# Building a Simple Speaker Identification System

*John McKenna*
*School of Computing, DCU*

## 1. Introduction

1.1. We will be using the Hidden Markov Model Toolkit (HTK). HTK is installed under Linux on the main lab machines. Both Linux and Windows versions are available for download on the web. For instance, install HTK within ***Cygwin*** on Windows, or you may be able to find binaries somewhere. The HTK Book can be found at http://www.computing.dcu.ie/~john/htk/htkbook.pdf or find the latest edition at the HTK website. Have a *quick* browse thru it. Familiarise yourself with where the tutorial and reference sections can be found.

1.2. What follows is a step-by step tutorial on how to create a simple speaker recogniser. If there are any problems with the tutorial, please let someone know, preferably via Moodle.

1.3. The tutorial is like a basic *recipe*. As your skills develop you should experiment with other *ingredients*. The HTK book will give you ideas.

## 2. The Grammar

2.1. HTK uses a finite state grammar that consists of variables defined by regular expressions. Create a file called `gram.txt` and place the following in it.

```
$speaker = John | Mary;
($speaker)
```

2.2. A word network must be created from the grammar. This can be done using `HParse`:

```
HParse gram.txt wdnet
```

2.2.1. Have a look at the resulting file `wdnet`. Does it make any sense?

## 3. Recording

3.1. Record both John and Mary uttering the vowel /a/ as in “father” (as you may have done at the doctor’s at some point in your life). Make four recordings of the vowel for each speaker. Place them in files `ja{1-4}.wav` and `ma{1-4}.wav`. Note the sampling rate at which the recordings were made.

## 4. Parameterisation

4.1. We must first extract relevant information from the speech spectra. This is called parameterisation. We will use Mel-frequency cepstral coefficients (MFCCs).

4.2. Note that if using Linux, create/edit files with a Linux-based editor, as different platforms use different markers to indicate the end of a line.

4.3. We must build a configuration file. Call it `config_wav2mfc`. In it place the following:

```
# Coding parameters
SOURCEKIND     = WAVEFORM
SOURCEFORMAT   = WAV
SOURCERATE     = Sampling Period in units of 10^-7s
TARGETKIND = MFCC_0_D_A
TARGETRATE = Frame skip duration in units of 10^-7s
SAVECOMPRESSED = T
SAVEWITHCRC = T
WINDOWSIZE = Frame duration in 10^-7s
USEHAMMING = T
PREEMCOEF = 0.97
NUMCHANS = 26
CEPLIFTER = 22
NUMCEPS = 12
ENORMALISE = T
```

4.4. Note that in your configuration file, you must replace the italicised pieces with actual values. Use any frame skip in the range 10-20ms. Use any frame duration in the range 20-40ms.

4.5. We can list the `wav` files (and corresponding targets) we wish to parameterise in a file. Let's call it `convert.scp` and assume that our data is stored in a directory named `Data` (Note that directory delimiters may need to change depending on operating system):

```
Data/ja1.wav Data/ja1.mfc
Data/ja2.wav Data/ja2.mfc
Data/ja3.wav Data/ja3.mfc
Data/ja4.wav Data/ja4.mfc
Data/ma1.wav Data/ma1.mfc
Data/ma2.wav Data/ma2.mfc
Data/ma3.wav Data/ma3.mfc
Data/ma4.wav Data/ma4.mfc
```

4.6. MFCCs are extracted from the `.wav` files with `HCopy`:

```
HCopy -T 1 -C config_wav2mfc -S convert.scp
```

4.7. Use `HList` to view the contents of any of the `mfc` files. Do they make sense?

## 5. Model Preparation

5.1. We will now initialise two speaker models with our training data.

5.2. First we will create another configuration file, `config_mfc`, to let the HTK tools know about our `mfc` files:

```
# Coding parameters
TARGETKIND = MFCC_0_D_A
TARGETRATE = ??
SAVECOMPRESSED = T
SAVEWITHCRC = T
WINDOWSIZE = ??
USEHAMMING = T
PREEMCOEF = 0.97
NUMCHANS = 26
CEPLIFTER = 22
NUMCEPS = 12
ENORMALISE = T
```

5.3. We will also list the `mfc` files we wish to use for training. We will use three tokens for training each model and retain the other for testing. List these in `training.scp`:

```
Data/ja1.mfc
Data/ja2.mfc
Data/ja3.mfc
Data/ma1.mfc
Data/ma2.mfc
Data/ma3.mfc
```

5.4. Create a new directory `hmm0` and place the following prototype model in a file called `proto`:

```
~o <VecSize> 39 <MFCC_0_D_A>
~h "proto"
    <BeginHMM>
     <NumStates> 5
     <State> 2
        <Mean> 39
          0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0
        <Variance> 39
          1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
1.0 1.0 1.0
     <State> 3
        <Mean> 39
          0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0
        <Variance> 39
          1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
1.0 1.0 1.0
     <State> 4
        <Mean> 39
```

```
          0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0
        <Variance> 39
          1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
1.0 1.0 1.0
     <TransP> 5
      0.0 1.0 0.0 0.0 0.0
      0.0 0.6 0.4 0.0 0.0
      0.0 0.0 0.6 0.4 0.0
      0.0 0.0 0.0 0.7 0.3
      0.0 0.0 0.0 0.0 0.0
    <EndHMM>
```

5.5. Now use HCompV to initialise the models with the training data:

```
HCompV -C config_mfc -f 0.01 -m -S training.scp -M
hmm0 proto
```

5.6. This results in the creation of two files – `proto` and `vFloors` – in the directory `hmm0`. These files must be edited in the following way:

5.6.1. An error occurs at this point which rearranges the order of the parts of the `MFCC_0_D_A` label as `MFCC_D_A_0`. This must be corrected.

5.6.2. The first three lines of proto must be cut and pasted into `vFloors`, which is then saved as `macros`.

5.6.3. A file called `hmmdefs` is created by copying and pasting the rest of the proto file once for each HMM and renaming the copies accordingly. Note that each HMM begis with ~h "*model_name*" and ends with `<EndHMM>`. For instance you could call your models "ja" and "ma".

## 6. Model Re-estimation

6.1. Now that we have initialised our two speaker-vowel models with global means and variances, we will use HERest to perform Baum-Welch training.

6.2. For this step we need to create two more files: `speakertrainphones0.mlf` and `phonemodels0`. The former contains what the HTK book refers to as 'phone-level transcriptions'. These specify the phones in each of the sound files to be used in training the HMMs. The latter simply lists the phone-level models.

6.3. `speakertrainmodels0.mlf` will look like:

```
#!MLF!#
"*/ja1.lab"
ja
.
"*/ja2.lab"
ja
.
"*/ja3.lab"
ja
.
"*/ma1.lab"
ma
.
"*/ma2.lab"
ma
.
"*/ma3.lab"
ma
```

6.4. `phonemodels0` …

```
ja
ma
```

6.5. We then re-estimate our models

```
HERest -C config_mfc -I speakertrainmodels0.mlf -S
training.scp -H hmm0/macros -H hmm0/hmmdefs -M hmm1
phonemodels0
```

6.6. Do this three times, each time putting the re-estimated models in a new directory: `hmm1` (as above), `hmm2`, `hmm3`.

## 7. Recognition Testing

7.1. The Dictionary

7.1.1. The dictionary defines (in alphabetical order) speaker models by their constituent parts, i.e. by each HMM associated with them. For this speaker recogniser each model consists of only one HMM. A word model might consist of several phone states. (For example the word "one" might be defined as "w uh n" – and John's model for the word one might be "jw juh jn"). Similarly, you could include SENT-START and SENT-END can be defined by a silence model ('sil'), but 'sil' need not be included in this simple system. See the tutorial example in the HTK book for more on silence models

7.1.2. Your dictionary – call it `dict` – will look like as below. Ensure there is a carriage return at the end of the file.

```
John            ja
Mary            ma
```

7.2. You also need to create a list of parameterised testing files and a list of models. The former will be of the same format as `training.scp` – call it `testing.scp`:

```
Data/ja4.mfc
Data/ma4.mfc
```

7.3. The latter is similar to `phonemodels0` but each model is enclosed in double quotes; call it `HmmList` and it will contain the following:

```
“ja”
“ma”
```

7.4. Now, having reminded yourself of the role of the grammar, we have all the elements in place to perform speaker recognition. Our recognised labels will be output to file `recout.mlf` when we carry out the recognition using `HVite` (which performs recognition using the Viterbi algorithm):

```
HVite -H hmm3/macros -H hmm3/hmmdefs -S testing.scp
-i recout.mlf -w wdnet dict HmmList
```

## 8. Results

8.1. We now want to gauge the accuracy of the recognition.

8.2. List the contents of the test files. The former will be similar in format to `speakertrainmodels0.mlf` except the models listed are at the speaker level rather than the phone level. Call it `speakertestmodels0.mlf`.

```
#!MLF!#
"*/ja4.lab"
John
.
"*/ma4.lab"
Mary
```

8.3. HResults is then used:

```
HResults -I speakertestmodels0.mlf HmmList
recout.mlf
```

8.4. Look up the HTK book to see how to:

8.4.1. interpret the results;

8.4.2. output a confusion matrix.

## 9. Gaussian Mixture Models

9.1. The above steps have allowed us to create a HMM for each speaker. HMMs model the speech features as a left-right sequence through time. This is also

the approach taken in speech recognition where the goal is to recognise the words spoken rather than the speaker. While HMMs can be used for speaker recognition – allowing us to model how a particular speaker produces a particular word or phone – more often a short cut is taken, where modelling the sequence of phones is dispensed with. Instead all of the characteristics of a speaker a bundled into a single state! As each state is modelled by a Gaussian distribution, this might seem like we are asking a single Gaussian to model all the sounds produced by a speaker. That would be true if we only allowed a single Gaussian to do the job. However we use Gaussian mixtures to model the wide probabilistic landscape we would expect for all the sounds that a particular speaker produces.

9.2. Your task now is to rebuild the speaker recogniser above, but using a Gaussian classifier. A command that will be useful is `HHEd` which allows us to edit the configuration of an existing model. While you could tie the distributions of your 3-emitting-state speaker models above, it may be simpler to simply start again, this time using only a single state for each speaker model.

9.3. The format of the HHEd command is as follows:

```
HHEd -H hmmdefs -H MMF2 ... -M newdir cmds.hed hmmlist
```

`The cmds.hed` file will contain the specific edit to perform on the specified models. For example, to increase the number of mixtures the syntax is "`MU` n itemList", e.g.:

```
MU 3 {*.state[2-4].mix}
```

See the HTK Book for more details. Note that is usually advisable to increase mixtures incrementally, e.g. 1, 2, 4, 8, …512 rather than splitting into 512 mixtures immediately.

9.4. Evaluate the performance of your Gaussian classifier. How does it compare with using HMMs?