary files
HMem	PROTECTSTAKS	Warn if stack is cut-back (debugging)
	TRACE	Trace control (default=0)
	STARTWORD	Set sentence start symbol (<s>)
	ENDWORD	Set sentence end symbol (</s>)
	UNKNOWNNAME	Set OOV class symbol (!!UNK)
	RAWMITFORMAT	Disable HTK escaping for LM tools
LWMap	INWMAPRAW	Disable HTK escaping for input word lists and maps
LWMap	OUTWMAPRAW	Disable HTK escaping for output word lists and maps
LCMap	INCMAPRAW	Disable HTK escaping for input class lists and maps
LCMap	OUTCMAPRAW	Disable HTK escaping for output class lists and maps
LCMap	UNKNOWNID	Set unknown symbol class ID (1)
LCMap	USEINTID	Use 4 byte ID fields to save binary models (see section 16.10.1)
LPCalc	UNIFLOOR	Unigram floor count (1)
LPCalc	KRANGE	Good-Turing discounting range (7)
LPCalc	*n*G_CUTOFF	*n*-gram cutoff (eg. 2G_CUTOFF) (1)
LPCalc	DCTYPE	Discounting type (TG for Turing-Good or ABS for Absolute) (TG)
LGBase	CHECKORDER	Check N-gram ordering in files

Table. 16.1 Configuration Parameters used in Operating Environment

16.10.1 USEINTID

Setting this to T as opposed to its default of F forces the LModel library to save language models using an unsigned int for each word ID as opposed to the default of an unsigned short. In most systems these lengths correspond to 4-byte and 2-byte fields respectively. Note that if you do not set this that LModel will automatically choose an int field size if the short field is too small – the

exception to this is if you have compiled with LM_ID_SHORT which limits the field size to an unsigned short, in which case the tool will be forced to abort; see section 16.11.1 below.

16.11 Compile-time configuration parameters

There are some compile-time switches which may be set when building the language modelling library and tools.

16.11.1 LM_ID_SHORT

When compiling the HTK language modelling library, setting LM_ID_SHORT (for example by passing -D LM_ID_SHORT to the C compiler) forces the compiler to use an unsigned short for each language model ID it stores, as opposed to the default of an unsigned int – in most systems this will result in either a 2-byte integer or a 4-byte integer respectively. If you set this then you *must* ensure you also set LM_ID_SHORT when compiling the HTK language modelling tools too, otherwise you will encounter a mismatch leading to strange results! (Your compiler may warn of this error, however). For this reason it is safest to set LM_ID_SHORT via a #define in LModel.h. You might want to set this if you know how many distinct word ids you require and you do not want to waste memory, although on some systems using shorts can actually be slower than using a full-size int.

Note that the run-time USEINTID parameter described in section 16.10.1 above only affects the size of ID fields when saving a binary model from LModel, so is independent of LM_ID_SHORT. The only restriction is that you cannot load or save a model with more ids than can fit into an unsigned short when LM_ID_SHORT is set – the tools will abort with an error should you try this.

16.11.2 LM_COMPACT

When LM_COMPACT is defined at compile time, when a language model is loaded then its probabilities are compressed into an unsigned short as opposed to being loaded into a float. The exact size of these types depends on your processor architecture, but in general an unsigned short is more than half as small as a float. Using the compact storage type therefore significantly reduces the accuracy with which probabilities are stored.

The side effect of setting this is therefore reduced accuracy when running a language model, such as when using LPlex; or a loss of accuracy when rebuilding from an existing language model using LMerge, LAdapt, LBuild or HLMCopy.

16.11.3 LMPROB_SHORT

Setting LMPROB_SHORT causes language model probabilities to be stored and loaded using a short type. Unlike LM_COMPACT, this option certainly *does* affect the writing of language model files. If you save a file using this format then you must ensure you reload it in the same way to ensure you obtain sensible results.

16.11.4 INTERPOLATE_MAX

If the library and tools are compiled with INTERPOLATE_MAX then language model interpolation in LPlex and the LPMerge library (which is used by LAdapt and LMerge) will ignore the individual model weights and always pick the highest probability from each of the models at any given point. Note that this option will *not* normalise the models.

16.11.5 SANITY

Turning on SANITY when compiling the library will add a word map check to LGBase and some sanity checks to LPCalc.

16.11.6 INTEGRITY_CHECK

Compiling with INTEGRITY_CHECK will add run-time integrity checking to the Cluster tool. Specifically it will check that the class counts have not become corrupted and that all maximum likelihood move updates have been correctly calculated. You should not need to enable this unless you suspect

a major tool problem, and doing so will slow down the tool execution. It could probe useful if you wanted to adapt the way the clustering works, however.

Part IV

Reference Section

Chapter 17

The HTK Tools

17.1 Cluster

17.1.1 Function

This program is used to statistically cluster words into deterministic classes. The main purpose of CLUSTER is to optimise a class map on the basis of the training text likelihood, although it can also import an existing class map and generate one of the files necessary for creating a class-based language model from the HTK language modelling tools.

Class-based language models use a reduced number of classes relative to the number of words, with each class containing one or more words, to allow a language model to be able to generalise to unseen training contexts. Class-based models also typically require less training text to produce a well-trained model than a similar complexity word model, and are often more compact due to the much reduced number of possible distinct history contexts that can be encountered in the training data.

CLUSTER takes as input a set of one or more training text gram files, which may optionally be weighted on input, and their associated word map. It then clusters the words in the word map into classes using a bigram likelihood measure. Due to the computational complexity of this task a sub-optimal greedy algorithm is used, but multiple iterations of this algorithm may be performed in order to further refine the class map, although at some point a local maximum will be reached where the class map will not change further.[1] In practice as few as two iterations may be perfectly adequate, even with large training data sets.

The algorithm works by considering each word in the vocabulary in turn and calculating the change in bigram training text likelihood if the word was moved from its default class (see below) to each other class in turn. The word is then moved to the class which increases the likelihood the most, or it is left in its current class if no such increase is found. Each iteration of the algorithm considers each word exactly once. Because this can be a slow process, with typical execution times measured in terms of a few hours, not a few minutes, the CLUSTER tool also allows *recovery* files to be written at regular intervals, which contain the current class map part-way through an iteration along with associated files detailing at what point in the iteration the class map was exported. These files are not essential for operation, but might be desirable if there is a risk of a long-running process being killed via some external influence. During the execution of an iteration the tool claims no new memory,[2] so it cannot crash in the middle of an iteration due to a lack of memory (it can, however, fail to start an iteration in the first place).

Before beginning an iteration, CLUSTER places each word either into a default class or one specified via the `-l`, import classmap, or `-x`, use recovery, options. The default distribution, given m classes, is to place the most frequent $(m-1)$ words into singleton classes and then the remainder into the remaining class. CLUSTER allows words to be considered in either decreasing frequency of occurrence order, or the order they are encountered in the word map. The popular choice is to use the former method, although in experiments it was found that the more random second approach typically gave better class maps after fewer iterations in practice.[3] The `-w` option specifies this choice.

During execution CLUSTER will always write a logfile describing the changes it makes to the classmap, unless you explicitly disable this using the `-n` option. If the `-v` switch is used then this logfile is written in explicit English, allowing you to easily trace the execution of the clusterer; without `-v` then similar information is exported in a more compact format.

Two or three special classes are also defined. The sentence start and sentence end word tokens are always kept in singleton classes, and optionally the unknown word token can be kept in a singleton class too – pass the `-k` option.[4] These tokens are placed in these classes on initialisation and no moves to or from these classes are ever considered.

Language model files are built using either the `-p` or `-q` options, which are effectively equivalent if using the HTK language modelling tools as black boxes. The former creates a word-given-class probabilities file, whilst the latter stores word counts and lets the language model code itself calculate the same probabilities.

[1] On a 65,000 word vocabulary test set with 170 million words of training text this was found to occur after around 45 iterations

[2] other than a few small local variables taken from the stack as functions are called

[3] Note that these schemes are approximately similar, since the most frequent words are most likely to be encountered sooner in the training text and thus occur higher up in the word map

[4] The author always uses this option but has not empirically tested its efficaciousness

17.1.2 Use

CLUSTER is invoked by the command line

```
Cluster [options] mapfile [mult] gramfile [[mult] gramfile ...]
```

The given word map is loaded and then each of the specified gram files is imported. The list of input gram files can be interspersed with multipliers. These are floating-point format numbers which must begin with a plus or minus character (e.g. `+1.0`, `-0.5`, etc.). The effect of a multiplier `mult` is to scale the n-gram counts in the following gram files by the factor `mult`. The resulting scaled counts are rounded to the nearest integer when actually used in the clustering algorithm. A multiplier stays in effect until it is redefined.

The allowable options to CLUSTER are as follows

`-c n` Use `n` classes. This specifies the number of classes that should be in the resultant class map.

`-i n` Perform `n` iterations. This is the number of iterations of the clustering algorithm that should be performed. (If you are using the `-x` option then completing the current iteration does not count towards the total number, so use `-i 0` to complete it and then finish)

`-k` Keep the special unknown word token in its own singleton class. If not passed it can be moved to or from any class.

`-l fn` Load the classmap `fn` at start up and when performing any further iterations do so from this starting point.

`-m` Record the running value of the maximum likelihood function used by the clusterer to optimised the training text likelihood in the log file. This option is principally provided for debugging purposes.

`-n` Do not write any log file during execution of an iteration.

`-o fn` Specify the prefix of all output files. All output class map, logfile and recovery files share the same filename prefix, and this is specified via the `-o` switch. The default is `cluster`.

`-p fn` Write a word-given-class probabilities file. Either this or the `-q` switch are required to actually build a class-based language model. The HTK language model library, LMODEL, supports both probability and count-based class files. There is no difference in use, although each allows different types of manual manipulation of the file. Note that if you do not pass `-p` or `-q` you may run CLUSTER at a later date using the `-l` and `-i 0` options to just produce a language model file.

`-q fn` Write a word-given-class counts file. See the documentation for `-p`.

`-r n` Write recovery files after moving `n` words since the previous recovery file was written or an iteration began. Pass `-r n` to disable writing of recovery files.

`-s tkn` Specify the sentence start token.

`-t tkn` Specify the sentence end token.

`-u tkn` Specify the unknown word token.

`-v` Use verbose log file format.

`-w [WMAP/FREQ]` Specify the order in which word moves are considered. Default is `WMAP` in which words are considered in the order they are encountered in the word map. Specifying `FREQ` will consider the most frequent word first and then the remainder in decreasing order of frequency.

`-x fn` Continue execution from recovery file `fn`.

CLUSTER also supports the standard options `-A`, `-C`, `-D`, `-S`, `-T`, and `-V` as described in section 4.4.

17.1.3 Tracing

CLUSTER supports the following trace options, where each trace flag is given using an octal base:

`00001` basic progress reporting.

`00002` report major file operations - good for following start-up.

`00004` more detailed progress reporting.

`00010` trace memory usage during execution and at end.

Trace flags are set using the `-T` option or the `TRACE` configuration variable.

17.2 HBuild

17.2.1 Function

This program is used to convert input files that represent language models in a number of different formats and output a standard HTK lattice. The main purpose of HBuild is to allow the expansion of HTK multi-level lattices and the conversion of bigram language models (such as those generated by HLStats) into lattice format.

The specific input file types supported by HBuild are:

1. HTK multi-level lattice files.
2. Back-off bigram files in ARPA/MIT-LL format.
3. Matrix bigram files produced by HLStats.
4. Word lists (to generate a word-loop grammar).
5. Word-pair grammars in ARPA Resource Management format.

The formats of both types of bigram supported by HBuild are described in Chapter 12. The format for multi-level HTK lattice files is described in Chapter 20.

17.2.2 Use

HBuild is invoked by the command line

```
HBuild [options] wordList outLatFile
```

The `wordList` should contain a list of all the words used in the input language model. The options specify the type of input language model as well as the source filename. If none of the flags specifying input language model type are given a simple word-loop is generated using the `wordList` given. After processing the input language model, the resulting lattice is saved to file `outLatFile`.

The operation of HBuild is controlled by the following command line options

`-b` Output the lattice in binary format. This increases speed of subsequent loading (default ASCII text lattices).

`-m fn` The matrix format bigram in `fn` forms the input language model.

`-n fn` The ARPA/MIT-LL format back-off bigram in `fn` forms the input language model.

`-s st en` Set the bigram entry and exit words to `st` and `en`. (Default `!ENTER` and `!EXIT`). Note that no words will follow the exit word, or precede the entry word. Both the entry and exit word must be included in the `wordList`. This option is only effective in conjunction with the `-n` option.

`-t st en` This option is used with word-loops and word-pair grammars. An output lattice is produced with an initial word-symbol `st` (before the loop) and a final word-symbol `en` (after the loop). This allows initial and final silences to be specified. (Default is that the initial and final nodes are labelled with `!NULL`). Note that `st` and `en` shouldn't be included in the `wordList` unless they occur elsewhere in the network. This is only effective for word-loop and word-pair grammars.

`-u s` The unknown word is `s` (default !NULL). This option only has an effect when bigram input language models are specified. It can be used in conjunction with the `-z` flag to delete the symbol for unknown words from the output lattice.

`-w fn` The word-pair grammar in `fn` forms the input language model. The file must be in the format used for the ARPA Resource Management grammar.

`-x fn` The extended HTK lattice in `fn` forms the input language model. This option is used to expand a multi-level lattice into a single level lattice that can be processed by other HTK tools.

`-z` Delete (zap) any references to the unknown word (see `-u` option) in the output lattice.

HBuild also supports the standard options `-A`, `-C`, `-D`, `-S`, `-T`, and `-V` as described in section 4.4.

17.2.3 Tracing

HBuild supports the following trace options where each trace flag is given using an octal base

`0001` basic progress reporting.

Trace flags are set using the `-T` option or the `TRACE` configuration variable.

17.3 HCompV

17.3.1 Function

This program will calculate the global mean and covariance of a set of training data. It is primarily used to initialise the parameters of a HMM such that all component means and all covariances are set equal to the global data mean and covariance. This might form the first stage of a *flat start* training scheme where all models are initially given the same parameters. Alternatively, the covariances may be used as the basis for Fixed Variance and Grand Variance training schemes. These can sometimes be beneficial in adverse conditions where a fixed covariance matrix can give increased robustness.

When training large model sets from limited data, setting a floor is often necessary to prevent variances being badly underestimated through lack of data. One way of doing this is to define a variance macro called `varFloorN` where `N` is the stream index. HCompV can also be used to create these variance floor macros with values equal to a specified fraction of the global variance.

Another application of HCompV is the estimation of mean and variance vectors for use in cluster-based mean and variance normalisation schemes. Given a list of utterances and a speaker pattern HCompV will estimate a mean and a variance for each speaker.

17.3.2 Use

HCompV is invoked via the command line

```
HCompV [options] [hmm] trainFiles ...
```

where `hmm` is the name of the physical HMM whose parameters are to be initialised. Note that no HMM name needs to be specified when cepstral mean or variance vectors are estimated (`-c` option). The effect of this command is to compute the covariance of the speech training data and then copy it into every Gaussian component of the given HMM definition. If there are multiple data streams, then a separate covariance is estimated for each stream. The HMM can have a mix of diagonal and full covariances and an option exists to update the means also. The HMM definition can be contained within one or more macro files loaded via the standard `-H` option. Otherwise, the definition will be read from a file called `hmm`. Any tyings in the input definition will be preserved in the output. By default, the new updated definition overwrites the existing one. However, a new definition file including any macro files can be created by specifying an appropriate target directory using the standard `-M` option.

In addition to the above, an option `-f` is provided to compute variance floor macros equal to a specified fraction of the global variance. In this case, the newly created macros are written to a file called `vFloors`. For each stream `N` defined for `hmm`, a variance macro called `varFloorN` is created. If a target directory is specified using the standard `-M` option then the new file will be written there, otherwise it is written in the current directory.

The list of train files can be stored in a script file if required. Furthermore, the data used for estimating the global covariance can be limited to that corresponding to a specified label.

The calculation of cluster-based mean and variances estimates is enabled by the option `-c` which specifies the output directory where the estiamted vectors should be stored.

The detailed operation of HCompV is controlled by the following command line options

`-c s` Calculate cluster-based mean/variance estimate and store results in the specified directory.

`-k s` Speaker pattern for cluster-based mean/variance estimation. Each utterance filename is matched against the pattern and the characters that are matched against `%` are used as the cluster name. One mean/variance vector is estimated for each cluster.

`-q s` For cluster-based mean/variance estimation different types of output can be requested. Any subset of the letters `nmv` can be specified. Specifying `n` causes the number of frames in a cluster to be written to the output file. `m` and `v` cause the mean and variance vectors to be included, respectively.

`-f f` Create variance floor macros with values equal to `f` times the global variance. One macro is created for each input stream and the output is stored in a file called `vFloors`.

`-l s` The string `s` must be the name of a segment label. When this option is used, HCompV searches through all of the training files and uses only the speech frames from segments with the given label. When this option is not used, HCompV uses all of the data in each training file.

`-m` The covariances of the output HMM are always updated however updating the means must be specifically requested. When this option is set, HCompV updates all the HMM component means with the sample mean computed from the training files.

`-o s` The string `s` is used as the name of the output HMM in place of the source name.

`-v f` This sets the minimum variance (i.e. diagonal elements of the covariance matrix) to the real value `f` (default value 0.0).

`-B` Output HMM definition files in binary format.

`-F fmt` Set the source data format to `fmt`.

`-G fmt` Set the label file format to `fmt`.

`-H mmf` Load HMM macro model file `mmf`. This option may be repeated to load multiple MMFs.

`-I mlf` This loads the master label file `mlf`. This option may be repeated to load several MLFs.

`-L dir` Search directory `dir` for label files (default is to search current directory).

`-M dir` Store output HMM macro model files in the directory `dir`. If this option is not given, the new HMM definition will overwrite the existing one.

`-X ext` Set label file extension to `ext` (default is `lab`).

HCompV also supports the standard options `-A`, `-C`, `-D`, `-S`, `-T`, and `-V` as described in section 4.4.

17.3.3 Tracing

HCompV supports the following trace options where each trace flag is given using an octal base

`00001` basic progress reporting.

`00002` show covariance matrices.

`00004` trace data loading.

`00010` list label segments.

Trace flags are set using the `-T` option or the `TRACE` configuration variable.

17.4 HCopy

17.4.1 Function

This program will copy one or more data files to a designated output file, optionally converting the data into a parameterised form. While the source files can be in any supported format, the output format is always HTK. By default, the whole of the source file is copied to the target but options exist to only copy a specified segment. Hence, this program is used to convert data files in other formats to the HTK format, to concatenate or segment data files, and to parameterise the result. If any option is set which leads to the extraction of a segment of the source file rather than all of it, then segments will be extracted from all source files and concatenated to the target.

Labels will be copied/concatenated if any of the options indicating labels are specified (`-i -l -x -G -I -L -P -X`). In this case, each source data file must have an associated label file, and a target label file is created. The name of the target label file is the root name of the target data file with the extension `.lab`, unless the `-X` option is used. This new label file will contain the appropriately copied/truncated/concatenated labels to correspond with the target data file; all start and end boundaries are recalculated if necessary.

When used in conjunction with HSLab, HCopy provides a facility for tasks such as cropping silence surrounding recorded utterances. Since input files may be coerced, HCopy can also be used to convert the parameter kind of a file, for example from WAVEFORM to MFCC, depending on the configuration options. Not all possible conversions can actually be performed; see Table 17.1 for a list of valid conversions. Conversions must be specified via a configuration file as described in chapter 5. Note also that the parameterisation qualifier `_N` cannot be used when saving files to disk, and is meant only for on-the-fly parameterisation.

17.4.2 Use

HCopy is invoked by typing the command line

```
HCopy [options] sa1 [ + sa2 + ... ] ta  [ sb1 [ + sb2 + ... ] tb ... ]
```

This causes the contents of the one or more source files `sa1`, `sa2`, ... to be concatenated and the result copied to the given target file `ta`. To avoid the overhead of reinvoking the tool when processing large databases, multiple sources and targets may be specified, for example

```
HCopy srcA.wav + srcB.wav tgtAB.wav srcC.wav tgtD.wav
```

will create two new files `tgtAB.wav` and `tgtD.wav`. HCopy takes file arguments from a script specified using the `-S` option exactly as from the command line, except that any newlines are ignored.

The allowable options to HCopy are as follows where all times and durations are given in 100 ns units and are written as floating-point numbers.

`-a i` Use level i of associated label files with the `-n` and `-x` options. Note that this is not the same as using the `TRANSLEVEL` configuration variable since the `-a` option still allows all levels to be copied through to the output files.

`-e f` End copying from the source file at time `f`. The default is the end of the file. If `f` is negative or zero, it is interpreted as a time relative to the end of the file, while a positive value indicates an absolute time from the start of the file.

`-i mlf` Output label files to master file `mlf`.

`-l s` Output label files to the directory `s`. The default is to output to the current directory.

`-m t` Set a margin of duration `t` around the segments defined by the `-n` and `-x` options.

`-n i [j]` Extract the speech segment corresponding to the `i`'th label in the source file. If `j` is specified, then the segment corresponding to the sequence of labels `i` to `j` is extracted. Labels are numbered from their position in the label file. A negative index can be used to count from the end of the label list. Thus, `-n 1 -1` would specify the segment starting at the first label and ending at the last.

`-s f` Start copying from the source file at time `f`. The default is 0.0, ie the beginning of the file.

`-t n` Set the line width to `n` chars when formatting trace output.

`-x s [n]` Extract the speech segment corresponding to the first occurrence of label `s` in the source file. If `n` is specified, then the `n`'th occurrence is extracted. If multiple files are being concatenated, segments are extracted from each file in turn, and the label must exist for each concatenated file.

`-F fmt` Set the source data format to `fmt`.

`-G fmt` Set the label file format to `fmt`.

`-I mlf` This loads the master label file `mlf`. This option may be repeated to load several MLFs.

`-L dir` Search directory `dir` for label files (default is to search current directory).

`-O fmt` Set the target data format to `fmt`.

`-P fmt` Set the target label format to `fmt`.

`-X ext` Set label file extension to `ext` (default is `lab`).

HCopy also supports the standard options `-A`, `-C`, `-D`, `-S`, `-T`, and `-V` as described in section 4.4.

Note that the parameter kind conversion mechanisms described in chapter 5 will be applied to all source files. In particular, if an automatic conversion is requested via the configuration file, then HCopy will copy or concatenate the converted source files, not the actual contents. Similarly, automatic byte swapping may occur depending on the source format and the configuration variable `BYTEORDER`. Because the sampling rate may change during conversions, the options that specify a position within a file i.e. `-s` and `-e` use absolute times rather than sample index numbers. All times in HTK are given in units of 100ns and are written as floating-point numbers. To save writing long strings of zeros, standard exponential notation may be used, for example `-s 1E6` indicates a start time of 0.1 seconds from the beginning of the file.

	Outputs									
Inputs	WAVEFORM	LPC	LPREFC	LPCEPSTRA	IREFC	MFCC	FBANK	MELSPEC	USER	DISCRETE
WAVEFORM	√	√	√	√	√	√	√	√		√
LPC		√	√	√	√					√
LPREFC		√	√	√	√					√
LPCEPSTRA		√	√	√	√					√
IREFC		√	√	√	√					√
MFCC						√				√
FBANK						√	√			√
MELSPEC						√	√	√		√
USER									√	√
DISCRETE										√

Table. 17.1 Valid Parameter Conversions

Note that truncations are performed *after* any desired coding, which may result in a loss of time resolution if the target file format has a lower sampling rate. Also, because of windowing effects, truncation, coding, and concatenation operations are not necessarily interchangeable. If in doubt, perform all truncation/concatenation in the waveform domain and then perform parameterisation as a last, separate invocation of HCopy.

17.4.3 Trace Output

HCopy supports the following trace options where each trace flag is given using an octal base

`00001` basic progress reporting.

`00002` source and target file formats and parameter kinds.

`00004` segment boundaries computed from label files.

`00010` display memory usage after processing each file.

Trace flags are set using the `-T` option or the `TRACE` configuration variable.

17.5 HDMan

17.5.1 Function

The HTK tool HDMan is used to prepare a pronunciation dictionary from one or more sources. It reads in a list of *editing* commands from a script file and then outputs an edited and merged copy of one or more dictionaries.

Each source pronunciation dictionary consists of comment lines and definition lines. Comment lines start with the `#` character (or optionally any one of a set of specified comment chars) and are ignored by HDMan. Each definition line starts with a word and is followed by a sequence of symbols (phones) that define the pronunciation. The words and the phones are delimited by spaces or tabs, and the end of line delimits each definition.

Dictionaries used by HDMan are read using the standard HTK string conventions (see section 4.6), however, the command `IR` can be used in a HDMan source edit script to switch to using this raw format. Note that in the default mode, words and phones should not begin with unmatched quotes (they should be escaped with the backslash). All dictionary entries must already be alphabetically sorted before using HDMan.

Each edit command in the script file must be on a separate line. Lines in the script file starting with a `#` are comment lines and are ignored. The commands supported are listed below. They can be displayed by HDMan using the `-Q` option.

When no edit files are specified, HDMan simply merges all of the input dictionaries and outputs them in sorted order. All input dictionaries must be sorted. Each input dictionary `xxx` may be processed by its own private set of edit commands stored in `xxx.ded`. Subsequent to the processing of the input dictionaries by their own unique edit scripts, the merged dictionary can be processed by commands in `global.ded` (or some other specified global edit file name).

Dictionaries are processed on a word by word basis in the order that they appear on the command line. Thus, all of the pronunciations for a given word are loaded into a buffer, then all edit commands are applied to these pronunciations. The result is then output and the next word loaded.

Where two or more dictionaries give pronunciations for the same word, the default behaviour is that only the first set of pronunciations encountered are retained and all others are ignored. An option exists to override this so that all pronunciations are concatenated.

Dictionary entries can be filtered by a word list such that all entries not in the list are ignored. Note that the word identifiers in the word list should match exactly (e.g. same case) their corresponding entries in the dictionary.

The edit commands provided by HDMan are as follows

`AS A B ...` Append silence models A, B, etc to each pronunciation.

`CR X A Y B` Replace phone `Y` in the context of `A_B` by `X`. Contexts may include an asterix `*` to denote any phone or a defined context set defined using the `DC` command.

`DC X A B ...` Define the set `A`, `B`, ... as the context `X`.

`DD X A B ...` Delete the definition for word `X` starting with phones `A`, `B`, ...

`DP A B C ...` Delete any occurrences of phones `A` or `B` or `C` ....

`DS src` Delete each pronunciation from source `src` unless it is the only one for the current word.

`DW X Y Z ...` Delete words (& definitions) `X`, `Y`, `Z`, ...

`FW X Y Z ...` Define `X`, `Y`, `Z`, ... as function words and change each phone in the definition to a function word specific phone. For example, in word `W` phone `A` would become `W.A`.

`IR` Set the input mode to raw. In raw mode, words are regarded as arbitrary sequences of printing chars. In the default mode, words are strings as defined in section 4.6.

`LC [X]` Convert all phones to be left-context dependent. If `X` is given then the 1st phone `a` in each word is changed to `X-a` otherwise it is unchanged.

`LP` Convert all phones to lowercase.

`LW` Convert all words to lowercase.

`MP X A B ...` Merge any sequence of phones `A B ...` and rename as `X`.

`RC [X]` Convert all phones to be right-context dependent. If `X` is given then the last phone `z` in each word is changed to `z+X` otherwise it is unchanged.

`RP X A B ...` Replace all occurrences of phones `A` or `B ...` by `X`.

`RS system` Remove stress marking. Currently the only stress marking system supported is that used in the dictionaries produced by Carnegie Melon University (system = cmu).

`RW X A B ...` Replace all occurrences of word `A` or `B ...` by `X`.

`SP X A B ...` Split phone `X` into the sequence `A B C ...`.

`TC [X [Y]]` Convert phones to triphones. If `X` is given then the first phone `a` is converted to `X-a+b` otherwise it is unchanged. If `Y` is given then the last phone `z` is converted to `y-z+Y` otherwise if `X` is given then it is changed to `y-z+X` otherwise it is unchanged.

`UP` Convert all phones to uppercase.

`UW` Convert all words to uppercase.

17.5.2 Use

HDMan is invoked by typing the command line

```
HDMan [options] newDict srcDict1 srcDict2 ...
```

This causes HDMan read in the source dictionaries `srcDict1`, `srcDict2`, etc. and generate a new dictionary `newDict`. The available options are

`-a s` Each character in the string `s` denotes the start of a comment line. By default there is just one comment character defined which is `#`.

`-b s` Define `s` to be a word boundary symbol.

`-e dir` Look for edit scripts in the directory `dir`.

`-g f` File `f` holds the global edit script. By default, HDMan expects the global edit script to be called `global.ded`.

`-h i j` Skip the first `i` lines of the `j`'th listed source dictionary.

`-i` Include word output symbols in the output dictionary.

`-l s` Write a log file to `s`. The log file will include dictionary statistics and a list of the number of occurrences of each phone.

`-m` Merge pronunciations from all source dictionaries. By default, HDMan generates a single pronunciation for each word. If several input dictionaries have pronunciations for a word, then the first encountered is used. Setting this option causes all distinct pronunciations to be output for each word.

`-n f` Output a list of all distinct phones encountered to file `f`.

`-o` Disable dictionary output.

`-p f` Load the phone list stored in file `f`. This enables a check to be made that all output phones are in the supplied list. You need to create a log file (`-l`) to view the results of this check.

`-t` Tag output words with the name of the source dictionary which provided the pronunciation.

`-w f` Load the word list stored in file `f`. Only pronunciations for the words in this list will be extracted from the source dictionaries.

`-Q` Print a summary of all commands supported by this tool.

HDMan also supports the standard options `-A`, `-C`, `-D`, `-S`, `-T`, and `-V` as described in section 4.4.

17.5.3 Tracing

HDMan supports the following trace options where each trace flag is given using an octal base

`00001` basic progress reporting

`00002` word buffer operations

`00004` show valid inputs

`00010` word level editing

`00020` word level editing in detail

`00040` print edit scripts

`00100` new phone recording

`00200` pron deletions

`00400` word deletions

Trace flags are set using the `-T` option or the `TRACE` configuration variable.

17.6 HEAdapt

17.6.1 Function

This program is used to perform adaptation of a set of HMMs using either maximum likelihood linear regression (MLLR), maximum a-posteriori (MAP) or both. The default is MLLR. In order to perform the adaptation, the first stage requires a state/frame alignment. As such the initial operation of HEAdapt follows HERest closely. The adaptation training data consists of one or more utterances each of which has a transcription in the form of a standard label file (segment boundaries are ignored). For each training utterance, a composite model is effectively synthesised by concatenating the phoneme models given by the transcription. Each mixture component's accumulators in HEAdapt are updated simultaneously by performing a standard Baum-Welch pass over each training utterance using the composite model. HEAdapt will also prune the α and β matrices, just as HERest.

HEAdapt is intended to operate on HMMs which have been fully trained using HCompV, HInit, HRest, HERest. HEAdapt supports multiple mixture diagonal covariance Gaussian HMMs only (i.e. `PLAINHS` and `SHAREDHS` systems), with a single data stream only, and parameter tying within and between models. HEAdapt also supports tee-models (see section 7.8), for handling optional silence and non-speech sounds. These may be placed between the units (typically words or phones) listed in the transcriptions, but they cannot be used at the start or end of a transcription. Furthermore, chains of tee-models are not permitted.

After accumulating statistics, HEAdapt estimates the mean and (optionally) the variance transforms. HEAdapt will output either the adapted HMM set (as an MMF), or a transform model file (TMF). The TMF can then be applied to the original model set (for instance when using HVite). Note that with MAP adaptation a transform is not available and a full HMM set must be output.

When HEAdapt is being run to calculate multiple regression transforms, the model set being adapted must contain a regression class tree. The regression class tree is constructed using the `RC` edit command in HHEd.

17.6.2 Use

HEAdapt is invoked via the command line

```
HEAdapt [options] hmmList adaptFile ...
```

This causes the set of HMMs given in `hmmList` to be loaded. The given list of adaptation training files is then used to perform one adaptation cycle. As always, the list of training files can be stored in a script file if required. On completion, HEAdapt outputs new updated versions of each HMM definition or a new transform models file.

The detailed operation of HEAdapt is controlled by the following command line options

`-b N` Set the number of blocks to be used in the block diagonal matrix representation of the mean transformation. This option will override the config setting `HADAPT:BLOCKS`, or if this is not set the default number of blocks is 1.

`-c f` Set the minimum forward probability fixed distance for the alpha pruning to `f`. Restrict the computation of the α values to just those for which the total log likelihood $\alpha_j(t)\beta_j(t)$ is within distance `f` of the total likelihood (default 10.0).

`-d dir` Normally HEAdapt looks for HMM definitions (not already loaded via MMF files) in the current directory. This option tells HEAdapt to look in the directory `dir` to find them.

`-f field desc` Set the description field `field` in the transform model file to `desc`. Currently the choices for `field` are `uid`, `uname`, `chan` and `desc`.

`-g` Perform global adaptation only.

`-i N` Update the transforms (incrementally) after accumulating statistics every `N` utterances. The default operation is static adaptation, i.e. after seeing ALL the adaptation data.

`-j f` MAP adaptation with scaling factor `f`. The default operation is MLLR adaptation. If MAP adaptation is to be performed the default value of `f` is 15.0

`-k` Use MLLR to transform the HMM model set before performing MAP

`-m f` Set the minimum threshold occupation count for a regression class to `f`. A separate regression transformation will be generated at the lowest level in the tree for which there is sufficient occupancy (data). This option will override the config setting `HADAPT:OCCTHRESH`. The default setting is 700.0.

`-o ext` This causes the file name extensions of the original models (if any) to be replaced by `ext`.

`-t f [i l]` Set the pruning threshold to `f`. During the backward probability calculation, at each time t all (log) β values falling more than `f` below the maximum β value at that time are ignored. During the subsequent forward pass, (log) α values are only calculated if there are corresponding valid β values. Furthermore, if the ratio of the $\alpha\beta$ product divided by the total probability (as computed on the backward pass) falls below a fixed threshold then those values of α and β are ignored. Setting `f` to zero disables pruning (default value 0.0). Tight pruning thresholds can result in HEAdapt failing to process an utterance. if the `i` and `l` options are given, then a pruning error results in the threshold being increased by `i` and utterance processing restarts. If errors continue, this procedure will be repeated until the limit `l` is reached.

`-u flags` By default HEAdapt creates transforms for the means only. This option causes the parameters indicated by the `flags` to be created; this argument is a string containing one or more of the letters `m` (mean) and `v` (variance). The presence of a letter enables the creation of the corresponding part of the transform.

`-w f` This sets the minimum variance (i.e. diagonal element of the covariance matrix) to the real value `f` (default value 0.0).

`-x ext` By default, HEAdapt expects a HMM definition for the label X to be stored in a file called `X`. This option causes HEAdapt to look for the HMM definition in the file `X.ext`.

`-B` Output the HMM definition files in binary format. If outputting a tmf, then this option specifies binary output for the tmf.

`-F fmt` Set the source data format to `fmt`.

`-G fmt` Set the label file format to `fmt`.

`-H mmf` Load HMM macro model file `mmf`. This option may be repeated to load multiple MMFs.

`-I mlf` This loads the master label file `mlf`. This option may be repeated to load several MLFs.

`-J tmf` Load a transform set from the transform model file `tmf`. The `tmf` is used to transform the `mmf` before performing the state/frame alignment, and a transform is calculated based on this state/frame alignment and the `mmf`.

`-K tmf` Save the transform set in the transform model file `tmf`.

`-L dir` Search directory `dir` for label files (default is to search current directory).

`-M dir` Store output HMM macro model files in the directory `dir`. If this option is not given, the new HMM definition will overwrite the existing one.

`-X ext` Set label file extension to `ext` (default is `lab`).

HEAdapt also supports the standard options `-A`, `-C`, `-D`, `-S`, `-T`, and `-V` as described in section 4.4.

17.6.3 Tracing

HEAdapt supports the following trace options where each trace flag is given using an octal base

`00001` basic progress reporting.

`00002` show the logical/physical HMM map.

Trace flags are set using the `-T` option or the `TRACE` configuration variable.

The library that HEAdapt utilises, called HAdapt supports other useful trace options. For library modules, tracing has to be performed via the config file and the module name must prefix the trace (e,g `HADAPT:TRACE=0001`). The following are HAdapt trace options where each trace flag is given using an octal base

`00001` basic progress reporting.

`00002` trace on the accumulations.

`00004` trace on the transformations.

`00010` output the auxiliary function score.

`00020` regression classes input/output tracing.

`00040` regression class tree usage.

`00200` detailed trace for accumulations at the class level.

17.7 HERest

17.7.1 Function

This program is used to perform a single re-estimation of the parameters of a set of HMMs using an *embedded training* version of the Baum-Welch algorithm. Training data consists of one or more utterances each of which has a transcription in the form of a standard label file (segment boundaries are ignored). For each training utterance, a composite model is effectively synthesised by concatenating the phoneme models given by the transcription. Each phone model has the same set of accumulators allocated to it as are used in HRest but in HERest they are updated simultaneously by performing a standard Baum-Welch pass over each training utterance using the composite model.

HERest is intended to operate on HMMs with initial parameter values estimated by HInit/HRest. HERest supports multiple mixture Gaussians, discrete and tied-mixture HMMs, multiple data streams, parameter tying within and between models, and full or diagonal covariance matrices. HERest also supports tee-models (see section 7.8), for handling optional silence and non-speech sounds. These may be placed between the units (typically words or phones) listed in the transcriptions but they cannot be used at the start or end of a transcription. Furthermore, chains of tee-models are not permitted.

HERest includes features to allow parallel operation where a network of processors is available. When the training set is large, it can be split into separate chunks that are processed in parallel on multiple machines/processors, consequently speeding up the training process.

Like all re-estimation tools, HERest allows a floor to be set on each individual variance by defining a variance floor macro for each data stream (see chapter 8).

HERest supports two specific methods for initilisation of model parameters , *single pass retraining* and *2-model reestimation*.

Single pass retraining is useful when the parameterisation of the front-end (e.g. from MFCC to PLP coefficients) is to be modified. Given a set of well-trained models, a set of new models using a different parameterisation of the training data can be generated in a single pass. This is done by computing the forward and backward probabilities using the original well-trained models and the original training data, but then switching to a new set of training data to compute the new parameter estimates.

In *2-model re-estimation* one model set can be used to obtain the forward backward probablilites which then are used to update the parameters of another model set. Contrary to *single pass retraining* the two model sets are not required to be tied in the same fashion. This is particulary useful for training of single mixture models prior to decision-tree based state clustering. The use of 2-model re-estimation in HERest is triggered by setting the config variables `ALIGNMODELMMF` or `ALIGNMODELDIR` and `ALIGNMODELEXT` together with `ALIGNHMMLIST` (see section 8.7).

HERest operates in two distinct stages.

1. In the first stage, one of the following two options applies
   (a) Each input data file contains training data which is processed and the accumulators for state occupation, state transition, means and variances are updated.
   (b) Each data file contains a dump of the accumulators produced by previous runs of the program. These are read in and added together to form a single set of accumulators.
2. In the second stage, one of the following options applies
   (a) The accumulators are used to calculate new estimates for the HMM parameters.
   (b) The accumulators are dumped into a file.

Thus, on a single processor the default combination 1(a) and 2(a) would be used. However, if N processors are available then the training data would be split into N equal groups and HERest would be set to process one data set on each processor using the combination 1(a) and 2(b). When all processors had finished, the program would then be run again using the combination 1(b) and 2(a) to load in the partial accumulators created by the N processors and do the final parameter re-estimation. The choice of which combination of operations HERest will perform is governed by the `-p` option switch as described below.

As a further performance optimisation, HERest will also prune the α and β matrices. By this means, a factor of 3 to 5 speed improvement and a similar reduction in memory requirements can be achieved with negligible effects on training performance (see the `-t` option below).

17.7.2 Use

HERest is invoked via the command line

```
HERest [options] hmmList trainFile ...
```

This causes the set of HMMs given in `hmmList` to be loaded. The given list of training files is then used to perform one re-estimation cycle. As always, the list of training files can be stored in a script file if required. On completion, HERest outputs new updated versions of each HMM definition. If the number of training examples falls below a specified threshold for some particular HMM, then the new parameters for that HMM are ignored and the original parameters are used instead.

The detailed operation of HERest is controlled by the following command line options

`-c f` Set the threshold for tied-mixture observation pruning to `f`. For tied-mixture `TIEDHS` systems, only those mixture component probabilities which fall within `f` of the maximum mixture component probability are used in calculating the state output probabilities (default 10.0).

`-d dir` Normally HERest looks for HMM definitions (not already loaded via MMF files) in the current directory. This option tells HERest to look in the directory `dir` to find them.

`-m N` Set the minimum number of training examples required for any model to `N`. If the actual number falls below this value, the HMM is not updated and the original parameters are used for the new version (default value 3).

`-o ext` This causes the file name extensions of the original models (if any) to be replaced by `ext`.

`-p N` This switch is used to set parallel mode operation. If `p` is set to a positive integer `N`, then HERest will process the training files and then dump all the accumulators into a file called `HERN.acc`. If `p` is set to 0, then it treats all file names input on the command line as the names of `.acc` dump files. It reads them all in, adds together all the partial accumulations and then re-estimates all the HMM parameters in the normal way.

`-r` This enables single-pass retraining. The list of training files is processed pair-by-pair. For each pair, the first file should match the parameterisation of the original model set. The second file should match the parameterisation of the required new set. All speech input processing is controlled by configuration variables in the normal way except that the variables describing the old parameterisation are qualified by the name `HPARM1` and the variables describing the new parameterisation are qualified by the name `HPARM2`. The stream widths for the old and the new must be identical.

`-s file` This causes statistics on occupation of each state to be output to the named file. This file is needed for the `RO` command of HHEd but it is also generally useful for assessing the amount of training material available for each HMM state.

`-t f [i l]` Set the pruning threshold to `f`. During the backward probability calculation, at each time t all (log) β values falling more than `f` below the maximum β value at that time are ignored. During the subsequent forward pass, (log) α values are only calculated if there are corresponding valid β values. Furthermore, if the ratio of the $\alpha\beta$ product divided by the total probability (as computed on the backward pass) falls below a fixed threshold then those values of α and β are ignored. Setting `f` to zero disables pruning (default value 0.0). Tight pruning thresholds can result in HERest failing to process an utterance. if the `i` and `l` options are given, then a pruning error results in the threshold being increased by `i` and utterance processing restarts. If errors continue, this procedure will be repeated until the limit `l` is reached.

`-u flags` By default, HERest updates all of the HMM parameters, that is, means, variances, mixture weights and transition probabilies. This option causes just the parameters indicated by the `flags` argument to be updated, this argument is a string containing one or more of the letters `m` (mean), `v` (variance) , `t` (transition) and `w` (mixture weight). The presence of a letter enables the updating of the corresponding parameter set.

`-v f` This sets the minimum variance (i.e. diagonal element of the covariance matrix) to the real value `f` (default value 0.0).

`-w f` Any mixture weight which falls below the global constant `MINMIX` is treated as being zero. When this parameter is set, all mixture weights are floored to `f * MINMIX`.

`-x ext` By default, HERest expects a HMM definition for the label X to be stored in a file called `X`. This option causes HERest to look for the HMM definition in the file `X.ext`.

`-B` Output HMM definition files in binary format.

`-F fmt` Set the source data format to `fmt`.

`-G fmt` Set the label file format to `fmt`.

`-H mmf` Load HMM macro model file `mmf`. This option may be repeated to load multiple MMFs.

`-I mlf` This loads the master label file `mlf`. This option may be repeated to load several MLFs.

`-L dir` Search directory `dir` for label files (default is to search current directory).

`-M dir` Store output HMM macro model files in the directory `dir`. If this option is not given, the new HMM definition will overwrite the existing one.

`-X ext` Set label file extension to `ext` (default is `lab`).

HERest also supports the standard options `-A`, `-C`, `-D`, `-S`, `-T`, and `-V` as described in section 4.4.

17.7.3 Tracing

HERest supports the following trace options where each trace flag is given using an octal base

`00001` basic progress reporting.

`00002` show the logical/physical HMM map.

`00004` report statistics on pruning.

`00010` show the alpha/beta matrices.

`00020` show the occupation counters.

`00040` show the transition counters.

`00100` show the mixture weight counters.

`00200` show the calculation of the output probabilities.

`00400` list the updated model parameters.

`01000` show the average percentage utilisation of tied mixture components.

Trace flags are set using the `-T` option or the `TRACE` configuration variable.

17.8 HHEd

17.8.1 Function

HHEd is a script driven editor for manipulating sets of HMM definitions. Its basic operation is to load in a set of HMMs, apply a sequence of edit operations and then output the transformed set. HHEd is mainly used for applying tyings across selected HMM parameters. It also has facilities for cloning HMMs, clustering states and editing HMM structures.

Many HHEd commands operate on sets of similar items selected from the set of currently loaded HMMs. For example, it is possible to define a set of all final states of all vowel models, or all mean vectors of all mixture components within the model X, etc. Sets such as these are defined by item lists using the syntax rules given below. In all commands, all of the items in the set defined by an item list must be of the same type where the possible types are

s	– state	t	– transition matrix
p	– pdf	w	– stream weights
m	– mixture component	d	– duration parameters
u	– mean vector	x	– transform matrix
v	– variance vector	i	– inverse covariance matrix
h	– HMM definition		

Most of the above correspond directly to the tie points shown in Fig 7.8. There is just one exception. The type "p" corresponds to a pdf (ie a sum of Gaussian mixtures). Pdf's cannot be tied, however, they can be named in a Tie (`TI`) command (see below) in which case, the effect is to join all of the contained mixture components into one pool of mixtures and then all of the mixtures in the pool are shared across all pdf's. This allows conventional *tied-mixture* or *semi-continuous* HMM systems to be constructed.

The syntax rules for item lists are as follows. An item list consists of a comma separated list of item sets.

itemList = "{" itemSet { "," itemSet } "}"

Each itemSet consists of the name of one or more HMMs (or a pattern representing a set of HMMs) followed by a specification which represents a set of paths down the parameter hierarchy each terminating at one of the required parameter items.

itemSet = hmmName . ["transP" | "state" state]
hmmName = ident | identList
identList = "(" ident { "," ident } ")"
ident = < char | metachar >
metachar = "?" | "*"

A hmmName consists of a single ident or a comma separated list of ident's. The following examples are all valid hmmName's:

```
aa  three  model001  (aa,iy,ah,uh)  (one,two,three)
```

In addition, an ident can contain the metacharacter "?" which matches any single character and the metacharacter "*" which matches a string of zero or more characters. For example, the item list

```
{*-aa+*.transP}
```

would represent the set of transition matrices of all loaded triphone variations of `aa`.

Items within states require the state indices to be specified

state = index ["." stateComp]
index = "[" intRange { "," intRange } "]"
intRange = integer ["-" integer]

For example, the item list

```
{*.state[1,3-5,9]}
```

represents the set of all states 1, 3 to 5 inclusive and 9 of all currently loaded HMMs. Items within states include durational parameters, stream weights, pdf's and all items within mixtures

stateComp = "dur" | "weights" | [" stream" index] "." "mix" [mix]

For example,

```
{(aa,ah,ax).state[2].dur}
```

denotes the set of durational parameter vectors from state 2 of the HMMs `aa`, `ah` and `ax`. Similarly,

```
{*.state[2-4].weights}
```

denotes the set of stream weights for states 2 to 4 of all currently loaded HMMs. The specification of pdf's may optionally include a list of the relevant streams, if omitted, stream 1 is assumed. For example,

```
{three.state[3].mix}
```

and

```
{three.state[3].stream[1].mix}
```

both denote a list of the single pdf belonging to stream 1 of state 3 of the HMM `three`.

Within a pdf, the possible item types are mixture components, mean vectors, and the various possible forms of covariance parameters

mix = index ["." ("mean" | "cov")]

For example,

```
{*.state[2].mix[1-3]}
```

denotes the set of mixture components 1 to 3 from state 2 of all currently loaded HMMs and

```
{(one,two).state[4].stream[3].mix[1].mean}
```

denotes the set of mean vectors from mixture component 1, stream 3, state 4 of the HMMs `one` and `two`. When `cov` is specified, the type of the covariance item referred to is determined from the `CovKind` of the loaded models. Thus, for diagonal covariance models, the item list

```
{*.state[2-4].mix[1].cov}
```

would denote the set of variance vectors for mixture 1, states 2 to 4 of all loaded HMMs.

Note finally, that it is not an error to specify non-existent models, states, mixtures, etc. All item list specifications are regarded as patterns which are matched against the currently loaded set of models. All and only those items which match are included in the set. However, both a null result and a set of items of mixed type do result in errors.

All HHEd commands consist of a 2 character command name followed by zero or more arguments. In the following descriptions, item lists are shown as `itemList(c)` where the character c denotes the type of item expected by that command. If this type indicator is missing then the command works for all item types.

The HHEd commands are as follows

```
AT i j prob itemList(t)
```

Add a transition from state `i` to state `j` with probability `prob` for all transition matrices in `itemList`. The remaining transitions out of state `i` are rescaled so that $\sum_k a_{ik} = 1$. For example,

```
AT 1 3 0.1 {*.transP}
```

would add a skip transition to all loaded models from state 1 to state 3 with probability 0.1.

```
AU hmmList
```

Use a set of decision trees to create a new set of models specified by the `hmmList`. The decision trees may be made as a result of either the `TB` or `LT` command.

Each model in `hmmList` is constructed in the following manner. If a model with the same logical name already exists in the current HMM set this is used unchanged, otherwise the model is synthesised from the decision trees. If the trees cluster at the model level the synthesis results in a logical model sharing the physical model from the tree that matches the new context. If the clustering was performed at the state level a prototype model (an example of the same phone model occurring in a different context) is found and a new HMM is constructed that shares the transition matrix with the prototype model but consists of tied states selected using the decision tree.

`CL hmmList`

Clone a HMM list. The file `hmmList` should hold a list of HMMs all of whose logical names are either the same as, or are context-dependent versions of the currently loaded set of HMMs. For each name in `hmmList`, the corresponding HMM in the loaded set is cloned. On completion, the currently loaded set is discarded and replaced by the new set. For example, if the file `mylist` contained

```
A-A+A
A-A+B
B-A+A
B-B+B
B-B+A
```

and the currently loaded HMMs were just `A` and `B`, then `A` would be cloned 3 times to give the models `A-A+A`, `A-A+B` and `B-A+A`, and `B` would be cloned 2 times to give `B-B+B` and `B-B+A`. On completion, the original definitions for `A` and `B` would be deleted (they could be retained by including them in the new `hmmList`).

`CO newList`

Compact a set of HMMs. The effect of this command is to scan the currently loaded set of HMMs and identify all identical definitions. The physical name of the first model in each identical set is then assigned to all models in that set and all model definitions are replaced by a pointer to the first model definition. On completion, a new list of HMMs which includes the new model tyings is written out to file `newList`. For example, suppose that models `A`, `B`, `C` and `D` were currently loaded and `A` and `B` were identical. Then the command

```
CO tlist
```

would tie HMMs `A` and `B`, set the physical name of `B` to `A` and output the new HMM list

```
A
B A
C
D
```

to the file `tlist`. This command is used mainly after performing a sequence of parameter tying commands.

`DP s n id ...`

Duplicates a set of HMMs. This command is used to replicate a set of HMMs whilst allowing control over which structures will be shared between them. The first parameter controls duplication of tied structures. Any macros whose type appears in string `s` are duplicated with new names and only used in the duplicate model set. The remaining shared structures are common through all the model sets (original and duplicates). The second parameter defines the number of times the current HMM set should be duplicated with the remaining `n` parameters providing suffices to make the original macro identifiers unique in each duplicated HMM set.

For instance the following script could be used to duplicate a set of tied state models to produce gender dependent ones with tied variances.

```
MM "v_" { (*).state[2-4].mix[1-2].cov }
DP "v" 2 ":m" ":f"
```

The `MM` command converts all variances into macros (with each macro referring to only one variance). The `DP` command then duplicates the current HMM set twice. Each of the duplicate sets will share the tied variances with the original set but will have new mixture means, weights and state macros. The new macro names will be constructed by appending the id `":m"` or `":f"` to the original macro name whilst the model names have the id appended after the base phone name (so `ax-b+d` becomes `ax-b:m+d` or `ax-b:f+d`.

`FA varscale`

Computes an average within state variance vector for a given HMM set, using statistics generated by HERest (see `LS` for loading stats). The average variance vector is scaled and stored in the HMM set, any variance floor vectors present are replaced. Subsequently, the variance floor is applied to all variances in the model set. This can be inhibited by setting `APPLYVFLOOR` to `FALSE`.

`FC`

Converts all covariances in the modelset to full. This command takes an HMM set with diagonal covariances and creates full covariances which are initialised with the variances of the diagonal system. The tying structure of the original system is kept intact.

`FV file`

Loads one variance floor macro per stream from file. The file containing the variance floor macros can, for example, be generated by HCompV. Any variance floor vectors present in the model set are replaced. Secondly the variance floor is applied to all variances. This can be inhibited but setting `APPLYVFLOOR` to `FALSE`.

`HK hsetkind`

Converts model set from one kind to another. Although hsetkind can take the value PLAINHS, SHAREDHS, TIEDHS or DISCRETEHS, the HK command is most likely to be used when building tied-mixture systems (hsetkind=TIEDHS).

`JO size minw`

Set the size and minimum mixture weight for subsequent Tie (`TI`) commands applied to pdf's. The value of `size` sets the total number of mixtures in the tied mixture set (*codebook*) and `minw` sets a floor on the mixture weights as a multiple of `MINMIX`. This command only applies to tying item lists of type "p" (see the Tie `TI` command below).

`LS statsfile`

This command is used to read in the HERest statistics file (see the HERest `-s` option) stored in `statsfile`. These statistics are needed for certain clustering operations. The statistics file contains the occupation count for every HMM state.

`LT treesfile`

This command reads in the decision trees stored in `treesfile`. The trees file will consist of a set of questions defining contexts that may appear in the subsequent trees. The trees are used to identify either the state or the model that should be used in a particular context. The file would normally be produced by `ST` after tree based clustering has been performed.

`MD nmix itemlist`

Decrease the number of mixture components in each pdf in the `itemList` to `m`. This employs a stepwise greedy merging strategy. For a given set of mixture components the pair with minimal merging cost is found and merged. This is repeated until only `m` mixture components are left. Any defunct mixture components (i.e. components with a weight below `MINMIX`) are deleted prior to this process.

Note that after application of this command a pdf in `itemlist` may consist of fewer, but not more than `m` mixture components.

As an example, the command

```
MD 6 {*-aa+*.state[3].mix}
```

would decrease the number of mixture components in state 3 of all triphones of `aa` to 6.

`MM macro itemList`

This command makes each item (I=1..N) in `itemList` into a macro with name `nameI` and a usage of one. This command can prevent unnecessary duplication of structures when HMMs are cloned or duplicated.

`MT triList newTriList`

Make a set of triphones by merging the currently loaded set of biphones. This is a very specialised command. All currently loaded HMMs must have 3 emitting states and be either left or right context-dependent biphones. The list of HMMs stored in `triList` should contain one or more triphones. For each triphone in `triList` of the form `X-Y+Z`, there must be currently loaded biphones `X-Y` and `Y+Z`. A new triphone `X-Y+Z` is then synthesised by first cloning `Y+Z` and then replacing the state information for the initial emitting state by the state information for the initial emitting state of `X-Y`. Note that the underlying physical names of the biphones used to create the triphones are recorded so that where possible, triphones generated from tied biphones are also tied. On completion, the new list of triphones including aliases is written to the file `newTriList`.

`MU m itemList(p)`

Increase the number of non-defunct mixture components in each pdf in the `itemList` to `m` (when `m` is just a number) or by `m` (when `m` is a number preceeded by a `+` sign. A defunct mixture is one for which the weight has fallen below `MINMIX`. This command works in two steps. Firstly, the weight of each mixture in each pdf is checked. If any defunct mixtures are discovered, then each is successively replaced by a non-defunct mixture component until either the required total number of non-defunct mixtures is reached or there are no defunct mixtures left. This replacement works by first deleting the defunct mixture and then finding the mixture with the largest weight and splitting it. The split operation is as follows. The weight of the mixture component is first halved and then the mixture is cloned. The two identical mean vectors are then perturbed by adding 0.2 standard deviations to one and subtracting the same amount from the other.

In the second step, the mixture component with the largest weight is split as above. This is repeated until the required number of mixture components are obtained. Whenever, a mixture is split, a count is incremented for that mixture so that splitting occurs evenly across the mixtures. Furthermore, a mixture whose *gconst* value falls more than four standard deviations below the mean is not split.

As an example, the command

```
MU 6 {*-aa+*.state[3].mix}
```

would increase the number of mixture components in state 3 of all triphones of `aa` to 6.

`NC N macro itemList(s)`

N-cluster the states listed in the `itemList` and tie each cluster `i` as macro `macroi` where `i` is 1,2,3,...,`N`. The set of states in the `itemList` are divided into `N` clusters using the following furthest neighbour hierarchical cluster algorithm:

```
create 1 cluster for each state;
n = number of clusters;
while (n>N) {
   find i and j for which g(i,j) is minimum;
   merge clusters i and j;
}
```

Here `g(i,j)` is the inter-group distance between clusters `i` and `j` defined as the maximum distance between any state in cluster `i` and any state in cluster `j`. The calculation of the inter-state distance depends on the type of HMMs involved. Single mixture Gaussians use

$$d(i,j) = \frac{1}{S}\sum_{s=1}^{S}\left[\frac{1}{V_s}\sum_{k=1}^{V_s}\frac{(\boldsymbol{\mu}_{isk}-\boldsymbol{\mu}_{jsk})^2}{\boldsymbol{\sigma}_{isk}\boldsymbol{\sigma}_{jsk}}\right]^{\frac{1}{2}} \tag{17.1}$$

where V_s is the dimensionality of stream s. Fully tied mixture systems (ie `TIEDHS`) use

$$d(i,j) = \frac{1}{S}\sum_{s=1}^{S}\left[\frac{1}{M_s}\sum_{m=1}^{M_s}(c_{ism} - c_{jsm})^2\right]^{\frac{1}{2}} \tag{17.2}$$

and all others use

$$d(i,j) = -\frac{1}{S}\sum_{s=1}^{S}\frac{1}{M_s}\sum_{m=1}^{M_s}\log[b_{js}(\boldsymbol{\mu}_{ism})] + \log[b_{is}(\boldsymbol{\mu}_{jsm})] \tag{17.3}$$

where $b_{js}(\boldsymbol{x})$ is as defined in equation 7.1 for the continuous case and equation 7.3 for the discrete case. The actual tying of the states in each cluster is performed exactly as for the Tie (`TI`) command below. The macro for the `i`'th tied cluster is called `macroi`.

`QS name itemList(h)`

Define a question `name` which is true for all the models in `itemList`. These questions can subsequently be used as part of the decision tree based clustering procedure (see `TB` command below).

`RC N identifier [itemlist]`

This command is used to grow a regression class tree for adaptation purposes. A regression class tree is grown with `N` terminal or leaf nodes, using the centroid splitting algorithm with a Euclidean distance measure to cluster the model set's mixture components. Hence each leaf node specifies a particular mixture component cluster. The regression class tree is saved with the macro identifier `identifier_N`. Each Gaussian component is also labelled with a regression class number (corresponding to the leaf node number that the Gaussian component resides in). In order to grow the regression class tree it is necessary to load in a `statsfile` using the `LS` command. It is also possible to specify an `itemlist` containing the "non-speech" sound components such as the silence mixture components. If this is included then the first split made will result in one leaf containing the specified non-speech sound conmponents, while the other leaf will contain the rest of the model set components. Tree contruction then continues as usual.

`RN hmmIdName`

Rename or add the hmm set identifier in the global options macro to `hmmIdName`.

`RM hmmFile`

Load the hmm from `hmmFile` and subtract the mean from state 2, mixture 1 of the model from every loaded model. Every component of the mean is subtracted including deltas and accelerations.

`RO f [statsfile]`

This command is used to remove outlier states during clustering with subsequent `NC` or `TC` commands. If `statsfile` is present it first reads in the HERest statistics file (see `LS`) otherwise it expects a separate `LS` command to have already been used to read in the statistics. Any subsequent `NC`, `TC` or `TB` commands are extended to ensure that the occupancy clusters produced exceeds the threshold `f`. For `TB` this is used to choose which questions are allowed to be used to split each node. Whereas for `NC` and `TC` a final merging pass is used and for as long the smallest cluster count falls below the threshold `f`, then that cluster is merged with its nearest neighbour.

`RT i j itemList(t)`

Remove the transition from state `i` to `j` in all transition matrices given in the `itemList`. After removal, the remaining non-zero transition probabilities for state `i` are rescaled so that $\sum_k a_{ik} = 1$.

`SH`

Show the current HMM set. This command can be inserted into edit scripts for debugging. It prints a summary of each loaded HMM identifying any tied parameters.

`SK skind`

Change the sample kind of all loaded HMMs to `skind`. This command is typically used in conjunction with the `SW` command. For example, to add delta coefficients to a set of models, the `SW` command would be used to double the stream widths and then this command would be used to add the `_D` qualifier.

`SS N`

Split into N independent data streams. This command causes the currently loaded set of HMMs to be converted from 1 data stream to N independent data streams. The widths of each stream are determined from the single stream vector size and the sample kind as described in section 5.13. Execution of this command will cause any tyings associated with the split stream to be undone.

`ST filename`

Save the currently defined questions and trees to file `filename`. This allows subsequent construction of models using for new contexts using the `LT` and `AU` commands.

`SU N w1 w2 w3 .. wN`

Split into N independent data streams with stream widths as specified. This command is similar to the `SS` command except that the width of each stream is defined explicity by the user rather than using the built-in stream splitting rules. Execution of this command will cause any tyings associated with the split stream to be undone.

`SW s n`

Change the width of stream `s` of all currently loaded HMMs to `n`. Changing the width of stream involves changing the dimensions of all mean and variance vectors or covariance matrices. If `n` is greater than the current width of stream `s`, then mean vectors are extended with zeroes and variance vectors are extended with 1's. Covariance matrices are extended with zeroes everywhere except for the diagonal elements which are set to 1. This command preserves any tyings which may be in force.

`TB f macro itemList(s or h)`

Decision tree cluster all states in the given `itemList` and tie them as `macroi` where `i` is 1,2,3,.... This command performs a top down clustering of the states or models appearing in `itemlist`. This clustering starts by placing all items in a single root node and then choosing a question from the current set to split the node in such a way as to maximise the likelihood of a single diagonal covariance Gaussian at each of the child nodes generating the training data. This splitting continues until the increase in likelihood falls below threshold `f` or no questions are available which do not pass the outlier threshold test. This type of clustering is only implimented for single mixture, diagonal covariance untied models.

`TC f macro itemList(s)`

Cluster all states in the given `itemList` and tie them as `macroi` where `i` is 1,2,3,.... This command is identical to the `NC` command described above except that the number of clusters is varied such that the maximum within cluster distance is less than the value given by `f`.

`TI macro itemList`

Tie the items in `itemList` and assign them to the specified `macro` name. This command applies to any item type but all of the items in `itemList` must be of the same type. The detailed method of tying depends on the item type as follows:

state(s) the state with the largest total value of `gConst` in stream 1 (indicating broad variances) and the minimum number of defunct mixture weights (see `MU` command) is selected from the item list and all states are tied to this typical state.

transitions(t) all transition matrices in the item list are tied to the last in the list.

mixture(m) all mixture components in the item list are tied to the last in the list.

mean(u) the average vector of all the mean vectors in the item list is calculated and all the means are tied to this average vector.

variance(v) a vector is constructed for which each element is the maximum of the corresponding elements from the set of variance vectors to be tied. All of the variances are then tied to this maximum vector.

covariance(i) all covariance matrices in the item list are tied to the last in the list.

xform(x) all transform matrices in the item list are tied to the last in the list.

duration(d) all duration vectors in the item list are tied to the last in the list.

stream weights(w) all stream weight vectors in the item list are tied to the last in the list.

pdf(p) as noted earlier, pdf's are tied to create tied mixture sets rather than to create a shared pdf. The procedure for tying pdf's is as follows

1. All mixtures from all pdf's in the item list are collected together in order of mixture weight.
2. If the number of mixtures exceeds the join size J [see the Join (`JO`) command above], then all but the first J mixtures are discarded.
3. If the number of mixtures is less than J, then the mixture with the largest weight is repeatedly split until there are exactly J mixture components. The split procedure used is the same as for the MixUp (`MU`) command described above.
4. All pdf's in the item list are made to share all J mixture components. The weight for each mixture is set proportional to the log likelihood of the mean vector of that mixture with respect to the original pdf.
5. Finally, all mixture weights below the floor set by the Join command are raised to the floor value and all of the mixture weights are renormalised.

`TR n`

Change the level of detail for tracing and consists of a number of separate flags which can be added together. Values 0001, 0002, 0004, 0008 have the same meaning as the command line trace level but apply only to a single block of commands (a block consisting of a set of commands of the name). A value of 0010 can be used to show current memory usage.

`UT itemList`

Untie all items in `itemList`. For each item in the item list, if the usage counter for that item is greater than 1 then it is cloned, the original shared item is replaced by the cloned copy and the usage count of the shared item is reduced by 1. If the usage count is already 1, the associated macro is simply deleted and the usage count set to 0 to indicate an unshared item. Note that it is not possible to untie a pdf since these are not actually shared [see the Tie (`TI`) command above].

17.8.2 Use

HHEd is invoked by typing the command line

```
HHEd [options] edCmdFile hmmList
```

where `edCmdFile` is a text file containing a sequence of edit commands as described above and `hmmList` defines the set of HMMs to be edited (see HModel for the format of HMM list). If the models are to be kept in separate files rather than being stored in an MMF, the configuration variable `KEEPDISTINCT` should be set to true. The available options for HHEd are

`-d dir` This option tells HHEd to look in the directory `dir` to find the model definitions.

`-o ext` This causes the file name extensions of the original models (if any) to be replaced by `ext`.

`-w mmf` Save all the macros and model definitions in a single master macro file `mmf`.

`-x s` Set the extension for the edited output files to be `s` (default is to to use the original names unchanged).

`-z` Setting this option causes all aliases in the loaded HMM set to be deleted (zapped) immediately before loading the definitions. The result is that all logical names are ignored and the actual HMM list consists of just the physically distinct HMMs.

`-B` Output HMM definition files in binary format.

`-H mmf` Load HMM macro model file `mmf`. This option may be repeated to load multiple MMFs.

`-M dir` Store output HMM macro model files in the directory `dir`. If this option is not given, the new HMM definition will overwrite the existing one.

`-Q` Print a summary of all commands supported by this tool.

HHEd also supports the standard options `-A`, `-C`, `-D`, `-S`, `-T`, and `-V` as described in section 4.4.

17.8.3 Tracing

HHEd supports the following trace options where each trace flag is given using an octal base

`00001` basic progress reporting.

`00002` intermediate progress reporting.

`00004` detailed progress reporting.

`00010` show item lists used for each command.

`00020` show memory usage.

`00100` show changes to macro definitions.

`00200` show changes to stream widths.

`00400` show clusters.

`00800` show questions.

`01000` show tree filtering.

`02000` show tree splitting.

`04000` show tree merging.

`10000` show good question scores.

`20000` show all question scores.

`40000` show all merge scores.

Trace flags are set using the `-T` option or the `TRACE` configuration variable.

17.9 HInit

17.9.1 Function

HInit is used to provide initial estimates for the parameters of a single HMM using a set of observation sequences. It works by repeatedly using Viterbi alignment to segment the training observations and then recomputing the parameters by pooling the vectors in each segment. For mixture Gaussians, each vector in each segment is aligned with the component with the highest likelihood. Each cluster of vectors then determines the parameters of the associated mixture component. In the absence of an initial model, the process is started by performing a uniform segmentation of each training observation and for mixture Gaussians, the vectors in each uniform segment are clustered using a modified K-Means algorithm[5].

HInit can be used to provide initial estimates of whole word models in which case the observation sequences are realisations of the corresponding vocabulary word. Alternatively, HInit can be used to generate initial estimates of *seed* HMMs for phoneme-based speech recognition. In this latter case, the observation sequences will consist of segments of continuously spoken training material. HInit will *cut* these out of the training data automatically by simply giving it a segment label.

In both of the above applications, HInit normally takes as input a prototype HMM definition which defines the required HMM topology i.e. it has the form of the required HMM except that means, variances and mixture weights are ignored[6]. The transition matrix of the prototype specifies both the allowed transitions and their initial probabilities. Transitions which are assigned zero probability will remain zero and hence denote non-allowed transitions. HInit estimates transition probabilities by counting the number of times each state is visited during the alignment process.

HInit supports multiple mixtures, multiple streams, parameter tying within a single model, full or diagonal covariance matrices, tied-mixture models and discrete models. The output of HInit is typically input to HRest.

Like all re-estimation tools, HInit allows a floor to be set on each individual variance by defining a variance floor macro for each data stream (see chapter 8).

17.9.2 Use

HInit is invoked via the command line

```
HInit [options] hmm trainFiles ...
```

This causes the means and variances of the given `hmm` to be estimated repeatedly using the data in `trainFiles` until either a maximum iteration limit is reached or the estimation converges. The HMM definition can be contained within one or more macro files loaded via the standard `-H` option. Otherwise, the definition will be read from a file called `hmm`. The list of train files can be stored in a script file if required.

The detailed operation of HInit is controlled by the following command line options

`-e f` This sets the convergence factor to the real value `f`. The convergence factor is the relative change between successive values of $P_{max}(O|\lambda)$ computed as a by-product of the Viterbi alignment stage (default value 0.0001).

`-i N` This sets the maximum number of estimation cycles to `N` (default value 20).

`-l s` The string `s` must be the name of a segment label. When this option is used, HInit searches through all of the training files and cuts out all segments with the given label. When this option is not used, HInit assumes that each training file is a single token.

`-m N` This sets the minimum number of training examples so that if fewer than `N` examples are supplied an error is reported (default value 3).

`-n` This flag suppresses the initial uniform segmentation performed by HInit allowing it to be used to update the parameters of an existing model.

[5] This algorithm is significantly different from earlier versions of HTK where K-means clustering was used at every iteration and the Viterbi alignment was limited to states

[6] Prototypes should either have GConst set (the value does not matter) to avoid `HTK` trying to compute it or variances should be set to a positive value such as 1.0 to ensure that GConst is computable

`-o s` The string `s` is used as the name of the output HMM in place of the source name. This is provided in HInit since it is often used to initialise a model from a prototype input definition. The default is to use the source name.

`-u flags` By default, HInit updates all of the HMM parameters, that is, means, variances, mixture weights and transition probabilities. This option causes just the parameters indicated by the `flags` argument to be updated, this argument is a string containing one or more of the letters `m` (mean), `v` (variance), `t` (transition) and `w` (mixture weight). The presence of a letter enables the updating of the corresponding parameter set.

`-v f` This sets the minimum variance (i.e. diagonal element of the covariance matrix) to the real value `f` The default value is 0.0.

`-w f` Any mixture weight or discrete observation probability which falls below the global constant `MINMIX` is treated as being zero. When this parameter is set, all mixture weights are floored to `f * MINMIX`.

`-B` Output HMM definition files in binary format.

`-F fmt` Set the source data format to `fmt`.

`-G fmt` Set the label file format to `fmt`.

`-H mmf` Load HMM macro model file `mmf`. This option may be repeated to load multiple MMFs.

`-I mlf` This loads the master label file `mlf`. This option may be repeated to load several MLFs.

`-L dir` Search directory `dir` for label files (default is to search current directory).

`-M dir` Store output HMM macro model files in the directory `dir`. If this option is not given, the new HMM definition will overwrite the existing one.

`-X ext` Set label file extension to `ext` (default is `lab`).

HInit also supports the standard options `-A`, `-C`, `-D`, `-S`, `-T`, and `-V` as described in section 4.4.

17.9.3 Tracing

HInit supports the following trace options where each trace flag is given using an octal base

`000001` basic progress reporting.

`000002` file loading information.

`000004` segments within each file.

`000010` uniform segmentation.

`000020` Viterbi alignment.

`000040` state alignment.

`000100` mixture component alignment.

`000200` count updating.

`000400` output probabilities.

Trace flags are set using the `-T` option or the `TRACE` configuration variable.

17.10 HLEd

17.10.1 Function

This program is a simple editor for manipulating label files. Typical examples of its use might be to merge a sequence of labels into a single composite label or to expand a set of labels into a context sensitive set. HLEd works by reading in a list of *editing* commands from an edit script file and then makes an edited copy of one or more label files. For multiple level files, edit commands are applied to the *current level* which is initially the first (i.e. 1). Other levels may be edited by moving to the required level using the `ML` Move Level command.

Each edit command in the script file must be on a separate line. The first two-letter mnemonic on each line is the command name and the remaining letters denote labels[7]. The commands supported may be divided into two sets. Those in the first set are used to edit individual labels and they are as follows

`CH X A Y B` Change `Y` in the context of `A_B` to `X`. `A` and/or `B` may be a `*` to match any context, otherwise they must be defined by a `DC` command (see below). A block of consecutive `CH` commands are effectively executed in parallel so that the contexts are those that exist before any of the commands in the block are applied.

`DC A B C ..` define the context `A` as the set of labels `B`, `C`, etc.

`DE A B ..` Delete any occurrences of labels `A` or `B` etc.

`FI A Y B` Find `Y` in the context of `A_B` and count the number of occurrences.

`ME X A B ..` Merge any sequence of labels `A B C` etc. and call the new segment `X`.

`ML N` Move to label level `N`.

`RE X A B ..` Replace all occurrences of labels `A` or `B` etc. by the label `X`.

The commands in the second set perform global operations on whole transcriptions. They are as follows.

`DL [N]` Delete all labels in the current level. If the optional integer arg is given, then level `N` is deleted.

`EX` Expand all labels either from words to phones using the first pronunciation from a dictionary when it is specified on the command line otherwise expand labels of the form `A_B_C_D_...` into a sequence of separate labels `A B C D .....` This is useful for label formats which include a complete orthography as a single label or for creating a set of sub-word labels from a word orthography for a sub-word based recogniser. When a label is expanded in this way, the label duration is divided into equal length segments. This can only be performed on the root level of a multiple level file.

`FG X` Mark all unlabelled segments of the input file of duration greater than T_g msecs with the label `X`. The default value for T_g is 50000.0 (=5msecs) but this can be changed using the `-g` command line option. This command is mainly used for explicitly labelling inter-word silences in data files for which only the actual speech has been transcribed.

`IS A B` Insert label `A` at the start of every transcription and `B` at the end. This command is usually used to insert silence labels.

`IT` Ignore triphone contexts in `CH` and `FI` commands.

`LC [X]` Convert all phoneme labels to left context dependent. If `X` is given then the first phoneme label `a` becomes `X-a` otherwise it is left unchanged.

`NB X` The label `X` (typically a short pause) should be ignored at word boundaries when using the context commands `LC`, `RC` and `TC`.

[7] In earlier versions of HTK, HLEd command names consisted of a single letter. These are still supported for backwards compatibility and they are included in the command summary produced using the `-Q` option. However, commands introduced since version 2.0 have two letter names.

`RC [X]` Convert all phoneme labels to right context dependent. If `X` is given then the last phoneme label `z` becomes `z+X` otherwise it is left unchanged.

`SB X` Define the label `X` to be a sentence boundary marker. This label can then be used in context-sensitive change commands.

`SO` Sort all labels into time order.

`SP` Split multiple levels into multiple alternative label lists.

`TC [X[Y]]` Convert all phoneme labels to Triphones, that is left and right context dependent. If `X` is given then the first phoneme label `a` becomes `X-a+b` otherwise it is left unchanged. If `Y` is given then the last phoneme label `z` becomes `y-z+Y` otherwise if `X` is given then it becomes `y-z+X` otherwise it is left unchanged.

`WB X` Define `X` to be an inter-word label. This command affects the operation of the `LC`, `RC` and `TC` commands. The expansion of context labels is blocked wherever an inter-word label occurs.

The source and target label file formats can be defined using the `-G` and `-P` command line arguments. They can also be set using the configuration variables `SOURCELABEL` and `TARGETLABEL`. The default for both cases is the HTK format.

17.10.2 Use

HLEd is invoked by typing the command line

```
HLEd [options] edCmdFile labFiles ..
```

This causes HLEd to be applied to each `labFile` in turn using the edit commands listed in `edCmdFile`. The `labFiles` may be master label files. The available options are

`-b` Suppress label boundary times in output files.

`-d s` Read a dictionary from file `s` and use this for expanding labels when the `EX` command is used.

`-i mlf` This specifies that the output transcriptions are written to the master label file `mlf`.

`-g t` Set the minimum gap detected by the `FG` to be `t` (default 50000.0 = 5msecs). All gaps of shorter duration than `t` are ignored and not labelled.

`-l s` Directory to store output label files (default is current directory). When output is directed to an MLF, this option can be used to add a path to each output file name. In particular, setting the option `-l '*'` will cause a label file named `xxx` to be prefixed by the pattern `"*/xxx"` in the output MLF file. This is useful for generating MLFs which are independent of the location of the corresponding data files.

`-m` Strip all labels to monophones on loading.

`-n fn` This option causes a list of all new label names created to be output to the file `fn`.

`-G fmt` Set the label file format to `fmt`.

`-I mlf` This loads the master label file `mlf`. This option may be repeated to load several MLFs.

`-P fmt` Set the target label format to `fmt`.

`-X ext` Set label file extension to `ext` (default is `lab`).

HLEd also supports the standard options `-A`, `-C`, `-D`, `-S`, `-T`, and `-V` as described in section 4.4.

17.10.3 Tracing

HLED supports the following trace options where each trace flag is given using an octal base

`000001` basic progress reporting.

`000002` edit script details.

`000004` general command operation.

`000010` change operations.

`000020` level split/merge operations.

`000040` delete level operation.

`000100` edit file input.

`000200` memory usage.

`000400` dictionary expansion in `EX` command

Trace flags are set using the `-T` option or the `TRACE` configuration variable.

17.11 HList

17.11.1 Function

This program will list the contents of one or more data sources in any HTK supported format. It uses the full HTK speech input facilities described in chapter 5 and it can thus read data from a waveform file, from a parameter file and direct from an audio source. HLIST provides a dual rôle in HTK. Firstly, it is used for examining the contents of speech data files. For this function, the TARGETKIND configuration variable should not be set since no conversions of the data are required. Secondly, it is used for checking that input conversions are being performed properly. In the latter case, a configuration designed for a recognition system can be used with HLIST to make sure that the translation from the source data into the required observation structure is exactly as intended. To assist this, options are provided to split the input data into separate data streams (-n) and to explicitly list the identity of each parameter in an observation (-o).

17.11.2 Use

HList is invoked by typing the command line

```
HList [options] file ...
```

This causes the contents of each file to be listed to the standard output. If no files are given and the source format is HAUDIO, then the audio source is listed. The source form of the data can be converted and listed in a variety of target forms by appropriate settings of the configuration variables, in particular TARGETKIND[8].

The allowable options to HLIST are

-d Force each observation to be listed as discrete VQ symbols. For this to be possible the source must be either DISCRETE or have an associated VQ table specified via the VQTABLE configuration variable.

-e N End listing samples at sample index N.

-h Print the source header information.

-i N Print N items on each line.

-n N Display the data split into N independent data streams.

-o Show the observation structure. This identifies the rôle of each item in each sample vector.

-p Playback the audio. When sourcing from an audio device, this option enables the playback buffer so that after displaying the sampled data, the captured audio is replayed.

-r Print the raw data only. This is useful for exporting a file into a program which can only accept simple character format data.

-s N Start listing samples from sample index N. The first sample index is 0.

-t Print the target header information.

-F fmt Set the source data format to fmt.

HLIST also supports the standard options -A, -C, -D, -S, -T, and -V as described in section 4.4.

17.11.3 Tracing

HLIST supports the following trace options where each trace flag is given using an octal base

00001 basic progress reporting.

Trace flags are set using the -T option or the TRACE configuration variable.

[8] The TARGETKIND is equivalent to the HCOERCE environment variable used in earlier versions of HTK

17.12 HLMCopy

17.12.1 Function

The basic function of this tool is to copy language models. During this operation the target model can be optionally adjusted to a specific vocabulary, reduced in size by applying pruning parameters to the different n-gram components and written out in a different file format. Previously unseen words can be added to the language model with unigram entries supplied in a unigram probability file. At the same time, the tool can be used to extract word pronunciations from a number of source dictionaries and output a target dictionary for a specified word list. HLMCopy is a key utility enabling the user to construct custom dictionaries and language models tailored to a particular recognition task.

17.12.2 Use

HLMCopy is invoked by the command line

```
HLMCopy [options] inLMFile outLMFile
```

This copies the language model `inLMFile` to `outLMFile` optionally applying editing operations controlled by the following options.

`-c n c` Set the pruning threshold for n-grams to c. Pruning can be applied to the bigram and higher components of a model (n¿1). The pruning procedure will keep only n-grams which have been observed more than c times. Note that this option is only applicable to count-based language models.

`-d f` Use dictionary `f` as a source of pronunciations for the output dictionary. A set of dictionaries can be specified, in order of priority, with multiple `-d` options.

`-f s` Set the output language model format to `s`. Possible options are `TEXT` for the standard ARPA-MIT LM format, `BIN` for Entropic *binary* format and `ULTRA` for Entropic *ultra* format.

`-n n` Save target model as n-gram.

`-m` Allow multiple identical pronunciations for a single word. Normally identical pronunciations are deleted. This option may be required when a single word/pronunciation has several different output symbols.

`-o` Allow pronunciations for a single word to be selected from multiple dictionaries. Normally the dictionaries are prioritised by the order they appear on the command line with only entries in the first dictionary containing a pronunciation for a particular word being copied to the output dictionary.

`-u f` Use unigrams from file `f` as replacements for the ones in the language model itself. Any words appearing in the output language model which have entries in the unigram file (which is formatted as `LOG10PROB WORD`) use the likelihood (`log10(prob)`) from the unigram file rather than from the language model. This allows simple language model adaptation as well as allowing unigram probabilities to be assigned words in the output vocabulary that do not appear in the input language model. In some instances you may wish to use LNorm to renormalise the model after using `-u`.

`-v f` Write a dictionary covering the output vocabulary to file `f`. If any required words cannot be found in the set of input dictionaries an error will be generated.

`-w f` Read a word-list defining the output vocabulary from `f`. This will be used to select the vocabulary for both the output language model and output dictionary.

HLMCopy also supports the standard options `-A`, `-C`, `-D`, `-S`, `-T`, and `-V` as described in section 4.4.

17.12.3 Tracing

HLMCopy supports the following trace options where each trace flag is given using an octal base

`00001` basic progress reporting.

Trace flags are set using the `-T` option or the `TRACE` configuration variable.

17.13 HLRescore

17.13.1 Function

HLRescore is a general lattice post-processing tool. It reads lattices (for example produced by HVite) and applies one of the following operations on them:

- finding 1-best path through lattice
- pruning lattice using forward-bacward scores
- expanding lattices with new language model
- calculating various lattice statistics

A typical scenario for the use of HLRescore is the application of a higher order n-gram to the word lattices generated with HVite and a bigram. This would involve the following steps:

- lattice generation with HVite using a bigram
- lattice pruning with HLRescore (`-t`)
- expansion of lattices using a trigram (`-n`)
- finding 1-best transcription in the expanded lattice (`-f`)

Another common use of HLRescore is the tuning of the language model scaling factor and the word insertion penalty for use in recognition. Instead of having to re-run a decoder many times with different parameter settings the decoder is run once to generate lattices. HLRescore can be used to find the best transcription for a give parameter setting very quickly. These different transcriptions can then be scored (using HResults) and the parameter setting that yields the lowest word error rate can be selected.

17.13.2 Use

HLRescore is invoked via the command line

```
HLRescore [options] vocabFile LatFiles......
```

HLRescore reads each of the lattice files and performs te requested operation(s) on them. At least one of the following operations must be selected: find 1-best (`-f`), write lattices (`-w`), calculate statistics (`-c`).

The detailed operation of HLRescore is controlled by the following command line options

`-i mlf` Output transcriptions to master file `mlf`.

`-l s` Directory in which to store label/lattice files.

`-n lm` Load ARPA-format n-gram language model from file `lm` and expand lattice with this LM. All acoustic scores are unchanged but the LM scores are replaced and lattices nodes (i.e. contexts) are expanded as required by the structure of the LM.

`-o s` Choose how the output labels should be formatted. `s` is a string with certain letters (from `NSCTWM`) indicating binary flags that control formatting options. `N` normalise acoustic scores by dividing by the duration (in frames) of the segment. `S` remove scores from output label. By default scores will be set to the total likelihood of the segment. `C` Set the transcription labels to start and end on frame centres. By default start times are set to the start time of the frame and end times are set to the end time of the frame. `T` Do not include times in output label files. `W` Do not include words in output label files when performing state or model alignment. `M` Do not include model names in output label files.

`-t f [a]` Perform lattice pruning after reading lattices with beamwidth `f`. If second argument is given lower beam to limit arcs per second to `a`.

`-u f` Perform lattice pruning before writing output lattices. Otherwise like `-t`.

`-p f` Set the word insertion log probability to `f` (default 0.0).

`-a f` Set the acoustic model scale factor to `f`. (default value 1.0).

`-r f` Set the dictionary pronunciation probability scale factor to `f`. (default value 1.0).

`-s f` Set the grammar scale factor to `f`. This factor post-multiplies the language model likelihoods from the word lattices. (default value 1.0).

`-d` Take pronunciation probabilities from the dictionary instead of from the lattice.

`-c` Calculate and output lattice statistics.

`-f` Find 1-best transcription (path) in lattice.

`-w` Write output lattice after processing.

`-q s` Choose how the output lattice should be formatted. `s` is a string with certain letters (from `ABtvaldmn`) indicating binary flags that control formatting options. `A` attach word labels to arcs rather than nodes. `B` output lattices in binary for speed. `t` output node times. `v` output pronunciation information. `a` output acoustic likelihoods. `l` output language model likelihoods. `d` output word alignments (if available). `m` output within word alignment durations. `n` output within word alignment likelihoods.

`-y ext` This sets the extension for output label files to `ext` (default `rec`).

`-F fmt` Set the source data format to `fmt`.

`-G fmt` Set the label file format to `fmt`.

`-H mmf` Load HMM macro model file `mmf`. This option may be repeated to load multiple MMFs.

`-I mlf` This loads the master label file `mlf`. This option may be repeated to load several MLFs.

`-J tmf` Load a transform set from the transform model file `tmf`.

`-K tmf` Save the transform set in the transform model file `tmf`.

`-P fmt` Set the target label format to `fmt`.

HLRescore also supports the standard options `-A`, `-C`, `-D`, `-S`, `-T`, and `-V` as described in section 4.4.

17.13.3 Tracing

HLRescore supports the following trace options where each trace flag is given using an octal base

`0001` enable basic progress reporting.

`0002` output generated transcriptions.

`0004` show details of lattice I/O

`0010` show memory usage after each lattice

Trace flags are set using the `-T` option or the `TRACE` configuration variable.

17.14 HLStats

17.14.1 Function

This program will read in a HMM list and a set of HTK format transcriptions (label files). It will then compute various statistics which are intended to assist in analysing acoustic training data and generating simple language models for recognition. The specific functions provided by HLSTATS are:

1. number of occurrences of each distinct logical HMM and/or each distinct physical HMM. The list printed can be limited to the N most infrequent models.
2. minimum, maximum and average durations of each logical HMM in the transcriptions.
3. a matrix of bigram probabilities
4. an ARPA/MIT-LL format text file of back-off bigram probabilities
5. a list of labels which cover the given set of transcriptions.

17.14.2 Bigram Generation

When using the bigram generating options, each transcription is assumed to have a unique entry and exit label which by default are `!ENTER` and `!EXIT`. If these labels are not present they are inserted. In addition, any label occurring in a transcription which is not listed in the HMM list is mapped to a unique label called `!NULL`.

HLSTATS processes all input transcriptions and maps all labels to a set of unique integers in the range 1 to L, where L is the number of distinct labels. For each adjacent pair of labels i and j, it counts the total number of occurrences $N(i,j)$. Let the total number of occurrences of label i be $N(i) = \sum_{j=1}^{L} N(i,j)$.

For matrix bigrams, the bigram probability $p(i,j)$ is given by

$$p(i,j) = \begin{cases} \alpha N(i,j)/N(i) & \text{if } N(i) > 0 \\ 1/L & \text{if } N(i) = 0 \\ f & \text{otherwise} \end{cases}$$

where f is a floor probability set by the `-f` option and α is chosen to ensure that $\sum_{j=1}^{L} p(i,j) = 1$.

For back-off bigrams, the unigram probablities $p(i)$ are given by

$$p(i) = \begin{cases} N(i)/N & \text{if } N(i) > u \\ u/N & \text{otherwise} \end{cases}$$

where u is unigram floor count set by the `-u` option and $N = \sum_{i=1}^{L} \max[N(i),\mathrm{u}]$.

The backed-off bigram probabilities are given by

$$p(i,j) = \begin{cases} (N(i,j) - D)/N(i) & \text{if } N(i,j) > t \\ b(i)p(j) & \text{otherwise} \end{cases}$$

where D is a discount and t is a bigram count threshold set by the `-t` option. The discount D is fixed at 0.5 but can be changed via the configuration variable `DISCOUNT`. The back-off weight $b(i)$ is calculated to ensure that $\sum_{j=1}^{L} p(i,j) = 1$, i.e.

$$b(i) = \frac{1 - \sum_{j \in B} p(i,j)}{1 - \sum_{j \in B} p(j)}$$

where B is the set of all words for which $p(i,j)$ has a bigram.

The formats of matrix and ARPA/MIT-LL format bigram files are described in Chapter 12.

17.14.3 Use

HLSTATS is invoked by the command line

```
HLStats [options] hmmList labFiles ....
```

The `hmmList` should contain a list of all the labels (ie model names) used in the following label files for which statistics are required. Any labels not appearing in the list are ignored and assumed to be out-of-vocabulary. The list of labels is specified in the same way as for a HMM list (see HMODEL) and the logical⇒ physical mapping is preserved to allow statistics to be collected about physical names as well as logical ones. The `labFiles` may be master label files. The available options are

`-b fn` Compute bigram statistics and store result in the file `fn`.

`-c N` Count the number of occurrences of each logical model listed in the `hmmList` and on completion list all models for which there are `N` or less occurrences.

`-d` Compute minimum, maximum and average duration statistics for each label.

`-f f` Set the matrix bigram floor probability to `f` (default value 0.0). This option should be used in conjunction with the `-b` option.

`-h N` Set the bigram hashtable size to medium(N=1) or large (N=2). This option should be used in conjunction with the `-b` option. The default is small(N=0).

`-l fn` Output a list of covering labels to file `fn`. Only labels occurring in the `labList` are counted (others are assumed to be out-of-vocabulary). However, this list may contain labels that do not occur in any of the label files. The list of labels written to `fn` will however contain only those labels which occur at least once.

`-o` Produce backed-off bigrams rather than matrix ones. These are output in the standard ARPA/MIT-LL textual format.

`-p N` Count the number of occurrences of each physical model listed in the `hmmList` and on completion list all models for which there are `N` or less occurrences.

`-s st en` Set the sentence start and end labels to `st` and `en`. (Default `!ENTER` and `!EXIT`).

`-t n` Set the threshold count for including a bigram in a backed-off bigram language model. This option should be used in conjunction with the `-b` and `-o` options.

`-u f` Set the unigram floor probability to `f` when constructing a back-off bigram language model. This option should be used in conjunction with the `-b` and `-o` options.

`-G fmt` Set the label file format to `fmt`.

`-I mlf` This loads the master label file `mlf`. This option may be repeated to load several MLFs.

HLSTATS also supports the standard options `-A`, `-C`, `-D`, `-S`, `-T`, and `-V` as described in section 4.4.

17.14.4 Tracing

HLSTATS supports the following trace options where each trace flag is given using an octal base

`00001` basic progress reporting.

`00002` trace memory usage.

`00004` show basic statistics whilst calculating bigrams. This includes the global training data entropy and the entropy for each distinct label.

`00010` show file names as they are processed.

Trace flags are set using the `-T` option or the `TRACE` configuration variable.

17.15 HParse

17.15.1 Function

The HParse program generates word level lattice files (for use with e.g. HVite) from a text file syntax description containing a set of rewrite rules based on extended Backus-Naur Form (EBNF). The EBNF rules are used to generate an internal representation of the corresponding finite-state network where HParse network nodes represent the words in the network, and are connected via sets of links. This HParse network is then converted to HTK V2 word level lattice. The program provides one convenient way of defining such word level lattices.

HParse also provides a *compatibility mode* for use with HParse syntax descriptions used in HTK V1.5 where the same format was used to define both the word level syntax and the dictionary. In compatibility mode HParse will output the word level portion of such a syntax as an HTK V2 lattice file (via HNet) and the pronuciation information as an HTK V2 dictionary file (via HDict).

The lattice produced by HParse will often contain a number of `!NULL` nodes in order to reduce the number of arcs in the lattice. The use of such `!NULL` nodes can both reduce size and increase efficiency when used by recognition programs such as HVite.

17.15.2 Network Definition

The syntax rules for the textual definition of the network are as follows. Each node in the network has a `nodename`. This node name will normally correspond to a word in the final syntax network. Additionally, for use in compatibility mode, each node can also have an external name.

```
name      =  char{char}
nodename  =  name [ "%" ( "%" | name ) ]
```

Here `char` represents any character except one of the meta chars { } [ ] < >| = $ ( ) ; \ / *. The latter may, however, be escaped using a backslash. The first name in a `nodename` represents the name of the node ("internal name"), and the second optional name is the "external" name. This is used only in compatibility mode, and is, by default the same as the internal name.

Network definitions may also contain variables

```
variable  =  $name
```

Variables are identified by a leading $ character. They stand for sub-networks and must be defined before they appear in the RHS of a rule using the form

```
subnet    =  variable "=" expr ";"
```

An `expr` consists of a set of alternative sequences representing parallel branches of the network.

```
expr      =  sequence {"|" sequence}
sequence  =  factor{factor}
```

Each sequence is composed of a sequence of factors where a factor is either a node name, a variable representing some sub-network or an expression contained within various sorts of brackets.

```
factor    =  "(" expr ")"        |
             "{" expr "}"        |
             "<" expr ">"        |
             "[" expr "]"        |
             "<<" expr ">>"      |
             nodename            |
             variable
```

Ordinary parentheses are used to bracket sub-expressions, curly braces { } denote zero or more repetitions and angle brackets <> denote one or more repetitions. Square brackets [ ] are used to enclose optional items. The double angle brackets are a special feature included for building context dependent loops and are explained further below. Finally, the complete network is defined by a list of sub-network definitions followed by a single expression within parentheses.

```
network   =  {subnet} "(" expr ")"
```

Note that C style comments may be placed anywhere in the text of the network definition.

As an example, the following network defines a syntax for some simple edit commands

```
$dir   = up | down | left | right;
$mvcmd = move $dir | top | bottom;
$item  = char | word | line | page;
$dlcmd = delete [$item];   /* default is char */
$incmd = insert;
$encmd = end [insert];
$cmd = $mvcmd|$dlcmd|$incmd|$encmd;
({sil} < $cmd {sil} > quit)
```

Double angle brackets are used to construct contextually consistent context-dependent loops such as a word-pair grammar.[9] This function can also be used to generate consistent triphone loops for phone recognition[10]. The entry and exit conditions to a context-dependent loop can be controlled by the invisible pseudo-words TLOOP_BEGIN and TLOOP_END. The right context of TLOOP_BEGIN defines the legal loop start nodes, and the left context of TLOOP_END defines the legal loop finishers. If TLOOP_BEGIN/TLOOP_END are not present then all models are connected to the entry/exit of the loop.

A word-pair grammar simply defines the legal set of words that can follow each word in the vocabulary. To generate a network to represent such a grammar a right context-dependent loop could be used. The legal sentence set of sentence start and end words are defined as above using TLOOP_BEGIN/TLOOP_END.

For example, the following lists the legal followers for each word in a 7 word vocabulary

ENTRY	-	show, tell, give
show	-	me, all
tell	-	me, all
me	-	all
all	-	names, addresses
names	-	and, names, addresses, show, tell, EXIT
addresses	-	and, names, addresses, show, tell, EXIT
and	-	names, addresses, show, tell

HParse can generate a suitable lattice to represent this word-pair grammar by using the following specification:

```
$TLOOP_BEGIN_FLLWRS = show|tell|give;
$TLOOP_END_PREDS    = names|addresses;
$show_FLLWRS        = me|all;
$tell_FLLWRS        = me|all;
$me_FLLWRS          = all;
$all_FLLWRS         = names|addresses;
$names_FLLWRS       = and|names|addresses|show|tell|TLOOP_END;
$addresses_FLLWRS   = and|names|addresses|show|tell|TLOOP_END;
$and_FLLWRS         = names|addresses|show|tell;

( sil <<
      TLOOP_BEGIN+TLOOP_BEGIN_FLLWRS |
      TLOOP_END_PREDS-TLOOP_END |
      show+show_FLLWRS |
      tell+tell_FLLWRS |
      me+me_FLLWRS |
      all+all_FLLWRS |
      names+names_FLLWRS |
      addresses+addresses_FLLWRS |
      and+and_FLLWRS
  >> sil )
```

[9] The expression between double angle brackets must be a simple list of alternative node names or a variable which has such a list as its value

[10] In HTK V2 it is preferable for these context-loop expansions to be done automatically via HNet, to avoid requiring a dictionary entry for every context-dependent model

where it is assumed that each utterance begins and ends with `sil` model.

In this example, each set of contexts is defined by creating a variable whose alternatives are the individual contexts. The actual context-dependent loop is indicated by the `<< >>` brackets. Each element in this loop is a single variable name of the form `A-B+C` where `A` represents the left context, `C` represents the right context and `B` is the actual word. Each of `A`, `B` and `C` can be nodenames or variable names but note that this is the only case where variable names are expanded automatically and the usual `$` symbol is not used[11]. Both `A` and `C` are optional, and left and right contexts can be mixed in the same triphone loop.

17.15.3 Compatibility Mode

In HParse compatibility mode, the interpretation of the ENBF network is that used by the HTK V1.5 HVite program. in which HParse ENBF notation was used to define both the word level syntax and the dictionary. Compatibility mode is aimed at converting files written for HTK V1.5 into their equivalent HTK V2 representation. Therefore HParse will output the word level portion of such a ENBF syntax as an HTK V2 lattice file and the pronunciation information is optionally stored in an HTK V2 dictionary file. When operating in compatibility mode and not generating dictionary output, the pronunciation information is discarded.

In compatibility mode, the reserved node names `WD_BEGIN` and `WD_END` are used to delimit word boundaries—nodes between a `WD_BEGIN/WD_END` pair are called "word-internal" while all other nodes are "word-external". All `WD_BEGIN/WD_END` nodes must have an "external name" attached that denotes the word. It is a requirement that the number of `WD_BEGIN` and the number of `WD_END` nodes are equal and furthermore that there isn't a direct connection from a `WD_BEGIN` node to a `WD_END`. For example a portion of such an HTK V1.5 network could be

```
$A        =  WD_BEGIN%A ax WD_END%A;
$ABDOMEN  =  WD_BEGIN%ABDOMEN ae b d ax m ax n WD_END%ABDOMEN;
$ABIDES   =  WD_BEGIN%ABIDES ax b ay d z WD_END%ABIDES;
$ABOLISH  =  WD_BEGIN%ABOLISH ax b aa l ih sh WD_END%ABOLISH;
 ... etc


( <
   $A | $ABDOMEN | $ABIDES | $ABOLISH | ... etc
> )
```

HParse will output the connectivity of the words in an HTK V2 word lattice format file and the pronunciation information in an HTK V2 dictionary. Word-external nodes are treated as words and stored in the lattice with corresponding entries in the dictionary.

It should be noted that in HTK V1.5 any ENBF network could appear between a `WD_BEGIN/WD_END` pair, which includes loops. Care should therefore be taken with syntaxes that define very complex sets of alternative pronunciations. It should also be noted that each dictionary entry is limited in length to 100 phones. If multiple instances of the same word are found in the expanded HParse network, a dictionary entry will be created for only the first instance and subsequent instances are ignored (a warning is printed). If words with a NULL external name are present then the dictionary will contain a NULL output symbol.

Finally, since the implementation of the generation of the HParse network has been revised[12] the semantics of variable definition and use has been slightly changed. Previously variables could be redefined during network definition and each use would follow the most recent definition. In HTK V2 only the final definition of any variable is used in network expansion.

17.15.4 Use

HParse is invoked via the command line

```
HParse [options] syntaxFile latFile
```

[11] If the base-names or left/right context of the context-dependent names in a context-dependent loop are variables, no `$` symbols are used when writing the context-dependent nodename.

[12] With the added benefit of rectifying some residual bugs in the HTK V1.5 implementation

HParse will then read the set of ENBF rules in `syntaxFile` and produce the output lattice in `latFile`.

The detailed operation of HParse is controlled by the following command line options

`-b` Output the lattice in binary format. This increases speed of subsequent loading (default ASCII text lattices).

`-c` Set V1.5 compatibility mode. Compatibility mode can also be set by using the configuration variable V1COMPAT (default compatibility mode disabled).

`-d s` Output dictionary to file `s`. This is only a valid option when operating in compatibility mode. If not set no dictionary will be produced.

`-l` Include language model log probabilities in the output These log probabilities are calculated as $-\log$(number of followers) for each network node.

HParse also supports the standard options `-A`, `-C`, `-D`, `-S`, `-T`, and `-V` as described in section 4.4.

17.15.5 Tracing

HParse supports the following trace options where each trace flag is given using an octal base

`0001` basic progress reporting.

`0002` show final HParse network (before conversion to a lattice)

`0004` print memory statistics after HParse lattice generation

`0010` show progress through glue node removal.

Trace flags are set using the `-T` option or the `TRACE` configuration variable.

17.16 HQuant

17.16.1 Function

This program will construct a HTK format VQ table consisting of one or more codebooks, each corresponding to an independent data stream. A codebook is a collection of indexed reference vectors that are intended to represent the structure of the training data. Ideally, every compact cluster of points in the training data is represented by one reference vector. VQ tables are used directly when building systems based on Discrete Probability HMMs. In this case, the continuous-valued speech vectors in each stream are replaced by the index of the closest reference vector in each corresponding codebook. Codebooks can also be used to attach VQ indices to continuous observations. A typical use of this is to preselect Gaussians in probability computations. More information on the use of VQ tables is given in section 5.14.

Codebook construction consists of finding clusters in the training data, assigning a unique reference vector (the cluster centroid) to each, and storing the resultant reference vectors and associated codebook data in a VQ table. HQuant uses a top-down clustering process whereby clusters are iteratively split until the desired number of clusters are found. The optimal codebook size (number of reference vectors) depends on the structure and amount of the training data, but a value of 256 is commonly used.

HQuant can construct both linear (i.e. flat) and tree-structured (i.e. binary) codebooks. Linear codebooks can have lower quantisation errors but tree-structured codebooks have $\log_2 N$ access times compared to N for the linear case. The distance metric can either be Euclidean, diagonal covariance Mahalanobis or full covariance Mahalanobis.

17.16.2 VQ Codebook Format

Externally, a VQ table is stored in a text file consisting of a header followed by a sequence of entries representing each tree node. One tree is built for each stream and linear codebooks are represented by a tree in which there are only right branches.

The header consists of a magic number followed by the covariance kind, the number of following nodes, the number of streams and the width of each stream.

header = magic type covkind numNodes numS swidth1 swidth2 ...

where magic is a magic number which is usually the code for the parameter kind of the data. The type defines the type of codebook

type = linear (0) , binary tree-structured (1)

The covariance kind determines the type of distance metric to be used

covkind = diagonal covariance (1), full covariance (2), euclidean (5)

Within the file, these covariances are stored in inverse form.

Each node entry has the following form

node-entry = stream vqidx nodeId leftId rightId
mean-vector
[inverse-covariance-matrix | inverse-variance-vector]

Stream is the stream index for this entry. Vqidx is the VQ index corresponding to this entry. This is the number that appears in vector quantised speech files. In tree-structured code-books, it is zero for non-terminal nodes. Every node has a unique integer identifier (distinct from the VQ index) given by nodeId. The left and right daughter of the current node are given by leftId and rightId. In a linear codebook, the left identifier is always zero.

Some examples of VQ tables are given in Chapter 11.

17.16.3 Use

HQuant is invoked via the command line

```
HQuant [options] vqFile trainFiles ...
```

where `vqFile` is the name of the output VQ table file. The effect of this command is to read in training data from each `trainFile`, cluster the data and write the final cluster centres into the VQ table file.

The list of training files can be stored in a script file if required. Furthermore, the data used for training the codebook can be limited to that corresponding to a specified label. This can be used, for example, to train phone specific codebooks. When constructing a linear codebook, the maximum number of iterations per cluster can be limited by setting the configuration variable `MAXCLUSTITER`. The minimum number of samples in any one cluster can be set using the configuration variable `MINCLUSTSIZE`.

The detailed operation of HQuant is controlled by the following command line options

`-d` Use a diagonal-covariance Mahalonobis distance metric for clustering (default is to use a Euclidean distance metric).

`-f` Use a full-covariance Mahalonobis distance metric for clustering (default is to use a Euclidean distance metric).

`-g` Output the global covariance to a codebook. Normally, covariances are computed individually for each cluster using the data in that cluster. This option computes a global covariance across all the clusters.

`-l s` The string `s` must be the name of a segment label. When this option is used, HQuant searches through all of the training files and uses only the speech frames from segments with the given label. When this option is not used, HQuant uses all of the data in each training file.

`-n S N` Set size of codebook for stream `S` to `N` (default 256). If tree-structured codebooks are required then `N` must be a power of 2.

`-s N` Set number of streams to `N` (default 1). Unless the `-w` option is used, the width of each stream is set automatically depending on the size and parameter kind of the training data.

`-t` Create tree-structured codebooks (default linear).

`-w S N` Set width of stream `S` to `N`. This option overrides the default decomposition that HTK normally uses to divide a parameter file into streams. If this option is used, it must be repeated for each individual stream.

`-F fmt` Set the source data format to `fmt`.

`-G fmt` Set the label file format to `fmt`.

`-I mlf` This loads the master label file `mlf`. This option may be repeated to load several MLFs.

`-L dir` Search directory `dir` for label files (default is to search current directory).

`-X ext` Set label file extension to `ext` (default is `lab`).

HQuant also supports the standard options `-A`, `-C`, `-D`, `-S`, `-T`, and `-V` as described in section 4.4.

17.16.4 Tracing

HQuant supports the following trace options where each trace flag is given using an octal base

`00001` basic progress reporting.

`00002` dump global mean and covariance

`00004` trace data loading.

`00010` list label segments.

`00020` dump clusters.

`00040` dump VQ table.

Trace flags are set using the `-T` option or the `TRACE` configuration variable.

17.17 HRest

17.17.1 Function

HRest performs basic Baum-Welch re-estimation of the parameters of a single HMM using a set of observation sequences. HRest can be used for normal isolated word training in which the observation sequences are realisations of the corresponding vocabulary word.

Alternatively, HRest can be used to generate *seed* HMMs for phoneme-based recognition. In this latter case, the observation sequences will consist of segments of continuously spoken training material. HRest will *cut* these out of the training data automatically by simply giving it a segment label.

In both of the above applications, HRest is intended to operate on HMMs with initial parameter values estimated by HInit.

HRest supports multiple mixture components, multiple streams, parameter tying within a single model, full or diagonal covariance matrices, tied-mixture models and discrete models. The outputs of HRest are often further processed by HERest.

Like all re-estimation tools, HRest allows a floor to be set on each individual variance by defining a variance floor macro for each data stream (see chapter 8). If any diagonal covariance component falls below 0.00001, then the corresponding mixture weight is set to zero. A warning is issued if the number of mixtures is greater than one, otherwise an error occurs. Applying a variance floor via the `-v` option or a variance floor macro can be used to prevent this.

17.17.2 Use

HRest is invoked via the command line

```
HRest [options] hmm trainFiles ...
```

This causes the parameters of the given `hmm` to be re-estimated repeatedly using the data in `trainFiles` until either a maximum iteration limit is reached or the re-estimation converges. The HMM definition can be contained within one or more macro files loaded via the standard `-H` option. Otherwise, the definition will be read from a file called `hmm`. The list of train files can be stored in a script file if required.

The detailed operation of HRest is controlled by the following command line options

`-c f` Set the threshold for tied-mixture observation pruning to `f`. When all mixtures of all models are tied to create a full tied-mixture system, the calculation of output probabilities is treated as a special case. Only those mixture component probabilities which fall within `f` of the maximum mixture component probability are used in calculating the state output probabilities (default 10.0).

`-e f` This sets the convergence factor to the real value `f`. The convergence factor is the relative change between successive values of $P(O|\lambda)$ (default value 0.0001).

`-i N` This sets the maximum number of re-estimation cycles to `N` (default value 20).

`-l s` The string `s` must be the name of a segment label. When this option is used, HRest searches through all of the training files and cuts out all segments with the given label. When this option is not used, HRest assumes that each training file is a single token.

`-m N` Sets the minimum number of training examples to be `N`. If fewer than `N` examples are supplied then an error is reported (default value 3).

`-t` Normally, training sequences are rejected if they have fewer frames than the number of emitting states in the HMM. Setting this switch disables this reject mechanism[13].

`-u flags` By default, HRest updates all of the HMM parameters, that is, means, variances, mixture weights and transition probabilities. This option causes just the parameters indicated by the `flags` argument to be updated, this argument is a string containing one or more of the letters `m` (mean), `v` (variance), `t` (transition) and `w` (mixture weight). The presence of a letter enables the updating of the corresponding parameter set.

[13] Using this option only makes sense if the HMM has skip transitions

`-v f` This sets the minimum variance (i.e. diagonal element of the covariance matrix) to the real value `f`. This is ignored if an explicit variance floor macro is defined. The default value is 0.0.

`-w f` Any mixture weight or discrete observation probability which falls below the global constant `MINMIX` is treated as being zero. When this parameter is set, all mixture weights are floored to `f * MINMIX`.

`-B` Output HMM definition files in binary format.

`-F fmt` Set the source data format to `fmt`.

`-G fmt` Set the label file format to `fmt`.

`-H mmf` Load HMM macro model file `mmf`. This option may be repeated to load multiple MMFs.

`-I mlf` This loads the master label file `mlf`. This option may be repeated to load several MLFs.

`-L dir` Search directory `dir` for label files (default is to search current directory).

`-M dir` Store output HMM macro model files in the directory `dir`. If this option is not given, the new HMM definition will overwrite the existing one.

`-X ext` Set label file extension to `ext` (default is `lab`).

HRest also supports the standard options `-A`, `-C`, `-D`, `-S`, `-T`, and `-V` as described in section 4.4.

17.17.3 Tracing

HRest supports the following trace options where each trace flag is given using an octal base

`000001` basic progress reporting.

`000002` output information on the training data loaded.

`000004` the observation probabilities.

`000010` the alpha matrices.

`000020` the beta matrices.

`000040` the occupation counters.

`000100` the transition counters.

`000200` the mean counters.

`000400` the variance counters.

`001000` the mixture weight counters.

`002000` the re-estimated transition matrix.

`004000` the re-estimated mixture weights.

`010000` the re-estimated means.

`020000` the re-estimated variances.

Trace flags are set using the `-T` option or the `TRACE` configuration variable.

17.18 HResults

17.18.1 Function

HResults is the HTK performance analysis tool. It reads in a set of label files (typically output from a recognition tool such as HVite) and compares them with the corresponding reference transcription files. For the analysis of speech recognition output, the comparison is based on a Dynamic Programming-based string alignment procedure. For the analysis of word-spotting output, the comparison uses the standard US NIST FOM metric.

When used to calculate the sentence accuracy using DP the basic output is recognition statistics for the whole file set in the format

```
------------------------- Overall Results --------------------
SENT:  %Correct=13.00 [H=13, S=87, N=100]
WORD:  %Corr=53.36, Acc=44.90 [H=460,D=49,S=353,I=73,N=862]
================================================================
```

The first line gives the sentence-level accuracy based on the total number of label files which are identical to the transcription files. The second line is the word accuracy based on the DP matches between the label files and the transcriptions [14]. In this second line, H is the number of correct labels, D is the number of deletions, S is the number of substitutions, I is the number of insertions and N is the total number of labels in the defining transcription files. The percentage number of labels correctly recognised is given by

$$\%\text{Correct} = \frac{H}{N} \times 100\% \tag{17.4}$$

and the accuracy is computed by

$$\text{Accuracy} = \frac{H - I}{N} \times 100\% \tag{17.5}$$

In addition to the standard HTK output format, HResults provides an alternative similar to that used in the US NIST scoring package, i.e.

```
|================================================================|
|            # Snt |  Corr    Sub    Del    Ins    Err  S. Err |
|----------------------------------------------------------------|
| Sum/Avg |   87   |  53.36  40.95   5.68   8.47  55.10  87.00 |
`----------------------------------------------------------------'
```

When HResults is used to generate a confusion matrix, the values are as follows:

%c The percentage correct in the row; that is, how many times a phone instance was correctly labelled.

%e The percentage of incorrectly labeled phones in the row as a percentage of the total number of labels in the set.

An example from the HTKDemo routines:

```
====================== HTK Results Analysis =======================
  Date: Thu Jan 10 19:00:03 2002
  Ref : labels/bcplabs/mon
  Rec : test/te1.rec
      : test/te2.rec
      : test/te3.rec
------------------------ Overall Results --------------------------
SENT: %Correct=0.00 [H=0, S=3, N=3]
```

[14] The choice of "Sentence" and "Word" here is the usual case but is otherwise arbitrary. HResults just compares label sequences. The sequences could be paragraphs, sentences, phrases or words, and the labels could be phrases, words, syllables or phones, etc. Options exist to change the output designations 'SENT' and 'WORD' to whatever is appropriate.

```
WORD: %Corr=63.91, Acc=59.40 [H=85, D=35, S=13, I=6, N=133]
------------------------ Confusion Matrix -------------------------
       S   C   V   N   L  Del [ %c / %e]
   S   6   1   0   1   0    0 [75.0/1.5]
   C   2  35   3   1   0   18 [85.4/4.5]
   V   0   1  28   0   1   12 [93.3/1.5]
   N   0   1   0   7   0    1 [87.5/0.8]
   L   0   1   1   0   9    4 [81.8/1.5]
Ins    2   2   0   2   0
=================================================================
```

Reading across the rows, %c indicates the number of correct instances divided by the total number of instances in the row. %e is the number of incorrect instances in the row divided by the total number of instances (N).

Optional extra outputs available from HResults are

- recognition statistics on a per file basis
- recognition statistics on a per speaker basis
- recognition statistics from best of N alternatives
- time-aligned transcriptions
- confusion matrices

For comparison purposes, it is also possible to assign two labels to the same equivalence class (see `-e option`). Also, the *null* label `???` is defined so that making any label equivalent to the null label means that it will be ignored in the matching process. Note that the order of equivalence labels is important, to ensure that label `X` is ignored, the command line option `-e ??? X` would be used. Label files containing triphone labels of the form `A-B+C` can be optionally stripped down to just the class name `B` via the `-s` switch.

The word spotting mode of scoring can be used to calculate hits, false alarms and the associated figure of merit for each of a set of keywords. Optionally it can also calculate ROC information over a range of false alarm rates. A typical output is as follows

```
------------------------ Figures of Merit -------------------------
      KeyWord:    #Hits     #FAs  #Actual      FOM
            A:        8        1       14    30.54
            B:        4        2       14    15.27
      Overall:       12        3       28    22.91
-------------------------------------------------------------------
```

which shows the number of hits and false alarms (FA) for two keywords `A` and `B`. A label in the test file with start time t_s and end time t_e constitutes a hit if there is a corresponding label in the reference file such that $t_s < t_m < t_e$ where t_m is the mid-point of the reference label.

Note that for keyword scoring, the test transcriptions must include a score with each labelled word spot and all transcriptions must include boundary time information.

The FOM gives the % of hits averaged over the range 1 to 10 FA's per hour. This is calculated by first ordering all spots for a particular keyword according to the match score. Then for each FA rate f, the number of hits are counted starting from the top of the ordered list and stopping when f have been encountered. This corresponds to *a posteriori* setting of the keyword detection threshold and effectively gives an upper bound on keyword spotting performance.

17.18.2 Use

HResults is invoked by typing the command line

```
HResults [options] hmmList recFiles ...
```

This causes HResults to be applied to each `recFile` in turn. The `hmmList` should contain a list of all model names for which result information is required. Note, however, that since the context dependent parts of a label can be stripped, this list is not necessarily the same as the one used to

perform the actual recognition. For each `recFile`, a transcription file with the same name but the extension `.lab` (or some user specified extension - see the `-X` option) is read in and matched with it. The `recfiles` may be master label files (MLFs), but note that even if such an MLF is loaded using the `-I` option, the list of files to be checked still needs to be passed, either as individual command line arguments or via a script with the `-S` option. For this reason, it is simpler to pass the `recFile` MLF as one of the command line filename arguments. For loading reference label file MLF's, the `-I` option must be used. The reference labels and the recognition labels must have different file extensions. The available options are

`-a s` change the label `SENT` in the output to `s`.

`-b s` change the label `WORD` in the output to `s`.

`-c` when comparing labels convert to upper case. Note that case is still significant for equivalences (see `-e` below).

`-d N` search the first `N` alternatives for each test label file to find the most accurate match with the reference labels. Output results will be based on the most accurate match to allow NBest error rates to be found.

`-e s t` the label `t` is made equivalent to the label `s`. More precisely, `t` is assigned to an equivalence class of which `s` is the identifying member.

`-f` Normally, HResults accumulates statistics for all input files and just outputs a summary on completion. This option forces match statistics to be output for each input test file.

`-g fmt` This sets the test label format to `fmt`. If this is not set, the `recFiles` should be in the same format as the reference files.

`-h` Output the results in the same format as US NIST scoring software.

`-k s` Collect and output results on a speaker by speaker basis (as well as globally). `s` defines a pattern which is used to extract the speaker identifier from the test label file name. In addition to the pattern matching metacharacters `*` and `?` (which match zero or more characters and a single character respectively), the character `%` matches any character whilst including it as part of the speaker identifier.

`-m N` Terminate after collecting statistics from the first `N` files.

`-n` Set US NIST scoring software compatibility.

`-p` This option causes a phoneme confusion matrix to be output.

`-s` This option causes all phoneme labels with the form `A-B+C` to be converted to `B`. It is useful for analysing the results of phone recognisers using context dependent models.

`-t` This option causes a time-aligned transcription of each test file to be output provided that it differs from the reference transcription file.

`-u f` Changes the time unit for calculating false alarm rates (for word spotting scoring) to `f` hours (default is 1.0).

`-w` Perform word spotting analysis rather than string accuracy calculation.

`-z s` This redefines the null class name to `s`. The default null class name is `???`, which may be difficult to manage in shell script programming.

`-G fmt` Set the label file format to `fmt`.

`-I mlf` This loads the master label file `mlf`. This option may be repeated to load several MLFs.

`-L dir` Search directory `dir` for label files (default is to search current directory).

`-X ext` Set label file extension to `ext` (default is `lab`).

HResults also supports the standard options `-A`, `-C`, `-D`, `-S`, `-T`, and `-V` as described in section 4.4.

17.18.3 Tracing

HResults supports the following trace options where each trace flag is given using an octal base

`00001` basic progress reporting.

`00002` show error rate for each test alternative.

`00004` show speaker identifier matches.

`00010` warn about non-keywords found during word spotting.

`00020` show detailed word spotting scores.

`00040` show memory usage.

Trace flags are set using the `-T` option or the `TRACE` configuration variable.

17.19 HSGen

17.19.1 Function

This program will read in a word network definition in standard HTK lattice format representing a Regular Grammar G and randomly generate sentences from the language $L(G)$ of G. The sentences are written to standard output, one per line and an option is provided to number them if required.

The empirical entropy H_e can also be calculated using the formula

$$H_e = \frac{\sum_k P(S_k)}{\sum_k |S_k|} \tag{17.6}$$

where S_k is the k'th sentence generated and $|S_k|$ is its length. The perplexity P_e is computed from H_e by

$$P_e = 2^{H_e} \tag{17.7}$$

The probability of each sentence $P(S_k)$ is computed from the product of the individual branch probabilities.

17.19.2 Use

HSGen is invoked by the command line

```
HSGen [options] wdnet dictfile
```

where `dictfile` is a dictionary containing all of the words used in the word network stored in `wdnet`. This dictionary is only used as a word list, the pronunciations are ignored.

The available options are

`-l` When this option is set, each generated sentence is preceded by a line number.

`-n N` This sets the total number of sentences generated to be `N` (default value 100).

`-q` Set quiet mode. This suppresses the printing of sentences. It is useful when estimating the entropy of $L(G)$ since the accuracy of the latter depends on the number of sentences generated.

`-s` Compute word network statistics. When set, the number of network nodes, the vocabulary size, the empirical entropy, the perplexity, the average sentence length, the minimum sentence length and the maximum sentence length are computed and printed on the standard output.

HSLab also supports the standard options `-A`, `-C`, `-D`, `-S`, `-T`, and `-V` as described in section 4.4.

17.19.3 Tracing

HSLab supports the following trace options where each trace flag is given using an octal base

`00001` basic progress reporting

`00002` detailed trace of lattice traversal

Trace flags are set using the `-T` option or the `TRACE` configuration variable.

17.20 HSLab

17.20.1 Function

HSLab is an interactive label editor for manipulating speech label files. An example of using HSLab would be to load a sampled waveform file, determine the boundaries of the speech units of interest and assign labels to them. Alternatively, an existing label file can be loaded and edited by changing current label boundaries, deleting and creating new labels. HSLab is the only tool in the HTK package which makes use of the graphics library HGraf.

When started HSLab displays a window which is split into two parts: a display section and a control section (see Fig 17.1). The display section contains the plotted speech waveform with the associated labels. The control section consists of a palette of buttons which are used to invoke the various facilities available in the tool. The buttons are laid out into three different groups depending on the function they perform. Group one (top row) contains buttons related to basic input/output commands. Group two (middle row) implements the viewing and record/playback functions. The buttons in group three (bottom row) are used for labelling. To invoke a particular function, place the mouse pointer onto the corresponding button and click once. All commands which require further interaction with the user after invocation will display guiding text in the message area telling the user what he or she is expected to do next. For example, to delete a label, the user will click on `Delete`, the message "Please select label to delete" will appear in the message area and the user will be expected to click in that part of the display section corresponding to the label to be deleted (not on the label itself).

A *marked region* is a slice of the waveform currently visible in the window. A region is marked by clicking on `Mark` and specifying two boundaries by clicking in the display section. When marked, a region will be displayed in inverse colours. In the presence of a marked region the commands `Play`, `Label` and `Label as` will be applied to the specified region rather than to the whole of the waveform visible on the screen. Part of the waveform can also be made into a marked region with the commands `Zoom Out` and `Select`. `Zoom Out` will take the user back to the previous level of magnification and the waveform being displayed before the execution of the command will become a marked region. `Select` will make the part of the waveform corresponding to a particular label into a marked region. This can be useful for playing back existing labels.

Labelling is carried out with `Label` and `Label as`. `Label` will assign *The Current Label* to a specified slice of the waveform, whilst `Label as` will prompt the user to type-in the labelling string. *The Current Label* is shown in the button in the bottom right corner of the control section. It defaults to "Speech" and it can be changed by clicking on the button it resides in. Multiple alternative transcriptions are manipulated using the `Set [?]` and `New` buttons. The former is used to select the desired transcription, the latter is used to create a new alternative transcription.

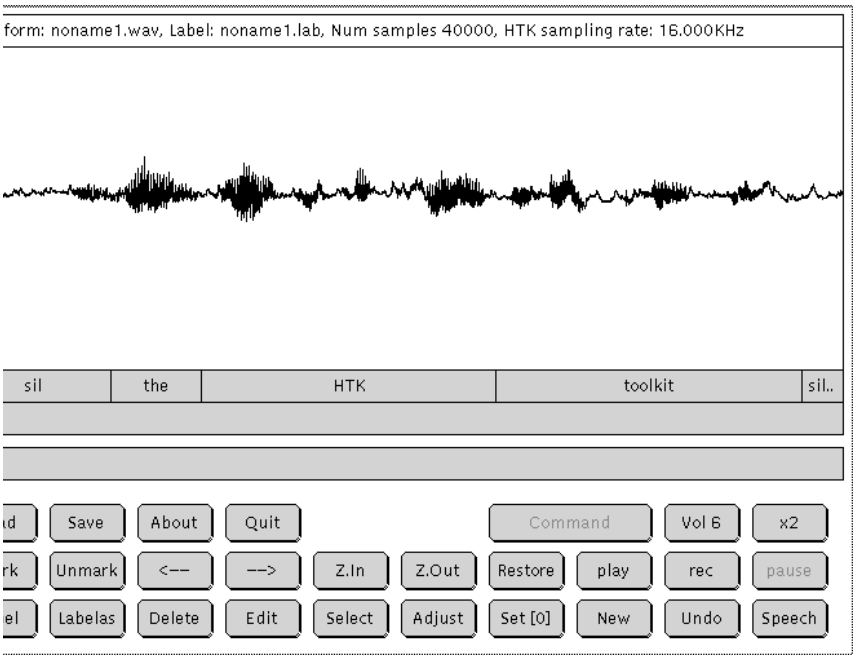


Fig. 17.1 HSLab display window

17.20.2 Use

HSLab is invoked by typing the command line

```
HSLab [options] dataFile
```

where `dataFile` is a data file in any of the supported formats with a `WAVEFORM` sample kind. If the given data file does not exist, then HSLab will assume that a new file is to be recorded with this name.

The available options for HSLab are

`-a` With this switch present, the numeric part of the global labelling string is automatically incremented after every `Label` operation.

`-i file` This option allows transcription files to be output to the named master label file (MLF).

`-n` Normally HSLab expects to load an existing label file whose name is derived from the speech data file. This option tells HSLab that a new empty transcription is to be created for the loaded data-file.

`-s string` This option allows the user to set the string displayed in the "command" button used to trigger external commands.

`-F fmt` Set the source data format to `fmt`.

`-G fmt` Set the label file format to `fmt`.

`-I mlf` This loads the master label file `mlf`. This option may be repeated to load several MLFs.

`-L dir` Search directory `dir` for label files (default is to search current directory).

`-X ext` Set label file extension to `ext` (default is `lab`).

HSLab also supports the standard options `-A`, `-C`, `-D`, `-S`, `-T`, and `-V` as described in section 4.4.

The following is a summary of the function of each HSLab button.

`Load` Load a speech data file and associated transcription. If changes have been made to the currently loaded transcription since it was last saved the user will be prompted to save these changes before loading the new file.

`Save` Save changes made to the transcription into a specified label file.

`About` Print information about HSLab.

`Quit` Exit from HSLab. If alterations have been made to the currently loaded transcription since it was last saved, the user will be prompted to save these changes before exiting.

`Command` This button is used to trigger an external command which can process the waveform file currently loaded in HSLab. This is accomplished by setting the environment variable `HSLABCMD` to the shell command required to perform the processing. When the `Command` button is pressed, any occurrence of `$` in the shell command is replaced by the name of the currently loaded waveform file. Note that only the filename without its extension is substituted. The string displayed in the "command" button can be changed using the `-s` option.

`Mark` Mark a region of the displayed waveform. The user is prompted to specify the start and the end point of a region with the mouse pointer. The marked region will be displayed in inverse colours. Only one region can be marked at a time.

`Unmark` Unmark a previously marked region.

`<--` Scroll the display to the left.

`-->` Scroll the display to the right.

`Z.In` Zoom into a part of the displayed waveform. If there is a currently marked region then that region will be zoomed into, otherwise, the user will be prompted to select a slice of the waveform by specifying two points using the mouse pointer.

`Z.Out` Restore the previous viewing level.

`Restore` Display the complete waveform into the window. Any marked regions will be unmarked.

`Play` If there is a marked region of the waveform then that portion of the signal will be played through the internal speaker. Otherwise, the command will apply to the waveform visible on the screen.

`Rec` This initiates recording from the audio input device. The maximum duration of a recording is limited to 2 mins at 16KHz sampling rate. Two bar-graphs are displayed: the first (red) shows the number of samples recorded, the second bar (green) displays the energy of the incoming signal. Once pressed, the `Rec` button changes into `Stop` which, in turn, is used to terminate the operation. When finished, the audio data stored in the buffer is written out to disk. Each recording is stored in alternating files `dataFile_0` and `dataFile_1`.

`Pause` Clicking on this button pauses/un-pauses the recording operation.

`Volume` This button is used to select the playback volume of the audio device.

`x1` This button selects the current level of waveform magnification. The available factors are $\times 1$, $\times 2$, $\times 4$, $\times 8$, $\times 16$, and $\times 32$.

`Label` If a marked region exists, then the waveform contained in the region will be labelled with *The Current Label*. Otherwise, the command will be applied to the waveform visible on the screen.

`Labelas` Same as above, however, the user is prompted to type in the labelling string.

`Delete` Delete a label.

`Edit` Edit the string of a label.

`Select` Select a label as a marked region.

`Adjust` Adjust the boundaries of a label. To select the label boundary to adjust, click in the display near to the label boundary.

`Set [?]` This button is used to select the current alternative transcription displayed and used in HSLAB.

`New` Creates a new alternative transcription. If an empty alternative transcription already exists, then a new transcription is not created.

`Undo` Single level undo operation for labelling commands.

`Speech` Change the current labelling string (the button in the bottom right of the control area).

The following "mouse" shortcuts are provided. To mark a region position the pointer at one of the desired boundaries, then press the left mouse button and while holding it down position the pointer at the other region boundary. Upon releasing the mouse button the marked region will be hilighted. To play a label position the mouse cursor anywhere within the corresponding label "slice" in the label area of the display and click the left mouse button.

17.20.3 Tracing

HSLAB does not provide any trace options.

17.21 HSmooth

17.21.1 Function

This program is used to smooth a set of context-dependent tied mixture or discrete HMM's using deleted interpolation. The program operates as a replacement for the second pass of HERest when working in *parallel mode*[15]. It reads in the N sets of accumulator files containing the statistics accumulated during the first pass and then interpolates the mixture weights between each context dependent model and its corresponding context independent version. The interpolation weights are chosen to maximise the likelihood of each deleted block with respect to the probability distributions estimated from all other blocks.

17.21.2 Use

HSmooth is invoked via the command line

```
HSmooth [options] hmmList accFile ...
```

where `hmmList` contains the list of context dependent models to be smoothed and each `accFile` is a file of the form `HERN.acc` dumped by a previous execution of HERest with the `-p` option set to `N`. The HMM definitions are loaded and then for every state and stream of every context dependent model X, the optimal interpolation weight is found for smoothing between the mixture weights determined from the X accumulators alone and those determined from the context independent version of X. The latter is computed simply by summing the accumulators across all context dependent allophones of X.

The detailed operation of HSmooth is controlled by the following command line options

`-b f` Set the value of epsilon for convergence in the binary chop optimisation procedure to `f`. The binary chop optimisation procedure for each interpolation weight terminates when the gradient is within epsilon of zero (default 0.001).

`-c N` Set maximum number of interpolation iterations for the binary chop optimisation procedure to be N (default 16).

`-d dir` Normally HSmooth expects to find the HMM definitions in the current directory. This option tells HSmooth to look in the directory `dir` to find them.

`-m N` Set the minimum number of training examples required for any model to `N`. If the actual number falls below this value, the HMM is not updated and the original parameters are used for the new version (default value 1).

`-o ext` This causes the file name extensions of the original models (if any) to be replaced by `ext`.

`-s file` This causes statistics on occupation of each state to be output to the named file.

`-u flags` By default, HSmooth updates all of the HMM parameters, that is, means, variances and transition probabilies. This option causes just the parameters indicated by the `flags` argument to be updated, this argument is a string containing one or more of the letters `m` (mean), `v` (variance) , `t` (transition) and `w` (mixture weight). The presence of a letter enables the updating of the corresponding parameter set.

`-v f` This sets the minimum variance (i.e. diagonal element of the covariance matrix) to the real value `f` (default value 0.0).

`-w f` Any mixture weight which falls below the global constant `MINMIX` is treated as being zero. When this parameter is set, all mixture weights are floored to `f * MINMIX`.

`-x ext` By default, HSmooth expects a HMM definition for the model X to be stored in a file called `X`. This option causes HSmooth to look for the HMM definition in the file `X.ext`.

`-B` Output HMM definition files in binary format.

`-H mmf` Load HMM macro model file `mmf`. This option may be repeated to load multiple MMFs.

[15] It is not,of course, necessary to have multiple processors to use this program since each 'parallel' activation can be executed sequentially on a single processor

`-M dir` Store output HMM macro model files in the directory `dir`. If this option is not given, the new HMM definition will overwrite the existing one.

HSmooth also supports the standard options `-A`, `-C`, `-D`, `-S`, `-T`, and `-V` as described in section 4.4.

17.21.3 Tracing

HSmooth supports the following trace options where each trace flag is given using an octal base

`00001` basic progress reporting.

`00002` show interpolation weights.

`00004` give details of optimisation algorithm.

Trace flags are set using the `-T` option or the `TRACE` configuration variable.

17.22 HVite

17.22.1 Function

HVite is a general-purpose Viterbi word recogniser. It will match a speech file against a network of HMMs and output a transcription for each. When performing N-best recognition a word level lattice containing multiple hypotheses can also be produced.

Either a word level lattice or a label file is read in and then expanded using the supplied dictionary to create a model based network. This allows arbitrary finite state word networks and simple forced alignment to be specified.

This expansion can be used to create context independent, word internal context dependent and cross word context dependent networks. The way in which the expansion is performed is determined automatically from the dictionary and HMMList. When all labels appearing in the dictionary are defined in the HMMList no expansion of model names is performed. Otherwise if all the labels in the dictionary can be satisfied by models dependent only upon word internal context these will be used else cross word context expansion will be performed. These defaults can be overridden by HNet configuration parameters.

HVite supports shared parameters and appropriately pre-computes output probabilities. For increased processing speed, HVite can optionally perform a beam search controlled by a user specified threshold (see `-t` option). When fully tied mixture models are used, observation pruning is also provided (see the `-c` option). Speaker adaptation is also supported by HVite both in terms of recognition using an adapted model set or a TMF (see the `-J` option), and in the estimation of a transform by unsupervised adaptation using maximum likelihood linear regression (MLLR) (see the `-j` option).

17.22.2 Use

HVite is invoked via the command line

```
HVite [options] dictFile hmmList testFiles ...
```

HVite will then either load a single network file and match this against each of the test files `-w netFile`, or create a new network for each test file either from the corresponding label file `-a` or from a word lattice `-w`. When a new network is created for each test file the path name of the label (or lattice) file to load is determined from the test file name and the `-L` and `-X` options described below.

If no `testFiles` are specified the `-w s` option must be specified and recognition will be performed from direct audio.

The `hmmList` should contain a list of the models required to construct the network from the word level representation.

The recogniser output is written in the form of a label file whose path name is determined from the test file name and the `-l` and `-x` options described below. The list of test files can be stored in a script file if required.

When performing N-best recognition (see `-n N` option described below) the output label file can contain multiple alternatives `-n N M` and a lattice file containing multiple hypotheses can be produced.

The detailed operation of HVite is controlled by the following command line options

`-a` Perform alignment. HVite will load a label file and create an alignment network for each test file.

`-b s` Use `s` as the sentence boundary during alignment.

`-c f` Set the tied-mixture observation pruning threshold to `f`. When all mixtures of all models are tied to create a full tied-mixture system, the calculation of output probabilities is treated as a special case. Only those mixture component probabilities which fall within `f` of the maximum mixture component probability are used in calculating the state output probabilities (default 10.0).

`-d dir` This specifies the directory to search for the HMM definition files corresponding to the labels used in the recognition network.

`-e` When using direct audio input, output transcriptions are not normally saved. When this option is set, each output transcription is written to a file called `PnS` where `n` is an integer which increments with each output file, `P` and `S` are strings which are by default empty but can be set using the configuration variables `RECOUTPREFIX` and `RECOUTSUFFIX`.

`-f` During recognition keep track of full state alignment. The output label file will contain multiple levels. The first level will be the state number and the second will be the word name (not the output symbol).

`-g` When using direct audio input, this option enables audio replay of each input utterance after it has been recognised.

`-i s` Output transcriptions to MLF `s`.

`-j i` Perform incremental MLLR adaptation every i utterances

`-k s1 s2` Set the description field `s1` in the transform model file to `s2`. Currently the choices for `field` are `uid`, `uname`, `chan` and `desc`.

`-l dir` This specifies the directory to store the output label files. If this option is not used then HVite will store the label files in the same directory as the data. When output is directed to an MLF, this option can be used to add a path to each output file name. In particular, setting the option `-l '*'` will cause a label file named `xxx` to be prefixed by the pattern `"*/xxx"` in the output MLF file. This is useful for generating MLFs which are independent of the location of the corresponding data files.

`-m` During recognition keep track of model boundaries. The output label file will contain multiple levels. The first level will be the model number and the second will be the word name (not the output symbol).

`-n i [N]` Use `i` tokens in each state to perform N-best recognition. The number of alternative output hypotheses `N` defaults to 1.

`-o s` Choose how the output labels should be formatted. `s` is a string with certain letters (from `NSCTWM`) indicating binary flags that control formatting options. `N` normalise acoustic scores by dividing by the duration (in frames) of the segment. `S` remove scores from output label. By default scores will be set to the total likelihood of the segment. `C` Set the transcription labels to start and end on frame centres. By default start times are set to the start time of the frame and end times are set to the end time of the frame. `T` Do not include times in output label files. `W` Do not include words in output label files when performing state or model alignment. `M` Do not include model names in output label files when performing state and model alignment.

`-p f` Set the word insertion log probability to `f` (default 0.0).

`-q s` Choose how the output lattice should be formatted. `s` is a string with certain letters (from `ABtvaldmn`) indicating binary flags that control formatting options. `A` attach word labels to arcs rather than nodes. `B` output lattices in binary for speed. `t` output node times. `v` output pronunciation information. `a` output acoustic likelihoods. `l` output language model likelihoods. `d` output word alignments (if available). `m` output within word alignment durations. `n` output within word alignment likelihoods.

`-r f` Set the dictionary pronunciation probability scale factor to `f`. (default value 1.0).

`-s f` Set the grammar scale factor to `f`. This factor post-multiplies the language model likelihoods from the word lattices. (default value 1.0).

`-t f [i l]` Enable beam searching such that any model whose maximum log probability token falls more than `f` below the maximum for all models is deactivated. Setting `f` to 0.0 disables the beam search mechanism (default value `0.0`). In alignment mode two extra parameters `i` and `l` can be specified. If the alignment fails at the initial pruning threshold `f`, then the threshold will by increased by `i` and the alignment will be retried. This procedure is repeated until the alignment succeeds or the threshold limit `l` is reached.

`-u i` Set the maximum number of active models to `i`. Setting `i` to `0` disables this limit (default 0).

`-v f` Enable word end pruning. Do not propagate tokens from word end nodes that fall more than `f` below the maximum word end likelihood. (default `0.0`).

`-w [s]` Perform recognition from word level networks. If `s` is included then use it to define the network used for every file.

`-x ext` This sets the extension to use for HMM definition files to `ext`.

`-y ext` This sets the extension for output label files to `ext` (default `rec`).

`-z ext` Enable output of lattices (if performing NBest recognition) with extension `ext` (default off).

`-L dir` This specifies the directory to find input label (when `-a` is specified) or network files (when `-w` is specified).

`-X s` Set the extension for the input label or network files to be `s` (default value `lab`).

`-F fmt` Set the source data format to `fmt`.

`-G fmt` Set the label file format to `fmt`.

`-H mmf` Load HMM macro model file `mmf`. This option may be repeated to load multiple MMFs.

`-I mlf` This loads the master label file `mlf`. This option may be repeated to load several MLFs.

`-J tmf` Load a transform set from the transform model file `tmf`.

`-K tmf` Save the transform set in the transform model file `tmf`.

`-P fmt` Set the target label format to `fmt`.

HVite also supports the standard options `-A`, `-C`, `-D`, `-S`, `-T`, and `-V` as described in section 4.4.

17.22.3 Tracing

HVite supports the following trace options where each trace flag is given using an octal base

`0001` enable basic progress reporting.

`0002` list observations.

`0004` frame-by-frame best token.

`0010` show memory usage at start and finish.

`0020` show memory usage after each utterance.

Trace flags are set using the `-T` option or the `TRACE` configuration variable.

17.23 LAdapt

17.23.1 Function

This program will adapt an existing language model from supplied text data. This is accomplished in two stages. First, the text data is scanned and a new language model is generated. In the second stage, an existing model is loaded and adapted (merged) with the newly created one according to the specified ratio. The target model can be optionally pruned to a specific vocabulary. Note that you can only apply this tool to word models or the class n-gram component of a class model – that is, you cannot apply it to full class models.

17.23.2 Use

LAdapt is invoked by the command line

```
LAdapt [options] -i weight inLMFile outLMFile [texttfile ...]
```

The text data is scanned and a new LM generated. The input language model is then loaded and the two models merged. The effect of the weight (0.0-1.0) is to control the overall contribution of each model during the merging process. The output to outLMFile is an n-gram model stored in the user-specified format.

The allowable options to LAdapt are as follows

`-a n` Allow upto `n` new words in input text (default 100000).

`-b n` Set the n-gram buffer size to n. This controls the size of the buffer used to accumulate n-gram statistics whilst scanning the input text. Larger buffer sizes will result in more efficient operation of the tool with fewer sort operations required (default 2000000).

`-c n c` Set the pruning threshold for n-grams to c. Pruning can be applied to the bigram (n=2) and longer (n¿2) components of the newly generated model. The pruning procedure will keep only n-grams which have been observed more than c times.

`-d s` Set the root n-gram data file name to `s`. By default, n-gram statistics from the text data will be accumulated and stored as `gram.0, gram.1, ...`, etc. Note that a larger buffer size will result in fewer files.

`-f s` Set the output language model format to `s`. Possible options are `text` for the standard ARPA-MIT LM format, `bin` for Entropic *binary* format and `ultra` for Entropic *ultra* format.

`-g` Use existing n-gram data files. If this option is specified the tool will use the existing gram files rather than scanning the actual texts. This option is useful when adapting multiple language models from the same text data or when experimenting with different merging weights.

`-i w f` Interpolate with model `f` using weight `w`. Note that at least one model must be specified with this option.

`-j n c` Set weighted discounting pruning for n grams to c. This cannot be applied to unigrams (n=1).

`-n n` Produce n-gram language model.

`-s s` Store `s` in the header of the gram files.

`-t` Force Turing-Good discounting if configured otherwise.

`-w fn` Load word list from `fn`. The word list will be used to define the target model's vocabulary. If a word list is not specified, the target model's vocabulary will have all words from the source model(s) together with any new words encountered in the text data.

`-x` Create a count-based model.

LAdapt also supports the standard options `-A`, `-C`, `-D`, `-S`, `-T`, and `-V` as described in section 4.4.

17.23.3 Tracing

LADAPT supports the following trace options where each trace flag is given using an octal base

`00001` basic progress reporting

`00002` monitor buffer saving

`00004` trace word input stream

`00010` trace shift register input

Trace flags are set using the `-T` option or the `TRACE` configuration variable.

17.24 LBuild

17.24.1 Function

This program will read one or more input gram files and generate/update a back-off n-gram language model as described in section 14.5. The `-n` option specifies the order of the final model. Thus, to generate a trigram language model, the user may simply invoke the tool with `-n 3` which will cause it to compute the FoF table and then generate the unigram, bigram and trigram stages of the model. Note that intermediate model/FoF files will not be generated.

As for all tools which process gram files, the input gram files must each be sorted but they need not be sequenced. The counts in each input file can be modified by applying a multiplier factor. Any n-gram containing an id which is not in the word map is ignored, thus, the supplied word map will typically contain just those word and class ids required for the language model under construction (see LSubset).

LBuild supports Turing-Good and absolute discounting as described in section **??**.

17.24.2 Use

LBuild is invoked by typing the command line

```
LBuild [options] wordmap outfile [mult] gramfile .. [mult] gramfile ..
```

The given word map file is loaded and then the set of named gram files are merged to form a single sorted stream of n-grams. Any n-grams containing ids not in the word map are ignored. The list of input gram files can be interspersed with multipliers. These are floating-point format numbers which must begin with a plus or minus character (e.g. `+1.0`, `-0.5`, etc.). The effect of a multiplier `x` is to scale the n-gram counts in the following gram files by the factor `x`. A multiplier stays in effect until it is redefined. The output to `outfile` is a back-off n-gram language model file in the specified file format.

See the LPCalc options in section 18.1 for details on changing the discounting type from the default of Turing-Good, as well as other configuration file options.

The allowable options to LBuild are as follows

`-c n c` Set cutoff for `n`-gram to `c`.

`-d n c` Set weighted discount pruning for `n`-gram to `c` for Seymore-Rosenfeld pruning.

`-f t` Set output model format to `t` (TEXT, BIN, ULTRA).

`-k n` Set discounting range for Good-Turing discounting to [1..n].

`-l f` Build model by updating existing LM in `f`.

`-n n` Set final model order to `n`.

`-t ff` Load the FoF file `f`. This is only used for Turing-Good discounting, and is not essential.

`-u c` Set the minimum occurrence count for unigrams to `c`. (Default is 1)

`-x` Produce a counts model.

LBuild also supports the standard options `-A`, `-C`, `-D`, `-S`, `-T`, and `-V` as described in section 4.4.

17.24.3 Tracing

LBuild supports the following trace options where each trace flag is given using an octal base

`00001` basic progress reporting.

Trace flags are set using the `-T` option or the `TRACE` configuration variable.

17.25 LFoF

17.25.1 Function

This program will read one or more input gram files and generate a *frequency-of-frequency* or *FoF* file. A FoF file is a list giving the number of times that an n-gram occurs just once, the number of times that an n-gram occurs just twice, etc. The format of a FoF file is described in section 16.6.

As for all tools which process gram files, the input gram files must each be sorted but they need not be sequenced. The counts in each input file can be modified by applying a multiplier factor. Any n-gram containing an id which is not in the word map is ignored, thus, the supplied word map will typically contain just those word and class ids required for the language model under construction (see LSubset).

LFoF also provides an option to generate an estimate of the number of n-grams which would be included in the final language model for each possible cutoff by setting `LPCALC: TRACE = 2`.

17.25.2 Use

LFoF is invoked by typing the command line

```
LFoF [options] wordmap foffile [mult] gramfile .. [mult] gramfile ..
```

The given word map file is loaded and then the set of named gram files are merged to form a single sorted stream of n-grams. Any n-grams containing ids not in the word map are ignored. The list of input gram files can be interspersed with multipliers. These are floating-point format numbers which must begin with a plus or minus character (e.g. `+1.0`, `-0.5`, etc.). The effect of a multiplier `x` is to scale the n-gram counts in the following gram files by the factor `x`. A multiplier stays in effect until it is redefined. The output to `foffile` is a FoF file as described in section 16.6.

The allowable options to LFoF are as follows

`-f N` set the number of FoF entries to N (default 100).

`-n N` Set n-gram size to `N` (defaults to max).

LFoF also supports the standard options `-A`, `-C`, `-D`, `-S`, `-T`, and `-V` as described in section 4.4.

17.25.3 Tracing

LFoF supports the following trace options where each trace flag is given using an octal base

`00001` basic progress reporting

Trace flags are set using the `-T` option or the `TRACE` configuration variable.

17.26 LGCopy

17.26.1 Function

This program will copy one or more input gram files to a set of one or more output gram files. The input gram files must each be sorted but they need not be sequenced. Unless word-to-class mapping is being performed, the output files will, however, be sequenced. Hence, given a collection of unsequenced gram files, LGCopy can be used to generate an equivalent sequenced set. This is useful for reducing the number of parallel input streams that tools such as LBuild must maintain, thereby improving efficiency.

As for all tools which can input gram files, the counts in each input file can be modified by applying a multiplier factor. Note, however, that since the counts within gram files are stored as integers, use of non-integer multiplier factors will lead to the counts being rounded in the output gram files.

In addition to manipulating the counts, the `-n` option also allows the input grams to be truncated by summing the counts of all equivalenced grams. For example, if the 3-grams `a x y 5` and `b x y 3` were truncated to 2-grams, then `x y 8` would be output. Truncation is performed before any of the mapping operations described below.

LGCopy also provides options to map gram words to classes using a class map file and filter the resulting output. The most common use of this facility is to map out-of-vocabulary (OOV) words into the unknown symbol in preparation for building a conventional word n-gram language model for a specific vocabulary. However, it can also be used to prepare for building a class-based n-gram language model.

Word-to-class mapping is enabled by specifying the class map file with the `-w` option. Each n-gram word is then replaced by its class symbol as defined by the class map. If the `-o` option is also specified, only n-grams containing class symbols are stored in the internal buffer.

17.26.2 Use

LGCopy is invoked by typing the command line

```
LGCopy [options] wordmap [mult] gramfile .... [mult] gramfile ...
```

The given word map file is loaded and then the set of named gram files are input in parallel to form a single sorted stream of n-grams. Counts for identical n-grams in multiple source files are summed. The merged stream is written to a sequence of output gram files named `data.0`, `data.1`, etc. The list of input gram files can be interspersed with multipliers. These are floating-point format numbers which must begin with a plus or minus character (e.g. `+1.0`, `-0.5`, etc.). The effect of a multiplier `x` is to scale the n-gram counts in the following gram files by the factor `x`. The resulting scaled counts are rounded to the nearest integer on output. A multiplier stays in effect until it is redefined. The scaled input grams can be truncated, mapped and filtered before being output as described above.

The allowable options to LGCopy are as follows

`-a n` Set the maximum number of new classes that can be added to the word map (default 1000, only used in conjuction with class maps).

`-b n` Set the internal gram buffer size to n (default 2000000). LGCopy stores incoming n-grams in this buffer. When the buffer is full, the contents are sorted and written to an output gram file. Thus, the buffer size determines the amount of process memory that LGCopy will use and the size of the individual output gram files.

`-d` Directory in which to store the output gram files (default current directory).

`-i n` Set the index of the first gram file output to be n (default 0).

`-m s` Save class-resolved word map to `fn`.

`-n n` Normally, n-gram size is preserved from input to output. This option allows the output n-gram size to be truncated to n where n must be less than the input n-gram size.

`-o n` Output class mappings only. Normally all input n-grams are copied to the output, however, if a class map is specified, this options forces the tool to output only n-grams containing at least one class symbol.

`-r s` Set the root name of the output gram files to `s` (default "gram").

LGCopy also supports the standard options `-A`, `-C`, `-D`, `-S`, `-T`, and `-V` as described in section 4.4.

17.26.3 Tracing

LGCopy supports the following trace options where each trace flag is given using an octal base

`00001` basic progress reporting.

`00002` monitor buffer save operations.

Trace flags are set using the `-T` option or the `TRACE` configuration variable.

17.27 LGList

17.27.1 Function

This program will list the contents of one or more HLM gram files. In addition to printing the whole file, an option is provided to print just those n-grams containing certain specified words and/or ids. It is mainly used for debugging.

17.27.2 Use

LGLIST is invoked by typing the command line

```
LGList [options] wmapfile gramfile ....
```

The specified gram files are printed to the output. The n-grams are printed one per line following a summary of the header information. Each n-gram is printed in the form of a list of words followed by the count.

Normally all n-grams are printed. However, if either of the options `-i` or `-f` are used to add words to a *filter list*, then only those n-grams which include a word in the filter list are printed.

The allowable options to LGLIST are as follows

`-f w` Add word `w` to the filter list. This option can be repeated, it can also be mixed with uses of the `-i` option.

`-i n` Add word with id `n` to the filter list. This option can be repeated, it can also be mixed with uses of the `-f` option.

LGLIST also supports the standard options `-A`, `-C`, `-D`, `-S`, `-T`, and `-V` as described in section 4.4.

17.27.3 Tracing

LGLIST supports the following trace options where each trace flag is given using an octal base

`00001` basic progress reporting.

Trace flags are set using the `-T` option or the `TRACE` configuration variable.

17.28 LGPrep

17.28.1 Function

The function of this tool is to scan a language model training text and generate a set of gram files holding the n-grams seen in the text along with their counts. By default, the output gram files are named `gram.0`, `gram.1`, `gram.2`, etc. However, the root name can be changed using the `-r` option and the start index can be set using the `-i` option.

Each output gram file is sorted but the files themselves will not be sequenced (see section 16.5). Thus, when using LGPrep with substantial training texts, it is good practice to subsequently copy the complete set of output gram files using LGCopy to reorder them into sequence. This process will also remove duplicate occurrences making the resultant files more compact and faster to read by the HLM processing tools.

Since LGPrep will often encounter new words in its input, it is necessary to update the word map. The normal operation therefore is that LGPrep begins by reading in a word map containing all the word ids required to decode all previously generated gram files. This word map is then updated to include all the new words seen in the current input text. On completion, the updated word map is output to a file of the same name as the input word map in the directory used to store the new gram files. Alternatively, it can be output to a specified file using the `-w` option. The sequence number in the header of the newly created word map will be one greater than that of the original.

LGPrep can also apply a set of "match and replace" edit rules to the input text stream. The purpose of this facility is not to replace input text conditioning filters but to make simple changes to the text after the main gram files have been generated. The editing works by passing the text through a window one word at a time. The edit rules consist of a pattern and a replacement text. At each step, the pattern of each rule is matched against the window and if a match occurs, then the matched word sequence is replaced by the string in the replaced part of the rule. Two sets of gram files are generated by this process. A "negative" set of gram files contain n-grams corresponding to just the text strings which were modified and a "positive" set of gram files contain n-grams corresponding to the modified text. All text for which no rules matched is ignored and generates no gram file output. Once the positive and negative gram files have been generated, the positive grams are added (i.e. input with a weight of +1) to the original set of gram files and the negative grams are subtracted (i.e. input with a weight of -1). The net result is that the tool reading the full set of gram files receives a stream of n-grams which will be identical to the stream that it would have received if the editing commands had been applied to the text source when the original main gram file set had been generated.

The edit rules are stored in a file and read in using the `-f` option. They consist of set definitions and rule definitions, each written on a separate line. Each set defines a set of words and is identified by an integer in the range 0 to 255

```
<set-def>     = '#'<number> <word1> <word2> ... <wordN>.
```

For example,

```
#4 red green blue
```

defines set number 4 as being the 3 words "red", "green" and "blue". Rules consist of an *application factor*, a *pattern* and and a *replacement*

```
<rule-def>    = <app-factor> <pattern> : <replacement>
<pattern>     = { <word> | '*' | !<set> | %<set> }
<replacement> = { '$'<field> | string } % $' - work around emacs
                                        % colouring bug
```

The application factor should be a real number in the range 0 to 1 and it specifies the proportion of occurrences of the pattern which should be replaced. The pattern consists of a sequence of words, wildcard symbols ("`*`") which match anyword, and set references of the form `%n` denoting any word which is in set number `n` and `!n` denoting any word which is not in set number `n`. The replacement consists of a sequence of words and field references of the form `$i` which denotes the `i'th` matching word in the input.

As an example, the following rules would translate 50% of the occurrences of numbers in the form "one hundred fifty" to "one hundred and fifty" and 30% of the occurrences of "one hundred" to "a hundred".

```
#0 one two three four five six seven eight nine fifty sixty seventy
#1 hundred
0.5 * * hundred %0 * * : $0 $1 $2 and $3 $4 $5
0.3 * * !0 one %1  * * : $0 $1 $2 a $4 $5 $6
```

Note finally, that LGPrep processes edited text in a parallel stream to normal text, so it is possible to generate edited gram files whilst generating the main gram file set. However, normally the main gram files already exist and so it is normal to suppress gram file generation using the `-z` option when using edit rules.

17.28.2 Use

LGPrep is invoked by typing the command line

```
LGPrep [options] wordmap [textfile ...]
```

Each text file is processed in turn and treated as a continuous stream of words. If no text files are specified standard input is used and this is the more usual case since it allows the input text source to be filtered before input to LGPrep, for example, using LCond.pl (in `LMTutorial/extras/`).

Each n-gram in the input stream is stored in a buffer. When the buffer is full it is sorted and multiple occurrences of the same n-gram are merged and the count set accordingly. When this process ceases to yield sufficient buffer space, the contents are written to an output gram file.

The word map file defines the mapping of source words to the numeric ids used within HLM tools. Any words not in the map are allocated new ids and added to the map. On completion, a new map with the same name (unless specified otherwise with the `-w` option) is output to the same directory as the output gram files. To initialise the first invocation of this updating process, a word map file should be created with a text editor containing the following:

```
Name=xxxx
SeqNo=0
Language=yyyy
Entries=0
Fields=ID
\Words\
```

where `xxxx` is an arbitrarily chosen name for the word map and `yyyy` is the language. Fields specifying the escaping mode to use (`HTK` or `RAW`) and changing `Fields` to include frequency counts in the output (i.e. `FIELDS = ID,WFC`) can also be given. Alternatively, they can be added to the output using command line options.

The allowable options to LGPrep are as follows

`-a n` Allow upto `n` new words in input texts (default 100000).

`-b n` Set the internal gram buffer size to `n` (default 2000000). LGPrep stores incoming n-grams in this buffer. When the buffer is full, the contents are sorted and written to an output gram file. Thus, the buffer size determines the amount of process memory that LGPrep will use and the size of the individual output gram files.

`-c` Add word counts to the output word map. This overrides the setting in the input word map (default off).

`-d` Directory in which to store the output gram files (default current directory).

`-e n` Set the internal edited gram buffer size to `n` (default 100000).

`-f s` Fix (i.e. edit) the text source using the rules in `s`.

`-h` Do not use HTK escaping in the output word map (default on).

`-i n` Set the index of the first gram file output to be `n` (default 0).

`-n n` Set the output n-gram size to `n` (default 3).

`-q` Tag words at sentence start with underscore (_).

`-r s` Set the root name of the output gram files to `s` (default "gram").

`-s s` Write the string `s` into the source field of the output gram files. This string should be a comment describing the text source.

`-w s` Write the output map file to `s` (default same as input map name stored in the output gram directory).

`-z` Suppress gram file output. This option allows LGPrep to be used just to compute a word frequency map. It is also normally applied when applying edit rules to the input.

`-Q` Print a summary of all commands supported by this tool.

LGPrep also supports the standard options `-A`, `-C`, `-D`, `-S`, `-T`, and `-V` as described in section 4.4.

17.28.3 Tracing

LGPrep supports the following trace options where each trace flag is given using an octal base

`00001` basic progress reporting.

`00002` monitor buffer save operations.

`00004` Trace word input stream.

`00010` Trace shift register input.

`00020` Rule input monitoring.

`00040` Print rule set.

Trace flags are set using the `-T` option or the `TRACE` configuration variable.

17.29 LLink

17.29.1 Function

This tool will create the link file necessary to use the word-given-class and class-given-class components of a class n-gram language model

Having created the class n-gram component with LBUILD and the word-given-class component with CLUSTER, you can then create a third file which points to these two other files by using the LLINK tool. This file is the language model you pass to utilities such as LPLEX. Alternatively if run with its `-s` option then LLINK will link the two components together and create a single resulting file.

17.29.2 Use

LLINK is invoked by the command line

```
LLink [options] word-classLMfile class-classLMfile outLMfile
```

The tool checks for the existence of the two existing component language model files, with `word-classLMfile` being the word-given-class file from CLUSTER and `class-classLMfile` being the class n-gram model generated by LBUILD. The word-given-class file is read to discover whether it is a count or probability-based file, and then an appropriate link file is written to `outLMfile`. This link file is then suitable for passing to LPLEX. Optionally you may overrule the count/probability distinction by using the `-c` and `-p` parameters. Passing the `-s` parameter joins the two files into one single resulting language model rather than creating a third link file which points to the other two.

The allowable options to LLINK are as follows

`-c` Force the link file to describe the word-given-class component as a 'counts' file.

`-p` Force the link file to describe the word-given-class component as a 'probabilities' file.

`-s` Write a single file containing both the word-class component and the class-class component. This single resulting file is then a self-contained language model requiring no other files.

LLINK also supports the standard options `-A`, `-C`, `-D`, `-S`, `-T`, and `-V` as described in section 4.4.

17.29.3 Tracing

LLINK supports the following trace options where each trace flag is given using an octal base

`00001` basic progress reporting

Trace flags are set using the `-T` option or the `TRACE` configuration variable.

17.30 LMerge

17.30.1 Function

This program combines one or more language models to produce an output model for a specified vocabulary. You can only apply it to word models or the class n-gram component of a class model – that is, you cannot apply it to full class models.

17.30.2 Use

LMERGE is invoked by typing the command line

```
LMerge [options] wordList inModel outModel
```

The word map and class map are loaded, word-class mappings performed and a new map is saved to `outMapFile`. The output map's name will be set to

```
Name = inMapName%%classMapName
```

The allowable options to LMERGE are as follows

`-f s` Set the output LM file format to `s`. Available options are `text`, `bin` or `ultra` (default `bin`).

`-i f fn` Interpolate with model `fn` using weight `f`.

`-n n` Produce an `n`-gram model.

LMERGE also supports the standard options `-A`, `-C`, `-D`, `-S`, `-T`, and `-V` as described in section 4.4.

17.30.3 Tracing

LMERGE Does not provide any trace options. However, trace information is available from the underlying library modules LWMAP and LCMAP by setting the appropriate trace configuration parameters.

17.31 LNewMap

17.31.1 Function

This tool will create an empty word map suitable for use with LGPrep.

17.31.2 Use

LNewMap is invoked by the command line

```
LNewMap [options] name mapfn
```

A new word map is created with the file name 'mapfn', with its constituent `Name` header set to the text passed in 'name'. It also creates default `SeqNo`, `Entries`, `EscMode` and `Fields` headers in the file. The contents of the `EscMode` header may be altered from the default of `RAW` using the `-e` option, whilst the `Fields` header contains `ID` but may be added to using the `-f` option.

The allowable options to LNewMap are therefore

`-e esc` Change the contents of the `EscMode` header to `esc`. Default is `RAW`.

`-f fld` Add the field `fld` to the `Fields` header.

LNewMap also supports the standard options `-A`, `-C`, `-D`, `-S`, `-T`, and `-V` as described in section 4.4.

17.31.3 Tracing

LNewMap supports the following trace options where each trace flag is given using an octal base

`00001` basic progress reporting

Trace flags are set using the `-T` option or the `TRACE` configuration variable.

17.32 LNorm

17.32.1 Function

The basic function of this tool is to renormalise language models, optionally pruning the vocabulary at the same time or applying cutoffs or weighted discounts.

17.32.2 Use

LNorm is invoked by the command line

```
LNorm [options] inLMFile outLMFile
```

This reads in the language model `inLMFile` and writes a new language model to `outLMFile`, applying editing operations controlled by the following options. In many respects it is similar to HLMCopy, but unlike HLMCopy it will always renormalise the resulting model.

`-c n c` Set the pruning threshold for n-grams to c. Pruning can be applied to the bigram and higher components of a model (n¿1). The pruning procedure will keep only n-grams which have been observed more than c times. Note that this option is only applicable to count-based language models.

`-d f` Set weighted discount pruning for `n`-gram to `c` for Seymore-Rosenfeld pruning. Note that this option is only applicable to count-based language models.

`-f s` Set the output language model format to `s`. Possible options are `TEXT` for the standard ARPA-MIT LM format, `BIN` for Entropic *binary* format and `ULTRA` for Entropic *ultra* format.

`-n n` Save target model as n-gram.

`-w f` Read a word-list defining the output vocabulary from `f`. This will be used to select the vocabulary for the output language model.

LNorm also supports the standard options `-A`, `-C`, `-D`, `-S`, `-T`, and `-V` as described in section 4.4.

17.32.3 Tracing

LNorm supports the following trace options where each trace flag is given using an octal base

`00001` basic progress reporting.

Trace flags are set using the `-T` option or the `TRACE` configuration variable.

17.33 LPlex

17.33.1 Function

This program computes the perplexity and out of vocabulary (OOV) statistics of text data using one or more language models. The perplexity is calculated on per-utterance basis. Each utterance in the text data should start with a sentence start symbol (`<s>`) and finish with a sentence end (`</s>`) symbol. The default values for the sentence markers can be changed via the config parameters `STARTWORD` and `ENDWORD` respectively. Text data can be supplied as an HTK Master Label File (MLF) or as plain text (`-t` option). Multiple perplexity tests can be performed on the same texts using separate n-gram components of the model(s). OOV words in the test data can be handled in two ways. By default the probability of n-grams containing words not in the lexicon is simply not calculated. This is useful for testing closed vocabulary models on texts containing OOVs. If the `-u` option is specified, n-grams giving the probability of an OOV word conditioned on its predecessors are discarded, however, the probability of words in the lexicon can be conditioned on context including OOV words. The latter mode of operation relies on the presence of the unknown class symbol (`!!UNK`) in the language model (the default value can be changed via the config parameter `UNKNOWNNAME`). If multiple models are specified (`-i` option) the probability of an n-gram will be calculated as a sum of the weighted probabilities from each of the models.

17.33.2 Use

LPlex is invoked by the command line

```
LPlex [options] langmodel labelFiles ...
```

The allowable options to LPlex are as follows

`-c n c` Set the pruning threshold for n-grams to c. Pruning can be applied to the bigram (n=2) and trigram (n=3) components of the model. The pruning procedure will keep only n-grams which have been observed more than c times. Note that this option is only applicable to the model generated from the text data.

`-e s t` Label `t` is made equivalent to label `s`. More precisely `t` is assigned to an equivalence class of which `s` is the identifying member. The equivalence mappings are applied to the text and should be used to map symbols in the text to symbols in the language model's vocabulary.

`-i w fn` Interpolate with model `fn` using weight `w`.

`-n n` Perform a perplexity test using the n-gram component of the model. Multiple tests can be specified. By default the tool will use the maximum value of n available.

`-o` Print a sorted list of unique OOV words encountered in the text and their occurrence counts.

`-t` Text stream mode. If this option is set, the specified test files will be assumed to contain plain text.

`-u` In this mode OOV words can be present in the n-gram context when predicting words in the vocabulary. The conditional probability of OOV words is still ignored.

`-w fn` Load word list in `fn`. The word list will be used as the restricting vocabulary for the perplexity calculation. If a word list file is not specified, the target vocabulary will be constructed by combining the vocabularies of all specified language models.

`-z s` Redefine the null equivalence class name to `s`. The default null class name is `???`. Any words mapped to the null class will be deleted from the text.

LPlex also supports the standard options `-A`, `-C`, `-D`, `-S`, `-T`, and `-V` as described in section 4.4.

17.33.3 Tracing

LPlex supports the following trace options where each trace flag is given using an octal base

`00001` basic progress reporting.

`00002` print information after each utterance processed.

`00004` display encountered OOVs.

`00010` display probability of each n-gram looked up.

`00020` print each utterance and its perplexity.

Trace flags are set using the `-T` option or the `TRACE` configuration variable.

17.34 LSubset

17.34.1 Function

This program will resolve a word map against a class map and produce a new word map which contains the class-mapped words. The tool is typically used to generated a vocabulary-specific n-gram word map which is then supplied to LBUILD to build the actual language models.

All class symbols present in the class map will be added to the output map. The `-a` option can be used to set the maximum number of new class symbols in the final word map. Note that the word-class map resolution procedure is identical to the the one used in LSUBSET when filtering n-gram files.

17.34.2 Use

LSUBSET is invoked by typing the command line

```
LSubset [options] inMapFile classMap outMapFile
```

The word map and class map are loaded, word-class mappings performed and a new map is saved to `outMapFile`. The output map's name will be set to

```
Name = inMapName%%classMapName
```

The allowable options to LSUBSET are as follows

`-a n` Set the maximum number of new classes that can be added to the output map (default 1000).

LSUBSET also supports the standard options `-A`, `-C`, `-D`, `-S`, `-T`, and `-V` as described in section 4.4.

17.34.3 Tracing

LSUBSET does not provide any trace options. However, trace information is available from the underlying library modules LWMAP and LCMAP by seeting the appropriate trace configuration parameters.

Chapter 18

Configuration Variables

This chapter tabulates all configuration variables used in HTK.

18.1 Configuration Variables used in Library Modules

Table 18.1: Library Module Configuration Variables

Module	Name	Default	Description
HParm HWave	SOURCEFORMAT	HTK	File format of source
	TARGETFORMAT	HTK	File format of target
HLabel HAudio HWave HParm	SOURCERATE	0.0	Sample rate of source in 100ns units
HParm HWave	TARGETRATE	0.0	Sample rate of target in 100ns units
HAudio	LINEOUT	T	Enable audio output to machine line output
	PHONESOUT	T	Enable audio output to machine phones output
	SPEAKEROUT	F	Enable audio output to machine internal speaker
	LINEIN	T	Enable audio input from machine line input
	MICIN	F	Enable audio input from machine mic input
HWave	NSAMPLES		Num samples in alien file input via a pipe
	HEADERSIZE		Size of header in an alien file
	BYTEORDER		Define byte order VAX or other
	STEREOMODE		Select channel: RIGHT or LEFT
HParm	SOURCEKIND	ANON	Parameter kind of source
	TARGETKIND	ANON	Parameter kind of target
	SAVECOMPRESSED	F	Save the output file in compressed form
	SAVEWITHCRC	T	Attach a checksum to output parameter file
	ADDDITHER	0.0	Level of noise added to input signal
	ZMEANSOURCE	F	Zero mean source waveform before analysis
	WINDOWSIZE	256000.0	Analysis window size in 100ns units
	USEHAMMING	T	Use a Hamming window
	DOUBLEFFT	F	Use twice the required size for FFT
	PREEMCOEF	0.97	Set pre-emphasis coefficient
	LPCORDER	12	Order of lpc analysis
	NUMCHANS	20	Number of filterbank channels
	LOFREQ	-1.0	Low frequency cut-off in fbank analysis
	HIFREQ	-1.0	High frequency cut-off in fbank analysis
	WARPFREQ	1.0	Frequency warping factor

Module	Name	Default	Description
	WARPLCUTOFF		Lower frequency threshold for non-linear warping
	CMEANDIR		Directory to find cepstral mean vecotrs
	CMEANMASK		Filename mask for cepstral mean vecotrs
	VARSCALEDIR		Directory to find cepstral variance vecotrs
	VARSCALEMASK		Filename mask for cepstral variance vecotrs
	VARSCALEFN		Filename of global variance scaling vector
	COMPRESSFACT	0.33	Amplitude compression factor for PLP
HLabel HParm	V1COMPAT	F	HTK V1 compatibility setting
HWave	NATURALREADORDER	F	Enable natural read order for binary files
HShell	NATURALWRITEORDER	F	Enable natural write order for binary files
HParm	USEPOWER	F	Use power not magnitude in fbank analysis
	NUMCEPS	12	Number of cepstral parameters
	CEPLIFTER	22	Cepstral liftering coefficient
	ENORMALISE	T	Normalise log energy
	ESCALE	0.1	Scale log energy
	SILFLOOR	50.0	Energy silence floor in dBs
	DELTAWINDOW	2	Delta window size
	ACCWINDOW	2	Acceleration window size
	VQTABLE	NULL	Name of VQ table
	SIMPLEDIFFS	F	Use simple differences for delta calculations
	RAWENERGY	T	Use raw energy
	AUDIOSIG	0	Audio signal number for remote control
	USESILDET	F	Enable speech/silence detector
	MEASURESIL	T	Measure background silence level
	OUTSILWARN	T	Print a warning message to `stdout` before measuring audio levels
	SPEECHTHRESH	9.0	Threshold for speech above silence level (in dB)
	SILENERGY	0.0	Average background noise level (in dB) - will normally be measured rather than supplied in configuration
	SPCSEQCOUNT	10	Window over which speech/silence decision reached
	SPCGLCHCOUNT	0	Maximum number of frames marked as silence in window which is classified as speech whilst expecting start of speech
	SILSEQCOUNT	100	Number of frames classified as silence needed to mark end of utterance
	SILGLCHCOUNT	2	Maximum number of frames marked as silence in window which is classified as speech whilst expecting silence
	SILMARGIN	40	Number of extra frames included before and after start and end of speech marks from the speech/silence detector
HLabel	STRIPTRIPHONES	F	Enable triphone stripping
	TRANSALT	0	Filter all but specified label alternative
	TRANSLEV	0	Filter all but specified label level
	LABELSQUOTE	NULL	Select method for quoting in label files
	SOURCELABEL	HTK	Source label format
	TARGETLABEL	HTK	Target label format
HMem	PROTECTSTAKS	F	Enable stack protection

Module	Name	Default	Description
HMODEL	CHKHMMDEFS	T	Check consistency of HMM defs
	SAVEBINARY	F	Save HMM defs in binary format
	KEEPDISTINCT	F	Keep orphan HMMs in distinct files
	SAVEGLOBOPTS	T	Save ~o with HMM defs
	ORPHANMACFILE	NULL	Last resort file for new macros
	HMMSETKIND	NULL	Kind of HMM Set
	ALLOWOTHERHMMS	T	Allow MMFs to contain HMM definitions which are not listed in the HMM List
	DISCRETELZERO	F	Map DLOGZERO to LZERO in output probability calculations
HNET	FORCECXTEXP	F	Force triphone context expansion to get model names (is overridden by ALLOWCXTEXP)
	FORCELEFTBI	F	Force left biphone context expansion to get model names ie. don't try triphone names
	FORCERIGHTBI	F	Force right biphone context expansion to get model names ie. don't try triphone names
	ALLOWCXTEXP	T	Allow context expansion to get model names
	ALLOWXWRDEXP	F	Allow context expansion across words
	FACTORLM	F	Factor language model likelihoods throughout words rather than applying all at transition into word. This can increase accuracy when pruning is tight and language model likelihoods are relatively high.
	CFWORDBOUNDARY	T	In word-internal triphone systems, context-free phones will be treated as word boundaries
HREC	FORCEOUT	F	Forces the most likely partial hypothesis to be used as the recognition result even when no token reaches the end of the network by the last frame of the utterance
HSHELL	ABORTONERR	F	Causes HError to abort rather than exit
	NONUMESCAPES	F	Prevent writing in 012 format
	MAXTRYOPEN	1	Maximum number of attempts which will be made to open the same file
	EXTENDFILENAMES	T	Support for extended filenames
HTRAIN	MAXCLUSTITER	10	Maximum number of cluster iterations
	MINCLUSTSIZE	3	Minimum number of elements in any one cluster
	BINARYACCFORMAT	T	Save accumulator files in binary format
HFB	HSKIPSTART	-1	Start of skip over region (debugging only)
	HSKIPEND	-1	End of skip over region (debugging only)
HADAPT	USEVAR	F	Compute variance transform
	ADPTSIL	T	Transform the silence
	BLOCKS	1	Number of blocks used in the block diagonal matrix implementation
	SAVEBINARY	F	Save HMMs/transforms in binary format
	OCCTHRESH	700	Minimum occupation before computing a regression class transform for a node
	TRACE	0	Trace setting
	HWAVEFILTER		Filter for waveform file input

Module	Name	Default	Description
HShell	HPARMFILTER		Filter for parameter file input
	HLANGMODFILTER		Filter for language model file input
	HMMLISTFILTER		Filter for HMM list file input
	HMMDEFFILTER		Filter for HMM definition file input
	HLABELFILTER		Filter for Label file input
	HNETFILTER		Filter for Network file input
	HDICTFILTER		Filter for Dictionary file input
	LGRAMFILTER		Filter for gram file input
	LWMAPFILTER		Filter for word map file input
	LCMAPFILTER		Filter for class map file input
	LMTEXTFILTER		Filter for text file input
	HWAVEOFILTER		Filter for waveform file output
	HPARMOFILTER		Filter for parameter file output
	HLANGMODOFILTER		Filter for language model file output
	HMMLISTOFILTER		Filter for HMM list file output
	HMMDEFOFILTER		Filter for HMM definition file output
	HLABELOFILTER		Filter for Label file output
	HNETOFILTER		Filter for Network file output
	HDICTOFILTER		Filter for Dictionary file output
	LGRAMOFILTER		Filter for gram file output
	LWMAPOFILTER		Filter for word map file output
	LCMAPOFILTER		Filter for class map file output
LModel	RAWMITFORMAT	F	Disable HTK escaping for LM tools
	USEINTID	F	Use 4 byte ID fields to save binary models
LWMap	INWMAPRAW	F	Disable HTK escaping for input word lists and maps
	OUTWMAPRAW	F	Disable HTK escaping for output word lists and maps
	STARTWORD	<s>	Set sentence start symbol
	ENDWORD	</s>	Set sentence end symbol
LCMap	INCMAPRAW	F	Disable HTK escaping for input class lists and maps
	OUTCMAPRAW	F	Disable HTK escaping for output class lists and maps
	UNKNOWNNAME	!!UNK	Set OOV class symbol
	UNKNOWNID	1	Set unknown symbol class ID
LPCalc	UNIFLOOR	1	Unigram floor count
	KRANGE	7	Good-Turing discounting range
	*n*G_CUTOFF	1	*n*-gram cutoff (eg. 2G_CUTOFF)
	DCTYPE	TG	Discounting type (TG for Turing-Good or ABS for Absolute
LGBase	CHECKORDER	F	Check N-gram ordering in files

18.2 Configuration Variables used in Tools

Module	Name	Default	Description
HCompV	UPDATEMEANS	F	Update means
	SAVEBINARY	F	Load/Save in binary format
	MINVARFLOOR	0.0	Minimum variance floor
HCopy	NSTREAMS	1	Number of streams
	SAVEASVQ	F	Save only the VQ indices
	SOURCEFORMAT	HTK	File format of source
	TARGETFORMAT	HTK	File format of target
	SOURCEKIND	ANON	Parameter kind of source
	TARGETKIND	ANON	Parameter kind of target
HERest	SAVEBINARY	F	Load/Save in binary format
	BINARYACFORMAT	T	Load/Save accumulators in binary format
	ALIGNMODELMMF		MMF file for alignment (2-model reest)
	ALIGNHMMLIST		Model list for alignment (2-model reest)
	ALIGNMODELDIR		Dir containing HMMs for alignment (2-model reest).
	ALIGNMODELEXT		Ext to be used with above Dir (2model-reest)
HEAdapt	SAVEBINARY	F	Load/Save in binary format
HHEd	TREEMERGE	T	After tree splitting, merge leaves
	TIEDMIXNAME	TM	Tied mixture base name
	APPLYVFLOOR	T	Apply variance floor to model set
	USELEAFSTATS	T	Use stats to obtain tied state pdf's
HParse	V1COMPAT	F	Enable compatibility with HTK V1.X
HResults	REFLEVEL	0	Label level to be used as reference
	TESTLEVEL	0	Label level to be scored
	STRIPCONTEXT	F	Strip triphone contexts
	IGNORECASE	F	If enabled, converts labels to uppercase
	NISTSCORE	F	Use NIST fomatting
	PHRASELABEL	SENT	Label for phrase level statistics
	PHONELABEL	WORD	Label for word level statistics
	SPEAKERMASK	NULL	If set then report on a per speaker basis
HVite	RECOUTPREFIX	NULL	Prefix for direct audio output name
	RECOUTSUFFIX	NULL	Suffix for direct audio output name
	SAVEBINARY	F	Save transforms as binary
HLStats	DISCOUNT	0.5	Discount constant for backoff bigrams
HList	AUDIOSIG	0	Audio signal numberfor remote control
	SOURCERATE	0.0	Sample rate of source in 100ns units
	TRACE	0	Trace setting

Table 18.2: Tool Specific Configuration Variables

Chapter 19

Error and Warning Codes

When a problem occurs in any HTK tool, either error or warning messages are printed.

If a warning occurs then a message is sent to standard output and execution continues. The format of this warning message is as follows:

```
WARNING [-nnnn] Function: 'Brief Explanation' in HTool
```

The message consists of four parts. On the first line is the tool name and the error number. Positive error numbers are fatal, whilst negative numbers are warnings and allow execution to continue. On the second line is the function in which the problem occurred and a brief textual explanation. The reason for sending warnings to standard output is so that they are synchronised with any trace output.

If an error occurs a number of error messages may be produced on standard error. Many of the functions in the HTK Library do not exit immediately when an error condition occurs, but instead print a message and return a failure value back to their calling function. This process may be repeated several times. When the HTK Tool that called the function receives the failure value, it exits the program with a fatal error message. Thus the displayed output has a typical format as follows:

```
  ERROR [+nnnn] FunctionA: 'Brief explanation'
  ERROR [+nnnn] FunctionB: 'Brief explanation'
  ERROR [+nnnn] FunctionC: 'Brief explanation'
 FATAL ERROR - Terminating program HTool
```

Error numbers in HTK are allocated on a module by module and tool by tool basis in blocks of 100 as shown by the table shown overleaf. Within each block of 100 numbers the first 20 (0 - 19) and the final 10 (90-99) are reserved for standard types of error which are common to all tools and library modules.

All other codes are module or tool specific.

19.1 Generic Errors

+??00 Initialisation failed
The initialisation procedure for the tool produced an error. This could be due to errors in the command line arguments or configuration file.

+??01 Facility not implemented
HTK does not support the operation requested.

+??05 Available memory exhausted
The operation requires more memory than is available.

+??06 Audio not available
The audio device is not available, either there is no driver for the current machine, the library was compiled with `NO_AUDIO` set or another process has exclusive access to the audio device.

HCopy	1000-1099	HShell	5000-5099
HList	1100-1199	HMem	5100-5199
HLEd	1200-1299	HMath	5200-5299
HLStats	1300-1399	HSigP	5300-5399
HDMan	1400-1499		
HSLab	1500-1599	HAudio	6000-6099
		HVQ	6100-6199
		HWave	6200-6299
HCompV	2000-2099	HParm	6300-6399
HInit	2100-2199	HLabel	6500-6599
HRest	2200-2299		
HERest	2300-2399	HGraf	6800-6899
HSmooth	2400-2499		
HQuant	2500-2599	HModel	7000-7099
HHEd	2600-2699	HTrain	7100-7199
HEAdapt	2700-2799	HUtil	7200-7299
		HFB	7300-7399
		HAdapt	7400-7499
HBuild	3000-3099		
HParse	3100-3199	HDict	8000-8099
HVite	3200-3299	HLM	8100-8199
HResults	3300-3399	HNet	8200-8299
HSGen	3400-3499	HRec	8500-8599
HLRescore	4000-4100	HLat	8600-8699
LCMap	15000-15099	LAdapt	16400-16499
LWMap	15100-15199	LPlex	16600-16699
LUtil	15200-15299	HLMCopy	16900-16999
LGBase	15300-15399	Cluster	17000-17099
LModel	15400-15499	LLink	17100-17199
LPCalc	15500-15599	LNewMap	17200-17299
LPMerge	15600-15699		

+??10 Cannot open file for reading
Specified file could not be opened for reading. The file may not exist or the filter through which it is read may not be set correctly.

+??11 Cannot open file for writing
Specified file could not be opened for writing. The directory may not exist or be writable by the user or the filter through which the file is written may not be set correctly.

+??13 Cannot read from file
Cannot read data from file. The file may have been truncated, incorrectly formatted or the filter process may have died.

+??14 Cannot write to file
Cannot write data to file. The file system is full or the filter process has died.

+??15 Required function parameter not set
You have called a library routine without setting one of the arguments.

+??16 Memory heap of incorrect type
Some library routines require you to pass them a heap of a particular type.

+??19 Command line syntax error
The command line is badly formed, refer to the manual or the command summary printed when the command is executed without arguments.

+??9? Sanity check failed
Several functions perform checks that structures are self consistent and that everything is functioning correctly. When these sanity checks fail they indicate the code is not functioning as intended. These errors should not occur and are not correctable by the user.

19.2 Summary of Errors by Tool and Module

HCopy

+1030 Non-existent part of file specified
HCopy needed to access a non-existent part of the input file. Check that the times are specified correctly, that the label file contains enough labels and that it corresponds to the data file.

±1031 Label file formatted incorrectly
HCopy is only able to properly copy label files with the same number of levels/alternatives. When using labels with multiple alternatives only the first one is used to determine segment boundaries.

+1032 Appending files of different type/size/rate
Files that are joined together must have the same parameter kind and sample rate.

−1089 ALIEN format set
Input/output format has been set to `ALIEN`, ensure that this was intended.

HList

HLEd

+1230 Edit script syntax error
The HLEd command script contains a syntax error, check the input script against the descriptions of each command in section 17.10 or obtained by running `HLEd -Q`.

±1231 Operation invalid
You have either exceeded HLEd limits on the number of boundaries that can be specified, tried to perform an operation on a non-existent level or tried to sort an auxiliary level into time order. None of these operations are supported.

+1232 Cannot find pronunciation
The dictionary does not contain a valid pronunciation (only occurs when attempting expansion from a dictionary).

−1289 ALIEN format set
Input/output format has been set to `ALIEN`, ensure that this was intended.

HLStats

+1328 Load/Make HMMSet failed
The model set could not be loaded due to either an error opening the file or the data within being inconsistent.

±1330 No operation specified
You have invoked HLStats but have not specified an operation to be performed.

−1389 ALIEN format set
Input format has been set to `ALIEN`, ensure that this was intended.

HDMan

±1430 Limit exceeded
HDMan has several built in limits on the number of different pronunciation, phones, contexts and command arguments. This error occurs when you try to exceed one of them.

±1431 Item not found
Could not find item for deletion. Check that it actually occurs in the dictionary.

±1450 Edit script file syntax error
The HDMan command script contains a syntax error, check the input script against the descriptions of each command in section 17.5 or obtained by running `HDMan -Q`.

±1451 Dictionary file syntax error
One of the input dictionaries contained a syntax error. Ensure that it is in a HTK readable form (see section 12.7).

±1452 Word out of order in dictionary error
Entries in the dictionary must be sorted into alphabetical (ASCII) order.

HSLab

−1589 ALIEN format set
Input/output format has been set to `ALIEN`, ensure that this was intended.

HCompV

+2020 HMM does not appear in HMMSet
Supplied HMM filename does not appear in HMMSet. Check correspondence between HMM filename and HMMSet.

+2021 Not enough data to calculate variance
There are not enough frames of data to evaluate a reliable estimate of variance. Use more data.

+2028 Load/Make HMMSet failed
The model set could not be loaded due to either an error opening the file or the data within being inconsistent.

+2030 Needs continuous models
HCompV can only operate on models with an HMM set kind of `PLAINHS` or `SHAREDHS`.

+2039 Speaker pattern matching failure
The specified speaker pattern could not be matched against a given untterance file name.

+2050 Data does not match HMM
An aspect of the data does not match the equivalent aspect in the HMMSet. Check the parameter kind of the data.

−2089 ALIEN format set
Input format has been set to `ALIEN`, ensure that this was intended.

HInit

+2120 Unknown update flag
Unknown flag set by `-u` option, use combinations of `tmvw`.

+2121 Too little data
Not enough data to reliably estimate parameters. Use more training data.

+2122 Segment with fewer frames than model states
Segment may be too short to be matched to model, do not use this segment for training.

+2123 Cannot mix covariance kind in a single mix
Covariance kind of all mixture components in any one state must be the same.

+2124 Bad covariance kind
Covariance kind of mixture component must be either `FULLC` or `DIAGC`.

+2125 No best mix found
The Viterbi mixture component allocation failed to find a most likely component with this data. Check that data is not corrupt and that parameter values produced by the initial uniform segmentation are reasonable.

+2126 No path through segment
The Viterbi segmentation failed to find a path through model with this data. Check that data is not corrupt and that a valid path exists through the model.

+2127 Zero occurrence count
Parameter has had no data assigned to it and cannot be updated. Ensure that each parameter can be estimated by using more training data or fewer parameters.

+2128 Load/Make HMMSet failed
The model set could not be loaded due to either an error opening the file or the data within being inconsistent.

+2129 HMM not found
HMM missing from HMMSet. Check that the HMMSet is complete and has not been corrupted.

+2150 Data does not match HMM
An aspect of the data does not match the equivalent aspect in the HMMSet. Check the parameter kind of the data.

+2170 Index out of range
Trying to access a mixture component or VQ index beyond the range in the current HMM.

−2189 ALIEN format set
Input format has been set to `ALIEN`, ensure that this was intended.

HRest

+2220 Unknown update flag
Unknown flag set by `-u` option, use combinations of `tmvw`.

+2221 Too few training examples
There are fewer training examples than the minimum set by the `-m` option (default 3). Either reduce the value specified by `-m` or use more training examples.

+2222 Zero occurrence count
Parameter has had no data assigned to it and cannot be updated. Ensure that each parameter can be estimated by using more training data or fewer parameters.

+2223 Floor too high
Mix weight floor has been set so high that the sum over all mixture components exceeds unity. Reduce the floor value.

−2225 Defunct Mix X.Y.Z
Not enough training data to re-estimate the covariance vector of mixture component Z in stream Y of state X. The weight of the mixture component is set to 0.0 and it will never recover even with further training.

+2226 No training data
None of the supplied training data could be used to re-estimate the model. Data may be corrupt or has been floored.

+2228 Load/Make HMMSet failed
The model set could not be loaded due to either an error opening the file or the data within being inconsistent.

+2250 Data does not match HMM
An aspect of the data does not match the equivalent aspect in the HMMSet. Check the parameter kind of the data.

−2289 ALIEN format set
Input format has been set to `ALIEN`, ensure that this was intended.

HERest

+2320 Unknown update flag
Unknown flag set by `-u` option, use combinations of `tmvw`.

+2321 Load/Make HMMSet failed
The model set could not be loaded due to either an error opening the file or the data within being inconsistent.

−2326 No transitions
No transition out of an emitting state, ensure that there is a transition path from beginning to end of model.

+2327 Floor too high
Mix weight floor has been set so high that the sum over all mixture components exceeds unity. Reduce the floor value.

+2328 No mixtures above floor
None of the mixture component weights are greater than the floor value, reduce the floor value.

−2330 Zero occurrence count
Parameter has had no data assigned to it and cannot be updated. Ensure that each parameter can be estimated by using more training data or fewer parameters.

−2331 Not enough training examples
Model was not updated as there were not enough training examples. Either reduce the minimum specified by `-m` or use more data.

−2389 ALIEN format set
Input format has been set to `ALIEN`, ensure that this was intended.

HSmooth

+2420 Unknown update flag
Unknown flag set by `-u` option, use combinations of `tmvw`.

+2421 Invalid HMM set kind
HSmooth can only be used if HMM set kind is either `DISCRETE` or `TIED`.

+2422 Too many monophones in list
HSmooth is limited to HMMSets containing fewer than 500 monophones.

+2423 Different number of states for smoothing
Monophones and context-dependent models have differing numbers of states.

−2424 No transitions
No transition out of an emitting state, ensure that there is a transition path from beginning to end of model.

+2425 Floor too high
Mix weight floor has been set so high that the sum over all mixture components exceeds unity. Reduce the floor value.

−2427 Zero occurrence count
Parameter has had no data assigned to it and cannot be updated. Ensure that each parameter can be estimated by using more training data or fewer parameters.

−2428 Not enough training examples
Model was not updated as there were not enough training examples. Either reduce the minimum specified by `-m` or use more data.

+2429 Load/Make HMMSet failed
The model set could not be loaded due to either an error opening the file or the data within being inconsistent.

HQuant

+2530 Stream widths invalid
The chosen stream widths are invalid. Check that these match the parameter kind and are specified correctly.

+2531 Data does not match codebook
Ensure that the parameter kind of the data matches that of the codebook being generated.

HHEd

+2628 Load/Make HMMSet failed
The model set could not be loaded due to either an error opening the file or the data within being inconsistent.

±2630 Tying null or different sized items
You have executed a tie command on items which do not have the appropriate structure or the structures are not matched. Ensure that the item list refers only to the items that you wish to tie together.

−2631 Performing operation on no items
The item list was empty, no operation is performed.

+2632 Command parameter invalid
The parameters to the command are invalid either because they refer to parts of the model that do not exist (for instance a state that does not appear in the model) or because they do not represent an acceptable value (for instance HMMSet kind is not `PLAINHS`, `SHAREDHS`, `TIEDHS` or `DISCRETEHS`).

+2634 Join parameters invalid or not set
Make sure than the join parameters (set by the `JO` command) are reasonable. In particular take care that the floor is low enough to ensure that when summed over all the mixture components the sum is below 1.0.

+2635 Cannot find matching item
Search for specified item was unsuccessful. When this occurs with the `CL` or `MT` commands ensure that the appropriate monophone/biphone models are in the current HMMSet.

−2637 Small gConst
A small gConst indicates a very low variance in that particular Gaussian. This could be indicative of over-training of the models.

−2638 No typical state
When tying states together a search is performed for the distribution with largest variance and all tied states share this distribution. If this cannot be found the first in the list will be used instead.

−2639 Long macro name
In general macro names should not exceed 20 characters in length.

+2640 Not implemented
You have asked HHEd to perform a function that is not implemented.

+2641 Invalid stream split
The specified number/width of the streams does not agree with the parameter kind/vector size of the models.

+2650 Edit script syntax error
The HHEd command script contains a syntax error, check the input script against the descriptions of each command in section 17.8 or obtained by running `HHEd -Q`.

+2651 Command range error
The value specified in the command script is out of range. Ensure that the specified state exists and the the value given is valid.

±2655 Stats file load error
Either loading occupation statistics for the second time or executing an operation that needs the statistics loaded without loading them.

+2660 Trees file syntax error
The trees file format did not correspond to that expected. Ensure that the file is complete and has not been corrupted.

+2661 Trees file macro/question not recognised
The question or macro referred to does not exist. Ensure that the file is complete and has not been corrupted.

+2662 Trying to sythesize for unknown model
There is no tree or prototype model for the new context. Ensure that a tree has been constructed for the base phone.

±2663 Invalid types to tree cluster
Tree clustering will only work for single Gaussian diagonal covariance untied models of similar topology.

HEAdapt

+2720 Invalid update flag
Means (m) and variances (v) can be updated, check that the update flags are correct.

−2721 No update requested
Mean transformation will be updated even though none specified.

+2728 Load/Make HMMSet failed
The model set could not be loaded due to either an error opening the file or the data within being inconsistent.

+2730 Invalid HMM kind
MLLR adaptation only available for PLAIN or SHARED systems.

+2731 Invalid HMM kind
MLLR adaptation only currently available for single stream data.

−2740 Invalid Adaptation Mode
Incremental MAP adaptation is not supported, use in static mode.

−2741 Invalid Adaptation Mode
MLLR and MAP adaptation does not support incremental adaptation. If incremental MLLR and static MAP adaptation is required, perform the steps separately.

−2789 Alien format set
Input format has been set to ALIEN, ensure that this was intended.

HBuild

±3030 Mismatch between command line and language model
Ensure that the `!ENTER` and `!EXIT` words are correctly defined and that the supplied files are of the appropriate type.

±3031 Unknown word
Ensure that the word list corresponds to the language model/lattice supplied.

HParse

±3130 Variable not defined
You have referenced a network that has not yet been defined. Check that all networks are defined before they are referenced.

±3131 Loop or word expansion error
There is either a mismatch between the `WD_BEGIN WD_END` pairs or a triphone loop is badly formed.

±3132 Dictionary error
When generating a dictionary a word exceeded the maximum number of phones, a word occurred twice or no dictionary was produced.

±3150 Syntax error in HParse file
The HParse network definition contains a syntax error, check the input file against the network description in section 17.15.

HVite

+3228 Load/Make HMMSet failed
The model set could not be loaded due to either an error opening the file or the data within being inconsistent.

±3230 Unsupported operation
HVite is not able to perform the operation requested

±3231 Data does not match HMMs
There is a mismatch between the data file and the HMMSet. Ensure that the data is parameterised in the correct format and the configuration parameters match those used during training.

+3232 MMF Load Error
The HMMSet does not contain a well-formed regression class tree.

−3289 ALIEN format set
Input/output format has been set to `ALIEN`, ensure that this was intended.

HResults

−3330 Empty file
The file was empty and will be skipped.

+3331 Unknown label
The label did not appear in the list supplied to HResults. This error will only occur if calculating confusion matrices so normally the contents of the word list file will have no effect on results.

+3332 Too many labels
HResults will only generate confusion statistics for a small number of labels.

±3333 Cannot calculate word spot results
When calculating word spotting results the label files need to have both times and scores present.

−3389 ALIEN format set
Input format has been set to `ALIEN`, ensure that this was intended.

HSGen

−3420 Network malformed
The word network is malformed. The information in a node (word and following arcs) is set incorrectly.

HLRescore

−4089 ALIEN format set
Input/output format has been set to `ALIEN`, ensure that this was intended.

HShell

+5020 Command line processing error

+5021 Command line argument type error

+5022 Command line argument range error
The command line is badly formed. Ensure that it matches the syntax and values expected by the command (check the manual page or the syntax obtained by running HTool without any arguments).

+5050 Configuration file format error
HShell was unable to parse the configuration. Check that it is of the format described in section 4.3.

+5051 Script file format error
Check that the script file is just a list of file names and that if any file names are quoted that the quotes occur in pairs.

+5070 Module version syntax error
A module registered with HShell with an incorrectly formatted version string (which should be of the form `"!HVER!HModule: Vers.str [WHO DD/MM/YY]"`).

+5071 Too many configuration parameters
The size of the buffer used by one of the tools or modules to read its configuration parameters was exceeded. Either reduce the total number of configuration parameters in the file or make more of then specific to their particular module rather than global.

+5072 Configuration parameter of wrong type
The configuration parameter is of the wrong type. Check that its type agrees with that shown in chapter 18.

+5073 Configuration parameter out of range
The configuration parameter is out of range.

HMem

+5170 Heap parameters invalid
You have tried to create a heap with unreasonable parameters. Adjust these so that the growth factor is positive and the initial block size is no larger than the maximum. For `MSTAK` the element size should be 1.

+5171 Heap not found
The specified heap could not be found, ensure that it has not been deleted or memory overwritten.

+5172 Heap does not support operation
The heap is of the wrong type to support the requested operation. In particular it is not possible to `Reset` or `Delete` a `CHEAP`.

+5173 Wrong element size for MHEAP
You have tried to allocate an item of the wrong size from a `MHEAP`. All items on a `MHEAP` must be of the same size.

+5174 Heap not initialised
You have tried to allocate an item on a heap that has not yet been created. Ensure that `CreateHeap` is called to initialise the heap before any items are allocated from it.

+5175 Freeing unseen item
You have tried to free an item from the wrong heap. This can occur if the wrong heap is specified, the item pointer has been corrupted or the item has already been freed implicitly by a `Reset/DeleteHeap` call.

HMath

+5220 Singular covariance matrix
The covariance matrix was not invertible. This may indicate a lack of training data or linearly dependent parameters.

+5270 Size mismatch
The input parameters were of incompatible sizes.

+5271 Log of negative
Result would be logarithm of a negative number.

HSigP

+5320 No results for WaveToLPC
Call did not include Vectors for the results.

+5321 Vector size mismatch
Input vectors were of mismatched sizes.

−5322 Clamped samples during zero mean
During a zero mean operation samples were clipped as they were outside the allowable range.

HAudio

+6020 Replay buffer not active
Attempt to access a replay buffer when one was not attached.

+6021 Cannot StartAudio without measuring silence
An attempt was made to start audio input through the silence detector without first measuring or supplying the background silence values.

+6070 Audio frame size/rate invalid
The choice of frame period and window duration are invalid. Check both these and the sample rate.

−6071 Setting speech threshold below silence
The thresholds used in the speech detector have been set so that the threshold for detecting speech is set below that of detecting silence.

HVQ

+6150 VQ file format error
The VQ file was incorrectly formatted. Ensure that the file is complete and has not been corrupted.

+6151 VQ file range error
A value from the VQ file was out of range. Ensure that the file is complete and has not been corrupted.

+6170 Magic number mismatch
The VQ magic number (normally based on parameter kind) does not match that expected. Check that the parameter kind used to quantise the data and create the VQ table matches the current parameter kind.

+6171 VQ table already exists
All VQ tables must have distinct names. This error will occur if you try to create or load a VQ table with the same name as one already loaded.

+6172 Invalid covariance kind
Entries in VQ tables must have either `NULLC`, `FULLC` or `INVDIAGC` covariance kind.

+6173 Node not in table
A node was missing from the VQ table. Ensure that the VQ table was properly created or that the file was complete.

+6174 Stream codebook mismatch
The number or size of streams in the VQ table does not match that requested.

HWave

+6220 Cannot fseek/ftell
Unless the wave file is read through a pipe fseek and ftell are expected to work correctly so that HWave can calculate the file size. If this error occurs when using an input pipe, supply the number of samples in the file using the configuration variable `NSAMPLES`.

+6221 File appears to be a infinite
HWave cannot determine the size of the file.

+6230 Config parameter not set
A necessary configuration parameter has not been set. Determine the correct value and place this in the configuration file before re-invoking the tool.

+6250 Premature end of header
HWave could not read the complete file header.

+6251 Header contains invalid data
HWave was unable to successfully parse the header. The header is invalid, of the wrong type or be a variation that HWave does not handle.

+6252 Header missing essential data
The header was missing a piece of information necessary for HWave to load the file. Check the processing of the input file and re-process if necessary.

+6253 Premature end of data
The file ended before all the data was read correctly. Check that the file is complete, has not been corrupted and where necessary `NSAMPLES` is set correctly.

+6254 Data formated incorrectly
The data could not be decoded properly. Check that the file was complete and processed correctly.

+6270 File format invalid
The file format is not valid for the operation requested.

+6271 Attempt to read outside file
You have tried to read a sample outside of the waveform file.

HParm

+6320 Configuration mismatch
The data file does not match the configuration. Check the configuration file is correct.

+6321 Invalid parameter kind
Parameter kind is not valid. Check the configuration file.

+6322 Conversion not possible
The specified conversion is not possible. Check the configuration is correct and re-code the data from waveform files if necessary.

+6323 Audio error
An audio error has been detected. Check the HAudio configuration and the audio device.

+6324 Buffer not initialised
Ensure that the buffer is used in the correct manner.

+6325 Silence detection failed
The silence detector was not initialised correctly before use.

+6328 Load/Make HMMSet failed
The model set could not be loaded due to either an error opening the file or the data within being inconsistent.

+6350 CRC error
The CRC does not match that of the data. Check the data file is complete and has not been corrupted.

−6351 Byte swapping not possible
HParm will attempt to byte swap parameter files but this may not work if the floating point representation of the machine that generated the file is different from that which is reading it.

+6352 File too short to parameterise
The file does not contain enough data to produce a single observation. Check the file is complete and not corrupt. If it is, it should be discarded.

+6370 Unknown parameter kind
The specified parameter kind is not recognised. Refer to section 5.18 for a list of allowable parameter kinds and qualifiers.

+6371 Invalid parameters for coding
The chosen parameters are not valid for coding. Choose different ones.

+6372 Stream widths not valid
Cannot split the data into the specified number of streams. Check that the parameter kind is correct and matches any models used.

+6373 Buffer/observation mismatch
The observation parameter kind should match that of the input buffer. Check that the configuration parameter kind is correct and matches that of any models used.

+6374 Buffer size too small for window
Calculation of delta parameters requires a window larger than the buffer size chosen. Increase the size of the buffer.

+6375 Frame not in buffer
An attempt was made to access a frame that does not appear in the buffer. Make sure that the file actually contains the specified frame.

+6376 Mean/Variance normalisation failed
The mean or variance normalisation vector from the file specified by the normalisation dir and mask cannot be applied. Make sure the file format is correct and the vectors are of the right dimension.

HLabel

+6520 MLF index out of range
An attempt was made to access an MLF that has not been loaded or to load too many MLFs.

+6521 fseek/ftell not possible
HLabel needs random access to MLFs. This error is generated when this is not possible (for instance if access is via a pipe).

+6550 HTK format error

+6551 MLF format error

+6552 TIMIT format error

±6553 ESPS format error

+6554 SCRIBE format error
A label file was formatted incorrectly. Label file formats are described in chapter 6.

+6570 Level out of range
Attempted to access a non-existent label level. Check that the correct label file has been loaded.

+6571 Label out of range
Attempted to access a non-existent label. Check that the correct label file has been loaded and that the correct level is being accessed.

+6572 Invalid format
The specified file format is not valid for the particular operation.

HMODEL

+7020 Cannot find physical HMM
No physical HMM exists for a particular logical model. Check that the HMMSet was loaded or created correctly.

+7021 INVDIAG internal format
Attempts to load or save models with `INVDIAG` covariance kind will fail as this is a purely internal model format.

±7023 varFloor should be variance floor
HMODEL reserves the macro name `varFloorN` as the variance floor for stream `N`. These should be variance macros (type `v`) of the correct size for the particular stream.

+7024 Variance tending to 0.0
A variance has become too low. Start using a variance floor or increase the amount of training data.

+7025 Bad covariance kind
The particular functionality does not support the covariance kind of the mixture component.

+7030 HMM set incomplete or inconsistent
The HMMSet contained missing or inconsistent data. Check that the file is complete and has not been corrupted.

+7031 HMM parameters inconsistent
Some model parameters were inconsistent. Check that the file is complete and has not been corrupted.

±7032 Option mismatch
All HMMs in a particular set must have consistent options.

+7035 Unknown macro
Macro does not exist. Check that the name is correct and appears in the HMMSet.

+7036 Duplicate macro
Attempted to create a macro with the same name as one already present. Choose a different name.

+7037 Invalid macro
Macro had invalid type. See section 7.3 describes the allowable macro types.

+7050 Model file format error

+7060 HMM List format error
The file was formated incorrectly. Check the file is complete and has not been corrupted.

+7070 Invalid HMM kind
Invalid HMMSet kind. Check that this is specified correctly.

+7071 Observation not compatible with HMMSet
Attempted to calculate an observation likelihood for an observation not compatible with the HMMSet. Check that the parameter kind is set correctly.

HTRAIN

+7120 Clustering failed
Almost certainly due to a lack of data, reduce the number of clusters requested or increase amount of data.

+7150 Accumulator file format error
Cannot read an item from an accumulator file. Check that file is complete and not corrupted.

+7170 Unsupported covariance kind
Covariance kind must be either `FULLC`, `DIAGC` or `INVDIAGC`.

+7171 Item out of range
Attempt made to access data beyond expected range. Check that the item number is correct.

+7172 Tree size must be power of 2
Requested codebook size must be a power of 2 when using tree based clustering.

−7173 Segment empty
Empty data segment in file. Check that file has not become corrupted and that the start and end segment times are correct.

HUTIL

+7220 HMMSet empty
A scan was initiated for a HMMSet with no members.

+7230 Item list parse error
The item list syntax was incorrect. Check the item list specification in section 17.8.

+7231 Item list type error
Each item in a particular list should be of the same type and size.

+7250 Stats file format error
Stats file is of wrong format. Note the format of the stats file has changed in HTK_V2.0 and old files will need converting to the new format.

+7251 Stats file model error
A model name encountered in the stats file is invalid check that the model set corresponds to that used to generate the stats file and that the stats file is complete and has not been corrupted.

+7270 Accessing non-existent macro
Attempt to perform operation on non-existent macro.

+7271 Member id out of range
Attempt to perform set operation on out of range member.

HFB

+7321 Unknown model
Model in HMM List not found in HMMSet, check that the correct HMM List is being used.

+7322 Invalid output probability
Mixture component probability has not been set. This should not occur in normal use.

+7323 Beta prune failed on taper
Utterance is possibly too short for minimum duration of model sequence. Check transcription.

−7324 No path through utterance
No path was found on the beta training pass, relax the pruning threshold.

−7325 Empty label file
No labels found in label file, check label file.

+7326 Single-pass retraining data mismatch
Paired training files must contain the same number of observations. Use original data to re-parameterise.

±7332 HMM with unreachable states
HMM has an unreachable state, check transition matrix.

−7333 Transition matrix with discontinuity
Check transition matrix.

+7350 Data does not match HMM
An aspect of the data does not match the equivalent aspect in the HMMSet. Check the parameter kind of the data.

HAdapt

±7410 TMF load error
Can't open the input TMF.

−7420 Can't calculate the auxilliary function
Only possible with mean and variance transforms, check update flags.

+7421 MMF load error
MMF does not contain a HMM identifier, use HHEd to generate one.

+7422 MMF load error
MMF does not contain a regression class, use HHEd to generate one

+7425 Mismatch number of Gaussian components
The number of Gaussian components found for a regression class does not much the number expected. Check the MMF is not corrupted.

+7430 MMF load error
Can't find the regression class tree.

+7431 Can't add regression class accumulate
Problem adding a frame of accumulation at the component level.

±7440 Symbol not found
Could not read in symbol.

+7450 Invalid HMM kind
Can only adapt PLAIN and SHARED systems.

+7460 Can't create block transformation matrix
Check number of blocks compatible with vector size.

+7470 MAP weight hook is NULL

HDict

+8050 Dictionary file format error
The dictionary file is not correctly formatted. Section 12.7 describes the HTK dictionary file format.

HLM

+8150 LM syntax error
The language model file was formatted incorrectly. Check the file is complete and has not been corrupted.

±8151 LM range error
The specified value(s) for the language model probability are not valid. Check the input files are correct.

HNet

+8220 No such word
The specified word does not exist or does not have a valid pronunciation.

−8221 Duplicate pronunciations removed
During network generations duplicate identical pronunciations of the same word are removed.

+8230 Contexts not consistent
HNet can only deal with the standard HTK method for specifying context `left-phone+right` and will only allow context free phones if they are context independent and only form part of the word. This may be indicative of an inconsistency between the symbols in the dictionary and the hmms as defined. There may be a model/phone in the dictionary that has not been defined in the HMM list or may not have a corresponding model. See also section 12.8 on context expansion.

+8231 No such model
A particular model could not be found. Make sure that the network is being expanded in the correct fashion and then ensure that your HMM list will cover all required contexts.

+8232 Lattice badly formed
Could not convert lattice to network. The lattice should have a single well defined start and a single well defined end. When cross word expansion is being performed the number of `!NULL` words that can be concatenated in a string is limited.

+8250 Lattice format error
The lattice file is formatted incorrectly. Ensure that the lattice is of the format described in chapter 20.

+8251 Lattice file data error
The value specified in the lattice file is invalid.

+8252 Lattice file with multiple start/end nodes
A lattice should have only one well defined start node and one well defined end node.

+8253 Lattice with invalid sub lattices
The sub lattices referred to by the main lattices are malformed.

HRec

±8520 Invalid HMM
One of the HMMs in the network is invalid. Check that the HMMSet has been correctly initialised.

+8521 Network structure invalid
The network is incorrectly structured. Take care to avoid loops that can be traversed without consuming observations (this may occur if you introduce any 'tee' words in which all the models making up that word contain tee-transitions). Also ensure that the recogniser and the network have been created and initialised correctly.

+8522 Lattice structure invalid
The lattice was incorrectly formed. Ensure that the lattice was created properly.

±8570 Recogniser not initialised correctly
Ensure the recogniser is initialised and used correctly.

+8571 Data does not match HMMs
The observation does not match the HMM structure. Check the parameter kind of the data and ensure that the data is matched to the HMMs.

HLat

8621 Lattice incompatible with dictionary
The lattice refers to a pronunciation variant (filed `v=`) that doesn't exist in the current dictionary.

±8622 Lattice structure invalid
The lattice does not meet the requirements for some operation. All lattices must have unique start and end nodes and for many operations the lattices need to be acyclic (i.e. be a Directed Acyclic Graph).

8623 Start or end word not found
The specified lattice start or end word could not be found in the dictionary.

8624 Lattice end node label invalid
The lattice end node must either be labelled with `!NULL` or the specified end word (default: `!SENT_END`)

8690 Lattice operation not supported
The requested operation is not supported, yet.

8691 Lattice processing sanity check faile
During processing an internal sanity check failed. This should never happen..

HGraf

+6870 X11 error
Ensure that the `DISPLAY` variable is set and that the X11 window system is configured correctly.

LCMap

+15050 Unlikely num map entries[`n`] in `XYZ`
A negative or infeasibly large number of class map entries have been specified.

+15051 ReadMapHeader: UNKxxx configs must be set for hdrless map
There is no header on the map so you must set UNKNOWNID and UNKNOWNNAME.

+15052 No name in `XYZ`
No NAME header in class map.

+15053 Unknown escmode `XYZ` in `XYZ`
ESCMODE header must specify either HTK or RAW.

+15054 Class name `XYZ` duplicate in `XYZ`
Two classes in the class map have the same name, which is not allowed.

+15055 Bad index `n` for class `XYZ` in `XYZ`
A class index less than 1 or greater than or equal to BASEWORDNDX (defined at compile time in LWMap - default is 65536) was found in the class map. If you need more than BASEWORDNDX classes then you must recompile HTK with a new base word value.

+15056 Number of entries = `n` for class `XYZ` in `XYZ`
There must be at least one member in each class - empty classes are not allowed.

+15057 Bad type `XYZ` for class `XYZ` in `XYZ`
Classes must be defined using either IN or NOTIN.

LWMap

+15150 Word list/word map file format error
Check that the word list/word map file is correctly formatted.

+15151 Unlikely num map entries[`n`] in `XYZ`
A negative or infeasibly large number of word map entries have been specified.

+15152 No NAME header in `XYZ`
No NAME header in word map.

+15153 No SEQNO header in `XYZ`
No SEQNO header in word map.

+15154 Unknown escmode `XYZ` in `XYZ`
ESCMODE header must specify either HTK or RAW.

+15155 Word name `XYZ` is duplicated in `XYZ`
There are duplicate words in the word map, which is not allowed.

LUtil

+15250 Header format error
Ensure that word maps and/or n-gram files used by the program start with the appropriate header.

LGBase

+15330 n-gram file consistency check failure
The n-gram file is incompatible with other resources used by the program.

+15340 File `XYZ` is `n`-gram but inset is `n`-gram
The specified input gram file is not of the expected gram size.

+15341 Requested N[`n`] greater than gram size [`n`]
An n-gram was requested which was larger than any of those supplied in the input files.

+15345 n-grams out of order
The input gram file is not correctly sorted.

+15350 n-gram file format error
Ensure that n-gram files used by the program are formatted correctly and start with the appropriate header.

LMODEL

+15420 Cannot find n-gram component
The internal structure of the language model is corrupted. This error is usually caused when an n-gram (a, b, c) is encountered without the presence of n-gram (a, b).

+15430 Incompatible probability kind in conversion
The currently used language model does not allow the required conversion operation. This error is caused by attempting to prune a model stored in the ultra file format.

+15440 Cannot prune models in ultra format
Pruning of language models stored in *ultra* file format is not supported.

+15445 Word ID size error
Language models with vocabularies of over 65,536 words require the use of larger word identifiers. This is a sanity check error.

−15450 Word `XYZ` not in unigrams - skipping n-gram.
There should be a unigram count for each word in other length grams.

+15450 Language model file format error
The language model file is formatted incorrectly. Check the file is complete and has not been corrupted.

−15451 Extraneous line warning
Extra lines were found on the end of a file and are being ignored.

−15460 Model order reduced
Due to the effects of pruning the model order is automatically reduced.

LPCALC

+15520 Unable to find FLEntry to attach
Indicates that the LM data structures are corrupt. This is normally caused by NGram files which have not been sorted.

+15525 Attempt to overwrite entries when attaching
Indicates that the LM structure is corrupt. Ensure that the word map file used is suitable for decoding the NGram database files.

−15540 `n`-gram cutoff out of range
An inapplicable cutoff was ignored.

+15540 Pruning error
The pruning parameters specified are not compatible with the parameters of the language model.

LPMERGE

+15620 Unable to find word in any model
Indicates that the target model vocabulary contains a word which cannot be found in any of the source models.

LPLEX

+16620 symbol `XYZ` not in word list
The sentence start symbol, sentence end symbol and OOV symbol (only if OOVs are to be included in the perplexity calculation) must be in the language model's vocabulary. Note that the vocabulary list is either specified with the `-w` option or is implicitly derived from the language model.

+16625 Unable to find word `XYZ` in any model
Ensure that all words in the vocabulary list specified with the `-w` option are present in at least one of the language models.

+16630 Maximum number of unique OOVs reached
Too many OOVs encountered in the input text.

−16635 Transcription file `fn` is empty
The label file does not contain any words.

−16640 Word too long, will be split: `XYZ`
The word read from the input stream is of over 200 characters.

−16645 Text buffer size exceeded (`n`)
The maximum number of words allowed in a single utterance has been reached.

+16650 Maximum utterance length in a label file exceeded (limit is compiled to be `n` tokens)
No label file utterance end has been encountered within `n` tokens – perhaps this is a text file and you forgot to pass the `-t` option?

HLMCopy

+16920 Maximum number of phones reached
When HLMCopy is used to copy dictionaries, the target dictionary's phone table is composed by combining the phone tables of all source dictionaries. Check that the number of different phones resulting from combining the phone tables of the source dictionaries does not exceed the internally set limit.

+16930 Cannot find definition for word `XYZ`
When copying dictionaries, ensure that each word in the vocabulary list occurs in at least one source dictionary.

Cluster

+17050 Word `XYZ` found in class map but not in word map
All words in the class map must be found in the word map too.

−17051 Unknown word token `XYZ` was explicitly given with -u, but does not occur in the word map
This warning appears if you specify an unknown word token which is not found in the word map.

+17051 Token not found in word list
Sentence start, end and unknown (if used) tokens must be found in the word map.

+17052 Not all words were assigned to classes
A classmap was imported which did not include all words in the word map.

−17053 Word `XYZ` is in word map but not in any gram files
The stated word will remain in whichever class it is already in - either as defaulted to or supplied via the input class map.

Chapter 20

HTK Standard Lattice Format (SLF)

20.1 SLF Files

Lattices in HTK are used for storing multiple hypotheses from the output of a speech recogniser and for specifying finite state syntax networks for recognition. The HTK standard lattice format (SLF) is designed to be extensible and to be able to serve a variety of purposes. However, in order to facilitate the transfer of lattices, it incorporates a core set of common features.

An SLF file can contain zero or more sub-lattices followed by a main lattice. Sub-lattices are used for defining sub-networks prior to their use in subsequent sub-lattices or the main lattice. They are identified by the presence of a `SUBLAT` field and they are terminated by a single period on a line by itself. Sub-lattices offer a convenient way to structure finite state grammar networks. They are never used in the output word lattices generated by a decoder. Some lattice processing operations like lattice pruning or expansion will destroy the sub-lattice structure, i.e. expand all sub-lattice references and generate one unstructured lattice.

A lattice consists of optional header information followed by a sequence of node definitions and a sequence of link (arc) definitions. Nodes and links are numbered and the first definition line must give the total number of each.

Each link represents a word instance occurring between two nodes, however for more compact storage the nodes often hold the word labels since these are frequently common to all words entering a node (the node effectively represents the end of several word instances). This is also used in lattices representing word-level networks where each node is a word end, and each arc is a word transition.

Each node may optionally be labelled with a word hypothesis and with a time. Each link has a start and end node number and may optionally be labelled with a word hypothesis (including the pronunciation variant, acoustic score and segmentation of the word hypothesis) and a language model score.

The lattice must have exactly one start node (no incoming arcs) and one end node (no outgoing arcs). The special word identifier `!NULL` can be used for the start and end node if necessary.

20.2 Format

The format is designed to allow optional information that at its most detailed gives full identity, alignment and score (log likelihood) information at the word and phone level to allow calculation of the alignment and likelihood of an individual hypothesis. However, without scores or times the lattice is just a word graph. The format is designed to be extensible. Further field names can be defined to allow arbitrary information to be added to the lattice without making the resulting file unreadable by others.

The lattices are stored in a text file as a series of fields that form two blocks:

- A header, specifying general information about the lattice.
- The node and link definitions.

Either block may contain comment lines, for which the first character is a '#' and the rest of the line is ignored.

All non-comment lines consist of fields, separated by white space. Fields consist of an alphanumeric field name, followed by a delimiter (the character '=' or '~') and a (possibly "quoted") field value. Single character field names are reserved for fields defined in the specification and single character abbreviations may be used for many of the fields defined below. Field values can be specified either as normal text (e.g. `a=-318.31`) or in a binary representation if the '=' character is replaced by '~'. The binary representation consists of a 4-byte floating point number (IEEE 754) or a 4-byte integer number stored in big-endian byte order by default (see section 4.9 for a discussion of different byte-orders in HTK).

The convention used to define the current field names is that lower case is used for optional fields and upper case is used for required fields. The meaning of field names can be dependent on the context in which they appear.

The header must include a field specifying which utterance was used to generate the lattice and a field specifying the version of the lattice specification used. The header is terminated by a line which defines the number of nodes and links in the lattice.

The node definitions are optional but if included each node definition consists of a single line which specifies the node number followed by optional fields that may (for instance) define the time of the node or the word hypothesis ending at that node.

The link definitions are required and each link definition consists of a single line which specifies the link number as well as the start and end node numbers that it connects to and optionally other information about the link such as the word identity and language model score. If word identity information is not present in node definitions then it must appear in link definitions.

20.3 Syntax

The following rules define the syntax of an SLF lattice. Any unrecognised fields will be ignored and no user defined fields may share the first character with pre-defined field names. The syntax specification below employs the modified BNF notation used in section 7.11. For the node and arc field names only the abbreviated names are given and only the text format is documented in the syntax.

```
latticedef = laticehead
             lattice { lattice }


latticehead = "VERSION=" number
              "UTTERANCE=" string
              "SUBLAT=" string
              { "vocab=" string | "hmms=" string | "lmname=" string |
                "wdpenalty=" floatnumber | "lmscale=" floatnumber |
                "acscale=" floatnumber | "base=" floatnumber | "tscale=" floatnumber }

lattice = sizespec
          { node }
          { arc }

sizespec = "N=" intnumber "L=" intnumber

node = "I=" intnumber
       { "t=" floatnumber | "W=" string  |
         "s=" string | "L=" string | "v=" intnumber }

arc = "J=" intnumber
      "S=" intnumber
      "E=" intnumber
      { "a=" floatnumber  | "l=" floatnumber  | "a=" floatnumber  | "r=" floatnumber  |
        "W=" string | "v=" intnumber | "d=" segments }
```

```
segments =  ":" segment {segment}
segment =  string [ "," floatnumber [ "," floatnumber ]] ":"
```

20.4 Field Types

The currently defined fields are as follows:-

```
  Field        abbr o|c Description

Header fields
  VERSION=%s     V  o  Lattice specification adhered to
  UTTERANCE=%s   U  o  Utterance identifier
  SUBLAT=%s      S  o  Sub-lattice name
  acscale=%f        o  Scaling factor for acoustic likelihoods
  tscale=%f         o  Scaling factor for times (default 1.0, i.e.\ seconds)
  base=%f           o  LogBase for Likelihoods (0.0 not logs, default base e)
  lmname=%s         o  Name of Language model
  lmscale=%f        o  Scaling factor for language model
  wdpenalty=%f      o  Word insertion penalty

Lattice Size fields
  NODES=%d       N  c  Number of nodes in lattice
  LINKS=%d       L  c  Number of links in lattice

Node Fields
  I=%d                 Node identifier.  Starts node information
  time=%f        t  o  Time from start of utterance (in seconds)
  WORD=%s        W wc  Word (If lattice labels nodes rather that links)
  L=%s             wc  Substitute named sub-lattice for this node
  var=%d         v wo  Pronunciation variant number
  s=%s           s  o  Semantic Tag

Link Fields
  J=%d                 Link identifier.  Starts link information
  START=%d       S  c  Start node number (of the link)
  END=%d         E  c  End node number (of the link)
  WORD=%s        W wc  Word (If lattice labels links rather that nodes)
  var=%d         v wo  Pronunciation variant number
  div=%s         d wo  Segmentation (modelname, duration, likelihood) triples
  acoustic=%f    a wo  Acoustic likelihood of link
  language=%f    l  o  General language model likelihood of link
  r=%f           r  o  Pronunciation probability

Note: The word identity (and associated 'w' fields var,div and acoustic) must
      appear on either the link or the end node.

      abbr is a possible single character abbreviation for the field name
      o|c indicates whether field is optional or compulsory.
```

20.5 Example SLF file

The following is a real lattice (generated by the HTK Switchboard Large Vocabulary System with a 54k dictionary and a word fourgram LM) with word labels occurring on the end nodes of the links.

Note that the `!SENT_SENT` and `!SENT_END` "words" model initial and final silence.

```
VERSION=1.0
UTTERANCE=s22-0017-A_0017Af-s22_000070_000157.plp
lmname=/home/solveb/hub5/lib/lang/fgintcat_54khub500.txt
```

```
lmscale=12.00  wdpenalty=-10.00
vocab=/home/solveb/hub5/lib/dicts/54khub500v3.lvx.dct
N=32   L=45
I=0    t=0.00  W=!NULL
I=1    t=0.05  W=!SENT_START         v=1
I=2    t=0.05  W=!SENT_START         v=1
I=3    t=0.15  W=!SENT_START         v=1
I=4    t=0.15  W=!SENT_START         v=1
I=5    t=0.19  W=HOW                 v=1
I=6    t=0.29  W=UM                  v=1
I=7    t=0.29  W=M                   v=1
I=8    t=0.29  W=HUM                 v=1
I=9    t=0.70  W=OH                  v=1
I=10   t=0.70  W=O                   v=1
I=11   t=0.70  W=KOMO                v=1
I=12   t=0.70  W=COMO                v=1
I=13   t=0.70  W=CUOMO               v=1
I=14   t=0.70  W=HELLO               v=1
I=15   t=0.70  W=OH                  v=1
I=16   t=0.70  W=LOW                 v=1
I=17   t=0.71  W=HELLO               v=1
I=18   t=0.72  W=HELLO               v=1
I=19   t=0.72  W=HELLO               v=1
I=20   t=0.72  W=HELLO               v=1
I=21   t=0.73  W=CUOMO               v=1
I=22   t=0.73  W=HELLO               v=1
I=23   t=0.77  W=I                   v=1
I=24   t=0.78  W=I'M                 v=1
I=25   t=0.78  W=TO                  v=1
I=26   t=0.78  W=AND                 v=1
I=27   t=0.78  W=THERE               v=1
I=28   t=0.79  W=YEAH                v=1
I=29   t=0.80  W=IS                  v=1
I=30   t=0.88  W=!SENT_END           v=1
I=31   t=0.88  W=!NULL
J=0     S=0    E=1    a=-318.31   l=0.000
J=1     S=0    E=2    a=-318.31   l=0.000
J=2     S=0    E=3    a=-1094.09  l=0.000
J=3     S=0    E=4    a=-1094.09  l=0.000
J=4     S=2    E=5    a=-1063.12  l=-5.496
J=5     S=3    E=6    a=-1112.78  l=-4.395
J=6     S=4    E=7    a=-1086.84  l=-9.363
J=7     S=2    E=8    a=-1876.61  l=-7.896
J=8     S=6    E=9    a=-2673.27  l=-5.586
J=9     S=7    E=10   a=-2673.27  l=-2.936
J=10    S=1    E=11   a=-4497.15  l=-17.078
J=11    S=1    E=12   a=-4497.15  l=-15.043
J=12    S=1    E=13   a=-4497.15  l=-12.415
J=13    S=2    E=14   a=-4521.94  l=-7.289
J=14    S=8    E=15   a=-2673.27  l=-3.422
J=15    S=5    E=16   a=-3450.71  l=-8.403
J=16    S=2    E=17   a=-4635.08  l=-7.289
J=17    S=2    E=18   a=-4724.45  l=-7.289
J=18    S=2    E=19   a=-4724.45  l=-7.289
J=19    S=2    E=20   a=-4724.45  l=-7.289
J=20    S=1    E=21   a=-4796.74  l=-12.415
J=21    S=2    E=22   a=-4821.53  l=-7.289
J=22    S=18   E=23   a=-435.64   l=-4.488
J=23    S=18   E=24   a=-524.33   l=-3.793
```

```
J=24    S=19   E=25   a=-520.16   l=-4.378
J=25    S=20   E=26   a=-521.50   l=-3.435
J=26    S=17   E=27   a=-615.12   l=-4.914
J=27    S=22   E=28   a=-514.04   l=-5.352
J=28    S=21   E=29   a=-559.43   l=-1.876
J=29    S=9    E=30   a=-1394.44  l=-2.261
J=30    S=10   E=30   a=-1394.44  l=-1.687
J=31    S=11   E=30   a=-1394.44  l=-2.563
J=32    S=12   E=30   a=-1394.44  l=-2.352
J=33    S=13   E=30   a=-1394.44  l=-3.285
J=34    S=14   E=30   a=-1394.44  l=-0.436
J=35    S=15   E=30   a=-1394.44  l=-2.069
J=36    S=16   E=30   a=-1394.44  l=-2.391
J=37    S=23   E=30   a=-767.55   l=-4.081
J=38    S=24   E=30   a=-692.95   l=-3.868
J=39    S=25   E=30   a=-692.95   l=-2.553
J=40    S=26   E=30   a=-692.95   l=-3.294
J=41    S=27   E=30   a=-692.95   l=-0.855
J=42    S=28   E=30   a=-623.50   l=-0.762
J=43    S=29   E=30   a=-556.71   l=-3.019
J=44    S=30   E=31   a=0.00      l=0.000
```

Index

ABORTONERR, 50
accumulators, 8, 127
accuracy figure, 40
ACCWINDOW, 65
adaptation, 13, 42
    adaptation modes, 134
    generating transforms, 43
    global transforms, 43, 135
    MAP, 42, 138
    MLLR, 42, 135
    MLLR formulae, 141
    regression tree, 43, 110, 135, 153
    supervised adaptation, 42, 134
    transform model file, 43, 137
    unsupervised adaptation, 42, 134, 187
ADDDITHER, 60
ALIEN, 71
all-pole filter, 60
ALLOWCXTEXP, 173
ALLOWXWRDEXP, 40, 173
analysis
    FFT-based, 29
    LPC-based, 29
ANON, 58
ARPA-MIT LM format, 221, 222
AT command, 34, 154
AU command, 39, 40, 42, 152
audio output, 72, 73
audio source, 72
AUDIOSIG, 73
average log probability, 183

back-off bigrams, 167
    ARPA MIT-LL format, 168
backward probability, 8
Baum-Welch algorithm, 8
Baum-Welch re-estimation, 6, 7
    embedded unit, 125
    isolated unit, 124
Bayes' Rule, 3
beam width, 127, 179
<BeginHMM>, 97
binary chop, 160
binary storage, 110, 146
binning, 62
Boolean values, 49
bootstrapping, 10, 17, 118
byte swapping, 53, 60
byte-order, 60
BYTEORDER, 60

C-heaps, 52
CEPLIFTER, 61, 63
cepstral analysis
    filter bank, 62
    liftering coefficient, 61
    LPC based, 61
    power vs magnitude, 62
cepstral coefficients
    liftering, 61
cepstral mean normalisation, 63
CFWORDBOUNDARY, 175
CH command, 91
check sums, 79
CHKHMMDEFS, 97
Choleski decomposition, 98
CL command, 36, 146
class id, 217
Class language models, 211, 223
class map
    as vocabulary list, 220
    complements, 219
    defining unknown, 219
    header, 218
cloning, 35, 36, 146
CLUSTER, 232–234
cluster merging, 39
clustering
    data-driven, 149
    tracing in, 152
    tree-based, 150
CO command, 40, 150
codebook, 108
codebook exponent, 6
codebooks, 6
coding, 29
command line
    arguments, 30, 47
    ellipsed arguments, 48
    integer argument formats, 47
    options, 15, 47
    script files, 30
compile-time parameters
    INTEGRITY_CHECK, 228
    INTERPOLATE_MAX, 228
    LMPROB_SHORT, 228
    LM_COMPACT, 228
    LM_ID_SHORT, 228
    SANITY, 228
compression, 79
configuration files, 15, 48–50

default, 48
format, 49
types, 49
configuration parameters
USEINTID, 227
compile-time, 228
operating environment, 53, 227
switching, 128
configuration variables, 15
display, 50
summary, 310
confusion matrix, 185
context dependent models, 145
continuous speech recognition, 118
count encoding, 221
Count-based language models, 209
covariance matrix, 95
cross-word network expansion, 176
cross-word triphones, 146

data insufficiency, 37
data preparation, 16, 23
DC command, 91, 173
DE command, 90
decision tree-based clustering, 151
decision trees, 37
loading and storing, 152
decoder, 178
alignment mode, 181
evaluation, 182
forced alignment, 186
live input, 188
N-best, 189
operation, 178
organisation, 180
output formatting, 187
output MLF, 183
progress reporting, 182
recognition mode, 181
rescoring mode, 182
results analysis, 183
trace output, 183
using adapted HMMs, 187
decompression filter, 53
defunct mixture components, 124
defunct mixtures, 153
deleted interpolation, 160
deletion errors, 183
delta coefficients, 65
DELTAWINDOW, 65
dictionaries, 161
dictionary
construction, 25, 171
edit commands, 172
entry, 25
format, 25
formats, 171
output symbols, 171
digit recogniser, 165
direct audio input, 71
signal control
keypress, 73
silence detector
speech detector, 72
DISCOUNT, 168
DISCRETE, 75
discrete data, 156
discrete HMM
output probability scaling, 108
discrete HMMs, 108, 155
discrete probability, 95, 107
DISCRETEHS, 103
DP command, 154
duration parameters, 95
duration vector, 113

EBNF, 19, 165
edit commands
single letter, 90
edit file, 90
embedded re-estimation, 32
embedded training, 10, 18, 118, 125
<EndHMM>, 97
energy suppression, 73
COMPRESSFACT, 64
ENORMALISE, 49, 65
environment variables, 53
error message
format, 315
error number
structure of, 315
error numbers
structure of, 50
errors, 50
full listing, 315
ESCALE, 65
EX command, 85, 186
extended Backus-Naur Form, 165
extended filenames, 48
extensions
mfc, 29
scp, 30
wav, 26

Figure of Merit, 19, 185
file formats
ALIEN, 71
Audio Interchange (AIFF), 70
Esignal, 68, 69
HTK, 66, 69
NIST, 69
NOHEAD, 71
OGI, 70
SCRIBE, 70
Sound Designer(SDES1), 70
Sun audio (SUNAU8), 70
TIMIT, 69
WAV, 71
files

adding checksums, 79
compressing, 79
configuration, 48
copying, 78
language models, 221, 223
listing contents, 76
network problems, 53
opening, 53
script, 47
VQ codebook, 76
filters, 53
fixed-variance, 123
flat start, 17, 27, 31, 118, 123
float values, 49
FoF file, 296
counts, 221
header, 221
FoF files, 221
FOM, 19, 185
`FORCECXTEXP`, 40, 173
forced alignment, 13, 30, 186
`FORCELEFTBI`, 173
`FORCEOUT`, 183
`FORCERIGHTBI`, 173
forward probability, 7
forward-backward
embedded, 125
isolated unit, 124
Forward-Backward algorithm, 7
frequency-of-frequency, 221
full rank covariance, 97

Gaussian mixture, 6
Gaussian pre-selection, 74
GCONST value, 114
generalised triphones, 150
`global.ded`, 172
global options, 112
global options macro, 121
global speech variance, 117
gram file
input, 221
sequencing, 221
gram files
count encoding, 220
format, 220
header, 220
grammar, 165
grammar scale factor, 40
grand variance, 148

Hamming Window, 59
HAudio, 15
`HAUDIO`, 71
HBuild, 19, 167, 169, 235–236
HCompV, 17, 31, 117, 123, 237–238
`HCONFIG`, 48
HCopy, 16, 29, 48, 78, 157, 239–241
HDict, 14
HDMan, 19, 25, 172, 242–244
HEAdapt, 13, 18, 43, 134, 187, 245–247
headers, 217
`HEADERSIZE`, 71
HERest, 10, 18, 32, 118, 125, 145, 248–250
HGraf, 15
HHEd, 18, 33, 42, 119, 144, 251–259
`HIFREQ`, 62
HInit, 7, 9, 17, 47, 117, 260–261
`HK` command, 159
HLabel, 14
HHEd, 36
HLEd, 16, 28, 35, 90, 186, 262–264
HList, 16, 76, 265
HLM, 14, 167
HLMCopy, 266
LNorm, 306
HLRescore, 267–268
HLStats, 16, 167, 269–270
HMath, 14, 46
HMem, 14, 46, 52
HMM
binary storage, 37
build philosophy, 18
cloning, 35
definition files, 31
definitions, 4, 94
editor, 18
instance of, 11
parameters, 95
triphones, 35
HMM definition
stream weight, 99
basic form, 96
binary storage, 110
covariance matrix, 97
formal syntax, 111
global features, 97
global options, 112
global options macro, 98
macro types, 102
macros, 101
mean vector, 97
mixture components, 97
multiple data streams, 99
stream weight, 99
symbols in, 96
tied-mixture, 107
transition matrix, 97
HMM lists, 92, 103, 104, 125
HMM name, 97
HMM refinement, 144
HMM sets, 103
types, 103
HMM tying, 104
HModel, 14
HNet, 13, 14
HParm
SILENERGY, 72
SILGLCHCOUNT, 72

SILMARGIN, 72
SILSEQCOUNT, 72
SPCGLCHCOUNT, 72
SPCSEQCOUNT, 72
SPEECHTHRESH, 72
HParm, 15
HParse, 19, 24, 165, 271–274
HParse format, 165
compatibility mode, 167
in V1.5, 167
variables, 166
HQuant, 16, 275–276
HRec, 13, 15, 180
HRest, 9, 17, 117, 277–278
HResults, 19, 183, 279–282
HSGen, 19, 26, 170, 283
HShell, 14, 46
HSigP, 14
HSKind, 103
HSLab, 16, 26, 284–287
HSmooth, 19, 160, 288–289
HTrain, 15
HUtil, 15
HEAdapt, 140
HVite, 10, 13, 18, 19, 34, 40, 180, 187, 290–292
HVQ, 14
HWave, 15
HWAVEFILTER, 69

insertion errors, 183
integer values, 49
Interpolating language models, 210
<InvCovar>, 97
isolated word training, 117
item lists, 36, 147
indexing, 147
pattern matching, 147

JO command, 159

K-means clustering, 120

label files, 83
ESPS format, 85
HTK format, 84
SCRIBE format, 85
TIMIT format, 85
labels
changing, 91
context dependent, 92
context markers, 85
deleting, 90
editing, 90
external formats, 84
merging, 91
moving level, 92
multiple level, 84
replacing, 91
side-by-side, 83
sorting, 90
LAdapt, 293–294
LLink, 303
LNewMap, 305
language model scaling, 183
language models
bigram, 167
lattice
comment lines, 335
field names, 335
format, 19, 334
header, 334
language model scale factor, 190
link, 334
N-best, 12
node, 334
rescoring, 13
syntax, 335
lattice generation, 189
lattices, 334
LBuild, 295
LFoF, 221, 296
LGCopy, 297–298
LGList, 299
LGPrep, 300–302
library modules, 14
likelihood computation, 4
linear prediction, 60
cepstra, 61
LINEIN, 72
LINEOUT, 72
live input, 41
<LLTCovar>, 98
LM file formats
ARPA-MIT format, 221
binary, 221, 223
class, 224, 225
class counts, 224
class probabilities, 224
ultra, 221
LMerge, 304
LOFREQ, 62
log arithmetic, 9
LPC, 61
LPCEPSTRA, 61
LPCORDER, 61
LPlex, 307–308
LPREFC, 61
LS command, 43, 152, 153
LSubset, 309
LT command, 40, 42, 152

M-heaps, 52
macro definition, 101
macro substitution, 102
macros, 34, 101
special meanings, 102
types, 102
marking word boundaries, 146

master label files, 28, 83, 86, 122, 125
  embedded label definitions, 86
  examples, 88
  multiple search paths, 86
  pattern matching, 87
  patterns, 28, 88
  search, 87
  sub-directory search, 87
  syntax, 87
  wildcards, 87
master macro file, 106
master macro files, 31
  input/output, 145
matrix dimensions, 97
MAXCLUSTITER, 157
maximum model limit, 183
MAXTRYOPEN, 53
ME command, 91
<Mean>, 97
mean vector, 95
MEASURESIL, 188
mel scale, 62
HIFREQ, 63
LOFREQ, 63
MELSPEC, 62
WARPFREQ, 62
WARPLCUTOFF, 63
WARPUCUTOFF, 63
memory
  allocators, 52
  element sizes, 52
  statistics, 52
memory management, 52
MFCC coefficients, 29, 97
MICIN, 72
minimum occupancy, 38
MINMIX, 153, 159, 160
<Mixture>, 97, 114
mixture component, 95
mixture incrementing, 152
mixture splitting, 153
mixture tying, 159
mixture weight floor, 153
ML command, 92
MLF, 28, 83
MMF, 31, 106
model compaction, 40
model training
  clustering, 149
  compacting, 150
  context dependency, 145
  embedded, 125
  embedded subword formulae, 132
  forward/backward formulae, 130
  HMM editing, 145
  in parallel, 127
  initialisation, 119
  isolated unit formulae, 132
  mixture components, 120
  pruning, 126
  re-estimation formulae, 129
  sub-word initialisation, 122
  tying, 146
  update control, 122
  Viterbi formulae, 129
  whole word, 121
monitoring convergence, 122, 126
monophone HMM
  construction of, 30
MP command, 173
MT command, 154
MU command, 153
mu law encoded files , 69
multiple alternative transcriptions, 189
multiple hypotheses, 334
multiple recognisers, 180
multiple streams, 73
  rules for, 73
multiple-tokens, 13

N-best, 13, 189
n-gram language model, 295
N-grams, 167
NATURALREADORDER, 53
NATURALWRITEORDER, 53
NC command, 149
network type, 174
networks, 161
  in recognition, 162
  word-internal, 40
new features
  in Version 2.1, 21
  in Version 3.1, 20
  in Version 3.2, 19
ngram
  count encoding, 221
  files, 220
NIST, 19
NIST format, 184
NIST scoring software, 184
NIST Sphere data format, 69
non-emitting states, 10
non-printing chars, 51
NSAMPLES, 71
NUMCEPS, 61, 63
CMEANDIR, 64
CMEANMASK, 64
NUMCHANS, 49, 63
VARSCALEDIR, 64
VARSCALEFN, 64
VARSCALEMASK, 64
<NumMixes>, 97, 109
<NumStates>, 112

observations
  displaying structure of, 78
operating system, 46
outlier threshold, 38
output filter, 53

output lattice format, 189
output probability
    continuous case, 6, 95
    discrete case, 95
OUTSILWARN, 188
over-short training segments, 124

parameter estimation, 116
parameter kind, 66
parameter tie points, 102
parameter tying, 36
parameterisation, 29
partial results, 183
path, 10
    as a token, 11
partial, 10
Perplexity, 208
phone alignment, 34
phone mapping, 34
phone model initialisation, 118
phone recognition, 176
phones, 118
PHONESOUT, 72
phonetic questions, 151
pipes, 46, 53
PLAINHS, 104
pre-emphasis, 59
PREEMCOEF, 59
Problem solving, 214
prompt script
    generationof, 26
prototype definition, 16
pruning, 18, 32, 179, 183
    in tied mixtures, 159
pruning errors, 127

QS command, 38, 151
qualifiers, 57, 66
    _A, 65
    _T, 65
    _C, 67, 80
    _D, 65
    _E, 65
    _K, 67, 80
    _N, 66, 73
    _O, 65
    _V, 75, 158
    _V, 79
    _Z, 63
    codes, 67
    ESIG field specifiers, 68
    summary, 81

RAWENERGY, 65
RC command, 43, 153
RE command, 90
realignment, 34
recogniser evaluation, 40
recogniser performance, 184
recognition
    direct audio input, 41
    errors, 184
    hypothesis, 179
    network, 178
    output, 41
    overall process, 163
    results analysis, 41
    statistics, 184
    tools, 19
recording speech, 26
RECOUTPREFIX, 189
RECOUTSUFFIX, 189
reflection coefficients, 60
regression formula, 65
removing outliers, 150
results analysis, 19
RN, 154
RN command, 43
RO command, 38, 145, 150
RP command, 173
RT command, 154

SAVEASVQ, 75, 158
SAVEBINARY, 111, 146
SAVECOMPRESSED, 79
SAVEWITHCRC, 79
script files, 47, 121
search errors, 180
segmental k-means, 17
sentence generation, 171
sequenced gram files, 221
SH command, 150
SHAREDHS, 103
short pause, 33
signals
    for recording control, 188
silence floor, 65
silence model, 33, 34, 148
SILFLOOR, 65
simple differences, 66
SIMPLEDIFFS, 66
single-pass retraining, 128
singleton clusters, 150
SK command, 154
SLF, 19, 24, 161, 163
    arc probabilities, 165
    format, 163
    null nodes, 164
    word network, 164
SO command, 90
software architecture, 14
SOURCEFORMAT, 68
SOURCEKIND, 57, 188
SOURCELABEL, 84, 92
SOURCERATE, 58, 71
SP command, 173
speaker identifier, 185
SPEAKEROUT, 72
speech input, 56

automatic conversion, 57
bandpass filtering, 62
blocking, 58
byte order, 60
DC offset, 59
direct audio, 71
dynamic coefficents, 65
energy measures, 65
filter bank, 62
general mechanism, 56
Hamming window function, 59
monitoring, 76
pre-emphasis, 59
pre-processing, 59
summary of variables, 80
target kind, 57
speech/silence detector, 188
SS command, 154, 159
ST command, 40, 152
stacks, 52
standard lattice format, 19, 24, 161, 163
definition, 334
standard options, 50
-A, 50
-C, 48, 50
-D, 50
-F, 68, 84
-G, 84
-I, 86
-L, 87, 122
-S, 47, 50, 87
-T, 50, 122
-V, 50
summary, 55
state clustering, 37
state transitions
adding/removing, 154
state tying, 37, 150
statistics
state occupation, 37
statistics file, 38, 145, 150
<Stream>, 99, 107
stream weight, 6, 95
<StreamInfo>, 98, 112
streams, 6
stress marking, 25
string matching, 183
string values, 49
strings
metacharacters in, 51
output of, 51
rules for, 51
SU command, 154
sub-lattices, 169, 334
SUBLAT, 170, 334
SW command, 154
<SWeights>, 99

TARGETKIND, 49, 57, 158
TARGETRATE, 58
task grammar, 23, 167
TB command, 39, 151
TC command, 92, 146, 149
tee-models, 33, 109
in networks, 167
termination, 50
TI command, 34, 148
tied parameters, 146
tied-mixture system, 107
tied-mixtures, 148, 158
build procedure, 159
output distribution, 107
tied-state, 103
TIMIT database, 24, 90
token history, 179
token passing, 179
Token Passing Model, 11
total likelihood, 8, 9
TR command, 38
tracing, 50, 225
training
sub-word, 118
whole-word, 117
training tools, 16
TRANSALT, 84, 92
transcription
orthographic, 27
transcriptions
model level, 187
phone level, 187
word level, 187
transitions
adding them, 33
TRANSLEV, 84, 92
<TransP>, 97
tree building, 39
tree optimisation, 152
triphones
by cloning, 35
from monophones, 35
notation, 36
synthesising unseen, 42
word internal, 35
two-model re-estimation, 128
tying
examples of, 148
exemplar selection, 148
states, 37
transition matrices, 36

UF command, 145
under-training, 160
uniform segmentation, 120
unknown class, 219
unseen triphones, 40, 150
synthesising, 152
up-mixing, 152
upper triangular form, 97

USEHAMMING, 59
USEPOWER, 62
USESILDET, 72, 188
UT command, 148

V1COMPAT, 66, 167
varFloorN, 122
variance
    flooring problems, 37
<Variance>, 97
variance floor macros, 31
    generating, 123
variance floors, 122
<VecSize>, 97, 112
vector dimensions, 97
vector quantisation, 74
    code book external format, 76
    distance metrics, 75
    type of, 76
    uses of, 75
Viterbi training, 7, 120
vocabulary list, 218, 220
VQTABLE, 75

warning codes
    full listing, 315
warning message
    format, 315
warnings, 50
waveform capture, 188
WB command, 35, 146
whole word modelling, 117
whole word recognition, 177
WINDOWSIZE, 49, 58
WLR, 12
word equivalence, 185
word id, 217
word insertion penalty, 40
word internal, 35
Word Link Record, 12
word list, 25
word map
    entries, 218
    example of, 218
    header, 217
word N-best, 13
expansion rules, 173
word networks
    tee-models in, 167
word spotting, 177
word-end nodes, 11, 12, 179
word-internal network expansion, 176
word-loop network, 170
word-pair grammar, 170

ZMEANSOURCE, 59