Lecture 2: Acoustic Features, Acoustic Model Training, and Unix Scripting

Lecturer: Mark Hasegawa-Johnson (jhasegaw@uiuc.edu)
TA: Sarah Borys (sborys@uiuc.edu)

January 8, 2009

Contents

1 Speech Recognition Subtasks

This minicourse takes an *a la carte* approach to the different sub-tasks involved in the training and testing of an automatic speech recognizer. Here are some of those sub-tasks.

- *Denoising/Dereverberation.* Cell phone speech and TV soundtracks are difficult to process using ASR, because of the background noise/music. Any speech recorded without a headset microphone is almost impossible to recognize, because of reverberation.
  - The best solution, by far, is to make your talkers wear a head-mounted microphone.
  - If that fails, try the matlab techniques listed in lecture 1, including...
  - svn://mickey.ifp.uiuc.edu/lee_vad
  - http://ssli.ee.washington.edu/people/chiaping/mva.html

- *Waveform→Features.* Many different waveforms correspond to the same phone; many different phones may correspond to the same waveform. In order to be useful for ASR, the waveform must be converted so that there is a more consistent correspondence between the phone state being spoken, q, and the observed acoustic feature vector, x_t. There are two broad classes of feature representations in speech recognition:

  1. Spectral/Cepstral representations are basically decorrelated log-magnitude-Fourier-transforms. The basic transformations include Fourier transform (because the basilar membrane performs a frequency analysis), throwing away phase (because the ear is relatively insensitive to phase—though some types of inter-frequency relative phase are quite audible!), logarithmic or cube-root compression (because loudness is proportional to the cube root of power), warping the frequency axis so that regions with equal perceptual weight have equal volume in the feature space, and decorrelation using a KLT=PCA (Karhunen-Loeve transform=principal components analysis) or DCT (discrete cosine transform).
  2. Discriminative representations are designed to optimally discriminate among a group of pre-specified class labels. I consider these discriminative transforms to be a model of the fabled "grandmother cell" in the human cortex—the neuron that fires only when you see your grandmother. Cells with nearly that level of specificity have been found in the auditory cortex [6]. Grandmother cells can be simulated with an NN/HMM tandem system [10], an SVM/HMM tandem system [5], and/or an AdaBoost/HMM tandem system [17]. Useful software for training grandmother cells includes:
     - SPRACH/QuickNet (http://www.icsi.berkeley.edu/d̃pwe/projects/sprach/sprachcore.html). License: Not quite open source — redistribution and non-commercial use allowed, commercial use and redistribution of modified code not allowed. Code: C. Distributed recognition models: demo is available for the Berkeley Restaurant Project (BeRP) dialogue system.
     - WEKA (http://www.cs.waikato.ac.nz/ml/weka/) is used by most people who just want to train a neural net, and don't want to write their own back-propagation code. It has been used for phone classification tasks and for a number of language modeling tasks; I don't know if anybody has used it yet for Tandem...
     - Ditto for SNoW, except that it uses multiplicative weight updates rather than additive updates, giving it a better ability to winnow away useless features (http://l2r.cs.uiuc.edu/ cogcomp/software.php)
     - Support vector machines are usually trained using similar configuration + scripting methods. LibSVM (http://www.csie.ntu.edu.tw/ cjlin/libsvm/) and SVM-light (http://svmlight.joachims.org/) are similar, but with some complementary functionality.
     - PVTK (Periodic Vector Toolkit: http://www.isle.uiuc.edu/software) includes three programs specialized for the purposes of concatenating and transforming spectral vectors, extracting spectral vectors to train a neural net or SVM, and applying a bank of SVMs to a huge speech database in batch mode.

- *Language Model Training.* A "language model" specifies the probability that Emilia User says word w_4 given that she has already said words w_1, w_2, w_3. The basic algorithm for language modeling is dead simple, and by tomorrow afternoon, you will be able to write it in perl or ruby; one simply needs to compute

$$p(w_4|w_1, w_2, w_3) = \frac{N(w_1, w_2, w_3, w_4)}{N(w_1, w_2, w_3)}$$

  where $N(\cdot)$ is the frequency count of event $\cdot$ in a training database. Language model training tools help you to do more complex things, like combine counts from a small task-dependent database and a large task-independent database, or reduce the number of words in your dictionary (to avoid garden-pathing your recognizer). Useful packages for this purpose include

  - SRILM, the SRI language modeling toolkit: http://www.speech.sri.com/projects/srilm/
  - The language modeling toolkit distributed with HTK (http://htk.eng.cam.ac.uk/)

- OpenFST (http://www.openfst.org) can help to combine language models, or language models with pronunciation models.

- *Acoustic Model Training.* An "acoustic model" specifies the relationship between the phone state at time t, q_t, and the acoustic spectrum, x_t. Since x_t is a real-valued vector, instead of a symbol, the probability $p(x_t|q_t)$ must be a real-valued function. Most often, we model it using a weighted sum of Gaussians:

$$p(x_t|q) = \sum_{k=1}^{K} c_{qk} \prod_{d=1}^{D} \frac{1}{\sqrt{2\pi\sigma_{qkd}^2}} e^{-\frac{(x_{td}-\mu_{qkd})^2}{2\sigma_{qkd}^2}}$$

  where x_{td} is the dth component of the vector x_t, and the parameters c_{qk}, μ_{qkd}, and σ_{qkd}^2 have been learned from training data. Wonderfully efficient tools exist to learn those parameters.

  - HTK (http://htk.eng.cam.ac.uk) includes what I have found to be the most efficient program (HERest) for training acoustic models.
  - GMTK (http://ssli.ee.washington.edu/~bilmes/gmtk/) treats both acoustic and language models as specific examples of dynamic Bayesian networks, therefore it is one of the most flexible set of tools for training either (at least among large-vocabulary tools)
  - The HMM Toolbox for matlab (http://www.cs.ubc.ca/ murphyk/Software/HMM/hmm.html) is *by far* the easiest way to train an HMM (especially if you're already familiar with matlab), and is effective for training corpora up to perhaps one hour of speech; beyond that, it starts to be a bit cumbersome.
  - Sphinx (http://cmusphinx.sourceforge.net/) is a complete ASR system including code to train acoustic models, and pre-trained acoustic models.
  - ISIP (http://www.isip.piconepress.com/projects/speech/) is a complete open-source speech recognizer, including code to train acoustic models, and pre-trained acoustic models

- *Automatic Speech Recognition (Decoding).* A "decoder" is an algorithm that accepts three inputs: (1) a trained language model, (2) a trained acoustic model, and (3) a waveform that you want to recognize. As output, it transcribes the waveform. HTK, GMTK, HMMToolbox, Sphinx, and ISIP all include dynamic decoders (run-time compiled search graph, hence usually quite slow for large-vocabulary tasks, but convenient because they don't require any extra search-graph-compilation step). Static decoders (pre-compiled search graph) tend to be much faster; the two open source static decoders are

  - Julius, http://julius.sourceforge.jp/en_index.php has with features including a built-in two-pass recognition using bigrams in the forward pass and trigrams in the backward pass.
  - HDecode, http://htk.eng.cam.ac.uk, is reputed to be as fast as Julius, but is not yet nearly as well documented as the rest of HTK.

2 Acoustic Features

All acoustic features commonly used in automatic speech recognition are based on the spectrogram. A "spectrogram" is a time-frequency representation of the signal, $X(t, f)$, where t may be in seconds and f may be in Hertz. Specifically, $X(t_0, f)$ is the log magnitude of the Fourier transform, at frequency f, of the signal multiplied by a short window $w(t)$ ($w(t)$ is usually 20-30ms in length):

$$X(t_0, f) = |\mathcal{F}\{x(t + t_0)w(t)\}| \quad (1)$$

$$\text{SPEC}(t_0, f) = \log X(t_0, f) \quad (2)$$

Phonetically readable spectrogram displays compute $X(t_0, f)$ once per 2ms, and humans can hear acoustic events with a temporal precision of about 2ms, but in order to save memory and computational complexity, speech recognition algorithms usually only compute $X(t_0, f)$ once per 10ms. The only speech events not well represented by a 10ms skip are stop bursts. Perhaps more important, a 10ms skip-time makes it difficult to deal with impulsive background noise (clicks).

2.1 Mel Spectra

Human hearing is characterized by two different semi-logarithmic frequency scales: the Bark scale, and the mel scale. First, the inner ear integrates spectral energy in bands of width equal to one Bark; for this reason, if the total noise power within 1/6 octave on either side of a tone is more than about 4dB above the power of the tone, the tone will be masked. One Bark is equal to about 1/6 octave (two semitones, or 12%), with a minimum value of about 312Hz. Second, the just-noticeable-difference between the frequencies of two pure tones is about 3 mels. One mel is equal to about 0.1% (about 1/700 octave), with a minimum value of about 0.6Hz.

Both of these scales suggest that the ability of humans to discriminate two sounds is determined by the spectral energy as a function of semilog-frequency (Bark frequency or mel frequency), not linear frequency (Hertz). Most speech recognition features average the spectral energy within bandwidths of 1 Bark, or about 90-120 mels, in order to eliminate distinctions that depend on the difference between two frequencies that are within 1 Bark of each other. This frequency-dependent smearing or smoothing process substantially increases the accuracy of a speech recognizer.

Frequency-dependent smearing can be done using sub-band filters or wavelets, but is most often done by, literally, adding up the Fourier transform coefficients in bands of width equal to about 90 mels (resulting in roughly 32 bands between 0Hz and 8000Hz). The resulting coefficients are called mel-frequency spectral coefficients, or MFSC:

$$\tilde{X}(t_0, k) = \sum_{f=0}^{4000} X(t_0, f) H_k(f) \tag{3}$$

$$\text{MFSC}(t_0, k) = \log \tilde{X}(t_0, k) \tag{4}$$

The weighting functions, $H_k(f)$, are usually triangular, centered at frequencies f_k that are about 90 mels apart [7].

2.2 Mel Cepstra

Neighboring spectral coefficients are highly correlated: for example, if there is a spectral peak near center frequency f_k, then $\text{MFSC}(t, k)$ is often large, but usually, so is $\text{MFSC}(t, k+1)$. Probabilistic models such as diagonal-covariance Gaussians work best if the features are uncorrelated. It has been shown that the MFSCs can be pretty well decorrelated by transforming them using a discrete cosine transform or DCT:

$$\text{MFCC}(t_0, m) = \mathcal{C}\left\{\text{MFSC}(t_0, k)\right\} \tag{5}$$

$$\tilde{c}(m) = \frac{1}{K} \sum_{b=1}^{K} \log \tilde{S}(b) \cos\left[\frac{\pi m}{K}\left(b - \frac{1}{2}\right)\right] \tag{6}$$

The DCT, $\mathcal{C}$, is just the real part of a Fourier transform. Thus, in effect, the MFSCs are the inverse Fourier transform of the log of the frequency-warped spectrum. $\text{MFCC}(t, 0)$ is similar to the average spectral energy; $\text{MFCC}(t, 1)$ is similar to the average spectral tilt. The peak corresponding to F1 occurs at $\text{MFCC}(t, F_s/F_1)$.

Eq. 5 is useful for decorrelating the MFSCs, but it actually doesn't change the perceptual distance between two sounds. Because the DCT is a unitary transform, the distance between two MFSC vectors is exactly the same as the distance between two MFCC vectors:

$$\sum_k \left(\text{MFSC}_1(k) - \text{MFSC}_2(k)\right)^2 = \sum_m \left(\text{MFCC}_1(m) - \text{MFCC}_2(m)\right)^2 \tag{7}$$

Thus it's possible to think of Eq. 5 as a meaningless mathematical convenience — it simplifies the probabilistic model, but doesn't change the representation of human perception.

2.3 LPCC

Humans are most sensitive to the frequencies and amplitudes of spectral peaks. There is some evidence that humans completely ignore zeros in the spectrum, except to the extent that the zeros change the amplitudes

of their surrounding peaks. Thus it may be reasonable to smooth the spectrum in some way that carefully represents the frequencies and amplitudes of peaks, while ignoring inter-phoneme differences in the amplitudes of spectral valleys. It has been shown [16] that focusing on the peak can be accomplished by using a parameterized spectrum $A(t_0, f)$, and by choosing its parameters according to the following constrained minimization:

$$A(t_0, f) = \arg\min \sum_f \left(\frac{X^2(t_0, f)}{A^2(t_0, f)} \right) \quad \text{subject to } \sum_f X^2(t_0, f) = \sum_f A^2(t_0, f) \tag{8}$$

When the vocal folds clap together, the resulting impulse-like sound excites the resonance of the vocal tract (formants); the sound pressure in the vocal tract continues to ring like a bell for a few milliseconds afterward. During this time, the waveform looks like a sum of exponentially decaying sine waves. Once you have figured out the frequencies and decay rates of these sine waves (formants), you can predict each sample of the speech signal from its $2N$ previous samples (N is the number of formants; $2N$ because you need to know both the frequency and decay rate). The prediction coefficients are called linear prediction coefficients, or LPC [1]. The LPC coefficients a_i specify a model of the spectrum:

$$A(t_0, f) = \left| \frac{1}{1 - \sum_{i=1}^{2N} a_i e^{j2\pi i f F_s}} \right| \tag{9}$$

where F_s is the sampling frequency. If a_i are chosen according to Eq. 8 (as they usually are), then $A(t, f)$ is a smoothed spectrum that accurately represents the amplitudes and frequencies of spectral peaks, possibly at the cost of more

The linear predictive cepstral coefficients (LPCC) are computed by taking the DCT of $\log A(t_0, f)$, in order to decorrelate it:

$$\text{LPCC}(t_0, m) = \mathcal{C}\left\{\log A(t_0, f)\right\} \tag{10}$$

The LPC smoothing process (Eq. 9) changes the implied perceptual distance between two sounds: sounds with different spectral valleys become more similar, while sounds with different spectral peaks become less similar. The LPC→LPCC transformation (Eq. 10) doesn't change the distance between two sounds; it just decorrelates the features.

2.4 Perceptual LPC (PLP)

Perceptual LPC combines together the frequency-dependent smoothing of MFSC with the peak-focused smoothing of LPC [9]. The spectral model $\tilde{A}(t_0, k)$ is chosen according to

$$\tilde{A}(t_0, k) = \arg\min \sum_k \left(\frac{\tilde{X}^2(t_0, f)}{\tilde{A}^2(t_0, f)} \right) \quad \text{subject to } \sum_k \tilde{X}^2(t_0, k) = \sum_k \tilde{A}^2(t_0, k) \tag{11}$$

PLP coefficients are never used directly, except for the purpose of displaying a pretty smoothed spectrum. In speech recognition, they are always DCT'd:

$$\text{PLP}(t_0, m) = \mathcal{C}\left\{\log \tilde{A}(t_0, k)\right\} \tag{12}$$

2.5 Computing Acoustic Features with HCopy

The HTK program `HCopy` can convert a waveform into any of the acoustic feature types described above.

If you are on a machine that doesn't have HTK already installed, download the source code and/or binaries from http://htk.eng.cam.ac.uk. You will have to register (free). While you're there, download the HTK book. If you plan to do anything serious, you will need chapters 4-17.

The `HCopy` program copies a speech file. In the process of copying, it can (1) convert from one feature type to another, (2) convert from other file formats to HTK file format, (3) extract subsegments corresponding to labeled phonemes and/or specified start and end times, or (4) concatenate multiple files. `HCopy` is usually called (from the unix or DOS command line) as

```
HCopy  -S scriptfile.scp -T 1 -C config.txt
```

The `-T 1` option tells any HTK program to print trace (debugging) information to standard error. Higher trace settings print more information; `-T 1` will just print the name of each speech file as it is converted.

An HTK "script file" is a list of the files to be processed. All HTK tools can take a script file using the `-S` option. The content of a script file is literally appended to the command line; thus, for example, the command shown above could also be implemented by typing

```
HCopy  -S scriptfile.scp
```

and by including, as the first line in the script file, the characters `-T 1 -C config.txt`. In general, it is considered good form to specify options like `-T` and `-C` on the command line, and to use the script file only to specify the filename arguments.

The filename arguments of `HCopy` come in inputfile-outputfile pairs, e.g., the file might include

```
timit/TRAIN/DR1/FCJF0/SA1.WAV data/fcfj0sa1.plp
timit/TRAIN/DR1/FCJF0/SA2.WAV data/fcfj0sa2.plp
...
```

The line breaks are optional; you can specify all of `HCopy`'s filename arguments on the same line, but your script file will be less readable that way. A script file like the one above could be generated using the following bash script:

```
#!/bin/bash
foreach d ‘ls timit/TRAIN‘; do
  foreach s ‘ls timit/TRAIN/$d‘; do
    foreach f ‘ls timit/TRAIN/$d/$s/*.WAV‘; do
      output=‘echo $f | sed ’/WAV/s/.*\//data\//;/WAV/s/WAV/plp/;’‘;
      echo $f $output;
    done
  done
done
```

All of the processing to be performed is specified in the configuration file, which could be called `config.txt`. Here is an example:

```
SOURCEFORMAT = NIST
SOURCEKIND = WAVEFORM
TARGETKIND = PLP_E_D_A_Z_C
TARGETRATE = 100000.0
WINDOWSIZE = 250000.0
NUMCHANS = 32
CEPLIFTER = 27
NUMCEPS = 12
```

Here are what the lines of the configuration file mean:

- `SOURCEFORMAT` specifies the file format of the input. This can be `NIST` (for NIST or LDC distributed SPHERE files), or `WAV` (for Microsoft WAV files), or `HTK` (for HTK-format files).
- `TARGETFORMAT`, if specified, would declare the desired format of the output. Only waveform data can be output in any form other than HTK. Since HTK is the default, TARGETFORMAT doesn't need to be specified.
- `SOURCEKIND` specifies that the input is waveform data. This line is optional, since WAVEFORM is the default input data kind.
- `TARGETKIND` specifies that the outputs are PLP cepstral coefficients (Sec. 2.4). Other useful options include `MELSPEC` ($\tilde{X}(t_0, k)$, Sec. 2.1), `FBANK` (MFSCs, Sec. 2.1), `MFCC` (Sec. 2.2), and `LPCEPSTRA` (Sec. 2.3). The modifiers have the following meanings:

- _E: Append a normalized energy to each PLP vector. The "normalized energy" is the log Energy, normalized so that the highest value in each utterance file is exactly 1.0.
- _D: Append the delta-PLPs and delta-energy to each PLP vector.
- _A: Append delta-delta-PLPs and delta-delta-energy to each PLP vector.
- _T: Append delta-delta-delta-PLPs and delta-delta-energy to each PLP vector.
- _Z: Perform cepstral mean subtraction: from each cepstral vector, subtract the average cepstral vector, where the average is computed over the entire file. This option is most useful when the recognizer will be tested with a different microphone, or in a different kind of room, than was used to train the recognizer. Under other conditions, it may hurt recognition performance.
- _C: Compress the coefficients. Compressed HTK files are half as large as uncompressed files, and compressed files may actually have *better* precision than uncompressed files.

- `WINDOWSIZE` specifies the length of the window $w(t)$, in 100ns units. The example specifies a 25ms window ($25ms = 25000\mu s = 250000 \times 100ns$).
- `TARGETRATE` specifies the frame skip parameter, in 100ns units. The example specifies one frame per 10ms ($10ms = 10000\mu s = 100000 \times 100ns$).
- `NUMCHANS` specifies the number of mel-frequency filterbanks to use. Thus, in Eq. 11, k goes from 1 to 32.
- `NUMCEPS` specifies the number of cepstra to keep. Thus the total acoustic feature vector will have dimension 39: 12 cepstra, then the energy, then the deltas of all 13 coefficients, then their delta-deltas.
- `CEPLIFTER` specifies that the cepstrum should be "liftered," in order to slightly de-emphasize the lower-order cepstra. Liftering is useful because the low-order cepstra (small m) are usually much larger in amplitude than the high-order cepstra (large m). Without liftering, speech recognition distance measures would completely ignore the high-order cepstra. Liftering multiplies the cepstra by a lifter of length $L = 27$:

$$\mathrm{PLP}'(t,m) = \mathrm{PLP}(t,m) \times \left(1 + \frac{L}{2}\sin\frac{\pi m}{L}\right) \tag{13}$$

2.6 HTK File Format

The HTK file format is one of the best available formats for the compact but precise storage of periodic real-valued spectral or cepstral vectors (e.g., one vector every 10ms for 2 seconds, or one vector per day for 365 days, or any other such file). The main advantage of HTK format is the fast, fixed-length, high-accuracy data compression scheme. The main disadvantage is the unnecessarily restrictive type checking.

An HTK file is a 12-byte header, followed optionally by compression information, followed by sample vectors stored in either 4-byte floating point or 2-byte short integer format. All data are stored in IEEE standard floating point, with big-endian data order (high-order byte stored first), thus if you are writing C code to read or write an HTK file on a little-endian machine (e.g., Intel or AMD), you will need to implement byte swapping. The header contains the following variables:

nSamples	Number of sample vectors in the file (4 byte integer)
sampPeriod	Sample period in 100ns units (4 byte integer)
sampSize	Number of bytes in each sample vector (2 byte integer)
parmKind	Two-byte code specifying feature kind

The `parmKind` code specifies the base kind (PLP, MFCC, etc), plus all modifiers (_E_D_A...) using a code specified on p. 66 of the HTK book. The `sampSize` specifies the number of *bytes* per vector, not the number of `dimensions:` thus if vectors are compressed, `sampSize` is twice the number of dimensions, otherwise four times.

If the file is uncompressed, the rest of the file is a series of parameter vectors, stored in 4-byte floating point format (IEEE standard big-endian).

If the file is compressed, the next $8N_d$ bytes contain $2N_d$ floating point numbers: $A[1] \ldots A[N_d]$, and $B[1] \ldots B[N_d]$, where N_d is the dimension of each sample vector. After the compression information, the

sample vectors themselves are stored as 2-byte big-endian integers, $D[t, m]$. The value of the real features are computed as:

$$\text{PLP}(t, m) = \frac{1}{A[m]} \left(D[t, m] + B[m] \right) \tag{14}$$

The values of $A(m)$ and $B(m)$ are chosen as specified on p.67 of the HTK book, in order to make sure that $\max_t D[t, m] = 1$ and $\min_t D[t, m] = -1$. Since the coefficients are scaled up to take full advantage of the 16-bit short integer, the signal to quantization noise ratio (SQNR) of features encoded this way is truly 2^16, or 96dB — better, in some cases, than the SQNR of a naive floating-point encoding.

3 Acoustic Model Training: Overview

The first time that you train a speech recognizer, you're likely to spend most of your time training the acoustic models. Future lectures will come back and work through these steps in more detail, using an example script called train.pl; in this lecture, I'd like to just introduce the steps, and then talk about the different types of information that you need, from your database, in order to get started.

Remember that an acoustic model is just a parameterized function, $p(x_t|q)$, where q is the phone state, and x_t is the speech feature vector (PLP or MFCC or Tandem or whatever). In a complete system, $p(x_t|q)$ is a mixture of Gaussians, but it starts out simple:

1. First, we create a universal background unimodal Gaussian (UBUG), $p(x_t)$. That previous sentence is a high-falutin' way of saying that we should compute the mean and variance of each feature. The HTK command line tool `HCompV` does this.

2. Second, we train monophone models (one HMM for each phoneme of the language, typically about 40 phones for a single-language recognizer) with Gaussian PDFs.

   (a) Use your dictionary, together with your non-time-aligned orthographic transcriptions, in order to produce a non-time-aligned phone transcription. `HLEd` does this.

   (b) Create prototype HMMs for each phone. In order to make sure that the initial settings of the likelihood functions are not completely nonsense, we initially just copy the UBUG into the definition for each phone. `HHEd` does this.

   (c) Use the EM algorithm (expectation maximization) to re-estimate the parameters of each phone model. Much has been written about the EM algorithm, but here are the references I like best: Baum proved it [2, 3], Liporace applied it to Gaussians [13], Juang et al. applied it to mixture Gaussians [11], Levinson and Rabiner wrote the best tutorial for those who want to know how it works but not why ([12], republished as [15]), and Bilmes wrote the best tutorial for those who want to know why it works [4]. `HERest` does this.

   (d) Fifth, it is often useful to time-align the phone transcriptions, and in particular, to let the recognizer tell you if some of your talkers are using non-standard pronunciations. This can be done using `HVite`.

   (e) Sixth, you should re-re-estimate your acoustic models using the time-aligned and corrected transcriptions. `HRest` or `HERest` can be used for this purpose.

   (f) Speaker-Adaptive training would go about here, but we have not yet used it at Illinois. Speaker-Adaptive training [8, 14] compensates for speaker differences during acoustic model training: each speaker's training data is linearly transformed so that it more closely resembles the training data for a "prototype" speaker. In this way, acoustic models are made more precise, because the Gaussian doesn't have to model inter-speaker variability—instead, inter-speaker variability is handled by a separate speaker normalization step.

3. Triphones! Triphones are context-dependent acoustic models, e.g., the /t/ in "this tree" can now be different from the /t/ in "that tip." We use the following notation: a /t/, preceded by an /s/, and followed by an /r/, is written /s-t+r/. Notice that this is a type of /t/; the /s-/ and /+r/ bits specify only the phonetic context. At Illinois we also usually specify prosodic context (lexical stress, phrase position), but this is hard to do in a corpus-independent way.

(a) First, the transcription is converted from a time-aligned monophone to a time-aligned triphone transcription, using `HLEd`.

(b) Second, triphone models are created for every distinct triphone. To start with, each triphone model duplicates the PDF of its corresponding monophone; this cloning is done using `HHEd`.

(c) Use `HRest` and/or `HERest` to re-train.

(d) You probably don't have enough data to train distinct acoustic models for each and every three-phone sequence in the English language. `HHEd` can be used to cluster the triphones, and `HERest` or `HRest` can re-estimate the clustered triphones.

4. Now you are finally ready to convert the Gaussian PDFs into Mixture Gaussian PDFs! This is done using several iterations of `HHEd` and `HERest`, one after the other. This process must be done slowly, because any mixture Gaussian with $K > 1$ suffers from spurious and undesirable global optimum parameter settings (that phrase, translated into plain English: if you try to learn a 128-component mixture Gaussian all at once without proper initialization, the training algorithm will learn a set of parameters that work really well for the training data and really badly for anything else, usually including at least one nearly-zero variance parameter). In order to avoid these spurious global optima, we usually upmix slowly, e.g., going from one Gaussian to two, then to four, and so on, checking the variances at each step to make sure no variance parameter is getting too small.

5. Finally, HMMs should be discriminatively trained, using `HMMIRest`. Discriminative training is just like EM training, but with a different optimality criterion—one that better predicts the final error rate of the recognizer. In order to avoid over-fitting the recognizer to the training database, the strict error-rate criterion is regularized, with regularization constant adjusted to give you the degree of generality you require.

4 Bash, Sed, Awk

Why not use C all the time? The answer is that some tasks are easier to perform with other programming languages:

- Manipulate file hierarchies: use bash and sed.
- (Simple manipulation of tabular text files: gawk)
- Manipulate text files and most non-text: use ruby.
- Manipulate non-text files: use ruby, with embedded C-language or FORTRAN functions that run the most computationally intensive bits (use a profiler to determine which those are).

bash is a POSIX-compliant command interpreter, meaning that, like the DOSshell, you can type in a program name, and the program will run. Unlike the DOSshell, bash is also a pretty good programming language (not as good as BASIC or Ruby, but better than DOSshell or tcsh).

- Key advantages of bash: if you are writing a bash program (a "shell script"), and if you want to know whether or not a particular line of code will do what you want it to do, you can just cut it out from the text editor, paste it into an X-terminal, and hit return. The line will run natively, and you can look at the response. This is by far the easiest way to debug any program that I've ever worked on; I also use this method to debug perl and ruby.
- Another key advantage of bash: In unix, the heading `#!/bin/bash` makes the file executable—if you save the following text in a file called "train_monophones.sh," then make the file world-executable by using the command

  ```
  chmod a+r train_monophones.sh
  ```

  then you can execute it just like any other program, i.e., by typing the following at the command prompt:

  ```
  train_monophones.sh
  ```

- Another key advantage of bash: the bash man page is one of the best (though the O'Reilly books are also good). It is hard to navigate at first, but well worth the effort; type

  ```
  man bash
  ```

  and spend some time browsing it (first, to see what's there) then reading bits that look interesting to you.

- Key disadvantages of bash: it suffers from a cryptic and cumbersome syntax, because it was originally developed as a command-line interface, and the programming constructs were glued on as an afterthought (the gluing process has been going on for four decades now, but backward-compatibility is a powerful force...) bash inherits syntax from 'sh', a command interpreter written at AT&T in the days when every ASCII character had to be chiseled on stone tablets in triplicate; thus bash uses characters economically. Three rules will help you to use bash effectively:

  1. Keep in mind that ; ; and ;; mean very different things. { and ${ mean very different things. [standing alone is a synonym for the command 'test'.
  2. When trying to figure out how bash parses a line, you need to follow the seven steps of command expansion in the same order that bash follows them: brace expansion, tilde expansion, variable expansion, command substitution, arithmetic expansion, word splitting, and filename expansion, in that order. No, really, I'm serious. Trying to read bash as pseudo-English leads only to frustration.
  3. When writing your own bash scripts, trial and error is usually the fastest method. Use the 'echo' command frequently, with appropriate variable expansions at each level, so you can see what bash thinks it is doing.

About awk or gawk: the only thing you absolutely need to know about gawk is that if you type the following command into the bash prompt, the file foo.txt will contain the M'th, N'th, and P'th columns from the file bar.txt (where M,N,and P should be any single digits):

```
awk '{printf("%s\t%s\t%s\n",$M,$N,$P)}' bar.txt > foo.txt
```

4.1 Installation

If you are on a unix system, bash, sed, gawk, and perl are probably already installed. If not, ask your system administrator.

If you are on Windows, download the Cygwin setup program from http://www.cygwin.com. Choose "install everything" unless you're very short on disk space; the default installation leaves out many tools that you really want to have available. Cygwin installation can be run as many times as you like; anything already installed on your PC will not be re-installed. In the screen that asks you which pieces of the OS you want to install, be sure to select (DOC)→(man), (Interpreters)→(gawk,perl), and (TEXT)→(less). I also recommend (Math)→(bc), a simple text-based calculator, and (Network)→(inetutils,openssh). You can also install a complete X-windows server and set of clients from (XFree86)→(fvwm,lesstif,Xfree86-base,XFree86-startup,etc.), allowing you to (1) install X-based programs on your PC, and (2) run X-based programs on any unix computer on the network, with I/O coming from windows on your PC. Setting up X requires a little extra work; see http://xfree86.cygwin.com.

If cygwin is installed in your computer in the directory c:/cygwin, it will create a stump of a unix hierarchy starting in that directory. For example, the directory c:/cygwin/usr/local/bin is available under cygwin as /usr/local/bin. Arbitrary directories elsewhere on your c: drive are available as /cygdrive/c/....

In order to use cygwin effectively, you need to set the environment variables HOME (to specify your home directory), DISPLAY (if you are using X-windows), and most importantly, PATH (to specify directories that should be searched for useful programs. This list should include at least /bin;/usr/bin;/usr/local/bin;/usr/X11R6/bin).

In order to use bash, sed, awk, and perl, you will also need a good ASCII text editor. You can download Xemacs from http://www.xemacs.org.

4.2 Reading

Manual pages are available, among other places, at http://www.gnu.org/manual. If your computer is set up correctly, you can also read the bash man page by typing 'man bash' at the cygwin/bash prompt.

- Read the bash manual page, sections: (Basic Shell Features)→(Shell Syntax, Shell Commands, Shell Parameters, Shell Expansions). (Shell Builtins)→(Bourne Shell Builtins, Bash Conditional Expressions, Shell Arithmetic, Shell Scripts). Alternatively, you can try reading the tutorial chapter in the O'Reilly bash book.
- Read the sed manual page, or the sed tutorial chapter in the O'Reilly 'sed and awk' book.
- 'gawk' is another name for 'awk' on GNU-based systems — the 'gawk' program has a few more features than the original 'awk' program. You may eventually want to learn gawk or awk, but it's not required. The section of the gawk man page called 'Getting Started with awk' is pretty good. So are the tutorial chapters in the O'Reilly 'sed and awk' book.

4.3 A bash/sed/awk example

Here is a shell script that trains monophone models, using HTK, on TIMIT speech data. This script assumes that you already have the following text files: (1) TRAIN.mlf, a time-aligned phonetic transcription of the training data, therefore it is possible to start with `HInit` instead of `HCompV`; (2) config_HCopy and config_HCompV contain configuration settings that specify how some of the HTK commands should behave.

```
#!/bin/bash
#
# This file is a record of procedures used to train monophone HMMs
#

##################################################
# Labeling

# Get a list of the monophones
awk '/[\.!]/{next;}{print $3}' TRAIN.mlf | sort | uniq > monophones.phf;

##################################################
# Feature file creation

# List the waveforms
ls timit/train/*/*/*.WAV > TRAIN.scp

# Specify the MFCC that should be created for each waveform
awk '{printf("%s %s2\n",$1)}' TRAIN.scp | sed 's/WAV2/mfc/' > TRAIN2.scp

# Convert the data
HCopy -T 1 -C config_HCopy -S TRAIN2.scp

##################################################
# HMM Training

# Get a list of just the MFCC files
awk '{print $2}' TRAIN2.scp > TRAIN1.scp

# Compute the flat-start means and variances
HCompV -I TRAIN.mlf -C config_HCompV -m -f 0.01 -S TRAIN1.scp -M hmm0 proto_nostreams.mmf
```

```
# Separate the macros and the HMM
head -3 hmm0/proto_nostreams > hmm0/macros;
cat hmm0/vFloors >> hmm0/macros;
tail +4 hmm0/proto_nostreams | sed 's/proto_nostreams/proto/' > hmm0/proto;

# Create each of the monophone models
mkdir hmm1;
for phn in `cat monophones.phf`; do
  HInit -I TRAIN.mlf -S TRAIN1.scp -H hmm0/macros -M hmm1 -T 1 -C config_HCompV -l $phn hmm0/proto;
  sed "s/proto/$phn/" hmm1/proto > hmm1/$phn;
done

# Re-estimate the monophone models
mkdir hmm2;
for phn in `cat monophones.phf`; do
 HRest -I TRAIN.mlf -i 100 -S TRAIN1.scp -H hmm1/macros -T 1 -M hmm2 -C config_HCompV -l $phn hmm1/$
done

# Concatenate all of the HMM models into an hmmdefs file
cat hmm2/? hmm2/?? hmm2/??? > hmm2/hmmdefs;

# Embedded re-estimation
for rep in 3 4 5 6 7 8 9 10 11 12; do
  echo "HERest: $rep";
  mkdir hmm$rep;
  pre=`echo $rep 1 - p | dc`;
  HERest -I TRAIN.mlf -C config_HCompV -t 250.0 150.0 1000.0 -T 0 -S TRAIN1.scp -H hmm${pre}/macros
done
```

5 Acoustic Model Training: Text Files

Here are some of the text files that you need in order to train an acoustic model using HTK.

5.1 Transcriptions: Master Label File

A *master label file* (MLF) in HTK contains information about the order and possibly the time alignment of all training files or all test files. The MLF must start with the seven characters "#!MLF!#" followed by a newline. After the global header line comes the name of the first file, enclosed in double-quote characters (); the filename should have extension .lab, and the path should be replaced by "*". The next several lines give the phonemes from the first file, and the first file entry ends with a period by itself on a line. For example:

```
#!MLF!#
"*/SI1972.lab"
0 1362500 sil
1362500 1950000 p
21479375 22500000 sil
.
"*/SI1823.lab"
...
```

In order to use the initialization programs HInit and HRest, the start time and end time of each phoneme must be specified in units of 100ns (10 million per second). In TIMIT, the start times and end times are specified in units of samples (16,000 per second), so the TIMIT PHN files need to be converted. The times shown above in 100ns increments, for example, correspond to the following sample times in file SI1972.PHN:

```
0 2180 h#
2180 3120 p
...
```

Notice that the "h#" symbol in SI1972.PHN has been changed into "sil". TIMIT phoneme labels are too specific; for example, it is impossible to distinguish "pau" (pause) from "h#" (sentence-initial silence) or from "tcl" (/t/ stop closure) on the basis of short-time acoustics alone. For this reason, when converting .PHN label files into entries in a MLF, you should also change phoneme labels as necessary in order to eliminate non-acoustic distinctions. Some possible label substitutions are s/pau/sil/ (silence), s/h#/sil/, s/tcl/sil/, s/pcl/sil/, s/kcl/sil/, s/bcl/vcl/ (voiced closure), s/dcl/vcl/, s/gcl/vcl/, s/ax-h/axh/, s/axr/er/, s/ix/ih/, s/ux/uw/, s/nx/n/ (nasal flap), s/hv/hh/. The segments /q/ (glottal stop) and /epi/ (epinthetic stop) can be deleted entirely.

All of the conversions described above can be done using a single perl script that searches through the TIMIT/TRAIN hierarchy. Every time it finds a file that matches the pattern `S[IX]\d+.PHN` (note: this means it should ignore files SA1.PHN and SA2.PHN), it should add necessary entries to the files TRAIN1.scp, TRAIN2.scp, and TRAIN.mlf, as shown above. When the program is done searching the TIMIT/TRAIN hierarchy, it should search TIMIT/TEST, creating the files TEST1.scp, TEST2.scp, and TEST.mlf. See the scripts `peruse_tree.pl` and `sph2mlf.pl` in `transcription_tools.tgz` for an example. Alternatively, you could create the scripts by doing the tree searching in bash, and use perl for just the regex substitution:

```
for d in `ls TRAIN`; do
  for s in `ls TRAIN/$d/`; do
   ls TRAIN/$d/$s/S[IX]*.WAV | ` |
     perl -e \
      'while($a=<>){$b=$a;$a=~s/.*\//data\//;$a=~s/\.WAV/\.mfc/;print "$b  $a\n";}' \
      >> TRAIN2.scp;
  done
done
```

Finally, just in case you are not sure what phoneme labels you wound up with after all of that conversion, you can use perl's hash-table feature to find out what phonemes you have:

```
awk '/[\.!]/{next;}{print \$3}' TRAIN.mlf | sort | uniq > monophones
```

The first block of awk code skips over any line containing a period or exclamation point. The second block of awk code looks at remaining lines, and prints out the third column of any such lines. The unix sort and uniq commands sort the resulting phoneme stream, and throw away duplicates.

5.2 Database File Listing: Script Files

A *script* file in HTK is a list of speech or feature files to be processed. HTK's feature conversion program, HCopy, expects an ordered list of pairs of input and output files. HTK's training and test programs, including HCompV, HInit, HRest, HERest, and HVIte, all expect a single-column ordered list of acoustic feature files. For example, if the file TRAIN2.scp contains

```
d:/timit/TIMIT/TRAIN/DR8/MTCS0/SI1972.WAV data/MTCS0SI1972.MFC
d:/timit/TIMIT/TRAIN/DR8/MTCS0/SI2265.WAV data/MTCS0SI2265.MFC
...
```

then the command line "HCopy -S TRAIN2.scp ..." will convert SI1972.WAV and put the result into data/MTCS0SI1972.MFC (assuming that the "data" directory already exists). Likewise, the command "HInit -S TRAIN1.scp ..." works if TRAIN1.scp contains

```
data/MTCS0SI1972.MFC
data/MTCS0SI2265.MFC
...
```

The long names of files in the "data" directory are necessary because TIMIT files are not fully specified by the sentence number. The sentence SX3.PHN, for example, was uttered by talkers FAJW0, FMBG0, FPLS0, MILB0, MEGJ0, MBSB0, and MWRP0. If you concatenate talker name and sentence number, as shown above, the resulting filename is sufficient to uniquely specify the TIMIT sentence of interest.

5.3 HMM Listing: PHF file

Almost all HTK tools require you to list the names of the acoustic models you wish to train. I have invented the extension "PHF" (phone file) for this type of file, just so that I can read my bash scripts a little more easily.

Usually, you have one acoustic model per monophone, or one acoustic model per triphone, therefore a PHF file for the automatic recognition of re-iterant speech (speech where your subject says "ma ma ma ma ma ma") might look like:

```
a
m
sil
```

5.4 Dictionaries

Essentially all dictionaries have the same format: each line contains one word, followed by its pronunciation. There are subtle differences in the encoding that MUST be considered if you try to merge dictionaries. It is useful to merge dictionaries because English contains more "words" (space-delimited orthographic units) than you thought it did; for example, the 39k Switchboard dictionary contains 10,500 words that are not in the 91k pronlex dictionary (not counting word fragments, of which there are an additional 6000). Some of these are word fragments, some are not.

- Pronlex (the 1997 Callhome English lexicon, LDC lexicon release number LDC97L20; 90988 entries) uses one-letter ARPABET notation, syllable boundaries marked with '.', primary and secondary stresses marked with "' and '+':

  ```
  administration .@dm+In.Istr'eS.In
  ```

- The Mississippi State Switchboard transcriptions include a dictionary with 38910 entries, in dict/switchboard.dict. Phones use 2-letter ARPABET; stress and syllabification are not marked

  ```
  administration ae d m ih n ih s t r ey sh ih n
  ```

- Radio Speech corpus includes a lexicon representing the pronunciation of speaker F1A (Disk1/f1a/labspeech/f1alab.p Words end in comma for some reason; phones are labeled in 2-letter ARPABET, syllables are marked by '*', primary and secondary stresses with +1 or +2:

  ```
  administration, ae d * m ih+2 n * ih * s t r ey+1 * sh ax n
  ```

- TIMIT includes DOC/TIMITDIC.TXT (6200 entries). Phones are in 2-letter ARPABET, syllable boundaries unmarked, primary and secondary stress marked with numbers:

  ```
  administration  /ax d m ih2 n ix s t r ey1 sh ix n/
  ```

HTK requires dictionaries to be in Switchboard format, except that HTK does not care what phoneme encoding you use (if you want to distinguish stressed and unstressed vowels, HTK has no problem with that).

If several lines start with the same word, they are interpreted as equally probable alternative interpretations of the same word, e.g.

```
object ax b jh eh k t
object aa b jh eh k t
```

Pronunciation probabilities may be specified as

```
object 0.3 ax b jh eh k t
object 0.7 aa b jh eh k t
```

Of course it would be better to treat these words as linguistically distinct, e.g.

```
object_v ax b jh eh k t
object_n aa b jh eh k t
```

. . . but in that case, you have to make sure that your transcriptions match your dictionary. None of the speech corpora mentioned in Section 1, except Radio Speech, include transcriptions with Part of Speech marked.

References

[1] B. S. Atal and S. L. Hanauer. Speech analysis and synthesis by linear prediction of the speech wave. *J. Acoust. Soc. Am.*, 50:637–655, 1971.

[2] Leonard E. Baum and J. A. Eagon. An inequality with applications to statistical estimation for probabilistic functions of Markov processes and to a model for ecology. *Bull. Am. Math. Soc.*, 73:360–363, 1967.

[3] Leonard E. Baum and George R. Sell. Growth transformations for functions on manifolds. *Pacific Journal of Mathematics*, 27(2):211–227, 1968.

[4] Jeff A. Bilmes. A gentle tutorial of the EM algorithm and its application to parameter estimation for Gaussian mixture and hidden Markov models. Technical Report TR-97-021, International Computer Sciences Institute, Berkeley, CA, 1997.

[5] Sarah Borys and Mark Hasegawa-Johnson. Distinctive feature based SVM discriminant features for improvements to phone recognition on telephone band speech. In *Proc. Interspeech*, pages 697–700, Lisbon, Portugal, 2005.

[6] Robert P. Carlyon and Shihab Shamma. An account of monaural phase sensitivity. *J. Acoust. Soc. Am.*, 114(1):333–348, 2003.

[7] Steven Davis and Paul Mermelstein. Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *IEEE Trans. Acoustics, Speech, and Signal Processing*, ASSP-28(4):357–366, August 1980.

[8] Mark J. F. Gales and Philip Woodland. Speech recognition the HTK way: Tutorial session. In *Proc. ICASSP*, 2006.

[9] Hynek Hermansky. Perceptual linear predictive (plp) analysis of speech. *J. Acoust. Soc. Am.*, 87(4):1738–1752, 1990.

[10] Hynek Hermansky, Dan Ellis, and Sangita Sharma. Tandem connectionist feature extraction for conversational HMM systems. In *Proc. ICASSP*, Istanbul, Turkey, 2000.

[11] Bin H. Juang, Stephen E. Levinson, and Man Mohan Sondhi. Maximum likelihood estimation for multivariate mixture observations of Markov chains. *IEEE Trans. on Information Theory*, 32(2):307–309, 1986.

[12] S. E. Levinson, L. R. Rabiner, and M. M. Sondhi. An introduction to the application of the theory of probabilistic functions of a markov process to automatic speech recognition. *Bell System Technical Journal*, 62(4):1035–1074, Apr. 1983.

[13] Louis A. Liporace. Maximum likelihood estimation of multivariate observations of Markov sources. *IEEE Trans. on Information Theory*, 28(5):729–734, 1982.

[14] Spyros Matsoukas, Jean-Luc Gauvain, Gilles Adda, Thomas Colthurst, Cia-Lin Kao, Owen Kimball, Lori Lamel, Fabrice Lefevre, Jeff Z. Ma, John Makhoul, Long Nguyen, Rohit Prasad, Richard Schwartz, Holger Schwenk, and Bing Xiang. Advances in transcription of broadcast news and conversational telephone speech within the combined EARS BBN/LIMSI system. *IEEE Trans. Audio, Speech and Language*, 14(5):1541–1556, 2006.

[15] L.R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 1989.

[16] L.R. Rabiner and R.W. Schafer. *Digital Processing of Speech Signals*. Prentice-Hall Inc., New Jersey, 1978.

[17] Xi Zhou, Xiaodan Zhuang, Ming Liu, Mark Hasegawa-Johnson, and Thomas Huang. HMM-based acoustic event detection with AdaBoost feature selection. In *CLEAR Evaluation and Workshop (Classification of Events, Activities and Relationships)*, Baltimore, May 2007.