Lab 7: Training monophone models

University of Edinburgh

March 20, 2022

Last time we got familiar with some of Kaldi's tools, we set up a data directory for the WSJ corpus, and extracted features. This time we will train Hidden Markov Model-Gaussian Mixture Model (HMM-GMM) systems on top of those features. Commands that you should run are in a box with a red border, and notes in a box with a blue border:

```
Code to execute will appear in red boxes.
```

Notes will appear in blue boxes.

1 Building an acoustic model

Let's begin by opening a terminal window and `cd` to your workdir:

```
cd ~/asr_labs/wsj
source path.sh
```

Before we start building the HMM-GMM models run the following script to check that you have the necessary files from the previous lab:

```
./local/lab1_check.sh
```

If that says everything is fine then continue to the next step.

1.1 Monophone models

Let's start training a monophone system. We will first take the 2000 shortest utterances from the dataset to train the system on.

```
utils/subset_data_dir.sh --shortest data/train 2000 data/train_2kshort
```

Kaldi has a script called `steps/train_mono.sh`. If we run it without any arguments we get the following usage message:

```
Usage: steps/train_mono.sh [options] <data-dir> <lang-dir> <exp-dir>
```

Kaldi needs a data directory and a language directory, and will store the model in the experiment directory. Plugging in for those, run the following command:

```
steps/train_mono.sh  --nj 4 data/train_2kshort data/lang exp/mono
```

The option `--nj 4` instructs Kaldi to split computation into four parallel jobs. This can significantly reduce computation time, but only as long as there are at least equal number of processor cores which are not in use.

To monitor running processes, try using `top` or `htop` in a second terminal window. While running the monophone training you should see at least four running programmes.

Building the monophone system may take a while, so open another terminal window and `cd` again to the workdir:

```
cd ~/asr_labs/wsj
source path.sh
```

A folder we didn't look at much last time was the language directory, `data/lang`. Let's begin to have a look at the phones that are defined in our model:

```
less data/lang/phones.txt
```

Note that our phoneset is word-position dependent, so there's going to be an `UH_B` for when it's at the beginning of a word and an `UH_E` for when it's at the end. How many phones have we defined in our model? (hint: use `wc`)

Similarly, the following file lists the words in our language model.

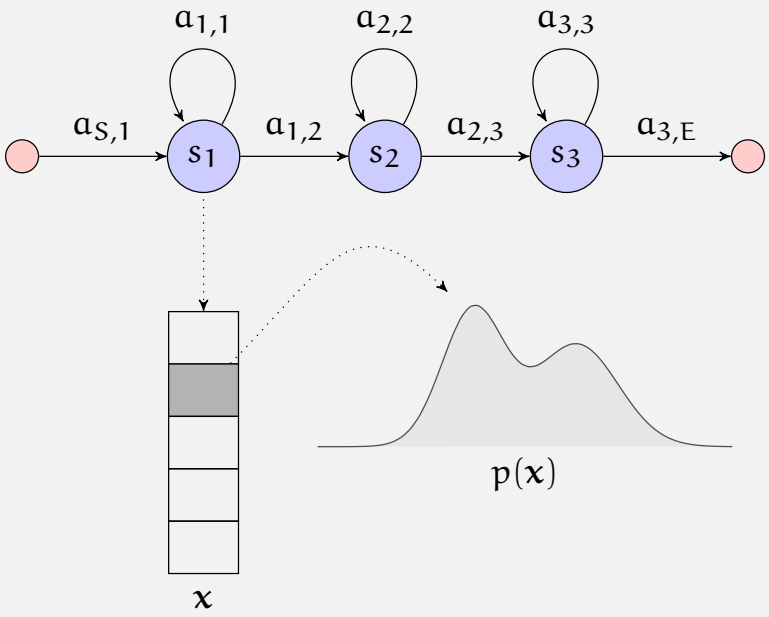
```
less data/lang/words.txt
```

Next, have a look at the topology:

```
less data/lang/topo
```

This sets out normal three-state HMMs per phone. The transition probabilities are defined after the <`Transition`> tags.

Recall from the lectures how we use HMMs and GMMs to model the data. Relate this figure to the topology and the acoustic model generated in Kaldi.



Compare the topology to the phones we just looked at. Which phone has a different topology? How is it different? Can you relate the topology to the figure above?

By now the initial monophone model should have been written. Inspect it by running the following command:

```
gmm-copy --binary=false exp/mono/0.mdl - | less
```

The top should look familiar. Underneath the topology there will be a <Triples> tag. This maps each phone and one of its three states (or a noise/silence and one of its five states) to a unique number. That is, there are $num\_phones \times num\_phone\_states + num\_noises \times num\_noise\_states = 1083$ triples. Scroll down a few hundred lines more until you see tags such as <DiagGMM>. Can you work out what kind of information is stored here? How does it relate to the `Triples` in the topology? (How many PDFs are there?)

Last time we mentioned that binaries that take read or write *specifiers*, such as `ark:`, write in binary by default unless you append the `,t` flag (e.g. `ark,t:`). For binaries that read normal files, you instead provide the flag `--binary=false`, as above.

To get a summary of the information in the initial acoustic model, write

```
gmm-info exp/mono/0.mdl
```

When the monophone models are finished training, have a look at the final trained model:

```
gmm-info exp/mono/final.mdl
```

What has changed?
We can have a look at some of the mixture Gaussians by running:

```
gmm-copy --binary=false exp/mono/final.mdl - |\
  python2.7 local/plot_gmm.py -
```

Note the backslash at the end of the line: \ . This allows you to write bash expressions over multiple lines and can help readability in scripts.

If you get errors that you are missing a Python library, you can install these libraries by running:

```
pip2.7 install --user numpy scipy matplotlib backports
```

You can safely ignore any runtime errors that "numpy.dtype size changed".

You should see something similar to Figure **??**. These are the Gaussian Mixture Model densities corresponding to the first twelve cepstral coefficients for one of the states in the model.

This is a good time to dig a little bit deeper into the scripts. Have a look at the monophone training script:

```
less steps/train_mono.sh
```

At the top there are listed some default parameters. About halfway through there's a parameter called `totgauss`. This sets the upper limit of the number of Gaussians to include in the final trained model. Search for the string `gmm-est` by typing (within `less`):

```
/gmm-est
```

This is the command that actually estimates the GMMs. Notice that the script is passing in a mix-up parameter, which is based upon the `totgauss` variable above. Leave `less` by pressing `q`.

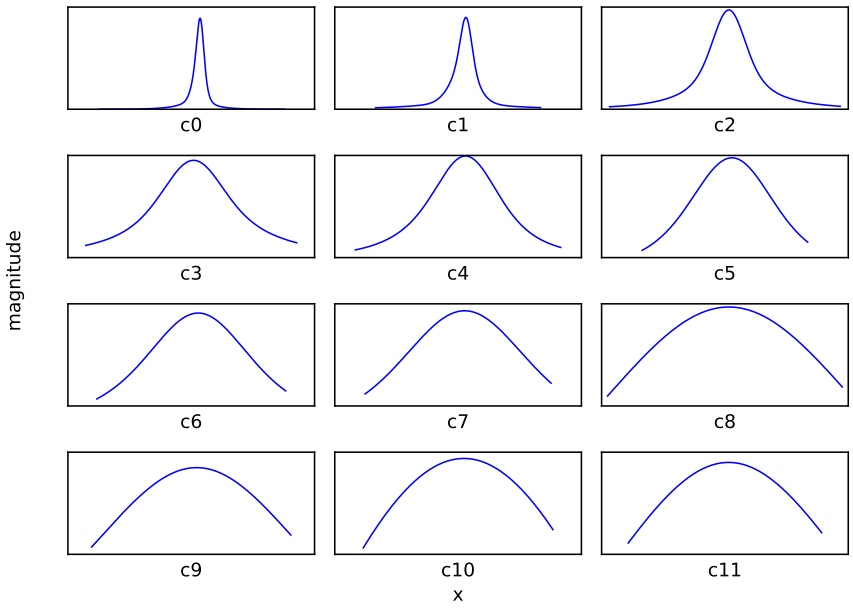


Figure 1: GMMs for the first 12 cepstral coefficients for a random density.

When you're using `less` to look at a file, you can search for text in that file by typing `/` followed immediately by the string you're interested in and hitting enter. Press `n` repeatedly to loop through all occurrences of the search string.

Lets start decoding our monophone model. Run the following commands:

```
utils/mkgraph.sh data/lang_test_bg exp/mono exp/mono/graph

steps/decode.sh --nj 4 \
 exp/mono/graph data/test exp/mono/decode_test
```

While we're decoding, in the other window, we'll look at the alignments from our monophone model. The alignments are stored compressed using `gzip`. Note that in the below command we unzip them and pipe them directly in as the read specifier. We then apply the `,t` tag to get readable output from the write specifier and finally we output only the first line.

```
convert-ali --reorder=false exp/mono/final.mdl \
  exp/mono/final.mdl exp/mono/tree \
ark:"gunzip -c exp/mono/ali.1.gz |" ark,t:- | head -1
```

These numbers are called transition-ids. To see how they relate to our model, type

```
show-transitions data/lang/phones.txt exp/mono/0.mdl
```

Can you see how this relates to the topology we looked at above? What happens if you look at the final trained model compared to the initial (0) model? That is, try the following command and compare the output to the previous command:

```
show-transitions data/lang/phones.txt exp/mono/final.mdl
```

What has changed?

We can combine this information and display it in a more friendly way. Type the following command:

```
show-alignments data/lang/phones.txt exp/mono/final.mdl \
  "ark:gunzip -c exp/mono/ali.1.gz |" | head -n 2
```

The first line now contains the same transition ids, but grouped in a particular way (how?). The second line shows the phones.

(It is perfectly fine to skip this box)

Above we used the command `convert-ali` to view the alignments stored in `ali.1.gz`. However, if we just wanted to see exactly what was in the file, we could have just used:

```
copy-int-vector "ark:gunzip -c exp/mono/ali.1.gz|" \
  ark,t:- | head -n 1
```

Note that the output is different:

```
2 4 3 3 3 3 3 3 6 5 5 5 5
```

compared to

```
2 3 3 3 3 3 3 4 5 5 5 5 6
```

Here, id 4 indicates a transition from state 1 to 2, and id 3 indicates a self-loop in state 1. So in the first example we transition to state 2, and then perform a self-loop in state 1. Kaldi *reorders* transition probabilities, effectively placing the current state's self-loops at the end of the forward transition to the next state. This is done to optimise decoding later on, but doesn't affect the actual probabilities or results.

1.2 Decoding

Above we set off some jobs creating a graph and decoding the model. For this we're using a slightly different language directory which contains a bigram model of the phones, stored in `data/lang_test_bg/G.fst`. The first command above (`utils/mkgraph.sh`) combines the HMM structure in the trained model, any Context dependency, the Lexicon and the Grammar - collectively termed HCLG - and creates a *decoding graph* in the form of a Finite State Transducer (FST)[1]. The second script generates lattices of word sequences for the data given the model, which we then score.

```
grep WER exp/mono/decode_test/score_*/wer_* | utils/best_wer.sh
```

What Word Error Rate does the monophone model achieve on WSJ?
We're done! Next time we'll look at triphone neural network models.

1.3 Appendix: Common errors

- Forgot to source `path.sh`, check current path with `echo $PATH`
- No space left on disk: check `df -h`
- No memory left: check `top` or `htop`
- Lost permissions reading or writing from/to AFS: run `kinit && aklog`. To avoid this, run long jobs with the `longjob` command.
- Syntax error: check syntax of a Bash script without running it using `bash -n scriptname`
- Avoid spaces after \when splitting Bash commands over multiple lines
- Optional params:
- command line utilities: `--param=value`
- shell scripts: `--param value`
- Most file paths are absolute: make sure to update the paths if moving data directories
- Search the forums: `http://kaldi-asr.org/forums.html`
- Search the old forums: `https://sourceforge.net/p/kaldi/discussion`

[1] [?] is a good, related paper on FSTs in speech recognition.

1.4 Appendix: UNIX

- `cd dir` - change directory to `dir`, or the enclosing directory by `..`
- `cd -` - change to previous directory
- `ls -l` - see directory contents
- `less script.sh` - view the contents of `script.sh`
- `head -l` and `tail -l` - show first or last `l` lines of a file
- `grep text file` - search for `text` in `file`
- `wc -l file` - compute number of lines in `file`