EE627 – Term Project : Jul. 2013 Semester

August 12, 2013

Title : Build and demonstrate a real time continuous speech recognition system in English

Assigned to : Batch No. 1

TAs Assigned : Waquar Ahmad. He is Available @ACES 203 MiPS Lab EE Department. email : wahmad@iitk.ac.in

Objective :

The objective of this project is to build a continous speech recognition system for English. A real time speech recognitiion needs to be built using the TIMIT database. Experimental results for speech recognition in terms of word error rate (WER) also need to be provided.

Methodology To Be Followed :

The methodlolgy need to be followed for performing the recogniton experiments are as follows:

- Building the task grammar (a "language model")
- Constructing a dictionary for the models
- Creating transcription files for training data .
- Encoding the data (feature processing)
- (Re-)training the acoustic models
- Evaluating the recognizer against the test data
- Reporting recognition results
- Real time system development

Database for Training

The database used for training is TIMIT database. The TIMIT corpus of read speech is designed to provide speech data for acoustic-phonetic studies and for the development and evaluation of automatic speech recognition systems. TIMIT contains broadband recordings of 630 speakers of eight major dialects of American English, each reading ten phonetically rich sentences. The TIMIT corpus includes time-aligned orthographic, phonetic and word transcriptions as well as a 16-bit, 16kHz speech waveform file for each utterance. Corpus design was a joint effort among the Massachusetts Institute of Technology (MIT), SRI International (SRI) and Texas Instruments, Inc. (TI). The speech was recorded at TI, transcribed at MIT and verified and prepared for CD-ROM production by the National Institute of Standards and Technology (NIST). The TIMIT corpus transcriptions have been hand verified. Test and training subsets, balanced for phonetic and dialectal coverage, are specified. Tabular computer-searchable information is included as well as written documentation. The TIMIT database is available at MiPS Lab. Please contact the TA assigned for the same.

Tools To Be Used

The Tools to be used to implement this term project is the HTK Toolkit. The details of download, installation, and usage are available at the following URL :

http://htk.eng.cam.ac.uk/

Deliverables/ Submissions

The deliverables or submission procedures for the term project are as follows:

- Presentations and Report : Two set of presentations is required for every batch in this term project. The first presentation will be scheduled before mid sem and the second presentation will be scheduled before end sem. The marks will be distributed separately for two presentation. Additionally a report needs to be submitted detailing the project.
- System Demo in real time : The demonstration of the project is need to be carried out by each batch. The demonstration includes the real time presentation of the working model for recognition system.
- Code/script submissions : The code or script has to be submitted by each batch which will include complete details of the project.

Other Useful Links

The other useful links that might be helpful in preparation of the code or script are as follows :

- HTK archives : http://www.fit.vutbr.cz/~ihubeika/ZRE/lab/htk_ano_ne_english.pdf, http://ijcsi.org/papers/IJCSI-9-4-1-359-364.pdf
- Matlab Code : http://www.mathworks.in/company/newsletters/articles/developing-an-isolated-word-recognition-system-in-matlab.html

HTK (v.3.1): Basic Tutorial

Nicolas Moreau / 02.02.2002

Content

What is HTK?

HTK is the "Hidden Markov Model Toolkit" developed by the Cambridge University Engineering Department (CUED). This toolkit aims at building and manipulating **Hidden Markov Models** (HMMs). HTK is primarily used for **speech recognition** research (but HMMs have a lot of other possible applications…)

HTK consists of a set of library modules and tools available in C source form. It is available on free download, along with a complete documentation (around 300 pages).
See: http://htk.eng.cam.ac.uk/.

1 Yes/No Recognition System

In this tutorial, we propose to build a 2-word recogniser with a { "*Yes*", "*No*" } vocabulary, based on HTK tools. It's the most basic Automatic Speech Recognition (ASR) system that can be designed…

1.1 Construction steps

The main construction steps are the following:

- Creation of a training database: each element of the vocabulary is recorded several times, and labelled with the corresponding word.
- Acoustical analysis: the training waveforms are converted into some series of coefficient vectors.
- Definition of the models: a prototype of Hidden Markov Model (HMM) is defined for each element of the task vocabulary.
- Training of the models: each HMM is initialised and trained with the training data.
- Definition of the task: the grammar of the recogniser (what can be recognised) is defined.
- Recognition of an unknown input signal.
- Evaluation: the performance of the recogniser can be evaluated on a corpus of test data.

1.2 Work space organisation

It is recommended to create a directory structure such as the following:

- `data/` : to store training and test data (speech signals, labels, etc.), with 2 sub-directories `data/train/` and `data/test/` to separate the data used to train the recogniser from the ones used for performance evaluation.
- `analysis/` : to store files that concern the acoustical analysis step.
- `training/` : to store files that concern the initialisation and training steps.
- `model/` : to store the recogniser's models (HMMs).
- `def`/ : to store files that concern the definition of the task.
- `test/` : to store files that concern the test.

1.3 Standard HTK tool options:

Some standard options are common to every HTK tools. In the following, we will use some of them:

- `-A` : displays the command line arguments.
- `-D` : displays configuration settings.
- `-T 1` : displays some information about the algorithm actions.

To have the complete list:
see HTK documentation, p.50 (Chap.4, The Operating Environment).

2 Creation of the Training Corpus

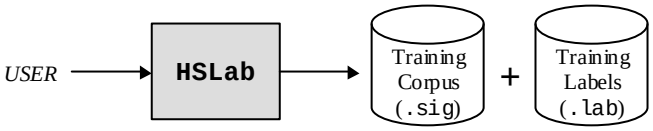

Fig. 1: Recording and labelling training data.

First, we have to record the "*Yes*" and "*No*" speech signals with which word models will be trained (the *training corpus*). Each speech signal has to be labelled, that is: associated with a text (a label) describing its content. Recording and labelling can be done with the `HSLab` HTK tool (but any other tool could be used).

To create and label a speech file:

```
HSLab  any_name.sig
```

The tool's graphical interface appears.

2.1 Record the Signal

Press "*Rec*" button to start recording the signal, then "*Stop*".
A buffer file called `any_name_0.sig` is automatically created in the current directory.
(if you make a new record, it is saved in a second buffer file called `any_name_1.sig`).

Remarks:
- The signal files (`.sig`) are here saved in a specific HTK format. It is however possible to use other audio format (.wav, etc.):
see HTK documentation, p.68 (Chap.5, Speech Input/Output).
- The default sampling rate is 16*kHz*.

2.2 Label the Signal

To label the speech waveform, first press "*Mark*", then select the region you want to label. When the region is marked, press "*Labelas*", type the name of the label, then press *Enter*.

In this tutorial, we will only record isolated words ("*Yes*" or "*No*") preceded and followed by a short silence. For each signal, we have to label 3 successive regions: start silence (with label `sil`), the recorded word (with label `yes` or `no`), and end silence (with label `sil`). These 3 regions cannot overlap with each other (but no matter if there is a little gap between them).
When the 3 labels have been written, press "*Save*": a label file called `any_name_0.lab` is created. At this point you can press "*Quit*".

Remark:
The `.lab` file is a simple text file. It contains for each label a line of the type:

```
4171250 9229375 sil
9229375 15043750 yes
15043750 20430625 sil
```

where numbers indicate the start and end sample time of each label. Such a file can be manually modified (for example to adjust the start and end of a label) or even created (the use of the `HSLab` tool is not required).

2.3 Rename the Files

After each recording/labelling, you have to rename the `.sig` and `.lab` files to your convenience (e.g. `yes01.sig` and `yes01.lab`).

10 records for each of the 2 words should be enough for this tutorial.

The signal files should be stored in a `data/train/sig/` directory (the training corpus), the labels in a `data/train/lab/` directory (the training label set).

For more details on the `HSLab` graphical interface:
see HTK documentation, p.237 (Reference Section, HSLab).

3 Acoustical Analysis

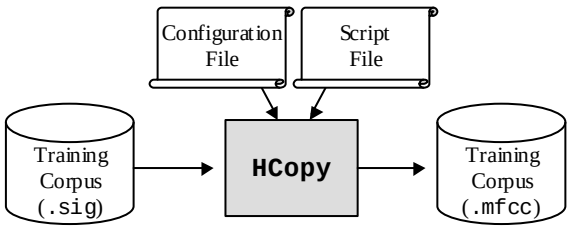

Fig. 2: Conversion of the training data.

The speech recognition tools cannot process directly on speech waveforms. These have to be represented in a more compact and efficient way. This step is called "acoustical analysis":

- The signal is segmented in successive frames (whose length is chosen between 20ms and 40ms, typically), overlapping with each other.
- Each frame is multiplied by a windowing function (e.g. Hamming function).
- A vector of acoustical coefficients (giving a compact representation of the spectral properties of the frame) is extracted from each windowed frame.

The conversion from the original waveform to a series of acoustical vectors is done with the `HCopy` HTK tool:

```
HCopy  -A  -D  -C analysis.conf  -S targetlist.txt
```

`analysis.conf` is a configuration file setting the parameters of the acoustical coefficient extraction.
`targetlist.txt` specifies the name and location of each waveform to process, along with the name and location of the target coefficient files.

3.1 Configuration Parameters

The configuration file is a text file ("#" can be used to introduce a comment). In this tutorial, the following configuration file will be used:

```
#
# Example of an acoustical analysis configuration file
#
SOURCEFORMAT = HTK            # Gives the format of the speech files
TARGETKIND = MFCC_0_D_A       # Identifier of the coefficients to use

# Unit = 0.1 micro-second :
WINDOWSIZE = 250000.0         # = 25 ms = length of a time frame
TARGETRATE = 100000.0         # = 10 ms = frame periodicity

NUMCEPS = 12                  # Number of MFCC coeffs (here from c1 to c12)
USEHAMMING = T                # Use of Hamming function for windowing frames
PREEMCOEF = 0.97              # Pre-emphasis coefficient
NUMCHANS = 26                 # Number of filterbank channels
CEPLIFTER = 22                # Length of cepstral liftering

# The End
```

List. 1: Analysis configuration file.

With such a configuration file, an MFCC (Mel Frequency Cepstral Coefficient) analysis is performed (prefix "`MFCC`" in the `TARGETKIND` identifier). For each signal frame, the following coefficients are extracted:

- The 12 first MFCC coefficients [c1,…, c12] (since `NUMCEPS = 12`)
- The "null" MFCC coefficient c0, which is proportional to the total energy in the frame (suffix "`_0`" in `TARGETKIND`)
- 13 "Delta coefficients", estimating the first order derivative of [c0, c1,…, c12] (suffix "`_D`" in `TARGETKIND`)
- 13 "Acceleration coefficients", estimating the second order derivative of [c0, c1,…, c12] (suffix "`_A`" in `TARGETKIND`)

Altogether, a 39 coefficient vector is extracted from each signal frame.

For more details on acoustical analysis configuration:
see HTK documentation, p.58-66 (Chap.5, Speech Input/Output).

3.2 Source / Target Specification

One or more "source file / target file" pairs (i.e. "original waveform / coefficient file") can be directly specified in the command line of `HCopy`. When to many data are to be processed, the `-S` option is used instead. It allows to specify a script file of the form:

```
data/train/sig/yes01.sig   data/train/mfcc/yes01.mfcc
data/train/sig/yes02.sig   data/train/mfcc/yes02.mfcc
      etc...
data/train/sig/no01.sig   data/train/mfcc/no01.mfcc
data/train/sig/no02.sig   data/train/mfcc/no02.mfcc
      etc...
```

List. 2: Conversion script file.

Such a text file can be automatically generated (using a Perl script, for instance).
The new training corpus (`.mfcc` files) is stored in the `data/train/mfcc/` directory.

For more details on the `HCopy` tool:
see HTK documentation, p.195 (Reference Section, HCopy).

4 HMM Definition

In this tutorial, 3 acoustical events have to be modelled with a Hidden Markov Model (HMM): "*Yes*", "*No*" and "*Silence*". For each one we will design a HMM.

The first step is to choose *a priori* a topology for each HMM:

- number of states
- form of the observation functions (associated with each state)
- disposition of transitions between states

Such a definition is not straightforward. There is actually no fixed rule for it.
Here, we will simply choose the same topology for each of the 3 HMMs (Fig.3):

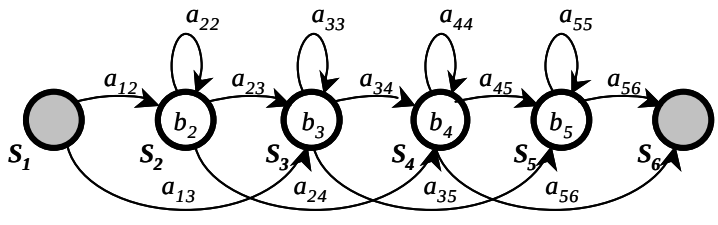

Fig. 3: Basic topology

The models consist actually of 4 "active" states $\{S_2, S_3, S_4, S_5\}$: the first and last states (here S_1 and S_6), are "non emitting" states (no observation function), only used by HTK for some implementation facilities reasons. The observation functions b_i are single gaussian distributions with diagonal matrices. The transition probabilities are quoted a_{ij}.

In HTK, a HMM is described in a text description file. The description file for the HMM depicted on Fig.**3** is of the form:

```
~o <VecSize> 39 <MFCC_0_D_A>
~h "yes"
<BeginHMM>
      <NumStates> 6
      <State> 2
         <Mean> 39
           0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
           0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
           0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
         <Variance> 39
           1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
           1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
           1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
      <State> 3
         <Mean> 39
           0.0 0.0 (...) 0.0
         <Variance> 39
           1.0 1.0 (...) 1.0
      <State> 4
         <Mean> 39
           0.0 0.0 (...) 0.0
         <Variance> 39
           1.0 1.0 (...) 1.0
      <State> 5
         <Mean> 39
           0.0 0.0 (...) 0.0
         <Variance> 39
           1.0 1.0 (...) 1.0
      <TransP> 6
           0.0 0.5 0.5 0.0 0.0 0.0
           0.0 0.4 0.3 0.3 0.0 0.0
           0.0 0.0 0.4 0.3 0.3 0.0
           0.0 0.0 0.0 0.4 0.3 0.3
           0.0 0.0 0.0 0.0 0.5 0.5
           0.0 0.0 0.0 0.0 0.0 0.0
<EndHMM>
```

List. 3: HMM description file (prototype).

`~o <VecSize> 39 <MFCC_0_D_A>`
is the header of the file, giving the coefficient vector size (39 coefficients here), and the type of coefficient (`MFCC_0_D_A` here).

`~h "yes" <BeginHMM> (...)<EndHMM>`
encloses the description of a HMM called “yes”.

`<NumStates> 6`
gives the total number of states in the HMM, including the 2 non-emitting states 1 and 6.

`<State> 2`
introduces the description of the observation function of state 2. Here we have chosen to use single-gaussian observation functions, with diagonal matrices. Such a function is entirely described by a mean vector and a variance vector (the diagonal elements of the autocorrelation matrix). States 1 and 6 are not described, since they have no observation function.

`<Mean> 39`
`0.0 0.0 (...) 0.0` (x 39)
gives the mean vector (in a 39 dimension observation space) of the current observation function. Every element is arbitrary initialised to 0: the file only gives the "prototype" of the HMM (its global topology). These coefficients will be trained later.

`<Variance> 39`
`1.0 1.0 (...) 1.0` (x 39)
gives the variance vector of the current observation function. Every element is arbitrary initialised to 1.

`<TransP> 6`
gives the 6x6 transition matrix of the HMM, that is:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} \end{bmatrix}$$

where a_{ij} is the probability of transition from state i to state j. Null values indicate that the corresponding transitions are not allowed. The other values are arbitrary initialised (but each line of the matrix must sum to 1): they will be later modified, during the training process.

Such a prototype has to be generated for each event to model.
In our case, we have to write a prototype for 3 HMMs that we will call "*yes*", "*no*", and "*sil*" (with headers `~h "yes"`, `~h "no"` and `~h "sil"` in the 3 description files).
These 3 files could be named `hmm_yes`, `hmm_no`, `hmm_sil` and be stored in a directory called `model/proto/`.

For more details on HMM description files:
see HTK documentation, p.94 (Chap.7, HMM Definition Files).

5 HMM Training

The training procedure is described on Fig.**4**:

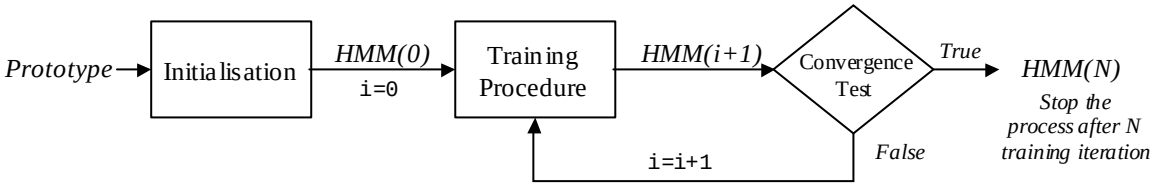

Fig. 4: Complete training procedure

5.1 Initialisation

Before starting the training process, the HMM parameters must be properly initialised with training data in order to allow a fast and precise convergence of the training algorithm.
HTK offers 2 different initialisation tools: `Hinit` and `HCompv`.

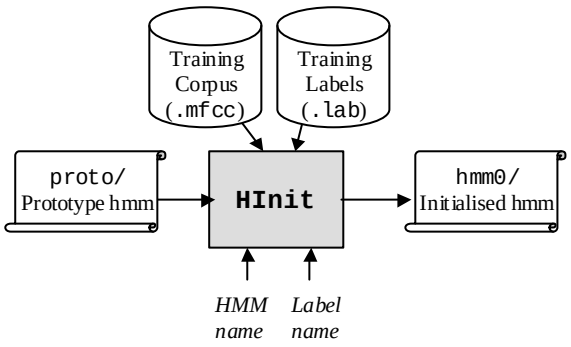

Fig. 5: Initialisation from a prototype.

HInit

The following command line initialises the HMM by time-alignment of the training data with a Viterbi algorithm:

```
HInit -A -D –T 1 -S trainlist.txt -M model/hmm0     \
      -H model/proto/hmmfile -l label -L label_dir   nameofhmm
```

nameofhmm is the name of the HMM to initialise (here: `yes`, `no`, or `sil`).
hmmfile is a description file containing the prototype of the HMM called `nameofhmm` (here: `proto/hmm_yes`, `proto/hmm_no`, or `proto/hmm_sil`).
trainlist.txt gives the complete list of the `.mfcc` files forming the training corpus (stored in directory `data/train/mfcc/`).
label_dir is the directory where the label files (`.lab`) corresponding to the training corpus (here: `data/train/lab/`).
label indicates which labelled segment must be used within the training corpus (here: `yes`, `no`, or `sil` because have used the same names for the labels and the HMMs, but this is not mandatory…)
model/hmm0 is the name of the directory (must be created before) where the resulting initialised HMM description will be output.

This procedure has to be repeated for each model (`hmm_yes`, `hmm_no`, `hmm_sil`).

Remark:
The HMM file output by `HInit` has the same name as the input prototype.

HCompv

The `HCompv` tool performs a "flat" initialisation of a model. Every state of the HMM is given the same mean and variance vectors: these are computed globally on the whole training corpus. The initialisation command line is in this case:

```
HCompv -A -D –T 1 -S trainlist.txt -M model/hmm0flat     \
       -H model/proto/hmmfile -f 0.01  nameofhmm
```

nameofhmm , **hmmfile** , **trainlist.txt** : see `HInit`.

model/hmm0flat : the output directory must be different from the one used with HInit (to avoid overwrite).

Remark: the **label** option can also be used. In that case the estimation of the global mean and variance is based on the corresponding labelled parts of the training corpus only.

We won't use HCompv to initialise our models (it was already done with HInit).
But HCompv also output, along with the initialised model, an interesting file called vFloors, which contains the global variance vector multiplied by a factor (see List.4). This factor can be set with the **-f** option (here: **0.01**).

```
      ~v varFloor1
      <Variance> 39
            5.196781e-001  2.138549e-001 (...) 3.203219e-003
```

List. 4: Variance floors macro file, vFloors.

The values stored in varFloor1 (called the "variance floor macro") can be used later during the training process as floor values for the estimated variance vectors.
During training iterations, it can happen that the number of training frames associated with a particular HMM state is very low. The estimated variance for that state may then have a very small value (variance is even null if only one training frame is available). The floor values can be used instead in that case, preventing the variance from being to small (and possibly causing computation errors).
Here, we will use HCompv only once, with any of our HMM prototype, in order to compute the varFloor1 macro described above. The corresponding vFloors file is output in directory model/hmm0flat/.

5.2 Training

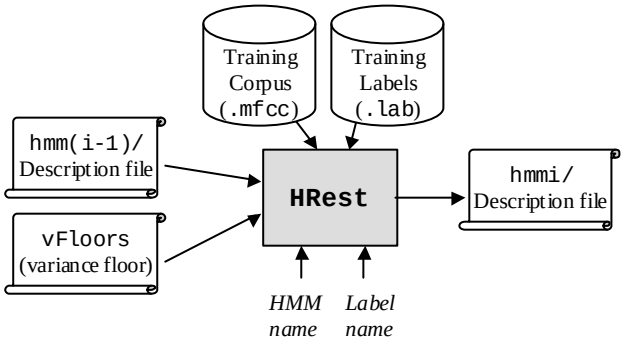

Fig. 6: A re-estimation iteration.

The following command line perform one re-estimation iteration with HTK tool HRest, estimating the optimal values for the HMM parameters (transition probabilities, plus mean and variance vectors of each observation function):

```
HRest -A -D -T 1 -S trainlist.txt -M model/hmmi -H vFloors   \
      -H model/hmmi-1/hmmfile -l label -L label_dir   nameofhmm
```

nameofhmm is the name of the HMM to train (here: `yes`, `no`, or `sil`).
hmmfile is the description file of the HMM called `nameofhmm`. It is stored in a directory whose name indicates the index of the last iteration (here `model/hmm`***`i-1/`*** for example).
vFloors is the file containing the variance floor macro obtained with `HCompv`.
trainlist.txt gives the complete list of the `.mfcc` files forming the training corpus (stored in directory `data/train/mfcc/`).
label_dir is the directory where the label files (`.lab`) corresponding to the training corpus (here: `data/train/lab/`).
label indicates the label to use within the training data (`yes`, `no`, or `sil`)
model/hmm*i* , the output directory, indicates the index of the current iteration ***i***.

This procedure has to be repeated several times for each of the HMM to train.
Each time, the `HRest` iterations (i.e. iterations within the current re-estimation iteration…) are displayed on screen, indicating the convergence through the `change` measure. As soon as this measure do not decrease (in absolute value) from one `HRest` iteration to another, it's time to stop the process. In our example, 2 or 3 re-estimation iterations should be enough.

The final word HMMs are then: `hmm3/hmm_yes`, `hmm3/hmm_no`, and `hmm3/hmm_sil`.

6 Task Definition

Every files concerning the task definition should be stored in a dedicated `def`/ directory.

6.1 Grammar and Dictionary

Before using our word models, we have to define the basic architecture of our recogniser (the *task grammar*). We will first define the most simple one: a start silence, followed by a single word (in our case "*Yes*" or "*No*"), followed by an end silence.

In HTK, the task grammar is written in a text file, according to some syntactic rules. In our case, the grammar is quite simple:

```
/*
 * Task grammar
 */

$WORD = YES | NO;

 ( { START_SIL } [ $WORD ] { END_SIL } )
```

List. 5: Basic task grammar.

The `WORD` variable can be replaced by `YES` or `NO`.
The brackets {} around `START_SIL` and `END_SIL` denotes zero or more repetitions (a long silence segment, or no silence at all before or after the word are then allowed).
The brackets [] around `$WORD` denotes zero or one occurrence (if no word is pronounced, it's possible to recognise silence only).

For more details on HTK syntactic rules:
see HTK documentation, p.163 (Chap.12, Networks, Dictionaries and Language Models).

The system must of course know to which HMM corresponds each of the grammar variables `YES`, `NO`, `START_SIL` and `END_SIL`. This information is stored in a text file called the *task dictionary*. In such a simple task, the correspondence is straightforward, and the task dictionary simply encloses the 4 entries:

```
YES        [yes]  yes
NO         [no]   no
START_SIL  [sil]  sil
END_SIL    [sil]  sil
```

List. 6: Task dictionary.

The left elements refer to the names of the task grammar variables. The right elements refer to the names of the HMMs (introduced by `~h` in the HMM definition files). The bracketed elements in the middle are optional, they indicate the symbols that will be output by the recogniser: the names of the labels are used here (by default, the names of the grammar's variables would have been used.)

Remark:
Don't forget the new line at the end of the file (if not, the last entry is ignored).

6.2 Network

The task grammar (described in file **`gram.txt`**) have to be compiled with tool `HParse`, to obtain the *task network* (written in **`net.slf`**):

```
HParse -A -D -T 1   gram.txt   net.slf
```

At this stage, our speech recognition task (Fig.7), completely defined by its **network**, its **dictionary**, and its **HMM set** (the 3 models stored in `model/hmm3/`), is ready for use.

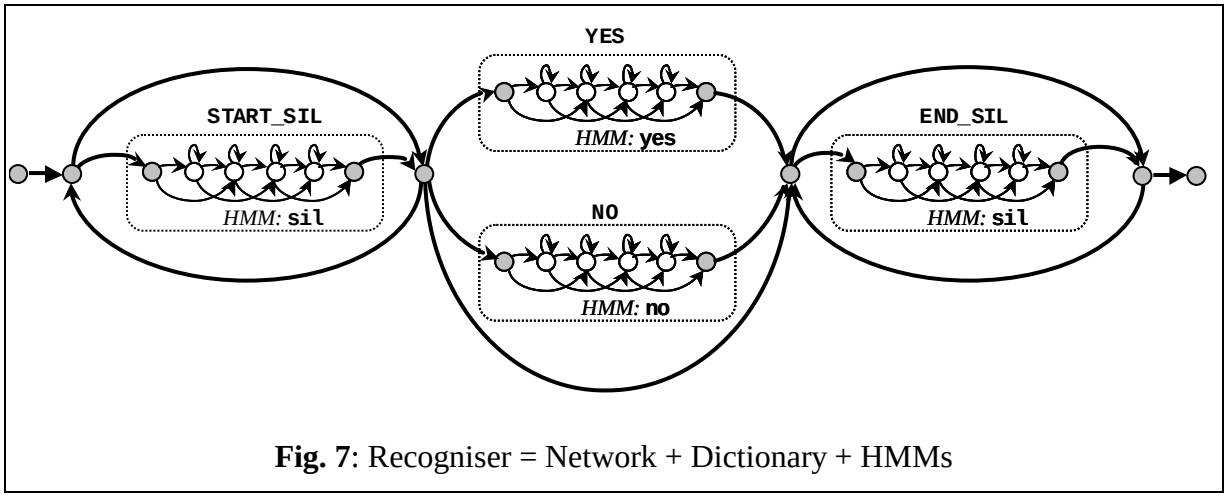

Fig. 7: Recogniser = Network + Dictionary + HMMs

Remark:
To be sure that no mistakes were made when writing the grammar, the tool `HSGen` can be used to test it:

```
HSGen -A -D  -n 10  -s net.slf  dict.txt
```

Where **dict.txt** is the task dictionary. Option **-n** indicates that 10 grammatically conform sentences (i.e. 10 possible recognition results) will be randomly generated and displayed.
Of course, it's not very useful here, but when grammars get more complicated, that can be very helpful…

7 Recognition

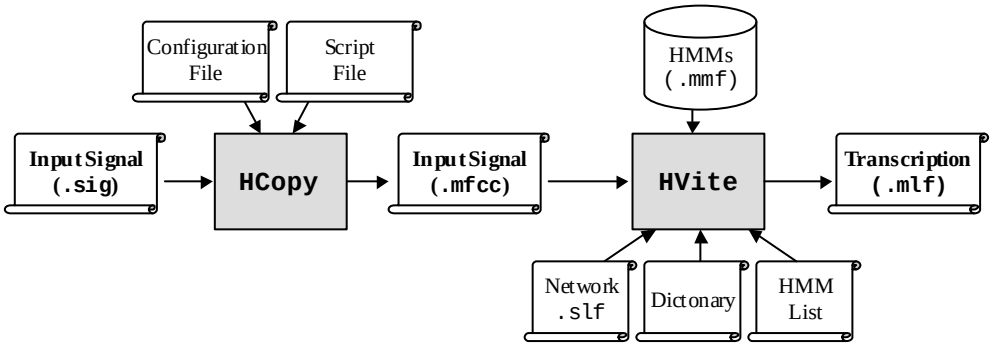

Fig. 8: Recognition process of an unknown input signal.

Let's come to the recognition procedure itself:

- An input speech signal `input.sig` is first transformed into a series of "acoustical vectors" (here MFCCs) with tool `HCopy`, in the same way as what was done with the training data (Acoustical Analysis step). The result is stored in an `input.mfcc` file (often called the *acoustical observation*).
- The input observation is then process by a Viterbi algorithm, which matches it against the recogniser's Markov models. This is done by tool `HVite`:

```
HVite -A -D -T 1  -H hmmsdef.mmf  -i reco.mlf  -w net.slf   \
      dict.txt  hmmlist.txt  input.mfcc
```

input.mfcc is the input data to be recognised.
hmmlist.txt lists the names of the models to use (`yes`, `no`, and `sil`). Each element is separated by a new line character. Don't forget to insert a new line after the last element.
dict.txt is the task dictionary.
net.slf is the task network.
reco.mlf is the output recognition transcription file.
hmmsdef.mmf contains the definition of the HMMs. It is possible to repeat the `-H` option and list our different HMM definition files, in our case:
`-H model/hmm3/hmm_yes  -H model/hmm3/hmm_no  -H model/hmm3/hmm_sil`
but it is more convenient (especially when there are more than 3 models) to gather every definitions in a single file (called a Master Macro File, with extension `.mmf`). Such a file is simply obtained by copying each definition after the other in a single file, without repeating the header information (see List.**7**).

```
      ~o <VecSize> 39 <MFCC_0_D_A>
      ~h "yes"
      <BeginHMM>
                  (definition...)
      <EndHMM>
      ~h "no"
      <BeginHMM>
                  (definition...)
      <EndHMM>
      ~h "sil"
      <BeginHMM>
                  (definition...)
      <EndHMM>
```

List. 7: Master Macro File (several HMM description in 1 file).

The output is stored in a file (`reco.mlf`) which contains the transcription of the input. If we use the file `data/train/mfcc/yes01.mfcc` as input data, for instance, we will get in `reco.mlf` such an output:

```
      #!MLF!#
      "../data/train/mfcc/yes01.rec"
      0 4900000 SIL -2394.001465
      4900000 12000000 YES -5159.434570
      12000000 18300000 SIL -3289.197021
      .
```

List. 8: Recognition output (recognised transcription).

In this example, 3 successive "word hypotheses" are recognised within the input signal. The target word "*Yes*" is correctly recognised. The start and end points of each hypothesis are given, along with their acoustic scores, resulting from the Viterbi decoding algorithm (right column).

A more interactive way of testing the recogniser is to use the "direct input" options of `HVite`:

```
HVite -A -D -T 1  -C directin.conf  -g  -H hmmsdef.mmf    \
      -w net.slf  dict.txt  hmmlist.txt
```

No input signal argument or output file are required in this case: a prompt `READY[1]>` appears on screen, indicating that the first input signal is recorded. The signal recording is stopped by a key-press. The recognition result is then displayed, and a prompt `READY[2]>` waiting for a new input immediately appears.
`-g` option allows to replay each input signal once.
`directin.conf` is a configuration file for `HVite`, allowing the use of direct audio input:

```
#
# HVite Configuration Variables for DIRECT AUDIO INPUT
#

# Parameters of the input signal
SOURCERATE = 625.0      # = 16 kHz
SOURCEKIND = HAUDIO
SOURCEFORMAT = HTK
```

```
# Conversion parameters of the input signal
TARGETKIND = MFCC_0_D_A      # Identifier of the coefficients to use
WINDOWSIZE = 250000.0        # = 25 ms = length of a time frame
TARGETRATE = 100000.0        # = 10 ms = frame periodicity
NUMCEPS = 12                 # Number of MFCC coeffs (here from c1 to c12)
USEHAMMING = T               # Use of Hamming function for windowing frames
PREEMCOEF = 0.97             # Pre-emphasis coefficient
NUMCHANS = 26                # Number of filterbank channels
CEPLIFTER = 22               # Length of cepstral liftering

# Defines the signal to be used for remote control
AUDIOSIG = -1                # Negative value = key-press control

# The End
```

List. 9: Configuration file for direct input recognition.

In order to allow direct extraction of the acoustical coefficients from the input signal, this file must contain the acoustical analysis configuration parameters previously used with the training data.

8 Performance Test

The recognition performance evaluation of an ASR system must be measured on a corpus of data different from the training corpus. A separate test corpus, with new "*Yes*" and "*No*" records, can be created in the `data/test/` directory as it was previously done with the training corpus. Once again, these data (stored in sub-directory `test/sig`) have to be hand-labelled (storage in `test/lab`) and converted (storage in `test/mfcc`).
(If you don't feel excited by the perspective of a new fastidious recording and labelling session, the training corpus may be used as test corpus here, since this tutorial's goal is just to learn how to use some HTK tools, not to get relevant performance measures…)

8.1 Master Label Files

Before measuring the performance, we need to create 2 files (called Master Label Files, with extension `.mlf`):

- The first one will contain the "correct" transcriptions of the whole test corpus, that is, the transcriptions obtained by hand-labelling. Let's call `ref.mlf` these reference transcriptions.
- The second one will contain the recognised transcriptions of the whole test corpus, that is, the hypothesised transcriptions yielded by the recogniser. Let's call `rec.mlf` these recognised transcriptions.

The performance measures will just result from the comparison between the reference transcription and the recognition hypothesis of each data.

A Master Label File has the following structure:

```
#!MLF!#
"path/data01.ext"
Label1
Label2
.
"path/data02.ext"
Label1
Label2
Label3
Label4
.
      (ETC...)
```

List. 10: Master Label File (several transcriptions in 1 file).

Each transcription is introduced by a file name and terminated by a period "`.`". A transcription consists of a sequence of labels separated by new-line characters. Optionally, each label can be preceded by start and end time indexes and /or followed by a recognition score (see List.8).

There is no HTK tool to create the reference transcription file `ref.mlf`. It must be written manually or with a script (see the `MakeRefMLF.pl` Perl script that I wrote, for instance).
The content of each label file (e.g. `yes01.lab`) has to be copied sequentially in `ref.mlf`, between the line giving the file name (e.g. `"*/yes01.lab"`: the path can be here replaced by a `*`) and the closing period.

The recognised transcription file `rec.mlf` can be obtained directly with `HVite`. This time `HVite` does not take a single input file name as argument, but the file names (`.mfcc`) of the entire test corpus, listed into a text file:

```
HVite -A -D -T 1  -S testlist.txt  -H hmmsdef.mmf  -i rec.mlf  \
      -w net.slf  dict.txt  hmmlist.txt
```

`hmmlist.txt`, **`dict.txt`**, **`net.slf`**, **`hmmsdef.mmf`**: the same as previously.
`rec.mlf` is the output recognition transcription file.
`testlist.txt` lists the names of the test files (`data/test/*.mfcc`).

After execution of the command, `rec.mlf` contains a series of transcription such as the one listed in List.8. Each transcription is introduced by the corresponding file name with a different extension (`.rec` instead of `.lab`).

8.2 Error Rates

The `ref.mlf` and `rec.mlf` transcriptions are compared with the HTK performance evaluation tool, `HResults`:

```
HResults -A -D -T 1  -e ??? sil  -I ref.mlf      \
         labellist.txt  rec.mlf  >  results.txt
```

`results.txt` contains the output performance statistics (example: List.11).
`rec.mlf` contains the transcriptions of the test data, as output by the recogniser.
`labellist.txt` lists the labels appearing in the transcription files (here: `yes`, `no`, and `sil`).
`ref.mlf` contains the reference transcriptions of the test data (obtained by hand-labelling).

`-e ??? sil` option indicates that the `sil` labels will be ignored when computing the performance statistics (since we are interested in the recognition rate of words "*Yes*" and "*No*" only).

```
====================== HTK Results Analysis =======================
  Date: Tue Dec 03 19:12:58 2002
  Ref : .\ref.mlf
  Rec : .\rec.mlf
------------------------ Overall Results --------------------------
SENT: %Correct=80.00 [H=8, S=2, N=10]
WORD: %Corr=80.00, Acc=80.00 [H=8, D=0, S=2, I=0, N=10]
===================================================================
```

List. 11: Results of a performance test.

List.11 shows an example of the kind of results that can be obtained. The first line (`SENT`) gives the sentence recognition rate (`%Correct=80.00`), the second one (`WORD`) gives the word recognition rate (`%Corr=80.00`).

In our case, the 2 rates are the same because our task grammar only allows "sentences" with one single word (apart from silences). It is an *isolated words* recognition task. Only the first line (`SENT`) should be considered here. `H=8` gives the number of test data correctly recognised, `S=2` the number of substitution errors (a "*Yes*" recognised as "*No*", or a "*No*" as "*Yes*") and `N=10` the total number of test data.

The statistics given on the second line (`WORD`) only make sense with more sophisticated types of recognition systems (e.g. *connected words* recognition tasks). For more details:
see HTK documentation, p.232 (Reference Section, HResults).

Hidden Markov Models (HTK)

Jan Černocký, FIT VUT Brno

This exercise deals with the isolated-word recognition using HMM. We will use the HTK toolkit from University of Cambridge (UK).

1 HTK

serves to define, train and recognize speech using HMM and contains tools for parameterization (feature extraction), evaluation of results, pronunciation dictionaries, and others. HTK is written in C-language and for non-commercial use, it can be downloaded from:
`http://htk.eng.cam.ac.uk/`
In this exercise, we will use a set of pre-compiled programs that we will run from the command line of Windows operating system:

- **HCopy** - as the name says, it should copy. While copying however, it can perform a conversion of speech data, for example from signal samples to MFCC vectors. The behavior is controlled by a configuration file.
- **HList** - visualizes (as text) a file with feature vectors.
- **HCompV** - initializes parameters of emission probability distribution functions (PDFs) in HMM states to global values for given word.
- **HRest** - retraining of model. It computes the values of the "soft" function assigning vectors to states (state occupation likelihood), followed by re-estimation of model parameters.
- **HParse** - converts human-readable form of recognition network to human-unreadable HTK format.
- **HVite** - Viterbi decoder or recognizer. For an unknown word, it computes the Viterbi probability of all models and finds the maximum. The model that "emitted" the given word with the maximum probability, wins.
- **HResults** - a tool for evaluation of recognition results – based on the correct transcriptions, it computes the word accuracy.

Running any of the programs without parameters shows a brief help.

2 The task

Create a speaker-independent recognizer for isolated words ANO/NE (yes/no in Czech). For the training, use data from 60 speakers from the Czech database "Číslovky" (each speaker has uttered both "ano" and "ne"). For testing, use data from 20 speakers.

Parameterization (feature extraction) should be done using 12 MFCC (Mel-frequency cepstral) coefficients and log-energy (in HTK notation MFCC_E). Complete the feature vector by the approximations of the 1st and 2nd derivative (Δ and $\Delta\Delta$ coefficients, in HTK notation MFCC_E_D_A). Set the frame-length to 25 ms, and the frame shift to 10 ms, you will therefore obtain 100 feature vectors per second.

Models will have left-right architecture, without state-skips. From i-th state, only transitions to i-th state and to $(i+1)$-th state are allowed. The models will have 7 states in total. The first and last are special non-emitting, there will therefore be 5 emitting states. The probability distribution function (PDF) in states will be a single Gaussian with diagonal covariance matrix. One PDF will therefore be described by a vector of 39 mean values and a vector of 39 variances.

3 Solution and comments

This section contains complete solution of the task. Details on the creation of lists, MLF files, etc., are in the enclosed `README` file[1].

[1] Many of the commands in README file will, however, run only under UNIX operating system...

Practical comments

Copy the contents of ..... to arbitrary local directory on your computer. The subdirectories contain:

- `cfg` — configuration files for HTK programs.
- `dics` — dictionaries.
- `net` — word networks for the recognition.
- `lists` — lists of models.
- `proto` — prototypes of models.
- `hmm0` — models initialized using `HCompV`.
- `hmm1` — models retrained using `HRest`.
- `data` — speech in raw format: no header, F_s=8000 Hz, 16-bit lin. Files with MFCC coefficients will be generated to the same directories. Files `*a0.raw` contain ANO, files `*a1.raw` contain NE.
- `mlf` — speech data transcriptions in Master-Label files.
- `scripts` — lists of files for HTK. The name 'scripts' is a bit misleading (scripts are usually batches of commands for operating system, for example `*.bat` under DOS). This notation is unfortunately common in the documentation of HTK, so that we will use it here, too.

Open a window with a command line (here, you will run HTK programs) and a File manager (for modifications and visualizations of text files).

3.1 Parameterization (Feature Extraction)

- Study the configuration file `cfg\hcopy.conf`:

```
BYTEORDER     = VAX        the byte order will be Intel-PC
SOURCEKIND    = WAVEFORM
SOURCEFORMAT  = NOHEAD     header-less files
SOURCERATE    = 1250       sampling period is 1250×100 ns = 1/8000
ZMEANSOURCE   = FALSE      no removal of DC offset

TARGETKIND    = MFCC_E     type of output features: MFCC and log-energy
TARGETFORMAT  = HTK
TARGETRATE    = 100000     sampling period of output feature vectors (frame shift) will be 10 ms
WINDOWSIZE    = 250000.0   frame length 25 ms
NUMCHANS      = 24         number of triangular filters used for the computation of MFCC
ENORMALISE    = TRUE       energy will be normalized.
```

- Study script-files `scripts\train.scp` a `scripts\test.scp`.
- Run the feature extraction for both training and test sets:
  `HCopy -T 1 -C cfg\hcopy.conf -S scripts\train.scp`
  `HCopy -T 1 -C cfg\hcopy.conf -S scripts\test.scp`
- Visualize one of created feature files as text (using `HList`) and in Matlab using `readhtk.m` function.

4 Model training

4.1 Initialization

- Study the prototypes of models in directory `proto`. Note, that allowed and forbidden transitions are "hard-wired" in the matrix of transition probabilities at the end of each model. Mean values are set to 0, variances to 1.
- Study the Master-Label file `mlf\train.mlf` The numbers before the label stand for beginning and end of file in hundreds of ns. For one file, check, if the length recorded in MLF corresponds to the file-size: time[100ns] = # of bytes / 2 / 8000 / 100×10^{-9}.

- Note, that the configuration file for model initialization `cfg\hcompv.conf` contains only one line:
  `TARGETKIND=MFCC_E_D_A`
  This means, that MFCC and energy coefficients (already on the disk in `*.mfc` files), will be on-line completed with Δ a $\Delta\Delta$ coefficients.

- Look at the script-file `scripts\train_htk.scp`

- Run the initialization for both models:
```
HCompV -T 7 -I mlf\train.mlf -l ANO -C cfg\hcompv.conf
       -m -S scripts\train_htk.scp -M hmm0 proto\ANO
HCompV -T 7 -I mlf\train.mlf -l NE -C cfg\hcompv.conf
       -m -S scripts\train_htk.scp -M hmm0 proto\NE
```

- Study resulting models in directory `hmm0`. What has changed?

4.2 Re-training of models

- Note, that the configuration file for re-training `cfg\hrest.conf` contains again only one line:
  `TARGETKIND=MFCC_E_D_A`

- Run the re-training of both models:
```
HRest -T 7 -I mlf\train.mlf -l ANO -C cfg\hrest.conf
      -S scripts\train_htk.scp -M hmm1 hmm0\ANO
HRest -T 7 -I mlf\train.mlf -l NE -C cfg\hrest.conf
      -S scripts\train_htk.scp -M hmm1 hmm0\NE
```

- Study resulting models in directory `hmm1`. What do you observe?

5 Recognition and evaluation

5.1 What else will we need

The results of training are two trained models in directory `hmm1`. We will however need a couple of other things:

- List of models. See file `lists\models`.

- Pronunciation dictionary. This dictionary contains the transcription of words in terms of models. In case we used smaller units (phonemes), it would contain for example: ANO=A N O. In our case, one word is modeled by one model, the pronunciation dictionary is therefore trivial: `dics\dictionary`.

- Recognition network. This network determines allowed sequences of words at the output of the recognizer. For us, this is ANO or NE. Hand-made and human-readable recognition network is in file: `net\oldnetwork`. For HTK, it is necessary to convert it to a human-unreadable form using:
  `HParse  net\oldnetwork net\network`

5.2 Recognition

for unknown files, it produces a transcription and stores it to MLF-file `mlf\testout.mlf`. The recognition is run using:
```
HVite -T 1 -C cfg\hvite.conf -d hmm1 -S scripts\test_htk.scp
      -i mlf\testout.mlf -w net\network dics\dictionary lists\models
```

- Look at the recognition results in the resulting Master Label file.

5.3 Evaluation

We are interested in the quality of the recognizer. In this experiment, we have a reference MLF with the correct transcription of test files: `mlf\test.mlf`. This can be compared to `HVite` output using:
`HResults -I mlf\test.mlf lists\models mlf\testout.mlf`
The most important number in the output of `HResults` is `Acc=` (word accuracy). How many % did you reach?

6 And more...

1. Record yourself a set of 10 WAV-files (8 kHz, 16 bit) containing ANO. For `HCopy`, use configuration file `hcopy_wav.conf`, which allows for reading of WAV-files. Create your own script-files for feature extraction and for the recognition. Create your own reference MLF file (in case MLF is used only as a reference for `HResults`, it is not necessary to fill the beginning and end times). Extract the features using `HCopy` and recognize using `HVite`. Evaluate the recognition accuracy using `HResults`. How many % did you reach?

2. In Matlab, add white noise to your files so that the signal-to-noise ratio (SNR) is 0 dB. You may for example move the original files to `xx_clean.wav` and then use the following sequence of Matlab-commands:

```
SNR = 0;
[s,fs,nbit] = wavread('xx_clean.wav');
s = s' - mean(s);
E = sum(s.^2) / length(s);              % energy of the signal
Enoise = E / 10^(SNR/10);               % energy of the noise
n = randn(1,length(s)) * sqrt(Enoise); % generating the noise
wavwrite (s + n, fs, nbit,'xx.wav');   % writing signal+noise to disk
```

   Listen to the resulting files (note, that for SNR=0 dB, the energy of noise is the same as the energy of signal!). Extract MFCC features, recognize and evaluate the word accuracy. What is your result?