# HTK -- TUTORIAL

F. Schiel    25.06.97 / 18.07.97

Abbreviations in this document:
$DATA = /n/weissbier/xb/schiel/NUMBERS
$SRCE = /n/weissbier/xa/schiel/NUMBERS
$DRSP = /u/drspeech/data/numbers95
$SULIN = ~sulin/speech/project/numbers_cs

If not explicitly given, a file is found or a command is issued in $SRCE.

==========================================================================
==========================================================================

## Contents

==========================================================================

==========================================================================

## PART 1  -  INTRODUCTION TO HTK

The following coarse introduction to HTK will give examples how to set up a basic recognizer for the Numbers_95 task. It includes the steps data preparation, model design, bootstrapping, silence modeling, embedded training and tuning. It does not give all the details of all the described HTK tools. For a more in-depth discussion of the HTK Tools please refer to Part 2 of this tutorial. Also, this part does not discuss the usage of the HTK libraries within user made frameworks.

Intended audience: people who like to know what HTK is and how it feels to use it.

==========================================================================

### 0. General

All HTK commands begin with the letter 'H' and reside in the ESPS installation of ICSI. To use them you must have a proper '.cshrc' to define all needed paths and parameters. An example for a '.cshrc' that should work on all ICSI platforms can be found in .cshrc.icsi

Since we have two floating licenses at the moment, maximal two users are allowed to issue as many HTK commands on as many machines they wish. The first time you issue a command a license is checked out for you. If both licenses are in use, you get a warning and the command is not executed (this can be a hassle in long batches; use 'HCheckout' to verify that a license is available). To free a used license issue the command 'HFree' after you're done.

In the following examples the issued command lines start all with a '%'. At the end of each action I give the sources and the produced output of this action.

### I. Preparation of Data

Before any training or recognition can be done with HTK, we have to set up the required data in a format that suits HTK. This section gives examples how to do that for the Numbers task.

#### 1. Speech Corpora

HTK requires either wave files or preprocessed feature files (htk) for each utterance in a separate file (there is nothing alike a pfile in HTK).
To preprocess wave files use the command HCopy.

a) train+cv set:

```
% HCopy -C config -S HCopytrain.slist
```

where:

HCopytrainlist is a 'script' file telling source and destination of each file:

```
/u/drspeech/data/numbers95/wavfile/cs/31/NU-3192.other1.wav /xa/schiel/NUM
   BERS/TRAINING/NU-3192.other1.htk
...
(Source: $SULIN/list/*list.txt)
```

config is the general HTK configuration file which in this case contains the following parameters:

```
SOURCEKIND = WAVEFORM
SOURCEFORMAT = NIST
TARGETKIND = MFCC_Z_E_D_A
```

```
LOPASS = 300
HIPASS = 3400
NUMCHANS = 26
NUMCEPS = 12
ENORMALIZE = T
CEPLIFTER = 22
TARGETRATE = 100000
WINDOWSIZE = 250000
ZMEANSOURCE = T
USEHAMMING = T
PREEMCOEF = 0.97
SAVECOMPRESSED = T
```

Some explanations:

TARGETKIND describes the type of used features. In this case it's:
Mel Frequency Cepstral Coefficients, with cepstral mean subtraction (Z),
log Energy (not C0!), Delta and Delta delta (Acceleration).

LOPASS, HIPASS set the boundaries for filterbank

NUMCHANS is the number of filterbank channels

ENORMALIZE is true (T) : log energy is normalized to 1.0 for each utterance

CEPLIFTER : re-scale cepstral coeff to emphazise higher orders, so that
all dimensions have about the same magnitude

TARGETRATE, WINDOWSIZE : all times are given in 100 nsec units

ZMEANSOURCE is true (T) : bias is subtracted from waveform if any

SAVECOMPRESSED : save htk files in a binary form to speed up reading

The contents of *.htk files can be viewed with the command HList (lots of
options)

```
Source: /u/drspeech/data/numbers95/wavfile/cs/...
        HCopytrain.slist
Target: $DATA/TRAINING
```

b) dev and test set :

Same procedure with appropriate script files

```
Source: /u/drspeech/data/numbers95/wavfile/cs/...
        HCopytest.slist
Target: $DATA/DEV

Source: /u/drspeech/data/numbers95/wavfile/cs/...
        HCopydev.slist
Target: $DATA/TEST
```

## 2. Scripts for Further Processing

Most of the HTK tools get their input files from the command line.
However, if there are thousands of them, it make sense to put them into so
called 'script' files.

```
% cat HCopy<set>.slist | gawk '{ print $2}' > <set>.slist

Source: HCopy<set>.slist
Target: <set>.slist
```

## 3. Dictionary

HTK dictionaries have a simple syntax as follows:

```
<lexical word> [<prob>] <hmm> [<hmm> ...]
```

E.g.

```
eight ah ey tcl t
eight eh ey tcl t
...
```

The dictionary for our task has 32 words and 180 pronunciations. We manually add two silence words '<s>' and '</s>' denoting beginning and ending silence to go conform with the LM.
Since all variants in the lexicon are of equal probability, we can omit this column.

Source: $SULIN/lex/nowaylex/simpcounts-gildea90per-boot_noway.dict
Target: DICT

## 4. Master Label Files (MLF)

HTK can use separate label files for each speech file, but more efficient is the usage of so called Master Label Files (MLF) that store the label files independently of the location of the wave files into one common structure.
The HTK tool HLEd is a general purpose label editor. It can be used to arrange individual label files into an MLF.

For our task we need two different types of label files:

a) Word reference MLFs for all sets (*.ref)

For each set (train+cv, dev, test) do:
- copy the word text files into a dir
- break them in to one word per line
```
  % cat <file> | tr -s ' ' '\012' > <newfile>
```
- transform them into a MLF
```
  % HLEd -l '*' -i <set>-ref.mlf dummy.led *.ref
  (dummy.led is an empty script)
```

Source: $DRSP/wrdfile/cs-clean
Target: <set>-ref.mlf

b) Segmental MLF for train+cv set (*.lab)

Several ways to get these:

A : there are manually segmented data

  Pool these together in a dir and then use HLEd to transform them into an MLF:

```
  % cd dir
  % HLEd -l '*' -i train+cv.mlf dummy.led *.lab
  (dummy.led is an empty script)
```

  Remark:
  HTK can read ESPS label files!

  Source: $DRSP/phnfile/cs-esps
  Target: train+cv.mlf

B : there is no segmental data

  Take the word reference MLF and expand it into a segmental MLF by using a pronunciation dictionary:

```
  % HLEd -l '*' -I train+cv-ref.mlf -i train+cv.mlf -d DICT ex.led
  (ex.led has only one command: EX)

  Remark:
  Always the first pronunciation in the dictionary is used!

  Source: DICT, train+cv-ref.mlf
  Target: train+cv.mlf
```

In our example we follow method A.

5. List of HMMs

Most of the HTK tools require a list of the used models as input. The loaded HMM definition files may include much  more model definitions, but only that one listed will be used. This can be very convenient, if you pool together different types of models into one set.

For our example task we want to train with the manually labelled data. Consequently the set of HMMs should cover both the dictionary and the set of segments found in the train+cv set.

First we analyze the dictionary for distinct hmms:

```
% HDMan -l lex.stat -o DICT
```

We find (lex.stat) only 32 used phonemes.

Then we analyze the train+cv set (MLF) for distinct HMMs:

```
% HLEd -n train+cv.stat -i dummy.mlf dummy.led train+cv.mlf
% rm dummy.mlf
```

and find (train+cv.stat) that the following phones needed for DICT cannot be trained because lack of data: /ae/, /uh/, /hv/, /m/
(These phones were selected by the transcribers to denote special cases of pronunciation and are therefore quite rare.)

To come around this we 'tie' them to similar phonemes. This is done by simply listing the physical model after the (now called) logical model:

Our resulting model list numbersphone.txt contains then 32 logical and 28 physical models and looks like:

```
ae ah
ah
ao
ax
ay
d
dcl
eh
er
ey
f
h#
hh
hv hh
ih
iy
k
kcl
l
m n
n
ow
r
s
```

```
  t
  tcl
  th
  uh ah
  uw
  v
  w
  z

  Source: DICT, train+cv.mlf
  Target: numbersphone.txt

  6. Language Model and Lattice

  HTK cannot use a language model directly. It requires a lattice
  file (*.lat) instead. Lattices can be calculated from different
  sources: Bigrams, finite-states, grammars, forced Viterbi ...

  The tool HBuild is used to compile lattices from other formats. It can
  input: bigram matrix, DARPA bigram, finite-state, etc.

  We use the DARPA bigram/backoff of Su-Lin.

  Source: $SULIN/lm/numbers_cs_train.abigram.gz
  Target: LM

  To calculate the lattice we issue:

  % HBuild -b -n LM -s '<s>' '</s>' WRDLIST LAT

  where:
  WRDLIST is simply the first column of the DICT. It is used by HBuild
  to reduce the output to the list of words in WRDLIST)
  Option '-s' defines the initial and final word symbols used for
  each utterance (default is !ENTRY and !EXIT)
  Option '-b' forces the output to be in binary format (default is ASCII)

  Source: LM
  Target: LAT

Now we have all the data we need to get started. Next we'll have to think
about the structure of the models itself.

=========================================================================

II. Topology of HMMs

Disclaimer:
For a theoretical description of HMM technique please refer elsewhere.
For the following I expect you to have the basic knowledge of how a HMM
is used and how it looks like.

HMMs in HTK are defined in ascii or binary files. You can have a
separate file for each different model (named by the model label) or you
can have all definitions packed into a 'Master Macro File' (MMF).


  1. Create prototypes

  First thing to do is to define a set of basic prototypes that are later
  cloned and used for your phone models.

  Edit a prototype file protos/proto-6mix
  with 4 different prototypes:

  2 - 5 states (named 'two' - 'five')
  6 mixtures per state
  diagonalized covariance matrices
```

The numbers are arbitrary (except of mix weights and transition probabilities, which must sum up to 1.0); only the structure is significant. Only transition probabilities that are non zero will be considered.

Remark: first and last states are virtual!

2. Analyze train+cv set for phone durations

To decide which models get which number of states we look at the average durations of the phones in the training set and then set up a mapping of each phoneme to one of the four prototypes:

```
         2-state-phones: 2.7  - 8.0  frames
         3-state-phones: 8.0  - 12.0
         4-state-phones: 12.0 - 16.0
         5-state-phones: 16.0 - 24.0
```

(The script struct.awk does the job for us.)

Source: train+cv-durations.txt
Target: state-phones.txt

========================================================================

III. Bootstrap, Viterbi training and Baum-Welch training

We are now ready to bootstrap the phone models. The recipe for that is to clone the appropriate prototype (mapped in state-phones.txt), bootstrap it, run Viterbi on the bootstrap data and run a final Baum-Welch on the bootstrap data.

1. HInit

The tool HInit does roughly the following:

- reads the prototype structure for phoneme X
- collect all segments from bootstrap corpus (here train+cv) of phoneme X
- divide the frame stream of each segment linear in number of states
- cluster data of each state in mixtures and calculate mean and variances
- run Viterbi training to all segments until overall likelihood converges

Example for the model /ah/ which is bootstrapped here into a prototype with three states:

```
% HInit -i 20 -l ah -H protos/proto-6mix -o ah -v 0.0001 -I total.mlf \
  -S train+cv.slist -M hmm1 -A -T 1 three
% mv hmm1/proto-6mix hmm1/ah
```

```
where:
Option '-i' is the maximal number of Viterbi training iterations
Option '-l' is the label to look for in the train+cv set for training
Option '-H' loads the prototype file (all prototypes are stored
            in one file)
Option '-o' is the label (macro name) of the resulting model
Option '-v' is the floored variance
Option '-I' is the MLF with the segmental information of the training set
Option '-S' is the 'script' file (list of files to process)
Option '-M' is the directory where to store the resulting model
Option '-A -T' tracing options
```

Since HInit has to be called for each phoneme, it makes sense to put that into a script (INIT).

WARNING:
The output HMM definition file has always the name of the prototype file. Therefore it's necessary to rename the output before the next HInit is

called (see script INIT) or else HInit will overwrite the last model definition file.

After INIT is run, we have a separate HMM definition file for each phoneme in dir hmm1. These can be combined into a so-called Master Macro File (MMF) by using the HMM editor HHEd:

```
% HHEd -w hmm1/NUMBERS.mmf -d hmm1 co.hed numbersphone.txt
```

where co.hed is a command file that does essentially nothing.

Source: protos/proto-6mix, train+cv.mlf, numbersphone.txt
Target: hmm1/NUMBERS.mmf

2. HRest

The tool HRest takes the same data (segments) like HInit and runs iterative Baum Welch training instead of Viterbi training.
It does not do any bootstrapping; that is HInit must be performed first!

Applied to the Numbers task it did not improve the quality of the models.
However, in the Verbmobil task we found a significant improvement.

=========================================================================

## IV. Testbed

We now have the first usable models to run a test.

1. Viterbi

The script TEST shows the call of a version of the recognizer HVite that performs recognition to a test set of data and reports the word accuracy to a corresponding reference label set. The test set can be one of the three sets, but we'll use only the dev set here.

Example for a call of HVite:

```
HVite -t 105.0 -p -2.0 -s 6.5 -i output.mlf -w LAT -H hmm1/NUMBERS.mmf \
  -T 1 -o ST -A -S dev.slist DICT numbersphone.txt
```

As you can see, there are a bunch of parameters that control the Viterbi search. The most important are:
- pruning width (beam width in log prob) (-t)
- LM weighting and offset (-s)
- word end penalty (-p)

These parameters have to be tuned to the dev set (details in Section VII).

Other parameters are:
Option '-i' is the recognition output in form of a MLF
Option '-w' is the used lattice (language model)
Option '-H' is the MMF with the model definitions (HHMs)
Option '-o' defines the format of the output MLF; here timing and scoring
            information is suppressed
Option '-S' script with list of processed files (dev set)

Source: hmm1/NUMBERS.mmf, numbersphone.txt, LAT, DICT
Target: output.mlf

2. Evaluation of results

Also in TEST you see the command HResults, which reads the output MLF of the HVite command and matches it to the reference MLF (here dev-ref.mlf)
The output is the standard error calculation of replacements, deletions and insertions.

A test to the bootstrapped and Viterbi-trained HMMs yields

```
82.62 [H=4117, D=233, S=323, I=256, N=4673] word accuracy
```

A test to the re-estimated models using HRest (see above) yields:
81.53 [H=4042, D=316, S=315, I=232, N=4673]

=========================================================================

## V. Silence Modeling

Up to now only the leading and trailing silence in the utterances is modeled during recognition by the words '<s>' and '</s>' that have the simple pronunciation /h#/' (silence model). Since the LM does not take into account inter-word-silence, I added a new 'silence phone' /sp/ to the end of each word in the dictionary. I build this model by:

- cloning the model /h#/ to /sp/
- deleting the first and third state in /sp/
- tying the remaining state to the second state of the model /h#/
- add a transition from the virtual start to the virtual end of /sp/ enabling it to be skipped totally (so called 'tee model')

Furthermore I added a transition from the third to the first state in the /h#/ model to make it more robust.

Source: hmm2
Target: hmm3

Test:
89.21 [H=4345, D=62, S=266, I=176, N=4673]

=========================================================================

## VI. Embedded Training

Up to now the HMMs were only trained within the fixed boundaries of the manual segmentation. Now we want to run an embedded training, that is the boundaries are iteratively aligned by the Viterbi.

### 1. Re-Align Training Set

The problem arises that we cannot perform an embedded training to the manual labels of train+cv, because since these are hand labels they contain more phonemes than our phone list numbersphone.txt. These phonemes are very sparse, so it makes no sense to train them anyway.

There are two ways out of this:
- map the sparse labels to known and trainable phonemes
- don't use the hand labels for embedded training but use a re-alignment of the train+cv set to the (multi-pronunciation) dictionary.

Since we use this dictionary for testing, the second alternative makes more sense.

To produce a new MLF with the best aligned dictionary words we can use the decoder tool HVite in the following command:

```
% HVite -l '*' -o STW -b '<s>' -a -H hmm3/NUMBERS.3.mmf \
     -i train+cv-ali.mlf -m -t 250.0 -I train+cv-ref.mlf \
     -y lab -S train+cv.slist DICT numbersphone.txt
```

Explanations:

```
-l '*'      : write output MLF with relative paths '*/' instead of full
              paths. This makes the MLF independent of location of the
              htk files
-o STW      : controls the format of the output to the MLF files;
              S : scores suppressed
              T : times suppressed
              W : word labels suppressed
-b '<s>'    : since we have no lattice here, use the word '<s>' as
```

```
               initial and ending silence model
-H ...       : HMM definition file
-i ...       : MLF output file containing the new alignment
-m           : keep track of boundaries
-t 250.0     : beam search pruning factor (rather irrelevant here)
-I ...       : word reference MLF input
-y lab       : extension of alignment label files in the output MLF
-S ...       : script file with list of processed files
```

The MLF train+cv-ali.mlf now contains sequences of labels that are all parts of the multi-pronunciation dictionary DICT.

Source: hmm3, train+cv-ref.mlf, DICT, numbersphone.txt
Target: train+cv-ali.mlf

### 2. Embedded Re-estimation

Now we can use the new alignments in train+cv-ali.mlf to run an embedded re-estimation. This is done with the command HERest in the script HEREST. HERest runs only one iteration over the data at a time. For iterative training you have to call the command again. This enables you to run a test to the dev set after each iteration to find the optimal word accuracy.

Example for one iteration:

```
% HERest -t 250.0 150.0 1000.0 -v 0.0001 \
  -H hmm3/NUMBERS.mmf -I train+cv-ali.mlf -T 00001 \
  -S train+cv.slist -M hmm4 -A numbersphone.txt
```

Explanations:

```
-t X Y Z  : beam search pruning for alignment. First the pruning factor is
            set to X. If the alignment fails, x is incremented by Y and
            retried. This is done until Z is reached. Then an error
            message is issued.
-M        : dir to store the re-estimated MMF
-T        : tracing. 1 = basic reporting (many other options)
-A        : repeat command line for logging purpose
```

Source: hmm3, train+cv-ali.mlf, DICT, numbersphone.txt
Target: hmm4 (1st it.), hmm5 (2nd it.), hmm6 (3rd it.)

```
Test:
hmm4 : 89.75 [H=4368, D=54, S=251, I=174, N=4673]
hmm5 : 89.81 [H=4377, D=47, S=249, I=180, N=4673]
hmm6 : 90.20
```

=========================================================================

## VII. Tuning

The Viterbi search needs to be tuned for the upcoming tests in Part 2. We want a compromise between a good performance and speed.

### 1. Beam Width Pruning (-t)

Up to now the pruning width was set arbitrarily. The script tune-prun.csh finds the optimal pruning parameter regarding to a certain decrease of performance from recognition without pruning. In this case I want the performance not drop more than 0.6 % absolute (half of significance at 0.01 level).

The optimal value for the pruning parameter is then 57.0

### 2. LM scaling (-s)

The script tune-gramscale.csh tunes in the maximum of the performance

regarding the weighting of the language model.
Since this is not a monotone function, the script uses a simple gradient search and stops in the first local maximum.

It turns out that the optimum is right were the factor was set arbitrarily at 6.5

## 3. Word End Penalty (-p)

The word end penalty adds a fixed value to the accumulated log likelihood each time a new word is entered during the Viterbi search. By this the relation of insertions to deletions can somewhat be steered.
Since the behavior of this parameter is quite unclear in terms of word accuracy, we just run a series of experiments varying this factor.

The best performance is achieved with a value of -9.0

Using these parameters a test run on the dev set takes about 22 minutes (approx. 2h without pruning).

=========================================================================
=========================================================================

# PART 2 - HTK TOOL BOX

The following section gives a more detailed discussion about most of the HTK tools and gives examples how to use them within the Numbers task.
This includes manipulation of label files, dictionaries, language models and lattices, manipulation of HMMs definitions and some miscellaneous stuff that I think is worth to know. Buggy behavior is reported with each command description. Also, in this part I report about the performance of triphones versus monophones and the optimal configuration for the Numbers task that I could find within 4 weeks.

Intended audience: people who like to work with HTK

=========================================================================

## 0. General

Aside from the training and decoding algorithms the HTK packages gives you some very powerful general purpose tools that are always needed if you work in speech recognition. Of course all of these are more or less adapted to the HTK formats and needs, but on the other hand the Cambridge people did a nice job to incorporate some of the main standards other than HTK in their tools.
Most of the tools works as script editors. That is, they read some input files, apply a script of commands stored in a separate script file and write the changed output to other files. All HTK tools have simple command line calls and none of them requires X, thus making scripting very easy.
When you work with HTK, you will use these tools very frequently and for sometimes unexpected purposes. Therefore, the following sections will give you some examples how to use these tools and (hopefully) a general idea, what else they are capable off.

===========================================================================

## I. HLEd - Script Editor for Label Files

The tool HLEd is a quite powerful editor for all kinds of label files used in HTK. Label files in HTK can have a complex structure with several layers (word, syllable, phone), but in the vast majority they are simple lists of symbols with optional timing information.
In most cases the usage of HLEd is as follows:

```
% HLEd -i output.mlf -T 1 script.led input.mlf
```

where

input.mlf and output.mlf are selfexplanatory
script.led stores the edit commands

MLFs were mentioned earlier in this tutorial. They are simply a more convenient way to store label information in a compact and location independent form. A MLF with word labels might be looking like this:

```
#!MLF!#
"*/NU-1008.streetaddr.lab"
seventy
two
.
"*/NU-1008.zipcode.lab"
zero
two
eight
...
```

A MLF with phonetic segments is looking like this:

```
#!MLF!#
"*/NU-19.streetaddr.lab"
0 1710000 h#
1710000 2160000 w
2160000 2950000 ah
2950000 3700000 n
3700000 4830000 s
4830000 5270000 ih
5270000 5680000 kcl
5680000 5870000 k
5870000 6370000 s
6370000 6910000 tcl
6910000 7360000 t
7360000 8780000 iy
8780000 9500000 n
9500000 10540000 h#
.
"*/NU-19.zipcode.lab"
0 1780000 h#
1780000 2920000 w
2920000 3560000 ah
...
```

Note that these MLFs are both location independent. That is, where ever a speech file is stored, the HTK tool will look just for the name of the corresponding label file, because the path is '*'.

Most frequent uses of HLEd are:
- combining individual label files into a MLF
- changing/deleting/inserting labels
- splitting/combining labels
- converting phones into biphones/triphones
- expanding word labels into phone labels
- combining phone labels to syllable labels and vice versa

Example:
The following command changes a monophone MLF into a triphone MLF:

```
% HLEd -n numberstriphone.txt -l '*' -i train+cv-tri.mlf \
   triphones.led train+cv-ali.mlf
```

where
Option '-n' gives the name of a file with all new created model names
Option '-l' causes the new MLF to be location independent (path '*/')

and the script triphones.led contains:

```
WB sp
WB h#
```

```
TC
```

Explanation: In HTK left and right context in label names is denoted by '-' and '+'. A model named /tcl-ah+f/ is a /ah/ model in the left context /tcl/ and right context /f/. The 'TC' command expands each model in the MLF train+cv-ali.mlf into the corresponding context dependent triphone model name. The preceding 'WB' commands define two models to be excepted from context expansion and these are the 'inter word' models /h#/ and /sp/. This results into so called within word context models.
Applied to the Numbers task this results into 247 distinct triphones.

TIP: You will get a somewhat misleading error, if you use a speech file together with a MLF that has a different path stored than the speech file. In that case, take a text editor and simply replace the absolute path in the MLF by '*'. The consequent usage of the the option -l '*' with all HTK tools that produce MLFs can avoid such behavior.

=========================================================================

## II. HDMan - Script Editor for Dictionaries

In a similar way like HLEd the contents of dictionaries may be manipulated by the script editor HDMan. Furthermore, it can be used to combine several different dictionaries in different formats into a HTK compatible target dictionary. For each input an individual script files can be defined and a global script file works on the combined output. We do not need HDMan for the Numbers task, therefore I don't give an example here.

HDMan can also be used for determining the list of phone symbols used in a dictionary (see Part 1, Section I.5. for an example).

WARNING:
HDMan requires the input dictionaries to be sorted. This caused some trouble as I used it for the Verbmobil task, because HDMan followed a different sorting table than ASCII when it comes to special characters like '"'. The workaround is to issue the HDMan command 'IR' (for input raw data) in the individual editing script. It does not work in the global editing script and the command is not called 'IM RAW' like said in the HTK Book!

=========================================================================

## III. HLStats, HBuild - Create Language Models and Lattices

Fortunately we already had a ARPA format bigram model when we get started with our example task. But we could have calculated our own language model from the training set using the tool HLStats.

HLStats is a general statistics tool that works on label files or MLFs.
It can be used to calculate:
- number of occurrence of labels
- durations (average, min, max)
- bigram matrix
- backoff bigram (ARPA)
- list of coverage labels from a set of labeled data

Here is an example how we could have calculated the ARPA backoff bigram model from the Numbers train+cv set:

```
% HLStats -b LM -o WRDLIST train+cv-ref.mlf
```

WARNINGS:
The Option -I does not work! Use the input MLF file instead of the regular label files (last argument on command line) like in the above example and it works.
HLStats does not assign the label '!NULL' to out-of-vocabulary-words as said in the reference section. This is quite annoying because you cannot see the out-of-voc rate and you cannot compute bigrams that include

knowledge about out-of-voc occurrences.

The tool HBuild was mentioned earlier to convert the language model (or grammatical model) into a lattice that can be then used by HVite.
What it essentially does is to expand the language model into a fully looped back lattice with all bigrams. This seems to be impossible for very large dictionaries and in fact I never tried it with more than 3500 words. However, HBuild does take into account the backoffs by inserting virtual nodes in the lattice where all the common words of one backoff are 'collected' before connected to the next loop. By that, I guess, it's possible to use very large dictionaries as well.

WARNING:
In HTK versions less than 2.1 HBuild comes up with a lattice that cannot be read by HVite, if the words contain special characters like '"'. A workaround is to edit the lattice and quote these special characters with a '\' before passing it to HVite. This bug was fixed in HTK 2.1.

=========================================================================

## IV. HHEd  -  Manipulation of HMMs

This is the far most powerful and important tool in HTK. Like his siblings it works as a script editor, but input and output are HMM definition files (or MMFs).
Since model definitions may be stored either in separate files or together in one or more MMFs, there are different ways to use HHEd. One possible usage for our example task would be:

```
% HHEd -H hmm2/NUMBERS.mmf -w hmm3/NUMBERS.mmf script.hed numbersphone.txt
```

where:
Option '-w' defines the output MMF
Option '-H' defines the input MMF
script.hed stores the editing commands
numbersphone.txt is the list of HMM names that are to be edited

### 1. HHEd Commands

Like in HLEd and HDMan the editor commands are two capital letter mnemonics.
For example RT = remove transition:

```
RT i j itemlist
```

will remove the transition from state i to state j in all transition matrices found in the itemlist and re-normalize the remaining transition probabilities.

There are 26 different, sometimes very specialized commands in the reference section of HHEd. The most important things HHEd can do are:
- clone to bi- or triphones
- tie states or parts of states into macros
- manipulate transitions
- clustering of states (classic or decision trees)
- compact HMM definitions (replace identical definitions by pointers)
- split mixtures

### 2. HHEd Itemlists

HHEd uses a sort of pattern matching language to address parts of the loaded models in the commands. These patterns are referred to as 'itemlist' in the synopsis of the different commands. A complete description of this can be found in the reference section, but the general idea is to view a HMM as a C-like structure together with some UNIX-like pattern matching features.

Some examples will make the idea clear:

```
{ah.transP}                         : transition matrix of model /ah/
```

| | |
|---|---|
| {*.transP} | : transition matrices of all models |
| {*-ah+f.transP} | : transition matrices of all triphone models /ah/ with right context /f/ |
| {ah.state[2]} | : the second state of model /ah/ |
| {ah.state[2].mix[5]} | : the 5th mixture of the second state of model /ah/ |
| {(ah,ae,ax).state[2-5]} | : the 2nd to 5th states of the models /ah/, /ae/ and /ax/ |
| {*.state[3].mix[2-5].mean} | : the means in the 2nd to 5th mixture in state 3 of all loaded models |
| {*.state[2-5].mix[2-5].cov} | : the variances or covariance matrices (depending on the model) of... |

TIPS:
Note that states and mixtures are numbered beginning with 1, but the first and last states are virtual and do not contain any mixtures.
The pattern in itemlist may be 'over-specified' in that sense that it may refer to items that are not existent.
For example, if you like to tie all variances in all 6 mixtures in all states in all loaded models together, but some models have 3 states while other have 4 states, you can address all these by the following command:

```
TI glob_cov {*.state[2-5].mix[1-6].cov}
```

In this example all variances that match the itemlist will be replaced by a so-called macro 'glob_cov' (for macros see next point), while the macro itself is computed of the maximum values found in all replaced variances.

HHEd will notice that some of your loaded models actually don't have a state number 5 and give a warning. But it will correctly tie the existing variances together.

## 3. HHEd Macros

HHEd macros are technically just simple replace operations like C macros. However, HHEd macros each have a certain type that refers to a part of a HMM definition. You can determine the type of a macro by its definition or its call:

Examples:

```
~v "glob_var"
   <Variance> 4
   1.0 0.8 0.7 1.0
```

might be a simple version of the macro "glob_var" we saw in the last example. The 'v' in the first line denotes this macro of being the type 'diagonal variance vector'. HHEd will check that this macro is only called in the appropriate places, namely where a variance vector is required in a HMM model definition.

HHEd macros are therefore an easy way to use shared structures (the usually result of some tying operation).

Macros definitions (like in C) have to appear before they are referred to. HHEd takes care for that, if you write everything into a MMF. If you use distributed model definitions (in several files) you have to take care that the macros are loaded first (usually by the option -H).

Technically, everything in a MMF is a macro, even the global option values (~o) and the model definitions itself (~h). But these two are never referred to and hence should not be called macros.

## 4. Some Examples on the Numbers Task

### a) Combine distributed HMM definitions into one single MMF

After bootstrapping our models (see Part 1, Section III) we had a separate file with the model definition for each phone. To combine these into a single MMF we simply load them into HHEd, do some dummy script and write them out:

```
% HHEd -w hmm2/NUMBERS.mmf -d hmm1 co.hed numbersphone.txt
```

where:
Option '-d' tells HHEd where to look for the individual model definition files (default is the current directory)
co.hed is a dummy script that does nothing

Note that here the file numbersphone.txt contains merely a list of HMM names and HHEd looks for files with the same names in the directory hmm1. If HHEd cannot find a file corresponding to a name in numbersphone.txt, it will give an error message and exit.

b) Silence models

This is a more detailed description, what was mentioned in Part 1, Section IV. We want to introduce a short optional silence model /sp/ at the end of each word. We already have a 3-state model /h#/ trained to the silence at the begin and end of each utterance.

First we take a standard text editor, copy the model definition of /h#/ (~h "h#") to a new model definition named /sp/ (~h "sp") in our MMF hmm1/NUMBERS.mmf.
Then we delete the second and fourth state and edit the transition matrix accordingly (the values are not important; only the rows must be sum up to 1.0) and store the whole thing in hmm2/NUMBERS.mmf.
We now have a MMF with an additional model definition /sp/ for a one-state silence model. We issue the following HHEd command:

```
% HHEd -w hmm3/NUMBERS.mmf -H hmm2/NUMBERS.mmf sil.hed numbersphone.txt
```

where the script sil.hed contains:

```
AT 4 2 0.2 {h#.transP}
AT 1 3 0.3 {sp.transP}
TI h#st {h#.state[3],sp.state[2]}
```

The first AT command ('add transition') will add a 'backward transition' into the begin and end silence model /h#/ to make it more robust.
The second AT command will add a transition from the (virtual) first state to the (virtual) last state of the model /sp/, thus making it optional (it can be be jumped over without emitting any likelihood). This is called a 'tee model' (after the UNIX 'tee' command).
The following tie command TI will create a macro named "h#st" that represents the middle state of the model /h#/ and is also tied to the single emitting state of model /sp/.

If you look at the resulting MMF hmm3/NUMBERS.mmf, you will find a new macro of type '~s "h#st" right at the top of the file followed by a full state definition with 6 mixtures. Way down the file in the model definitions of /h#/ and /sp/ this macro will be referred to where the second resp. first emitting states are defined.

c) Number of Mixtures

In our example in Part 1 we started with prototypes that had 6 mixtures per state. However, this was only (a pretty good) guess of me. To be sure that you have chosen the optimal topology for your models there is no way to avoid the heuristic try-and-fail method.
I ran a series of trainings on different number of mixtures. It is recommended to start with a single Gaussian model, train it until it converges on the dev set and then increase the number of mixtures by one, train them and so on.

The splitting of mixtures can be done like follows:

```
% HHEd -H hmm3-1mix/NUMBERS.mmf -M hmm3-2mix mu.hed numbersphone.txt
```

where mu.hed contains the single command:

```
MU 2 {*.state[2-6].mix}
```

The MU command ('mix up') will take each mixture found in the itemlist
and convert them in the number of mixtures given in the command (here: 2).
The algorithm is quite complex (please refer to the reference section for
details); it is not merely a split of the means in the direction of the
max variance, but MU even takes care that the resulting mixtures have a
reasonable weight (deleting mixtures that fall under a weight threshold),
that the split is distributed evenly over mixtures, etc.

I did this iterative procedure of splitting and training and found that
the best performance was achieved with 7 mixtures:

```
Word accuracy: 90.02 [H=4374, D=53, S=234, I=178, N=4661]
```

(Note that this is done with pruning! Therefore it's actually better than
the results reported in Part 1, Section VI.)

d) Triphones

We already produced a MLF with triphones of the training set in Section I
using HLEd. The same operation gave us a list numberstriphone.txt of the
247 occurring triphones in the training set. Now we want to produce the
corresponding triphone model definitions from 1-mixture monophones:

```
% HHEd -H hmm3-1mix/NUMBERS.mmf -M hmm3-1mix-tri \
  triphones.hed numbersphone.txt
```

where triphone.hed contains the clone command:

```
CL numberstriphone.txt
```

The CL command will read the list of triphones from numberstriphone.txt,
look for corresponding monophones in the loaded model definitions and clone
them into a MMF with triphone definitions. Note that the monophones
itself are NOT copied to the output MMF, except they were also listed
in the file numberstriphone.txt.

(The -M option is another way to redirect the output: the output MMF will
have the same name as the input MMF, but written to the directory
hmm3-1mix-tri)

Now we have two problems that give a wonderful example of what things you
run into working with HTK:

1. The DICT still contains the phones /ae/, /uh/, /hv/ and /m/ (you
remember that these were in the manually labeled training data, but could
not be trained). Since the list of triphones was derived from the
converted MLF train+cv-ali.mlf that was produced by a forced alignment, it
now happens that these phones do not occur in the triphone list.  Because
the forced alignment always outputs the physical model name.  So, the
dictionary and the model set are now incompatible. We solve this by
replacing this phone symbols in DICT by the ones already used in Part 1.

2. We find some triphones in DICT that are not found in the training set
The reason is simply that the forced alignment did not choose some
variants in DICT during the forced alignment.
Two possible ways out of this:
 a) synthesis the required 4 triphones (lot of work)
 b) delete the 16 variants where these special triphone occur
I did the latter and ran a test on the monophone baseline system to make
sure that this didn't hurt performance. (It turned out that even without
any multi-pronunciation we receive the same performance!)

Now we can re-estimate the triphones using the script HEREST. After 4
iterations we achieve:

Word accuracy: 81.39 [H=3979, D=82, S=414, I=337, N=4475]

This is not very surprising at first, because these models are not tied
and probably have not enough data to come up with robust parameters.

e) Clustering and Tying States

I took the 247 triphones and tried several clustering techniques to them.
I spare you the details here and just mention that the biggest problem is
the huge variety of parameters you can use in HTK.

The best configuration with triphones I could find was:
- cluster states class independent and with a minimum of 10
  'visits' per state to a maximum within cluster distance of 0.05
- re-estimate (3 it.)
- 'mix up' to 6 mixtures per state
- re-estimate (8 it.)

Word accuracy: 87.11 [H=4303, D=61, S=261, I=274, N=4625]

I don't claim that this is the optimum that you can achieve with
triphones on this task; I merely ran out of time (and furthermore my
scratch disk crashed and I lost all the data about triphone experiments).

TIPS to HHEd:
Note that the input MMF may store more model definitions as actually
being edited. Only the models listed in the model list (last
argument) will be considered for manipulation.
Sometimes, the input models will not be not be copied to the
output MLF. For example, if you give a set of monophone models as an
input and let HHEd clone them into triphone models, the output MMF will
only contain those triphones and no copies of the original monophones.
The TI command will behave very different depending on what type you use
it. Be sure that you understand the different ways how TI will calculate
the resulting macro from a tying operation.

=========================================================================

V. Miscs

The following is a collection of miscellaneous stuff

0. Documentation

The doc of HTK consists of the HTK Book by Steve Young et al. (including
the reference section) and a html version of the book
(file:/u/drspeech/opt/htk/HTKBook/HTKBook.html). Nothing more! There
are no info or man pages to the HTK tools. If you are planing to use the
libraries in your own software, there is no way around to read the source
code itself (Have Fun!).

1. Our Top Performance on Numbers

I took the 7-mixtures monophone models and tied the variances in each
state of the models with less than 500 observations in the training set.
The idea was to make these badly estimated states more robust. Then I ran
4 iteration of HEREST.

Word accuracy: 90.38 [H=4378, D=51, S=226, I=171, N=4655]

I tuned in the word end penalty to -9.0 and achieved

Word accuracy: 90.53 [H=4363, D=56, S=239, I=146, N=4658]

Then I switched off the beam search pruning and achieved

Word accuracy: 91.53 [H=4408, D=50, S=215, I=131, N=4673]

This best result was done with a model of roughly 61152 acoustic parameters.

## 2. Weighted pronunciations and Re-scoring of Lattices

As mentioned earlier, HTK (since version 2.1) can use a posteriori probabilities in the dictionary to weight different pronunciations of the same word. There is no need that these sum up to 1.0 and there is an additional option '-r' of HVite that scales up the weighting in the search (this option is nowhere documented!).

Before I found out about this new feature, I tried to re-score lattices produced by HVite during recognition and then let HVite match them to the signal again. However, this always resulted in considerable worse results, even if I did nothing to the lattice at all. I do not know exactly the reason for this, but it might be that the lattice is pruned in some sort and looses the best search path that way.

## 3. The Mixture Problem

Better start with a lesser number of mixtures and work your way up. You cannot go in the reverse direction, that is there is now way to merge mixtures in HTK. If you intend to do some state clustering with decision trees, you must use single Gaussian models first.

## 4. Efficiency

```
Some CPU times on a Ultra 1 with 128MB (averages):
HInit 28 models 1 mix                : 3:43 h
HVite without pruning                : 2:23 h
HVite with pruning 57.0              : 0:22 h
HERest one iteration                 : 0:35 h
```

If you use HInit with more than a few hours of data, it becomes very slowly. I don't know the reason for this, because the slow part is not the iterative training bit the data collection part. Matbe it's simply bad implemented.
For example the HInit of one model in 11 h of speech lasts 9 CPU h in average.

## 5. Technical Stuff

Some configurations cause HVite to hog memory like a gulo gulo.
Especially if you use the option '-n' with values higher than 10 (lattice output).

HInit 'collects' all segments into RAM first before starting any processing.
This can be a problem, if you have more than 100.000 segments of one single class.

If you have LaTeX style words in your dictionary or other quote signs (like in "d'accord", HTK tools will interpret a leading quote sign as a opening quotation and miss the closing quote sign.
This will result in error messages like:

```
+??13 read string to long
```

Workaround: Quote all " and ' in your dictionary with \

## 6. What I really missed

No way to combine existing HMM definitions into larger models (eg. phones to syllables). You have to do it yourself.

No way to gradually re-estimate HMM in embedded training. Consider for instance that you have trained up very robust monophones, then switch to triphones, where there is much less data for each model. A good thing would be to have some mechanism that used deleted interpolation between the different sets using the occupation statistics.