10/30/2012

GUIDELINES FOR USING SPHINX

VERSION 1.0

Jigar Gada
IIT Bombay

Prof. Preeti Rao
IIT Bombay

Dr. Samudravijaya K.
Tata Institute of Fundamental Research, Mumbai

Acknowledgment: Nikul, Namrata, Vishal, Pinki, Pranav, Tejas

TABLE OF CONTENTS

Chapter1.
Introduction

CMU Sphinx is one of the most popular speech recognition applications for Linux and it can correctly capture words. It also gives the developers the ability to build speech systems, interact with voice and build something unique and useful. Speech recognition in Sphinx is based on Hidden Markov model (HMM).

An HMM-based system, like all other speech recognition systems, functions by first learning the characteristics (or parameters) of a set of sound units, and then using what it has learned about the units to find the most probable sequence of sound units for a given speech signal. The process of learning about the sound units is called ***training***. The process of using the knowledge acquired to deduce the most probable sequence of units in a given signal is called ***decoding***, or simply recognition.

Accordingly, you will need those components of the SPHINX system that you can use for training and for recognition. In other words, you will need the SPHINX ***Trainer*** and a SPHINX ***decoder.***

CMU Sphinx toolkit has a number of packages for different tasks and applications. It's sometimes confusing what to choose. To clean-up, here is the list:

- Sphinxbase — support library
- Sphinx3 v8 — adjustable, modifiable recognizer
- CMUclmtk — language model tools
- Sphinxtrain — acoustic model training tools

You can download CMU Sphinx Toolkit from sourceforge.

For installing Sphinx, Follow the guide:

Installation guide.

Chapter 2

Training

2.1 Overview of training

Basically 3 components of Acoustic Model are computed for training:

Q = q1, q2, q3,.....qn	the **subphones** represented as a set of **states**
A = a01 a02 . . . an1 . . . ann	a **subphone transition probability matrix A**, each aij representing the probability for each subphone of taking a **self-loop** or going to the **next** subphone. Together, Q and A implement a pronunciation lexicon , an HMM state graph structure for each word that the system is capable of recognizing.
B = $b_i(o_t)$	A set of **observation likelihoods**:, also called **emission probabilities**, each expressing the probability of a cepstral feature vector (observation o_t) being generated from subphone state i.

In order to train a system, we'll need a training corpus of utterances. For simplicity assume that the training corpus is separated into separate wavefiles, each containing a sequence of spoken utterances. For each wavefile, we'll need to know the correct sequence of words. We'll thus associate with each wavefile a transcription (a string of words). We'll also need a pronunciation lexicon (**dictionary**) and a **phoneset**, defining a set of (untrained) phone HMMs. From the transcription, lexicon, and phone HMMs, we can build a "whole sentence" HMM for each sentence, as shown in the figure below:

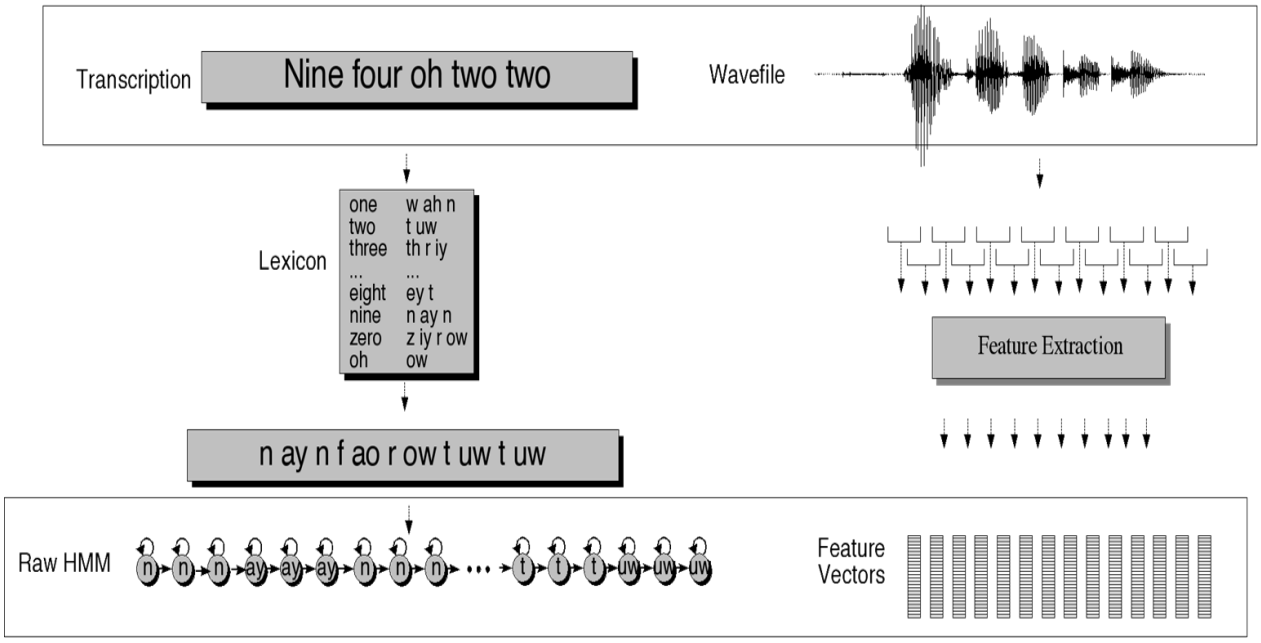


Fig: The input to the embedded training algorithm; a wave file of spoken digits with a corresponding transcription. The transcription is converted into a raw HMM, ready to be aligned and trained against the cepstral features extracted from the wav file.

This might have given you a fair idea of what all things are required for Training.

2.2 Files required for training:

- A **control file** (fileIDs) containing the list of filenames. It should **not** contain the extension **.wav**. An example of the entries in this file:

```
F13MH01A0011I302_silRemoved
F13MH01A0011I303_silRemoved
F13MH01A0011I304_silRemoved
F13MH01A0011I305_silRemoved
```

- A **transcript file** in which the transcripts corresponding to the wav files are listed in exactly the same order as the feature filenames in the control file. An example of the entries in this file:

```
<s> gahuu </s> (F13MH01A0011I305_silRemoved)
<s> jvaarii </s> (F13MH01A0011I306_silRemoved)
<s> makaa </s> (F13MH01A0011I307_silRemoved)
<s> paqdharaa </s> (F13MH01A0011I308_silRemoved)
```

- 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 i.e. You may have extra words not present in the transcripts. The dictionary must have all alternate pronunciations marked with parenthesized serial numbers starting from (2) for the second pronunciation. The marker (1) is omitted. Here's an example:

```
bsnl  b i e s e n e l
bsnl(2) b i SIL e s SIL e n SIL e l

ahamadapuura  a h ax m a d p uu r
ahamadapuura(2) a h m a d p uu r
```

- 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> : beginning-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 modelled 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 "+".

For example, the filler file may contain extra words as:

```
+hm+  +hm+
+laugh+ +laugh+
+vn+  +vn+
+babble+  +babble+
```

If your phone file has extra phones which are not used in your dictionary or your dictionary has phones which are not present in the phone file then the Sphinx trainer will throw an **error**. This is obvious because you are trying to train a phoneme for which there is no training data.

2.3 Checklist for Training

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. Verify the phonelist against the dictionary and fillerdict.

You need not worry about task 3 as the Phone file will be generated by the script.

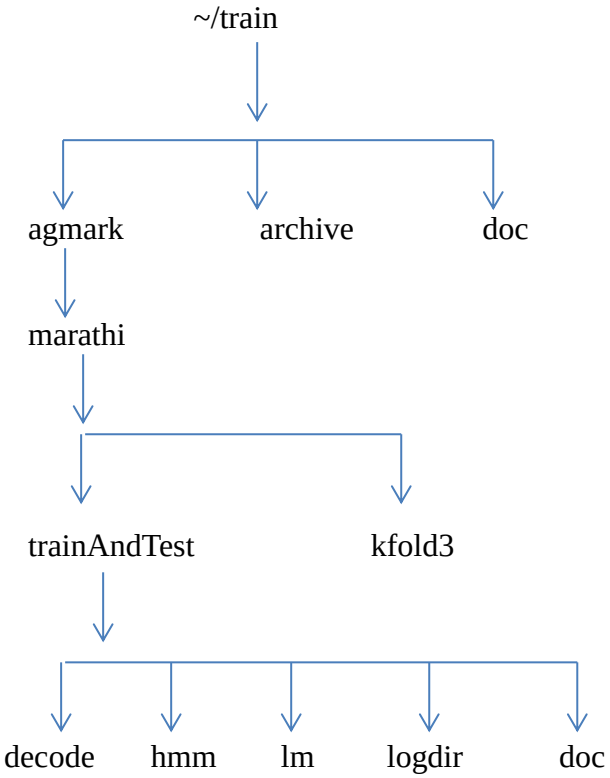
2.4 Summary

In summary, the basic embedded training procedure is as follows:
Given: phoneset, pronunciation lexicon (dictionary), and the transcribed wavefiles

1. Build a “whole sentence” HMM for each sentence.
2. Initialize A probabilities to 0.5 (for loop-backs or for the correct next subphone) or to zero (for all other transitions).
3. Initialize B probabilities by setting the mean and variance for each Gaussian to the global mean and variance for the entire training set.
4. Run multiple iterations of the Baum-Welch algorithm.

2.5 Training in Sphinx using the script

The SPHINX trainer consists of a set of programs, each responsible for a well-defined task and a set of scripts that organizes the order in which the programs are called. You have to compile the code in your favourite platform. Generally Linux is used. All the further details are w.r.t Linux.
To begin training, create a directory structure as follows to simplify tasks:



NOTE: ~ doesn't necessarily be /home. It can be as per your choice. It just represents the path before /train. In all the later cases, replace ~ with path before /train (**Very Imp**)

E.g. If my sphinx folder is stored at /home/jigar/train, then replace ~ by /home/jigar

The folders in this directory are

agmark

Refers to agmark commodities database.

agmark/marathi/

Directory which is used for train/test of agmark database of Marathi language.

agmark/marathi/trainAndTest/

This and its subdirectories are used to train a database and test on the same database using Sphinx3.

agmark/marathi/kfold3/

This and subdirectories are used to train and test a database using the kfold validation scheme.

All the scripts to test and train using the simple methodology of training and testing on same database are located at

~/sphinx3/agmark/marathi/trainAndTest/hmm/scripts_pl

The scripts at above location are a combination of c shell and perl scripts.

The following steps are to be followed to train and test (decode) the new database (here agmark):

(The asssumption is that the sphinx3 related programs are kept in folders inside /pub/tools.

If the programs are located at any other location then modify paths below accordingly.)

Set the following environment variables in **~/.cshrc** file

setenv SPHINXDIR /pub/tools/sphinx3 (decoder)

setenv SPHINXBASE /pub/tools/sphinxbase

setenv SPHINXTRAIN /pub/tools/sphinxtrain

setenv CMULM /pub/tools/cmuclmtk/

setenv SCLITE_HOME /pub/tools/sctk-2.4.0

setenv SPHINX3_HOME /home/YOURNAME/train (YOURNAME means your name as it appears under home)

SPHINX3_HOME is the path of the directory created above i.e. /home/jigar/train

1b.Add the following to the PATH variable in ~/.cshrc

/pub/tools/*/bin/* \

1c. Execute the following command to reflect the changes

source ~/.cshrc; rehash

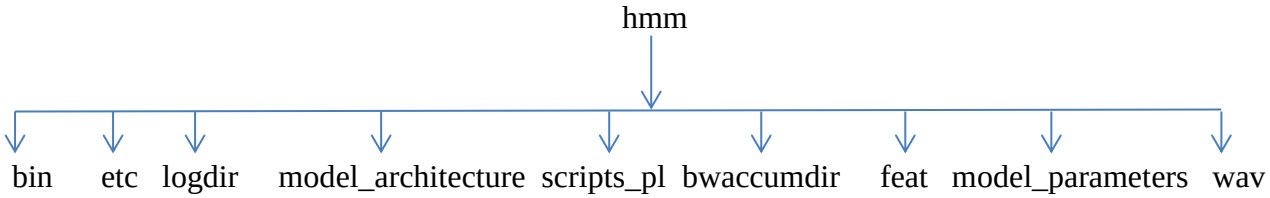
Once the Environment Variables and directory structure is set, run the following command in ~/train/agmark/marathi/trainAndTest/hmm

/pub/tools/sphinxtrain/scripts_pl/setup_SphinxTrain.pl -task $databaseName

//above path refers to where the sphinxtrain is installed. $databasename can be anything. You can set it to agmark. It can be changed later.

This will create the required directory structure in the /**hmm**.

After running this perl file, the directory structure in /hmm will look like this



Copy the following files in /hmm/scripts_pl

- initializeAndTrain.csh
- initializeAndTest.csh
- multiPronunciationDict.pl
- removeDisfluencyFromTrans.pl
- evaluateUsingSclite.csh
- RunAll_forceAlign.pl

- RunAll_forceAlign1.pl
- addSpkIdsToTrans.pl
- lmprepare.csh

Copy these files in /hmm/etc

- feat.params
- sphinx_train.cfg.template

Copy following files in ~/sphinx3/doc

- Dictionary
- Filler dictionary
- FileIds
- Transcription file

Copy following files in /hmm

- training.csh
- train_force_align.csh
- testing.csh

2.5.1 Make following database dependent changes in following files:

1. **initializeAndTrain.csh**
   If the above directory structure is maintained then no need to make changes else change the paths of ASR_HOME & DATA_HOME

   ASR_HOME --> path in which /hmm directory is present.

   In the above case it will be ~/train/agmark/marathi/trainAndTest

   DATA_HOME --> path in which /doc is present

   In the above case it will be ~/train

2. Open the file /hmm/etc/**sphinx_train.cfg.template** & make following changes:

   6. $CFG_BASE_DIR = "~/train/agmark/marathi/trainAndTest/hmm";

   // path to the /hmm directory

   8. $CFG_WAVFILES_DIR = "/media/F/common_database/marathi/AllWavFiles";

   // path where all the WavFiles are stored.

   23. $CFG_SPHINXTRAIN_DIR = "/home/raag/Pinki/data_trained/workspace/tools/sphinxtrain";

   // path where sphinxtrain is installed

3. If the speech data is sampled at 8000Hz, then make the following changes in etc/**feat.params**

   You can also change other parameters like lower & upper frequency limit (for telephonic data keep the upper frequency limit as 4000 or 3500 and lower frequency as 133), no. of fft points, window length, no. of Mel filters, dither etc.

   -samprate 8000

   -lowerf 133.33334

   -upperf 3500

   -nfft 256

   -wlen 0.025

   -nfilt 33

   -ncep 13

   -dither yes

4. Open the file **training.csh**

   Enter the number of senones and gaussians.

   To decide the number of senones, follow the link: No. of senones.

   Set the names of dictionary, filler dict, transcription and wavlist as stored in ~/train/doc/

   Enter the database name and path for DATA_HOME. (DATA_HOME will be ~/train)

After setting all parameters and arranging the directory, goto hmm/ and run the file training.csh in csh shell.

This will begin the training.

You can check the details of training in the /hmm/**trainLog."$database".txt** file. If there is any error it will be shown in the log file. So keep checking the log file at regular intervals to know the progress of training.

You should see output like this:

```
MODULE: 00 verify training files
O.S. is case sensitive ("A" != "a").
Phones will be treated as case sensitive.
  Phase 1: DICT - Checking to see if the dict and filler dict agrees with the phonelist file.
    Found 1219 words using 40 phones
  Phase 2: DICT - Checking to make sure there are not duplicate entries in the dictionary
  Phase 3: CTL - Check general format; utterance length (must be positive); files exist
  Phase 4: CTL - Checking number of lines in the transcript should match lines in control file
  Phase 5: CTL - Determine amount of training data, see if n_tied_states seems reasonable.
    Total Hours Training: 1.52594188034191
    This is a small amount of data, no comment at this time
  Phase 6: TRANSCRIPT - Checking that all the words in the transcript are in the dictionary
    Words in dictionary: 1216
    Words in filler dictionary: 3
```

Phase 7: TRANSCRIPT - Checking that all the phones in the transcript are in the phonelist, and all phones in the phonelist appear at least once.

The training process is organized in a list of successive stages, each of which builds on the results of the previous one. The first step (**01.vector_quantize**) constructs a generic model of the acoustic feature space without distinguishing phones. In the second step (**20.ci_hmm**), individual phones are trained without reference to their context. The third step (**30.cd_hmm_untied**) trains separate models for every triphone (meaning a phone with a specific left and right context). In the fourth step (**40.buildtrees**), these triphones are clustered using decision trees, and in the next step (**45.prunetree**), the trees are pruned to produce a set of basic acoustic units known as *senones* or *tied states*. Finally, senone models are trained in the last (**50.cd_hmm_tied**) step.

If everything is set up correctly, you won't have to run each step manually. You can just use the RunAll.pl script to run the entire training process:

perl scripts_pl/**RunAll.pl**

2.5.2 Explanation of script in brief:

training.csh

- It will change the name of dict, filler dict, transcription file and wavlist file to{$database}.dic, {$database}.filler, {$database}_train.transcription, {$database}_train.wavList in the doc folder itself.
- Change the databasename, no. of gaussians and senones in sphinx_train.cfg.template and change the databasename in initialiseAndTrain.csh
- Runs the script scripts_pl/initializeAndTrain.csh

initializeAndTrain.csh

- The main aim of this script is to remove any discrepancies present in the dict, filler dict, transcription and wavList files.
- It will create a Dictionary file which contains words only present in the transcription thus saving time while training.

  e.g. in my case the original transcription contained 2671 words while the new dictionary contained 1467 words.
- {$database}.phone file is created which will list the set of phones present in the dictionary.
- It will run the script make_feats.pl which will create an .mfc file for each of the wav files and store them in /hmm/feat/
- Training will finally begin by running the script scripts_pl/RunAll.pl

Once the training is completed, the models will be stored in /hmm/model_parameters_{$database}.

If the training is done for 1000 senones and 16 Gaussians then following dir will be present in the above folder:

${database}_s1000_g16.cd_cont_1000 ${database}_s1000_g16.cd_cont_1000_8

${database}_s1000_g16.cd_cont_1000_1 ${database}_s1000_g16.cd_cont_initial

${database}_s1000_g16.cd_cont_1000_16 ${database}_s1000_g16.cd_cont_untied

${database}_s1000_g16.cd_cont_1000_2 ${database}_s1000_g16.ci_cont

${database}_s1000_g16.cd_cont_1000_4 ${database}_s1000_g16.ci_cont_flatinitial

The models of usage are in the folder ${database}_s1000_g16.cd_cont_1000_16. Copy the **mdef** file present from model_architecture/${database}_s1000_g16.cd_cont_1000 to

${database}_s1000_g16.cd_cont_1000_16.

Thus all the **required files for decoding** are present in the folder

${database}_s1000_g16.cd_cont_1000_16.

Files which are present are:

- **means**
- **variances**
- **mixture_weights**
- **transition_matrices**
- **mdef**

For details of each of the file, follow the link:

Details of Model Files

You could also check these files by yourself. The files means, variances and mixture_weights and transition matrix are in binary form. To read these files, use the following command:
Goto model_parameters/${database}_s1000_g16.cd_cont_1000_16

- printp -gaufn means
- printp -gaufn variances
- printp -mixwfn mixture_weights
- printp -tmatfn transition_matrices

2.6 Scene behind Training in Sphinx (in brief)

For detailed Explanation, Follow the link below:

Sphinx Training elaborately.

2.6.1 CREATING THE CI MODEL DEFINITION FILE

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. During the training, the states are referenced only by these numbers.

Check the ${databse}.ci.mdef file in model_architecture/

79 n_base
No. of base phones

0 n_tri
No. of triphones

316 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 modelled by a 3-state HMM, this number will be
79 phones*4 states = 316

237 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, 79*3=237

237 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 modelling. This number is thus again the total number of emitting states, 79*3=237

79 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 79

2.6.2 CREATING THE HMM TOPOLOGY FILE

The number 4 is total the number of staes 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.
In this you have to specify the **no. of states** and **skipstate** as "yes" or "no" depending on whether you want the HMMs to have skipped state transitions or not.
The file looks as follows:

4
1 1 0 0
0 1 1 0
0 0 1 1

2.6.3 FLAT INITIALIZATION OF CI MODEL PARAMETERS

To begin training CI models, each of the files (means, variances, tmat, mixture_weights) must have some initial values. Global means and variances are computed using the vectors in the feature file an they are copied into means and variances of each state of each of the HMMs. The global mean and var file is stored in model_parameters/**${database}.ci_cont_flatinitial.**

2.6.4 TRAINING CONTEXT INDEPENDENT MODELS

Once flat initialization is done, you are ready to train the acoustic models for "base" or CI phones. Flat initialized models are re-estimated through the forward-backward or the Baum-Welch algorithm. Each iteration results in a better set of models. Training is done until you reach the convergence ratio. Trained CI models are stored in model_parameters/**${database}.ci_cont.**
The model parameters computed in the final iteration are now used to initialise the models for CD phones(triphones) with untied states.

2.6.5 CREATING THE CD UNTIED MODEL DEFINITION FILE

In this, HMM's are trained for all CD phones that are seen in the training data. First we create a mdef file for all the triphones in the training set. This file like the CI mdef file assigns unique Ids to each HMM state and serve as reference for handling CD-untied model parameters.
It is stored in model_architecture/**${database}.alltriphones.mdef.** (all the parameters will vary depending on the training data)

79 n_base

31496 n_tri

126300 n_state_map (30558*4 + 79*4)

94725 n_tied_state (30558*3 + 79*3)

237 n_tied_ci_state (79*3)

79 n_tied_tmat

All the triphones will have an atrribute:

b = word begining triphone
e = word ending triphone
i = word internal triphone
s = single word triphone

e.g. Consider the tri-phone **a SIL k i**
This means that the above triphone occurs in between a word.

Now, a Threshold is set based on the number of occurences of the triphone. The threshold is adjusted such that the total number of triphones above the threshold is less than the maximum number of triphones that the system can train. 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. It is stored in model_architecture/**${database}.untied.mdef**

79 n_base

5163 n_tri (Reduced no. of tri-phones)

20968 n_state_map

15726 n_tied_state

237 n_tied_ci_state

79 n_tied_tmat

2.6.6 FLAT INITIALIZATION OF CD UNTIED MODEL PARAMETERS

During this process, the model parameter files (means,variances,mixture_weights,tmat) corresponding to the CD untied model-definition file are generated. 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.

2.6.7 TRAINING CD UNTIED MODELS

Once the initialization is done, we train the CD-untied models using the Baum Welch Algorithm. The model parameters are stored in model_parameters/**${database}.cd_cont_untied.**

2.6.8 BUILDING DECISION TREES FOR PARAMETER SHARING

After CD-untied models, Decision trees are used to decode which of the HMM states of all the triphones are similar to each other so that data from all these states are collected together to train one global state i.e. **senone.**
For state tying, we require decision trees. The decision trees require CD-untied models and a set of pre-defined acoustic classes which share some common property.

We then generate linguistic questions to partition the data at any given node of a tree. Each question results in 1 partition and the question that results in best partition are chosen to partition data at that node. All linguistic questions are written in a single file:
model_architecture/**${database}.tree_questions file**. CI models are used to make linguistic questions.

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 as well as for SIL. So if there are 79 base phones with 3 states HMM, then **Total = 79*3 = 237 trees**.

2.6.9 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. 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.

2.6.10 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

The model parameters are srored in model_parameters/${database}.cd_cont_${no_of_gaussians}.

The mdef file for 2000 senones.

0.3

79 n_base

27546 n_tri

110500 n_state_map (27546*4 + 79*4)

2237 n_tied_state (2000 senones + 79*3 states)

237 n_tied_ci_state (79*3)

79 n_tied_tmat

2.6.11 INITIALIZING AND TRAINING CD TIED GAUSSIAN MIXTURE MODELS

HMM states can be modelled by either a single Gaussian distribution, or a mixture of Gaussian distributions. 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 initialize the training for 2 Gaussian per state models.

So the CD-tied training for models with 2^N Gaussians per state is done in N+ 1 step. 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 $N+1^{th}$ stage of CD-tied training)

The final set of parameters which are used foe decoding are stored in model_parameters/**${database}.cd_cont_${no_of_senones}_${no_of_gaussains}**

2.7 Training for different purpose:

2.7.1. Context Independent (CI) Training

For CI training, make the following changes in the file **RunAll.pl.**

```
my @steps =
    ("$ST::CFG_SCRIPT_DIR/00.verify/verify_all.pl",
     "$ST::CFG_SCRIPT_DIR/01.lda_train/slave_lda.pl",
     "$ST::CFG_SCRIPT_DIR/02.mllt_train/slave_mllt.pl",
     "$ST::CFG_SCRIPT_DIR/05.vector_quantize/slave.VQ.pl",
     "$ST::CFG_SCRIPT_DIR/10.falign_ci_hmm/slave_convg.pl",
     "$ST::CFG_SCRIPT_DIR/11.force_align/slave_align.pl",
     "$ST::CFG_SCRIPT_DIR/12.vtln_align/slave_align.pl",
     "$ST::CFG_SCRIPT_DIR/20.ci_hmm/slave_convg.pl",
     # "$ST::CFG_SCRIPT_DIR/30.cd_hmm_untied/slave_convg.pl",
     # "$ST::CFG_SCRIPT_DIR/40.buildtrees/slave.treebuilder.pl",
     # "$ST::CFG_SCRIPT_DIR/45.prunetree/slave.state-tying.pl",
     # "$ST::CFG_SCRIPT_DIR/50.cd_hmm_tied/slave_convg.pl",
     # "$ST::CFG_SCRIPT_DIR/60.lattice_generation/slave_genlat.pl",
     # "$ST::CFG_SCRIPT_DIR/61.lattice_pruning/slave_prune.pl",
     # "$ST::CFG_SCRIPT_DIR/62.lattice_conversion/slave_conv.pl",
     # "$ST::CFG_SCRIPT_DIR/65.mmie_train/slave_convg.pl",
     # "$ST::CFG_SCRIPT_DIR/90.deleted_interpolation/deleted_interpolation.pl",
    );

# Do the common initialization and state tying steps
foreach my $step (@steps) {
    my ($index) = ($step =~ m,.*/(\d\d)\.,);
    next if $index < $start;
    last if $index > $end;
    my $ret_value = RunScript($step);
    die "Something failed: ($step)\n" if $ret_value;
-- INSERT --
```

Comment all the section after **20.ci_hmm/slave_convg.pl.**

Follow the same procedure for training.
After training, the following directories will be present in hmm/model_parameters_{$database}

{$database}.ci_cont
{$database}.ci_cont_flatinitial

The CI models are present in the folder **{$database}.ci_cont** which can be used for decoding.

NOTE: In the above case the CI models have just **one** gaussian pdf per state. To train Multiple Gaussain CI models, make the following changes before training in **etc/sphinx_train.cfg.template** at around line 127:

(yes/no) Train multiple-gaussian context-independent models (useful

for alignment, use 'no' otherwise)

$CFG_CI_MGAU = 'no'; ------->> change this to '**Yes**' and train.

2.7.2 Force Aligned training

Sometimes audio in your database doesn't match the transcription properly. For example transcription file has the line “Hello world” but in audio actually “Hello hello world” is pronounced. Training process usually detects that and emits this message in the logs. If there are too many such errors it most likely means you misconfigured something, for example you had a mismatch between audio and the text caused by transcription reordering. If there are few errors, you can ignore them. You might want to edit the transcription file to put there exact word which were pronounced, in the case above you need to edit the transcription file and put “Hello hello world” on corresponding line. You might want to filter such prompts because they affect acoustic model quality. In that case you need to enable forced alignment stage in training. To do that edit **sphinx_train.cfg** line

$CFG_FORCEDALIGN = '**yes**';

and run training again. It will execute stages 10 and 11 and will filter your database.

Steps Involved:

A. Train CI models from original transcription.

B. Use the resulting models with s3align (from the s3flat distribution) to force-align, resulting in new transcription with pronunciations marked and silences inserted.

C. Use the forced-aligned transcript to re-train starting from the CI, and eventually to build CD models.

Initially copy the binary **sphinx3_align** from sphinx3/bin (where sphinx3 is the folder where the sphinx decoder is installed) in /hmm/bin for alignment. After this, change **CFG_FORCEDALIGN = 'yes'** in etc/sphinx_train.cfg and run the file scripts_pl/**RunAll_forceAlign.pl** for generating CI models. In RunAll_forceAlign.pl file, only stages up to 10 are run. After this run the script scripts_pl/**RunAll_forceAlign1.pl** which runs only stage 11 of RunAll.pl. This will create the force aligned transcripts which will be stored in the folder **falignout**. Check out the folder for more details. The force aligned transcripts are stored in the file **${database}.alignedtranscripts.**

Force aligned Transcripts will have extra silence markers inserted where needed and will also add pronunciation variant.

The force aligned file looks something like this:

```
<s> kRusxii utpanna baajaara samitii </s>  (F13MH01A0011I302_silRemoved)
<s> +babble+ baajaara <sil> samitii </s>  (F13MH01A0011I303_silRemoved)
<s> donashe baaviisa <sil> trepanna(2) caarashe ekasasxtxa(2) <sil> </s>
(F13MH01A0011I304_silRemoved)
<s> gahuu </s>  (F13MH01A0011I305_silRemoved)
```

As we can see <sil> is inserted in 2nd and 3rd wav file transcripts. Also pronunciation variants have been added in the 3rd file.

To avoid the above intricacy, run the script **train_force_align.csh** in the /hmm for force aligned training.

Explanation of the script:

- This script is same as the training.csh except for some additional changes:
- It copies the sphinx3_align binary in /hmm folder, runs the stages 10. & 11. for creating force aligned transcripts and re-trains using the new transcription.

Chapter 3

Sphinx Decoding

3.1 Theory

The decoder also consists of a set of programs, which have been compiled to give a single executable that will perform the recognition task, given the right inputs. The inputs that need to be given are: the trained acoustic models, a model index file, a language model, a language dictionary, a filler dictionary, and the set of acoustic signals that need to be recognized. The data to be recognized are commonly referred to as ***test data***.
In summary, the components provided to you for decoding will be:

1. The decoder source code
2. The language dictionary
3. The filler dictionary
4. The language model / Finite State Grammar(**FSG**)
5. The test data

In addition to these components, you will need the acoustic models that you have trained for recognition. You will have to provide these to the decoder. While you train the acoustic models, the trainer will generate appropriately named model-index files. A model-index file simply contains numerical identifiers for each state of each HMM, which are used by the trainer and the decoder to access the correct sets of parameters for those HMM states. With any given set of acoustic models, the corresponding model-index file must be used for decoding.

NOTE: Before using any of the scripts or commands mentioned below, first set the path of sphinxbase and sphinxDecoder in ~/.cshrc or ~/.bashrc file (depends on the shell you are using) so that you can access all the binaries of sphinx directly. Without doing this, you are going to get an error. If you have followed the training procedure then you have already done this. If not, refer the training section to set the environment variables.

Decoding can be done in one of the two ways:

1. Using n-gram Language Model.

To build an N-Gram language model for the task, you will need a sufficiently **large** sample of representative text. In real-world systems this is usually collected from digital text resources such as newswires and web sites, or it is created by transcribing speech data. The output of a language model can be a combination of words in your language model.
Details about n-gram model.

2. Using Finite state grammar (FSG)

Using a finite state grammar allows you to formally specify the language accepted by the speech recognizer. The output of an FSG has to be among the words specified in the gram mar. It cannot be a combination of words as in the case of n-gram language model.

Compare the 2 diagrams to understand what is written in the above 2 paragraphs:

1. LM diagram

2. FSG diagram

3.2 Preparing the n gram Language Model.

For preparing the Language model, only **transcription file** is required. This transcription file should be without **fileids** and **fillers.** This is because if the fillers are included then the trigrams, bi-grams and unigrams will also consist of fillers which we do not want.

e.g. -2.4742 +aah+ +aah+ -0.1573 (this is bigram model including fillers)

For detail of language model, follow the link:

Language Model

Run the following commands to create the Language Model:

➔ text2wfreq < transcription_file > data.wfreq

This command will generate a file data.wfreq which will give the frequency of each word in the transcription. It will look like this:

```
shahaanxnxava 119
piica 6
sheqgadaanxe 121
piika 8
piike 1
```

➔ wfreq2vocab < data.wfreq > data.vocab

This command will create file data.vocab which will list all the unique words from the transcription file. It will look like this:

```
aadhaara
aahe
aaheta
aaii
aajobaa
aalaa
```

➔ text2idngram -vocan data.vocab -idngram data.idngram < transcription_file

This will create file data.idngram which is an intermediate file for creating Language model file.

After this, create a file **data.ccs** which has following 2 lines:

```
<s>
</s>
```

➔ Idngram2lm -vocab data.vocab -idngram data.idngram -arpa data.lm -context data.ccs

This will create the language model file data.lm. However this is not in the binary format which is required for decoding. To do that we use the following command:

➔ sphinx3_lm_convert -i data.lm -o data.lm.DMP

data.lm.DMP is the final language model file which will be used for decoding.

You can also directly use the script **lmprepare_direct.csh** by setting the databasename and transciption fileName in the script.

set databaseName=data
// anyName which you want to set
set transFile=file.transcription
// transcription fileName. The file should be in the same folder as the script.
The language model file to be used for decoding is stored in **$databasename.lm.DMP**.

3.3 Preparing FSG

FSG details

Using a finite state grammar allows you to formally specify the language accepted by the speech recognizer. Internally, Sphinx uses a format to describe these grammars which is not easy for humans to write or read. However, a tool is provided called **sphinx_jsgf2fsg** which allows you to write a grammar in the JSpeech Grammar Format.

sphinx_jsgf2fsg is present in sphinxbase/bin/

Perl script **fsg.pl** is used to create the JSGF format.

Before running this file, you have to prepare the input file which will list all your words/sentences to be recognized.

You have to follow a particular **rule** for making the input file:

All the words/phrases in the input file should be **tab** separated. First 2 names in the file will be name of the FSG file which is tab separated. Following this, will be the words/ phrases which you want to recognize. All the words in a particular phrase should be space separated. All the words in the file should be in a single line.

E.g. input file:

```
district </t> fsg </t> navi </s> mumbai </t> laatuura </t> kolhaapuura </t> jalxagaava
```

So in the above input file, </t> is tab and </s> is space.

As per the rule, the name of the input file is district_fsg and the words/ phrases to be recognized are:

```
navi mumbai  (observe navi & mumbai in the input file are separated by space, not by tab)
laatuura
kolhaapuura
jalxagaava
```

Once you have created the input file, run the command

perl fsg.pl -l input_file

This will create an ouput file of JSGF format with the first 2 names you gave in input_file. The file will look like this:

```
#JSGF V1.0;

grammar district_fsg;

 public <district_fsg> = (navi mumbaii | laatuura | kolhaapuura | jalxagaava);
```

After you have created the JSGF grammar file, you need to convert it to a Sphinx FSG file using sphinx_jsgf2fsg. From the command-line, run:

sphinx_jsgf2fsg < district_fsg > district_fsg.fsg

This will create the Final FSG file which is stored in **district_fsg.fsg** and can be used for decoding. The file will look like this:

```
FSG_BEGIN <district_fsg.district_fsg>

NUM_STATES 9

START_STATE 0

FINAL_STATE 1

TRANSITION 0 2 1.000000

TRANSITION 2 4 0.250016 jalxagaava

TRANSITION 2 5 0.250016 kolhaapuura

TRANSITION 2 6 0.250016 laatuura

TRANSITION 2 7 0.250016 navi

TRANSITION 3 1 1.000000

TRANSITION 4 3 1.000000

TRANSITION 5 3 1.000000

TRANSITION 6 3 1.000000

TRANSITION 7 8 1.000000 mumbai

TRANSITION 8 3 1.000000

FSG_END
```

3.4 Decoding With Sphinx

As mentioned above, you need the following files for decoding:

- Trained models
- Dictionary
- Filler dictionary
- Language model/ FSG
- Test data

To get a list of arguments in Sphinx3 decode, just type sphinx3_decode in the command line. You will get a list of arguments and their usage.

In general, the configuration file needs, at the very least, to contain the following arguments:

- -hmm followed by the acoustic model directory
- -dict followed by the pronunciation dictionary file
- -fdict followed by filler dictionary
- -lm followed by the language model, **OR**
- -fsg followed by the grammar file
- -ctl followed by the control file (fileids)
- -mode fsg (for FSG, for n-gram let it be the default mode, so need to specify this for n-gram)

However, in this case, there are several additional arguments that are necessary:

- -adcin yes (tells the recognizer that input is audio, not feature files)
- -adchdr 44 (tells the recognizer to skip the 44-byte RIFF header)
- -cepext .wav (tells the recognizer input files have the .wav extension)
- -cepdir wav (tells the recognizer input files are in the 'wav' directory)
- -hyp followed by the output transcription file

In case your training arguments are different from default arguments, then set it manually by providing the following parameters.

- -hypseg \ (followed by Recognition result file, with word segmentations and scores)
- -logfn \ (followed by Log file (default stdout/stderr))
- -verbose yes \ (Show input filenames)
- -lowerf 133.33334 \ (Lower edge of filters)
- -upperf 3500 \ (upper edge of filters)
- -nfft 256 \ (Size of FFT)
- -wlen 0.0256 \ (Hamming window length)
- -nfilt 33 \ (Number of filter banks)
- -ncep 13 \ (Number of cep coefficients)
- -samprate 8000 \ (Sampling rate)
- -dither yes (Add 1/2-bit noise)

The parameters like lowerf, upperf, nfft, wlen, nfilt,ncep should be same as used for training for good results.

Follow the example below:

In this case, dictionary, filler dictionary, fileIDs are kept in the folder **doc/.**

Trained models are kept in the folder **model_1/.** This folder should contain the following files:

mdef mixture_weights means variances transition_matrices.

wav_district is the directory where all the wav files are stored. FileIds will consist of names of all the wav files in this folder.

Create a directory LogOutFiles in which all the output files will be dumped.

```
1.  sphinx3_decode  \
2.  -hmm model_1   \
3.  -lm data.lm.DMP \
4.  -lowerf 133.33334 \
5.  -upperf 3500 \
6.  -nfft 256 \
7.  -wlen 0.0256 \
8.  -nfilt 33 \
9.  -ncep 13 \
10. -samprate 8000 \
11. -mode fwdflat \
12. -dither yes  \
13. -dict doc/marathiAgmark1500.dic  \
14. -fdict doc/marathiAgmark1500.filler \
15. -ctl doc/marathiAgmark1500_train.fileids \
16. -adcin yes \
17. -adchdr 44 \
18. -cepext .wav \
19. -cepdir wav_district \
20. -hyp  LogOutfiles/1500spkr_s1000_g16.out.txt  \
21. -hypseg LogOutfiles/1500spkr_s1000_g16.hypseg  \
22. -logfn LogOutfiles/1500spkr_s1000_g16.log.txt \
```

For FSG, make changes in line no. 3 and line no. 11 (-mode fsg). In case of any error while decoding, check the file LogOutfiles/**1500spkr_s1000_g16.log.txt.**

There are some additional parameters for decoding which one can be varied to improve accuracy such as:

Parameters	Default values
beam	1.00E-055
pbeam	1.00E-050
wbeam	1.00E-035
subvqbeam	3.00E-003
maxwpf	20
maxhistpf	100
maxhmmpf	2000
ci_pbeam	1.00E-080
ds	1.00E-080
lw	9.5
wip	0.7
Nlextree	3

Most of the parameters above are related to **pruning** in the Viterbi algorithm.

To understand decoding and pruning in sphinx in detail, go through the following presentation which explains the Viterbi decoding for single word as well as continuous speech recognition, Pruning, Lextree structure and consideration of Language weight while decoding.

Presentation

The link below is a document which shows some of the experiments conducted by my colleague by varying the parameters and observing the accuracy.

Tuning parameters

3.4.1 Output of the decoder

1. $database_out.txt

2. $database_hypseg.txt

- **$database_out.txt** will look like this:

kRusxii utpanna baajaara samitii (F13MH01A0011I302_silRemoved)
pikaacaa baajaara samitii (F13MH01A0011I303_silRemoved)
donashe baaviisa trepanna caarashe ekasasxtxa (F13MH01A0011I304_silRemoved)
gahuu (F13MH01A0011I305_silRemoved)
jvaarii (F13MH01A0011I306_silRemoved)
makaa (F13MH01A0011I307_silRemoved)

E.g. makaa is the output of file F13MH01A0011I307_silRemoved.wav

- **$database_hypseg.txt** will look like this:

F13MH01A0011I302_silRemoved S -1381719 T -2663753 A -2646045 L -17708 0 168656 808 <s> 6 -928199 -16508 kRusxii 37 -868399 -548 utpanna 77 -133383 -252 baajaara 111 -956937 -192 samitii 159 72217 -1016 </s> 165

The acoustic likelihoods for each word as seen in the decoder are scaled at each frame by the maximum score for that frame. The final (total) scaling factor is written out in the decoder HYPSEG output as the number following the letter "S".

T= Total score without the scaling factor. (The real score is the sum of S and T)

A = Total Acoustic Score

L = Total Language Model Score

T = A + L.

Every word or <s> is preceded by 3 numbers. E.g. the word **kRusxii** is preceded by 3 numbers 6, - 928199, -16508. The first no. is the start frame, second no. is the Acoustic score and the third no. is the Language model score **for that word**.

3.4.2 Script for decoding for Test data = Train data

With the above directory structure maintained for training, go to the /hmm directory, open the file **testing.csh** and make the following changes:

Set the no. of gaussians, senones, database,wip **same** as given for training.

Open the file **scripts_pl/initializeAndTest.csh** and set the parameters in sphinx3_decode same as were set in **etc/feat.params** file.

In the /hmm directory Run the file **testing.csh**

This will begin the decoding. You can check the details of decoding in the file **testLog."$database".txt** and "**$database".log.txt**

Explanation of script in brief:

1. **testing.csh**

- This script will set the no. of Gaussians, Senones and databasename in initializeAndTest.csh
- Runs the script scripts_pl/initializeAndTest.csh

2. **initializeAndTest.csh**

- This script accepts 4 arguments:

  - fileids

  - Transcription

  - Dictionary

  - Filler Dictionary

- Creates following directories in /hmm

  feats wav models models/lm models/hmm

- Prepares Language model and stores it in models/lm

  The transcription file for language model preparation should not contain the speaker ids, it should however contain the context cues : <s> and </s>. The file **removeDisfluencyFromTrans.pl** is used to create the transcription which does not contain speaker IDs and is given as input to **lmprepare.csh**. This script id used to create the language model language model which is stored in **models/lm/${database}.lm.DMP**

- Copies all the Acoustic models from model_parameters to models/hmm.

- As the features (i.e. MFC files) have already been created during training, we will use the same feature files for decoding.

- We run the sphinx3_decode by giving the required arguments. Check the script for detailed explanation of each argument.

3.5 To get Phone and Frame level segmentations

sphinx3_align tool used to get phone and frame based segmentation of utterances. A phone segmented file can be used to get **duration of phones** and **Segment Acoustic score** for phones in a wav file.

3.5.1 Phone based segmentations

A phone segmented file for the word Akola looks like this:

SFrm	EFrm	SegAScr	Phone
0	7	-63667	SIL
8	10	-26332	+bn+
11	23	-73564	a SIL k b
24	9	-14270	k a o i
30	38	-37818	o k l i
39	48	-58553	l o aa i
49	60	-40815	aa l SIL e
61	65	-74936	SIL
Total score:		-389955	

SFrm - Start Frame
Efrm - End Frame
SegAScr – Segment Acoustic Score.
Duration for a particular Triphone = Efrm – SFrm +1.

To get a list of arguments in Sphinx3 align, just type sphinx3_align in the command line. You will get a list of arguments and their usage.

To get phone based segmentations, first extract the features of the wav file using **sphinx_fe.** You need to provide the following arguments for sphinx_fe.

```
sphinx_fe \
-verbose yes \          (Show input filenames)
-samprate 8000 \        (Sampling rate)
-dither yes  \          (Add 1/2-bit noise )
-nfilt 33    \          (no. of filter banks)
-lowerf 133.33334  \    (lower edge of filter)
-upperf 3500 \          (higher edge of filter)
-nfft 256  \            (size of FFT)
-c docs/fileid.txt \    ( directory of FileID)
-di decoding/data  \    (Directory where the wav files are stored)
```

-mswav yes \	(Defines input format as Microsoft Wav (RIFF))
-ei wav \	(Input extension to be applied to all input files)
-do features \	(directory where the features will be stored)
-eo mfc	(Output extension to be applied to all output files)

After feature extraction, give the following arguments to sphinx3_align:

sphinx3_align \	
-hmm model_1/ \	(directory in which the model files are stored i.e. Mean, variance, mixture_weights, transition_matrices, mdef)
-dict marathiAgmark1500.dic \	(dictionary file)
-fdict 850spkr.filler \	(filler dictionary)
-ctl docs/fileid.txt\	(file-ids)
-insent phone.insent \	(Input transcript file corresponding to control file)
-cepdir features/ \	(directory in which the feature files are stored)
-phsegdir phonesegdir/\	(directory in which you want to store phone-segmented files)
-wdsegdir aligndir/\	(directory in which word-segmented files are stored)
-outsent phone.outsent\	(Output transcript file with exact pronunciation/transcription)

sphinx3_align is used to the align the phones provided the **transcriptions are provided.** Thus in the above case **phone.insent** is the transcription file which needs to be provided while a transcription file **phone.outsent** will be created which will have forced-aligned transcripts.

Thus you will have all the phone segmented files in the directory **phonesegdir/** in the above case.

3.5.2 Frame level segmentation

For frame level segmentation, copy the file **main_align.c** included along with the file in source/sphinx3/src/programs/main_align.c (sphinx3 is the folder where the sphinx decoder is extracted)
OR
To get the location of main_align.c, just search for the file using the following command:
find / -name “main_align.c”

Replace this file with the one included along with this manual.

Install the decoder again after replacing.

To get the frame level segmentation, in **sphinx3_align**, add an additional argument along with the above arguments:
-stsegdir ___ (Output directory for state segmentation files)

A state segmented file looks something like this:

```
FrameIndex 0 Phone SIL PhoneID 10 SenoneID 30 state 0 Ascr      -7488
FrameIndex 1 Phone SIL PhoneID 10 SenoneID 31 state 1 Ascr      -6935
FrameIndex 2 Phone SIL PhoneID 10 SenoneID 31 state 1 Ascr      -1056
FrameIndex 3 Phone SIL PhoneID 10 SenoneID 31 state 1 Ascr      -518
FrameIndex 4 Phone SIL PhoneID 10 SenoneID 31 state 1 Ascr      -518
FrameIndex 5 Phone SIL PhoneID 10 SenoneID 32 state 2 Ascr     -23861
FrameIndex 6 Phone a SIL k b PhoneID 83 SenoneID 265 state 0 Ascr      -7349
FrameIndex 7 Phone a SIL k b PhoneID 83 SenoneID 265 state 0 Ascr     -10775
FrameIndex 8 Phone a SIL k b PhoneID 83 SenoneID 265 state 0 Ascr      -3892
FrameIndex 9 Phone a SIL k b PhoneID 83 SenoneID 265 state 0 Ascr      -2106
```

where:
FrameIndex – Frame no.
Phone – triphone or basephone.
PhoneID – ID alloted to a particular triphone or basephone.
SenoneID – It is ID of the pruned decision tree representing a bunch of contexts(which are again phones) and is labeled by a number.
PhoneID is unique for all the triphones while SenoneID might be the same for different triphone (not sure though)
Ascr – Acoustic Score for that frame.

The acoustic score is calculated by the formula:

$$SegAScr(p) = \sum_{i=1}^{NF(p)} \left(\log_{1.003} \left(\prod_{j=1}^{N} \frac{1}{(2\pi)^{\frac{d}{2}} |\Sigma_j|^{\frac{1}{2}}} \exp\left(-\frac{1}{2} (x-\mu_j)^{\mathrm{T}} (\Sigma_j)^{-1} (x-\mu_j) \right) \right) + Trans_{i,i-1} \right)$$

In short, greater the acoustic score (-1>-4) better is the likelihood.

Both the phone and frames level segmentations can be used for confidence measure in speech recognition.

3.6 Word Lattice and n-best list generation

3.6.1 Word Lattice

During recognition the decoder maintains not just the single best hypothesis, but also a number of alternatives or candidates. For example, REED is a perfectly reasonable alternative to READ. The alternatives are useful in many ways: for instance, in N-best list generation. To facilitate such *post-processing*, the decoder can optionally produce a ***word lattice*** output for each input utterance. This output records all the candidate words recognized by the decoder at any point in time, and their main attributes such as time segmentation and acoustic likelihood scores.

To generate a word lattice, you need to just add an extra argument while decoding.
-outlatdir ___ (name of the directory in which you want to store the lattices)

For each wav file, there will be a lattice. The lattice will have an extension **.lat.gz**.
You can view the lattice by any editor (i.e. vi, gedit)
For details of the lattice file, follow the link below:
Lattice structure

3.6.2 Generating N-best lists from lattices

N-best lists can be generated from the lattices by using the binary **sphinx3_astar**.
To get a list of arguments in sphinx3_astar, just type sphinx3_astar in the command line. You will get a list of arguments and their usage.

Provide the following arguments for getting the basic n-best List
sphinx3_astar \

-mdef model_1/mdef \	(Model definition input file)
-lm model_1/data.lm.DMP \	(Word trigram language model input file)
-dict doc/marathiAgmark1500.dic \	(Main pronunciation dictionary)
-fdict doc/marathiAgmark1500.filler \	(filler dictionary)
-ctl doc/marathiAgmark1500_train.fileids \	(Control file listing utterances to be processed)
-nbestdir nbestList \	(n-best list directory)
-inlatdir lat \	(directory in which word lattices are stored)
-logfn logNbest \	(Log file (default stdout/stderr))

n-best list will be created for each utterance and will have an extension **.nbest.gz**
The initial lines of an n-best list will look something like this:

```
T -1074558 A -895652 L -18580 0 -44212 0 <s> 8 -676308 -18334 buladxhaanxaa(2) 48 -175132 -246 </s> 262
T -1106496 A -893171 L -22204 0 -44212 0 <s> 8 -673827 -22090 vaatxaanxaa(2) 48 -175132 -114 </s> 262
T -1137655 A -1005897 L -13618 0 -44212 0 <s> 8 -786553 -13502 bhuiimuuga 48 -175132 -116 </s> 262
T -1144440 A -935276 L -21766 0 -44212 0 <s> 8 -715932 -21718 batxaatxaa 48 -175132 -48 </s> 262
T -1213091 A -1034004 L -18600 0 -42695 0 <s> 7 -816177 -18156 punxe(2) 48 -175132 -444 </s> 262
```

Where:
T = Total score
A = Acoustic score
L = Language model score

So, in the above utterance, the first best hypothesis output is buladxhaanxaa(2), second best is vaatxaanxaa(2) and so on..

Chapter 4
Evaluation using Sclite

Download sclite from the following link:
Sclite download
Include the path of the sclite in ~/.bashrc or ~/.cshrc file.

4.1 Theory

Sclite is a tool for scoring and evaluating the output of speech recognition by comparing the hypothesized text (HYP i.e output of decoder) output by the speech recognizer to the correct, or reference (REF i.e transcription) text. After comparing REF to HYP, (a process called alignment), statistics are gathered during the scoring process and a variety of reports can be produced to summarize the performance of the recognition system.

4.2 Parameters and Formatting

Parameters:

- The '-h' option is a required argument which specifies the input hypothesis file.
- The '-r' option, a required argument, specifies the input reference file which the hypothesis file(s) are compared to.

For sclite to give correct results, you need the two files in the same format.
As such, the transcription file looks like this:

```
<s> kRusxii utpanna baajaara samitii </s>  (F13MH01A0011I302_silRemoved)
<s> +babble+ baajaara <sil> samitii </s>  (F13MH01A0011I303_silRemoved)
<s> donashe baaviisa <sil> trepanna(2) caarashe ekasasxtxa(2) <sil> </s>
(F13MH01A0011I304_silRemoved)
<s> gahuu </s>  (F13MH01A0011I305_silRemoved)
<s> jvaarii </s>  (F13MH01A0011I306_silRemoved)
```

Whereas the output of the decoder file i.e.[name].out file looks like this:

```
kRusxii utpanna baajaara samitii (F13MH01A0011I302_silRemoved)
pikaaqcaq baajaara samitii (F13MH01A0011I303_silRemoved)
donashe baaviisa trepanna caarashe ekasasxtxa (F13MH01A0011I304_silRemoved)
gahuu (F13MH01A0011I305_silRemoved)
jvaarii (F13MH01A0011I306_silRemoved)
```

As seen above, the output file does not contain silence markers, fillers and if the transcripts are forced aligned; it will also contain alternate pronunciations which are not present in the output file. So we basically need to remove silence markers, fillers and alternate pronunciations from the transcription file.
Recall we created the language model from the above transcription file. While creating the Language model, we get an output "$database".transcription. This file does not contain file-ids and alternate pronunciations, so we use this file to remove the above discrepancies. The file looks like this:

```
<s>   kRusxii utpanna baajaara samitii     </s>
<s>    baajaara samitii     </s>
<s>   donashe baaviisa trepanna caarashe ekasasxtxa      </s>
<s>   gahuu     </s>
<s>   jvaarii     </s>
```

To get the same format as of the output file, we only need to append fileIDS and remove silence markers.

For getting the final score and removing the discrepancies, you can directly use the script **scripts_pl/evaluateUsingSclite.csh**. Remember to run this file in the /hmm folder because all the file names are with reference to this folder.

This script accepts 3 arguments:

- database name
- Transcription file (this file should be the one you get from language model output. The name of the transcription file should be "$database".transcription)
- fileIDS file ("$database".fileids)

This script uses two more scripts:

1. scripts_pl/addSpkIdsToTrans.pl

This script will append fileids to the transcription file.

2. scripts_pl/removeDisfluencyFromTrans.pl

This script will remove silence markers from the transcription.

After the formatting is done, we run sclite,

```
sclite \
-i spu_id \     ( Define the reference file, and it's format)
-r [name] \     (transcription file)
-h [name] \     (output file)
-O [name] \     (output directory)
-o dtl          (Defines the output reports. Default: 'sum stdout')
```

The results will be stored in the output directory with an extension **.dtl.**

4.3 Output file

The file will look like this:

SENTENCE RECOGNITION PERFORMANCE

sentences	44330	(Total no. of sentences)
with errors	17.7% (7850)	(Sentences with errors)
with substitutions	11.4% (5050)	
with deletions	3.1% (1382)	
with insertions	6.4% (2857)	

WORD RECOGNITION PERFORMANCE

Percent Total Error = 16.0% (11247)

Percent Correct = 89.6% (62873)

Percent Substitution	=	7.8% (5483)
Percent Deletions	=	2.6% (1853)
Percent Insertions	=	5.6% (3911)
Percent Word Accuracy	=	84.0%

Ref. words	=	(70209)
Hyp. words	=	(72267)
Aligned words	=	(74120)

Thus you can get all the details of your result.

Sclite also accepts additional arguments for additional functionalities. For details of each of the argument, follow the link:
Sclite argument details