INDEX

(This is under construction.)

1. Before you train
    - The general-procedure chart
    - Modeling context-dependent phones with untied states: some memory requirements
    - Data preparation
        - When you have a very small closed vocabulary
    - The set of base and higher order feature vectors
        - Feature streams
    - Force-alignment
2. Training continuous models
    - Creating the CI model definition file
    - Creating the HMM topology file
    - Flat initialization of CI model parameters
    - Training CI models
    - Creating the CD untied model definition file
    - Flat initialization of CD untied model parameters
    - Training CD untied models
    - Building decision trees for parameter sharing
        - Generating the linguistic questions
    - Pruning the decision trees
    - Creating the CD tied model definition file
    - Initializing and training cd tied gaussian mixture models
3. Training semi-continuous models
    - Vector quantization
    - Creating the CI model definition file
    - Creating the HMM topology file
    - Flat initialization of CI model parameters
    - Training CI models
    - Creating the CD untied model definition file
    - Flat initialization of CD untied model parameters
    - Training CD untied models
    - Building decision trees for parameter sharing
        - Generating the linguistic questions
    - Pruning the decision trees
    - Creating the CD tied model definition file
    - Initializing and training cd tied models
    - Deleted interpolation
4. SPHINX2 data and model formats
5. SPHINX3 data and model formats
6. Training multilingual models
7. The training lexicon
8. Converting SPHINX3 format models to SPHINX2 format
9. Updating or adapting existing model sets
10. Using the SPHINX-III decoder with semi-continuous and continuous models

---

This part of the manual describes the procedure(s) for training acoustic models using the Sphinx3 trainer. General training procedures are described first, and followed by more detailed descriptions of the programs and scripts used, and the analysis of their logs and other outputs.

BEFORE YOU TRAIN

THE GENERAL-PROCEDURE CHART

```
                         Training chart for the
                         sphinx2  trainer
                        =========================
                                OBSOLETE
               (The sphinx2 trainer is no longer used in CMU)



                         Training chart for the
                         sphinx3  trainer
                        =========================
                             type of model
                                   |
                    ----------------------------------
                    |                                |
               CONTINUOUS                      SEMI-CONTINUOUS
                    |                                |
                    |                          vector-qunatization
                    |                                |
                    ----------------------------------
                                   |...make ci mdef
                                   |...flat_initialize CI models
                             training CI models
                                   |...make cd untied mdef
                                   |...initialize
                                   |
                             training CD untied models
                                   |
                                   |
                                   |
                             decision tree building
                                   |...prune trees
                                   |...tie states
                                   |...make cd tied mdef
                             training CD tied models
                                   |
                                   |
recursive            ----------------------------------
gaussian splitting.. |                                |
                 continuous models              semi-continuous models
                     |                                |
                     |                                |
                -----------                           |
                |         |                    deleted interpolation
          decode with   ADAPT                         |
          sphinx3         |                           |---ADAPT
          decoder <-------                            |     |
                                                 ----------------
                          make cd tied mdef ...  | ............|
                          with decode dict and   |           convert to
                          pruned trees           |           sphinx2
```

```
                                  decode with           |
                                  sphinx3               |
                                  decoder               |
                                                        |
                                                   decode with
                                                   sphinx2
                                                   decoder
                                         (currently opensource
                                          and restricted to
                                          working with sampling
                                          rates 8khz and 16khz.
                                          Once the s3 trainer is
                                          released, this will have
                                          to change to allow
                                          people who train with
                                          different sampling rates
                                          to use this decoder)
```

back to index

BEFORE YOU TRAIN

MODELING CONTEXT-DEPENDENT PHONES WITH UNTIED STATES: SOME MEMORY REQUIREMENTS

Modeling Context-dependent phones (ex. triphones) with untied states requires the largest amount of hardware resources. Take a moment to check if you have enough. The resources required depend on the type of model you are going to train, the dimensionality and configuration of your feature vectors, and the number of states in the HMMs.

Semi-continuous models

To train 5-state/HMM models for 10,000 triphones:

```
5 states/triphone                     = 50,000 states
For a 4-stream feature-set, each      = 1024 floating point numbers/state
state has a total of 4*256 mixture
weights
                                      = 205Mb buffer for 50,000 states
```

Corresponding to each of the four feature streams, there are 256 means and 256 variances in the codebook. ALL these, and ALL the mixture weights and transition matrices are loaded in into the RAM, and during training an additional buffer of equal size is allocated to store intermediate results. These are later written out into the hard disk when the calculations for the current training iteration are complete. Note that there are as many transition matrices as you have phones (40-50 for the English language, depending on your dictionary) All this amounts to allocating well over 400 Mb of RAM.

This is a bottleneck for machines with smaller memory. No matter how large your training corpus is, you can actually train only about 10,000 triphones at the cd-untied stage if you have ~400 Mb of RAM (A 100 hour broadcast news corpus typically has 40,000 triphones). You could train more if your machine is capable of handling the memory demands effectively (this could be done, for example, by having a large amount of swap space). If you are training on multiple machines, *each* will require this much memory. In addition, at the end of each iteration, you have to transmit all buffers to a single machine that performs the norm. Networking issues need to be considered here.

The cd-untied models are used to build trees. The number of triphones you train at this stage directly affects the quality of the trees, which would have to be built using fewer triphones than are actually present in the training set if you do not have enough memory.

Continuous models

For 10,000 triphones:

```
5 states/triphone          = 50,000 states
39 means (assuming a
39-component feature
vector) and 39
variances per state        = 79 floating points per state
                           = 15.8Mb buffer for 50,000 states
```

Thus we can train 12 times as many triphones as we can when we have semicontinuous models for the same amount of memory. Since we can use more triphones to train (and hence more information) the decision trees are better, and eventually result in better recognition performance.

back to index

BEFORE YOU TRAIN

DATA PREPARATION

You will need the following files to begin training:

1. A set of **feature files** computed from the audio training data, one each for every recording you have in the training corpus. Each recording can be transformed into a sequence of feature vectors using a front-end executable provided with the SPHIN-III training package. Each front-end executable provided performs a different analysis of the speech signals and computes a different type of feature.

2. A **control file** containing the list of feature-set filenames with full paths to them. An example of the entries in this file:

   ```
   dir/subdir1/utt1
   dir/subdir1/utt2
   dir/subdir2/utt3
   ```

   Note that the extensions are not given. They will be provided separately to the trainer. It is a good idea to give unique names to all feature files, even if including the full paths seems to make each entry in the control file unique. You will find later that this provides a lot of flexibility for doing many things.

3. A **transcript file** in which the transcripts corresponding to the feature files are listed in exactly the

same order as the feature filenames in the control file.

4. A **main dictionary** which has all acoustic events and words in the transcripts mapped onto the acoustic units you want to train. Redundancy in the form of extra words is permitted. The dictionary must have all alternate pronunciations marked with paranthesized serial numbers starting from (2) for the second pronunciation. The marker (1) is omitted. Here's an example:

```

DIRECTING           D AY R EH K T I ng
DIRECTING(2)        D ER EH K T I ng
DIRECTING(3)        D I R EH K T I ng
```

5. A **filler dictionary**, which usually lists the non-speech events as "words" and maps them to user_defined phones. This dictionary must at least have the entries

```
<s>     SIL
<sil>   SIL
</s>    SIL
```

The entries stand for

```
<s>     : begining-utterance silence
<sil>   : within-utterance silence
</s>    : end-utterance silence
```

Note that the words <s>, </s> and <sil> are treated as special words and are required to be present in the filler dictionary. At least one of these must be mapped on to a phone called "SIL". The phone SIL is treated in a special manner and is required to be present. The sphinx expects you to name the acoustic events corresponding to your general background condition as SIL. For clean speech these events may actually be silences, but for noisy speech these may be the most general kind of background noise that prevails in the database. Other noises can then be modelled by phones defined by the user.

During training SIL replaces every phone flanked by "+" as the context for adjacent phones. The phones flanked by "+" are only modeled as CI phones and are not used as contexts for triphones. If you do not want this to happen you may map your fillers to phones that are not flanked by "+".

6. A **phonelist**, which is a list of all acoustic units that you want to train models for. The SPHINX does not permit you to have units other than those in your dictionaries. All units in your two dictionaries must be listed here. In other words, your phonelist must have exactly the same units used in your dictionaries, no more and no less. Each phone must be listed on a separate line in the file, begining from the left, with no extra spaces after the phone. an example:

```
AA
AE
OW
B
CH
```

Here's a quick checklist to verify your data preparation before you train:

1. Are all the transcript words in the dictionary/filler dictionary?
2. Make sure that the size of transcript matches the .ctl file.
3. Check the boundaries defined in the .ctl file to make sure they exist ie, you have all the frames that are listed in the control file
4. Verify the phonelist against the dictionary and fillerdict

When you have a very small closed vocabulary (50-60 words)

If you have only about 50-60 words in your vocabulary, and if your entire test data vocabulary is covered by the training data, then you are probably better off training word models rather than phone models. To do this, simply define the phoneset as your set of words themselves and have a dictionary that maps each word to itself and train. Also, use a lesser number of fillers, and if you do need to train phone models make sure that each of your tied states has enough counts (at least 5 or 10 instances of each).

BEFORE YOU TRAIN

THE SET OF BASE AND HIGHER ORDER FEATURE VECTORS

The set of feature vectors you have computed using the Sphinx front-end executable is called the set of **base** feature vectors. This set of base features can be extended to include what are called **higher order** features. Some common extensions are

a. The set of difference vectors, where the component-wise difference between *some* succeeding and preceding vector(s), used to get an estimate of the slope or trend at the current time instant, are the "extension" of the current vector. These are called "delta" features. A more appropriate name would be the "trend" features.
b. The set of difference vectors of difference vectors. The component-wise difference between the succeeding and preceding "delta" vectors are the "extension" of the current vector. These are called "double delta" features
c. The set of difference vectors, where the component-wise difference between the n^th succeeding and n^th preceding vector are the "extension" of the current vector. These are called "long-term delta" features, differing from the "delta" features in just that they capture trends over a longer window of time.
d. The vector composed of the first elements of the current vector and the first elements of some of the above "extension" vectors. This is called the "power" feature, and its dimensionality is less than or equal to the total number of feature types you consider.

Feature streams

In semi-continuous models, it is a usual practice to keep the identities of the base vectors and their "extension" vectors separate. Each such set is called a "feature stream". You must specify how many feature streams you want to use in your semi-continuous models and how you want them arranged. The feature-set options currently supported by the Sphinx are:

c/1..L-1/,d/1..L-1/,c/0/d/0/dd/0/,dd/1..L-1/ : read this as cepstra/second to last component, deltacepstra/second to last component,
cepstra/first component deltacepstra/first component doubledeltacepstra/first component, doubledeltacepstra/second to last component

This is a 4-stream feature vector used mostly in semi-continuous models. There is no particular advantage to this arrangement - any permutation would give you the same models, with parameters written in different orders.

Here's something that's not obvious from the notation used for the 4-stream feature set: the dimensionality of the 4-stream feature vector is 12cepstra+24deltas+3powerterms+12doubledeltas

the deltas are computed as the difference between the cepstra two frames removed on either side of the current frame (12 of these), followed by the difference between the cepstra four frames removed on either side of the current frame (12 of these). The power stream uses the first component of the two-frames-removed deltas, computed using C_0.

(more to come....)

TRAINING CONTINUOUS MODELS

CREATING THE CI MODEL DEFINITION FILE

The first step is to prepare a **model definition** file for the context independent (CI) phones. The function of a model definition file is to define or provide a unique numerical identity to every state of every HMM that you are going to train, and to provide an order which will be followed in writing out the model parameters in the model parameter files. During the training, the states are referenced only by these numbers. The model definition file thus partly specifies your **model architecture** and is thus usually stored in a directory named "model_architecture". You are of course free to store it where you please, unless you are running the training scripts provided with the SPHINX-III package.

To generate this **CI model definition file**, use the executable **mk_model_def** with the following flag settings:

FLAG	DESCRIPTION
-phonelstfn	phonelist
-moddeffn	name of the CI model definition file that you want to create. Full path must be provided
-n_state_pm	number of states per HMM in the models that you want to train. If you want to train 3 state HMMs, write "3" here, without the double quotes

Pipe the standard output into a log file **ci_mdef.log** (say). If you have listed only three phones in your phonelist, and specify that you want to build three state HMMs for each of these phones, then your model-definition file will look like this:

```
# Generated by /mk_model_def on Thu Aug 10 14:57:15 2000
0.3
3 n_base
0 n_tri
12 n_state_map
9 n_tied_state
9 n_tied_ci_state
3 n_tied_tmat
#
# Columns definitions
```

```
#base lft  rt p attrib   tmat  ...state id's ...
SIL    -   -  - filler    0    0       1      2     N
A      -   -  -    n/a    1    3       4      5     N
B      -   -  -    n/a    2    6       7      8     N

The # lines are simply comments. The rest of the variables mean the following:

  n_base      : no. of phones (also called "base" phones) that you have
  n_tri       : no. of triphones (we will explain this later)
  n_state_map : Total no. of HMM states (emitting and non-emitting)
                The Sphinx appends an extra terminal non-emitting state
                to every HMM, hence for 3 phones, each specified by
                the user to be modeled by a 3-state HMM, this number
                will be 3phones*4states = 12
  n_tied_state: no. of states of all phones after state-sharing is done.
                We do not share states at this stage. Hence this number is the
                same as the total number of emitting states, 3*3=9
n_tied_ci_state:no. of states for your "base" phones after state-sharing
                is done. At this stage, the number of "base" phones is
                the same as the number of "all" phones  that you are modeling.
                This number is thus again the total number of emitting
                states, 3*3=9
 n_tied_tmat   :The HMM for each CI phone has a transition probability matrix
                 associated it. This is the total number of transition
                 matrices for the given set of models. In this case, this
                 number is 3.

Columns definitions: The following columns are defined:
       base  : name of each phone
       lft   : left-context of the phone (- if none)
       rt    : right-context of the phone (- if none)
       p     : position of a triphone (not required at this stage)
       attrib: attribute of phone. In the phone list, if the phone is "SIL",
               or if the phone is enclosed by "+", as in "+BANG+", the sphinx
               understands these phones to be non-speech events. These are
               also called "filler" phones, and the attribute "filler" is
               assigned to each such phone. The base phones have no special
               attributes, and hence are labelled as "n/a", standing for
               "no attribute"
      tmat   : the id of the transition matrix associated with the phone
 state id's  : the ids of the HMM states associated with any phone. This list
               is terminated by an "N" which stands for a non-emitting
               state. No id is assigned to it. However, it exists, and is
               listed.
```

TRAINING CONTINUOUS MODELS

CREATING THE HMM TOPOLOGY FILE

The HMM topology file consists of a matrix with boolean entries, each entry indiactes whether a specific transition from state=row_number to state=column_number is permitted in the HMMs or not. For example a 3-state HMM with no skips permitted beteen states would have a topology file with the following entries:

```
4
1.0     1.0     0.0     0.0
0.0     1.0     1.0     0.0
```

```
0.0     0.0     1.0     1.0
```

The number 4 is total the number of sates in an HMMs. The SPHINX automatically appends a fourth non-emitting terminating state to the 3 state HMM. The first entry of 1.0 means that a transition from state 1 to state 1 (itself) is permitted. Accordingly, the transition matrix estimated for any phone would have a "transition-probability" in place of this boolean entry. Where the entry is 0.0, the corresponding transition probability will not be estimated (will be 0).

You can either write out the topology file manually, or use the script script make_topology.pl provided with the SPHINX package to do this. The script needs the following arguments:

```
        states_per_hmm : this is merely an integer specifying the
                         number of states per hmm
        skipstate      : "yes" or "no" depending on whether you
                         want the HMMs to have skipped state transitions
                         or not.
```

Note that the topology file is common for all HMMs and is a single file containing the topology definition matrix. This file also defines your model architecture and is usually placed in the model_architecture directory. This is however optional, but recommended. If you are running scripts from the SPHINX training package, you will find the file created in the model_architecture directory.

TRAINING CONTINUOUS MODELS

FLAT INITIALIZATION OF CI MODEL PARAMETERS

CI models consist of 4 parameter files :

- **mixture_weights**: the weights given to every Gaussian in the Gaussian mixture corresponding to a state
- **transition_matrices**: the matrix of state transition probabilities
- **means**: means of all Gaussians
- **variances**: variances of all Gaussians

To begin training the CI models, each of these files must have some initial entries, ie, they must be "initialized". The mixture_weights and transition_matrices are initialized using the executable **mk_flat**. It needs the following arguments:

FLAG	DESCRIPTION
-moddeffn	CI model definition file
-topo	HMM topology file
-mixwfn	file in which you want to write the initialized mixture weights
-tmatfn	file in which you want to write the initialized transition matrices
-nstream	number of independent feature streams, for continuous models this number should be set to "1", without the double quotes
-ndensity	number of Gaussians modeling each state. For CI models, this number should be set to "1"

To initialize the means and variances, global values of these parameters are first estimated and then copied into appropriate positions in the parameter files. The global mean is computed using all the vectors you have in your feature files. This is usually a very large number, so the job is divided into many parts. At this stage you tell the Sphinx how many parts you want it to divide this operation into (depending on the computing facilities you have) and the Sphinx "accumulates" or gathers up the vectors for each part separately and writes it into an intermediate buffer on your machine. The executable **init_gau** is used for this purpose. It needs the following arguments:

FLAG	DESCRIPTION
-accumdir	directory in which you want to write the intermediate buffers
-ctlfn	control file
-part	part number
-npart	total number of parts
-cepdir	path to feature files - this will be appended before all paths given in the control file
-cepext	filename extension of feature files, eg. "mfc" for files called a/b/c.mfc. Double quotes are not needed
-feat	type of feature
-ceplen	dimensionality of base feature vectors
-agc	automatic gain control factor(max/none)
-cmn	cepstral mean normalization(yes/no)
-varnorm	variance normalization(yes/no)

Once the buffers are written, the contents of the buffers are "normalized" or used to compute a global mean value for the feature vectors. This is done using the executable **norm** with the following flag settings:

FLAG	DESCRIPTION
-accumdir	buffer directory
-meanfn	file in which you want to write the global mean
-feat	type of feature
-ceplen	dimensionality of base feature vector

The next step is to "accumulate" the vectors for computing a global variance value. The executable **init_gau**, when called a second time around, takes the value of the global mean and collects a set of (vector-globalmean)^2 values for the entire data set. This time around, this executable needs the following arguments:

FLAG	DESCRIPTION
-accumdir	directory in which you want to write the intermediate buffers
-meanfn	globalmean file
-ctlfn	control file

-part	part number
-npart	total number of parts
-cepdir	path to feature files - this will be appended before all paths given in the control file
-cepext	filename extension of feature files, eg. "mfc" for files called a/b/c.mfc. Double quotes are not needed
-feat	type of feature
-ceplen	dimensionality of base feature vectors
-agc	automatic gain control factor(max/none)
-cmn	cepstral mean normalization(yes/no)
-varnorm	variance normalization(yes/no)

Again, once the buffers are written, the contents of the buffers are "normalized" or used to compute a global variance value for the feature vectors. This is again done using the executable **norm** with the following flag settings:

FLAG	DESCRIPTION
-accumdir	buffer directory
-varfn	file in which you want to write the global variance
-feat	type of feature
-ceplen	dimensionality of base feature vector

Once the global mean and global variance are computed, they have to be copied into the means and variances of every state of each of the HMMs. The global mean is written into appropriate state positions in a **means** file while the global variance is written into appropriate state positions in a **variances** file. If you are using the scripts provided with the SPHINX package, you will find these files with "flatinitial" as part of its name in the model_parameters directory.

The flat **means** and **variances** file can be created using the executable **cp_parm**. In order to be able to use this executable you will have to create a **copyoperations map** file which is a two-column file, with the left column id-ing the state *to* which the global value has to be copied, and the right column id-ing the state *from* which it has to be copied. If there are "nphones" CI phones and each state has "nEstate_per_hmm" EMITTING states, there will be ntotal_Estates = nphones * nEstate_per_hmm lines in the copyoperations map file; the state id-s (on the left column) run from 0 thru (ntotal_Estates - 1). Here is an example for a 3-state hmm (nEstate_per_hmm = 3) for two phones (nphones = 2) (ntotal_Estates = 6; so, state ids would vary from 0-5):

```
0   0
1   0
2   0
3   0
4   0
5   0
```

cp_parm requires the following arguments.

FLAG	DESCRIPTION

-cpopsfn	copyoperations map file
-igaufn	input global mean (or variance) file
-ncbout	number of phones times the number of states per HMM (ie, total number of states)
-ogaufn	output initialized means (or variances) file

cp_parm has to be run twice, once for copying the means, and once for copying the variances. This completes the initialization process for CI training.

TRAINING CONTINUOUS MODELS

TRAINING CONTEXT INDEPENDENT MODELS

Once the flat initialization is done, you are ready to begin training the acoustic models for the base or "context-independent" or CI phones. This step is called CI-training. In CI-training, the flat-initialized models are re-estimated through the forward-backward re-estimation algorithm called the Baum-Welch algorithm. This is an iterative re-estimation process, so you have to run many "passes" of the Baum-Welch re-estimation over your training data. Each of these passes, or iterations, results in a slightly better set of models for the CI phones. However, since the objective function maximized in each of theses passes is the likelihood, too many iterations would ultimately result in models which fit very closely to the training data. you might not want this to happen for many reasons. Typically, 5-8 iterations of Baum-Welch are sufficient for getting good estimates of the CI models. You can automatically determine the number of iterations that you need by looking at the total likelihood of the training data at the end of the first iteration and deciding on a "convergence ratio" of likelihoods. This is simply the ratio of the total likelihood in the current iteration to that of the previous iteration. As the models get more and more fitted to the training data in each iteration, the training data likelihoods typically increase monotonically. The convergence ratio is therefore a small positive number. The convergence ratio becomes smaller and smaller as the iterations progress, since each time the current models are a little less different from the previous ones. Convergence ratios are data and task specific, but typical values at which you may stop the Baum-Welch iterations for your CI training may range from 0.1-0.001. When the models are variance-normalized, the convergence ratios are much smaller.

The executable used to run a Buam-Welch iteration is called "bw", and takes the following example arguments for training continuous CI models:

FLAG	DESCRIPTION
-moddeffn	model definition file for CI phones
-ts2cbfn	this flag should be set to ".cont." if you are training continuous models, and to ".semi." if you are training semi-continuous models, without the double quotes
-mixwfn	name of the file in which the mixture-weights from the previous iteration are stored. Full path must be provided
-mwfloor	Floor value for the mixture weights. Any number below the floor value is set to the floor value.
-tmatfn	name of the file in which the transition matrices from the previous iteration are stored. Full path must be provided
-meanfn	name of the file in which the means from the previous iteration are stored. Full path must be provided

-varfn	name of the file in which the variances fromt he previous iteration are stored. Full path must be provided
-dictfn	Dictionary
-fdictfn	Filler dictionary
-ctlfn	control file
-part	You can split the training into N equal parts by setting a flag. If there are M utterances in your control file, then this will enable you to run the training separately on each (M/N)th part. This flag may be set to specify which of these parts you want to currently train on. As an example, if your total number of parts is 3, this flag can take one of the values 1,2 or 3
-npart	number of parts in which you have split the training
-cepdir	directory where your feature files are stored
-cepext	the extension that comes after the name listed in the control file. For example, you may have a file called a/b/c.d and may have listed a/b/c in your control file. Then this flag must be given the argument "d", without the double quotes or the dot before it
-lsnfn	name of the transcript file
-accumdir	Intermediate results from each part of your training will be written in this directory. If you have T means to estimate, then the size of the mean buffer from the current part of your training will be T*4 bytes (say). There will likewise be a variance buffer, a buffer for mixture weights, and a buffer for transition matrices
-varfloor	minimum variance value allowed
-topn	no. of gaussians to consider for computing the likelihood of each state. For example, if you have 8 gaussians/state models and topn is 4, then the 4 most likely gaussian are used.
-abeam	forward beamwidth
-bbeam	backward beamwidth
-agc	automatic gain control
-cmn	cepstral mean normalization
-varnorm	variance normalization
-meanreest	mean re-estimation
-varreest	variance re-estimation
-2passvar	Setting this flag to "yes" lets bw use the previous means in the estimation of the variance. The current variance is then estimated as $E[(x - prev\_mean)^2]$. If this flag is set to "no" the current estimate of the means are used to estimate variances. This requires the estimation of variance as $E[x^2] - (E[x])^2$, an unstable estimator that sometimes results in negative estimates of the variance due to arithmetic imprecision
-tmatreest	re-estimate transition matrices or not
-feat	feature configuration
-ceplen	length of basic feature vector

If you have run the training in many parts, or even if you have run the training in one part, the executable for Baum-Welch described above generates only intermediate buffer(s). The final model parameters, namely the means, variances, mixture-weights and transition matrices, have to be estimated using the values stored in these buffers. This is done by the executable called "norm", which takes the following arguments:

FLAG	DESCRIPTION
-accumdir	Intermediate buffer directory
-feat	feature configuration
-mixwfn	name of the file in which you want to write the mixture weights. Full path must be provided
-tmatfn	name of the file in which you want to write the transition matrices. Full path must be provided
-meanfn	name of the file in which you want to write the means. Full path must be provided
-varfn	name of the file in which you want to write the variances. Full path must be provided
-ceplen	length of basic feature vector

If you have not re-estimated any of the model parameters in the bw step, then the corresponding flag must be omitted from the argument given to the norm executable. The executable will otherwise try to read a non-existent buffer from the buffer directory and will not go through. Thus if you have set -meanreest to be "no" in the argument for bw, then the flag -meanfn must not be given in the argument for norm. This is useful mostly during adaptation.

Iterations of baum-welch and norm finally result CI models. The iterations can be stopped once the likelihood on the training data converges. The model parameters computed by norm in the final iteration are now used to initialize the models for context-dependent phones (triphones) with untied states. This is the next major step of the training process. We refer to the process of training triphones HMMs with untied states as the "CD untied training".

TRAINING CONTINUOUS MODELS

CREATING THE CD UNTIED MODEL DEFINITION FILE

The next step is the CD-untied training, in which HMMs are trained for all context-dependent phones (usually triphones) that are seen in the training corpus. For the CD-untied training, we first need to to generate a model definition file for all the triphones occuring in the training set. This is done in several steps:

First, a list of all triphones possible in the vocabulary is generated from the dictionary. To get this complete list of triphones from the dictionary, it is first necessary to write the list of phones in the following format:

```
phone1 0 0 0 0
phone2 0 0 0 0
phone3 0 0 0 0
phone4 0 0 0 0
...
```

The phonelist used for the CI training must be used to generate this, and the order in which the phones are listed must be the same.

Next, a temporary dictionary is generated, which has all words except the filler words (words enclosed in ++()++). The entry

```
SIL    SIL
```

must be added to this temporary dictionary, and the dictionary must be sorted in alphabetical order. The program "quick_count" provided with the SPHINX-III package can now be used to generate the list of all possible triphones from the temporary dictionary. It takes the following arguments:

FLAG	DESCRIPTION
-q	mandatory flag to tell quick_count to consider all word pairs while constructing triphone list
-p	formatted phonelist
-b	temporary dictionary
-o	output triphone list

Here is a typical output from quick_count

```
AA(AA,AA)s             1
AA(AA,AE)b             1
AA(AA,AO)1             1
AA(AA,AW)e             1
```

The "1" in AA(AA,AO)1 indicates that this is a word-internal triphone. This is a carry over from Sphinx-II. The output from quick_count has to be now written into the following format:

```
AA AA AA s
AA AA AE b
AA AA AO i
AA AA AW e
```

This can be done by simply replacing "(", ",", and ")" in the output of quick_count by a space and printing only the first four columns. While doing so, all instances of " 1" must be replaced by " i". To the top of the resulting file the list of CI phones must be appened in the following format

```
AA - - -
AE - - -
AO - - -
AW - - -
..
..
AA AA AA s
AA AA AE b
AA AA AO i
AA AA AW e
```

For example, if the output of the quick_count is stored in a file named "quick_count.out", the following perl command will generate the phone list in the desired form. perl -nae '$F[0] =~ s/\(|\)|\,/ /g; $F[0] =~ s/1/i/g; print $F[0]; if ($F[0] =~ /\s+$/){print "i"}; print "\n"' quick_count.out

The above list of triphones (and phones) is converted to the model definition file that lists all possible triphones from the dictionary. The program used from this is "mk_model_def" with the following arguments number of states per HMM

FLAG	DESCRIPTION
-moddeffn	model definition file with all possible triphones(alltriphones_mdef)to be written
-phonelstfn	list of all triphones
-n_state_pm	

In the next step we find the number of times each of the triphones listed in the alltriphones_mdef occured in the training corpus To do this we call the program "param_cnt" which takes the following arguments:

FLAG	DESCRIPTION
-moddeffn	model definition file with all possible triphones(alltriphones_mdef)
-ts2cbfn	takes the value ".cont." if you are building continuous models
-ctlfn	control file corresponding to your training transcripts
-lsnfn	transcript file for training
-dictfn	training dictionary
-fdictfn	filler dictionary
-paramtype	write "phone" here, without the double quotes
-segdir	/dev/null

param_cnt writes out the counts for each of the triphones onto stdout. All other messages are sent to stderr. The stdout therefore has to be directed into a file. If you are using csh or tcsh it would be done in the following manner:

```
(param_cnt [arguments] > triphone_count_file) >&! LOG
```

Here's an example of the output of this program

```
+GARBAGE+ - - - 98
+LAUGH+ - - - 29
SIL - - - 31694
AA - - - 0
AE - - - 0
...
AA AA AA s 1
AA AA AE s 0
AA AA AO s 4
```

The final number in each row shows the number of times that particular triphone (or filler phone) has occured in the training corpus. Not that if all possible triphones of a CI phone are listed in the all_triphones.mdef the CI phone itself will have 0 counts since all instances of it would have been mapped to a triphone.

This list of counted triphones is used to shortlist the triphones that have occured a minimum number (threshold) of times. The shortlisted triphones appear in the same format as the file from which they have been selected. The shortlisted triphone list has the same format as the triphone list used to generate the all_triphones.mdef. The formatted list of CI phones has to be included in this as before. So, in the earlier example, if a threshold of 4 were used, we would obtain the shortlisted triphone list as

```
AA - - -
AE - - -
AO - - -
AW - - -
..
..
AA AA AO s
..
```

The threshold is adjusted such that the total number of triphones above the threshold is less that the maximum number of triphones that the system can train (or that you wish to train). It is good to train as many triphones as possible. The maximum number of triphones may however be dependent on the memory available on your machine. The logistics related to this are described in the beginning of this manual.

Note that thresholding is usually done so to reduce the number of triphones, in order that the resulting models will be small enough to fit in the computer's memory. If this is not a problem, then the threshold can be set to a smaller number. If the triphone occurs too few times, however, (ie, if the threshold is too small), there will not be enough data to train the HMM state distributions properly. This would lead to poorly estimated CD untied models, which in turn may affect the decision trees which are to be built using these models in the next major step of the training.

A model definition file is now created to include only these shortlisted triphones. This is the final model definition file to be used for the CD untied training. The reduced triphone list is then to the model definition file using mk_model_def with the following arguments: number of states per HMM

FLAG	DESCRIPTION
-moddeffn	model definition file for CD untied training
-phonelstfn	list of shortlisted triphones
-n_state_pm	

Finally, therefore, a model definition file which lists all CI phones and seen triphones is constructed. This file, like the CI model-definition file, assigns unique id's to each HMM state and serves as a reference file for handling and identifying the CD-untied model parameters. Here is an example of the CD-untied model-definition file: If you have listed five phones in your phones.list file,

SIL B AE T

and specify that you want to build three state HMMs for each of these phones, and if you have one utterance listed in your transcript file:

<s> BAT A TAB </s> for which your dictionary and fillerdict entries are:

```
Fillerdict:
<s>   SIL
</s>  SIL
```

```
Dictionary:
A       AX
BAT     B AE T
TAB     T AE B
```

then your CD-untied model-definition file will look like this:

```
# Generated by /mk_model_def on Thu Aug 10 14:57:15 2000
0.3
5 n_base
7 n_tri
48 n_state_map
36 n_tied_state
15 n_tied_ci_state
5 n_tied_tmat
#
# Columns definitions
#base lft  rt p attrib   tmat  ...state id's ...
SIL     -    -  - filler    0    0       1      2     N
AE      -    -  -    n/a    1    3       4      5     N
AX      -    -  -    n/a    2    6       7      8     N
B       -    -  -    n/a    3    9       10     11    N
T       -    -  -    n/a    4    12      13     14    N
AE      B    T  i    n/a    1    15      16     17    N
AE      T    B  i    n/a    1    18      19     20    N
AX      T    T  s    n/a    2    21      22     23    N
B       SIL AE b     n/a    3    24      25     26    N
B       AE  SIL e    n/a    3    27      28     29    N
T       AE  AX e     n/a    4    30      31     32    N
T       AX  AE b     n/a    4    33      34     35    N

The # lines are simply comments. The rest of the variables mean the following:

  n_base      : no. of CI phones (also called "base" phones), 5 here
  n_tri       : no. of triphones , 7 in this case
  n_state_map : Total no. of HMM states (emitting and non-emitting)
                The Sphinx appends an extra terminal non-emitting state
                to every HMM, hence for 5+7 phones, each specified by
                the user to be modeled by a 3-state HMM, this number
                will be 12phones*4states = 48
  n_tied_state: no. of states of all phones after state-sharing is done.
                We do not share states at this stage. Hence this number is the
                same as the total number of emitting states, 12*3=36
n_tied_ci_state:no. of states for your CI phones after state-sharing
                is done. The CI states are not shared, now or later.
                This number is thus again the total number of emitting CI
                states, 5*3=15
 n_tied_tmat   : The total number of transition matrices is always the same
                 as the total number of CI phones being modeled. All triphones
                 for a given phone share the same transition matrix. This
                 number is thus 5.

Columns definitions: The following columns are defined:
       base  : name of each phone
       lft   : left-context of the phone (- if none)
       rt    : right-context of the phone (- if none)
       p     : position of a triphone. Four position markers are supported:
               b = word begining triphone
```

```
                e = word ending triphone
                i = word internal triphone
                s = single word triphone
        attrib: attribute of phone. In the phone list, if the phone is "SIL",
                or if the phone is enclosed by "+", as in "+BANG+", these
               phones are interpreted as non-speech events. These are
                also called "filler" phones, and the attribute "filler" is
                assigned to each such phone. The base phones and the
                triphones have no special attributes, and hence are
                labelled as "n/a", standing for "no attribute"
       tmat   : the id of the transition matrix associated with the phone
  state id's  : the ids of the HMM states associated with any phone. This list
                is terminated by an "N" which stands for a non-emitting
                state. No id is assigned to it. However, it exists, and is
                listed.
```

TRAINING CONTINUOUS MODELS

FLAT INITIALIZATION OF CD UNTIED MODEL PARAMETERS

In the next step in CD untied training, after the CD untied model definition file has been constructed, the model parameters are first intialized. During this process, the model parameter files corresponding to the CD untied model-definition file are generated. Four files are generated: means, variances, transition matrices and mixture weights. In each of these files, the values are first copied from the corresponding CI model parameter file. Each state of a particular CI phone contributes to the same state of the same CI phone in the Cd -untied model parameter file, and also to the same state of the *all* the triphones of the same CI phone in the CD-untied model parameter file. The CD-untied model definition file is of course used as a reference for this mapping. This process, as usual, is called "initialization".

Initialization for the CD-untied training is done by the executable called "init_mixw". It need the following arguments:

-src_moddeffn	source (CI) model definition file
-src_ts2cbfn	.cont.
-src_mixwfn	source (CI) mixture-weight file
-src_meanfn	source (CI) means file
-src_varfn	source (CI) variances file
-src_tmatfn	source (CI) transition-matrices file
-dest_moddeffn	destination (CD untied) model definition file
-dest_ts2cbfn	.cont.
-dest_mixwfn	destination (CD untied) mixtrue weights file
-dest_meanfn	destination (Cd untied) means file
-dest_varfn	destination (CD untied) variances file
-dest_tmatfn	destination (Cd untied) transition matrices file
-feat	feature configuration
-ceplen	dimensionality of base feature vector

TRAINING CONTINUOUS MODELS

TRAINING CD UNTIED MODELS

Once the initialization for CD-untied training is done, the next step is to actually train the CD untied models. To do this, as in the CI training, the Baum-Welch forward-backward algorithm is iteratively used. Each iteration consists of generating bw buffers by running the bw executable on the training corpus (this can be divided into many parts as explained in the CI training), follwed by running the norm executable to compute the final parameters at the end of the iteration. The arguments required by the bw executable at this stage are as follows.

FLAG	DESCRIPTION
-moddeffn	CD-untied model definition file
-ts2cbfn	this flag should be set to ".cont." if you are training continuous models, and to ".semi." if you are training semi-continuous models, without the double quotes
-mixwfn	name of the file in which the mixture-weights from the previous iteration are stored. Full path must be provided
-mwfloor	Floor value for the mixture weights. Any number below the floor value is set to the floor value.
-tmatfn	name of the file in which the transition matrices from the previous iteration are stored. Full path must be provided
-meanfn	name of the file in which the means from the previous iteration are stored. Full path must be provided
-varfn	name of the file in which the variances fromt he previous iteration are stored. Full path must be provided
-dictfn	Dictionary
-fdictfn	Filler dictionary
-ctlfn	control file
-part	You can split the training into N equal parts by setting a flag. If there are M utterances in your control file, then this will enable you to run the training separately on each $(M/N)^{th}$ part. This flag may be set to specify which of these parts you want to currently train on. As an example, if your total number of parts is 3, this flag can take one of the values 1,2 or 3
-npart	number of parts in which you have split the training
-cepdir	directory where your feature files are stored
-cepext	the extension that comes after the name listed in the control file. For example, you may have a file called a/b/c.d and may have listed a/b/c in your control file. Then this flag must be given the argument "d", without the double quotes or the dot before it
-lsnfn	name of the transcript file
-accumdir	Intermediate results from each part of your training will be written in this directory. If you have T means to estimate, then the size of the mean buffer from the current part of your training will be T*4 bytes (say). There will likewise be a variance buffer, a buffer for mixture weights, and a buffer for transition matrices

-varfloor	minimum variance value allowed
-topn	no. of gaussians to consider for likelihood computation
-abeam	forward beamwidth
-bbeam	backward beamwidth
-agc	automatic gain control
-cmn	cepstral mean normalization
-varnorm	variance normalization
-meanreest	mean re-estimation
-varreest	variance re-estimation
-2passvar	Setting this flag to "yes" lets bw use the previous means in the estimation of the variance. The current variance is then estimated as E[(x - prev_mean)^2]. If this flag is set to "no" the current estimate of the means are used to estimate variances. This requires the estimation of variance as E[x^2] - (E[x])^2, an unstable estimator that sometimes results in negative estimates of the variance due to arithmetic imprecision
-tmatreest	re-estimate transition matrices or not
-feat	feature configuration
-ceplen	length of basic feature vector

- The Baum-Welch step should be followed by the nomalization step. The executable "norm" must be used for this. The arguments required by the norm executable are the same as that for CI training, and are listed below:

FLAG	DESCRIPTION
-accumdir	Intermediate buffer directory
-feat	feature configuration
-mixwfn	name of the file in which you want to write the mixture weights. Full path must be provided
-tmatfn	name of the file in which you want to write the transition matrices. Full path must be provided
-meanfn	name of the file in which you want to write the means. Full path must be provided
-varfn	name of the file in which you want to write the variances. Full path must be provided
-ceplen	length of basic feature vector

The iterations of Baum-Welch and norm must be run till the likelihoods converge (ie, the convergence ratio reaches a small threshold value). Typically this happens in 6-10 iterations.

TRAINING CONTINUOUS MODELS

BUILDING DECISION TREES FOR PARAMETER SHARING

Once the CD-untied models are computed, the next major step in training continuous models is decision tree building. Decision trees are used to decide which of the HMM states of all the triphones (seen and unseen) are similar to each other, so that data from all these states are collected together and used to train one global state, which is called a "senone". Many groups of similar states are formed, and the number of "senones" that are finally to be trained can be user defined. A senone is also called a tied-state and is obviously shared across the triphones which contributed to it. The technical details of decision tree building and state tying are explained in the technical section of this manual. It is sufficient to understand here that for state tying, we require to build decision trees.

Generating the linguistic questions

The decision trees require the CD-untied models and a set of predefined phonetic classes (or classes of the acoustic units you are modeling) which share some common property. These classes or questions are used to partition the data at any given node of a tree. Each question results in one partion, and the question that results in the "best" partition (maximum increase in likelihood due to the partition) is chosen to partition the data at that node. All linguistic questions are written in a single file called the "linguistic questions" file. One decision tree is built for each state of each phone.

For example, if you want to build a decision tree for the contexts (D B P AE M IY AX OY) for any phone, then you could ask the question: does the context belong to the class vowels? If you have defined the class vowels to have the phones AE AX IY OY EY AA EH (in other words, if one of your linguistic questions has the name "VOWELS" and has the elements AE AX IY OY EY AA EH corresponding to that name), then the decision tree would branch as follows:

```
                     D B P AE M IY AX OY
                              |

       question: does this context belong to the class VOWELS ?
                              /\
                             /  \
                            /    \
                         yes      no
                         /         \
                        /           \
                     AE IY AX OY     D B P M
                        |             |
                      question     question
                       /\            /\
                      /  \          /  \
```

Here is an example of a "linguistic-questions" file:

```
ASPSEG      HH
SIL         SIL
VOWELS      AE AX IY OY EY AA EH
ALVSTP      D T N
DENTAL      DH TH
LABSTP      B P M
LIQUID      L R
```

The column on the left specifies the name gives to the class. This name is user defined. The classes consist of a single phone or a cluster of phones whaich share some common acoustic property. If your acoustic units are not completely phonetically motivated, or if you are training models for a language whose phonetic structure you are not completely sure about, then the executable classed "make_quests" provided with the SPHINX-III package can be used to generate the linguistic questions. It uses the CI models to make the questions, and needs the following arguments:

FLAG	DESCRIPTION
-moddeffn	CI model definition file
-meanfn	CI means file
-varfn	CI variances file
-mixwfn	CI mixture weights file
-npermute	A bottom-up top-down clustering algorithm is used to group the phones into classes. Phones are clustered using bottom-up clustering until npermute classes are obtained. The npermute classes are exhaustively partitioned into two classes and evaluated to identify the optimal partitioning of the entire phone set into two groups. An identical procedure is performed recursively on each of these groups to generate an entire tree. npermute is typically between 8 and 12. Smaller values of npermute result in suboptimal clustering. Larger values become computationally prohibitive.
-niter	The bottom-up top-down clustering can be iterated to give more optimal clusters. niter sets the number of iterations to run. niter is typiclly set to 1 or 2. The clustering saturates after 2 iterations.
-qstperstt	The algoritm clusters state distributions belonging to each state of the CI phone HMMs to generate questions. Thus all 1st states are clustered to generate one subset of questions, all 2nd states are clustered for the second subset, and so on. qstperstt determines how many questions are to be generated by clustering any state. Typically this is set to a number between 20 and 25.
-tempfn	
-questfn	output lingustic questions file

Once the linguistic questions have been generated, decision trees must be built for each state of each CI phone present in your phonelist. Decision trees are however not built for filler phones written as +()+ in your phonelist. They are also not built for the SIL phone. In order to build decision trees, the executable "bldtree" must be used. It takes the following arguments:

FLAG	DESCRIPTION
-treefn	full path to the directory in which you want the decision trees to be written
-moddeffn	CD-untied model definition file
-mixwfn	Cd-untied mixture weights file
-ts2cbfn	.cont.
-meanfn	CD-untied means file
-varfn	CD-untied variances file
-mwfloor	Floor value of the mixture weights. Values below this are reset to this value. A typical value is 1e-8
-psetfn	linguistic questions file
-phone	CI phone for which you want to build the decision tree

-state	The HMM state for which you want to build the decision tree. For a three state HMM, this value can be 0,1 or 2. For a 5 state HMM, this value can be 0,1,2,3 or 4, and so on
-stwt	This flag needs a string of numbers equal to the number of HMM-states, for example, if you were using 5-state HMMs, then the flag could be given as "-stwt 1.0 0.3 0.1 0.01 0.001". Each of these numbers specify the weights to be given to state distributions during tree building, beginning with the *current* state. The second number specifies the weight to be given to the states *immediately adjacent* to the current state (if there are any), the third number specifies the weight to be given to adjacent states *one removed* from the immediately adjacent one (if there are any), and so on. A typical set of values for 5 state HMMs is "1.0 0.3 0.1 0.01 0.001"
-ssplitmin	Complex questions are built for the decision tree by first building "pre-trees" using the linguistic questions in the question file. The minimum number of bifurcations in this tree is given by ssplitmin. This should not be lesser than 1. This value is typically set to 1.
-ssplitmax	The maximum number of bifurcations in the simple tree before it is used to build complex questions. This number is typically set to 7. Larger values would be more computationally intensive. This number should not be smaller than the value given for ssplitmin
-ssplitthr	Minimum increase in likelihood to be considered for a bifurcation in the simple tree. Typically set to a very small number greater than or equal to 0
-csplitmin	The minimum number of bifurcations in the decision tree. This should not be less than 1
-csplitmax	The maximum number of bifurcations in the decision tree. This should be as large as computationlly feasible. This is typically set to 2000
-csplitthr	Minimum increase in likelihood to be considered for a bifurcation in the decision tree. Typically set to a very small number greater than or equal to 0.
-cntthresh	Minimum number of observations in a state for it to be considered in the decision tree building process.

If, for example, you have a phonelist which contains the following phones

```
+NOISE+
SIL
AA
AX
B
```

and you are training 3 state HMMs, then you must build 9 decision trees, one each for each state of the phones AA, AX and B.

TRAINING CONTINUOUS MODELS

PRUNING THE DECISION TREES

Once the decision trees are built, they must be pruned to have as many leaves as the number of tied states (senones) that you want to train. Remember that the number of tied states does not include the CI states, which are never tied. In the pruning process, the bifurcations in the decision trees which resulted in the minimum increase in likelihood are progressively removed and replaced by the parent node. The selection of the branches to be pruned out is done across the entire collection of decision trees globally.

The executable to be used for decision tree pruning is called "prunetree" and requires the following arguments:

FLAG	DESCRIPTION
-itreedir	directory in which the full decision trees are stored
-nseno	number of senones that you want to train
-otreedir	directory to store the pruned decision trees
-moddeffn	CD-untied model definition file
-psetfn	lingistic questions file
-minocc	minimum number of observations in the given tied state. If there are fewer observations, the branches corresponding to the tied state get pruned out by default. This value should never be 0, otherwise you will end up having senones with no data to train (which are seen 0 times in the training set)

TRAINING CONTINUOUS MODELS

CREATING THE CD TIED MODEL DEFINITION FILE

Once the trees are pruned, a new model definition file must be created which

- contains all the triphones which are seen during training
- has the states corresponding to these triphones identified with senones from the pruned trees

In order to do this, the model definition file which contains all possible triphones from the current training dictionary can be used [alltriphones model definition file]. This was built during the process of building the CD-untied model definition file. Remember that the CD-untied model definition file contained only a selected number of triphones, with various thresholds used for selection. That file, therefore, cannot be used to build the CD-tied model definition file, except in the exceptional case where you are sure that the CD-untied model definition file includes *all* triphones seen during training. The executable that must be used to tie states is called "tiesate" and needs the following arguments:

FLAG	DESCRIPTION
-imoddeffn	alltriphones model definition file
-omoddeffn	CD-tied model definition file
-treedir	pruned tree directory
-psetfn	linguistic questions file

Here is an example of a CD-tied model definition file, based on the earlier example given for the CD-untied model definition file. The alltriphones model definition file:

```
# Generated by [path]/mk_model_def on Sun Nov 26 12:42:05 2000
# triphone: (null)
# seno map: (null)
#
0.3
```

```
5 n_base
34 n_tri
156 n_state_map
117 n_tied_state
15 n_tied_ci_state
5 n_tied_tmat
#
# Columns definitions
#base lft  rt p attrib tmat      ... state id's ...
  SIL   -    - - filler    0    0    1    2    N
   AE   -    - -    n/a    1    3    4    5    N
   AX   -    - -    n/a    2    6    7    8    N
    B   -    - -    n/a    3    9   10   11    N
    T   -    - -    n/a    4   12   13   14    N
   AE   B    T i    n/a    1   15   16   17    N
   AE   T    B i    n/a    1   18   19   20    N
   AX  AX   AX s    n/a    2   21   22   23    N
   AX  AX    B s    n/a    2   24   25   26    N
   AX  AX  SIL s    n/a    2   27   28   29    N
   AX  AX    T s    n/a    2   30   31   32    N
   AX   B   AX s    n/a    2   33   34   35    N
   AX   B    B s    n/a    2   36   37   38    N
   AX   B  SIL s    n/a    2   39   40   41    N
   AX   B    T s    n/a    2   42   43   44    N
   AX SIL   AX s    n/a    2   45   46   47    N
   AX SIL    B s    n/a    2   48   49   50    N
   AX SIL  SIL s    n/a    2   51   52   53    N
   AX SIL    T s    n/a    2   54   55   56    N
   AX   T   AX s    n/a    2   57   58   59    N
   AX   T    B s    n/a    2   60   61   62    N
   AX   T  SIL s    n/a    2   63   64   65    N
   AX   T    T s    n/a    2   66   67   68    N
    B  AE   AX e    n/a    3   69   70   71    N
    B  AE    B e    n/a    3   72   73   74    N
    B  AE  SIL e    n/a    3   75   76   77    N
    B  AE    T e    n/a    3   78   79   80    N
    B  AX   AE b    n/a    3   81   82   83    N
    B   B   AE b    n/a    3   84   85   86    N
    B SIL   AE b    n/a    3   87   88   89    N
    B   T   AE b    n/a    3   90   91   92    N
    T  AE   AX e    n/a    4   93   94   95    N
    T  AE    B e    n/a    4   96   97   98    N
    T  AE  SIL e    n/a    4   99  100  101    N
    T  AE    T e    n/a    4  102  103  104    N
    T  AX   AE b    n/a    4  105  106  107    N
    T   B   AE b    n/a    4  108  109  110    N
    T SIL   AE b    n/a    4  111  112  113    N
    T   T   AE b    n/a    4  114  115  116    N
```

is used as the base to give the following CD-tied model definition file with 39 tied states (senones):

```
# Generated by [path]/mk_model_def on Sun Nov 26 12:42:05 2000
# triphone: (null)
# seno map: (null)
#
0.3
5 n_base
34 n_tri
156 n_state_map
```

```
54 n_tied_state
15 n_tied_ci_state
5 n_tied_tmat
#
# Columns definitions
#base lft  rt p attrib tmat      ... state id's ...
  SIL   -   - - filler   0    0    1    2    N
   AE   -   - -    n/a   1    3    4    5    N
   AX   -   - -    n/a   2    6    7    8    N
    B   -   - -    n/a   3    9   10   11    N
    T   -   - -    n/a   4   12   13   14    N
   AE   B   T i    n/a   1   15   16   17    N
   AE   T   B i    n/a   1   18   16   19    N
   AX  AX  AX s    n/a   2   20   21   22    N
   AX  AX   B s    n/a   2   23   21   22    N
   AX  AX SIL s    n/a   2   24   21   22    N
   AX  AX   T s    n/a   2   25   21   22    N
   AX   B  AX s    n/a   2   26   21   27    N
   AX   B   B s    n/a   2   23   21   27    N
   AX   B SIL s    n/a   2   24   21   27    N
   AX   B   T s    n/a   2   25   21   27    N
   AX SIL  AX s    n/a   2   26   21   28    N
   AX SIL   B s    n/a   2   23   21   28    N
   AX SIL SIL s    n/a   2   24   21   28    N
   AX SIL   T s    n/a   2   25   21   28    N
   AX   T  AX s    n/a   2   26   21   29    N
   AX   T   B s    n/a   2   23   21   29    N
   AX   T SIL s    n/a   2   24   21   29    N
   AX   T   T s    n/a   2   25   21   29    N
    B  AE  AX e    n/a   3   30   31   32    N
    B  AE   B e    n/a   3   33   31   32    N
    B  AE SIL e    n/a   3   34   31   32    N
    B  AE   T e    n/a   3   35   31   32    N
    B  AX  AE b    n/a   3   36   37   38    N
    B   B  AE b    n/a   3   36   37   39    N
    B SIL  AE b    n/a   3   36   37   40    N
    B   T  AE b    n/a   3   36   37   41    N
    T  AE  AX e    n/a   4   42   43   44    N
    T  AE   B e    n/a   4   45   43   44    N
    T  AE SIL e    n/a   4   46   43   44    N
    T  AE   T e    n/a   4   47   43   44    N
    T  AX  AE b    n/a   4   48   49   50    N
    T   B  AE b    n/a   4   48   49   51    N
    T SIL  AE b    n/a   4   48   49   52    N
    T   T  AE b    n/a   4   48   49   53    N
```

TRAINING CONTINUOUS MODELS

INITIALIZING AND TRAINING CD TIED GAUSSIAN MIXTURE MODELS

The next step is to train the CD-tied models. In the case of continuous models, the HMM states can be modeled by either a single Gaussian distribution, or a mixture of Gaussian distributions. The number of Gaussians in a mixture-distribution must preferably be even, and a power of two (for example, 2,4,8,16, 32,..). To model the HMM states by a mixture of 8 Gaussians (say), we first have to train 1 Gaussian per state models. Each Gaussian distribution is then split into two by perturbing its mean slightly, and the resulting two distributions are used to intialize the training for 2 Gaussian per state models. These are further perturbed to initialize for 4 Gaussains per state models and a further split is done to initalize for the 8 Gaussians per state models. So the CD-tied training for models with 2^N Gaussians per state is done in N+1 steps. Each of these N+1 steps consists of

1. initialization
2. iterations of Baum-Welch followed by norm
3. Gaussian splitting (not done in the N+1th stage of CD-tied training)

The training begins with the initialization of the 1 Gaussian per state models. During initialization, the model parameters from the CI model parameter files are copied into appropriate positions in the CD tied model parameter files. Four model parameter files are created, one each for the means, variances, transition matrices and mixture weights. During initialization, each state of a particular CI phone contributes to the same state of the same CI phone in the CD-tied model parameter file, and also to the same state of the *all* the triphones of the same CI phone in the CD-tied model parameter file. The CD-tied model definition file is used as a reference for this mapping.

Initialization for the 1 gaussian per state models is done by the executable called **init_mixw**. It requires the following arguments:

-src_moddeffn	source (CI) model definition file
-src_ts2cbfn	.cont.
-src_mixwfn	source (CI) mixture-weight file
-src_meanfn	source (CI) means file
-src_varfn	source (CI) variances file
-src_tmatfn	source (CI) transition-matrices file
-dest_moddeffn	destination (CD tied) model def inition file
-dest_ts2cbfn	.cont.
-dest_mixwfn	destination (CD tied 1 Gau/state) mixture weights file
-dest_meanfn	destination (CD tied 1 Gau/state) means file
-dest_varfn	destination (CD tied 1 Gau/state) variances file
-dest_tmatfn	destination (CD tied 1 Gau/state) transition matrices file
-feat	feature configuration
-ceplen	dimensionality of base feature vector

The executables used for baum-welch, norm and Gaussaian splitting are **bw**, **norm** and **inc_comp**

The arguments needed by **bw** are

FLAG	DESCRIPTION
-moddeffn	CD tied model definition file
-ts2cbfn	this flag should be set to ".cont." if you are training continuous models, and to ".semi." if you are training semi-continuous models, without the double quotes
-mixwfn	name of the file in which the mixture-weights from the previous iteration are stored. Full path must be provided

-mwfloor	Floor value for the mixture weights. Any number below the floor value is set to the floor value.
-tmatfn	name of the file in which the transition matrices from the previous iteration are stored. Full path must be provided
-tpfloor	Floor value for the transition probabilities. Any number below the floor value is set to the floor value.
-meanfn	name of the file in which the means from the previous iteration are stored. Full path must be provided
-varfn	name of the file in which the variances fromt he previous iteration are stored. Full path must be provided
-dictfn	Dictionary
-fdictfn	Filler dictionary
-ctlfn	control file
-part	You can split the training into N equal parts by setting a flag. If there are M utterances in your control file, then this will enable you to run the training separately on each (M/N)th part. This flag may be set to specify which of these parts you want to currently train on. As an example, if your total number of parts is 3, this flag can take one of the values 1,2 or 3
-npart	number of parts in which you have split the training
-cepdir	directory where your feature files are stored
-cepext	the extension that comes after the name listed in the control file. For example, you may have a file called a/b/c.d and may have listed a/b/c in your control file. Then this flag must be given the argument "d", without the double quotes or the dot before it
-lsnfn	name of the transcript file
-accumdir	Intermediate results from each part of your training will be written in this directory. If you have T means to estimate, then the size of the mean buffer from the current part of your training will be T*4 bytes (say). There will likewise be a variance buffer, a buffer for mixture weights, and a buffer for transition matrices
-varfloor	minimum variance value allowed
-topn	no. of gaussians to consider for likelihood computation
-abeam	forward beamwidth
-bbeam	backward beamwidth
-agc	automatic gain control
-cmn	cepstral mean normalization
-varnorm	variance normalization
-meanreest	mean re-estimation
-varreest	variance re-estimation
-2passvar	Setting this flag to "yes" lets bw use the previous means in the estimation of the variance. The current variance is then estimated as $E[(x - prev\_mean)^2]$. If this flag is set to "no" the current estimate of the means are used to estimate variances. This requires the estimation of variance as $E[x^2] - (E[x])^2$, an unstable estimator that sometimes results in negative estimates of the variance due to arithmetic imprecision

-tmatreest	re-estimate transition matrices or not
-feat	feature configuration
-ceplen	length of basic feature vector

The arguments needed by **norm** are:

FLAG	DESCRIPTION
-accumdir	Intermediate buffer directory
-feat	feature configuration
-mixwfn	name of the file in which you want to write the mixture weights. Full path must be provided
-tmatfn	name of the file in which you want to write the transition matrices. Full path must be provided
-meanfn	name of the file in which you want to write the means. Full path must be provided
-varfn	name of the file in which you want to write the variances. Full path must be provided
-ceplen	length of basic feature vector

The arguments needed by **inc_comp** are:

FLAG	DESCRIPTION
-ninc	how many gaussians (per state) to split currently. You need not always split to double the number of Gaussians. you can specify other numbers here, so long as they are less than the number of Gaussians you currently have.This is a positive integer like "2", given without the double quotes
-ceplen	length of the base feature vector
-dcountfn	input mixture weights file
-inmixwfn	input mixture weights file
-outmixwfn	output mixture weights file
-inmeanfn	input means file
-outmeanfn	ouput means file
-invarfn	input variances file
-outvarfn	output variances file
-feat	type of feature

Back to index

TRAINING SEMI-CONTINUOUS MODELS

VECTOR QUANTIZATION

This is done in two steps. In the first step, the feature vectors are accumulated for quantizing the vector space. Not all feature vectors are used. Rather, a sampling of the vectors available is done by the executable "agg_seg". This executable simply "aggregates" the vectors into a buffer. The following flag settings must be used with this executable:

FLAG	DESCRIPTION
-segdmpdirs	directory in which you want to put the aggregate buffer
-segdmpfn	name of the buffer (file)
-segtype	all
-ctlfn	control file
-cepdir	path to feature files
-cepext	feature vector filename extension
-ceplen	dimensionality of the base feature vector
-agc	automatic gain control factor(max/none)
-cmn	cepstral mean normalization(yes/no)
-feat	type of feature. As mentioned earlier, the 4-stream feature vector is usually given as an option here. When you specify the 4-stream feature, this program will compute and aggregate vectors corresponding to all streams separately.
-stride	how many samples to ignore during sampling of vectors (pick every stride'th sample)

In the second step of vector quantization, an Expectation-Maximization (EM) algorithm is applied to segregate each aggregated stream of vectors into a codebook of N Gaussians. Usually N is some power of 2, the commonly used number is N=256. The number 256 can in principle be varied, but this option is not provided in the SPHINX-II decoder. So if you intend to use the SPHINX-II decoder, but are training models with SPHINX-III trainer, you must use N=256. It has been observed that the quality of the models built with 256 codeword codebooks is sufficient for good recognition. Increasing the number of codewords may cause data-insufficiency problems. In many instances, the choice to train semi-continuous models (rather than continuous ones) arises from insufficiency of training data. When this is indeed the case, increasing the number of codebooks might aggravate the estimation problems that might arise due to data insufficiency. Consider this fact seriously before you decide to increase N.

In SPHINX-III, the EM-step is done through a k-means algorithm carried out by the executable **kmeans_init**. This executable is usually used with the following flag settings:

```
     -grandvar   yes
     -gthobj     single
     -stride     1
     -ntrial     1
     -minratio   0.001
     -ndensity   256
     -meanfn     full_path_to_codebookmeans.file
     -varfn      full_path_to_codebookvariances.file
     -reest      no
     -segdmpdirs directory_in_which_you_want_to_put_aggregate.file
     -segdmpfn   aggregate.file
     -ceplen     dimensionality_of_feature_vector
     -feat       type_of_feature
```

```
    -agc        automatic_gain_control_factor(max/none)
    -cmn        cepstral_mean_normalization(yes/no)
```

Once the vector quantization is done, you have to flat-initialize your acoustic models to prepare for the first real step in training. The following steps explain the flat-initialization process:

TRAINING SEMI-CONTINUOUS MODELS

CREATING THE CI MODEL DEFINITION FILE

This procedure is the same as described for continuous models.

TRAINING SEMI-CONTINUOUS MODELS

CREATING THE HMM TOPOLOGY FILE

This procedure is the same as described for continuous models.

TRAINING SEMI-CONTINUOUS MODELS

FLAT INITIALIZATION OF CI MODEL PARAMETERS

In flat-initialization, all mixture weights are set to be equal for all states, and all state transition probabilities are set to be equal. Unlike in continuous models, the means and variances of the codebook Gaussians are not given global values, since they are already estimated from the data in the vector quantization step. To flat-initialize the mixture weights, each component of each mixture-weight distribution of each feature stream is set to be a number equal to 1/N, where N is the codebook size. The mixture_weights and transition_matrices are initialized using the executable **mk_flat**. It needs the following arguments:

FLAG	DESCRIPTION
-moddeffn	CI model definition file
-topo	HMM topology file.
-mixwfn	file in which you want to write the initialized mixture weights
-tmatfn	file in which you want to write the initialized transition matrices
-nstream	number of independent feature streams, for continuous models this number should be set to "1", without the double quotes
-ndensity	codebook size. This number is usually set to "256", without the double quotes

TRAINING SEMI-CONTINUOUS MODELS

TRAINING CI MODELS

This procedure is the same as described for continuous models, except

1. For the executable **bw**, the flags -tst2cbfn, -topn and -feat must be set to the values

FLAG	VALUE
-tst2cbfn	.semi.
-topn	This value should be lower than or equal to the codebook size. It decides how many components of each mixture weight distribution are used to estimate likelihoods during the baum-welch passes. Itaffects the speed of training. A higher value results in slower iterations
-feat	The specific feature type you are using to train the semi-continuous models

2. For the executable **norm**, the flag -feat must be set to the value

FLAG	VALUE
-feat	The specific feature type you are using to train the semi-continuous models

Also, it is important to remember here that the re-estimated means and variances now correspond to *codebook* means and variances. In semi-continuous models, the codebooks are also re-estimated during training. The vector quantization step is therefore only an *initialization* step for the codebooks. This fact will affect the way we do model adaptation for the semi-continuous case.

TRAINING SEMI-CONTINUOUS MODELS

CREATING THE CD UNTIED MODEL DEFINITION FILE

This procedure is the same as described for continuous models.

TRAINING SEMI-CONTINUOUS MODELS

FLAT INITIALIZATION OF CD UNTIED MODEL PARAMETERS

This procedure is the same as described for continuous models.

TRAINING SEMI-CONTINUOUS MODELS

TRAINING CD UNTIED MODELS

This procedure is the same as described for continuous models.

TRAINING SEMI-CONTINUOUS MODELS

BUILDING DECISION TREES FOR PARAMETER SHARING

This procedure is the same as described for continuous models.

TRAINING SEMI-CONTINUOUS MODELS

GENERATING THE LINGUISTIC QUESTIONS

This procedure is the same as described for continuous models.

TRAINING SEMI-CONTINUOUS MODELS

PRUNING THE DECISION TREES

This procedure is the same as described for continuous models.

TRAINING SEMI-CONTINUOUS MODELS

CREATING THE CD TIED MODEL DEFINITION FILE

This procedure is the same as described for continuous models.

TRAINING SEMI-CONTINUOUS MODELS

INITIALIZING AND TRAINING CD TIED MODELS

During initialization, the model parameters from the CI model parameter files are copied into appropriate positions in the CD tied model parameter files. Four model parameter files are created, one each for the means, variances, transition matrices and mixture weights. During initialization, each state of a particular CI phone contributes to the same state of the same CI phone in the CD-tied model parameter file, and also to the same state of the *all* the triphones of the same CI phone in the CD-tied model parameter file. The CD-tied model definition file is used as a reference for this mapping.

Initialization for the CD-tied training is done by the executable called **init_mixw**. It requires the following arguments:

-src_moddeffn	source (CI) model definition file
-src_ts2cbfn	.semi.
-src_mixwfn	source (CI) mixture-weight file
-src_meanfn	source (CI) means file
-src_varfn	source (CI) variances file
-src_tmatfn	source (CI) transition-matrices file
-dest_moddeffn	destination (CD tied) model def inition file
-dest_ts2cbfn	.semi.
-dest_mixwfn	destination (CD tied) mixture wei ghts file
-dest_meanfn	destination (CD tied) means file
-dest_varfn	destination (CD tied) variances fi le
-dest_tmatfn	destination (CD tied) transition matrices file
-feat	feature configuration
-ceplen	dimensionality of base feature vector

@@

TRAINING SEMI-CONTINUOUS MODELS

DELETED INTERPOLATION

Deleted interpolation is the final step in creating semi-continuous models. The output of deleted interpolation are semi-continuous models in sphinx-3 format. These have to be further converted to sphinx-2 format, if you want to use the SPHINX-II decoder.

Deleted interpolation is an iterative process to interpolate between CD and CI mixture-weights to reduce the effects of overfitting. The data are divided into two sets, and the data from one set are used to estimate the optimal interpolation factor between CI and CD models trained from the other set. Then the two data sets are switched and this procedure is repeated using the last estimated interpolation factor as an initialization for the current step. The switching is continued until the interpolation factor converges.

To do this, we need *two* balanced data sets. Instead of the actual data, however, we use the Bauim-Welch buffers, since the related math is convenient. we therefore need an *even* number of buffers that can be grouped into two sets. DI cannot be performed if you train using only one buffer. At least in the final iteration of the training, you must perform the training in (at least) two parts. You could also do this serially as one final iteration of training AFTER BW has converegd, on a non-lsf setup.

Note here that the norm executable used at the end of every Baum-Welch iteration also computes models from the buffers, but it does not require an even number of buffers. BW returns numerator terms and denominator terms for the final estimation, and norm performs the actual division. The number of buffers is not important, but you would need to run norm at the end of EVERY iteration of BW, even if you did the training in only one part. When you have multiple parts norm sums up the numerator terms from the various buffers, and the denominator terms, and then does the division.

The executable "delint" provided with the SPHINX-III package does the deleted interpolation. It takes the following arguments:

FLAG	DESCRIPTION
-accumdirs	directory which holds the baum-welch buffers
-moddeffn	CD-tied model-definition file
-mixwfn	CD-tied mixture weights files
-cilambda	initial interpolation factor between the CI models and the Cd models. It is the weight given given to the CI models initially. The values range from 0 to 1. This is typically set to 0.9
-ceplen	dimentionality of base feature vector
-maxiter	the number of iterations of deleted-interpolation that you want to run. DI can be slow to converge, so this number is typically between 1000-4000

(more to come...) *After the decision trees are built using semi-continuous models, it is possible to train continuous models. ci-semicontinuous models need to be trained for initializing the semicontinuous untied models. ci-continuous models need to be trained for initializing the continuous tied state models. the feature set can be changed after the decision tree building stage.* Back to index

SPHINX2 data and model formats

1. Feature set: This is a binary file with all the elements in each of the vectors stored sequentially. The header is a 4 byte integer which tells us how many floating point numbers there are in the file. This is followed by the actual cepstral values (usually 13 cepstral values per frame, with 10ms skip between adjacent frames. Framesize is usually fixed and is usually 25ms).

```
               <4_byte_integer header>
               vec 1 element 1
               vec 1 element 2
                 .
                 .
               vec 1 element 13
               vec 2 element 1
               vec 2 element 2
                .
                .
               vec 2 element 13
```

2. Sphinx2 semi-continuous HMM (SCHMM) formats:
   The sphinx II SCHMM format is rather complicated. It has the following main components (each of which has sub-components):
   - A set of codebooks
   - A "sendump" file that stores state (senone) distributions
   - A "phone" and a "map" file which map senones on to states of a triphone
   - A set of ".chmm" files that store transition matrices

   1. Codebooks: There are 8 codebook files. The sphinx-2 uses a four stream feature set:
      - cepstral feature: [c1-c12], (12 components)
      - delta feature: [delta_c1-delta_c12,longterm_delta_c1-longterm_delta_c12],(24 components)
      - power feature: [c0,delta_c0,doubledelta_c0], (3 components)
      - doubledelta feature: [doubledelta_c-doubledelta_c12] (12 components)

      The 8 codebooks files store the means and variances of all the gaussians for each of these 4 features. The 8 codebooks are,
      - cep.256.vec [this is the file of means for the cepstral feature]
      - cep.256.var [this is the file of variacnes for the cepstral feature]
      - d2cep.256.vec [this is the file of means for the delta cepstral feature]
      - d2cep.256.var [this is the file of variances for the delta cepstral feature]
      - p3cep.256.vec [this is the file of means for the power feature]
      - p3cep.256.var [this is the file of variances for the power feature]
      - xcep.256.vec [this is the file of means for the double delta feature]
      - xcep.256.var [this is the file of variances for the double delta feature]

      All files are binary and have the following format: [4 byte int][4 byte float][4 byte float][4 byte float]...... The 4 byte integer header stores the number of floating point values to follow in the file. For the cep.256.var, cep.256.vec, xcep.256.var and xcep.256.vec this value should be 3328. For d2cep.* it should be 6400, and for p3cep.* it should be 768. The floating point numbers are the components of the mean vectors (or variance vectors) laid end to end. So cep.256.[vec,var] have 256 mean (or variance) vectors, each 13 dimensions long, d2cep.256.[vec,var] have 256 mean/var vectors, each 25 dimensions long, p3cep.256.[vec,var] have 256 vectors, each of dimension 3, xcep.256.[vec,var] have 256 vectors of length 13 each.

The 0th component of the cep,d2cep and xcep distributions are not used in likelihood computation and are part of the format for purely historical reasons.

2. The "sendump" file: The "sendump" file stores the mixture weights of the states associated with each phone. (this file has a little ascii header, which might help you a little). Except for the header, this is a binary file. The mixture weights have all been transformed to 8 bit integer by the following operation intmixw = (-log(float mixw) >> shift) The log base is 1.0003. The "shift" is the number of bits the smallest mixture weight has to be shifted right to fit in 8 bits. The sendump file stores,

```
for each feature (4 features in all)
   for each codeword (256 in all)
     for each ci-phone (including noise phones)
       for each tied state associated with ci phone,
         probability of codeword in tied state
       end
       for each CI state associated with ci phone, ( 5 states )
         probability of codeword in CI state
       end
     end
   end
 end
```

The sendump file has the following storage format (all data, except for the header string are binary):

```
Length of header as 4 byte int (including terminating '\0')
 HEADER string (including terminating '\0')
 0 (as 4 byte int, indicates end of header strings).
 256 (codebooksize, 4 byte int)
 Num senones (Total number of tied states, 4 byte int)
 [lut[0],    (4 byte integer, lut[i] = -(i"<<"shift))
 prob_of_codeword[0]_of_feat[0]_1st_CD_sen_of_1st_ciphone (unsigned char)
 prob_of_codeword[0]_of_feat[0]_2nd_CD_sen_of_1st_ciphone (unsigned char)
 ..
 prob_of_codeword[0]_of_feat[0]_1st_CI_sen_of_1st_ciphone (unsigned char)
 prob_of_codeword[0]_of_feat[0]_2nd_CI_sen_of_1st_ciphone (unsigned char)
 ..
 prob_of_codeword[0]_of_feat[0]_1st_CD_sen_of_2nd_ciphone (unsigned char)
 prob_of_codeword[0]_of_feat[0]_2nd_CD_sen_of_2nd_ciphone (unsigned char)
 ..
 prob_of_codeword[0]_of_feat[0]_1st_CI_sen_of_2st_ciphone (unsigned char)
 prob_of_codeword[0]_of_feat[0]_2nd_CI_sen_of_2st_ciphone (unsigned char)
 ..
 ]
 [lut[1],    (4 byte integer)
 prob_of_codeword[1]_of_feat[0]_1st_CD_sen_of_1st_ciphone (unsigned char)
 prob_of_codeword[1]_of_feat[0]_2nd_CD_sen_of_1st_ciphone (unsigned char)
 ..
 prob_of_codeword[1]_of_feat[0]_1st_CD_sen_of_2nd_ciphone (unsigned char)
 prob_of_codeword[1]_of_feat[0]_2nd_CD_sen_of_2nd_ciphone (unsigned char)
 ..
 ]
 ... 256 times ..
 Above repeats for each of the 4 features
```

3. PHONE file: The phone file stores a list of phones and triphones used by the decoder. This is an ascii file It has 2 sections. The first section lists the CI phones in the models and consists of lines of the format

   ```
   AA      0       0       8       8
   ```

   "AA" is the CI phone, the first "0" indicates that it is a CI phone, the first 8 is the index of the CI phone, and the last 8 is the line number in the file. The second 0 is there for historical reasons.

   The second section lists TRIPHONES and consists of lines of the format

   ```
   A(B,C)P -1 0 num num2
   ```

   "A" stands for the central phone, "B" for the left context, and "C" for the right context phone. The "P" stands for the position of the triphone and can take 4 values "s","b","i", and "e", standing for single word, word beginning, word internal, and word ending triphone. The -1 indicates that it is a triphone and not a CI phone. num is the index of the CI phone "A", and num2 is the position of the triphone (or ciphone) in the list, essentially the number of the line in the file (beginning with 0).

4. map file: The "map" file stores a mapping table to show which senones each state of each triphone are mapped to. This is also an ascii file with lines of the form

   ```
    AA(AA,AA)s<0>       4
    AA(AA,AA)s<1>      27
    AA(AA,AA)s<2>      69
    AA(AA,AA)s<3>      78
    AA(AA,AA)s<4>     100
   ```

   The first line indicates that the 0th state of the triphone "AA" in the context of "AA" and "AA" is modelled by th 4th senone associated with the CI phone AA. Note that the numbering is specific to the CI phone. So the 4th senone of "AX" would also be numbered 4 (but this should not cause confusion)

5. chmm FILES: There is one *.chmm file per ci phone. Each stores the transition matrix associated with that partiular ci phone in following binary format. (Note all triphones associated with a ci phone share its transition matrix) (all numbers are 4 byte integers):
   - -10 (a header to indicate this is a tmat file)
   - 256 (no of codewords)
   - 5 (no of emitting states)
   - 6 (total no. of states, including non-emitting state)
   - 1 (no. of initial states. In fbs8 a state sequence can only begin with state[0]. So there is only 1 possible initial state)
   - 0 (list of initial states. Here there is only one, namely state 0)
   - 1 (no. of terminal states. There is only one non-emitting terminal state)
   - 5 (id of terminal state. This is 5 for a 5 state HMM)
   - 14 (total no. of non-zero transitions allowed by topology)

     ```
       [0 0 (int)log(tmat[0][0]) 0]   (source, dest, transition prob, source id)
     ```

```
[0 1 (int)log(tmat[0][1]) 0]
[1 1 (int)log(tmat[1][1]) 1]
[1 2 (int)log(tmat[1][2]) 1]
[2 2 (int)log(tmat[2][2]) 2]
[2 3 (int)log(tmat[2][3]) 2]
[3 3 (int)log(tmat[3][3]) 3]
[3 4 (int)log(tmat[3][4]) 3]
[4 4 (int)log(tmat[4][4]) 4]
[4 5 (int)log(tmat[4][5]) 4]
[0 2 (int)log(tmat[0][2]) 0]
[1 3 (int)log(tmat[1][3]) 1]
[2 4 (int)log(tmat[2][4]) 2]
[3 5 (int)log(tmat[3][5]) 3]
```

There are thus 65 integers in all, and so each *.chmm file should be 65*4 = 260 bytes in size.

(more to come...)

Back to index

SPHINX3 data and model formats

All senone-ids in the model files are with reference to the corresponding model-definition file for the model-set.

The means file

The ascii means file for 8 Gaussians/state 3-state HMMs looks like this:

```
param 602 1 8
mgau 0
feat 0

density 0 6.957e-01 -8.067e-01 -6.660e-01 3.402e-01 -2.786e-03 -1.655e-01
2.2 56e-02 9.964e-02 -1.237e-01 -1.829e-01 -3.777e-02 1.532e-03 -9.610e-01
-3.883e-0 1 5.229e-01 2.634e-01 -3.090e-01 4.427e-02 2.638e-01 -4.245e-02
-1.914e-01 -5.52 1e-02 8.603e-02 3.466e-03 5.120e+00 1.625e+00 -1.103e+00
1.611e-01 5.263e-01 2.4 79e-01 -4.823e-01 -1.146e-01 2.710e-01 -1.997e-05
-3.078e-01 4.220e-02 2.294e-01
 1.023e-02 -9.163e-02

density 1 5.216e-01 -5.267e-01 -7.818e-01 2.534e-01 6.536e-02 -1.335e-01
-1.3 22e-01 1.195e-01 5.900e-02 -2.095e-01 -1.349e-01 -8.872e-02 -4.965e-01
-2.829e-0 1 5.302e-01 2.054e-01 -2.669e-01 -2.415e-01 2.915e-01 1.406e-01
-1.572e-01 -1.50 1e-01 2.426e-02 1.074e-01 5.301e+00 7.020e-01 -8.537e-01
1.448e-01 3.256e-01 2.7 09e-01 -3.955e-01 -1.649e-01 1.899e-01 1.983e-01
-2.093e-01 -2.231e-01 1.825e-01
 1.667e-01 -2.787e-02

density 2 5.844e-01 -8.953e-01 -4.268e-01 4.602e-01 -9.874e-02 -1.040e-01
-3.  739e-02 1.566e-01 -2.034e-01 -8.387e-02 -3.551e-02 4.647e-03
-6.439e-01 -8.252e- 02 4.776e-01 2.905e-02 -4.012e-01 1.112e-01 2.325e-01
-1.245e-01 -1.147e-01 3.39 0e-02 1.048e-01 -7.266e-02 4.546e+00 8.103e-01
-4.168e-01 6.453e-02 3.621e-01 1.  821e-02 -4.503e-01 7.951e-02 2.659e-01
-1.085e-02 -3.121e-01 1.395e-01 1.340e-01
 -5.995e-02 -7.188e-02
```

```
.....

.....

density 7 6.504e-01 -3.921e-01 -9.316e-01 1.085e-01 9.951e-02 7.447e-02
-2.42 3e-01 -8.710e-03 7.210e-02 -7.585e-02 -9.116e-02 -1.630e-01
-3.008e-01 -3.175e-0 1 1.687e-01 3.389e-01 -3.703e-02 -2.052e-01 -3.263e-03
1.517e-01 8.243e-02 -1.40 6e-01 -1.070e-01 4.236e-02 5.143e+00 5.469e-01
-2.331e-01 1.896e-02 8.561e-02 1.  785e-01 -1.197e-01 -1.326e-01 -6.467e-02
1.787e-01 5.523e-02 -1.403e-01 -7.172e- 02 6.666e-02 1.146e-01

mgau 1
feat 0

density 0 3.315e-01 -5.500e-01 -2.675e-01 1.672e-01 -1.785e-01 -1.421e-01
9.0 70e-02 1.192e-01 -1.153e-01 -1.702e-01 -3.114e-02 -9.050e-02 -1.247e-01
3.489e-0 1 7.102e-01 -2.001e-01 -1.191e-01 -6.647e-02 2.222e-01 -1.866e-01
-1.067e-01 1.0 52e-01 7.092e-02 -8.763e-03 5.029e+00 -1.354e+00 -2.135e+00
2.901e-01 5.646e-01 1.525e-01 -1.901e-01 4.672e-01 -3.508e-02 -2.176e-01
-2.031e-01 1.378e-01 1.029e -01 -4.655e-02 -2.512e-02

density 1 4.595e-01 -8.823e-01 -4.397e-01 4.221e-01 -2.269e-03 -6.014e-02
-7.  198e-02 9.702e-02 -1.705e-01 -6.178e-02 -4.066e-02 9.789e-03
-3.188e-01 -8.284e- 02 2.702e-01 6.192e-02 -2.077e-01 2.683e-02 1.220e-01
-4.606e-02 -1.107e-01 1.16 9e-02 8.191e-02 -2.150e-02 4.214e+00 2.322e-01
-4.732e-02 1.834e-02 8.372e-02 -7 .559e-03 -1.111e-01 -3.453e-03 5.487e-02
2.355e-02 -8.777e-02 4.309e-02 3.460e-0 2 -1.521e-02 -3.808e-02
```

This is what it means, reading left to right, top to bottom:

Parameters for 602 tied-states (or senones), 1 feature stream, 8 Gaussians per state.

Means for senone no. 0, feature-stream no. 0. Gaussian density no. 0, followed by its 39-dimensional mean vector. (Note that each senone is a mixture of 8 gaussians, and each feature vector consists of 13 cepstra, 13 delta cepstra and 13 double delta cepstra)

```
Gaussian density no. 1, followed by its 39-dimensional mean vector.
Gaussian density no. 2, followed by its 39-dimensional mean vector.
.....
.....
Gaussian density no. 7, followed by its 39-dimensional mean vector.

Means for senone no. 1, feature-stream no. 0.
Gaussian density no. 0, followed by its 39-dimensional mean vector.
Gaussian density no. 1, followed by its 39-dimensional mean vector.
```

- and so on -

The variances file

```
param 602 1 8
mgau 0
feat 0

density 0 1.402e-01 5.048e-02 3.830e-02 4.165e-02 2.749e-02 2.846e-02
```

```
2.007e- 02 1.408e-02 1.234e-02 1.168e-02 1.215e-02 8.772e-03 8.868e-02
6.098e-02 4.579e- 02 4.383e-02 3.646e-02 3.460e-02 3.127e-02 2.336e-02
2.258e-02 2.015e-02 1.359e- 02 1.367e-02 1.626e+00 4.946e-01 3.432e-01
7.133e-02 6.372e-02 4.693e-02 6.938e- 02 3.608e-02 3.147e-02 4.044e-02
2.396e-02 2.788e-02 1.934e-02 2.164e-02 1.547e- 02

density 1 9.619e-02 4.452e-02 6.489e-02 2.388e-02 2.337e-02 1.831e-02
1.569e- 02 1.559e-02 1.082e-02 1.008e-02 6.238e-03 4.387e-03 5.294e-02
4.085e-02 3.499e- 02 2.327e-02 2.085e-02 1.766e-02 1.781e-02 1.315e-02
1.367e-02 9.409e-03 7.189e- 03 4.893e-03 1.880e+00 3.342e-01 3.835e-01
5.274e-02 4.430e-02 2.514e-02 2.516e- 02 2.863e-02 1.982e-02 1.966e-02
1.742e-02 9.935e-03 1.154e-02 8.361e-03 8.059e- 03

density 2 1.107e-01 5.627e-02 2.887e-02 2.359e-02 2.083e-02 2.143e-02
1.528e- 02 1.264e-02 1.223e-02 9.553e-03 9.660e-03 9.241e-03 3.391e-02
2.344e-02 2.220e- 02 1.873e-02 1.436e-02 1.458e-02 1.362e-02 1.350e-02
1.191e-02 1.036e-02 8.290e- 03 5.788e-03 1.226e+00 1.287e-01 1.037e-01
3.079e-02 2.692e-02 1.870e-02 2.873e- 02 1.639e-02 1.594e-02 1.453e-02
1.043e-02 1.137e-02 1.086e-02 8.870e-03 9.182e- 03
```

- and so on - The format is exactly as for the means file.

The mixture_weights file

The ascii mixure_weights file for 8 Gaussians/state 3-state HMMs looks like this:

```
mixw 602 1 8

mixw [0 0] 7.434275e+03
8.697e+02 9.126e+02 7.792e+02 1.149e+03 9.221e+02 9.643e+02 1.037e+03 8.002e+02

mixw [1 0] 8.172642e+03
8.931e+02 9.570e+02 1.185e+03 1.012e+03 1.185e+03 9.535e+02 7.618e+02 1.225e+03
```

This is what it means, reading left to right, top to bottom:

Mixtrue weights for 602 tied-states (or senones), 1 feature stream, 8 Gaussians per state (Each mixture weight is a vector with 8 components)

Mixture weights for senone no. 0, feature-stream no. 0, number of times this senone occured in the training corpus (instead of writing normalized values, this number is directly recorded since it is useful in other places during training [interpolation, adaptation, tree building etc]). When normalized (for example, by the decoder during decoding), the mixture weights above would read as: mixw 602 1 8 mixw [0 0] 7.434275e+03 1.170e-01 1.228e-01 1.048e-01 1.546e-01 1.240e-01 1.297e-01 1.395e-01 1.076e-01 mixw [1 0] 8.172642e+03 1.093e-01 1.171e-01 1.450e-01 1.238e-01 1.450e-01 1.167e-01 9.321e-02 1.499e-01

The transition_matrices file

The ascii file looks like this:

```
tmat 34 4
tmat [0]
 6.577e-01 3.423e-01
          6.886e-01 3.114e-01
```

```
                  7.391e-01 2.609e-01
tmat [1]
 8.344e-01 1.656e-01
         7.550e-01 2.450e-01
                  6.564e-01 3.436e-01
tmat [2]
 8.259e-01 1.741e-01
         7.598e-01 2.402e-01
                  7.107e-01 2.893e-01
tmat [3]
 4.112e-01 5.888e-01
         4.371e-01 5.629e-01
                  5.623e-01 4.377e-01
```

- and so on - This is what it means, reading left to right, top to bottom:

Transition matrices for 34 HMMs, each with four states (3 emitting states + 1 non-emitting state)

Transition matrix for HMM no. 0 (NOTE THAT THIS IS THE HMM NUMBER, AND NOT THE SENONE NUMBER), matrix.

```
Transition matrix for HMM no 1, matrix.
Transition matrix for HMM no 2, matrix.
Transition matrix for HMM no 3, matrix.
```

- and so on -

Explanation of the feature-vector components:

The 13 dimensional cepstra, 13 dimensional delta cepstra and 13 dimensional double-delta cepstra are arranged, in all model files, in the following order: 1s_12c_12d_3p_12dd (you can denote this by s3_1x39 in the decoder flags). The format string means: 1 feature-stream, 12 cepstra, 12 deltacepstra, 3 power and 12 doubledeltacepstra. The power part is composed of the 0th component of the cepstral vector, 0th component of the d-cepstral vector and 0th component of the dd-cepstral vector.

In the quantized models, you will see the string 24,0-11/25,12-23/26,27-38 In this string, the slashes are delimiters. The numbers represent various components occuring in each of the 3 codebooks. In the above string, for instance, the first codebook is composed of the 24th component of the feature vector (s3_1x39) followed by components 0-11. The second codeword has components 25, followed by components 12-23, and the third codeword is composed of components 26 and 27-28. This basically accounts for the odd order in s3_1x39. By constructing the codewords in this manner, we ensure that the first codeword is composed entirely of cepstral terms, the second codeword of delta cepstral terms and the third codeword of double delta terms.

s3_1x39 is a historical order. It can be disposed of in any new code that you write. Writing the feature vector components in different orders has no effect on recognition, provided training and test feature formats are the same.

TRAINING MULTILINGUAL MODELS

Once you have acoustic data and the corresponding transcriptions for any language, and a lexicon which translates words used in the transcription into sub-word units (or just maps them into some reasonable-looking acoustic units), you can use the SPHINX to train acoustic models for that language. You do not need anything else.

The linguistic questions that are needed for building the decision trees are automatically designed by the SPHINX. Given the acoustic units you choose to model, the SPHINX can automatically determine the best combinations of these units to compose the questions. The hybrid algorithm that the SPHINX uses clusters state distributions of context-independent phones to obtain questions for triphonetic contexts. This is very useful if you want to train models for languages whose phonetic structure you do not know well enough to design your own phone classes (or if a phonetician is not available to help you do it). An even greater advantage comes from the fact that the algorithm can be effectively used in situations where the subword units are not phonetically motivated. Hence you can comfortably use any set of acoustic units that look reasonable to you for the task.

If you are completely lost about the acoustic units but have enough training data for all (or most) words used in the transcripts, then build word models instead of subword models. You do not have to build decision trees. Word models are usually context-independent models, so you only have to follow through the CI training. Word models do have some limitations, which are currently discussed in the non-technical version of this manual.

Back to index

THE TRAINING LEXICON

Inconsistencies in the training lexicon can result in bad acoustic models. Inconsistencies stem from the usage of a phoneset with phones that are confusible in the pattern space of our recognizer. To get an idea about the confusibility of the phones that you are using, look at the per-frame log likelihoods of the utterances during training. A greater number of phones in the lexicon should ordinarily result in higher log likelihoods. If you have a baseline to compare with, and this is *not* the case, then it means that the phoneset is more diffuse over the pattern space (more compact, if you observe the opposite for a smaller phone set), and the corresponding distributions are wider (sharper in the other case). Generally, as the number of applicable distributions decreases over a given utterance, the variances tend to become larger and larger. The distributions flatten out since the areas under the distributions are individually conserved (to unity) and so the overall per frame likelihoods are expected to be lower.

The solution is to fix the phoneset, and to redo the lexicon in terms of a phoneset of smaller size covering the acoustic space in a more compact manner. One way to do this is to collapse the lexicon into syllables and longer units and to expand it again using a changed and smaller phoneset. The best way to do this is still a research problem, but if you are a native speaker of the language and have a good ear for sounds, your intuition will probably work. The SPHINX will, of course, be able to train models for any new phoneset you come up with.

Back to index

CONVERTING SPHINX3 FORMAT MODELS TO SPHINX2 FORMAT

To convert the 5 state/HMM, 4 feature stream semi-continuous models trained using the Sphinx3 trainer into the Sphinx2 format (compatible with the Sphinx2 decoder), programs in the following directories must be compiled and used:

```
-----------------------------------------------------------------------
program directory       corresponding    function
                        executable       of executable
-----------------------------------------------------------------------
mk_s2cb                 mk_s2cb          makes s2 codebooks
mk_s2hmm                mk_s2hmm         makes s2 mixture weights
mk_s2phone              mdef2phonemap    makes phone and map files
mk_s2seno               makesendmp       makes senone dmp files
-----------------------------------------------------------------------


Variables needed:
-----------------
s2dir  : sphinx_2_format directory
s3dir  : sphinx_3_format directory
s3mixw : s3dir/mixture_weights
s3mean : s3dir/means
s3var  : s3dir/variances
s3tmat : s3dir/transition_matrices
s3mdef : s3dir/mdef_file (MAKE SURE that this mdef file
                          includes all the phones/triphones needed for
                          the decode. It should ideally be made from
                          the decode dictionary, if the decode vocabulary
                          is fixed)


Usage:
------
mk_s2cb
        -meanfn   s3mean
        -varfn    s3var
        -cbdir    s2dir
        -varfloor 0.00001

mk_s2hmm
        -moddeffn s3mdef
        -mixwfn   s3mixw
        -tmatfn   s3tmat
        -hmmdir   s2dir

makesendmp
s2_4x $s3mdef .semi. $s3mixw 0.0000001 $s2dir/sendump
(the order is important)
cleanup: s2dir/*.ccode s2dir/*.d2code s2dir/*.p3code s2dir/*.xcode

mdef2phonemap
grep -v "^#" s3mdef | mdef2phonemap s2dir/phone s2dir/map
```

make sure that the mdef file used in the programs above includes all the triphones needed. The programs (especially the makesendmp program) will not work if any tied state is missing from the mdef file. This can happen if you ignore the dictionary provided with the models and try to make a triphone list using another dictionary. Even though you may have the same phones, there may be enough triphones missing to leave out some leaves in the pruned trees altogether (since they cannot be associated with any of the new triphones states). To avoid this, use the dictionary provided. You may extend it by including new words.

UPDATING OR ADAPTING EXISTING MODELS SETS

In general one is better off training speaker specific models if sufficient data (at least 8-10 hours) are available. If you have less data for a speaker or a domain, then the better option is to adapt any existing models you have to the data. Exactly how you adapt would depend on the kind of acoustic models you're using. If you're using semi-continuous models, adaptation could be performed by interpolating speaker specific models with speaker-independent models. For continuous HMMs you would have to use MLLR, or one of its variants. To adapt or update existing semicontinuous models, follow these steps:

1. Compute features for the new training data. The features must be computed in the same manner as your old training features. In fact, the feature computation in the two cases must be identical as far as possible.
2. Prepare transcripts and dictionary for the new data. The dictionary must have the same phoneset as was used for training the models. The transcripts must also be prepared in the same manner. If you have new filler phones then the fillerdict must map them to the old filler phones.
3. The new training transcript and the corresponding ctl file can include the old training data IF all you are doing is using additional data from the SAME domain that you might have recently acquired. If you are adapting to a slightly different domain or slightly different acoustic conditions, then use only the new data.
4. Starting with the existing deleted-interpolated models, and using the same tied mdef file used for training the base models and the same training parameters like the difference features, number of streams etc., run through one or two passes of Baum-Welch. However, this must be done without re-estimating the means and variances. Only the mixture-weights must be re-estimated. If you are running the norm after the Baum-Welch, then make sure that the norm executable is set to normalize only the mixture weights.
5. Once the mixture weights are re-estimated, the new mixture weights must be interpolated with the ones you started with. The executable "mixw_interp" provided with the SPHINX package may be used for this. You can experiment with various mixing weights to select the optimal one. This is of course the simplest update/adaptation technique. There are more sophisticated techniques which will be explained here later.

The mixw_interp executable:

This is used in model adaptation for interpolating between two mixture weight files. It requires the following flags:

FLAG	DESCRIPTION
-SImixwfn	The original Speaker-Independent mixture weights file
-SDmixwfn	The Speaker Dependent mixture weight file that you have after the bw iterations for adaptation
-tokencntfn	The token count file
-outmixwfn	The output interpolated mixture weight parameter file name

-SIlambda	Weight given to SI mixing weights

USING THE SPHINX-III DECODER WITH SEMI-CONTINUOUS AND CONTINUOUS MODELS

There are two flags which are specific to the type of model being used, the rest of the flags are independent of model type. The flags you need to change to switch from continuous models to semi-continuous ones are:

- the -senmgaufn flag would change from ".cont." to ".semi."
- the -feat flag would change from the feature you are using with continuous models to the feature you are using with the semicontinuous models (usually it is s3_1x39 for continuous models and s2_4x for semi-continuous models)

Some of the other decoder flags and their usual settings are as follows:

```
         -logbase 1.0001 \
         -bestpath     0 \
         -mdeffn $mdef \
         -senmgaufn .cont. \
         -meanfn $ACMODDIR/means \
         -varfn $ACMODDIR/variances \
         -mixwfn $ACMODDIR/mixture_weights \
         -tmatfn $ACMODDIR/transition_matrices \
         -langwt  10.5  \
         -feat s3_1x39 \
         -topn 32 \
         -beam 1e-80 \
         -nwbeam 1e-40 \
         -dictfn $dictfn \
         -fdictfn $fdictfn \
         -fillpenfn $fillpenfn \
         -lmfn $lmfile \
         -inspen 0.2 \
         -ctlfn $ctlfn \
         -ctloffset $ctloffset \
         -ctlcount  $ctlcount \
         -cepdir $cepdir \
         -bptblsize 400000 \
         -matchsegfn $matchfile \
        -outlatdir $outlatdir \
         -agc none \
         -varnorm yes \
```

BEFORE YOU TRAIN

FORCE-ALIGNMENT

Multiple pronunciations are not automatically considered in the SPHINX. You have to mark the right pronunciations in the transcripts and insert the interword silences. For this

a) remove the non-silence fillers from your filler dictionary and put them in your regular dictionary

b) Remove *all* silence markers (<s>, <sil> and </s>) from your training transcripts

For faligning with semi-continuous models, use the binary s3align provided with the trainer package with the following flag settings. For faligning with continuous models, change the settings of the flags -senmgaufn (.cont.), -topn (no. of Gaussians in the Gaussian mixture modeling each HMM state), -feat (the correct feature set):

```
         -outsentfn
         -insentfn
         -ctlfn
         -ctloffset      0
         -ctlcount
         -cepdir
         -dictfn
         -fdictfn        < filler dictionary>
         -mdeffn
         -senmgaufn      .semi.
         -meanfn
         -varfn
         -mixwfn
         -tmatfn
         -topn           4
         -feat           s2_4x
         -beam           1e-90
         -agc
         -cmn
         -logfn
```

last modified: 22 Nov. 2000