# Masters in Computer Speech Text and Internet Technology

# Module: Speech Practical

# HMM-based Speech Recognition

## 1 Introduction

This practical is concerned with phone-based continuous speech recognition using Hidden Markov Models (HMMs). The aim is to train a set of speaker independent models on a subset of the TIMIT Acoustic-Phonetic speech database and then measure the performance on a second different subset. The software required to train the models is provided but the recogniser is to be written by you in C++. The practical should allow you to experiment with some of the ideas presented in the Speech Processing I and Speech Processing II modules. You will find it helpful to refer to your notes for these modules when doing the practical and writing up.

Initially a small set of seven broad class phone models will be used, rather than a full phone inventory, for the experiments. The base-line recognition performance achieved will not be outstanding, hence the final part of the practical involves investigating possible approaches to improving the performance of the basic recogniser.

The remainder of this document is organised as follows. Section 2 outlines the theory of speech recognition using HMMs and gives an outline algorithm for implementing a connected phone recogniser. Section 3 gives a brief introduction to the HTK Hidden Markov Model Toolkit and explains how to use the main tools for training HMMs. Section 4 then describes a set of C++ classes that act as the interface to the HTK C library module. This can be used to simplify the construction of your phone recogniser. Section 5 describes the detailed procedure to follow in the practical. Finally, section 6 gives some information about the practical write-up.

## 2 Speech Recognition using HMMs

An HMM consists of a set of states connected by arcs (see Figure 1). Each arc from state $i$ to state $j$ has an associated transition probability $a_{ij}$. Each internal state $j$ has an associated output probability density function $b_j(\mathbf{o})$ denoting the probability density of generating speech vector $\mathbf{o}$ in state $j$. The entry and exit states do not generate vectors and are called confluent states. Each HMM is a simple statistical model of speech production. It is assumed that the HMM makes one state transition per time period and each time it enters a new internal state, it randomly generates a speech vector. Thus, in Figure 1, the HMM has been in states $S = \{1, 2, 2, 3, 4, 5\}$ and has generated vectors $\mathbf{O} = \{\mathbf{o}_1, \mathbf{o}_2, \mathbf{o}_3, \mathbf{o}_4\}$. The likelihood of generating this sequence is

$$p(\mathbf{O}, S) = a_{12}\ b_2(\mathbf{o}_1)\ a_{22}\ b_2(\mathbf{o}_2)\ a_{23}\ b_3(\mathbf{o}_3)\ a_{34}\ b_4(\mathbf{o}_4)\ a_{45} \tag{1}$$

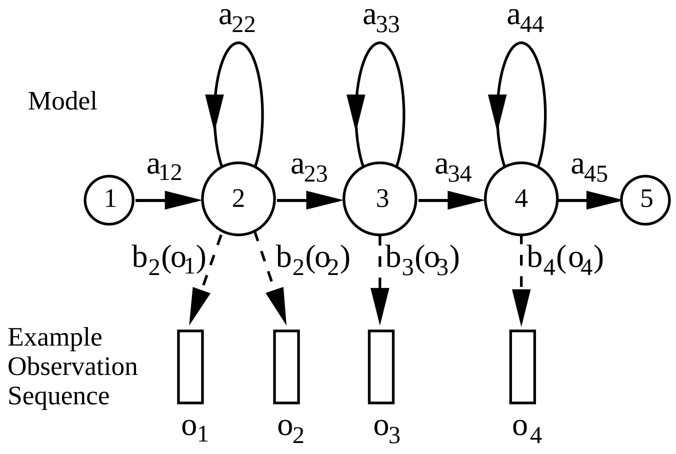


Figure 1: Basic HMM Structure

The output probability density functions are represented by multivariate Gaussians so that if vector $\mathbf{o}$ contains components $v_1$ to $o_D$ then

$$b_j(\mathbf{o}) = \prod_{d=1}^{D} \frac{1}{\sqrt{2\pi\sigma_{jd}^2}} \exp\left(-\frac{(o_d - \mu_{jd})^2}{2\sigma_{jd}^2}\right) \tag{2}$$

In this equation, $\mu_{jd}$ and $\sigma_{jd}^2$ are the mean and variance of the $d$'th component of the speech vectors generated by the $j$'th state. Thus, the vector of means associated with a given state can be regarded as the *typical* speech vector modelled by that state. Notice in (2) that the individual components of $\mathbf{o}$ are assumed to be statistically independent.[1] A *diagonal covariance matrix* is used.

In an HMM based speech recogniser, a single HMM is used to represent each linguistic unit (word, syllable, phoneme etc.). Recognition of a single isolated unit then depends on finding, for each HMM, the maximum likelihood of that HMM generating the unknown sequence of speech vectors. The HMM with the highest such likelihood then identifies the unit. In principle, the maximum likelihood could be found by trying every possible state sequence and using a formula of the form of (1). However, this would be hopelessly inefficient and fortunately there is a better way. Let $\Phi_j(t)$ represent the maximum likelihood of matching some given model against speech vectors $\mathbf{o}_1$ to $\mathbf{o}_t$ of the unknown utterance such that the state sequence used to obtain this likelihood starts at state 1 and ends in state $j$. If $\Phi_j(t-1)$ is known for all $j$, then $\Phi_j(t)$ can be found for $1 < j < N$ and $1 \leq t \leq T$ by the recursion

$$\Phi_j(t) = \max_{1 \leq i < N} \{\Phi_i(t-1)a_{ij}\}\, b_j(\mathbf{o}_t) \tag{3}$$

where $\Phi_1(0) = 1$, $\Phi_1(t) = 0$, $\Phi_i(0) = 0\ \forall i \neq 1$ and T is the total number of frames in the unknown speech. After evaluating (3) for frames 1 through to $T$, the required

[1]In practice, the single Gaussian distribution given by equation 2 cannot accurately model the distribution of real speech. Hence, mixture Gaussians are often used where the probability density of a vector is computed as the weighted sum of single Gaussians.

maximum value can be computed by

$$\Phi_{max} = \max_{1<i<N} \{\Phi_i(T) a_{iN}\} \tag{4}$$

In practice the long sequence of products implied by (3) would quickly lead to arithmetic under-flow, hence log values are used so that all multiplications are replaced by additions. Let $S_j(t) = \log(\Phi_j(t))$ then (3) and (4) above may be rewritten as

$$S_j(t) = \max_{1\leq i<N} \{S_i(t-1) + \log(a_{ij})\} + \log(b_j(\mathbf{o}_t)) \tag{5}$$

and

$$S_{max} = \max_{1<i<N} \{S_i(T) + \log(a_{iN})\} \tag{6}$$

Equations (5) and (6) provide the basis of all HMM recognisers.

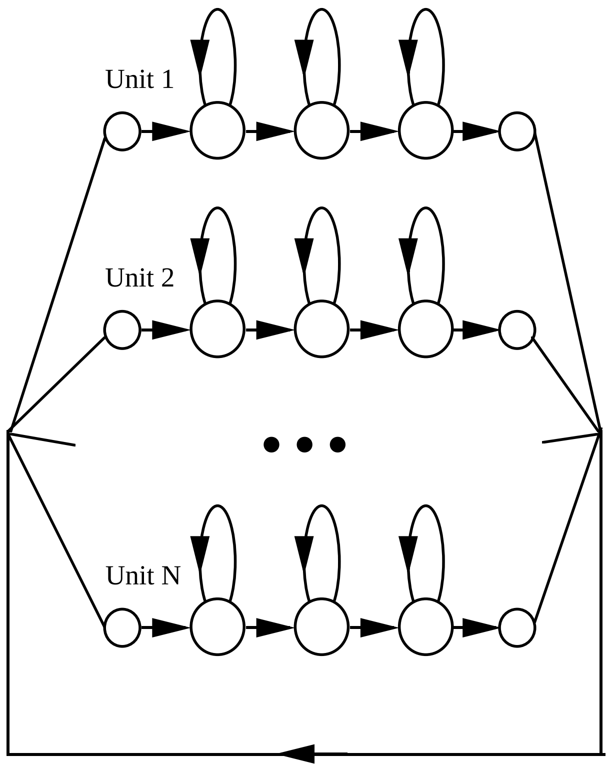


Figure 2: Combining HMMs for Connected Unit Recognition

Given the structure of the basic HMM shown in Figure 1, it should be clear that recognition of continuous speech can be achieved by simply connecting all of the HMMs together into a loop as shown in Figure 2. However, the direct extension of equations (5) and (6) to deal with this case becomes complicated due to the need to determine not just the maximum likelihood but also the sequence of models which gives that maximum. Furthermore, if it is wished to improve performance by restricting the set of allowable HMM sequences (e.g. by the application of phonotactic or syntactic knowledge), then things can get very complicated indeed.

Fortunately, implementation of connected unit recognisers is made much simpler by viewing equations (5) and (6) in a rather different light. Let the recogniser be based on the actual HMM structure and let each state of the model hold a single movable token. Each token q contains two pieces of information

|  |  |
|---|---|
| q.prob$_j$ - | a log likelihood corresponding to $S_j(t)$ in equation(5) |
| q.start$_j$ - | the index of the speech frame at which the token in state $j$ entered the model (only needed in connected unit case) |

Equations (5) and (6) above are now replaced by the following algorithm.

```
Initialise:
        Store a token with q.prob = 0 in state 1 of every model;
        Store a token with q.prob = $-\infty$ i.e. (log(0)) in all other states

Algorithm:

        for (each frame t = 1; t <= T; t++) {
            for (each model m = 0; m < M; m++) {
                for (each model state j = 2; j < N; j++) {
                    find the predecessor state i in model m for
                    which q.prob$_i$ + log($a_{ij}$) is maximum;
                    copy that token into state j
                    incrementing q.prob$_j$ by log($a_{ij}$) + log($b_j(\mathbf{o}_t)$)
                }

                for (model state N) {
                    find the predecessor state i in model m for
                    which q.prob$_i$ + log($a_{iN}$) is maximum;
                    copy that token into state N
                    incrementing q.prob$_N$ by log($a_{iN}$)
                }
            }
            †
        }
```

It is very important to note that the propagation of tokens in the above algorithm must be done *simultaneously* for all states and your implementation of the above outline code needs to ensure this. On completion, the model holding the token in state N with the highest likelihood identifies the unknown speech.

Notice that states 1 and N have to be treated differently since they are not real HMM states but are necessary to allow a set of models to be *glued* together. One check you should apply when implementing this algorithm is to make sure that all token scores are computed by summing exactly the same number of log output probability density values (i.e. one per time frame).

The utility of the above *token passing* algorithm is that it extends naturally to the connected model case by simply allowing the best token in state N of any model to propagate around the loop into state 1 of all the connected models. At the same time, the identity of the model which generated that token and its start value must be recorded in an array. To do this define:

w.model[t] - the identity of model generating best token at frame t
w.start[t] - value of t.start for that token

and add the following additional code at the point marked † in the preceding token passing algorithm.

```
find the model m holding the token q
in state N with the highest probability;
w.model[t] = m;
w.start[t] = q.start;
q.start = t + 1; q.prob = q.prob + P;
copy q into state 1 of all models
```

where P, the *insertion penalty*, is a constant inter-model *log* transition probability which can be arbitrarily adjusted to control the rate of insertion errors. On completion, the identity of the most probable sequence can be determined by tracing back through the w array since

| | |
|---|---|
| w.model[T] | holds the identity of the last HMM |
| w.model[w.start[T]-1] | holds the identity of the penultimate HMM |
| etc | |

The description of the connected unit token passing algorithm is now complete. A fuller description is given in the HTK Book [1]. It is recommended that you implement the speech recogniser for this practical using this token passing paradigm.

## 3 The HTK Toolkit

The HTK Toolkit consists of two main parts: a library of useful routines for accessing speech files, label files, HMM definitions etc; and a set of tools (i.e programs) for building and testing HMM recognisers [2]. In this practical, you will use the HTK tools to edit label files, train HMMs and analyse the performance of your recogniser. You will also use the library via the specially constructed C++ interface classes described in the next section. Descriptions of the tools referred to in this section are given in the HTK Book [1]. There is an on-line HTML version of the HTK Book which can be accessed by typing

```
netscape file://localhost/usr/cstit/speech2/docs/htkbook/htkbook.html
```

or (if registered)

```
netscape http://htk.eng.cam.ac.uk/prot-docs/HTKBook/htkbook.html
```

[2]General information about HTK can be found at the HTK website `http://htk.eng.cam.ac.uk`

HTK Tools manipulate three types of files: label files, speech data files and HMM definition files. All tools are invoked by typing the name of the tool followed by options and arguments. Typing just the name of the tool returns brief usage information (for options and arguments) for all HTK tools. Additional control of HTK tools is provided by setting parameters in a configuration file. For example if the file `config` contained the following

```
SOURCEFORMAT = HTK
```

then typing the command

```
HTool -C config ...
```

would cause HTool to set the variable `SOURCEFORMAT` to the string `HTK`.

Label files are text-based and in their simplest form consist of a sequence of lines in the form

```
<start-time> <end-time> <label>
```

where `<start-time>` and `<end-time>` denote the start and end times of `<label>` in 100ns units. The HTK tool `HLEd` can be made to transform a label file by writing a series of simple edit commands.[3] For example, suppose the file `edt` contains the following

```
RE C p k t d g b
RE V iy eh ae uh
```

then the command

```
HLEd -i bclabs.mlf -l '*' -n broad.lst edt a b c ...
```

would process each label file `a b c` ... and first sort all labels into time order, then replace all occurrences of p, k, t, d, g, and b by C and finally replace all occurrences of iy, eh, ae and uh by V. The new edited files are stored in a single composite label file called a *master label file* (MLF). The `-l` option makes the names of the label files embedded in `bclabs.mlf` independent of the absolute location of the speech data files that they refer to. In addition to the output MLF file, the file `broad.lst` is created containing a list of all newly created labels (just C and V in this case). The full set of edit commands is given in the HTK Book. Note that when processing TIMIT label files, `HLEd` must be told that its input files are not in standard HTK format via the `-G` switch. Also, it is convenient to list all of the label file names to be edited and store them in a single so-called *script* file referenced by the `-S` option. Thus, in practice, HLEd would be invoked as

```
HLEd -G TIMIT -i bclabs.mlf -l '*' -n broad.lst -S labs.scp edt
```

where `labs.scp` is the name of script file. Note that filenames in a script file are not subject to filename expansion[4] completion and full path names are typically used.

Hidden Markov Models are created initially using `HInit` which takes as input a *prototype HMM definition* and a set of training data files and produces as output a

[3]The full list of HLEd commands is displayed by `HLEd -Q`, and a full explanation is in the HTK Book.
[4]Filename expansion is performed for command line arguments by the shell.

new initialised HMM. For example, the following would create a new HMM called `C` in directory `hmm0` from prototype `f24dm1` by scanning all of the supplied data files and extracting all segments of speech labelled by `C`[5].

```
HInit -l C -M hmm0 -o C -I bclabs.mlf -S train.scp f24dm1
```

where `train.scp` holds a list of all the training files.

Once the initial HMMs have been built, the HTK tool `HRest` can be used to re-estimate their parameters using the Baum-Welch algorithm

```
HRest -l C -M hmm1 -I bclabs.mlf -S train.scp hmm0/C
```

which would create a new version of C in directory `hmm1`. This improves the training set likelihood and sometimes improves recognition results.

The final stage of HMM training is to avoid using the hand-produced label boundaries by a technique known as embedded training. This is done by `HERest` which simultaneously re-estimates the parameters of a full set of HMM definitions. For example, the command

```
HERest -d hmm1 -M hmm2 -I bclabs.mlf -S train.scp broad.lst
```

will re-estimate all the models listed in `broad.lst` and stored in `hmm1` and output the new models to `hmm2`. Note that whereas `HRest` repeats the re-estimation procedure repeatedly until convergence is reached, `HERest` performs just one cycle of re-estimation. By default `HERest` merges all output HMMs into a single file called a master macro file (MMF). This can be prevented by setting the configuration variable `KEEPDISTINCT` to true[6].

The HMMs created by the above tools should be tested by using them to recognise the test sentences using your own phone recogniser. The output of this recogniser should be a label file in standard HTK format as described earlier. When all of the test files have been processed, the tool `HResults` can be used to compare them with the hand-produced labels derived from the TIMIT `.phn` files. `HResults` uses a dynamic programming based string alignment procedure to first find the minimum error string alignment and then accumulates statistics on substitutions, deletions and insertion errors.

Assuming that all of the label files output by the recogniser are stored in a directory called `rec` and the hand-produced labels are in the MLF `bclabs.mlf` as before, then the command.

```
HResults -I bclabs.mlf broad.lst rec/*
```

will output the percentage number of correct labels in the recogniser output files and the percentage accuracy. Normally when optimising an HMM system the percentage accuracy is maximised (this includes the effect of insertions while % correct does not). `HResults` can also generate a confusion matrix by including the `-p` option and prints aligned transcriptions of each recognised file by including the `-t` option.

Finally, the contents of the speech data files can be examined using the tool `HList` (see the HTK Book for details.) Note also that HMM definition files and label files can be examined using the standard UNIX commands `cat` and `more`.

[5]HTK does not automatically generate directories. If you want to run this command you will need to generate directory `hmm0`. A description of where the prototypes are is in section 5.2.

[6]Add the line `KEEPDISTINCT=T` to the configuration file `config` and include `-C config` on the command line.

## 4 The HTK Interface Classes

As explained in the previous section, HTK tools manipulate three types of file: labels, speech data and HMM definitions. To build an HMM recogniser you only need access to speech data and the HMM definitions, the C++ interface classes provide this. A full listing of the class definitions are given in Appendix B. Here basic use of the classes is explained.

Before using classes, the HTK library must be initialised by calling the function `InitHTK`. This takes as arguments the standard C command line parameters `argc` and `argv`. The latter can then be accessed sequentially using the function `NextArg` to determine the type of the next command line argument (string, integer, option switch or float) and the functions `GetXXXArg()` to actually read each argument. Thus, the structure of a program that uses the interface classes is as follows

```
int main(int argc, char *argv[])
{
  int iarg;
  char *sarg;

  InitHTK(argc,argv);
  if (NumArgs() == 0)         /* program expects some arguments */
    ReportUsage();
  if (NextArg() != STRINGARG) /* first arg is a string */
    HError(1,"HTool: string arg expected");
  sarg = GetStrArg();
  if (NextArg() != INTARG)    /* second arg should be an integer */
    HError(1,"HTool: integer arg expected");
  iarg = GetIntArg();

  ...

  return 0;
}
```

A set of HMM definitions is represented internally within a program by an instance of the class `HMMSystem`. A set of HMMs is read in by using the method `loadHMMSystem`. This method takes as arguments the name of the directory holding the models and the name of a file containing a list of all the model names. Once an HMM set has been loaded, individual HMMs can be accessed using the `HMMModel` class. A model may be extracted using the `getModel` method. The `HMMModel` class gives access to the model parameters using a variety of methods. The following program fragment illustrates the main ideas. The code is also available in `Demo1.cc`.

```
char *hlist, *hdir;
HMMSystem hsys;
HMMModel hmm;
HMMVector mean,var;
HMMMatrix mat;
int nmod,i,j,N,vsize;

InitHTK(argc,argv);
if (!InfoPrinted() && NumArgs() == 0)
  ReportUsage();
if (NumArgs() == 0) Exit(0);
if (NextArg() != STRINGARG)
  HError(1,"HMM directory expected");
hdir = GetStrArg();
if (NextArg() != STRINGARG)
  HError(1,"HMM list expected");
hlist = GetStrArg();

// load HMM system
hsys.loadHMMSystem(hdir,hlist);
cout << "HMMs loaded\n" << endl;
// get global HMM set info
cout << "Accessing individual models" << endl;
nmod = hsys.numModels();    // total number of HMMs
vsize = hsys.sizeVector();  // observation dimension
mean.setSize(vsize);    // vectors to hold mean
i = 0;
// get the i'th model: model numbers indexed from 0
hsys.getModel(i,hmm);
N = hmm.numStates();
j = 2;
// get the mean and the variance from the j'th state
// assuming 1 mixture component per state
hmm.getMean(j,1,mean);
// print the mean of state i of model j
for (i=1; i <= vsize; i++)
  printf("%d. %f\n",i,mean[i]);

// Or use the built in functions
if (hmm.getCovKind(j,1) == DIAGC) {
  var.setSize(vsize);              // allocate variance
  hmm.getVariance(j,1,var);
  var.showVector("Variance");
} else if (hmm.getCovKind(j,1) == FULLC) {
  mat.setSize(vsize);              // allocate covariance
  hmm.getCovMatrix(j,1,mat);
```

```
    mat.showMatrix("Covariance Matrix");

    // and just the leading diagonal
    for (i=1; i <= vsize; i++)
      printf("%d. %f\n",i,mat[i][i]);

  }
```

The instance of class `HMMVector` gives access to an HTK array of $N$ floats indexed 1 to $N$. Each instance must be created and initialised explicitly using the `setSize` method as shown.

Speech data is accessed via the `HMMParmBuf` class. A speech file is loaded using the `loadSpeech` method. Once loaded, individual speech frames are extracted using the method `getVector`. The following fragment illustrates the main ideas. The code is also available in `Demo2.cc`.

```
  char *spf;         // name of speech file
  HMMParmBuf p;      // parameter buffer
  int n,i,j,vsize;
  HMMVector v;

  InitHTK(argc,argv);
  if (NextArg() != STRINGARG)
    HError(1,"Speech file name expected");
  spf = GetStrArg();
  p.loadSpeech(spf);     // load file of speech vectors
  n = p.numVectors();    // total num vectors
  vsize =p.sizeVector(); // size of each vector
  v.setSize(vsize);
  for (i=0; i<n; i++) {
    p.getVector(i,v);
    cout << i << " : ";
    for (j=1; j<=vsize; j++)
      cout << v[j] << " ";
    cout << endl;
  }
  p.unloadSpeech();      // delete loaded speech from memory
```

Finally, given an instance of class HMMModel `h` and a frame of speech data `v`, the method

```
h.getTrans(i,j)
```

returns the log probability of making a transition from state `i` to state `j` of model `h` (i.e $\log(a_{ij})$) and

```
h.getOProb(v,j)
```

returns the *log* probability density of generating vector `v` in state `j` of model `h` (i.e. $\log(b_j(\mathbf{o}))$).

# 5 Practical

## 5.1 Timescale

The practical has two parts. The timescale for the practical is

| Weeks | Task |
|---|---|
| **1-4** | Basic models and Viterbi decoder design |
| **5-8** | System refinement |

After writing the basic Viterbi decoder you should ensure that you get the same results as the supplied executable. When you have confirmed the results are the same get the code checked by a demonstrator. If you don't think you will completed this part of the practical in the allotted time talk to the demonstrator at the start of week 4 and ensure that a working version is completed *before* the start of week 5.

## 5.2 Files and Directories

The following files are supplied for this practical. They may be found in the directory `/usr/cstit/speech2` on the system.

(a) `data` - a directory containing two subdirectories each of which holds parameterised speech data files derived from TIMIT dialect region 5 (Southern). A fuller description of the database is in the `docs` directory detailed below.

- train - 70 speakers x 8 sentences
- test - 28 speakers x 8 sentences

Each sentence is stored in a HTK format data file encoded as a sequence of parameter vectors (frames). Each frame consists of 24 log filterbank amplitudes. The frame period is 10msecs.

(b) `labs` - a directory containing two subdirectories each of which holds TIMIT label files for the data files stored in `data`.

(c) `word` - a directory containing two subdirectories each of which holds the orthographic transcriptions for the data files stored in `data`.

(d) `src` - a directory containing the HTK interface class definitions described in section 4. It also contains an outline structure for a suitable recogniser program in the file `PRec.cc` and a `makefile`. In addition, the two demo files are in this directory.

(e) `hmm` - a directory holding the following example prototype hmm definitions.

- `f24dm1` - basic single Gaussian HMM
- `f24fm1` - 1 single full covariance Gaussian HMM

(f) `edfiles` - a directory holding the following example hmm and label edit files

- `mu.hed` - increases the number of components of all states to 2
- `fc.hed` - converts all covariance matrices from diagonal to full covariance matrices
- `broad.led` - converts labels from the 61 TIMIT phone set to the 7 broad classes
- `phone.led` - converts labels from the 61 phone TIMIT set to a 45 phone set[7]
- `left.led` - converts context independent labels to left context dependent labels

**(g)** `docs` - a directory containing documentation

- `handout.[ps,pdf]` - postscript handout related to the practical
- `timitspkrs.txt` - text document describing the speakers
- `timitdic.txt` - text document describing the dictionary used
- `timitmap.txt` - text document describing the mapping from timit speaker name to filename used in this practical.

**(h)** `flists` - a directory containing lists of the training and test data

- `train.scp` - the 560 training sentences
- `test.scp` - the 224 test sentences

**(i)** `scripts` - a directory containing an example script.

Note that to run any HTK tool you must add `/usr/cstit/speech2/htk/bin/bin.linux` to your `PATH` variable (normally set in your `.profile`).

You may find it useful to use *shell scripts* [8] to perform repeated operations and/or run experiments for this practical. If you are unfamiliar with the capabilities/syntax of shell scripts an example is given in the `scripts` directory.

## 5.3 Basic Models and Viterbi Decoder Design

This section is concerned with writing a simple decoder and examining the performance of a basic broad class HMM system.

1. Examine the list of phones used in TIMIT (see Appendix A). The label edit file `broad.led` will be used to map the TIMIT phone labels one of the following broad classes.

| | | |
|---|---|---|
| V | - | vowel |
| L | - | liquid or glide |
| C | - | stop consonant |
| F | - | weak fricative |
| Z | - | strong fricative |
| N | - | nasal |
| S | - | silence |

[7]Feel free to use other mappings if you want. If you do ensure that the mapping is consistent with the broad class mapping for scoring purposes.

[8]see `http://www-h.eng.cam.ac.uk/help/tpl/unix/scripts/scripts.html` for help on shell scripts

Look at the edit file `broad.led` and check that you understand the operations of this edit file.

2. Use the HTK tool `HLEd` with the `broad.led` label edit file to convert the given TIMIT label files into broad class transcriptions.
3. Use `HInit` and the HMM prototype `f24dm1` to create an initial set of broad class HMMs.
4. Design and implement a continuous recogniser using the supplied classes. You may use the outline supplied in `PRec.cc` if you wish or you can start from scratch. The majority of the code to be changed, if the outline is used, is in the `HMMViterbi` class definition. The basic algorithm to use is described in section 2. A complete compiled version of the program can be found in `PRecX`.
5. Test the initial set of HMMs using your recogniser. You should examine the performance of the recogniser on both the test data and the training data. Ensure that the insertion penalty is appropriately set. Comment on the performance of the models on the two sets of data. Try and improve the basic models using `HRest`.
6. Perform four iterations of embedded re-estimation using `HERest`. Monitor the recognition performance improvement at each iteration. Produce a confusion matrix for the final models. Attempt to explain the major confusions made by your recogniser

**After you have completed this section of the practical contact a demonstrator who will check that your decoder implementation is correct.**

### 5.4 System Refinement

The aim of this part of the practical is to refine the performance of the broad class classifier. There are a number of techniques that you are required to investigate. You are expected to design a "good" recogniser by combining techniques, rather than simply implementing them individually. In the time allowed it is not possible to investigate all possible combinations, so intelligent selection (based on reading and experiments) of appropriate combinations is required. For example you may consider the number of model parameters and the issue of generalising from the training data to test data.

**It is more important to show that you know how to go about designing a good system, rather than purely obtaining the lowest error rate.**

The section has two parts. The first is the *initial refinement* section. This will make use of some standard HTK tools. This section should not require any modifications to the recogniser. All sections of this part of the practical should be examined. In addition there is the *extended refinement* part of the practical. You are required to investigate *one* of the extended refinements. Again it is more important to fully investigate an option rather than simply obtaining the lowest error rate.

Some of the systems that can be built will take relatively large amounts of memory. This may result in you exceeding your disk quota on the teaching system (`150Mb`). This can be avoided by deleting (use `rm`) old model sets, and intermediary model sets, that you don't need anymore. You can also make use of temporary disk space by creating a directory on `/local/scratch` using

```
sudo mkscratchdir
```

Files stored in `/local/scratch` and its subdirectories are stored locally to the machine you are working on at the time - each cstit workstation has its own separate `/local/scratch` space. Files stored there are deleted when a workstation reboots, and may be deleted automatically if left untouched for several days. This should *not* be used for permanent storage of important material!

1) **Initial Refinement**

   You should investigate *all* topics in the initial refinement section.

   (a) **Acoustic Model Units**

   - **Phone Models**: rather than using the broad classes, phone models may be used. Using the `phone.led` edit file convert the 61 TIMIT phone set into a 45 phone set. Retrain the models and find the phone performance on the training and test data. Either by mapping the label files or using the `-e` option in `HResults` obtain the equivalent broad class performance.

   (b) **Front-End Processing**

   - **Cepstral** parameters: the HMMs used in this practical assume that the components of each speech vector are statistically independent, hence diagonal covariance matrices are used. For filter bank data this is a very poor assumption. HTK tools can automatically convert the supplied filter bank format to MFCC (Mel Frequency Cepstral Coefficient) format. To do this, create a new HMM prototype with `MFCC_E` set instead of `FBANK_E` and then set the configuration variable `TARGETKIND` to `MFCC_E`. The `_E` qualifier here indicates that overall frame energy is appended to the parameter vector. You may need to alter the size of the mean and variance vectors. You can also vary the dimensionality of the converted vectors by setting the environment variable `NUMCEPS` (default value 12) to the required value. Note that this will work with your recogniser too since it uses the same HTK input facilities as the standard HTK tools. The vector lengths in the prototype HMM definition must be appropriate for the final feature vector used rather than the input vector. Investigate how the performance changes as the number of Cepstral parameters varies.
   - **Delta** parameters: as well as automatic conversions, HTK tools can append so-called delta coefficients to the input vector. You do this by adding `_D` to the the sample kind in the prototype HMM definition, and also to the value of `TARGETKIND`. The vector lengths in the prototype

HMM definition must be appropriate for the final converted vector rather than the input vector. For example, if you set the sample kind to `MFCC_E_D` and `NUMCEPS` to 8, then the final vector size would be 18 (8 MFCC coefficients, 1 energy coefficient, 8 delta MFCC coefficients and 1 delta energy coefficient). Delta-delta parameters may also be added using the `_A` qualifier. Investigate how the performance on the training and test data changes when delta and delta-delta parameters are used.

**(c) Model Structure**

- **Gaussian Mixture Models**: the basic system uses a single Gaussian component to model the data. The number of components may be increased using the `HHEd` command. An example edit file to do this is in `mu.hed`. You should check that you understand what this command does to the model set (details in the HTK Book). After increasing the number of components perform multiple iterations of `HERest` should be performed to estimate the multiple component model parameters. Increase the number of components investigating the training and test data performance at each stage. Examine how the log-likelihood of the training data varies throughout the training process (use `-T 1` in the `HERest` command line). For more complex system, it is also useful to use *pruning* during training. This can be switched on using, for example, `-t 200.0 200.0 1000.0` in the `HERest` command line.
  When training Gaussian mixture models the standard HTK *mixing-up* scheme should be used. Here the number of Gaussian components in the system is gradually increased. An example training routine might be:

  **(i)** Train a single component system;
  **(ii)** Using `HHEd` increase the number of Gaussian components per state to 2;
  **(iii)** Perform 4 iterations of Baum-Welch training using `HERest`;
  **(iv)** Using `HHEd` increase the number of Gaussian components per state to 4;
  **(v)** Perform 4 iterations of Baum-Welch training using `HERest`;
  **(vi)** Using `HHEd` increase the number of Gaussian components per state to 6;
  **(vii)** Perform 4 iterations of Baum-Welch training using `HERest`.

  You should keep increasing the number of Gaussian components per state until you feel that you have fully investigated the system.
  The training data log-likelihood should be checked after each run of `HERest` and plotted for the write-up. Comment on the shape of the graph.

**2) Extended Refinement**

You should investigate *one* topic in the extended refinement section.

**(a) Acoustic Model Units**

- **Context Dependent Models**: use `HLEd` to convert the label files to be left-context dependent (or some other context dependency of your choice). Use the HMM editor `HHEd` for this. For example if you want to add left context to the broad labels use

  ```
  HLEd -l '*' -n bcleft.lst -i bcleft.mlf edfiles/left.led bclabs.mlf
  ```

  This converts the broad labels in MLF `bclabs.mlf` into left context labels, stored in `bcleft.mlf`, and produces a list of the left context models in `bleft.lst`. If desired, the set of broad HMMs can then be cloned using the `HHEd` command and the `CL` command. Finally, use `HERest` to re-estimate the parameters of the new enlarged model set and retest on the training and test data. Note that the `-s` option should be used with `HResults` to ensure that the context dependent part of the labels is ignored.

  Two sets of experiments should be run.

  – Any sequence of context dependent models are allowed. This requires no alterations to the search. For example a sequence could be

  ```
  S S-V C-V L-S
  ```

  – Restrict sequences so that only valid contexts can follow. This requires modifying the recogniser so that only consistent sequences of models are allowed. For example

  ```
  S S-V V-C C-S
  ```

  Note: HTK supports both left and right contexts as well as triphone models.

**(b) Front-End Processing** These extensions require no modifications to the decoder, but do require a little knowledge of `matlab`.

- **Full Covariance Matrices**: rather than using diagonal covariance matrices full covariance matrices may be used. This is an interesting contrast to the global linear transforms investigated in this section. Full covariance matrices may be considered as Gaussian component specific linear transforms. Models may be converted to full covariance models using the `HHEd` command with `fc.hed` edit file[9] or using the `f24fm1` HMM prototype.
- **Principal Component Analysis**: rather than using a standard frontend PCA may be used to generate a frontend tuned to the training data (see the Speech Processing I module notes for details). Using a full covariance prototype with the HTK tool `HCompV` command obtain the covariance matrix of the training data from the model file. The eigenvalues and eigenvectors may be obtained using `matlab` and the

[9]This command is an extension to the standard HTK `HHEd` commands.

`eig` command. Once the appropriate linear transform has been found it may be applied to the data using the frontend linear transform extension to HTK described in Appendix C

- **Linear Discriminant Analysis**: there are well known issues with using PCA for feature selection. LDA overcomes many of these difficulties. As described in Appendix D an estimate of the average within class covariance matrix $\mathbf{W}$ and between class covariance matrix $\mathbf{B}$ are required. These estimates may be obtained by training a full-covariance matrix system. A new program needs to be written to obtain these matrices from a trained full covariance matrix system. The code in `Demo1.cc` may be a useful place to start.

  Two forms of estimates may be obtained.

  – It may be assumed that every state/component occurs equally often. The within and between class covariances are then computed directly from the model parameters.

  – The counts are weighted by the actual occupancies. The occupancies for each state can be obtained by using the `-s filename` flag when running `HERest` (see HTK Book for further details).

  If time allows both estimates should be examined.

**(c) Language Model**

- **Unigram Language Model**: the basic system assumes that all of the broad classes/phones occur equally often. In practice this is not true. Using the training data estimate the unigram probabilities for each of the models. Running the command

  ```
  HLStats -d broad.lst bclabs.mlf
  ```

  will give duration statistics and number of occurrences in the MLF `bclabs.mlf` for the all the models in `broad.lst`. From these statistics it is simple to compute the unigram probabilities. Modify the basic Viterbi decoder so that the unigram probabilities are taken into account.

- **Bigram Language Model**: bigram language models may also be used. Again using the training data estimate bigram probabilities (taking into account any data sparsity issues). Again the HTK tool `HLStats` may be used to estimate the bigram probabilities from the training label files. The Viterbi decoder must then be modified to allow the use of this bigram language model.

For the write-up you should comment on any performance differences and comment on the language models. For example by comparing perplexities on the training and test data.

## 6 Write-Up

Your report for this practical should include the following (not necessarily in the order given)

- Brief description of the design of your recogniser. Include the program listing as an appendix (do not include the program listing appendix in the word count).
- Discussion of recogniser performance on the basic broad class models.
- Brief description of the theory behind each of the refinements that you have investigated. This should include why the refinement should help in speech recognition.
- Description of your attempts to improve the system performance. It is important that you describe how you arrived at your final "best" system. This should include any negative results (i.e. combinations of techniques that made the performance worse) if they helped in your choice of system. You should also include whether your results agree with the theory given in the lectures. Be sure to include results from at least one system using each of the required sections (possibly in combination with other techniques).

The report should **not** include a discussion of the basic HMM theory given in this practical handout. Your report should be no more than 6,000 words long.

## References

[1] S.J. Young, J.J. Odell, D.G. Ollason, V. Valtchev & P.C. Woodland *The HTK Book*. Entropic Cambridge Research Laboratory.

S.J. Young, P.C. Woodland, T. Hain
January 1995-2001

M.J.F. Gales
December 2001, 2002, 2004

## A Acoustic Phonetic Symbols used in TIMIT

This appendix describes the phonemic and phonetic symbols used in the TIMIT phonetic transcriptions. These include the following symbols:

1. the closure intervals of stops which are distinguished from the stop release. The closure symbols for the stops `b,d,g,p,t,k` are `bcl,dcl,gcl,pcl,tck,kcl`, respectively. The closure portions of `jh` and `ch`, are `dcl` and `tcl`.
2. allophones that do not occur in the lexicon. The use of a given allophone may be dependent on the speaker, dialect, speaking rate, and phonemic context, among other factors. Since the use of these allophones is difficult to predict, they have not been used in the phonemic transcriptions in the lexicon.
    - flap `dx`, such as in words ”muddy” or ”dirty”
    - nasal flap `nx`, as in ”winner”
    - glottal stop `q`, which may be an allophone of `t`, or may mark an initial vowel or a vowel-vowel boundary
    - voiced-h `hv`, a voiced allophone of `h`, typically found intervocalically
    - fronted-u `ux`, allophone of `uw`, typically found in alveolar context
    - devoiced-schwa `ax-h`, very short, devoiced vowel, typically occurring for reduced vowels surrounded by voiceless consonants
3. other symbols include two types of silence; `pau`, marking a pause, and `epi`, denoting epenthetic silence which is often found between a fricative and a semivowel or nasal, as in ”slow”, and `h#`, used to mark the silence and/or non-speech events found at the beginning and end of the signal.

```
POSSIBLE PHONETIC
SYMBOL    EXAMPLE WORD   TRANSCRIPTION
------    ------------   -------------
Stops:
b          bee           BCL B iy
d          day           DCL D ey
g          gay           GCL G ey
p          pea           PCL P iy
t          tea           TCL T iy
k          key           KCL K iy
dx         muddy, dirty  m ah DX iy, dcl d er DX iy
q          bat           bcl b ae Q
```

| | | |
|---|---|---|
| Affricates: | | |
| jh | joke | DCL JH ow kcl k |
| ch | choke | TCL CH ow kcl k |
| | | |
| Fricatives: | | |
| s | sea | S iy |
| sh | she | SH iy |
| z | zone | Z ow n |
| zh | azure | ae ZH er |
| f | fin | F ih n |
| th | thin | TH ih n |
| v | van | V ae n |
| dh | then | DH e n |
| | | |
| Nasals: | | |
| m | mom | M aa M |
| n | noon | N uw N |
| ng | sing | s ih NG |
| em | bottom | b aa tcl t EM |
| en | button | b ah q EN |
| eng | washington | w aa sh ENG tcl t ax n |
| nx | winner | w ih NX axr |
| | | |
| Semivowels and Glides: | | |
| l | lay | L ey |
| r | ray | R ey |
| w | way | W ey |
| y | yacht | Y aa tcl t |
| hh | hay | HH ey |
| hv | ahead | ax HV eh dcl d |
| el | bottle | bcl b aa tcl t EL |

Vowels:

| | | |
|---|---|---|
| iy | beet | bcl b IY tcl t |
| ih | bit | bcl b IH tcl t |
| eh | bet | bcl b EH tcl t |
| ey | bait | bcl b EY tcl t |
| ae | bat | bcl b AE tcl t |
| aa | bott | bcl b AA tcl t |
| aw | bout | bcl b AW tcl t |
| ay | bite | bcl b AY tcl t |
| ah | but | bcl b AH tcl t |
| ao | bought | bcl b AO tcl t |
| oy | boy | bcl b OY |
| ow | boat | bcl b OW tcl t |
| uh | book | bcl b UH kcl k |
| uw | boot | bcl b UW tcl t |
| ux | toot | tcl t UX tcl t |
| er | bird | bcl b ER dcl d |
| ax | about | AX bcl b aw tcl t |
| ix | debit | dcl d eh bcl b IX tcl t |
| axr | butter | bcl b ah dx AXR |
| ax-h | suspect | s AX-H s pcl p eh kcl k tcl t |

Others:

| SYMBOL | DESCRIPTION |
|---|---|
| pau | pause |
| epi | epenthetic silence |
| h# | begin/end marker (non-speech events) |
| 1 | primary stress marker |
| 2 | secondary stress marker |

## B Class Definitions

### B.1 HMMVector

```
class HMMVector {
public:
  // set of constructors for various initialisations
  HMMVector(const int , MemHeap *);
  HMMVector(const int);
  HMMVector(MemHeap *memheap);
  HMMVector();
  // destructor reclaims memory
  ~HMMVector();

  // element access operators
  float &operator[](const int);
  const float &operator[](const int) const;

  // assignment operators
  const HMMVector &operator=(const HMMVector &);
  const HMMVector &operator=(const Vector);

  // general access functions to members
  const int getSize(); // return number of models
  void setHeap(MemHeap *);
  void setSize(const int);
  Vector getVector();

  // method to print vector to stdout
  void showVector(char *s);

private:
  int size;
  Vector vec;
  MemHeap *mem;
};
```

### B.2 HMMMatrix

```
class HMMMatrix {
public:
  // set of constructors for various initialisations
  HMMMatrix(const int , MemHeap *);
  HMMMatrix(const int);
  HMMMatrix(MemHeap *memheap);
  HMMMatrix();
  // destructor reclaims memory
  ~HMMMatrix();

  // element access operators
  Vector &operator[](const int);
  const Vector &operator[](const int) const;

  // assignment operators
  const HMMMatrix &operator=(const HMMMatrix &);
  const HMMMatrix &operator=(const Matrix);

  // general access functions to members
  const int getSize(); // return number of models
  void setHeap(MemHeap *);
  void setSize(const int);
  Matrix getMatrix();

  // method to print vector to stdout
  void showMatrix(char *s);

private:
  int size;
  Matrix mat;
  MemHeap *mem;
};
```

### B.3 HMMModel

```
class HMMModel {
public:
  // constructor and destructor
  HMMModel();
  ~HMMModel();

  // methods to set member values
  void setModel(MLink);  // set the HTK HMM

  // methods to access aspects of the model
  // the name
  char* getName();
  // the number of states
  const int numStates();
  // the number of components in a state
  const int numMixComps(const int);

  // and extract underlting aspects of the model
  void getMean(const int, const int, HMMVector&);    // the mean
  CovKind getCovKind(int state, int mixcomp);        // the Covariance Kind
  void getVariance(const int, const int, HMMVector&);// the variance
  void getCovMatrix(const int, const int, HMMMatrix&);// the covariance matrix
  float getWeight(const int, const int);             // the weight (prior)
  LogFloat getTrans(const int, const int);           // the transition

  // method to obtain the log-likelihood of a particular vector
  // from the specified state
  LogFloat getOProb(HMMVector&, const int);

private:
  MLink hmm;
  MemHeap *modHeap;
  const void check(int,int);
};
```

## B.4 HMMSystem

```
class HMMSystem {
public:
  // constructor and desctructor
  HMMSystem();
  ~HMMSystem();

  // load the specified model set
  void loadHMMSystem(char *, char *);

  // get information about the system
  int numModels(); // return number of models
  void getModel(const int, HMMModel &);
  int sizeVector();

  // print a summary of the system
  void showSystem(); // print out description of system

private:
  int nHMM;
  MemHeap hmmHeap;
  HMMSet hset;
  MLink *hmm;
};
```

## B.5 HMMViterbi

```
// Viterbi decoder built derived from the generic HMMDecoder

/* ------------------ Global Data Structures ----------------- */

#define MAXMODELS 50 /* max number of models */
#define MAXSTATES 10    /* max states in any model + 1 */
#define MAXFRAMES 1000  /* max number of speech vectors in any file */
#define FPERIOD 100000  /* frame period (HTK uses 100ns time units) */

/* ------------------------------------------------------------ */

class HMMViterbi : public HMMDecoder {
public:
  // method to set member values
  void setInsertPen(const LogFloat);

  // method to define how a viterbi decoder works
  virtual void doRecognition(char *);
  virtual void writeDecision(char *outdir, char *testfile);

private:
  int numPhones;
  int vSize;
  LogFloat insertPen;
  int nFrames;
  int bestp;

  // private methods for doing the decoding and traceback
  void Initialise(void);
  void StepModels(void);
  void PropagateBestToken(int, int);
  int BestExitToken(void);
  void RecordBestToken(int, int);
  void TraceBack(FILE *, int);
  LogFloat getLogLike();
};
```

## B.6 HMMParmBuf

```
class HMMParmBuf {
public:
  // constructor and destructor
  HMMParmBuf();
  ~HMMParmBuf();

  // methods to load (and unload) HTK  speech files
  void loadSpeech(char *);
  void unloadSpeech();

  // methods for accessing the speech data
  int numVectors();
  int sizeVector();
  void getVector(int, HMMVector&);

private:
  MemHeap heap;
  Observation *obs;
  ParmBuf pbuf;
};
```

## C HTK Linear Transform Extension

A linear transform of the frontend can be specified using the

```
HPARM:MATTRANFN = filename.mat
```

configuration variable, where *filename.mat* specifies the filename of the HTK linear transformation matrix. This line should be added to your configuration file. An example of the matrix file is

```
<MATRIX>
<SOURCEKIND> FBANK_E
<NUMINPUT> 4 <NUMOUTPUT> 3
     1.0 0.0 0.0 0.0
     0.0 1.0 0.0 0.0
     0.0 0.0 1.0 0.0
<ENDMATRIX>
```

This takes a frontend of `FBANK_E` of dimensionality 4 and extracts the first three elements of the feature vector. Using the `TARGETKIND` delta and delta-delta parameters may be added[10]. The matrix (rather than it's transpose as described in the speech processing 1) is applied to the data. To apply the transpose the equivalent would be

```
<MATRIX> <TRANSPOSE>
<SOURCEKIND> FBANK_E
<NUMINPUT> 4 <NUMOUTPUT> 3
     1.0 0.0 0.0
     0.0 1.0 0.0
     0.0 0.0 1.0
     0.0 0.0 0.0
<ENDMATRIX>
```

Alternatively the linear transform may be applied *after* the addition of any delta and delta-delta parameters. This then has the form

```
<MATRIX>
<SOURCEKIND> FBANK_E_D  <POSTQUAL>
<NUMINPUT> 8 <NUMOUTPUT> 3
     1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
     0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
     0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0
<ENDMATRIX>
```

This matrix takes a frontend of dimensionality 4 with a `TARGETKIND` of `FBANK_E_D` and extracts the first, fifth and second elements of the feature vector.

[10]If `TARGETKIND` is used to change the basekind, for example from `FBANK` to `MFCC`, this is performed *before* the application of the matrix. It is therefore necessray to ensure that the correct `SOURCEKIND` and input size is specified.

## D Linear Discriminant Analysis

In principal component analysis only considered the global covariance matrix os considered in the orthogonalisation procedure. This has well known issues, for example it is not invariant to scaling. An alternative approach, when there are class labels associated with the training data, is to use linear discriminant analysis. This aims to select features that are better at *discrimination*.

The aim in LDA is to select a set of features that have a small *within-class* variance, $\mathbf{W}$, and a large *between-class* variance, $\mathbf{B}$. We need to find the transformation $\mathbf{A}$ that achieves this. A new system can be trained in this transformed space

$$\tilde{\mathbf{o}}_t = \mathbf{A}'\mathbf{o}_t \tag{7}$$

$\mathbf{A}$ does not need to be a square matrix. The dimensionality of the transformed features may be less than that of the original features. The assumptions behind the use of LDA are:

1. each class can be represented as a single Gaussian, all within-class covariances are the same;
2. the class centroids (means) can be represented by a single Gaussian.

To estimate the transform $\mathbf{A}$ an estimate of the average within class covariance, $\mathbf{W}$, and between class covariance $\mathbf{B}$ are required.

- For the unweighted (all states/components occur equally often)

$$\mathbf{W} = \frac{1}{N_T}\sum_{j=1}^{N_T}\mathbf{\Sigma}_j$$

$$\mathbf{B} = \frac{1}{N_T}\sum_{j=1}^{N_T}(\boldsymbol{\mu}_j - \boldsymbol{\mu}_g)(\boldsymbol{\mu}_j - \boldsymbol{\mu}_g)'$$

  where $N_T$ total number of states and $\boldsymbol{\mu}_g$ is the average of all the means.

- For the weighted estimates

$$\mathbf{W} = \frac{1}{T}\sum_{t=1}^{T}\sum_{j=1}^{N_T} L_j(t)(\mathbf{o}_t - \boldsymbol{\mu}_j)(\mathbf{o}_t - \boldsymbol{\mu}_j)'$$

$$\mathbf{B} = \frac{1}{T}\sum_{t=1}^{T}\sum_{j=1}^{N_T} L_j(t)(\boldsymbol{\mu}_j - \boldsymbol{\mu}_g)(\boldsymbol{\mu}_j - \boldsymbol{\mu}_g)'$$

  where $\boldsymbol{\mu}_g$ is the mean of all the data and $T$ is the total number of frames. Note in this case $\mathbf{T} = \mathbf{W} + \mathbf{B}$ where $\mathbf{T}$ is the total covariance matrix of the data.

The first step is to decorrelate the feature vector space. The within-class covariance matrix is transformed so that the dimensions are independent and have unit variance (transform $\mathbf{U}_w\mathbf{\Lambda}_w^{-\frac{1}{2}}$). For details see the Speech Processing I lecture notes. The between-class covariance is obtained in this new domain.

$$\tilde{\mathbf{B}} = \mathbf{\Lambda}_w^{-\frac{1}{2}}\mathbf{U}'_w\mathbf{B}\mathbf{U}_w\mathbf{\Lambda}_w^{-\frac{1}{2}}$$

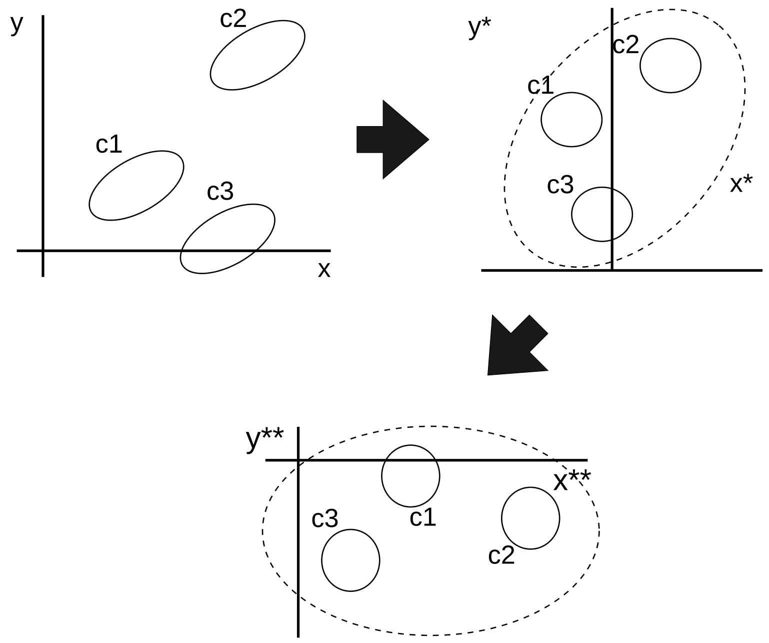
A rotation to diagonalise the between-class covariance is performed (transform $\tilde{\mathbf{U}}_b$). The largest elements of the resulting diagonal covariance matrix are selected. The complete transformation is

$$\mathbf{A} = \mathbf{U}_w\mathbf{\Lambda}_w^{-\frac{1}{2}}\tilde{\mathbf{U}}_b\mathbf{F}$$

**F** is a truncation matrix which selects the dimensions with the largest $d$ eigenvalues. For example **F** may have the form

$$\mathbf{F} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}$$

if the first and third dimensions have the largest eigenvalues.



Take a two dimensional example. Need a single dimension that discriminates classes $c1$, $c2$ and $c3$. From the diagram below, it may be seen that dimension $x^{**}$ is the best in terms of the LDA criterion.