Lab 3: Word recognition and triphone models

University of Edinburgh

February 6, 2019

Note: At the end of this lab (Section 1.7) we will ask you to create an account for access to Google Cloud computing. If you don't make it through the entire lab, please make sure you go through the steps in that section before the next lab.

So far we have been working with phone recognition on TIMIT. In this lab we will move instead to word recognition. This requires a word-level language model, for which we will be using the Wall Street Journal (WSJ) language model.

Path errors

If you get errors such as `command not found`, try sourcing the path again:

```
source path.sh
```

In general, every time you open a new terminal window and `cd` to the work directory, you would need to source the path file in order to use any of the Kaldi binaries.

There is an appendix at the end of every lab with the most typical mistakes.

1 Word recognition

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

```
cd ~/asrworkdir
```

```
source path.sh
```

Before we start building new models run the following script to gather some new files we need to do word recognition:

```
./local/lab3_setup.sh
```

The message "`lab 3 preparation succeeded`" should appear.

1.1 Word-level language model

When you ran the above command it will have generated a new language model directory: `data/lang_wsj`, new training and test directories, and a monophone experiment directory that we have trained for you already.

There's a crucial difference to the new language model directory from the previous one. Compare the `words.txt` file in each:

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

```
less data/lang_wsj/words.txt
```

What is different between the two? Let's also have a look at the new phone-set:

```
less data/lang_wsj/phones.txt
```

Our phones are now dependent upon where they occur in the words. For an explanation of the suffixes, see

```
less data/lang_wsj/phones/word_boundary.txt
```

To see how this relates to the lexicon, open the following file:

```
less data/lang_wsj/phones/align_lexicon.txt
```

Now search (remember `/`) for "`^ SPEECH`", which will show the first occurrence of a word that starts with "`SPEECH`". How does the mapping relate to the position-dependent phones we saw above?

Finally, have a look at the new training directory which now maps from utterance to words instead of phones:

```
less data/train_words/text
```

1.2 Monophone models

We have provided monophone models and alignments, as well as a decode on the test set. These are placed in `exp/words/mono` and `exp/words/mono_ali`.

Have a look at the score for the decode on the test set:

```
more exp/word/mono/decode_test/scoring_kaldi/best_wer
```

more

The command `more` is very similar to `less`, apart from that it prints the output to the console directly.

Ok! There's lots of room for improvement...

1.3 Triphone models

Let's start training a triphone model with delta and delta-delta features.

Backslashes

Backslashes in Bash (\) which is present in the next box, are simply a way of splitting commands over multiple lines. That is, the following two commands are identical:

```
some_script.sh --some-option somefile.txt
```

and

```
some_script.sh --some-option \
  somefile.txt
```

You may remove the backslash and type these commands on a single line. But sometimes when writing scripts, avoiding too long commands on a single line can help readability.

Be careful about spaces after \. Bash will expect a newline immediately, and a space here before the newline will make a script crash.

Run the following command to train a triphone system. This might take a few minutes...

```
steps/train_deltas.sh 2500 15000 data/train_words \
  data/lang_wsj exp/word/mono_ali exp/word/tri1
```

- While that is running, open another terminal window, change directory to the work directory and source the path.

1.4 Delta features

Above we started running a script called `train_deltas.sh`. This trains triphone models on top of `MFCC+delta+deltadelta` features. To avoid having to store features with `delta+deltadelta` applied, Kaldi adds this in an online fashion,

just as another pipe, using the programme `add-deltas`. Remember how we checked the feature dimension of the features in the first lab? Run the following command.

```
feat-to-dim scp:data/train/feats.scp -
```

What do you expect the dimension to be be after applying `add-deltas`? Run the following command.

```
add-deltas scp:data/test/feats.scp ark:- | feat-to-dim ark:- -
```

1.5 Logs

Kaldi creates detailed logs during training. These can be very helpful when things go wrong. By now we should have some of the first ones created for the triphone training:

```
less exp/word/tri1/log/acc.1.1.log
```

Notice that on the top, the entire command which Kaldi ran (as set out by the script) is displayed. For this example it runs a command called `gmm-acc-stats-ali`, and then if you look closely there is a feature pipeline using the programmes `apply-cmvn` and `add-deltas` which applies these transforms and additions to the features in an online fashion.

1.6 Triphones

When we ran the triphone modelling script above we also passed two numbers, 2500 and 15000. These are respectively the number of leaves in the decision tree and the total number of Gaussians across all states in our model.

Triphone clustering

As you may recall from the lectures, having a separate model for each triphone is generally not feasible. With typically 48 phones we would require $48 \times 48 \times 48$, i.e. more than 110000 models. We don't have enough data to see all those, so we cluster them using a decision tree. The *number of leaves* parameter then sets the maximum number of leaves in the decision tree, and the *number of gaussians* the maximum number of Gaussians distributed across the leaves. So on average our model will have an average of $\frac{numgauss}{numleaves}$ Gaussians per leaf.

To see how many states have been seen, run the following command:

```
sum-tree-stats --binary=False - \
  exp/word/tri1/1.treeacc | head -1
```

The first number indicates the number of states with statistics. Dividing that number by three will roughly give the number of seen triphones (why is this?).

Let's have a closer look at the clustering in Kaldi. It's also a good opportunity to look a bit deeper at the Kaldi scripts. Open the training script we just ran by typing

```
less steps/train_deltas.sh
```

At the top is a configuration section with several default parameters. These are the parameters that the script can take by passing `--param setting` to the script when running it. Most scripts in Kaldi are set up this way. The first one is a variable called `stage`, this can be really useful to start a script partway through. Scroll down till line 88 which says

```
if [ $stage -le 2 ]; then
```

This is saying that if the stage variable is less than or equal to two, run this section. This is the section that builds the decision tree. It first calls a binary called `cluster-phones`, this uses e.g. k-means clustering to cluster similar phones. These clusters will be the basis for the questions in the decision tree. Kaldi doesn't use predefined questions such as "is the left phone a fricative?", but rather estimates them from the data. It writes these using the numeric phone identities to a file called `exp/word/tri1/questions.int`. Let's look at it using a utility that maps the integer phone identities to their more readable names. Leave `less` by pressing `q` and run the next command:

```
utils/int2sym.pl data/lang_wsj/phones.txt \
  exp/word/tri1/questions.int | less
```

Each line is a cluster. Some of these clusters are really large, but hopefully some of them should make sense. We use these to build a clustering tree, stored in `exp/word/tri1/tree`. Exit `less` and run the following command:

```
copy-tree --binary=false exp/word/tri1/tree - | less
```

This file sets out the entire clustering tree. We'll only get a gist of what it represents by looking at a few lines. The further down this file, the deeper we move into the tree. Move down to line 50 by typing the following letters on the keyboard while in `less`:

```
50g
```

You should see a line that says

```
SE 0 [ 40 41 42 43 92 93 94 95 96 97 98 99 228 229 230 231 ]
```

The numbers in the brackets are phones. By now you should be able to figure out which phones these represent (hint: language directory). `SE` stands for "SplitEventMap", which is essentially a split in the tree. The following number is typically either 0, 1, or 2, and they stand for left, centre and right. That is, for this line we are asking whether the *left* (0) phone is one of 40, 41, 42, ..., 231. If that is true, we

- proceed to the next line, asking whether the left phone is one of `[40,41,42,43]`, if that is true we
- proceed to the next line, this time asking whether the *right* (2) context is one of `[1,2,3,4,5]` (which phone is this?)

Finally, we proceed to the next line where `CE` stands for a ConstantEventMap and indicates a leaf in tree. If our previous question was true then we choose pdf-id 1302, and if not, we choose 1091. The subtree we have been looking at is illustrated in Figure 1. Can you tell how we would get pdf-id 1039?

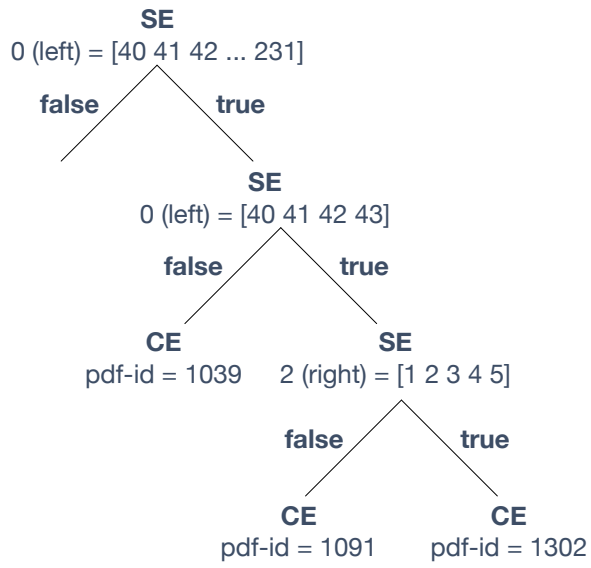

Figure 1: Tree clustering example

Ok, by now the triphone system should have finished training. Let's create a decoding graph and decode our triphone system, this may take a few minutes:

```
utils/mkgraph.sh data/lang_wsj_test_bg \
  exp/word/tri1 exp/word/tri1/graph
```

```
steps/decode.sh --nj 4 exp/word/tri1/graph \
  data/test_words exp/word/tri1/decode_test
```

When the decoding have finished, score the directory by running:

```
local/score_words.sh data/test_words exp/word/tri1/graph \
  exp/word/tri1/decode_test
```

We can then have a look at the WER by typing:

```
more exp/word/tri1/decode_test/scoring_kaldi/best_wer
```

That should be considerably better than the monophone system, but there's still lots of room for improvement. The typical progression in Kaldi would now be to train a system on top of decorrelated features and then train a system with speaker adaptive training. But that's for another lab...

We're (almost) done! Next time we'll look at training hybrid neural network models. So as a last step, we need to prepare a setup with Google Cloud that we will use next time.

1.7 Google Cloud setup

The next lab will use Google Cloud so that we can run neural networks on GPUs. It's convenient to create the account now, should there be any problems. To create your account follow these steps:

1. Get your coupon by following the instructions in the coupon retrieval link.
2. Once you receive your coupon, follow the email instructions to add your coupon to your account.
3. Once you have added your coupon, join the ASRLab-2019 Google Group using the same Google account you used to redeem your coupon. This ensures access to the shared disk images.
4. Once logged in, click on Projects (on the left hand side of the search bar on top of the page) Name your project sxxxxxxx-ASRLab - replacing the sxxxxxxx with your student number. For billing, choose the ASR course.
5. Make sure that the financial source for your project is the ASRLab credit by clicking the 3 lines (hamburger)§ icon at the top left corner and then clicking billing → go to linked billing account.
6. If it's not set to the ASRLab credits then set it by going to billing → manage billing accounts → My projects. Click the 3 dots under the Actions column for the relevant project and click change billing account. Select the ASRLab credit from your coupon.

7. Start the project

That's it! See you next time.

1.8 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

1.9 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`