ASR Suite for Dummies – USER MANUAL

Hearing Research Laboratory, Dept. of Psychology

University of Essex, Colchester

First version: 15th October 2011

Last revision: 27th October 2011

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.

1. 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.

2. 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.

3. 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.

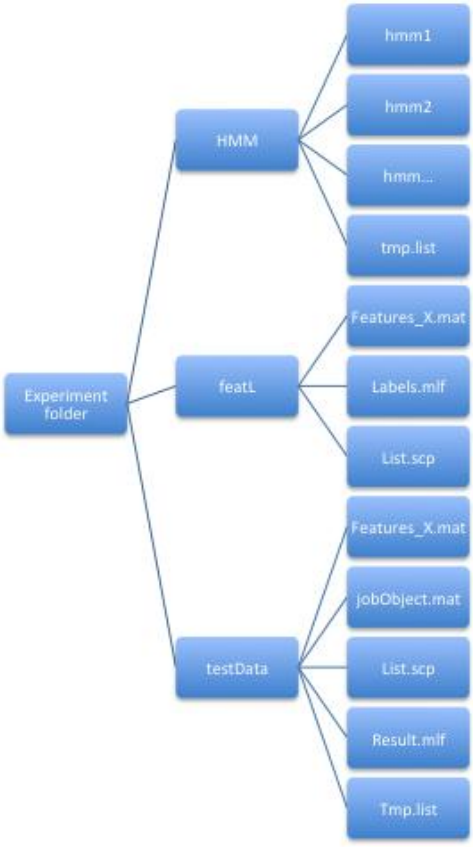
This setup is particularly powerful in a *nix environment when used in conjunction with GNU screen http://www.gnu.org/s/screen/. From each virtual terminal, a separate, low-resource instance of Matlab can be launched with the command flag “-nodisplay”. Each instance of Matlab can then spread the job 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. The Boolean “MAPop*” properties determine which groups of inner haircells - categorised by spontaneous rate - are used to make the auditory spectrogram. To switch between, or combine haircell types, the user must have specified more than one spontaneous rate in the MAP parameter file. This ia done by adding values to the tauCa array (see MAP documentation for full details).

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:

i) Truncation saves disk space. This is an important consideration when generating 1000s of feature files.

ii) The amount of data used in training and testing the HMM is reduced and so execution time is faster.

iii) 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 to the front-end of the model.

Code walkthrough

The script associated with this second tutorial is called Exp_Tutorial_2.m. Rather than dissect each individual line of the script, only particular lines of interest are discussed.

```
expName = 'Tutorial';
dataFolderPrefix = 'hello_world_recycle';
```

The variable expName has the same value as in the previous tutorial. The software will then be able to find the pre-trained HMM in this folder. The data folders are given a different prefix, so that they are not confused with the testing features generated in the previous tutorial.

From here downwards, nearly all lines of code are identical to the previous example, with the exception of the following excerpt that is commented out.

```
% if isMasterNode && ~isdir(xL.opFolder)
%     mkdir(xL.opFolder);
%     xL = xL.assignFiles;
%     xL.storeSelf;
% end
```

There is no need to store the learning parameters as in this tutorial because there is no need to train a new recogniser. However, the learning parameters still need to be defined in memory because the testing jobs start life as identical copies of the training job that are subsequently modified. An alternative (and more robust) method for doing this would be to load the old training job from disk and then modify the parameters from there, but for the sake unambiguity, the parameters are explicitly redefined in the tutorial script.

Once the faux training parameters have been recalled, the testing jobs are then created by cloning and changing the original parameters. The first testing job makes the following changes.

```
%These are the interesting differences between training and testing
xR{nn}.numWavs = testWavs; %MAX = 358
xR{nn}.noiseLevToUse = nzLevel(tmpIdx);
xR{nn}.MAPparamChanges= {'DRNLParams.a=400;'};
```

The first two lines of code set the number of test wav files to be converted to features and the noise levels that the recogniser should be tested at. The third line simulates an outer haircell dysfunction by reducing the gain applied to the non-linear path of the DRNL (Refer to Manassa Panda's 2010 PhD Thesis for more information). An 'a' value of about 400 should raise the pure tone thresholds of the dummy by approximately 20 dB relative to a normal hearing dummy, although this is highly dependent on the 'Normal' parameter file used, so the readers mileage may vary. A reduction in the 'a' parameter also reduces the dynamic range of the compressive region of the basilar membrane input/output function leading to a simulated recruitment. Furthermore, a desensitised outer-

periphery will be a less effective driver of the efferent reflexes that are derived from signals present in the inner periphery. As a result of this simulated impairment, the recognition scores generated from this testing job should be far worse than those generated from the unimpaired model in the previous tutorial so long as the numbers of testing wavs used are sufficient.

The script also defines a second testing job with the following parameter modifications

```
    %These are the interesting differences between training and testing
    xR{nn}.numWavs = testWavs; %MAX = 358
    xR{nn}.noiseLevToUse = nzLevel(tmpIdx);
    xR{nn}.MAPparamChanges= {'DRNLParams.a=400;'};

    xR{nn}.mainGain = [27.2013;   26.0797;   26.0939;   26.7997;   26.0520]; % gain
    xR{nn}.TCdBO    = [37;   37;   37;   37;   37] %Compression thresholds (OUTPUT 2nd filt)
    xR{nn}.TMdBO    = [20;   20;   20;   20;   20]; %MOC thresholds (OUTPUT from 2nd filt)
    xR{nn}.ARthresholddB = 85;       % dB SPL (input signal level) =>200 to disable
    xR{nn}.MOCtau = 1;
    xR{nn}.useAid = 1;
```

The first three lines are the same as for the previously defined set of testing jobs. The set of lines under the blank line define hearing aid parameters. The last of the parameters is called useAid, which slots the hearing aid processing in before the MAP model when set to true.

The full list of hearing aid parameters can be found in the job class around line #100. The parameters and wrapper function for the Essex aid are all discussed in a companion document. It should be relatively simple for any competent Matlab user to swap out the EssexAid algorithm and drop in their own code to test the potential benefits of different hearing aid processing strategies. All the user would need to do be to swap out the line at ~#542 in the job class:

```
stimulus = EssexAid_guiEmulatorWrapper(stimulus, sampleRate, obj);
```

This line can be replaced with any function that processes that takes the raw stimulus in pascals, returning the processed stimulus, also in pascals.

Once the jobs have been defined that recycle the previously created HMM, the testing jobs can now be worked on. There is no need to create the training jobs in this tutorial and so the appropriate line has been commented out.

```
% worker(xL.opFolder);
maxConds = nn;
if ~isMasterNode     for nn = 1:maxConds
        worker(xR{nn}.opFolder);
    end
end
```

Similarly, in the final code segment, all lines referring to training the recogniser have been commented out as this is not a requirement when recycling a HMM.

Using the HMM class helper methods

The HMM class has many small helper methods built in that allow the user to perform common tasks with ease. For example, an HMM object can be initialised from a folder containing a trained HMM by issuing the following command:

```
>> y = cHMM(pathToTrainedHMM);
```

This HMM can then be tested on any folder containing test features, so long as the features are compatible with the HMM (i.e. correct number of elements per vector). This process is relatively fast compaired to the training phase.

```
>>y.createSCP(pathToTestFeatures);
>>y.test(pathToTestFeatures);
```

Another useful helper function is the static method “score”. This returns the word and sentence scores for a given folder in percent correct and does not even requite an object of the class to run. It can be called using the class name or an object name as the prefix:

>>cHMM.score(`pathToTestFeatures`);

Combined, the helper functions make it very easy to recycle old HMMs for testing with new data or vice versa.

Appendix A: cJob properties and member functions

Publicly Accessible Properties

wavFolder – This is the path to the directory containing the speech wav files.

opFolder – This is a path to the directory where the features should be stored once they have been created.

noiseFolder - This is the path to the directory containing the noise wav files.

wavList – This is a structure array of directory information about all of the wav files to be used in the current job. While this property is public, it is quite difficult to set by hand and the appropriate member function (`assignFiles`) should be used to populate this array in most instances.

todoStatus – This is an integer array where each element refers to an individual wav file. The values can either be 0=not started, 1=assigned to a node and not started or being processed, 2=completed and feature file successfully written. Under most circumstances, the user will never directly access this property.

participant = 'Normal' – This is the name of the parameter file for MAP to use.

noiseName = '8TalkerBabble'; - This is the name of the wav file to use for the background noise.

numWavs = 5; - This is the number of wav files to use. It determines the length of the wavList array.

noiseLevToUse = -200; -This is the level of the background noise in dB SPL RMS.

speechLevToUse = 50; - As above but for speech.

speechLevStd = 0; - The standard deviation (in dB) of the speech levels between successive wav files. Training a recogniser over a range of speech levels can give more robust recognition performance when testing at different speech levels at the expense of peak performance.

`noiseLevStd = 0;` The standard deviation (in dB) of the noise levels between successive wav files. Training a recogniser over a range of noise levels can give more robust recognition performance when testing under different noise levels at the expense of peak performance when the recogniser is trained and tested at the same SNR.

freezeNoise = 0;- By default, the sample of background noise added to the speech is a randomly selected portion form a long sample. Setting this property to true stops the randomisation so that the noise is always started from the first element in the array.

speechDist = 'None'; - The distribution of the random speech levels (Guassian/Uniform/None)

noiseDist = 'None'; - The distribution of the random nooise levels (Guassian/Uniform/None)

meanSNR = 20; - This variable can be set to maintain a SNR if a random noise distribution is used.

noisePreDur = 0.0; - How long (in seconds) the background noise should run before the onset of the speech material.

noisePostDur = 0.0; - How long (in seconds) the background noise should run after the offset of the speech material.

truncateDur = 0.0; - How much of the noise added prior to the onset of the speech material should be removed after the feature has been generated. It is normally necessary to allow the dummy to settle into an adapted state before hitting it with speech material, hence the noise added before the speech onset. However, once the features have been generated then the noise can be removed to save disk space.

```
currentSpeechLevel – Internally used variable: do not edit – SHOULD NOT BE PUBLIC!?
```

```
currentNoiseLevel – Internally used variable: do not edit – SHOULD NOT BE PUBLIC!?
```

useSpectrogram = false; - Uses a simple fft-based spectrogram in place of the auditory spectrogram generated using map.

numCoeff = 9; - the upper number of DCT coefficients to use at each epoch from the auditory spectrogram.

The following group of properties can be set to Matlab axis handles. If set, the object will plot the associated data as each wav is processed. This gives the user a window into the output of different stages when running the speech recognition experiment.

```
probHaxes   = []; - Firing rate probability

probHaxesSM = []; - as above after temporal smoothing into 10ms bins

featHaxes   = []; - features from DCT

reconHaxes  = []; - either sacf or rate features reconstructed from the DCT coefficients. This gives the user som insight into whether a sufficient number of coefficients are being used to encode the interesting portions of the features.
```

The following group of properties are associated with the SACF should the user decide to make features from the SACF rather than directly from the firing rate probability. Refer to the MAP documentation for specific details on these parameters.

```
useSACF             = false;

SACFacfTau          = 2; % > 1 = Wiegrebe mode

SACFnBins           = 128;

SACFminLag          = 1 / 4000;

SACFmaxLag          = 1 / 50;

SACFlambda          = 10e-3;
```

The following group of properties are associated with MAP

MAProot = fullfile('..'); - Path to the root directory of map

MAPplotGraphs = 0; - Determine whether to use MAP's own plot routines each time it is called

MAPDRNLSave = 0; - Save the basilar membrane motion after each run for exploration.

MAPopLSR = 0; - Use low spontaneous rate IHC fibres in the generation of rate features

MAPopMSR = 0; - Use low spontaneous rate IHC fibres in the generation of rate features

MAPopHSR = 1; - Use low spontaneous rate IHC fibres in the generation of rate features

MAPparamChanges = {}; - Parameter values that differ from those specified in the file.

The following group of properties are related to HTK.

frameshift = 10; Frame rate in ms

sampPeriodFromMsFactor = 10000; This number is appropriate for a 10 ms window. Refer to the HTK book for details.

paramKind = 9; 9 is a code that specifies that a userdefined format is to be used. See page 73 of the HTK book for more details.

removeEnergyStatic = false; - If set to true, the first DCT coefficient will be removed before the feature vector is saved to disk. This removes information about the overall level of the stimulus and will also preven deltas and accelerations from being calculated from the first coefficient.

doCMN = false; Option to use cepstral mean normalisation.

The remainder of the parameters are related to the hearing aid. These parameters are discussed in the accompanying document.

useAid = 0; - This is the only property that is not part of the EssexAid wrapper class. When set to true, this applies the aid processing at the interface between the acoustic signal and the dummy.

Member Functions

`function obj = cJob(LearnOrRecWavs, jobFolder)`
This is the constructor. There are two optional parameters that can be passed on initialisation. The first is `LearnOrRecWavs` that can be the character 'L' or 'R'. This value is used internally by the object to set the paths to the corpora appropriately, The other value is a path to a folder to use for file output.

`function obj = assignWavPaths(obj, LearnOrRecWavs)`
This function tells the object where to find the recorded sound materials.

`function obj = lockJobList(obj)`
This function places a file mutex in the output folder preventing access from other jobs while the master job file is updated.

`function obj = unlockJobList(obj)`
This function safely removes the mutex once file operations have been completed by a node.

`function storeSelf(obj)`
This function safely overwrites the master job list.

`function loadSelf(obj)`
This function safely loads the master job list.

`function value = get.jobLockTxtFile(obj)`
Simple get method to return the full path of the file mutex.

`function checkStatus(obj)`
This returns the progress of the current job and places some visual feedback related to the progress in the command window.

`function obj = initMAP(obj)`
This adds all of the necessary folders to the working path so that MAP can function.

`function obj = assignFiles(obj)`
This populates the wavList array with file names that are going to be processed.

`function obj = genFeat(obj, currentWav)`
This function generates features for a given wav file according to the rules specified in the object.

`function obj = opForHTK(obj, currentWav, featureData)`
This function writes the feature vector to disk in a binary format that is readable by the HTK tools.

`function [stimulus, sampleRate] = getStimulus(obj, currentWav)`
This function generates a stimulus to supply to MAP, performing any resampling, noise addition, truncation/extension of the signal or noise that is necessary.

`function [finalFeatures, ANprobabilityResponse] = processWavs(obj, currentWav)`
This is the main function within the class that is responsible for the actual feature generation once all of the housekeeping has been organised.

Static Methods

These are methods that can be called directly from the class definition and do not require an object of the class.

```
function ANsmooth = makeANsmooth(ANresponse, sampleRate, winSize, hopSize)
```

This function takes any 2-dimensional array of data and temporally smoothes it along the 2nd dimension according to the parameters. IN this class, it is used to smooth the rate representation of a sound before the DCT is computed.

```
function ANfeatures = makeANfeatures(ANrate, numCoeff)
```

This calls the DCT function and truncated the DCT features accordingly.

Appendix B: cHMM properties and member functions

This class is internally much simpler than the job class as it is just a wrapper for the HTK binaries.

Publicly Accessible Properties

`hmmFolder`

This is the name of the folder where the HMM should be save to disk as it is being trained. It is the name for the folder where the HMM can be retrieved if it has already been trained.

```
paramType      = 'USER_D_A'; %DELTAS and ACCELERATIONS
```

This is a string that describes the type of features in the files to HTK. See the book for more information.

```
numCoeff   = 27;
```

This is the number of coefficients at each epoch of the feature. This is the number of DCT coefficients plus the deltas and accelerations. Therefore, if the first 9 DCT features are used with both deltas and accelerations to be calculated by HTK, then this value should be set to 27.

```
HERestDataPath       = fullfile(pwd, 'def', 'HERest_digit');
```

Directory containing instructions for various stages of model re-estimation conducted during the training process.

```
binPath              = fullfile(pwd, 'def', 'bin');
```

This is the path to some binary files that make initialised HMMS. I am not sure of the legalities associated with redistribution of these and so they may need rewriting in Matlab. This should be a simple task.

```
configFile           = fullfile(pwd, 'def', 'config_tr_zcpa12');
```

Path to a file containing some HTK flags

`trainWordListFile`

Path to training word list

`testWordListFile`

Path to training word list (that does not include the zero or seven in this tutorial)

`wordNetFile`

Path to the word network file

```
dictFile   = fullfile(pwd, 'def', 'Grammar_digit', 'noSevenZeroDict');
```

Path to the dictionary file

Member Functions

```
function obj = cHMM(hmmFolder)
```
The constructor is very simple. It just takes a path to the hmm folder. The HMM folder will then be populated with subdirectories during training that will contain the HMM after each parameter re-estimation. If the hmm folder already contains a trained HMM, then the object will be able to use this HMM for testing.

```
function genProto(obj)
```
Generates a prototype HMM at the beginning of training.

```
function boolans = istrained(obj)
```
This function looks in the HMM folder for a trained HMM that has been trained all the way to the final stage of parameter re-estimation. If such a model exists, the function returns true.

```
function train(obj, trainFeatureFolder)
```
This is the main function of the class that performs the training. It is responsible for calling all of the necessary HTK commands to achieve this task.

```
function test(obj, testFeatureFolder)
```
This is the testing function that generates MLF files in each of the directories containing testing features. Scores/confusion matrices/whatever can then be derived from the MLF files.

```
function value = get.protoFile(obj)
```
This is a function that selects the appropriate prototype file depending on the state of some other parameters. This contains many conditions that were once tested and then abandoned.

Static Methods

These are methods that can be called directly from the class definition and do not require an object of the class.

```
function createMLF(mapFileFolder)
```
This method creates an HTK compatible MLF file from the file names within the folder. This MLF file is used during the training stage, to tell HTK what each feature file contains. This is in contrast with the testing phase, where the recogniser generates its own MLF file indicating what it hypothesises is within each speech feature file.

```
function createSCP(mapFileFolder)
```
This function just generates a text document containing a list of files within the given directory.

```
function score(testMLFpath)
```
This function compares the MLF file generated by the recogniser in the testing phase with the files that were tested and generated word and sentence scores in percent correct.

```
function scoreWholeFolder(folderToScore, searchString)
```
This performs the score function in a loop for all of the recognition data folders in an umbrella directory. Scores are given for folders containing the generated MLF files that contain the search string.

```
function opNum = htk_str2num(ipString)
```
Utility function that converts text strings into integer strings.