# ASR Suite for Dummies (pun intended)

## Overview

This document is a software manual, describing the tools used to yoke the simulated hearing aid to a model of the patients impaired hearing (the dummy), and how the dummy can be subsequently connected to an automatic speech recogniser. The idea is that if an automatic speech recogniser is trained using feature vectors derived from a model of normal hearing, then the speech recogniser will perform optimally when tested with feature vectors derived from the same normal-hearing dummy. If the recogniser is tested using feature vectors derived from an impaired dummy, then one would expect the recognition score to drop significantly. The representation of the acoustic stimulus can be modified by a hearing aid algorithm before being presented to the impaired dummy. The potential of the hearing aid signal processing to alleviate some of the reduction in speech intelligibility caused by the impairment may then be assessed by comparing the aided and unaided recognition scores. This software framework should eventually enable the tailoring of hearing aid parameters to individuals in their absence.

Most of this tutorial is spent explaining the test functions that use the recogniser classes provided. This is to help the user understand the way that I construct recognition experiments, so that modification or a complete redesign of the software is made easier. Due to the different paradigms, and the different stimuli used in speech recognition experiments, a generic MATLAB software suite would need to be very complex. In the author’s experience, Matlab software suites that attempt to do too much end up imposing methodologies on the user that may eventually turn out to be restrictive. The code given here will definitely need heavy modification to work on anything outside the digit triplet recognition task described. Therefore, this document is not a “HOWTO”, but a “HOWI”, and should provide an avenue to Matlab script writers who are interested in using speech recognition to evaluate auditory-related signal processing.

The software suite exists in a “userPrograms” folder within the main Matlab model of the Auditory Periphery (MAP) folder. This tutorial document assumes that the reader is already familiar and comfortable with using MAP. The full documentation for MAP can be found at:

<http://www.essex.ac.uk/psychology/department/research/hearing_models.html>

## Files included in the suite

The evaluation of the hearing aid is done with the assistance of 3 main classes.

### cEssexAid.m

This is the file that contains the class definition of the Essex Aid wrapper. The Essex Aid is a novel hearing aid algorithm developed at the University of Essex. Once the user is familiar with the software framework, it should be relatively straightforward to swap out the hearing aid algorithm so that any hearing aid algorithm can be tested.

### cJob.m

This is the class definition that is the real workhorse of the operation. It provides many utility functions to assist the user in creating feature sets for use in training and testing the speech recogniser. This class also contains code for scheduling, so that many sounds can be processed in parallel on one or many machines should you have the resources available to do so.

### cHMM.m

This is a wrapper class for the Hidden Markov Tookkit (HTK) <http://htk.eng.cam.ac.uk/>. This class contains helper functions that provide the recogniser with the necessary information for training a HMM and scoring the results produced.

\*\*\*

Other functions within the parent directory include

### worker.m

This is an autonomous little function that takes a path to a folder containing a list of jobs as an input argument. A job is defined here as a list of wav files that need to each be converted into a feature vector. It searches the directory for jobs to do and works until all of the jobs are complete.

### Exp\_Tutorial\_X.m

Files prefixed with “Exp\_” are the files that users will edit most frequently. They are at the top of the function call stack and initiate all of the tasks required to run a recognition experiment.

### MAPwrap.m

This is a very simple wrapper for MAP, making it easy to call from the classes within the suite. MAP also uses a lot of global variables, so the wrapper also provides a barrier to prevent potential conflicts between variable names.

\*\*\*

There are also two folders in the parent directory

### /def

This contains the definition files required by HTK, such as grammar rules, dictionaries and hmm prototypes. Full descriptions of these files are beyond the scope of this document, but detailed information can be found here <http://htk.eng.cam.ac.uk/>

### /ASRfiles

This is a folder containing some additional utility functions used by the main classes.

\*\*\*

## Tutorial

One of the best ways to learn is by doing. The tutorial is split into two sections that represent a typical workflow. The first part of the tutorial shows the reader how to make a new HMM and then test it. The second part shows the reader how to use an existing HMM to test new features. The new data might have come from a different dummy and/or hearing aid processing.

Before any work with the recogniser can be accomplished, the speech material and recogniser software must be in place.

### Install HTK

HTK needs to be compiled for your platform and added to the path. The method for doing this will be different under different operating systems. See the following link for information on how to add programs to the path under Windows.

<http://lmgtfy.com/?q=windows+add+to+path>

Once the HTK binaries have been successfully added to the path, the individual tools should be available from Matlab. This can be tested by issuing the following command (the >> should not be typed):

>> !HVite -V

This command should then output some version information about HTK into the command window. If the command fails to return version information, then logging off and then back on again should solve the problem.

### Get appropriate speech material

The software provided is designed to work with the AURORA 2.0 TI digits corpus available here:

<http://www.elda.org/article52.html>

The clean training data should be in wav format and placed into one directory. The clean, digit-triplet test sound files should placed in a separate directory.

It is also possible to make a custom corpus so long as the following rules and file naming conventions are adhered to.

1. Speech material should be recorded in (or converted to) wav file format. Any sampling rate can be used, as the Matlab scripts will resample the speech files appropriately. The speech should be recorded in single channel format.
2. Training data should be recorded as strings of digits between 1 and approximately 7 digits per file. Test data files should contain 3 digits. Digits should include a fairly even mixture of “oh”, “one”, “two”, “three”, “four”, “five”, “six”, “eight”, “nine”. The bisyllabic digits “seven” and “zero” should not be used.
3. Speech files should be named like the following example, “FAC\_8O4A.wav”. The first character in the string refers to the gender of the talker. The next two characters are a unique identifier for the specific person doing the talking. These two characters should be followed by an underscore. The next characters are the string of numbers that are uttered in the sound file. These numbers are terminated with a capital “A” and the wav extension. IMPORTANT NOTE: If the file contains the utterance “oh”, the alphabetic character “O” should be used rather than the numerical character “0”.
4. The files in the training and testing corpora should be unique.

### Get appropriate noise material

Any noise material of suitable duration (substantially longer than the longest sound file containing speech) can but used. The tutorials here use the “factory1” noise sample from the freely available NOISEX database.

<http://spib.rice.edu/spib/select_noise.html>

### Show the cJob class where to find the sound files

The main job class (cJob.m) needs to know the location of the speech material. The speech material can be stored at any location on the user’s computer, but the following piece of code needs to be updated accordingly. If different computers on different platforms are used then the paths can be set accordingly. For example, I use a windows machine in the office, a mac at home, and a linux machine to run large jobs. All of these systems store the corpora in different locations.

if isunix

if ismac

lWAVpath = '~/ASR/reducedAURORA/TrainingData-Clean/';

rWAVpath = '~/ASR/reducedAURORA/TripletTestData/';

obj.noiseFolder = '~/ASR/noises';

else

lWAVpath = '/scratch/nnn/corpora/AURORA digits (wav)/TrainingData-Clean/';

rWAVpath = '/scratch/nnn/corpora/AURORA digits (wav)/TripletTestData/';

obj.noiseFolder = '/scratch/nrclark/corpora/noises';

end

else

lWAVpath = 'C:\corpora\AURORA digits (wav)\TrainingData-Clean';

rWAVpath = 'C:\corpora\AURORA digits (wav)\TripletTestData';

obj.noiseFolder = 'C:\corpora\noises';

end

### Setting an output folder

The speech recognition experiments involve generating a large number of files that need to be stored somewhere. It is possible to change the output folder on an experiment by experiment basis, but the user may wish to have a top level directory in which all ASR data is stored. To do this, find and amend the following code segment.

else

if isunix

if ismac

obj.opFolder = '~/ASR/exps/\_foo';

else

obj.opFolder = '/scratch/nrclark/exps/\_foo';

end

else

obj.opFolder = 'D:\exps\\_foo';

end

end

## Exp\_Tutorial\_1

### Training and testing a recogniser

If everything has been set correctly, it should now be possible to run Exp\_Tutorial\_1 without generating errors. This function can be used as a template to run all kinds of recognition experiments. The following text breaks down each part of the function, describing what happens in each block of code.

### Parallelism

The first line of the file declares the function and states that it takes one input argument, isMasterNode.

function Exp\_Tutorial\_1(isMasterNode)

Speech recognition experiments are very processor intensive, so the software suite has been carefully designed to run in parallel across many instances of Matlab. This allows results to be generated in a fraction of the time of a serial process if enough computing power is available. The simple scheduling software was written in house, and so it does not require any special Matlab licenses for clustering. The variable in the function definition above, isMasterNode, is a Boolean type that lets the current Matlab instance know if it is the master node.

The master node is the most important node, responsible for generating all of the job information and interfacing with HTK. Because the mater node has the responsibility of generating the job lists and storing them, the variable isMasterNode must be set to true when the experiment function is first called. While the experiment is running in one Matlab instance, it is then possible to share the workload by running the same command in another Matlab instance with isMasterNode set to false. In theory, there is no upper limit to the number of helper nodes.

Under windows, the simplest way to do this is to open a few Matlab instances and then set the experiment running with one master node. The other Matlab windows can be used as helper nodes. The demo experiments are designed to run entirely in the command window. No other aspects of the Matlab integrated development environment are needed. Therefore, small performance gains may be attained by running Matlab from the command prompt with the flag “-nodesktop”. This will run Matlab with the command window only. From there, just change directory to the working directory and invoke the experiment function from the command line. Trial and error must be used to find the optimum number of Matlab instances for a specific machine.

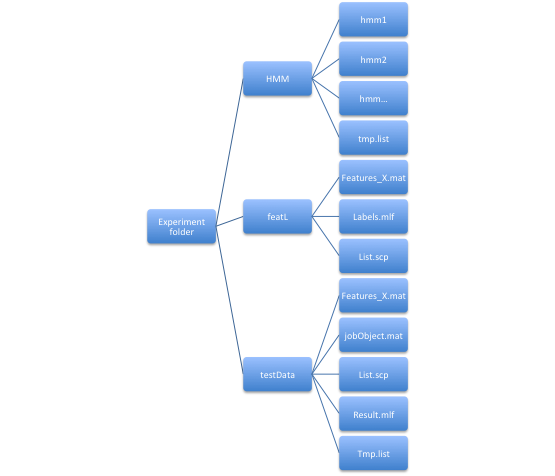
This setup is particularly powerful in a \*nix environment when used in conjunction with GNU screen <http://www.gnu.org/s/screen/>. From each virtual terminal, a separate, low-resource instance of Matlab can be launched with the command flag “-nodisplay”. Each instance of Matlab can then spread the job load as in the Windows example. If a number of machines are available with access to the same network attached storage, then the nodes can be spread across the different machines, with multiple nodes running on each machine. Of course, these nodes can be running a mixture of any operating system supported by Matlab.

### Experiment data directory structure

The speech recognition experiment process involves creating many small configuration and data files. The hierarchy chart below shows the directory structure generated for a typical experiment. All files related to a particular HMM are stored within a top-level experiment folder. The experiment folder contains at least two other folders. These are the HMM directory and the featL directory. There can also be any number of folders containing test features.

The HMM directory contains numbered hmm subdirectories that each contain a different version of the HMM after the sequential parameter re-estimations that occur during the training stage. The hmm36 folder contains the most refined HMM that is used to evaluate the experimental data. The HMM directory also contains a file called tmp.list. This is just a data file used by HTK to locate the training material.

The featL directory contains the training features. The “L” is used to signify learning as opposed to “R” for recognition. This is to avoid the obvious confusion that might arise if single character representations were used for training and testing features. The “L” and “R” abbreviations are also used throughout the software scripts. The saved training feature files correspond to individual wav files and thus have the same file names but with a .mat instead of a .wav extension. The featL directory also contains a file called labels.mlf. This text file explains the digits contained within each feature file in an HTK readable format. The list.scp file is a text document with the names of all of the feature files in the folder that HTK should use for training the recogniser.



The last file type in the training feature directory is the jobObject.mat file. This is a Matlab readable data file containing a copy of the instance of the cHMM class associated with the data set. This data object contains all of the information about the experiment, including MAP parameters, HMM parameters, and other instructions on how to generate features. Each processing node loads this object when initialised, so that it knows how to process the wav files appropriately. Each note processes a randomly assigned set of wav files and then updates this data object so that other nodes know which files are not yet processed, which files are currently cued for processing, and which files have been processed.

There can be an unlimited number of test data folders. The contents of these folders are each very similar to the featL directory, but contain test features. Furthermore, if the recogniser has been tested with the test features in that folder, the folder will also contain a result.mlf file. This is a text file that lists all of all of the files in the folder along with the digits that the recogniser has decided that they most likely contain. A simple % correct score can be extracted from the result file using a script built into the cJob class definition. Each test data folder contains test features for a single condition, so if the user wanted to evaluate the recogniser at 5 different SNRs, then 5 data folders would be required with a name that uniquely identifies that particular condition.

Returning to the analysis of the experiment function, the following code organises the directory structure.

%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Set up the basic folders

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

expName = 'Tutorial';

dataFolderPrefix = 'hello\_world';

if isunix

expFolderPrefix = '/scratch/nrclark/exps/';

else

expFolderPrefix = 'D:\Exps';

end

% expFolderPrefix = pwd;

expFolder = fullfile(expFolderPrefix,expName);

hmmFolder = fullfile(expFolder,'hmm');

The experiment name is defined by the variable expName. This is the name of the top-level folder that contains training features, testing features, and the hmm.

The dataFolderPrefix variable is a character string that precedes each folder containing test features. Typically, the user will create a HMM and iteratively test it, tweaking parameters according to conclusions drawn from previous results. For example, the user could give the first test of the HMM the dataFolderPrefix of ‘first\_test’. This would be the prefix for each of the folders that make up the recognition curve. The rest of the folder name will be derived from a parameter that would typically be SNR, but could be anything else. Based on the results, the user may later decide to change the MAP parameters, but test the existing HMM using these new parameters. All the user would need to change would be the dataFolderPrefix and the appropriate parameters.

The last user definable parameter is the expFolderPrefix. This is just a path to the root directory where the user wants to store all of the experimental data.

### Set some general parameters

Once the directory structure has been sorted, the next bit of code organises how the recogniser will be trained.

%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Sort out the training (LEARNING) condition

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

learnFolder = fullfile(expFolder,'featL');

xL = cJob('L', learnFolder);

xL.participant = 'Normal';

xL.MAPparamChanges= {'DRNLParams.rateToAttenuationFactorProb=0;', 'OMEParams.rateToAttenuationFactorProb=0;' };

xL.noiseLevToUse = -200;

xL.speechLevToUse = 60;

xL.MAPopHSR = 1;

xL.MAPopMSR = 0;

xL.MAPopLSR = 0;

xL.numCoeff = 14;

xL.removeEnergyStatic = 0;

%%%%% Group of params that will influence simulation run time %%%%%%%

xL.numWavs = 10; %MAX=8440

testWavs = 5; %MAX = 358

nzLevel = [-200 40:10:70];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

xL.noisePreDur = 2;

xL.noisePostDur = 0.1;

xL.truncateDur = xL.noisePreDur-0.1;

xL.noiseName = 'factory1';

An object of the cJob class is assigned to the variable xL. The cJob constructor is called with two arguments. The first argument is a flag telling the object that it is a training/learning job. The second argument is a path to where that job should put all of its data.

Once xL is defined, many of its default properties can be modified. The default properties can be found by either looking through the lines of code at the top of the class definition, or looking through the xL object just after instantiation using the debugger (All properties and member functions are described in the Appendices).

The participant property is the name of the parameter file to use in the parameterStore folder of MAP. The MAPparamChanges property allows the user to specify any deviations from the parameters specified in the parameter file. For this tutorial, the acoustic reflex and cochlear efferent feedback loops are disabled by zeroing the appropriate rate to attenuation factors.

The next two properties control the noise and speech levels, where the values are RMS dB SPL. **POSSIBLY OBSOLETE:** The Boolean “MAPop\*” properties determine which inner haircell types contribute to making the auditory spectrogram.

For purposes of data reduction, the auditory spectrogram is transformed into 10-ms segments and the spectrum at each epoch is data compressed using the DCT. The first and second order differences with respect to time are also extracted from the DCT coefficients and fed to the recogniser. The numCoeff property determines the number of DCT coefficients used to encode the auditory spectrogram at each epoch. The Boolean removeEnergyStatic property allows the user to remove the first DCT coefficient, but retain the difference values. In the tutorial example, the static energy coefficient is retained.

The subsequent few lines of code are enclosed in a comment block stating that they significantly influence the simulation run time. The first of these commands sets the numWavs property of xL. For this object, this is the number of wav files to be included in the training corpus. The more wav files used for training, the better the results will be in the testing phase. However, training time grows linearly with the number of wavs used while the returns diminish when using more than 1000 wav files (in the authors experience with this model, recogniser and speech corpus). If the computational resources are available, then there is no reason not to use the entire 8440 wav files, but general comparisons can be made by looking at results from a recogniser trained on as few as 600 wav files. In the tutorial example, 10 wav files are used for training. When training, HTK will give warning messaged in the Matlab command window stating that there are too few examples of each of the digits in the dictionary. This will produce garbage results, but will assert whether or not the software suite and paths are set correctly for the first run. The user can then experiment with larger training corpora when convenient. The next two lines of code within the block are not properties of xL, but are local function variables. These variables are associated with the testing phase of the experiment but are defined in this block as they also significantly influence the run time of the experiment. The variable testWavs is the number of wav files that should be used for testing the recogniser in each experimental condition. Again, the bigger the number, the more accurate the results. However, this is also at the expense of run time. The variable nzLevel is an array of noise levels (dB SPL) to use when testing the recogniser. In the tutorial example, noise levels are sampled every 10 dB. The granularity of the level sampling must be considered as the run time of the simulation will depend on the total number of testing conditions.

The properties defined after the comment block but before the if statement are related to the background noise. The noisePreDur property determines the duration of noise (in seconds) to be presented to MAP in isolation before the onset of the speech material. The MAP model contains numerous temporally dynamic processes and it is important to allow them to reach equilibrium before the onset of the speech material. One second is normally more than enough lead in time for the noise. However, if the user wished to experiment with particularly long time constants in any pre processing, such as the MAP model or hearing aid, then the pre roll should be at least 3 times greater than the longest time constant. This is because values returned from a sliding exponential integration window are only negligible at time intervals greater than 3 multiples of the time constant. The noisePostDur property determines the duration of noise (in seconds) to be presented to MAP in isolation after the offset of the speech material. The truncateDur variable determines the duration of the noise added at the beginning of the composite speech and noise sample to be discarded before the feature vector is saved. The truncation occurs after the auditory spectrogram has been generated as there is no need to retain information about the excess noise added to the start of the stimulus. The truncation has three main benefits:

1. Truncation saves disk space. This is an important consideration when generating 1000s of feature files.
2. The amount of data used in training and testing the HMM is reduced and so execution time is faster.
3. In the tutorial example, the resulting auditory spectrogram has 100ms of noise information at the beginning and end of the stimulus. This helps to give a more robust silence model in the trained HMM.

The final property in this group is noiseName, which is a string containing the name of the noise wav file.

### Wrapping up and saving the training job

The final bit of code relating to the training of the recogniser before the actual act of training the recogniser is given below.

if isMasterNode && ~isdir(xL.opFolder)

mkdir(xL.opFolder);

xL = xL.assignFiles;

xL.storeSelf;

end

The first command within the if statement block creates the new experiment folder. The second command executes a member function called assignFiles within the xL object. This member function randomly allocates wav files from the appropriate corpus to the training job and sets a flag associated with each wav file stating that the wav file has not yet been converted into a feature vector. This code block only executes if the function is running as the master node (as defined when calling the function) and if the directory does not exist. Should a failure occur such as a crash or power outage, this allows the user to resume work on the speech recognition job by just running the experiment function again. If the user wishes to restart a job, perhaps because of a typo when setting a parameter, then the experiment directory must be renamed or deleted, or the experiment name in the function must be changed. Otherwise, the function assumes recovery mode.

### Setting parameters for the testing job

Up to this point, the code in the tutorial function has been used to set parameters. No training of the recogniser has occurred at this point. The code described in this section finalises the setting of the testing parameters before the training and testing commences.

%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Sort out the testing (RECOGNITION) conditions

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

recConditions = numel(nzLevel);

tmpIdx=0;

for nn = 0\*recConditions+1:1\*recConditions

tmpIdx=tmpIdx+1;

xR{nn} = xL; %simply copy the "Learn" object and change it a bit below

recFolder = fullfile(expFolder,[dataFolderPrefix num2str(nn)]);

xR{nn}.opFolder = recFolder;

%These are the interesting differences between training and testing

xR{nn}.numWavs = testWavs; %MAX = 358

xR{nn}.noiseLevToUse = nzLevel(tmpIdx);

xR{nn}.MAPparamChanges= {'DRNLParams.rateToAttenuationFactorProb=0;'};

%Now just to wrap it up ready for processing

if isMasterNode && ~isdir(xR{nn}.opFolder)

mkdir(xR{nn}.opFolder);

xR{nn} = xR{nn}.assignWavPaths('R');

xR{nn} = xR{nn}.assignFiles;

xR{nn}.storeSelf;

end

end

When testing a recogniser, the user will normally want to try groups of parameters that are variations on a theme so that trends can be observed. The example given in the code generates a recognition job that tests the speech material over the range of SNRs defined earlier in the function with no cochlear efferent attenuation. In the tutorial function, there is another for loop block, below the one shown here in the text, which is identical in every way apart from the MAPparamChanges property and the range of the “nn” index variable (more on this below). An unlimited number of these for loop blocks can be pasted one after another, so long as care is taken to update the index variable appropriately.

The integer recConditions variable represents the number of recognition conditions for each parameter variation. It is defined as the number of noise levels in this tutorial example, as it is fairly common to test a recogniser at a range of SNRs. The tmpIdx variable is a temporary index, as the name suggests, that is used within each for loop and reset prior to each new for loop.

The first important line in the loop block is the statement xR{nn} = xL. Each test condition job begins life as an exact copy of the training job that is subsequently modified. The test (or ‘R’ for recognition) job is placed into a cell array with the index nn. The current experiment has 5 noise levels and 2 different sets of parameters, so the nn index will count from 1 to 10. The counter value is appended to the expFolderPrefix string to make a unique folder name for the testing features for that specific parameter set and SNR.

The next three lines under the comment “These are the interesting differences between training and testing” are indeed the changes that are made to the testing condition to make it a training condition. In the tutorial example, the number of wav files is changed, the noise level is set, and a call to MAPParamChanges is made. The parameter change specified here is redundant as it is already one of the parameters of the training job. However, it never hurts to be specific.The final block of lines in the if statement store each job ready for processing. This is accomplished using the same methods described to store the training job.

tmpIdx=0;

for nn = 1\*recConditions+1:2\*recConditions

tmpIdx=tmpIdx+1;

xR{nn} = xL; %simply copy the "Learn" object and change it a bit below

recFolder = fullfile(expFolder,[dataFolderPrefix num2str(nn)]);

xR{nn}.opFolder = recFolder;

%These are the interesting differences between training and testing

xR{nn}.numWavs = testWavs; %MAX = 358

xR{nn}.noiseLevToUse = nzLevel(tmpIdx);

xR{nn}.MAPparamChanges= {'DRNLParams.rateToAttenuationFactorProb=-10^(-10/20);'};

%Now just to wrap it up ready for processing

if isMasterNode && ~isdir(xR{nn}.opFolder)

mkdir(xR{nn}.opFolder);

xR{nn} = xR{nn}.assignWavPaths('R');

xR{nn} = xR{nn}.assignFiles;

xR{nn}.storeSelf;

end

end

There is also a second for loop that generates another set of jobs for the recogniser, but using a fixed, 10-dB SNR cochlear efferent. This change is applied when the MAPparamChanges property is redefined. Another subtle difference is in the for loop statement line where nn is defined as 1\*recConditions+1:2\*recConditions, instead of 0\*recConditions+1:1\*recConditions. This is to keep job identities and folder names unique.

### Notes on performance warnings

The tutorial example function produced numerous M-lint warnings relating to expanding array sizes within for loops. These warnings can be safely ignored as the performance hit is negligible relative to the computation time required to produce each feature vector. Memory allocation of arrays would complicate the script and provide no measurable performance benefits in this instance.

### Feature Generation

Once the files explaining the jobs for training and testing have been created, the features can be generated. This is the most time consuming and computationally expensive part of the procedure. Even so, the code in the experiment function to do the feature generation is remarkably simple once the jobs have been created.

There is a script in the software suite called worker. This takes a single job object as a variable and will busily perform the task of converting wav files into feature vectors until all of the wav files in the list have been converted. The code below shows the order in which the jobs are processed.

worker(xL.opFolder);

maxConds = nn;

if ~isMasterNode %dont bother wasting master node effort on testing jobs (for now)

for nn = 1:maxConds

worker(xR{nn}.opFolder);

end

end

All nodes are assigned the task of generating the training features as nothing can be done with HTK until a set of training features are available. The if statement only allows slave nodes to generate the testing features at this stage. This is so the master node can get on with training the recogniser ready for when the test features become available. The training of the recogniser can only be done on a single node. Therefore, it is more efficient to have the recogniser being trained while other nodes are still generating features, rather than stop all nodes to wait for the recogniser to be trained before doing anything else.

### Training and testing the recogniser

%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Train and test the recogniser - a job for the master node only

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if isMasterNode

while(~all(xL.todoStatus==2))

disp('Waiting on straggler nodes to complete their jobs before HMM is trained . . .')

pause(30); %Wait for 30 seconds before looking again

xL.lockJobList;

xL = xL.loadSelf; %Reload in case changed

xL.unlockJobList;

end

As the comment block in the above code snippet suggests, the remainder of the function is for the attention of the master node only. The while loop checks every 30 seconds to see if the training features have all been created before moving on to the actual training. Each element of the integer array property todoStatus can be set to 0, 1, or 2. Each element of the array corresponds to a wav file in the processing list. A value of 0 means that the wav file is open for any node to grab for processing, a value of 1 mans that the wav file has been grabbed by a node and will be processed shortly, a value of 2 mans that a feature file has been successfully created for that particular wav file.

The member function lockJobList places a file mutex into the current job folder to stop any other nodes editing the job file while it is being inspected by the master node. The member function loadSelf updated the job stored in master node memory from the job file on disk. The job file on disk may have been changed by a slave node since it was last checked by the master node. The member function unlockJobList removes the file mutex once the inspection is complete, enabling slave nodes to update the job file if necessary.

y = cHMM(hmmFolder);

y.numCoeff = (xL.numCoeff-logical(xL.removeEnergyStatic)) \* 3;

y.createSCP(xL.opFolder)

y.createMLF(xL.opFolder)

y.train(xL.opFolder) %This node can be busy training, even if test jobs are being processed

Once all of the training features have been generated, a HMM class is instantiated by passing the path of the hmm to the constructor. If the hmm folder already contains a trained HMM, then the HMM class will accommodate this, allowing the user to easily recycle trained recognisers. In the tutorial example, the HMM folder is empty and so a new recogniser must be created.

Once the recogniser has been created, the numCoeff property is changed. This refers to the number of coefficients in the feature vector at each time epoch. In the example given, the number of coefficients is three times the number of DCT coefficients because deltas and accelerations of the features must be included. The HMM class will then automatically select the appropriate HMM prototype given this value. NOTE: THE CODE IN cHMM IS A BIT STALE AND MAY NEED LOOKING AT.

The member function createSCP, surprisingly, creates an SCP file in the hmm directory. Remember form earlier, that the list.scp is just a big list of feature vector names required by HTK. Similarly, the createMLF member function generates a script in an HTK compatible format to tell the recogniser what digits the feature files actually contain. Finally, the member function train wraps up the complex list of HTK commands required to train a recogniser into a simple function.

% ALLOW MASTER NODE TO MUCK IN WITH GENERATING TESTING FEATURES ONCE

% HMM HAS BEEN TRAINED

for nn = 1:maxConds

worker(xR{nn}.opFolder);

end

Once the recogniser has been trained, the code pasted above allows the master node to join in with generating testing features. If the recognition experiment is running using just a single node then the generation of testing features will commence at this point.

xR{end}.lockJobList;

xR{end} = xR{end}.loadSelf; %Reload changes

xR{end}.unlockJobList;

while(~all(xR{end}.todoStatus==2))

disp('Waiting on straggler nodes to complete their jobs before HMM is tested . . .')

pause(30); %Wait for 30 seconds before looking again

xR{end}.lockJobList;

xR{end} = xR{end}.loadSelf; %Reload incase changed

xR{end}.unlockJobList;

end

for nn = 1:maxConds

y.createSCP(xR{nn}.opFolder);

y.test(xR{nn}.opFolder);

end

Once the testing features have been created, the code snippet above shows that a check is made to see if all of the jobs have been completed before anything else happens. Each node allocates between 8 and 16 wav files to itself at a time between updating the job object stored on disk, so it is common for the master node to have to wait a few minutes at this stage if more than 1 node is being used. Once the waiting is over, an SCP file is created like for the test set, but no MLF file is needed as the testing procedure produces its own MLF that may or may not be accurate.

%Show all of the scores in the command window at the end

for nn = 1:maxConds

y.score(xR{nn}.opFolder);

end

The final bit of code in the script scores the recogniser generated MLF script against the actual digits spoken. For this, a static method is used in the cHMM class rather than the scoring tools included with HTK. The Matlab code gives a correct score only when the correct digit is identified in the correct position within the triplet. In contrast, the HTK scoring tools use a dynamic programming strategy. The score method (or member function depending on your preferred object-oriented lingo) displays both the percent correct word score and percent correct sentence score. The word score is based on each individual digit and the sentence score is based on the recogniser identifying every digit correctly in a triplet. Score is a static method and so it can be used from the command line given any folder containing appropriate files without the associated object.

### Play

If you have time, try enlarging the number of files included in the training and testing sets. This should produce recognition scores that are not garbage.

## Exp\_Tutorial\_2

### Overview

The previous tutorial script demonstrated a method for training and testing an automatic speech recogniser using two different sets of parameters in the testing stage. The second tutorial script shows the user how to recycle an existing HMM, and how to attach a hearing aid simulation.

## Appendix A: cJob properties and member functions

### Publicly Accessible Properties

Aardvark

### Member Functions

Aardvark

## Appendix B: cHMM properties and member functions

### Publicly Accessible Properties

Aardvark

### Member Functions

Aardvark

## Appendix C: cEssexAid properties and member functions

### Publicly Accessible Properties

Aardvark

### Member Functions

Aardvark