Automatic Speech Recognition 2011-12: Lab-session sheet (4) — Continuous speech recognition (part 2 of 3) —

Hiroshi Shimodaira (*Revision* : 1.3)

In this lab-session we will

- increase the number of Gaussian mixture components in the pdfs for the monophone models
- see what effects that has on accuracy
- convert monophone models into triphone (i.e. context dependent) models using a decision tree-based clustering technique.
- convert the single Gaussian triphone HMMs into multiple mixture component triphone HMMs.
- see what effects those have on accuracy

1 Updating your environment

To make your experimental environment up-to-date, please re-run the initialisation command we used last week:

```
% cd WorkDir [1]
% /afs/inf.ed.ac.uk/group/teaching/asr/bin/init-t3
```

2 Multiple mixture component monophone models

After you have carried out the embedded training of single Gaussian monophone models several times to get hmm9, it's time to increase the number of mixture components of each state. This is done by iterative "mixture splitting" with the HHEd command.

2.1 Copying source models to a new directory

At first, you need to determine the model number to increase the mixture component. If it is "`models/R1/hmm9`" [2], do as follows (here I assume your current directory is *WorkDir*):

```
% mkdir models/R2                                  ··· we are going to use a new directory, R2.
% mkdir models/R2/hmm0                  ··· create a directory where source models are stored.
% cp models/R1/hmm9/MODELS models/R2/hmm0                  ··· copy the monophone models.
```

2.2 Splitting

Run the following script, which calls HHEd, to split a single Gaussian to two components.

```
% ./scripts/mixup
```

This automatically creates a new directory, `models/R2/hmm1`, for new models.

You can take a look at the new models to see what changes have been made:

```
% less ./models/R2/hmm1/MODELS
```

2.3 Retraining

The new models above need retraining. Run the following command to run HERest for three times to get models up to hmm4.

```
% ./scripts/train_monophones_realigned -R2 1 2 3
```

The option `-R2` is essential, as it tells the script to use the models under `R2` rather than the default directory `R1`.

[1] `/afs/inf.ed.ac.uk/group/teaching/asr/Work/` *YourLoginName*

[2] Further experimental results will depend on which model you've chosen here. It would be interesting to see how different performance you will get depending on the initial model number.

2.4 Evaluation

Test the new models in `models/R2/hmm4` with the following command:

```
% ./scripts/recognise_with_monophones_small -R2 4
```

which will produce recognition output:

```
recognised/R2/hmm4_si_dt5a-div3_output.mlf
```

and log files:

```
logs/recognise/R2/log-last                  ··· a whole log of the command invoked last
logs/recognise/R2/log-4-si_dt5a-div3  ··· digest of the log, which is also shown on display
```

Make sure that you have not got the warning message of "No token survived..." many times (a few times should be fine).

Run the following command to get recognition accuracy.

```
% ./scripts/show_results_small -R2 4
```

Are the new models more accurate than the single-Gaussin component models from part 1? (It would be interesting to evaluate other previous models, e.g. hmm1, hmm2, hmm3.)

It should be noted that those script files with "`_small`" in name indicate that the script uses a small test set, "`file_lists/si_dt5a-div3.scp`" (123 utterances) rather than the subset of "`file_lists/si_dt5a.scp`" (368 utterances). Using a small test-set can make the evaluation much faster, but it might loose accuracy in estimating recognition performance. However, we need to carry out a large number of experiments in the lab sessions, we will use this small subset for evaluation.

2.5 Increasing more number of mixture components

We can repeat the above splitting and retraining operation to get models with more number of mixture components.

The question is how we should change the numbers of mixture components along with repetitive training. Recall the local-optimum problem with HMM parameter estimation - it would not be a good idea that we increase the number of mixture components very quickly, e.g. $1 \rightarrow 10 \rightarrow 50$. The following table shows an example of the process:

# of mix. components	splitting	retraining	final model
(1)	-	-	(R1/hmm9)
2	hmm0 → hmm1	hmm1, hmm2, hmm3	hmm4
3	hmm4 → hmm10	hmm11, hmm12, hmm13	hmm14
4	hmm14 → hmm20	hmm21, hmm22, hmm23	hmm24
5	hmm24 → hmm30	hmm31, hmm32, hmm33	hmm34
7	hmm34 → hmm40	hmm41, hmm42, hmm43	hmm44
10	hmm44 → hmm50	hmm51, hmm52, hmm53	hmm54
⋮	⋮	⋮	⋮

Try increasing the number of mixture components further to see how recognition accuracy/speed changes, and draw graphs[3]. To that end, read the next section 2.6 at first, and copy "`scripts/mixup`" to your own script[4], e.g. "`scripts/my-mixup`", and edit the script to change some definitions. You will also need to copy "`edfiles/mixup2.hed`" to your own edit command file, e.g. "`edfiles/my-mixup.hed`".

Go over your notes and make sure you understand what you just did, what mixtures of Gaussian are, and why increasing the number of mixture components improves accuracy.

[3] It is a good idea that you compare the graph of likelihood on training data and the graph of recognition accuracies on test data.

[4] You will find another script, `scripts/mixups`, which is a more sophisticated version of the script, and does not require you to prepare the `mixup.hed` instruction file.

2.6 Details on mixup and HHEd commands

The "mixup" script calls HHEd in the following manner:

```
HHEd -C configs/config.basic -T 1 \
    -H models/R2/hmm0/MODELS \
    -M models/R2/hmm1 \
    edfile/mixup2.hed \
    model_lists/mono-sp.list
```

Here `models/R2/hmm0/MODELS` is used as source models, and a new model file is stored in the directory `models/R2/hmm1`. Operation (edit) commands such as increasing the number of mixture components should be given to HHEd through a file, which is specified as the 1st argument to HHEd.

As you will see, the edit command file, "`edfile/mixup2.hed`", consists of only one line:

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

where "2" followed by "MU" is the number of Gaussian mixture components you want to have for each state (i.e. 2, 3, and 4) of all the models.

3 Decision tree-based tied-state triphone models

We can improve the monophone models by making them context dependent. Among several types of training algorithms for context-dependent models, we will employ a decision tree-based clustering technique to have triphone models whose parameters are effectively shared, i.e. "tied".

The procedure will look like this

Step 1a create triphone labels and triphone model list.

Step 1b create single-Gaussian cross-word triphones by cloning single-Gaussian monophone models.

Step 1c train single mixture cross-word triphones.

Step 2a carry out clustering to create single-Gaussian tied-state triphone models from the models above.

Step 2b train the tied-state triphone models.

Step 3 split Gaussian component to get multiple-mixture tied-state triphone models, and train and evaluate the models.

3.1 Step 1: Cross-word triphone models

3.1.1 Step 1a: preparing triphone labels

Run the following script to convert phone labels into context dependent form and also create a list of triphone models that appear in the training data.

```
% ./scripts/mk_xwrd_labels
```

Make sure that a context dependent (triphone) label file, `labels/phone/xwrd_tri.mlf` has been created from `labels/phone/mono-aligned2.mlf` and a list of triphone models, `model_lists/xwrd_tri.list` has been created as well.

3.1.2 Step 1b: creating cross-word triphone models

Run the following script to create initial cross-word triphones by simply cloning monophone models in `models/R1/hmm9`.

```
% ./scripts/mk_xwrd_triphones
```

The initial model will be stored under `models/R7/hmm0`.

Because the total file size of triphone models is much larger than that of monophone models, model files hear-after will be stored in binary format rather than ascii format for efficiency, and thus you cannot take a look at them directly.

3.1.3 Step 1c: training cross-word triphone models

Run the following command to train the triphones up to hmm2 (or further).

```
% ./scripts/train_xwrd_triphones 0 1 2
```

The cross-word triphone models created so far are only those triphones that appeared in the training set, and thus a lot of triphones might be missing.

3.2 Step 2: Single-Gaussian tied-state triphone models

3.2.1 Step 2a: clustering HMM states

Run the following script to carry out top-down clustering of HMM states

```
% ./scripts/mk_tied_triphones
```

which will produce a long message both on display and to a file:

```
% less logs/train/R9/log-hhed
```

You will find a line like this:

```
TB: Stats 24->1 [4.2%]  { 60309->2086 [3.5%] total }
```

which shows that a total of 60,309 HMM states were merged into 2,086 clustered.

The script above creates those lists:

`model_lists/full_tri.list`	list of all possible triphone models
`model_lists/trees`	current questions defined and trees
`model_lists/treeg.list`	phyical models and associated logical models

See HTKBook[section 3.3.2, 10.5, and 17.8] for details, and see the script file, `scripts/mk_tied_triphones` to understand how HHEd is called for clustering.

3.2.2 Step 2b: training tied-states triphone models

```
% ./scripts/train_tied_triphones 0 1 2 3
```

3.2.3 Step 2c: evaluation

Run the following scripts.

```
% ./scripts/recognise_with_tied_triphones_small 4
```

```
% ./scripts/show_results_small -R9 4
```

3.3 Step 3: Mixup – multiple mixture triphone models

Much the same way for the monophone models,

```
% ./scripts/mixup_tied
```

which will read in `models/R9/hmm4` and store the new models with two mixture components into `models/R9/hmm10`.

```
% ./scripts/train_tied_triphones 10 11 12 13
```

Copy `mixup_tied` to your own file to change the number of mixture components and model numbers, and repeat the procedure above.

For evaluation, the same evaluation scripts shown in Step 2c can be used.

4 Implementation of decoding in HTK

ASR system uses $P(X|W)$ and $P(W)$ to find the best word sequence W.

- These estimated probabilities are different in reliability and dynamic range.
- More shorter words tend to have higher scores than those with fewer longer words in real implementation.

$$
\begin{aligned}
W^* &= \arg\max_W P(X|W)\,P(W) \\
&\Rightarrow \arg\max_W P(X|W)\,P(W)^{LMS}\,IP^{N(W)} \qquad \cdots \text{ modified formula} \\
&= \arg\max_W \log P(W|X) + LMS \log P(W) + N(W) \log IP
\end{aligned}
$$

LMS :	language model weight (LM scaling factor)
IP :	insertion penalty
$N(W)$:	number of words in W

- Interpretation of LMS
  As $LMS \to 0$, $P(W)^{LMS} \to 1$,
  i.e. the smaller LMS becomes, the less important the LM is.

- Interpretation of IP
  Assuming a uniform LM (every word has an equal occurrence probability), $P(W)$ for fewer longer words (i.e. smaller $N(W)$) is greater than $P(W)$ for more shorter words (i.e. larger $N(W)$). IP is used to balance this.

$0 < IP \leq 1$ (i.e. $\log IP < 0$)	the smaller IP becomes, the more shorter words are penalised (i.e. fewer longer words are preferred)
$1 < IP$ (i.e. $\log IP > 0$)	the larger IP becomes, the more shorter words are preferred. (i.e. the more insertion errors.)

- LMS and IP are determined heuristically (on validation data). Larger LMS usually needs larger IP.