Massachusetts Institute of Technology
Department of Electrical Engineering & Computer Science

6.345 Automatic Speech Recognition
Spring, 2003

assigned:04/04/2003
due:04/16/2003

Assignment 8

Speech recognition using HMM-based systems:
CMU SPHINX-3 and SPHINX-4

---

1. Introduction

In this lab, you will learn to use a complete state-of-art HMM-based speech recognition system. An HMM-based system, like all other speech recognition systems, is a statistical pattern classifier. It associates an HMM with each sound unit. It first learns the parameters of these HMMs, and then uses the HMMs to find the most probable sequence of sound units for a given speech signal. The process of learning HMM parameters is called *training*. The process of using these to deduce the most probable sequence of units in a given signal is called *decoding*, or simply recognition.

Accordingly, a recognition system has two major components: a *trainer*, and a *decoder*. In this lab, you will learn to use the trainer of the SPHINX-3 system, designed at Carnegie Mellon University and written in the C programming language; and the decoder of the SPHINX-4 system, an opensource state-of-art system being written in the JAVA programming language. Since SPHINX-4 is a developing system, (to which you too can contribute!) , you will learn to use the decoder *in its most recent state at the time of your lab exercise*.

It is the aim of this lab to help you focus on several important issues involved in deploying an HMM-based ASR system, and to aid you in getting a feel for the current state of HMM-based recognition technology.

2. What will be given to you

Everything that you need from external sources for training and decoding will be provided to you. You will however have to train your own acoustic models in order to decode. This lab exercise will take you through the necessary steps.

Components provided for training

The SPHINX-3 trainer that will be provided to you consists of a set of C programs which have been compiled for your operating system (linux). You are only expected to use the pre-compiled system executables for training. The source code is provided for your information, for those of you who are curious about the software aspects of SPHINX or want to implement any small changes to the code based on your own ideas. Working with the source code is not expected of you in this lab.

The trainer learns the parameters of the models of the sound units using a set of sample speech signals. These signals comprise a **training database**. A training database comprised of 1600 speech signals will be provided to you. The trainer also needs to be told which sound units you want it to learn the parameters of, and at least one sequence in which they might have occured in every speech signal in your training database. This information is provided to the trainer through a file called the **transcript file**, in which the sequence of words and non-speech sounds are written exactly as they occurred in a speech signal, followed by a tag which can be used to associate this sequence with the corresponding speech signal. The trainer then looks into a **dictionary** which maps every word into at least one sequence of sound units, to derive a sequence of sound units associated with each signal. Thus, in addition to the speech signals, you will also be given a set of transcripts for the database (in a single file) and two dictionaries, one in which legitimate words in the language are mapped sequences of sound units (or sub-word units), and another in which non-speech sounds are mapped to corresponding non-speech or speech-like sound units. We will refer to the former as the **language dictionary** and to the latter as the **filler dictionary**.

The components provided to you for training will be:

1. The trainer executables
2. Acoustic signals for training (training data)
3. The corresponding transcript file
4. The language dictionary
5. The filler dictionary

Components provided for decoding

The SPHINX-4 decoder that will be provided to you consists of a set of JAVA programs which have been pre-compiled. JAVA executables are independent of the kind of operating system used by your machine. If you want to run your decodes on a different machine, with a different operating system, you can simply copy over the SPHINX-4 files to that machine, make sure it does have a JAVA compiler installed on it, and run the decodes.

Although a decoder consists of a set of programs like the trainer, decoding is usually performed through a single executable, to which you provide a set of 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**.

The components provided to you for decoding will be:

1. The language dictionary
2. The filler dictionary
3. The language model
4. The test data

In addition to these components, you will need to provide the **acoustic models** that you have trained as inputs 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 from the files which contain your trained acoustic models. With any given set of acoustic models, the corresponding model-index file must be used for decoding. If you would like to know more about the structure of the model-index file, you will find a description at the following URL: *http://www.cs.cmu.edu/~rsingh/sphinxman/fr4.html* under the link *Creating the CI model definition file*.

3. Setting up your system and getting familiar with it through a preliminary run

To set up your system, log into your machine with your userid. You will find the password on the note attached to this document. Once you are logged in, type the following command from the command line:

```
start_lab8.cmd
```

This will create a new directory called **lab8/** and will install in it all the files necessary for you to complete this lab exercise. Within the **lab8/** directory, you will find two directories called **SPHINX3/** and **sphinx4**. The SPHINX3 directory contains the following subdirectories:

- **s3trainer/** : The SPHINX-3 trainer source code and executables.
- **lists/**: The set of all user-supplied training files. You will be training acoustic models using a database called the Resource Management (RM) database. You can find more information on the Resource Management database at *http://www.ldc.upenn.edu/Catalog/LDC93S3B.html*, if you are curious. In the **lists/** directory you will find a list of training utterances (**train.wavlist**), and a set of test utterances (**test.wavlist**) from this database. If you wish to listen to any of the files listed in ***.wavlist**, give the command **waveform_play [waveform filename]**. In addition to ***.wavlist** you will also find a transcript file (**RM.1600.trans**) containing transcriptions corresponding to the files listed in **train.wavlist** , a language dictionary for training (**RM.dictionary**), a filler dictionary (**filler.dict**), and a phone list (**RM.phonelist**) containing all the sub-word units used in the language dictionary as well as an additional symbol **SIL** for silence.
- **c_scripts/** : The set of scripts that you will use to train continuous density HMMs for the RM database. This directory also contains a script that you will use for computing features (MFCCs) for your training and test utterances. Remember that the recognition system will be trained on these features and not on the waveforms directly.

The sphinx4 directory contains many subdirectories, of which the following specific ones are of use to you in this lab:

- **edu/** : This contains the Java source code for the SPHINX-4 decoder.

- **classes/** : This contains the compiled Java classes from the code in the directory **edu/**. These will be used by the Java Virtual Machine (JVM) installed on your computer to run the decoder.
- **tests/performance/resource-management/** : This is the directory in which you will do the decoding. It contains a file (**rm1.props**) that will point the decoder to the acoustic models you train, a file that will subsequently run the SPHINX-4 decoder (**Makefile**) through a simple *make* command.
- **tests/performance/resource-management/RM_models_and_otherfiles/**: This contains a list of 100 speech files you will decode (**s4.100.ctl**), a language model for decoding (**RM.2880.bigram.arpa**), a dictionary for decoding (**400.dict**), and a filler dictionary for decoding (**filler.dict**).In this lab, you are not expected to modify the language model file, but you *can* modify the dictionary used for decoding. Note that the dictionary that is used for decoding is different from that used for training. This is usually the case in speech recognition systems. The decoding dictionary is much more carefully crafted than the training dictionary, in order to minimize the possibility of confusion or to accomodate newer words, pronunciations, and other conditions.
- **tests/live/** : This contains files that you will need to do *livemode decoding*. In this mode of decoding, you can speak into your machine's microphones, and your speech is directly read by the decoder from your machine's audio device and recognized. The recognition hypotheses are displyed on your screen through a Java GUI. The files in this directory will be automatically set for you to perform livemode decoding. For your information, the file **rm1_live.props** points to your acoustic models, decoding dictionaries *etc.*, and the file **Makefile** runs the live decoding through a single command *make live* from you.

How to perfom a preliminary training run

The very first thing that you must do is to compute feature files from the speech data provided to you for training. To do this, enter the directory **SPHINX3/c_scripts/**, and type the following command on the command line

compute_mfcc_train.csh ../lists/train.wavlist

This script will compute, for each training utterance, a sequence of 13-dimensional Mel-frequency cepstral coefficients (MFCCs). This step takes approximately 10 minutes to complete on a fast network, but may take up to 40 minutes if the network is slow. The MFCCs will be placed automatically in a directory called **SPHINX3/feature_files/**. Note that the type of features vectors you use for training and recognition, outside of this lab, is not restricted to MFCCs. You could use any feature type, including those derived from phenomena concurrent with speech, such as video recordings. SPHINX-3 and SPHINX-4 can use features of any type or dimensionality. MFCCs are currently known to be the best single-feature parametrization for good speech recognition performance in HMM-based systems under most acoustic conditions.

Once the command prompt returns, you can begin to train the system. Before you do this, however, compute the features for the test data too. This is not really needed just now, but is simple enough to do by typing the following command from the command line:

compute_mfcc_test.csh ../lists/test.wavlist

This will put the MFCC files you will use for decoding in a specific location which will be accessed by the decoder automatically.

To train the system enter the directory **SPHINX3/c_scripts**. Within this directory are five directories numbered sequentially from 01 through 05. Enter each directory starting from **01** and run the script called **slave*.csh** within that directory. These scripts will launch jobs on the SLS batch system, and the jobs will take a few minutes each to run through. Before you run any script, note the directory contents of SPHINX3. After you run each **slave*.csh** note the contents again. Several new directories will have been created. These directories contain files which are being generated in the course of your training. At this point you need not know about the contents of these directories, though some of the directory names may be self explanatory and you may explore them if you are curious. After you launch the very first **slave*.csh**, watch the screen output till the command prompt returns. Only then enter the next directory (02) in the specified sequence and launch the **slave*.csh** in that directory. Repeat this process until you have run the **slave*.csh** in all five directories.

You would now have completed the training. You will find the acoustic models in a directory called **SPHINX3/model_parameters/RM.cd_continuous_8gau**. There are four files in this directory which constitute your acoustic models: **means**, **variances**, **transition_matrices** and **mixture_weights**. These are binary files. Their ascii versions ***.ascii**, are also in the same directory, and will be used by the SPHINX-4 decoder. In addition to these, you will find an index file for these models called **RM.1000.mdef** in the directory **SPHINX3/model_architecture/**. This file is used by the system to associate the appropriate set of HMM parameters with the HMM for each sound unit you are modeling. Later in this document we will explain some aspects of the training process in more detail.

How to perfom a preliminary decode

This is simple. Go to the directory **sphinx4/tests/performance/resource_management/** and from the command line, type:

make batchmode_bigram

This will display the recognition output on your screen. You can also simultaneously store it in some file **filename.log** by giving the following command instead:

make batchmode_bigram |& tee filename.log

As mentioned before, SPHINX-4 is an opensource system, still under development. It is actively improving in many ways as time passes, and at any time you can download the latest version of this system for your personal use. If you would like to do that, you can visit the webpage **http://cmusphinx.sourceforge.net** and follow the download instructions written there. You can also download a Java complier from **http://java.sun.com/j2se/1.4/**.

For your lab exercise, the SPHINX-4 decoder is set to work in *batchmode*, where it works off precomputed feature files stored on the disk. The decoder can run in other modes as well. In a *simulated livemode*, a decoder uses speech signals stored on a disk and decodes it in small blocks, computing features from the blocks, using them and then discarding them. In *livemode* decoding, speech captured actively by your machine's audio device is directly read by the decoder and recognized. A simple GUI demonstrating this also is available at **http://cmusphinx.sourceforge.net/cgi-bin/twiki/view/Sphinx4/HowToBuildAndRunSphinx4#_Live_Mode_Demo_**

The log file you just generated, **filename.log**, contains the decoded hypotheses and several other useful pieces of information, including recognition accuracy and error rates. The format of this file is explained in greater detail later in this document, but for now, note the lines which look like

```
Words: 832   Matches: 786    WER: 6.971%
Sentences: 100   Matches: 68   SentenceAcc: 68.000%
```

These lines (and many others) appear after every file that is decoded. The values against each label increase as more and more files (or sentences) are decoded. The label **Sentences: 100** implies that 100 sentences have been decoded so far. The Word Error Rate by this time is seen to be **WER: 6.971%**. Note this number. In your log file, the WER reported at the 100th sentence should be within +-1 of this number. If it isn't, report this. There may be some problem with your preliminary training.

4. Miscellaneous tools

Two useful tools are provided to you alongwith the trainer executables in the directory **SPHINX3/s3trainer/bin.linux/**. The tools are described below. Instructions on how to use these tools are in **toolname_instructions** files which you will find alongside the executables.

1. Phone and triphone frequency analysis tool: This is the executable **mk_mdef_gen** . You can use this to count the relative frequencies of occurrence of your basic sound units (phones and triphones) in the training database. Since HMMs are statistical models, what you are aiming for is to design your basic units such that they occur frequently enough for their models to be well estimated, while maintaining enough information to minimize confusions between words. This issue is explained in greater detail in Appendix 1.
2. Tool for viewing the MFCC files: This is the executable **cepview**. You can view the cepstra in any feature file by following the instructions this tool's instruction file.

5. What is expected in this lab

The primary aim of this lab is to make you reasonably familiar with the way typical HMM-based systems work and respond to various settings. You are expected to "chart" the system performance in some manner of your own choosing, with respect to some aspect of the system that affects recognition performance. Some important aspects that you can study are described below. Choose one of them, or at most two, and experiment with them, think about them, study them in whatever way you think is most useful. Then simply write down, in a page or two, how you studied the system and what you found. For example you could make a graph, or a table, and try to explain the trends seen in it.

Remember that speech recognition is a complex engineering problem and you are not expected to be able to fully relate the aspects you study to much of the rest of the system in this single lab session. Just do your reasonable best.

6. How to train, and key training issues

You are now ready to begin your lab exercise. For every training and decoding run, you will need to first give it a name. We will refer to the experiment name of your choice by **[experimentname]**. For example, you can name the first experiment *exp1*, and the second experiment *exp2*, etc. Your choice of **[experimentname]** will be appended automatically to all relevant file created under that experiment for easy identification. All directories and files needed for this experiment will reside in a directory named **lab8/[experimentname]**. To begin a new experiment enter the directory **lab8** and give the

command

SPHINX3/c_scripts/create_newexpt.csh [experimentname]

This will create a setup similar to the preliminary training and test setup in a directory called **experimentname/** in your base directory After this you must work entirely within this **[experimentname]** directory.

Your lab exercise begins with training the system using the MFCC feature files that you have already computed during your preliminary run. file. However, when you train this time, you can take certain decisions based on what you have learned so far in your coursework and the information that is provided to you in this document. The decisions that you take will affect the quality of the models that you train, and thereby the recognition performance of the system.

The following are some important aspects you can look at with respect to training:

1. Decide what sound units you are going to allow the system to train. You can vary the quality of your acoustic models by choosing different sound units. To do this, look at the training dictionary **[experimentname]/lists/RM.dictionary** and the filler dictionary **[experimentname]/lists/filler.dict**, and note the sound units in these. A list of all sound units in these dictionaries is also written in the file **[experimentname]/lists/RM.phonelist** . There are many ways to approach the issue of good sound units:
    - Study the dictionaries to see if the sound units are consistently used and minimally confusible. Look at **Appendix 1** for an explanation. If they are not, change the usage as appropriate.
    - Given some set of sound units, you can check whether these units, and the triphones they can form (for which you will be building models ultimately), are well represented in the training data. It is important that the sound units being modeled be well represented in the training data in order to estimate the statistical parameters of their HMMs reliably. To study their occurrence frequencies in the data, you may use the tool **mk_mdef_gen** . Instructions for the use of this tool are given in its corresponding instruction file.

      You can change the occurence statistics of triphones easily by changing by merging or splitting existing sound units in the dictionaries. By merging of sounds units we mean the clustering of two different sound units into a single entity. For example, you may want to model the sounds "Z" and "S" as a single unit (instead of maintaining them as separate units). To merge these units, which are represented by the symbols Z and S in the language dictionary given, simply replace all instances of Z and S in the dictionary by a common symbol (which could be Z_S, or an entirely new symbol). By splitting of sound units we mean the introduction of multiple new sound units in place of a single sound unit. This is the inverse process of merging. For example, if you found a language dictionary where all instances of the sounds Z and S were represented by the same symbol, you might want to replace this symbol by Z for some words and S for others. Sound units can also be restructured by grouping specific sequences of sound into a single sound. For example, you could change all instances of the sequence "IX D" into a single sound IX_D. This would introduce a new symbol in the dictionary while maintaining all previously existing ones. The number of sound units is effectively increased by one in this case. There are other techniques used for redefining sound units for a given task. If you can think of any other way of redefining dictionaries or sound units that you can properly justify, we encourage you to try it.

   Once you re-design your units, alter the file **RM.phonelist** accordingly. Make sure you do not have spurious empty spaces or lines in this file. *Note again that you may bypass this design procedure altogether and use the phonelist and dictionaries as they have been provided to you*. Note that since you have changed the sound units in your training dictionary, you *must* change them in the decoding dictionary too because the system can only recognize the sound units it has learned. So make similar changes in the decoding dictionary **[experimentnAME]/sphinx4/tests/performance/resource_management/RM_models_and_otherfiles/400.dict**

   Once you have fixed your dictionaries and the phonelist file, edit the file **variables.def** in **[experimentname]/c_scripts/** to enter the following training parameters:

    - **set dictionary =** your training dictionary with full path (do not change if you have decided not to change the dictionary)
    - **set fillerdict =** your filler dictionary with full path (do not change if you have decided not to change the dictionary)
    - **set phonefile =** your phonelist with full path (do not change if you have decided not to change the dictionary)

2. Another aspect you can study is the effect of variation in the number of parameters the system must train from the given training data. This is easily managed by resetting the following flags in **variables.def**
    - **set statesperhmm =** This is set to either 3 or 5 in standard systems. Try setting it to other values (remaining in the range 3-9 is recommended). The number of states in an HMM is related to the time-varying characteristics of the sound units. Sound units which are highly time-varying need more states to represent them. The time-varying nature of the sounds is also partly captured by the "skipstate" variable that is described below.

- **set skipstate =** set this to "no" or "yes" without the double quotes. This variable controls the topology of your HMMs. When set to "yes", it allows the HMMs to skip states. However, note that the HMM topology used in this system is a strict left-to-right Bakis topology. If you set this variable to "no", any given state can only transition to the next state. In all cases, self transitions are allowed. See the figures in **Appendix 2** for further reference. You will find the HMM toplogy file in the directory called **model_architecture/** in your current base directory (**experimentname**).
- **set gaussiansperstate =** set this to any number from 4 to 8. Going beyond 8 is not advised because of the small training data set you have been provided with. The distribution of each state of each HMM is modeled by a mixture of Gaussians. This variable determines the number of Gaussians in this mixture. The number of HMM parameters to be estimated increases as the number of Gaussians in the mixture increases. Therefore, increasing the value of this variable may result in less data being available to estimate the parameters of every Gaussian. However, increasing its value also results in finer models, which can lead to better recognition. Therefore, it is necessary at this point to think judiciously about the value of this variable, keeping both these issues in mind. Remember that it is possible to overcome data insufficiency problems by sharing the Gaussian mixtures amongst many HMM states. When multiple HMM states share the same Gaussian mixture, they are said to be shared or tied. These shared states are called tied states (also referred to as senones). The number of mixtures you train will ultimately be exactly equal to the number of tied states you specify, which in turn can be controlled by the "n_tied_states" parameter described below.
- **set n_tied_states =** set this number to any value between 500 and 2500. This variable allows you to specify the total number of shared state distributions in your final set of trained HMMs (your acoustic models). States are shared to overcome problems of data insufficiency for any state of any HMM. The sharing is done in such a way as to preserve the "individuality" of each HMM, in that only the states with the most similar distributions are "tied". The **n_tied_states** parameter controls the degree of tying. If it is small, a larger number of possibly dissimilar states may be tied, causing reduction in recognition performance. On the other hand, if this parameter is too large, there may be insufficient data to learn the parameters of the Gaussian mixtures for all tied states. (An explanation of state tying is provided in **Appendix 3**). If you are curious, you can see which states the system has tied for you by looking at the ascii file **SPHINX3/model_architecture/[experimentname].n_tied_states.mdef** . and comparing it with the file **SPHINX3/model_architecture/[experimentname].untied.mdef**. These files list the phones and triphones for which you are training models, and assign numerical identifiers to each state of their HMMs. When you change the **n_tied_states**, though, before you decode make sure that the file **[experimentname]/sphinx4/tests/performance/resource_management/rm1.props** points to the right mdef file and acoustic model files for your experiment.

3. Finally, you can control the degree of fitting (or overfitting!) of the estimated models by controlling the number of passes the Baum-Welch algorithm makes over the training data to train the models. This aspect can be controlled by setting the following flag values in **variables.def**:
   - **set convergence_ratio =** set this to a number between 0.1 to 0.001. This number is the ratio of the difference in likelihood between the current and the previous iteration of Baum-Welch to the total likelihood in the previous iteration. Note here that the rate of convergence is dependent on several factors such as initialization, the total number of parameters being estimated, the total amount of training data, and the inherent variability in the characteristics of the training data. The more iterations of Baum-Welch you run, the better you will learn the distributions of your data. However, the minor changes that are obtained at higher iterations of the Baum-Welch algorithm may not affect the performance of the system. Keeping this in mind, decide on how many iterations you want your Baum-Welch training to run in each stage. This is a subjective decision which has to be made based on the first convergence ratio which you will find written at the end of the logfile for the second iteration of your Baum-Welch training (**SPHINX3/logdir/0*/[experimentname].*.2.norm.log**. Usually, 5-15 iterations are enough, depending on the amount of data you have. Do not train beyond 15 iterations.
   - **set maxiter =** set this to an integer number between 5 to 15. This limits the number of iterations of Baum-Welch to the value of **maxiter**.

Once you have made all the changes desired, you must train a new set of models. You can accomplish this by re-running all the **slave*.csh** scripts from the directories [experimentname]/c_scripts/01* through [experimentname]/c_scripts/05*.

7. How to decode, and key decoding issues

To decode, simply go to the directory **[experimentname]/sphinx4/tests/performance/resource_management/**, make sure you have made any necessary changes to the decoding dictionary in **RM_models_and_otherfiles/400.dict** to match the changes made in your training dictionary, make sure the acoustic model files pointed to in the file **rm1.props** are correct, and then, from the command line, type:

make batchmode_bigram |& tee filename.log

The most important issues about the efficient use of a decoder are those relating to its decoding speed, memory usage and

power consumption. Since you are working on desktop computers in this lab, power consumption is not a big issue. The performance of a decoder is often dependent on the tradeoff between settings made to minimize resource usage and maximize speed, those necessary to accomodate a particular size and complexity of the acoustic models, language models and dictionary, and the recognition accuracy. Many of these tradeoffs must be decided at the training stage. Further changes in settings are done in the decoder. In this lab we will consider some very basic decoder settings that affect logistic requirements, speed, and recognition performance. Note that while in many cases performance is what matters the most, today speech recognition systems must be deployed in many scenarios which place stringent demands on memory and speed as well. These have become very important issues in recent times. So before you use any speech recognition system, it is a good idea to try to take stock of your available resources, so that you can use them prudently.

SPHINX-4 is being written in JAVA. This is an object oriented language like C++, but runs through an intermediate object-file interpreter called the Java Virtual Machine (JVM), which converts your compiled object files into an internal format that will run on your specific operating system. So far as you (the user) are concerned, code that you have compiled in JAVA on any machine, runs on any other with a different operating system, provided it has a JVM installed on it. JAVA has an additional feature - it does not use pointers, and you do not need to explicitly free any memory in your code. JAVA has a Garbage Collector (GC) that periodically senses which parts of the allocated memory are not being used, and frees them. It is logical to expect that working through an intermediate interpreter, and perpetual GC checks would slow down any JAVA based application. However, this has largely become a non-factor due to many reasons.

There are some things you can do yourself to manage memory efficiently. You can specify what are called *heapsizes* that you want the decoder to use through the flags **GC_FLAGS** in the **Makefile**. The value of **-ms** sets the *starting heapsize* and the value of **-mx** sets the *maximum heapsize*. A *heapsize* in JAVA relates to the amount of memory that JVM can allocate to your programs. It affects the action of the GC. It might help to find out a little more about the CPU and memory resources on your machine to get an idea of where you stand *vis-a-vis* these flag settings. This is however not intended to be a part of your lab work. Memory and speed can be managed in many other ways as well. SPHINX-4 has not yet been optimized for memory and speed, and you can contribute to the system in these areas after the lab, if you are interested.

Active lists and Beamwidths There is another way in which both memory usage *and* speed can be controlled (and this is applicable to decoders written in any language): through the proper specification of the size of what are called *Active lists*. At any time instant during search, an active list is comprised of those Gaussian densities which must be explicitly computed by the decoder, given the current data vector. This is a subset of all Gaussians present in the acoustic models being used by the decoder, and have been reached by current paths in the trellis. You can control the size of the active list through the flags

```
edu.cmu.sphinx.search.ActiveList.absoluteBeamWidth
edu.cmu.sphinx.search.ActiveList.relativeBeamWidth
```

in the file **rm1.props**. The absolute beam width is conventionally a fixed number, and defines the maximum size of the active list. The relative beam width varies from time instant to time instant, and is defined with reference to the maximum scoring node in the decoder's trellis at the given instant. In sphinx4, it is a threshold score which is some percent of the current maximum score. Nodes which have a score lower than this threshold are not allowed propagate further. Active lists are described at **http://cmusphinx.sourceforge.net/cgi-bin/twiki/view/Sphinx4/HeapActiveList** You can find a description of beamwidths at **http://cmusphinx.sourceforge.net/cgi-bin/twiki/view/Sphinx4/Sphinx4Properties**.

Language weight The language-weight is an important parameter in any decoder. In SPHINX-4, the flag for this is

```
edu.cmu.sphinx.search.BreadthFirstSearchManager.languageWeight
```

in the **rm1.props**. A value between 6 and 13 is standard, and by default the language weight is not applied. The language model and the language weight are described in **Appendix 4**. Remember that the language weight decides how much relative importance you will give to the actual acoustic probabilities of the words in the hypothesis. A low language weight gives more leeway for words with high acoustic probabilities to be hypothesized, at the risk of hypothesizing spurious words. You may decode several times with different language weights, without re-training the acoustic models, to decide what is best for you.

Word insertion penalty Insertion penalty is an important heuristic parameter in any dynamic programming algorithm. The decoder is no exception. The flag for this is

```
edu.cmu.sphinx.search.Linguist.wordInsertionProbability
```

in the **rm1.props**. It is the number that decides how much penalty to apply to a new word during the search. If new words are not penalized, the decoder would tend to hypothesize the smallest words possible, since every new word inserted leads to an additional increase in the score of any path, as a result of the inclusion of the inserted word's language probability from the language model.

Using other acoustic models Finally, note that you can use any of the intermediate models that you generated during training, for decoding. Make sure to use the appropriate model-index file with each set of models. The model-index file is the same for the cd models, but is different for the cd-untied and ci models. You will find all acoustic models in

SPHINX3/model_parameters/, and all model-index files in **SPHINX3/model_architecture/**. Note that when training data are sparse, increasing the number of Gaussians/state increases the number of parameters you must train, and results in bad models when this number is too high, since the data available for training may be insufficient. to use other acoustic models, simply modify **rm1.props** to point to them.

Performance evaluation The decode logfile, **filename.log**, contains all the information you need to evaluate the performance of the decoder for this lab. For each feature file **foo*.mfc** decoded, you will find an entry which generally looks like:

```
Decoding: foo1.mfc

REF:       display posits for the hooked track with chart switches set to their default values
HYP:       display posits for the hooked track with chart switches set to their default values
ALIGN_REF: display posits for the hooked track with chart switches set to their default values
ALIGN_HYP: display posits for the hooked track with chart switches set to their default values

   Accuracy: 100.000%    Errors: 0  (Sub: 0  Ins: 0  Del: 0)
   Words: 14   Matches: 14    WER: 0.000%
   Sentences: 1   Matches: 1   SentenceAcc: 100.000%
   HypScore: -1.791845E7
   This  Time Audio: 4.50s  Proc: 17.55s  Speed: 3.90 X real time
   Total Time Audio: 4.50s  Proc: 17.55s  Speed: 3.90 X real time
   Mem Total: 437.55 Mb Free: 154.72 Mb Used: 282.83 Mb
 ========= statistics for batch=======
totalStates 242134.0
totalArcs 1065032.0
actualArcs 937912.0
totalTokensScored 1419847.0
curTokensScored 1419847.0
tokensCreated 3381309.0
--------------

Decoding: foo2.mfc

REF:       how many ships were in galveston may third
HYP:       how many ships were in galveston dates there
ALIGN_REF: how many ships were in galveston MAY   THIRD
ALIGN_HYP: how many ships were in galveston DATES THERE

   Accuracy: 90.909%    Errors: 2  (Sub: 2  Ins: 0  Del: 0)
   Words: 22   Matches: 20    WER: 9.091%
   Sentences: 2   Matches: 1   SentenceAcc: 50.000%
   HypScore: -1616598.2  ActScore: NONE
   This  Time Audio: 2.70s  Proc: 9.46s  Speed: 3.51 X real time
   Total Time Audio: 7.19s  Proc: 27.01s  Speed: 3.76 X real time
   Mem Total: 437.55 Mb Free: 168.25 Mb Used: 269.30 Mb
 ========= statistics for batch=======
totalStates 242134.0
totalArcs 1065032.0
actualArcs 937912.0
totalTokensScored 2267396.0
curTokensScored 847549.0
tokensCreated 5393013.0
--------------
```

This output corresponds to recognition of two feature files, foo1.mfc and foo2.mfc in sequence. In the example above, the lines begining with REF: and HYP: show the correct and hypothesized word sequences for the file respectively. the lines begining with ALGIN_REF: and ALIGN_HYP: show the result after the reference and decoded word sequences were aligned through a dynamic time warping (DTW) algorithm. The capitalized words were wrong in some sense.

In this example, foo1.mfc was correctly decoded. The entry **Accuracy** indicates the percentage of words in the test set that were correctly recognized so far (100% by the time foo1.mfc was decoded). Note again that this is a cumulative metric, and refers to all files decoded so far. However, this is not a sufficient metric - it is possible to correctly hypothesize all the words in the test utterances merely by hypothesizing a large number of words for each word in the test set. The spurious words, called insertions, must also be penalized when measuring the performance of the system. The entry **WER%** indicates the number of hypothesized words that were erroneous as a percentage of the actual number of words in the test set. This includes both words that were wrongly hypothesized (or deleted) and words that were spuriously inserted. Since the recognizer can, in principle, hypothesize many more spurious words than there are words in the test set, the percentage of errors can actually be greater than 100. All metric reported in the log file are cumulative metrics. In the example above, after foo2.mfc was decoded, of the 22 words in the reference test *so far*, transcripts 20 words (90.909%) were correctly hypothesized. In the process the recognizer hypothesized 2 spurious words (these include insertions, deletions and substitutions, which are reported under the entries **Ins:**, **Del:** and **Sub:** respectively.).

The line

```
Sentences: 2   Matches: 1   SentenceAcc: 50.000%
```

indicates that 2 sentences were decoded so far, of which 1 was completely correctly recognized, giving a sentence

accuracy of 50%. Sentence accuracy is a very important measure in tasks were absolutely correct recognition is necessary, such as recognizing an identity or a credit card number, or in machines with critical-outcome responses. The lines

```
This  Time Audio: 2.70s  Proc: 9.46s  Speed: 3.51 X real time
Total Time Audio: 7.19s  Proc: 27.01s  Speed: 3.76 X real time
Mem Total: 437.55 Mb Free: 168.25 Mb Used: 269.30 Mb
```

relate to speed of decoding and memory usage. From top to bottom, reading left to right, an interpretation of the above lines would be: "the duration of the current sentence was 2.7 seconds, processing time taken for it was 9.46 seconds, it was decoded in 3.51 times real-time; the total duration of utterances decoded so far was 7.19 seconds, total processing time was 27.01 seconds, average speed was 3.76 times real-time; Total memory that is being used by the JVM is 437.55 Mb, of which 168.25 Mb is still free and 269.30 Mb is being used". Note that JVM is memory-greedy. It will use as much memory as you allocate to it. The trick is to allocate just enough for your task, though the -mx and -ms flags described above.

Studying the errors made in the recognition process (the capitalized words in the aligned reference and hypothesized word sequences) often gives an idea of what might be done to improve recognition. For example, if the word "FOR" has been hypothesized as the word "FOUR" almost all the time, perhaps you need to correct the pronunciation for the word FOR in your decoding dictionary and include a pronunciation that maps the word FOR to the units used in the mapping of the word FOUR. Once you make these corrections, you must re-decode.

Livemode decoding This is merely meant to demonstrate how livemode decoding can be done with the models you trained. Go to the directory **sphinx4/tests/live**, make sure the file **rm1_live.props** points to the correct acoustic models, and give the command **make live** from the command line. A GUI will come up on your screen. Choose the "your experiment" button, and wait till it is ready for you. Then press the speak button and speak the sentence it prompts you to speak into the microphone. Press the speak button again after you are done speaking. The recognition output appears in the GUI hypothesis window. The recognition performance will probably be very bad. This is because a livemode decoder must be optimized in many ways with settings that are very specific to a given machine and task. That is not the focus of this lab. Note also that if the machine you are running this on is physically not the same as the machine you are speaking into, the livemode decoder will not work. In that case you will have to copy the setup over to your local machine and then run it. You may skip this part if you are not interested.

APPENDIX 1

If your transcript file has the following entries:

THIS CAR THAT CAT (file1)
CAT THAT RAT (file2)
THESE STARS (file3)

and your language dictionary has the following entries for these words:

CAT	K	AE	T		
CAR	K	AA	R		
RAT	R	AE	T		
STARS	S	T	AA	R	S
THIS	DH	I	S		
THAT	DH	AE	T		
THESE	DH	IY	Z		

then the occurrence frequencies for each of the phones are as follows (in a real scenario where you are training triphone models, you will have to count the triphones too):

K	3	S	3
AE	5	IY	1
T	6	I	1
AA	2	DH	4
R	3	Z	1

Since there are only single instances of the sound units IY and I, and they represent very similar sounds, we can merge them into a single unit that we will represent by I_IY. We can also think of merging the sound units S and Z which represent very similar sounds, since there is only one instance of the unit Z. However, if we merge I and IY, and we also merge S and Z, the words THESE and THIS will not be distinguishable. They will have the same pronunciation as you can see in the following dictionary with merged units:

CAT	K	AE	T		
CAR	K	AA	R		
RAT	R	AE	T		
STARS	S_Z	T	AA	R	S_Z
THIS	DH	I_IY	S_Z		
THAT	DH	AE	T		
THESE	DH	I_IY	S_Z		

If it is important in your task to be able to distinguish between THIS and THESE, at least one of these two merges should not be performed.

APPENDIX 3

Consider the following sentence.

CAT THESE RAT THAT

Using the first dictionary given in Appendix 1, this sentence can be expanded to the following sequence of sound units:

<sil> K AE T DH IY Z R AE T DH AE T <sil>

Silences (denoted as <sil> have been appended to the beginning and the end of the sequence to indicate that the sentence is preceded and followed by silence. This sequence of sound units has the following sequence of triphones

K(sil,AE) AE(K,T) T(AE,DH) DH(T,IY) IY(DH,Z) Z(IY,R) R(Z,AE) AE(R,T) T(AE,DH) DH(T,AE) AE(DH,T) T(AE,sil)

where A(B,C) represents an instance of the sound A when the preceding sound is B and the following sound is C. If each of these triphones were to be modeled by a separate HMM, the system would need 33 unique states, which we number as follows:

K(sil,AE)	0	1	2
AE(K,T)	3	4	5
T(AE,DH)	6	7	8
DH(T,IY)	9	10	11
IY(DH,Z)	12	13	14
Z(IY,R)	15	16	17
R(Z,AE)	18	19	20
AE(R,T)	21	22	23
DH(T,AE)	24	25	26
AE(DH,T)	27	28	29
T(AE,sil)	30	31	32

Here the numbers following any triphone represent the global indices of the HMM states for that triphone. We note here that except for the triphone T(AE,DH), all other triphones occur only once in the utterance. Thus, if we were to model all triphones independently, all 33 HMM states must be trained. We note here that when DH is preceded by the phone T, the realization of the initial portion of DH would be very similar, irrespective of the phone following DH. Thus, the initial state of the triphones DH(T,IY) and DH(T,AE) can be tied. Using similar logic, the final states of AE(DH,T) and AE(R,T) can be tied. Other such pairs also occur in this example. Tying states using this logic would change the above table to:

K(sil,AE)	0	1	2
AE(K,T)	3	4	5
T(AE,DH)	6	7	8
DH(T,IY)	9	10	11
IY(DH,Z)	12	13	14
Z(IY,R)	15	16	17
R(Z,AE)	18	19	20
AE(R,T)	21	22	5
DH(T,AE)	9	23	24
AE(DH,T)	25	26	5
T(AE,sil)	6	27	28

This reduces the total number of HMM states for which distributions must be learned, to 29. But further reductions can be achieved. We might note that the initial portion of realizations of the phone AE when the preceding phone is R is somewhat similar to the initial portions of the same phone when the preceding phone is DH (due to, say, spectral considerations). We could therefore tie the first states of the triphones AE(DH,T) and AE(R,T). Using similar logic other states may be tied to change the above table to:

K(sil,AE)	0	1	2

AE(K,T)	3	4	5
T(AE,DH)	6	7	8
DH(T,IY)	9	10	11
IY(DH,Z)	12	13	14
Z(IY,R)	15	16	17
R(Z,AE)	18	19	20
AE(R,T)	21	22	5
DH(T,AE)	9	23	11
AE(DH,T)	21	24	5
T(AE,sil)	6	25	26

We now have only 27 HMM states, instead of the 33 we began with. In larger data sets with many more triphones, the reduction in the total number of triphones can be very dramatic. The state tying can reduce the total number of HMM states by one or two orders of magnitude.

In the examples above, state-tying has been performed based purely on acoustic-phonetic criteria. However, in a typical HMM-based recognition system such as SPHINX, state tying is performed not based on acoustic-phonetic rules, but on other data driven and statistical criteria. These methods are known to result in much better recognition performance.

APPENDIX 4

Language Model: Speech recognition systems treat the recognition process as one of maximum a-posteriori estimation, where the most likely sequence of words is estimated, given the sequence of feature vectors for the speech signal. Mathematically, this can be represented as

$$Word1\ Word2\ Word3\ ... = \underset{Wd1\ Wd2\ ...}{argmax}\{P(feature\ vectors|Wd1\ Wd2\ ...)\ P(Wd1\ Wd2\ ...)\} \quad (1)$$

where Word1.Word2... is the recognized sequence of words and Wd1.Wd2... is any sequence of words. The argument on the right hand side of Equation 1 has two components: the probability of the feature vectors, given a sequence of words *P(feature vectors| Wd1 Wd2 ...)*, and the probability of the sequence of words itself, *P(Wd1 Wd2 ...)* . The first component is provided by the HMMs. The second component, also called the language component, is provided by a language model.

The most commonly used language models are N-gram language models. These models assume that the probability of any word in a sequence of words depends only on the previous N words in the sequence. Thus, a 2-gram or bigram language model would compute *P(Wd1 Wd2 ...)* as

$$P(Wd1\ Wd2\ Wd3\ Wd4\ ...) = P(Wd1)P(Wd2|Wd1)P(Wd3|Wd2)P(Wd4|Wd3)... \quad (2)$$

Similarly, a 3-gram or trigram model would compute it as

$$P(Wd1\ Wd2\ Wd3\ ...) = P(Wd1)P(Wd2|Wd1)P(Wd3|Wd2,Wd1)P(Wd4|Wd3,Wd2)\ ... \quad (3)$$

The language model provided for this lab is a bigram language model.

Language Weight: Although strict maximum a posteriori estimation would follow Equation (1), in practice the language probability is raised to an exponent for recognition. Although there is no clear statistical justification for this, it is frequently explained as "balancing" of language and acoustic probability components during recognition and is known to be very important for good recognition. The recognition equation thus becomes

$$Word1\ Word2\ Word3\ ... = \underset{Wd1\ Wd2\ ...}{argmax}\{P(feature\ vectors|Wd1\ Wd2\ ...)P(Wd1\ Wd2\ ...)^{alpha}\} \quad (4)$$

Here *alpha* is the language weight. Optimal values of *alpha* typically lie between 6 and 11.