TR-IT-0274

Auto Labeling

Wei Zhang

August 1998

ABSTRACT
This technical report describes a summer project to improve the quality of labeling of a speech database for use with the CHATR speech synthesiser. It shows that by use of appropriate post processing, the default labeling using HMM recognition techniques can be brought significantly closer to that produced by a human labeler.

Auto Labeling

Advanced Telecommunications Research Institute International
Interpreting Telephony Laboratories
Department 2
Wei Zhang
May 23, 1998

Introduction

CHATR is a multi-language synthesis system. It has a world renowned unit selection routine, and can produce excellent natural sounding speech. To do this, it requires the use of prosodically annotated speech databases consisting of many phoneme[1] units. Without a doubt, the quality of CHATR depends on the quality of these phoneme units. If these phoneme units contain noise or errors, then the output of CHATR suffers as a result.

[1] At time of writing, CHATR uses phoneme unit databases.

Background

CHATR contains over one hundred speech databases, some have been recorded at ATR, others have been recorded elsewhere, but all of them are created at ATR. The process for database creation is as follows:

1. Prepare text transcription.
2. Record spoken text.
3. Segment recorded speech with transcription into utterances.
4. Utterances are semi-automatically or hand labelled.
5. Phoneme labels are manually checked by labeller.
6. Create database.

If the database was not recorded at ATR, then the received data at the very least must contain a text transcription along with the recorded speech. The data is then processed according to the above steps.

Some of the problems associated with phoneme labeling include: audible portions of neighboring phonemes are included in another labeled phoneme, truncation of a phoneme such that it no longer sounds natural, silences not labeled, and phonemes are incorrectly labeled as another phoneme.

Objective

Understanding that consistent proper phoneme labels are very important to creating CHATR databases, a tool/method to check the phoneme labels is required. Such a tool/method has to be able to consistently check the phoneme labels as independently as possible from any human interaction, of course, some human interaction is necessary to ensure proper operation. This tool/method must also be flexible enough to be language independent as CHATR is a multi-lingual system.

Summary

The aligner[2] is a single C program using power and delta cepstrum as a basis for phoneme checks. This program was created to fix phoneme label problems such as misalignments, lack of silence labels, or incorrect phoneme labellings. Comparing the output of the aligner with hand checked phoneme labels, the average offset is between 5-10ms, except for vowels, which have an average offset of 10-15ms. The large offset for vowels is due to the fact that vowel transitions are normally smooth without a high delta cepstrum which is what the program uses to determine phoneme boundaries. An alternate method is to use the HTK package which uses Hidden Markov Models (HMMs). These HMMs must be first trained, then a Viterbi alignment is performed. The aligner performs better than HTK for more than half of the phonemes in the phoneme set, with the rest of the phonemes having better or equal performance with HTK. HTK tends to create phoneme boundaries that have large offsets from the original input label files, while because the aligner uses temporal proximity delta cepstrum/power information, it does not tend to move phoneme boundaries very far. Due to this property, running HTK, then the aligner does not seem to indicate any better performance than running the aligner alone. However, due to time constraints, an evaluation of running the aligner first, then HTK was not performed, but given opportunity, should be examined. Both the aligner and HTK suffer from the drawback of not being able to recognize incorrectly labelled phonemes, due to the difficulty of such a problem.

[2] The automatic phoneme label checker created at ATR, not to be confused with HTK's aligner.

Aligner

Objective

To create a program which will use the various speech characteristics of a speech file as a basis for phoneme segmentation.

Methodology

Introduction

Up to now, all phoneme verifications by hand were through the assistance of the Xwaves and Labeller's Workbench software. Upon examining the verification procedures, the frequency spectrum display in these programs was used heavily as an aid to determining inter-phoneme boundaries. A quick look at the Labeller's Workbench GUI indicates that not only is the frequency spectrum important but the power of the speech also plays a determining factor in aiding phoneme verification. This is the basis for much of the current verification tool, henceforth called the aligner, which is a single program written completely in C.

Methodology Used

The cepstrum is calculated from the utterance wave form[3]. The magnitude of the delta cepstrum is then calculated as follows:

$$DC_{mag} = \left(\sum_{k=0}^{N} \frac{\delta}{\delta t} c_k \right)^2$$

[3] Cepstrum data is calculated using HTK v1.5's HCode program.

Where c_k are the cepstral coefficients. The magnitude delta cepstrum gives a measure of how fast the cepstrum is changing, and thus serves as evidence to how fast the frequency spectrum is changing. The magnitude delta cepstrum can also be regarded as a probability, the higher the magnitude, the higher the likelihood of a inter-phoneme boundary. (Note that the opposite is not true, if a inter-phoneme boundary exists, then the delta cepstrum does not necessarily have to be high, as in the case of vowel to vowel boundaries) The magnitude delta cepstrum data is used exactly in this regard. To determine these boundaries, a relative threshold is determined. This threshold is automatically optimized for each utterance. The algorithm for this optimization is as follows: based on the number of phonemes that exist in an utterance label file, the aligner iteratively computes the optimum threshold such that the ratio of calculated inter-phoneme boundaries to the actual number of phonemes is fixed. This ratio is not one to one however, instead, it is around 1.3. This value has been determined by manual training on various speech databases. Once the threshold has been calculated, any magnitude delta cepstrum region which exceeds the threshold is evaluated and the local maxima is selected as a likely inter-phoneme boundary. Whether or not this boundary is used depends on whether there is a phoneme close by (a search window is used with the likely inter-phoneme boundary index as center). This process is repeated for each utterance.

Power spectral density (psd) function is used. The psd calculations produce power for 500Hz (note that this can be changed, see *CalculatePSD* on page 29) bandwidth units (e.g. 8kHz bandwidth data, psd calculations return an array of size 16, each element of the array representing spectrum magnitude for a 500Hz bandwidth section). Due to the limited sampling frequency input data (12kHz-16kHz) a much finer frequency scale could not be used. Once the power for each band are calculated, the power is normalized with a 12db/octave factor starting at 1kHz. This normalization is required because speech amplitude falls off with a 12db/octave[ii]. Any band that need to be filtered out can be filtered out (bandlimited noise), then the resulting power values are averaged. A threshold is calculated by a ratio of the difference between the maximum and minimum power for each utterance wave file (Threshold=[Max-Min]*Ratio+Min, where ratio is between 0 and 1). This ratio can be adjusted via the configuration file. To overcome the "microphone off silence" problem[4] mentioned in Methodology Drawbacks, two thresholds are used. One threshold is set at one third lower than the standard threshold. At any time, if the power drops lower than the low threshold value, the new threshold will be set equal to the low threshold value. If the power rises past the normal threshold value, the threshold is once again restored to the normal threshold value. This allows the program to adapt to the difficulty of having different silence levels within the utterance wave file (usually, there are two distinct silence levels). The minimum psd value used in determining the threshold is currently not the minimum psd value in the utterance, but rather an average of the start and ending silence periods. This was determined to be more robust than taking the absolute minimum value due to inconsistent recording level problems. A limitation is that the beginning and ending of an utterance must contain silence. See methodology drawbacks for more information. Once the power has been calculated, the following set of rules is used to determine how a phoneme is moved:

1. Silences/speech shorter than 20ms are ignored.
2. If phoneme boundary exists within search window at a high to low power crossing, move the phoneme boundary to this high to low crossing.
3. If phoneme boundary doesn't exist within search window at a high to low power crossing, store this time index and set shift flag.
4. At low to high power crossing, if shift flag is set, shift the time index of the first phoneme boundary in the silence period to the stored time index (step 2). Delete all phonemes left in the silence period. If phoneme previous to insertion point is silence, extend the silence, otherwise insert silence.
5. If the shift in step 4 exceeds the average phoneme duration, do not shift the phoneme.
6. If the phoneme following the silence is a plosive, do not insert or extend any silences.

[4] Some recordings contained two types of silences, one silence was recorded while the microphone was turned off, the other type of silence was recorded while the microphone was on but nothing was being said.

Duration statistics are used to determine whether a particular phoneme contains statistically likely durations. Initially, the minimum, mean, maximum, and standard deviation statistics were calculated for bi-phones (succeeding phoneme dependent) from the original unchecked labeled files, then as phonemes in each label file were evaluated and adjusted, the new duration was compared against the database of old durations. Because the new duration trends are different from the old duration trends, comparing the new phoneme durations against a database of new phoneme durations makes more sense. To compare phoneme durations against a database of new durations requires us to first create the database of new durations once all the phonemes have been adjusted, then go back and evaluate each phoneme duration against this new database. By these comparisons we can identify phonemes with durations shorter than a certain number of standard deviations away from the mean and take appropriate action.

Considered Methodology

Formants: formants are highly efficient, compact representation of the time-varying characteristics of speech[i], formants can also be used to identify certain types of vowels, however reliably detecting formants from low-level voiced speech is a very difficult process.

Triphone Duration statistics: There were not enough triphone units in databases to obtain meaningful duration statistics.

Delta delta cepstrum: Delta delta cepstrum can be used to determine the peak rate of change of the cepstrum (maximum delta cepstrum). However, using the simple rule, "if the central delta cepstrum value out of three sequential delta cepstrum values is the highest, then this is a local delta cepstrum maxima" is less computationally intensive yet also produces the peak rate of change of cepstrum.

Time domain magnitude power calculations were initially used to aid in the adjustment of inter-phoneme boundaries as well as determine silence segments within an utterance. However, a simple magnitude squared of the waveform data has proven to be inadequate for many reasons. First, on examination, low frequency amplitudes seem to be more intense than high frequency. Flanagan[ii] suggests that amplitudes fall off with a –12db/octave, thus before the power can be examined, a 12db/octave normalization factor must be applied. Furthermore, according to Olive[iii] and Fry[iv] speech starts somewhere around 70Hz ~ 80Hz, therefore a high pass filter is desired. The time domain magnitude calculations were also observed to be very susceptible to recording noise, such as the low frequency noise in nabi_A_040 in the FNA database. Not only was the power spectral density function necessary to perform the normalizations and filters, but it is also more robust against recording noise (low frequency noise, inconsistent recording levels,...etc.).

Methodology Drawbacks

The PSD uses a FFT algorithm. By nature of the mathematical computations required, the smaller the frame width (better time resolution, also known as wideband spectrum), frequency smearing will be at its worst, thus distinct bands of frequency calculated will contain values from nearby bands. We can eliminate this frequency smearing by increasing the frame width (narrowband spectrum), but at the cost of a coarser time resolution. For alignment purposes, a wideband spectrum is more desirable than a narrowband. It produces a finer time resolution alignment, and frequency smearing does not affect the results too greatly (the power for each frequency band is summed), however, for training purposes (as mentioned in the Suggestion section) a narrow band is preferable. Currently, a wide band spectrum is calculated.

Duration checking is used to identify phonemes with unusually short durations. However, duration has a very large standard deviation and as such, cannot be used to reliably identify bad phonemes. It was hoped that perhaps it can be used to identify mis-labelled phonemes as well, since different phonemes might have different durations; however, once again, due to the large deviation, duration is unsuitable for such uses. The only reason duration is still used in the aligner (note that it can be turned off) is due to lack of a more feasible algorithm within the allotted time limits. Please see section on suggestions for more information.

The algorithm for determining the psd threshold is currently sensitive to power fluctuations that might occur in the utterance waveform due to poor recording environment, noise, and inconsistent recording levels. In some cases, the recording contained two silence power levels: the absolute silence when recording has started, but the microphone was not turned on ("microphone off silence"), and the other silence level in which the microphone is on but nothing is being said ("microphone on silence"). This is difficult to detect without listening to the waveform since by computer observation it is difficult to distinguish between the "microphone on silence" from real speech as compared to the "microphone off silence". In other cases, the recording level was changed during recording such that a small portion of silences had a much lower absolute power level than the majority of the silences. This was noted to be a problem in most utterances to certain extents. Although, the algorithm is sensitive to such power fluctuations, it is made more robust with the use of power spectral density calculations, as compared to the time domain magnitude squared power function.

How to use the program

What's Required

The aligner requires the configuration file called **.walignrc** be in the home directory of the user running the program. The program finds out the home directory with the environment variable HOME. If this variable is not set, the program will attempt to search the current directory for the **.walignrc** file. A description of the configuration file follows in the next section. The aligner also requires pre-calculated cepstrum files. These can be calculated with HTK's HCode as follows:

```
HCode -m -n 16 -p 32 -e -f 1 -w 5.0 -h -s 0.1 -F ESPS -O HTK [input
filename] [output filename]

Parameter Explanations:
      -m    Output mel-frequency cepstral coefficients (cepstrum data)
      -n    Sets number of output coefficients
      -p    Sets the number of analysis order
      -e    Append the normalised log energy to the frame
      -f    Sets frame period (frame shift) to 1 ms
      -w    Sets window duration (frame width) to 5 ms
      -h    Apply a hamming window
      -s    Set normalised log energy scaling factor to 0.1
      -F    Set input source file format to ESPS
      -O    Set output file format to HTK
```

The cepstral order can be changed to the default of 12, but 16 produces a finer cepstrum resolution. The number of analysis order is recommended to be set at twice of the cepstral coefficient. Make sure that if these parameters are changed, that they be changed appropriately in the aligner configuration file as well. Note that the frame shift and frame width values here do not correlate with the frame shift and frame width values in the aligner configuration file. The frame shift and frame width values in the configuration file is used for delta cepstrum and power calculations only. Note that these cepstrum files are created in the same way as the creation of cepstrum files during a CHATR database build. However, these cepstrum files have a much finer time resolution (1ms frame shift, 5ms frame width) as compared to those cepstrum files created for CHATR. This fine resolution is required for precise phoneme boundary checks.

Hcode unfortunately does not take wildcards, so to process multiple files, the following script can be used:

```
      for i in "$@"
```

```
do
   NAME=`basename $i .d`
   HCode -m -n 16 -p 32 -e -f 1 -w 5.0 -h -s 0.1 -F ESPS -O HTK
   "[path]/$NAME.d" "$NAME.mcep3"
done
```

If the input file is not in ESPS format, then it must be converted to ESPS format. In the case of raw wave files, the following script will call `btosps` to convert all the files given on the command line to ESPS format.

```
for i in "$@"
do
   NAME=`basename $i .wav`
   btosps -c "" -f [sampling frequency] "$i" "$NAME.d"
   rm $i
done
```

The aligner also requires headerless utterance wave files for the power calculations and ESPS format label files. Ensure that the proper directory and extension values have been entered into the configuration file.

Configuration File

This will explain the entries in the **.walignrc** configuration file and what should go into to them.

`Lab:`	This is the directory which contains the original ESPS label files.
`LabExt:`	This is the extension of the original label files (without the period), if nothing is entered, a default of **.lab** will be assumed.
`MFCC:`	This is the directory which contains the cepstrum files.

`MFCCExt:` This is the extension of the cepstrum files (without the period), if nothing is entered, a default of **.mcep3** will be assumed.

`Align:` This is the directory which contains the label files that you want to check. Note that this can be the same entry as the **Lab** entry. This is provided so that if one wishes to check only a subset of the label files, then they can copy that subset to another directory.

`AlignExt:` This is the extension of those files that need to be checked (without the period). A default of **.lab** will be assumed if this entry is left blank in the configuration field.

`OutPhoneme:` This is the directory to place the output label files.

`OutPhonemeExt:` This is the extension of the output label files. Make sure that if you are storing the output label files in the same place as the input label files, that you do not use the same extension or the old files will be overwritten. If this entry is left blank, the default is **.new**.

`Wave:` This is the directory where the original wave files (headerless) are kept. The wave files are used to calculate the power spectral density.

`WaveExt:` Extension of the wave files, if left empty, the default is **.wav**.

`DBfile:` The duration statistics will be saved to this file in the current directory.

`WaveType:` This is either set to **WAVEFORM** (headerless binary) or **FEA_SD** (ESPS format). Currently not used. All input wave files are assumed to be headerless wave files.

`SamplingFreq:` This is the sampling frequency of the wave files in Hz. Usually it is 16000 or 12000.

`CepOrder:` This is the cepstrum order of the cepstrum data. This should be the same as the parameter used in the creation of the cepstrum files. Please ensure that these values are the same, and that the cepstrum data includes the energy.

`FrameWidth:` This is the frame width that will be used in the delta cepstrum calculations and power spectral density calculations. This is usually set to 10ms.

`FrameShift:` This is the frame shift that will be used in the delta cepstrum calculations and power spectral density calculations. This is usually set to 2ms.

`PowerThreshold:` This value is used to determine a power threshold which is used to detect silences. The threshold is the product of the inverse of this value and the difference between the maximum and minimum power above the minimum (i.e. (max-min)/PowerThreshold+min). This value is usually set at 4, but if you find that you have silences which are not totally silent, you can increase this value to set a more stringent threshold. Contrarily, if you find that too many phoneme's decay region is too long, and contain silences, then you can decrease this value to set a more lax threshold.

`PowerWindow:` This determines the size of the search window. Within the window, a search is conducted to find the closest phoneme and modify this phoneme's index to the computed interphoneme boundary. The window size extends from average duration*1/**POWERWINDOW** before the start of the phoneme to average duration*1/**POWERWINDOW** after the phoneme in the case of delta cepstrum check. In the case of the power check, the window center is based around the calculated inter-phoneme boundary, and not on the phoneme (e.g. If this value is set at 2, then the window size would be ½*average phoneme duration [for this database]*2, the window extends from ½*avg. duration before the computed inter-phoneme boundary to ½*avg. duration after the alignment point). If this value is increased, the aggressiveness of the algorithm to search for a nearby phoneme is lowered, while if this value is decreased, the algorithm becomes more aggressive. For ideal results, this value needs to be adjusted, but 4 seems to set a reasonable limit.

Note that for all numerical values above, decimal values are allowed, except in the case of **CepOrder**.

Executing the Program

Once the required files are been created and the configuration file has been changed, all that is required now to run the program is just run it!

Drawbacks

No recordings are perfect, they all contain some form of noise or static. Depending on the severity of the noise, it limits the ability of the aligner to perform consistent boundary checks. Past experience has shown that particularly, low frequency noise is a problem. The implementation of the spectral density function has increased the robustness of the aligner, but tolerance to noise can be further improved.

The aligner, currently, does not have a reliable method to determine whether a phoneme has been mis-labeled. The difficulty here lies in the fact that in order to be able to “recognize” a phoneme, it needs to perform speech recognition to a certain degree. This can only be achieved by training models on given databases to obtain a basis for comparison. Having said this, even though most of the problems encountered deal with boundary mis-placements, the relatively few mis-labeled phonemes contribute to most of the quality degradations, hence the elimination of such mis-labeled phonemes is critical.

There are several factors which can be adjusted in the **.walignrc** configuration file which adjusts the aggressiveness of the aligner. Since each database is different, an optimum setting for one database might not be an optimum setting for another (however they will likely be fairly close). The only current way to determine optimum settings for such values is by iterative approach. First, run the aligner with the default settings, then modify them appropriately. Although there might be ways of automating this process as the delta cepstrum threshold previous used to be a user settable setting.

The cepstrum files are computationally intensive and can take considerably time. For an 80 megabyte database, it takes approximately 1.5 hours on a Sparc 10 equivalent machine. However, Sparc 10s are fairly slow machines, compared to the UltraSparc. Creating the cepstrum files on the UltraSparc should take considerably less time.

Evaluation

Methodology

A database with both the original utterance label files as well as labellers hand checked utterance label files was selected. The output utterance label files are then compared with the labellers hand checked utterance label files. For an entire database, we record for each type of phoneme, the absolute time difference between the auto aligned phoneme boundary and the labellers hand checked utterance label file, averaged for occurrences of that phoneme.

Methodology Drawbacks

The judgement of phoneme boundaries is a very difficult task. One person's "perfect" phoneme boundary might be different from another. Furthermore, the perceptibility of two vowels that differ by 10ms is absolutely minimal. The question that must be ultimately asked when evaluating phoneme boundaries is whether during synthesis, when the phonemes are being used, any difference in quality can be detected. Perhaps a better evaluation would be to take a database that has been automatically checked with the aligner, another without, and compare listening tests with CHATR. Such evaluation would be more beneficial; however, due to time constraints, this is currently not feasible. One thing is guaranteed with the use of an automatic tool, consistency.

I would just like to add that in consideration for an evaluation method, a different approach than the one above was initially considered. It consisted of calculating the average percentage moved of the difference between the original labeling and the manually checked labels. (i.e. If the original phoneme index is 1.100 sec, the manually checked phoneme label at 1.090 sec, and the auto checked phoneme at 1.095 sec, the percent moved would be (1.095-1.090)/(1.100-1.090) or 50%) By this method, the magnitude of the move is considered, so that even if the auto checked phoneme was 50ms away from the hand checked phoneme, if the time index change from the original label to the hand checked label is 500 ms, the percent change is 90% as compared to another auto checked phoneme, where it too has a 50ms offset from the hand checked label but the difference between the original and the hand checked is only 100ms, then this percentage change would be 50%. Clearly, the phoneme with the 90% change benefited more with the aligner than the one with the 50% change. Although this takes the magnitude into account, the end result is in a ratio, and thus, do not know what the average time offset is.

Results

The following data was gathered from the new FNA database. The speaker is Japanese female voice. The original label files are in `/dept2/work26/pi/data/chatr_dbs/T_TEST/T503/LBL.old`, the hand checked labels are in `/home/as75/nnagaoka/T_TEST/T503/LBL` and the wave files in `/dept2/work26/pi/data/chatr_dbs/T_TEST/T503/WAV`. The program used to create these results can be found in the Appendix.

The configuration settings used is as follows: `CepOrder=16, FrameWidth=10, FrameShift=2, PowerThreshold=4, PowerWindow=4` with no duration checking. Note that these are the normal default settings. (These settings were experimentally determined to have produced the best results for other databases)

The table lists the phonemes, the minimum time difference between hand checked and auto aligned, the average time difference, maximum time difference, phoneme occurrence in this database, and the standard deviation.

The aggressiveness of the algorithm differs in the PowerWindow setting, as well as whether the setting for a phoneme to move more than the average phoneme duration for a particular database was set or not. For less aggressive settings the PowerWindow value was set at 6, with the duration limited phoneme move parameter set to half the average phoneme duration. The aggressive setting used a PowerWindow of 2, and removed the average phoneme duration move parameter setting.

Phoneme	Minimum Offset (ms)	Average Offset (ms)	Maximum Offset (ms)	Phoneme Occurrences	Standard Deviation (ms)
#	0	5.666	1170.901	1474	31.619
I	0	11.497	112.26	275	15.819
N	0	12.206	161.27	874	17.527
U	0	12.751	101.91	268	15.048
a	0	13.161	487.68	3618	29.452
b	0	4.511	13.3	370	3.192
cch	1	3.036	7	12	1.81
ch	0	3.097	17.8	252	2.898
d	0	6.118	21.54	612	4.024
dd	0.191	4.725	10.78	4	4.324
e	0	17.889	561.2	1990	30.989
f	0	6.707	99.923	151	12.509
ff	9	9	9	1	0.002
g	0	5.113	105	735	9.701
h	0	5.088	91.182	461	8.252
i	0	11.699	391.3	2476	23.454
j	0	6.028	28.67	348	5.36
k	0	5.673	185.5	1431	11.074
kk	0	3.921	10	55	2.586
m	0	4.667	44.52	823	5.046
n	0	5.105	175	1298	7.992
o	0	12.731	437.296	3693	24.746
p	0	6.43	83.239	67	10.99
pp	0.216	7.447	19.77	39	5.758
r	0	6.264	28.88	1225	4.097
s	0	5.671	125.78	683	9.747
sh	0	4.039	73.82	559	8.666
ss	1	2.347	5	9	1.14
ssh	0	5.545	43.82	33	7.39
t	0	6.436	132.21	954	8.183
ts	0	3.885	77.98	251	9.054
tt	0	6.278	82.71	239	9.669
tts	5.28	5.28	5.28	1	NaN
u	0	11.808	259.37	2163	16.828
w	0	19.638	80.07	407	13.756
y	0	8.339	146.49	676	10.927
z	0	4.58	29.113	215	5.018

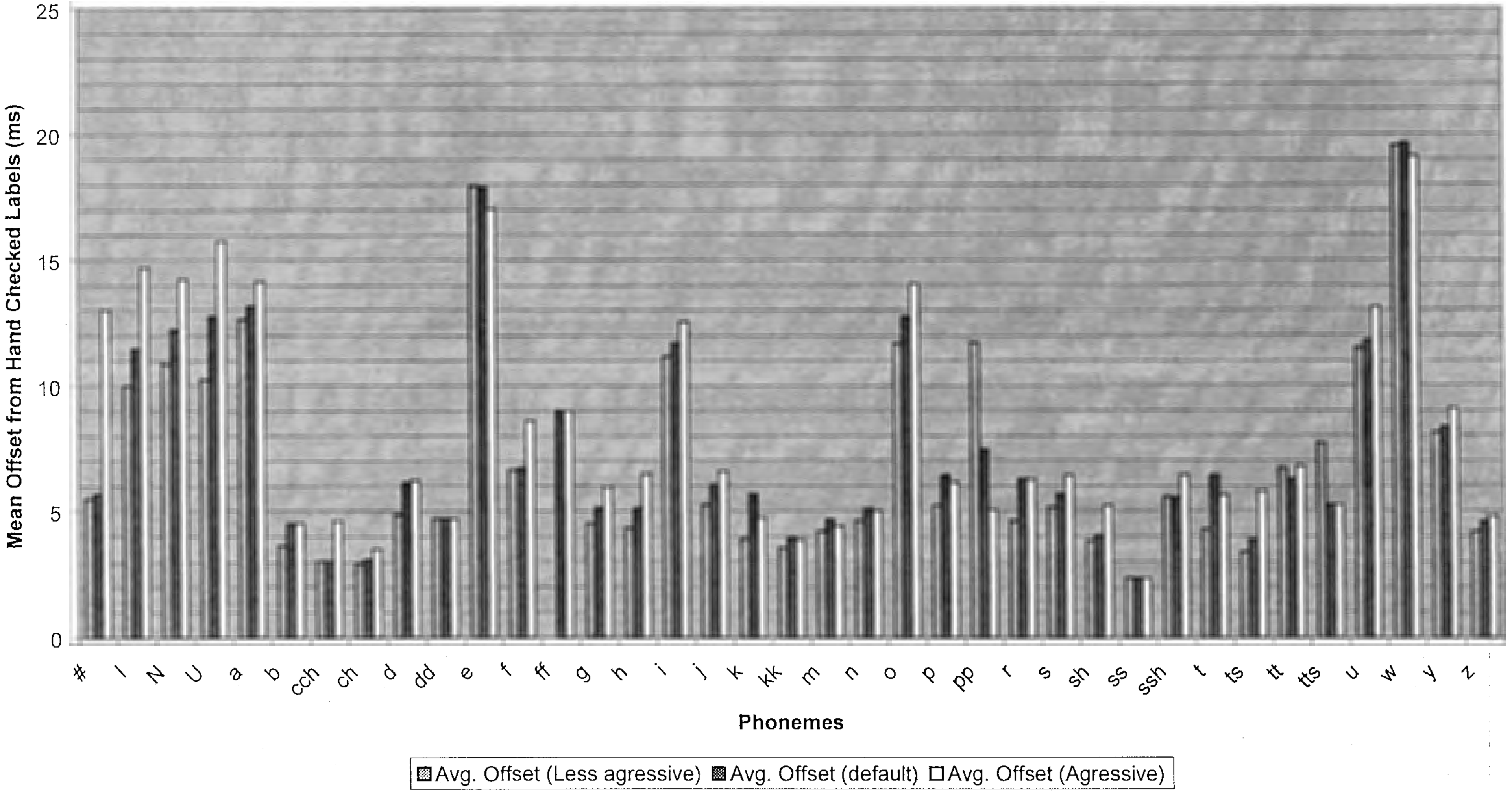
The following table is the result of the aligner using slightly aggressive parameter settings compared with the default settings.

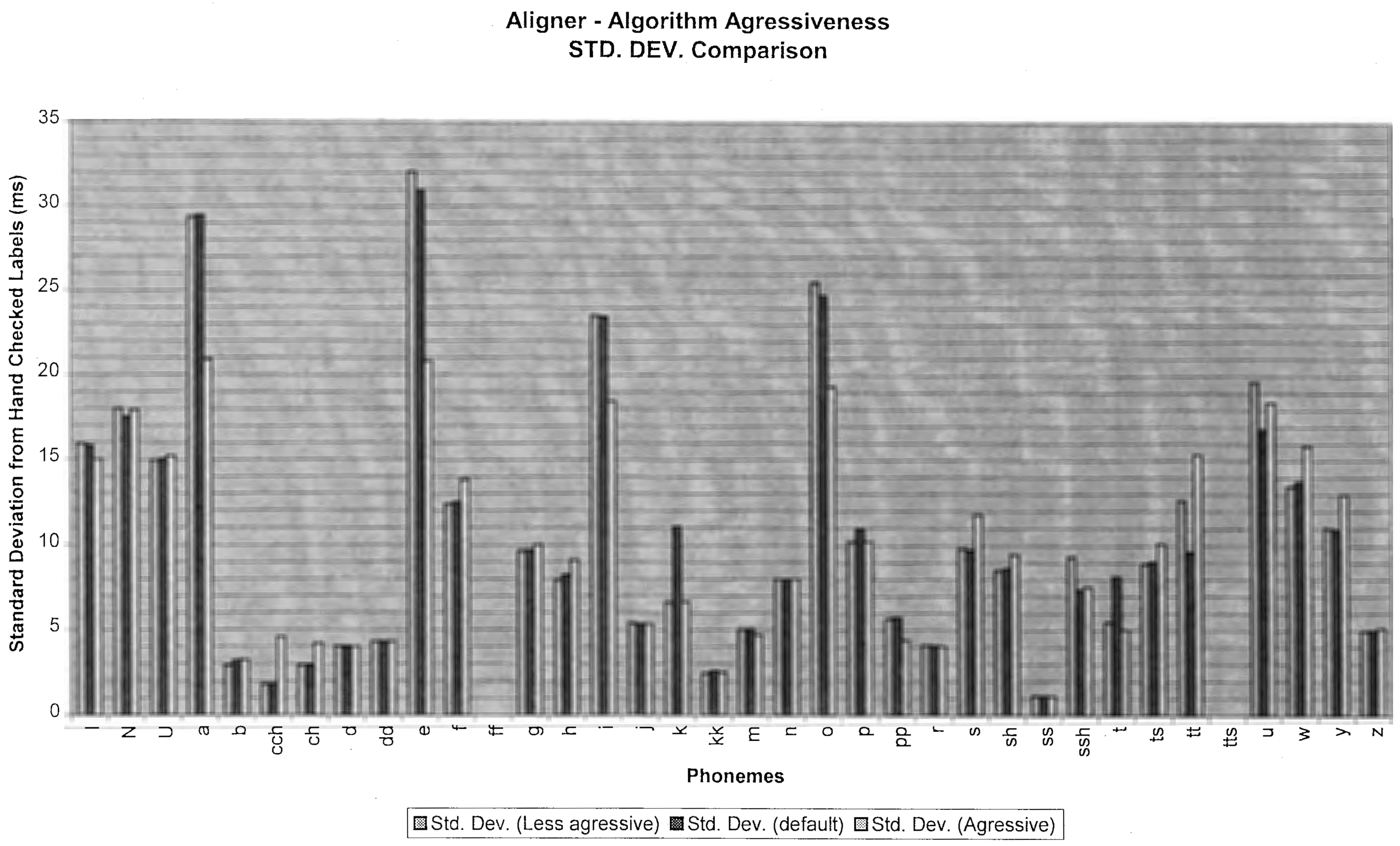
Phoneme	Minimum Offset (ms)	Average Offset (ms)	Maximum Offset (ms)	Phoneme Occurrences	Standard Deviation (ms)
#	0	12.997	1170.901	1515	52.529
I	0	14.712	63.17	275	15.016
N	0	14.251	161.27	876	17.933
U	0	15.779	101.91	267	15.23
a	0	14.158	487.68	3616	20.918
b	0	4.535	18	370	3.236
cch	1	4.619	19	12	4.609
ch	0	3.507	40.28	249	4.189
d	0	6.225	22	612	4.017
dd	0.191	4.725	10.78	4	4.324
e	0	17.049	319.56	1990	20.815
f	0	8.614	99.923	151	13.833
ff	9	9	9	1	0.002
g	0	5.969	105	735	10.021
h	0	6.481	91.182	460	9.112
i	0	12.563	267.78	2475	18.472
j	0	6.591	28.67	348	5.346
k	0	4.738	114.514	1424	6.666
kk	0	3.886	10	55	2.536
m	0	4.419	43.52	823	4.719
n	0	5.017	175	1297	8.012
o	0	14.066	347.553	3692	19.376
p	0	6.152	83.239	67	10.234
pp	0.216	5.05	15.44	39	4.425
r	0	6.277	38.44	1224	4.052
s	0	6.438	165	681	11.799
sh	0	5.244	60.92	560	9.487
ss	1	2.347	5	9	1.14
ssh	0	6.454	43.82	33	7.576
t	0	5.655	132.21	953	5.043
ts	0	5.805	77.98	251	10.12
tt	0	6.802	174.77	239	15.372
tts	5.28	5.28	5.28	1	NaN
u	0	13.153	360	2158	18.443
w	0	19.114	101.232	406	15.884
y	0	9.106	179.969	675	13.025
z	0	4.804	25	215	5.147

The following table lists the result of the aligner using slightly less aggressive parameter settings compared with the default.

Phoneme	Minimum Offset (ms)	Average Offset (ms)	Maximum Offset (ms)	Phoneme Occurrences	Standard Deviation (ms)
#	0	5.495	1170.901	1515	31.143
I	0	10.001	112.26	275	15.95
N	0	10.902	161.27	872	18.025
U	0	10.261	101.91	267	14.961
a	0	12.663	487.68	3614	29.357
b	0	3.647	13.3	368	2.915
cch	0	3.036	7	12	1.81
ch	0	2.92	18.64	252	2.947
d	0	4.864	21.54	611	4.032
dd	0.191	4.725	10.78	4	4.324
e	0	17.962	561.2	1988	32.06
f	0	6.647	99.923	151	12.391
ff	0	0	0	1	0
g	0	4.515	105	733	9.671
h	0	4.351	91.182	461	7.921
i	0	11.17	391.3	2471	23.521
j	0	5.246	27.292	347	5.408
k	0	3.903	110.5	1427	6.631
kk	0	3.576	8.88	55	2.447
m	0	4.205	44.52	823	5.041
n	0	4.635	175	1297	8.001
o	0	11.688	437.296	3687	25.519
p	0	5.221	83.239	67	10.218
pp	0	11.727	24.5	40	5.675
r	0	4.617	28.88	1222	4.114
s	0	5.161	125.78	683	9.829
sh	0	3.834	73.82	559	8.567
ss	1	2.347	5	9	1.14
ssh	0	5.572	55.82	33	9.337
t	0	4.3	132.21	950	5.487
ts	0	3.409	77.98	250	8.949
tt	0	6.716	82.71	239	12.647
tts	7.72	7.72	7.72	1	NaN
u	0	11.51	431.001	2158	19.656
w	0	19.558	80.07	407	13.476
y	0	8.129	146.49	674	11.035
z	0	4.201	29.113	215	4.991

Aligner - Algorithm Agressiveness
Mean Offset Comparison




Aligner - Algorithm Agressiveness
STD. DEV. Comparison
Standard Deviation from Hand Checked Labels (ms)
35
30
25
20
15
10
5
0
I
N
U
a
b
cch
ch
d
dd
e
f
ff
g
h
i
j
k
kk
m
n
o
p
pp
r
s
sh
ss
ssh
t
ts
tt
tts
u
w
y
z
Phonemes
▨ Std. Dev. (Less agressive)
■ Std. Dev. (default)
▨ Std. Dev. (Agressive)

Standard Deviation	*Mean Offset*	*Explanation*
increase	increase	Labels have wider spread, poor performance
decrease	unchanged/decrease	Phoneme labels are converging towards phoneme boundary near hand checked boundaries, desired.
decrease	increase	phoneme labels are converging towards boundary not near hand checked label, some cases desired.
increase	unchanged/decrease	Labels have wider spread, poor performance.

The graph shows that phonemes such as a, e, i, o, pp, and ssh have better performance when the algorithm was made more aggressive, whereas phonemes such as cch, ch, s, tt, u, w, y did not perform as well with an aggressive algorithm. One reason for this behaviour is the lack of phoneme specific checks of the algorithms. The algorithm is mostly phoneme independent with some exception of plosives which are currently used to aid alignment checks.

From the graph above, it is also clear that vowels have a much larger offset from hand checked labels, on average between 10ms to 15ms away from hand checked labels. This is due to the fact that vowel phoneme boundaries have fairly smooth cepstral transitions with little delta cepstrum near this area. This also means that vowel transitions are not easily detectable audibly compared with plosives or other phonemes, so a difference of ~10ms is not as crucial.

Programmer

Introduction

This section describes each function of the aligner in more detail. It assumes the reader has a general programming knowledge (pointers, structures,...etc). It might be helpful for programmers wishing to understand the program to have a copy of the source code while reading this, but it is not necessary for those who are simply interested in the basics of the aligner.

Description

When the aligner is run, and if **DEBUG** is defined, it opens a file called **walign.log**. This file will contain a trace of what the aligner is doing while running. Every time a function is called or every time a function exits, it will be logged to this file.

The first function call in the aligner is *readConfiguration()*. *readConfiguration* takes no parameters and returns no parameters. It reads in the **.walignrc** configuration file in the user's home directory. A **while** loop reads in every single line of the configuration file. After reading in each line, it compares it with any known fields, if it makes a match, it records the entry for that field, if the entry is invalid, the default is used, in the case where there are no defaults (i.e. the directory entries) it complains and terminates with an error code. Note that if some mandatory fields have been removed from the configuration file, the *readConfiguration* function will not terminate with an error; however, unpredictable results can occur.

Return from *readConfiguration* we check to see if **WAVETYPE** has been declared other than **WAVEFORM**, if so, then terminate with a message to convert ESPS files to headerless wave first.

getopt is a ESPS function call which aids in the parsing of command lines; however, it has some serious flaws such as the inability to pass more than one parameter per command line switch. This was the main reason a configuration file was created. *getopt* is only used to trap any command line switches and call *ReportUsage* which displays a simple message indicating that all settings are to be set in the configuration file.

readDatabase takes as input, a filename string and returns a pointer to **pNode,** which is a data structure which contains a phoneme entry, its time index, next phoneme, and statistical characteristics among some programming information (next, previous pointers...etc). *readDatabase* also reads in the average phoneme duration which is calculated and stored as a float in the beginning of the duration statistic database file. This average duration is used to determine the size of the search windows in the delta cepstrum and power routines. If duration statistics are not going to be used and routines removed from the program, then the search window size needs to be determined another way, either hard coded or the average duration stored somewhere else. The input filename string is used to point to the duration database file. The function tries to open this file, and read in first, the average duration value. If it could not open the database file, it calls the function *MakeDatabase,* passing to it, the directory which contains all the label files, the label file extensions, and the database filename. *MakeDatabase* will create the database, and upon return, *readDatabase* will read in all the phonemes and its corresponding duration. The function *balanceTree* is called to balance the resulting binary tree. The database file is sorted according to phoneme pairs, and so *balanceTree* will ensure we have a O(log(n))[5] search time. Otherwise, without *balanceTree,* we would have, not a binary tree, but basically a tree with a linked list configuration with a search time of O(n). After the phoneme information is read, *readDatabase* returns the head of the database tree.

[5] Big "O" notation. Defined as the order of the running time, i.e. O(log(n)) implies the algorithm has a running time order of log(n), where n is the number of elements.

MakeDatabase creates a text file database of phoneme pairs (phoneme and its following phoneme) along with the duration statistics. It reads each label file in order, until no more label files exist, and calculates the duration statistics. It creates a binary tree, and calls the function *tree* to perform insertions into this tree. Tree insertions are ordered based upon the phoneme pair, so the phoneme pair c-d would be inserted in the left child node of a node containing d-a, or inserted in the right child node of a node containing c-z. Once all label files have been read, then the binary tree containing phoneme pairs and its corresponding duration statistics are written out to a file through the function call *printDatabase*. The function then exits with no return value.

tree traverses a binary tree searching for any given phoneme pair, if the pair cannot be found, then it inserts the phoneme pair into an empty child node such that phoneme pair order is maintained. Note that *tree* is a recursive function, it calls itself while searching. The resulting binary tree that is created is dependent on the input data, the worse case is a linked list, the best case is a prefect binary tree, with log[base2](n) levels given n nodes.

printDatabase takes as input, the pointer to the head of the duration database node, and the file pointer to write the database to. *printDatabase* is a recursive function which traverses the database binary tree in alphabetically phonetic order, calling *printNode* to print out each node.

printNode takes as input the pointer to a phoneme node, and a file pointer. It prints out of the phoneme information to the file pointed to by the file pointer.

nextFile takes as input, a directory and a file extension. It returns the first file with the given extension in the directory, or in the case of subsequent calls, it returns the next file with the given extension. Upon exhausting the file list, it returns NULL. *nextFile* remembers the last file returned for any given directory, so if *nextFile* is called with extension ABC, then with extension DEF, any files with extension DEF that were passed during the search for the file with extension ABC will no longer be in the search space, similarly, the subsequent call to *nextFile* for files with extension ABC will no longer be able to find any files with such extension that were already looked at during the search for files with DEF extension. This is a limitation of the memory algorithm used within *nextFile* but should suit our purposes in this case, since the aligner does not search any one directory for files of different extensions before exhausting a directory's file list.

CheckAlign takes as input, the pointer to the duration database, and the utterance label file to check. When multiple files are to be checked, a **while** loop in the main function calls *CheckAlign* for each label file. In *CheckAlign*, the label file to check is read via *readPhonemeFile*, which returns a linked list, each node representing the phoneme and its time index. *rawfilename* is called to strip off any paths or extensions from a character string (the utterance label filename). Cepstrum filenames and wave filenames have the same root filename as the utterance label files, the only difference lie in the extensions. *CalculatePSD* is called to calculate the power spectral density. *CheckPower* then uses the psd data to realign phonemes or insert silences. *destroyPowerList* cleans up the memory used by the psd data. After the psd data is used, *CalculateDcep* calculates the delta cepstrum, followed by *CheckDcep* which uses the calculated delta cepstrum to further realign the phonemes. Similar to *destroyPowerList*, *destroyDcepList* is used to free up the memory taken by the delta cepstrum data. At this point, *OutputPhonemeFile* is called to write the newly realigned phoneme file. *destroyList* then frees the memory occupied by the utterance linked list. *CheckAlign* then exits with no return values.

readPhonemeFile takes as input, the label filename. It reads the phoneme and creates a linked list, returning the head of the linked list.

rawfilename takes the input string and searches for slashes which might indicate paths preceding filenames, if any is found, it performs some pointer manipulation to strip off the preceding path. Similarly, it strips off extensions by searching for periods.

CalculatePSD takes as input the raw filename. *CalculatePSD* then searches the appropriate directory and loads the wave file with the *ALReadWholeWaveFile* function call. The required number of data per psd sample point is first copied to a buffer, then this buffer is passed to *spctrm* the psd algorithm. Note that the parameters to the psd algorithm, **m** and **k** are related, and depending on the values of **m** and **k**, one can change the output (see description for *spctrm* for more information). However a strict restriction on **m** is that it must be a integer power of 2, this is required by the fast Fourier transform algorithm that is used in *spctrm*. Output of *spctrm* is contained in the array **p**. Each element of the array contains the power for a specific frequency band. The bandwidth of the frequency band is determined by the global define **PSDSTEP**. The power of these bands are then normalized by the function *freqNormalize* and summed. Note that since the array **p** contains band limited power, one can perform any combination of low/mid/high pass filters by simply not including a specific element of the array. In some cases, there are low frequency noise in the recorded waveform files, in such cases, it is filtered out. There is also evidence[v] that voice does not start at 0Hz, but rather somewhere around 100Hz, but further investigation is required. The resulting sum is converted to dBs and stored. Power calculations are calculated on a frame width of **FRAMEWIDTH** milliseconds and a frame shift of **FRAMESHIFT** milliseconds. These two parameters are user defined in the configuration file. While power calculations are being taking place, the minimum and maximum power values are recorded, these values are later used to determine the power threshold. Due to the various levels of silences that exist in the waveform, the minimum power value is calculated as the average of the beginning and ending silence periods. This of course assumes that in the beginning and end of an utterance, it is silent. Note that a normal distribution could be used to help determine the maximum and minimum power levels, but there are serious drawbacks. Please see the comments in the aligner in function *CheckPower* for more details. *CalculatePSD* returns the data in the **PowerDataFile** data structure.

ALReadWholeWaveFile takes as input a filename. This function reads this wave file and returns a pointer to the file. It also returns the number of samples that were read.

In successive calls to this function, if the input filename is the same as the previously requested file, then the same pointer to the file is returned.

spctrm takes as input: a pointer to the wave data, parameters **m**, **k** and **ovrlap**. **ovrlap** is either 1 or 0, and controls the type of result that is desired. For our purposes, **ovrlap** will always be set at 1. The parameter **m** determines how many frequency bands will be calculated (i.e. if **m** is 16 with a 16kHz sampled data [Nyquist frequency of 8kHz], 16 frequency bands will be calculated each with a bandwidth of 500Hz). The parameters **m** and **k** are related mathematically (see program for more details), with the restriction that **m** be a integer power of 2. The global define **PSDSTEP** specifies the frequency bandwidth of each band, however, this is not guaranteed as **m** is always rounded up to the nearest integer power of 2. **k** is calculated to satisfy the mathematically relationship. For input waveforms that are sampled at 12kHz or 16kHz, there is limited flexibility in choosing **PSDSTEP**, and care is advised so that **k** is not invalidated. There is no check in *CalculatePSD* for valid values of **k** (future work?).

freqNormalize takes as input the frequency of the power, and the power in magnitude (not dBs). It normalizes the power using a 12dB/octave scale according to Flanagan and returns the result.

CheckPower takes as input: the utterance linked list, **PowerDataFile** data structure, and database linked list. This function calculates two thresholds, a normal threshold and a low level threshold, and while examining the calculated power data (from either *CalculatePSD* or *CalculatePower*), examines the inter-phoneme boundaries and makes any adjustments if necessary. The outside loop examines the power data watching for any threshold crosses, and also changes the threshold to either the low threshold or the normal threshold if the power drops below the low threshold or rises past the normal threshold respectively. Once a threshold crossing occurs, another loop examines the next **SHTWINSIZE** milliseconds (step size of **SHTWINSTEP**) for possible threshold crossings. This loop prevents any short silences less than **SHTWINSIZE** from being labelled. Note that only short silences are watched for and that short phonemes are not affected. Once we have determined that we do not have a short silence, the program searches through the utterance linked list for the closest phoneme, it adjusts the phoneme to the threshold crossing time index and the process repeats. There is a window size bounded by **windowBegin** and **windowEnd** which limit how far from the threshold crossing we will search for a phoneme. If a phoneme cannot be found within the window, then this threshold crossing time index is either recorded or a pause is inserted depending on whether it is a speech to silence crossing or a silence to speech crossing respectively. In many cases, for speech to silence crossings, phonemes have been labeled such that it includes part of the trailing silence. In this case, although a phoneme was not found in the search for the speech to silence crossing (phoneme was labeled with a much longer duration) the closest following phoneme will be shifted to this new time index. In the case where the time index of a phoneme needs to be shifted to a new location, the shift is limited so that it only takes places if the shift does not exceed the average phoneme duration (approx. 100ms for FNA database). Once all adjustments have been made, if a phoneme is found to be labeled within a silence (according to the power calculations) then it will be deleted. Once all adjustments have been made, *CheckPower* returns the head of the utterance linked list.

destroyPowerList takes as input the **PowerDataFile** data structure. It frees the memory used by the data structure.

CalculatePower is the function which calculates the time-domain magnitude power of a wave form. The code is similar to that of *CalculatePSD* except for the call to *spctrm*. Please refer to the code for more information. This function is no longer used but included for the sake of completion.

CalculateDcep takes as input the raw filename. It reads the required cepstrum file from the **MFCC** directory, and calculates the derivative of the cepstrum. The derivatives are calculated through a weighted window on each cepstrum coefficient, the coefficients are then summed and squared to produce the final delta cepstrum value. The weight window is achieved through the *ALLSRweight* function call. The function returns the delta cepstrum data in the **dCepDataFile** data structure.

CheckDcep takes as input: the utterance linked list, pointer to **dCepDataFile** data structure which contains the delta cepstrum data, and the database linked list. The function first determines how many phonemes exist in the utterance, it then iteratively starts with a high delta cepstrum threshold, and counts the resulting likely inter-phoneme boundaries. If the number of inter-phoneme boundaries is lower than the number of phonemes multiplied by the manually trained ratio, it lowers the delta cepstrum threshold and repeats the process. This process continues until the number of inter-phoneme boundaries equal or exceed the product of the actual number of phonemes with the manually trained ratio. Note that during the inter-phoneme boundary counting process, it records the time indices and stores the result into an array called **dcepindex**. The size of **dcepindex** is initially calculated by the number of phonemes multiplied by a scaling factor that is larger than the manually trained ratio. However, this does not guarantee that **dcepindex** is sufficiently large enough to hold all resulting indices. The size of **dcepindex** is stored in **indsize**, and if the number of indices exceeds **indsize**, then the size of **dcepindex** is doubled (realloc). Once all the inter-phoneme boundaries have been stored in **dcepindex**, a loop goes through the utterance linked list (similar to *CheckPower*), at each phoneme, checking to see if any delta cepstrum calculated inter-phoneme boundaries exist within a window. The size of this window is the same as that used in *CheckPower*, the average phoneme duration scaled by **POWERWINDOW,** which is a user settable parameter in the configuration file. If it does find any inter-phoneme boundaries within this window, it adjusts the phoneme to the boundary time index. Note that if the global define **DEBUG** is defined, then at the end of this function, all the delta cepstrum inter-phoneme boundaries will be added to the utterance list. This is useful when debugging and exact delta cepstrum inter-phoneme boundaries need to be examined. In debug mode when viewing the newly aligned utterance with Xwaves, all phonemes labeled "DC" are inter-phoneme boundaries calculated from delta cepstrum. Note that in debug mode, the duration check must be turned off, since duration cannot be checked for phonemes labeled “DC”. *CheckDcep* returns the head of the utterance linked list.

destroyDcepList takes as input the **dCepDataFile** data structure and frees the memory used by it.

OutputPhonemeFile takes as input the utterance linked list as well as the raw filename. It creates the new utterance label file by traversing the utterance linked list writing each phoneme and its corresponding time index into an ESPS format label file.

destroyList takes as input the utterance linked list and frees the memory associated with it.

ExamDuration takes as input the utterance linked list, and a pointer to the duration database. If **STDLIM** was not set as “-1” in the configuration file (i.e. –1 turns off duration checking), this function will traverse the utterance linked list checking each phoneme’s duration with the function call to *CheckDuration. CheckDuration* returns a value indicating the number of standard deviations away from the mean for a phoneme’s duration. Using the value set in **STDLIM** as a deviation cutoff, all phonemes with durations outside this range will have it’s phoneme changed to **BADPHONEMEPWR**. **BADPHONEMEPWR** is currently defined as “X”. (Note that instead of replacing the phoneme with X, the aligner currently appends X to the phoneme. This is to assist debugging.) *ExamDuration* then calls *OutputPhonemeFile* to write the new utterance linked list.

CheckDuration takes as input a pointer to a phoneme node, as well as a pointer to the duration database. It uses the value stored in **cPho** in the phoneme node to search through the database, and compare the duration statistics. It then calculates how many deviations the current phoneme’s duration is away from the mean. *CheckDuration* returns this deviation value.

Suggestions

One way to detect incorrectly labeled phonemes, without having to resort to speech recognition, is to train phoneme models based on previously labeled data. There are many different possibilities to train these models, one method, is to use the spectral characteristics of phonemes. Different phonemes have different spectrograms, thus with the use of the power spectral density function, (which gives us frequency bandlimited power) we can use the statistical mean of the power spectral density information as a basis for comparison.

However there are several drawbacks, the first is that since the psd data is a very rough spectral representation (each frequency band is around 500Hz), it most likely cannot distinguish between spectrally closely related phonemes, e.g. different vowels, or different plosives, however, it probably can distinguish between vowels and fricatives, or plosives and fricatives. The psd data will also require a fair portion of space, currently each psd data frame (one psd data value) occupies **FRAMEWIDTH** milliseconds (10 milliseconds by default). Assuming an average of 100 milliseconds for each phoneme, each phoneme will require 10 frames, each frame an array of size equal to **p** mentioned above. Another drawback is that since the original data is unchecked, severe misalignments can cause the creation of models that are too generalized, reducing robustness.

Having mentioned the drawbacks, the psd algorithm is already in use, so to implement the psd model training does not require a significant amount of effort.

The power threshold is currently set as a ratio in the configuration file by the user. The default was chosen from past experience. The actual power threshold varies depending on the input file's maximum and minimum powers; however, there might exist an absolute power threshold which corresponds to the human threshold for audibility or inaudibility, this needs to be further investigated. If such a threshold exists, then the power threshold used in the aligner program can be set independently of input files.

Future Work

Handle ESPS format files. ESPS files have a fixed size header, that needs to be removed. From the header, user parameters such as sampling frequency can be determined.

The configuration file input routines does not check for the absence of parameters. (If the parameter **FrameWidth** was left out altogether from the configuration file) In such cases, the aligner may fail at a later point with error messages. A check at the end of the configuration file input routines can eloquently avoid such error messages, perhaps with a count of the number of read parameters, compared with how many there should be. Such errors would only arise only if parameters were intentionally removed, or if a older configuration file was used with a newer aligner.

Implement a cepstrum calculation function which will calculate the cepstrum from a given wave file. This relieves the aligner from using HTK to calculate cepstrum data.

In *calStats*, if there is only one sample phoneme in which standard deviation is trying to be calculated for, this will result in a square root error: **sqrt: DOMAIN error**, this is common since the standard deviation for a single value is not defined. A check could be put in place to prevent such cryptic and scary error messages.

HMMs

Introduction

The use of HMMs allows the use of forced Viterbi alignment which performs phoneme alignment based on trained HMMs. HTK v2.0 is used to train HMMs for phoneme alignments. HMMs have been touted as being very flexible, ability to perform very well since very robust HMMs can be trained, HTK also has the facility during alignment to select alternative spellings from a dictionary while also producing log probabilities during HMM training.

Use

This is a simple overview of the procedure involved in using HMMs to create forced Viterbi alignment. For a complete procedure list, see Appendix A.

1. Use HCopy to parameterise the speech waveforms into sequence of feature vectors. (MFCC)
2. Create flat start monophone HMMs.
3. Use HCompV to calculate global mean, variances, and variance floor.
4. Replicate dummy monophone HMM for each phoneme in phoneme set.
5. Embedded Re-estimation with HERest (executed three or four times).
6. Add short silence model[6] to HMMs.
7. Embedded Re-estimation with HERest (executed three or four times).
8. Create context dependent triphones.
9. Embedded Re-estimation with HERest (executed three or four times).
10. Cluster triphones (decision tree state tying)
11. Embedded Re-estimation with HERest (executed three or four times).
12. Forced Viterbi Alignment.

[6] Short silence model is trained on inter-word silences. Long (Normal) silence model is trained on beginning and ending utterance silences.

Drawbacks

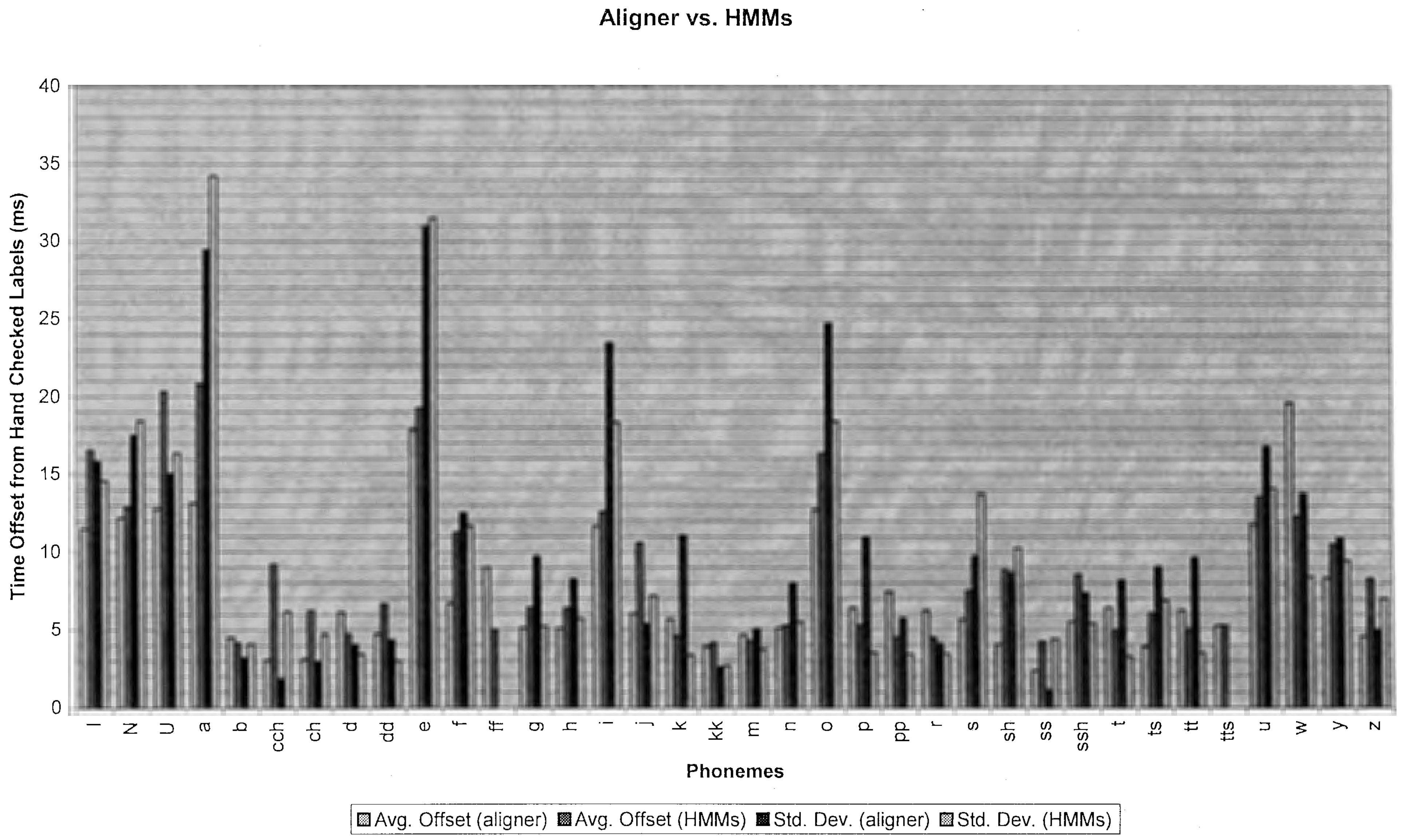
While the HTK manual provides a basic outline of the HMM training process, HMM training is overly flexible with many parameters and threshold values which require user adjustment that the default trained HMMs is susceptible to bad recordings, and produce mediocre results. HMMs also does not have the ability to detect incorrectly labelled phonemes[7]. Because the HMM training through HTK is a complicated process, debugging errors can be difficult and time consuming.

Evaluation

Following the above steps mentioned in Uses, after many iterations, the training of HMMs was finally made possible. The following table lists for each phoneme, the minimum time difference between the hand checked labels and the auto aligned using HMM labels, average time difference, maximum time difference, the frequency of the phoneme in the database, and the standard deviation of time difference.

[7] Not known at time of writing.

Phonem e	Minimum Offset (ms)	Average Offset (ms)	Maximum Offset (ms)	Phoneme Occurrences	Standard Deviation (ms)
#	0.346	23.217	1170.900	1706	41.192
I	0.000	16.537	77.170	319	14.569
N	0.000	12.844	237.000	878	18.436
U	0.000	20.355	78.910	278	16.395
X	1.880	34.844	95.570	28	29.296
a	0.000	20.821	334.000	3629	34.204
b	0.000	4.152	42.000	372	4.072
cch	0.000	9.202	21.000	12	6.184
ch	0.000	6.263	23.000	252	4.686
d	0.000	4.706	21.000	613	3.433
dd	2.220	6.690	9.900	4	2.980
e	0.000	19.276	340.000	1997	31.510
f	0.000	11.269	52.000	151	11.670
ff	5.000	5.000	5.000	1	NaN
g	0.000	6.443	48.030	738	5.216
h	0.000	6.402	45.000	475	5.698
i	0.000	12.582	309.300	2500	18.352
j	0.000	10.599	37.000	349	7.191
k	0.000	4.576	34.000	1434	3.347
kk	0.000	4.125	11.000	57	2.680
m	0.000	4.261	41.460	828	3.749
n	0.000	5.278	70.000	1305	5.531
o	0.000	16.360	240.000	3703	18.437
p	0.000	5.290	14.000	67	3.530
pp	0.410	4.514	18.000	52	3.426
r	0.000	4.510	21.050	1230	3.451
s	0.000	7.527	79.000	684	13.739
sh	0.000	8.851	83.000	561	10.265
ss	0.000	4.236	15.000	9	4.407
ssh	0.550	8.571	24.180	33	5.413
t	0.000	4.947	29.000	957	3.258
ts	0.000	6.093	48.000	251	6.895
tt	0.000	5.047	17.550	242	3.515
tts	5.280	5.280	5.280	1	NaN
u	0.000	13.584	159.530	2211	14.103
w	0.050	12.299	44.590	409	8.431
y	0.000	10.524	83.000	678	9.477
z	0.000	8.311	48.850	215	7.038

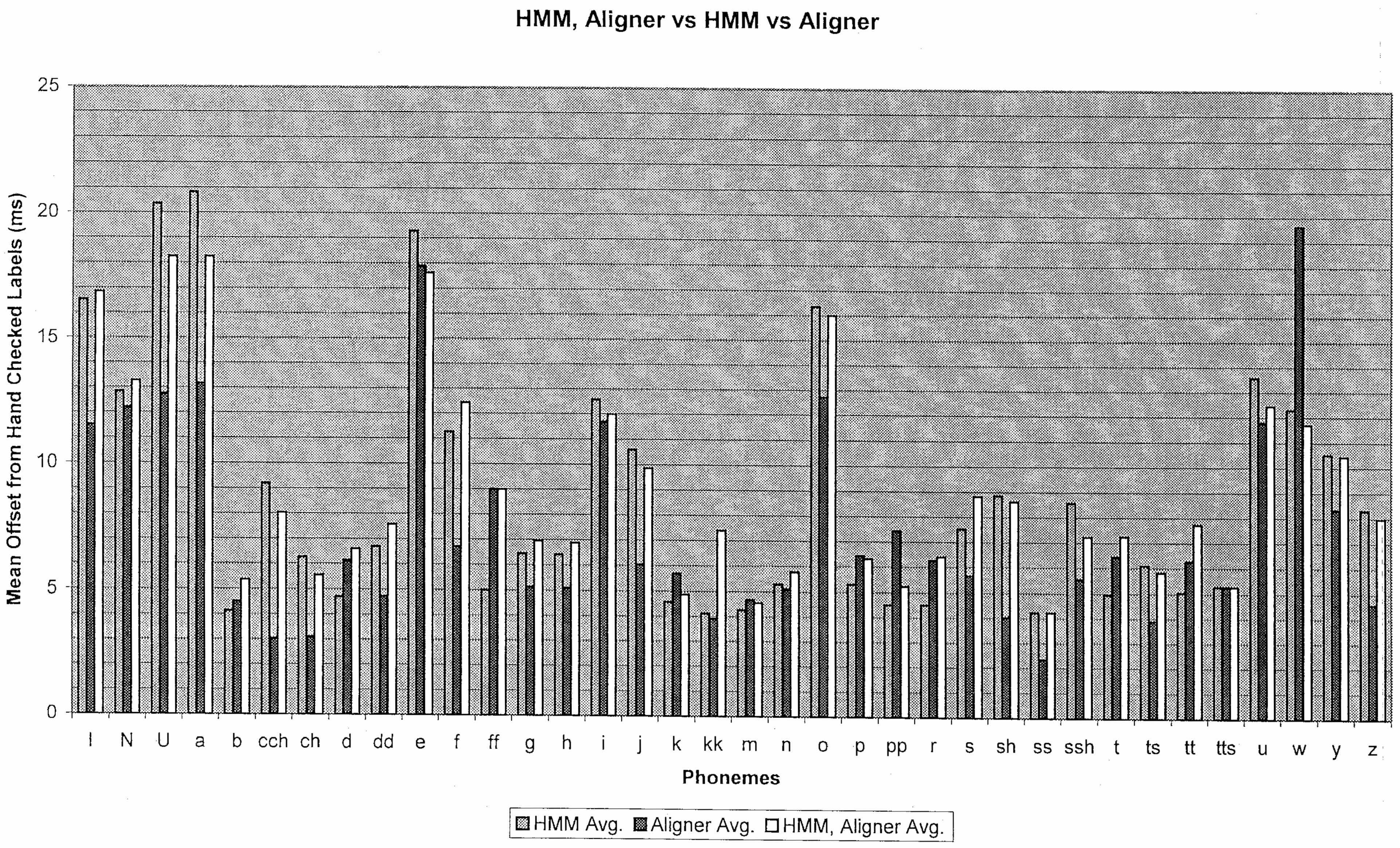

Aligner vs. HMMs
Time Offset from Hand Checked Labels (ms)
40
35
30
25
20
15
10
5
0
I
N
U
a
b
cch
ch
d
dd
e
f
ff
g
h
i
j
k
kk
m
n
o
p
pp
r
s
sh
ss
ssh
t
ts
tt
tts
u
w
y
z
Phonemes
▨ Avg. Offset (aligner)
▨ Avg. Offset (HMMs)
■ Std. Dev. (aligner)
▨ Std. Dev. (HMMs)

In comparing this data with earlier data produced from the aligner, we can see that for most phonemes, the aligner has a lower average time offset from the hand checked phoneme (with the exception of plosives, and few vowels). However, note that the standard deviation is much lower with HMM training than with the aligner.

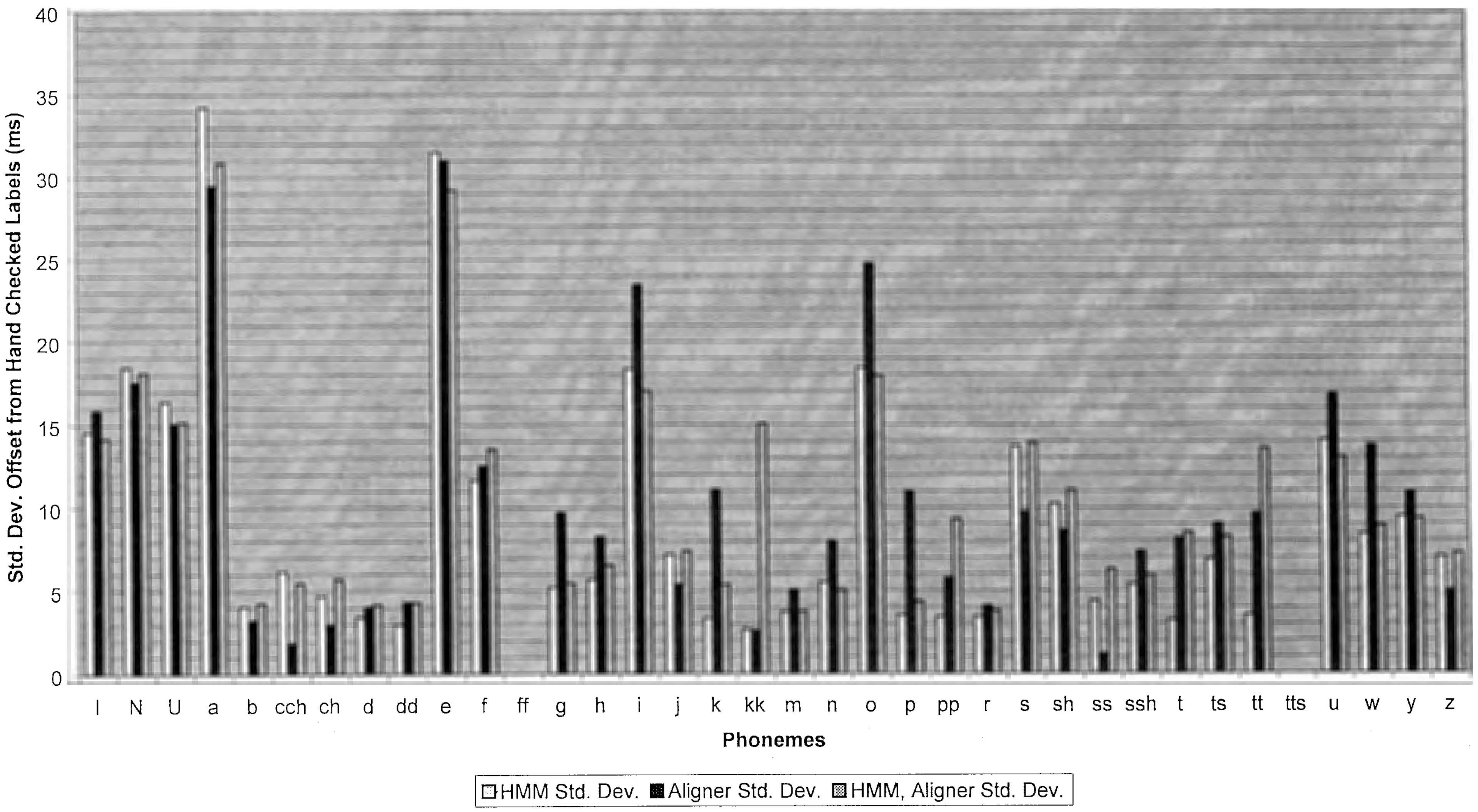
The following phonemes have been moved closer to hand checked labels with aligner than HMMs: I, N, U, a, b, cch, ch, e, f, j, s, sh, ss, z. While the following phonemes remained relatively unchanged: d, dd, kk. These phonemes had boundaries placed closer to the hand labelled boundaries with HMMS than aligner: g, h, i, k, n, o, p, pp, r, ssh, t, ts, tt, u, w, y.

The following table lists the results of the phoneme boundaries after first training with HMMs, then running aligner.

Phoneme	*Minimum Offset (ms)*	*Average Offset (ms)*	*Maximum Offset (ms)*	*Phoneme Occurrences*	*Standard Deviation (ms)*
#	0	8.531	1170.901	1491	32.369
I	0	16.841	63.17	319	14.196
N	0	13.282	237	877	18.109
U	0	18.258	78.91	274	15.149
X	1.88	34.712	95.57	28	29.421
a	0	18.256	334	3625	30.828
b	0	5.381	42	371	4.256
cch	2	8.036	21	12	5.466
ch	0	5.545	27	252	5.708
d	0	6.598	25	611	4.16
dd	1.78	7.58	13.809	4	4.275
e	0	17.627	340	1995	29.197
f	0	12.444	59	150	13.554
ff	9	9	9	1	0.002
g	0	6.947	48.03	736	5.46
h	0	6.88	45	475	6.545
i	0	11.993	309.3	2499	17.012
j	0	9.872	37	349	7.382
k	0	4.867	77	1434	5.383
kk	0	7.402	79	57	15.046
m	0	4.555	41.46	828	3.763
n	0	5.773	70	1304	5.043
o	0	16.001	240	3702	17.967
p	0	6.311	22	67	4.355
pp	0.216	5.238	66.609	52	9.33
r	0	6.391	26.05	1230	3.788
s	0	8.822	85.05	683	13.912
sh	0	8.617	83	561	11.066
ss	1	4.236	22	9	6.314
ssh	0	7.222	24.18	33	5.951
t	0	7.235	71	956	8.535
ts	0	5.838	48	251	8.259
tt	0	7.733	84.349	242	13.582
tts	5.28	5.28	5.28	1	NaN
u	0	12.458	159.53	2209	13.029
w	0	11.721	44.59	409	8.924
y	0	10.445	83	677	9.35
z	0	7.973	48.85	214	7.266


HMM, Aligner vs HMM vs Aligner
Mean Offset from Hand Checked Labels (ms)
25
20
15
10
5
0
I
N
U
a
b
cch
ch
d
dd
e
f
ff
g
h
i
j
k
kk
m
n
o
p
pp
r
s
sh
ss
ssh
t
ts
tt
tts
u
w
y
z
Phonemes
▨ HMM Avg.
▨ Aligner Avg.
□ HMM, Aligner Avg.

HMM, Aligner vs HMM vs Aligner



Running the aligner on HMM output produces boundaries close to the HMM original boundaries. HMMs have a tendency to create phonemes with large offsets as compared with the input versus the aligner. The aligner, since it uses information close to the phoneme labels (power, dcep) will tend not to move phonemes very far.

Future Work

Instead of trying to force a Viterbi alignment using tri-phone clusters, start first with a forced Viterbi alignment using monophone trained HMMs.

Conclusion

The aligner performs better than HMMs for half of the phonemes. This can be due to the general phoneme independent approach of the algorithm. Because the aligner uses speech information as a basis for phoneme checks, examining the phoneme being checked can improve performance.

HMMs are overly flexible, with many options, and the ability to train very robust HMMs. Performance can most likely be improved with additional tuning of pruning thresholds, triphone clustering thresholds, and other user settable parameters.

Both HTK and the aligner do not yet have abilities to recognize mislabelled phonemes due to the difficulty of the task.

Glossary

Cepstrum[vi] -	Mathematically defined as the Fourier transform of the log of the signal spectrum. Closely related to the frequency spectrum.
FFT -	Fast Fourier Transform. A version of the Fourier transform optimized for computation speed. Fourier transform converts a time domain signal to a frequency domain signal.
Formats[8] -	Transfer function of energy from the excitation source to the output can be described in terms of the natural frequencies or resonance of the tube. Such resonances are called *formants* for speech, and they represent the frequencies that pass the most acoustic energy from the source to the output.
GUI -	Graphical User Interface.
InterPhoneme Boundary -	The point at which one phoneme ends and another phoneme begins.
Labeling -	The process of dividing speech into smaller sections, each section corresponding to a unit (phoneme or otherwise).
Local Maxima -	A value within a locally defined region having the largest magnitude.
Magnitude -	Mathematically defined as the square of the amplitude.
Power spectral density -	Power of a signal at a specific time in the frequency domain.
Utterance -	A fragment of the original recorded speech which contains words or sentences.

[8] Rabiner Lawrence and Juang Biing-Hwang, Fundamentals of Speech Recognnition, Prentice-Hall International Inc., 1993

References

Lawrence Rabiner and Juang Biing-Hwang, Fundamentals of Speech Recognition, Prentice Hall International Editions, 1993

Olive Joseph P., Greenwood Alice, and Coleman John, Acoustics of American English Speech, Springer-Verlag, 1993

Press William H., Flannery Brian P., Teukolsky Saul A., Vetterling William T., Numerical Recipes in C, Cambridge University Press, 1988

Cambridge University Engineering Department Speech Group and Entropic Research Laboratories Inc., HTK Hidden Markov Model Toolkit Manual, 1993

Fry D. B., Acoustic Phonetics, Cambridge University Press, 1976

Ford R. D., Introduction to Acoustics, Elsevier Publishing, 1970

Knuth Donald E., The Art of Computing programming: Searching and Sorting, Addison Wesley, 1973

Odell Julian James, The Use of Context in Large Vocabulary Speech Recognition, Dissertation submitted to the University of Cambridge for the degree of Doctor of Philosophy, 1995

Further Readings

Donovan R. E., Trainable Speech Synthesis, Chapter 6, Modelling Improvements

Conference Proceedings of The Second ESCA/IEEE Workshop on Speech Synthesis, September 12-15, 1994.cfq

Acoustic Theory of Speech Production, orange and yellow binders. Ask Nick Campbell for details.

Appendix A – HMM Training Procedure

The following is a procedure list that was used for the purpose of training HMMs for forced Viterbi alignment. The training files are located under ~xwzhang/hp/fna:

```
export HBIN=/usr/local/HTK/HTK_V2.0/bin.hp700
export PATH=$PATH:$HBIN

use config1.cf as config.cf

1) HCopy -T 1 -C conf/config.cf -S scripts/tmp

use config2.cf as config.cf now

create master label file (nabi.mlf)

(Note that using the t.awk file, there were too many files to process all the
label files at once, thus "t.awk nabi_?_0*.lab" then "t.awk nabi_?_1*.lab"
...etc was used up to nabi_?_8*.lab. Then the individual mlf files were
concatenated together to form nabi.mlf)

(Note on the above paragraph, apparently, this only was a problem when on
the hp, running the script on a Sun worked without any problems.)

copy the phonetable file from /dept2/work25/pi/.../FNA/phonetable to lists/
directory as mono0.lst, which contains a list of phonemes used.
Remove extraneous info, leave only phoneme set. Don't forget to include "sil"

Create train.scp in scripts/ directory, which contains a list of MFC files
that were created up in step 1.

1.5) Initialize MMF models:

HCompV -C conf/config.cf -f 0.01 -m -S scripts/train.scp -M HMM/hmm0
HMM/hmm0/proto

This will create one HMM model with global speech means and variances. Also
creates a variance floor.

Copy the above HMM model into HMM/PreInit for each phoneme in the phoneset.
Adjust the transition probabilities for plosives (stops). Instead of the
standard 0 0 0.6 0.4 0 for hte third state, it should be 0 0 0.3 0.4 0.3.

Run script init.sh to initialize each HMM model.

combine all phoneme models in hmm0.5 to a single MMF and copy it into hmm1.
pre-pend the vFloors to the beginning of MMF to add variance floor. vFloors
was created with the HCompV command above.

2) HERest -C conf/config.cf -I labs/nabi.mlf -t 250.0 150.0 1000.0 -S
scripts/train.scp -T 1 -H HMM/hmm0.5/macros -H HMM/hmm0.5/MMF -M HMM/hmm1
lists/mono0.lst

HERest -C conf/config.cf -I labs/nabi.mlf -t 250.0 150.0 1000.0 -S
scripts/train.scp -T 1 -H HMM/hmm1/macros -H HMM/hmm1/MMF -M HMM/hmm2
lists/mono0.lst
```

```
HERest -C conf/config.cf -I labs/nabi.mlf -t 250.0 150.0 1000.0 -S
scripts/train.scp -T 1 -H HMM/hmm2/macros -H HMM/hmm2/MMF -M HMM/hmm3
lists/mono0.lst

HERest -C conf/config.cf -I labs/nabi.mlf -t 250.0 150.0 1000.0 -S
scripts/train.scp -T 1 -H HMM/hmm3/macros -H HMM/hmm3/MMF -M HMM/hmm4
lists/mono0.lst
```

copied MMF from hmm4/ directory to hmm5/ directory, then added short silence model. Took state 3 of the sil (long sil model), created short sp model, with same middle state. (SP model only has 3 states). Note transition Ps.

Need new master label files with short silence phonemes. labs/lab/t2.awk.

Note that once the MLF file was created, the path in the file had to be of the form of */nabi_A_001.lab ...etc, otherwise it complained it couldn't find the lab files.

use sil.hed as is, mono1.lst=mono0.lst+sp

```
3) HHEd -H HMM/hmm5/macros -H HMM/hmm5/MMF -M HMM/hmm6 scripts/sil.hed
lists/mono1.lst

4) HERest -C conf/config.cf -I labs/nabi_sp.mlf -t 250.0 150.0 1000.0 -S
scripts/train.scp -T 1 -m 0 -H HMM/hmm6/macros -H HMM/hmm6/MMF -M HMM/hmm7
lists/mono1.lst

HERest -C conf/config.cf -I labs/nabi_sp.mlf -t 250.0 150.0 1000.0 -S
scripts/train.scp -T 1 -m 0 -H HMM/hmm7/macros -H HMM/hmm7/MMF -M HMM/hmm8
lists/mono1.lst

HERest -C conf/config.cf -I labs/nabi_sp.mlf -t 250.0 150.0 1000.0 -S
scripts/train.scp -T 1 -m 0 -H HMM/hmm8/macros -H HMM/hmm8/MMF -M HMM/hmm9
lists/mono1.lst

HERest -C conf/config.cf -I labs/nabi_sp.mlf -t 250.0 150.0 1000.0 -S
scripts/train.scp -T 1 -m 0 -H HMM/hmm9/macros -H HMM/hmm9/MMF -M HMM/hmm10
lists/mono1.lst
```

use "make_triph_lab.led" (to make triphones)

```
5) HLEd -n lists/triphones0 -l '*' -i labs/nabitri.mlf
scripts/make_triph_lab.led labs/nabi_sp.mlf
```

use "make_triph_hmm.led"

```
6) HHEd -B -H HMM/hmm10/macros -H HMM/hmm10/MMF -M HMM/hmm11
scripts/make_triph_hmm.led lists/mono1.lst

7) HERest -B -C conf/config.cf -I labs/nabitri.mlf -t 250.0 150.0 1000.0 -m 0 -
S scripts/train.scp -T 1 -H HMM/hmm11/macros -H HMM/hmm11/MMF -M HMM/hmm12
lists/triphones0

HERest -B -C conf/config.cf -I labs/nabitri.mlf -t 250.0 150.0 1000.0 -m 0 -S
scripts/train.scp -T 1 -H HMM/hmm12/macros -H HMM/hmm12/MMF -M HMM/hmm13
lists/triphones0

HERest -B -C conf/config.cf -I labs/nabitri.mlf -t 250.0 150.0 1000.0 -m 0 -S
scripts/train.scp -T 1 -H HMM/hmm13/macros -H HMM/hmm13/MMF -M HMM/hmm14
lists/triphones0
```

Produce statistic files, we won't use the output HMM, only use the statistic file.

```
HERest -B -C conf/config.cf -I labs/nabitri.mlf -t 250.0 150.0 1000.0 -m 0 -S
scripts/train.scp -T 1 -s stat.hmm14 -H HMM/hmm14/macros -H HMM/hmm14/MMF -M
HMM/hmm15 lists/triphones0
```

We need to generate a question file used to cluster the triphones. For the nuuph phoneset (FNA, look in db_description of FNA root database dir to find out which phoneset it uses), look at chatr's nuuph_def.ch file in $CHATR_ROOT/ chatr/lib/data/

Once "ques" file has been modified (for the specific phoneme set), run:

ques mono1.lst which will out the question file used by HTK.

However some values need to be added, to the top of the output question file, let's use cluster_state_pros.scp as the name of that output,

```
RO 25.0 stat.hmm14
```

This will add the statistic file created with the last embedded reestimation. 25.0 means the minimum feature vectors that must be aligned to HMM states.

Change all occurences of THRESH to 50,

50 is the minimum increase in log likegood for using this question to split up a node.

25 and 50 are very lax thresholds (low), however, produce better alignment, gives higher degree of freedom to train embedded reestimation. Higher values produce more robust HMMs however less freedom to train embedded reestimation.

Also at the end of the cluster_state_pros.scp file, add:

```
CO "lists/tiedlist_state"
```

CO -> These triphones' HMM are completely mapped to other HMMs (instead of only state mappings). This will also include unseen triphones as well, as they need to be completely mapped to other HMMs (of course). The file pointed to by the CO command will be created by the HHED command below.

```
8) HHEd -B -T 12002 -H HMM/hmm14/macros -H HMM/hmm14/MMF -M HMM/hmm15
scripts/cluster_state_pros.scp lists/triphones0 > clustering.log
```

(Triphone clustering)

Mean Occupation Count= amount of feature vectors that are aligned to HMM states.

```
9) HERest -C conf/config.cf -I labs/nabitri.mlf -t 250.0 150.0 1000.0 -m 0 -S
scripts/train.scp -T 1 -H HMM/hmm15/macros -H HMM/hmm15/MMF -M HMM/hmm16
lists/tiedlist_state

HERest -C conf/config.cf -I labs/nabitri.mlf -t 250.0 150.0 1000.0 -m 0 -S
scripts/train.scp -T 1 -H HMM/hmm16/macros -H HMM/hmm16/MMF -M HMM/hmm17
lists/tiedlist_state

HERest -C conf/config.cf -I labs/nabitri.mlf -t 250.0 150.0 1000.0 -m 0 -S
scripts/train.scp -T 1 -H HMM/hmm17/macros -H HMM/hmm17/MMF -M HMM/hmm18
lists/tiedlist_state
```

```
HERest -C conf/config.cf -I labs/nabitri.mlf -t 250.0 150.0 1000.0 -m 0 -S
scripts/train.scp -T 1 -H HMM/hmm18/macros -H HMM/hmm18/MMF -M HMM/hmm19
lists/tiedlist_state
```

At this point, we need a word transcription file, and a dictionary. See page 194 under "Generating Forced Alignments".

However, we can also create a dummy word transcription file, with each file, and it's phonemes (as words), see labs/nabi_dum.mlf for details, compared with hp/training/labs/word2.mlf.

Also create a dummy dictionary, with each phoneme, as itself. Instead of the phoneme sequence for the word.

To create the dummy word transcription file (nabi_dum.mlf) we basically copy nabi.mlf, but we need to remove all the "sil", so use the nabi_wrd.awk script file.

created dummy phoneme dictionary dic/phondic.dic

```
HVite -o NC -b sil -C conf/config.cf -a -H HMM/hmm19/macros -H HMM/hmm19/MMF -f
-l results/al.hmm19 -m -t 250.0 -I labs/nabi_dum.mlf -S scripts/train.scp -T 1
dic/phondic.dic lists/tiedlist_state
```

to create ESPS format label files, use the parameter -P ESPS in the above line.

Endnotes

[i] Rabiner Lawrence and Juang Biing-Hwang, Fundamentals of Speech Recognnition, Prentice-Hall International Inc., 1993

[ii] Flanagan James L., Some Properties of the glotttal sound source, Journal of Speech and Hearing Research, vol. 1 (1958), pp 99-111.

[iii] Olive Joseph P., Greenwood Alice and Coleman John (AT&T Bell Laboratories), Acoustics of American English Speech- A Dynamic Approach, Springer-Verlag, 1993.

[iv] Fry D. B., Acoustic Phonetics, Cambridge University Press, 1976.

[v] Olive Joseph P., Greenwood Alice and Coleman John, Acoustics of American English Speech, Springer-Verlag, 1993.

[vi] Rabiner Lawrence and Juang Biing-Hwang, Fundamentals of Speech Recognition, Prentice Hall International Editions, 1993.