Kaldi-notes

Some notes on Kaldi

View on GitHub

This is an introduction to speech recognition using Kaldi. Follow one of the links to get started.

A PDF snapshot of this site/manual is available. Be aware that all link within the pdf go to the website.

Kaldi-notes is maintained by oxinabox
This page was generated by GitHub Pages. Tactile theme by Jason Long.

Kaldi-notes

Some notes on Kaldi

Other Resources

This document is quiet interlinked to other resources, as appropriate for each section and subsection. These and some father resources are summeriest below.

OpenFST

- OpenFST Documentation
  - Quick Tour
  - Examples

- Tutorial Sheet from University of Illinois
- Lecture Slides from University of Tokyo
- Speech Recognition with Weighted Finite-state Transducers (book chapter)

Kaldi

- Kaldi Documentation
  - Tutorial

- Several pages by Vassil Panayotov
  - This blog post on Graph construction
  - These instructions on recipe setup
    - This tutorial by the same author extends the above. But its web hosting does not seem stable, right now the google cached version can be used

- This Masters Thesis
- Lecture slides from National Taiwan Normal University
- Lecture Slides from Dan Povey (one of kaldi's creators)

Kaldi-notes is maintained by oxinabox
This page was generated by GitHub Pages. Tactile theme by Jason Long.

Kaldi-notes

Some notes on Kaldi

Required Knowledge

To make use of Kaldi, there is some significant prior required knowledge.

- Bash
- Awk
- Perl
- Python

While Kaldi is made in C++, knowledge of C++ is not required to use it.

Bash

Example run scripts in Kaldi are written in Bash. Not POSIX shell, but Bash, they contain some "Bashisms", which will not reliably work in other shells.

Bash is the main glue that holds Kaldi all together. It is used to prepare the data, prepare the language files, trigger the training, and print the results.

Knowing how to at least read bash is a must for using Kaldi. The others languages can be picked up as required, but bash is a must if you want to make use of the example scripts. You most likely *do* want to make use of the example scripts, they are some serious part of the documentation, and exist for most datasets – doing a lot of the work for you

Bash is however quiet easy for anyone who has work on the unix shell.

Key knowledge areas:

- Conditionals
- Loops
- Pipelines / input/output redirection
- variables
- the PATH

Awk

Awk is on of the standard tools for doing string manipulation on the command line, along with sed, grep, inline perl and simpler tools like tr, cut, head etc.

It is used a lot in the preparing of fst file inputs. A lot of inline Awk can be found in the aforementioned bash recipes.

It is a bit more complicated to understand than sed or grep, in that rather than a regex-tool it is a programming language. Many decent tutorials exist online.

Perl

Perl carries out a lot of the heavy lifting of kaldi setup. It is used for preparing data, where it gets to complex for Bash+Awk. It is used to facilitate the parallel (and/or distributed) task execution. It shows up throughout the examples.

Python

Python rarely shows up in the example scripts for kaldi, but it does show up. When it does, it is doing a task similar to those perl is used for. It is worth knowing, and using, as it is not a good idea to unleash more perl scripts into the wild. Combining in tools like Plumbum, it could also be used to replace Bash – this however is less portable. Other tools like pyp, can be used to replace Awk – again losing portability.

Kaldi-notes is maintained by oxinabox
This page was generated by GitHub Pages. Tactile theme by Jason Long.

Kaldi-notes

Some notes on Kaldi

Installing Kaldi

Building Kaldi

Follow the official instructions

Do not forget to first `cd` to `/kaldi-trunk/tools` then do a `make -j 8` to build to tools kaldi usesi.

Issues

```
- Issue: Requires libtool
   - Resolution: Build and install libtool, I did so by installing from [source](http://ftpmirror.gnu.org/libtool/libtool-2.4.5.tar.gz) locally (via config
- Resolution (alternative): Install with package manage (`apt-get`)
```

then try and build kaldi, first running .configure
Issue: Needs a BLAS.
Resolution: use OpenBlas
go back to `/kaldi-trunk/tools` and `make -j 8 openblas`
use it by running `./configure --openblas-root=../tools/OpenBLAS/install`

Issue: this Kaldi won't run with GCC 4.8.4
Resolution: install newer GCC from source
Follow instructions from GCC website
in particular for getting the dependencies

When it comes to running configure use: `../gcc-4.9.2/configure --disable-multilib`
when doing use `make -j 8` or it will take a very long time to build

Resolution (alternative) : install from backports

Installing Graph Viewer

For viewing the output of fstdraw, you need to convert it into a useful format. To do this you need `dot` which is part of graphviz. `apt-get install graphvis`

Adding things to your path

Since Kaldi has not been install to any location – just built in place. Nothing is on your path.

The build process, spreads out all the binaries into a number of folders in `\kaldi-trunk\src/*bin`, intermixing them with the source files. (so you can't just add the bin files to your path).

You might like to symlink all executables into one folder and add it to your path. The symlinking can be done with the following shell script:

```
for a in `find . -type f -executable -print`;
do
ln -s `pwd`/$a bins
done
```

This will put them all into the bins directory. Then you can edit your `.bashrc` file to add that to your path. e.g.:

```
PATH="/user/data7/20361362/kaldi/kaldi-trunk/src/bins:${PATH}"
```

Kaldi-notes

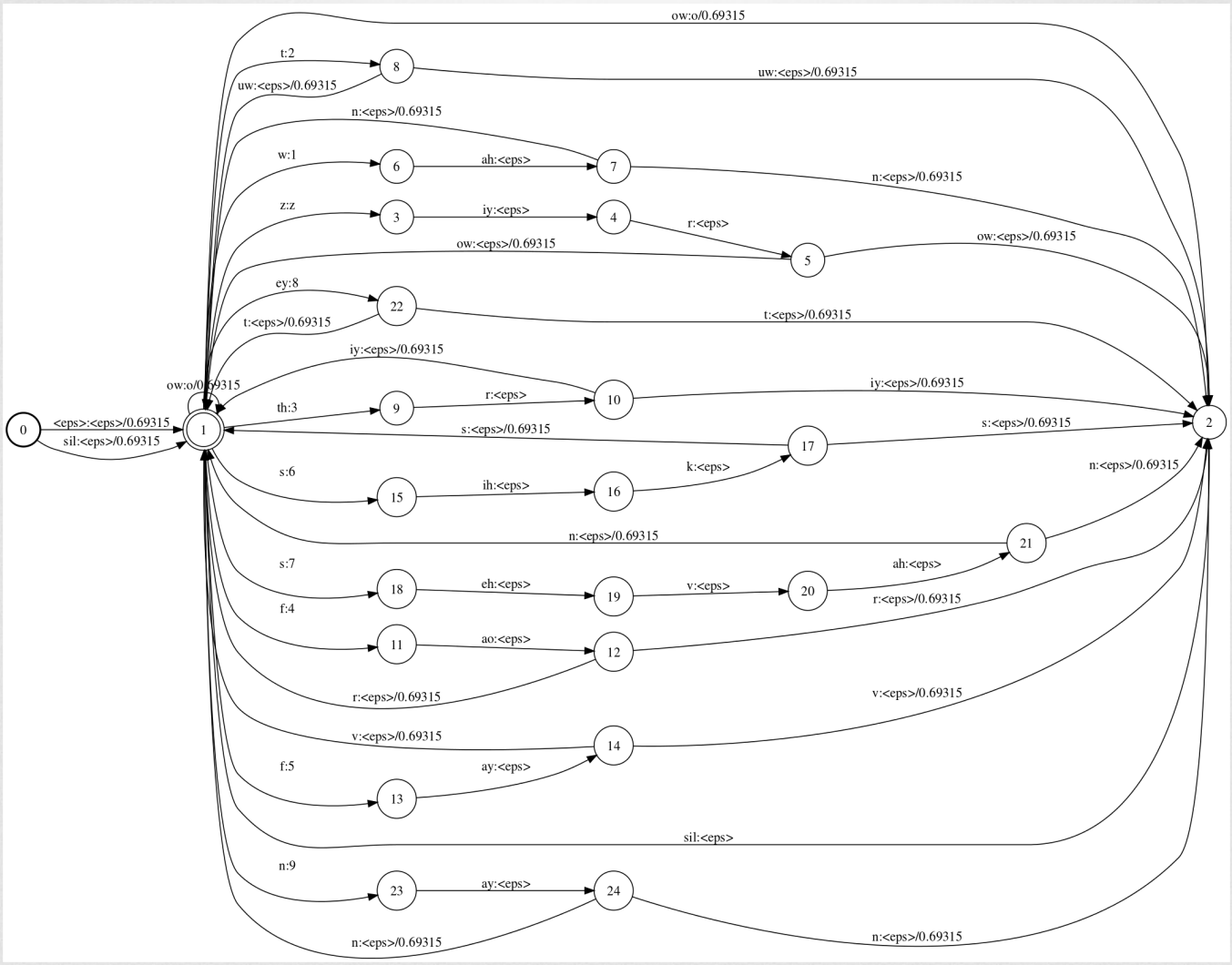
Some notes on Kaldi

Introduction to Finite State Transducers

Weighted Finite State Transducers is a generalisations of finite state machines. They can be used for many purposed, including implementing algorithms that are hard to write out otherwise – such as HMMs, as well as for the representation of knowledge – similar to a grammar.

Other places to get information

A descent set of slides can be found here
The OpenFst documentation and FST Examples are nonawful, though the shell and C++ sections are intermixed.
Speech Recognition with Weighted Finite-state Transducers a book chapter.



Above: An FST for pronouncing the digits 1-9 and two pronouncations of zero as: O (o) and zero (z), as used in TIDIGITS

Terminology

Symbols and Strings

Symbols come from some alphabet. They could be letters, words, phonemes, etc.

A string is a series of symbols from an alphabet, it can include the empty string. Matching the examples above, a string could be a word (spelt out), a sentence, a word (spelt out phonetically), etc.

A string can be represented as a Finite State Acceptor, where each symbol links to the state which links to the next.

Finite State Acceptor (FSA)

A Finite State Acceptor has the components of:

- a number of States
  - one or more of which is initial
  - one or more of which is terminal
- connections between states, with a input symbol (IE label)
  - the symbol could be the empty string (often written "-" or "" or "ε")
  - Not necessarily a one to one label to next state mapping (IE nondeterministic)

A FSA can be used to check if a string matches its pattern – it is computationally equivalent to a regular expression. It can also be used to generate strings which match that pattern.

FSA's can be treated as FSTs with same input and output symbols at each edge. Kaldi example scripts sometimes write them this way.

Finite State Transducers (FST)

A Finite State Transducer extends the Finite State Acceptor with the addition of:

- output labels on each edge
  - again the output can be the empty string.
    - it is common (such as in the TIDIGIT example above), to see only the first transition in a nonbranching substructure to be labels – the other states have nothing to add other than confirming we are in that chain. (which we might Not be)
    - The input alphabet and output alphabet do not have to be the same, and indeed are normally not.

A FST can be used to translate strings in its input alphabet to strings in its output alphabet, iff the input string matches the FSTs structure of allowed transitions. Thus if a FSA accepting its input alphabet is composed with it, it can translate the FSA. A series of FSAs can be composed, translating (matched) alphabet to alphabet, to get the desired output.

Weighted Finite State Acceptor/Transducer

As per the ordinal, but with a weight associated with each edge (as well as input, and output for transducers) This weight has a $\oplus$ and $\otimes$ operation defined on it, so that weight of alternatives and that cumulative weight along a path can be found.

- e.g. weight along a path is product of probabilities, and represents the probability of that input string.
- e.g. sum of weights on two edges is the probability of either of those alternitives.

Finite State Transducers in Kaldi

Kaldi uses FSTs (and FSAs), as a common knowledge representation for all things.

OpenFST

Filetypes

Textual FST/FSA definition: `.fst.txt`, `.fsa.txt`, `.txt`

Textual Representation of the finite state transducer or finite state acceptor respectively. These are the files you write to get things done, to describe your system.

In most of kaldi the `.fst.txt` / `.fsa.txt` is used. In other places it is just called `.txt`. In this document, it is always referred to by the former terms.

Line format:

Normal line `fromState toState inSymbol [outSymbol] [weight]`
Terminal state line `terminalState`

`fromState`, `toState`, and `terminalState` are integer state labels
`inSymbol`, `outSymbol` are textual strings being the name of the symbols from the respective input and output alphabets.
`outSymbol` should not be present in FSAs, and should always be present in FSTs

`weight` is a decimal number, indicating the weight of the edge. It must be present in Weighted FSTs/FSAs

Symbol table file: `.isyms`, `.osyms`, `.syms`, `.dict`, `.txt`

OpenFst like to refer to symbols by a positive integer. Since any finite alphabet is isomorphic to a subset of the positive integers, such a bijection exists, and can be created by enumerating each symbol.

For each FST you should have two of these files, one for the input alphabet and one for the output alphabet. For an FSA you should only have one – for the input alphabet. Under most circumstances these can be generated from the `.fst.txt` / `.fsa.txt` programatically. One such script for that is provided here in . Others exist throughout the kaldi example scripts, often using AWK oneliners.

In different places different extensions are used. The example script uses `.isyms` for symbol files generated from the input alphabet in the textual FSA/FSA description, and `.osyms` for that generated from the output alphabet.

Line Format:
`symbol integer`

`symbol` is a symbol from the alphabet being maps
`integer` is a unique positive integer (that is to say each integer only appears once in this file).

Binary FST/FSA: `.fst`, `.fsa`

This is the binary representation of the finite state transducer/acceptor. It is produced from the textual representation and symbol tables using `fstcompile`.

Graph of FST/FSA: `.dot`

It is a Graph Description Language File, produced by `fstdraw`. Piping in through `dot` can convert it into another more common format. E.g.: `cat example.dot | dot -Tsvg > example.svg` will convert `example.dot` to a SVG file. This is often done directly from the line that calls `fstdraw`.

OpenFST components

OpenFST is made up of several different command line applications. The three most used in kaldi are details briefly below:

Common convention

Input and Output

OpenFST commands which take a single input and produce a single output (such as `fstdraw` and `fstcompile`) have the basic usage of

```
fstcommand [FLAGS] [inputfile [outputfile]]
```

Which is to say an `inputfile` can optionally be provided, and if it is, then optionally an `outputfile` can be provided also.

If either is missing then input will be taken from standard in (IE piped in, or read from keyboard if no input pipe), and output will be sent to standard output (IE piped out, or printed to the terminal if there is no output pipe.), respectively.

Accessing Help (manpages)
Because OpenFST is not properly installed, it does not have entries in the man pages. To get help with a command use:

```
fstcommand --help | less
```

Compile: `fstcompile`

this converts a textural FST/FSA into a binary one.

FSA Usage: `fstcompile --acceptor --isymbols=<input.sym> [--keep_isymbols];`
FST Usage: `fstcompile --isymbols=<input.sym> -osymbols=<output.sym> [--keep_isymbols] [--keep_osymbols];`

Flags:

`--acceptor` : compiles it as an FSA, rather than a FST
`--isymbols=` , `--osymbols=` : specifies the input and output symbol tables
`--keep_isymbols` , `--keep_osymbols` : If set then the symbol stables as keeps in the binary file and do not need to be specified at later steps such as `fstdraw`

Draw: `fstdraw`

produces a `.dot` file graph, from a binary FST/FSA

FSA Usage: `fstdraw --acceptor --portait [--isymbols=<input.sym>] [--osymbols=<output.sym>]`
FST Usage: `fstdraw --portait [--isymbols=<input.sym>] [--osymbols=<output.sym>]`
Common Use example: `cat eg.fst | fstdraw --portait --isymbols=eg.isyms --osymbols | dot -Tsvg > eg.svg`

Flags:

`--portrait` this flag should **always** be set. If not set then image comes out rotated 90 degrees, and on a overly large canvas.
`--isymbols` , `--osymbols` , as before, but if not provided then symbols in the graphic will be replaced with their numeric representation, unless `--keep_isymbols` or `--keep_osymbols` was set in the compile step
`--acceptor` : draws a FSA, rather than a FST. Without it it will label the FSA with output labels.

Compose: `fstcompose`

Composed a FSA/FST with a FST

Usage: `fstcompose [--fst_compat_symbols=false] outer.[fst|fsa] inner.fst output.fst`

Applying an input to the Output FST is equivelent to first applying it to the Inner then applying the output of that to the Outer. i.e. `output(x)=outer(inner(x))`

`--fst_compat_symbols=false` : setting this to false (it defaults to true), may be required when composing FSTs/FSA where `--keep_isymbols` or `--keep_osymbols` was used and that the symbol files embedded while actually compatible are not the same files (it seems to store the filenames, which can be seen by running `strings` on a fst).

Other useful Commands

All the commands in OpenFst have a use. Other commands which I have found particularly useful, but do not have space to detail include;

`fstsymbols` manipulate and export the symbols tables in the binary FST/FSA
`fstproject` convert the FST into a FSA in either the input or output space by discarding the appropriate labels

Examples provided here

Several scripts are provided here to demonstrate how to make use of OpenFST, and to make using it easier. They can be downloaded from the Git backing this site. The section names below are also hyperlinks to download those scripts/files.

NOTE:

The example scripts assume openfst binaries are in your `PATH` . If you added all kaldi binerys during install step you will already have them. Otherwise you can add just the Openfst binaries by: Add to your `.bashrc` (or similar) `PATH="<...>/kaldi-trunk/tools/openfst/bin:${PATH}"` , where `<...>` is the math to the kaldi-trunk folder. then `source ~/.bashrc`

makeSymbols.py

makeSymbols.py is a script to make creating the symbol tables (which map symbols to arbitary unique integers) easier.

Usage: `python makeSymbols.py file fieldNumber`

`file` : the textual FST/FSA file (`.fst.txt` or `.fsa.txt usually`), to extract the symbols from
`fieldNumber` : which column of the file to take symbols from
input symbols use `fieldNumber` of 2
output symbols use `fieldNumber` of 3

The Symbols Table is output to standard out, and can be piped into a file

compileAndDraw.sh

the compileAndDraw.sh is a simple bash script that runs the whole process of compiling then drawing a FST/FSA.

Usage FSA: `bash compileAndDraw.sh filename.fsa.txt` Usage FST: `bash compileAndDraw.sh filename.fst.txt`

Note: unlike openfst programs this is file extension sensitive. It will make the appropriate call for a FSA or a FST based on the extension.

composeExample.sh

The composeExample.sh script runs though the creation then composition of the `dict.fst` and `sent.fsa`. It then outputs some sentences generated using the language model descried.

Usage: `bash composeExample.sh`

Example FSTs/FSAs

This folder contains 3 examples: The later two examples of sentence construction are based on ones provided in these lecture notes

simple.fsa.txt

simple.fsa.txt is a very simple Finite State Accepter.

dict.fst.txt

dict.fst.txt is a dictionary containing several words. There is only state in the dictionary – as far it its concerns words can be in any order

sent.fsa.txt

sent.fsa.txt is a grammar for a simple sentence, expressed as a finite state acceptor. Sentences can either be `determiner noun verb` or `determiner noun verb determiner noun` .

Kaldi-notes

Some notes on Kaldi

Introduction to training TIDIGITS

TIDIGITS is a comparatively simple connected digits recognition task. Like for many well-known corpora, Kaldi includes a example script for it. It is fairly typical for the example scripts – though simpler than most.

The example script can be found in `kaldi-trunk/egs/tidigits/s5/` all other scripts referred to here are relative to that path. Kaldi example scripts are all written to be run from that path (or it equivelent in other examples) even if they are located in a subfolder. Kaldi example scripts should only be run in `bash` – they will not necessarily work in other POSIX shells.

Be aware that a lot of the recipe code is shared between WSJ (Wall Street Journal), and all the other examples (including TIDIGITs). The `util/` and `steps/` folders in most of the example folders (including that for TIDIGITs), is a symlink to the matching folders in the WSJ example. You can very well make use of these scripts in your own recipes.

Other Resources:

The official Kaldi tutorial is not perfect (yet), but is a valuable resource. It is linked to in various sections throughout this document.

This tutorial seems good. Its web hosting does not seem stable, right now the google cached version can be used

The Major Steps

There are Four Steps to applying Kaldi to a task such as this.

Data Preparation:

- Locating the datafiles
- Parsing its annotations (e.g. Speaker Labels, Utterance Labels)
- Converting the audio data format

Language Preparation:

- Create Lexicon (Phoneme/Word dictionary)
- Create Grammar (Word Language Model)

Training Speech Recognizer:

- Training the GMMs
- Building the HMM graph

Evaluating the Speech Recognizer:

- Decoding and building the lattices
- Interpreting the results The full process can be carried out by running `bash run.sh`. Though you most likely need to edit at least the TIDIGITs path, and the `cmd.sh` (so that it is set to run locally, not on a cluster).

These instructions also briefly touch on some of the option that might be needed in more complicated tasks. They also go into some detail on things which are not done by the `run.sh` script, for example outputting utterance recognition lattice diagrams. The instructions do not provide any kind of solid introduction to HMMs or GMMs. Nor to bash or awk.

Kaldi-notes

Some notes on Kaldi

Data Preparation

The official kaldi documentation on this section. It is the basis of a lot of this section.

These steps are carried out by the script `local/tidigits_data_prep.sh`. It takes one parameter – the path to the dataset.

One should realize after looking at this section (and the next), just how valuable AWK and Bash (or equivalents) are for this task.

Locate the Dataset

on on the SIP network, the TIDIGITs data set can be found at `/user/data14/res/speech_data/TIDIGITs/`. Symlink it into a convenient location.

Split the Dataset into test and training

TIDIGITS is already split into test and training datasets. If it were not, you would need to do the split. It could be done at any time during the data preparation step, depending on when other useful informations (from the annotations), is available.

Parse its annotations

Annotations of the correct labels for each utterance need to be generated for the `test` and `training` directories.

Kaldi Script: `.scp` : Basically just a list of Utterances to Filenames

A Kaldi script file is just a mapping from record_id, to extended-filenames.

Line Format:

```
<recording_id> <extended_filename>
```

Recording ID

The recording ID is the first part of each line in a `.scp` file. If speaker id is available (which is is for TIDIGITs), it should form the first part of the recording id. Kaldi requires this not for speaker identification, but for purposes of sorting for training (`utt2spk` is for that).

The remained of the Speaker ID is arbitary, so long as it is unique. For convenience of generating the unique id, the example script for TIDIGITS uses `<speaker-id>_<transcription><sessionid>`.

As there is only one Utterance per recording in TIDIGITS, the Recording ID is the Utterance ID. (See below)

Extended Filename

The second part of the line is the extended filename Extended Filename is the term used by Kaldi, to refer to a string that is either the path to a wav-format file or it is a bash command that will output wav-format data to standard out, followed by a pipe symbol (`|`).

As the TIDIGITS data is in the SPHERE audio format, it needs to be converted to wav. So the sample scripts in Kaldi use `sph2pipe` to convert them, so the .scp files lines will look like: (assuming `sph2pipe` is on your PATH, otherwise Path to the executable will need to be used)

```
ad_16a sph2pipe -f wav ../TIDIGITs/test/girl/ad/16a.wav |
```

Segmentation File `segments`

If there were multiple utterances per recording then there would need to be a segmentation file as well, mapping Recording Ids and Start-End times to Utterance IDs. (See The official kaldi documentation on this section). As there is not, by not creating a `segments` file, Kaldi defaults to utterance id == recording id.

Text Transcription file `text`

The text transcription must be stored in a file, which the example calls `text`. Each line is an utterance-id followed by a transcription of what is said. E.g.:

```
ad_1oa 1 o
ad_1z1za 1 z 1 z
ad_1z6a 1 z 6
ad_23461a 2 3 4 6 1
```

Notice the Utterance-ID format as described above. Notice also, for later, that the transcription here is in word space, not phoneme space.

Utterance to Speaker Mappings `utt2spk`

This file maps each utterance id to a speaker id. Each line has the form `<utterance id> <speaker-id>`.

`spk2utt` is the opposite, and can be generated by using the script `utils/utt2spk_to_spk2utt.pl`. Each like starts with a speaker id, then has every utterance id they spoke.

Feature extraction

The feature extraction is carried out by the `run.sh` script, rather than by the `local/tidigits_data_prep.sh` script.

Extracting the MFCC Features

See this section of the kaldi tutorial

Mel-frequency cepstral coefficient (MFCCs) features. Done using the script `steps/make_mfcc.sh`

Compute Cepstral Mean and Variance Normalization statistics
Done using the script `steps/compute_cmvn_stats.sh`

Data Splitting.

The data needs to be divided up so that we can run many jobs in parallel. The data splitting is also carried out by the `steps/train_mono.sh` and `steps/decode.sh` scripts if it has not already been carried out, rather than by the `local/tidigits_data_prep.sh` script. It can however be carried out at anytime after the training and test directories are created, and features extracted.

It can be done with the script `utils/split_data.sh`. Usage:

```
utils/split_data.sh <data-dir> <num-splits>
```

`<data-dir>` is the directory where the data is. In this case it would be both of `data/test` and `data/train`
`<num-splits>` is the number of divisions of data needed. It should be the number of different Jobs.

Kaldi-notes

Some notes on Kaldi

Language Preparation

The official kaldi documentation on this section.

This section covers the same content as the recipe script in `/local/tidigits_prepare_lang.sh`

To understand this section you should first understand openFST.

The Phones

Kaldi expects a number of files to be in the `data/lang/phones/` directory. Most of them are not complex for TIDIGITS.

To facilitate the creation of this, it is useful to have a full list of phonemes. This could be created many ways. One way it to apply Awk to the lexicon (see next section).

These phone files a simple lists with one phone each line:

> `silence.txt`, `context_indep.txt` and `optional_silence.txt` all are made to just single like files containing `sil` the silence phoneme symbol.
> `nonsilence.txt` contains all other phonemes.

The following files would do a lot more in more complicated situations, but are simple for TIDIGITS - `sets.txt` each line contains a set of phones that should be considered to be the same phoneme (i.e. a set of morphemes for one phoneme). Since in TIDIGITS this is not a concern (we don't have access to a morpheme level transcription), each set contains just the one phoneme so all phonemes should be listed on there own line in this file. Including sil (the silence phoneme). - `disambig.txt` should be created and left empty. - `extra-questions.txt` should also be created and left empty.

Generating isymbols files from the phones.

Once you have a phonelist, it is very easy to enumerate it to create the isymbols file required for the phoneme-word FST.

Converting Symbol Phonelist to Integer Phone Lists

Once you have a isymbols file, each of the files created in the previous set need to be converted to lists of there matching integers rather than textual symbols. The files created this way have the same name, but with a `.int` extension.

The perl script `utils/sym2int.pl` is used for this. It can take a single parameters of the symbols file, and then will take on standard in, the symbolic (`.txt`) phone lists, and output (on standard out), there corresponding integer forms.

`silence`, `nonsilence`, `context_indep`, `optional_silence`, `disambig` all also need to be converter to colon separated list files (`.csl`), these are the same as the `.int` files, but instead of the integer phone representation being separated by linebreaks, they are separated with colons. A simplish job for Awk/sed.

roots.txt Decision Tree Roots

Kaldi makes use of Decision Trees for some functionality. See the documentation for the why and how of this.

It require a root definition file. For TIDIGITS is is very simple. `roots.txt` contains on each line, `shared split <phone-symmbol>`, and has one line for each phone. It is converted to `roots.int`, by converting each phone symbol to it integer representation.

Words and Out of Vocabulary Lists

A word symbol list will also need to be constructed for the FST. Again this can be generated from the lexicon (see below), with Awk.

The word symbols are simply: o, z ,1, 2, 3, 4, 6, 7, 8, 9.

To go with this Kaldi needs to be told what word to use when words that are not in the vocabulary list are found. Since no such words exist in TIDIGITS, it really doesn't mater what it done with them. But Kaldi requires the files.

Create a `oov.txt` with any one word in it. (Example script uses z). Create a oov.int with the matching integer for it. This can be done manually, or it could be done with `sym2int` on the words symbol list, created earlier. If you arranged you word symbols so that the int form of your oov word is the same is its text form then you are either over or under thinking this, and you could copy the `oov.txt` to `oov.int`

The Lexicon

The example recipe for TIDIGITs is quiet clever about constructing the phoneme to word FST. There script `util/make_lexicon_fst.pl` takes a lexicon file, and outputs a text FST file. Each line of the Lexicon file has the format:

```
<word> <phoneme> <phoneme> <phonen....
```

i.e. one word followed by its phoneme make up, specified with space delimited symbols.

The Lexicon for TIDIGITS:

```
z iy r ow
o ow
1 w ah n
2 t uw
3 th r iy
4 f ao r
5 f ay v
6 s ih k s
7 s eh v ah n
8 ey t
9 n ay n
```

Converting Lexicon to lexicon.fst.txt: `util/make_lexicon_fst.pl`

The Full break-down of how to use it (which will be output if no arguments are passed to it):

Usage: `make_lexicon_fst.pl [--pron-probs] lexicon.txt [silprob silphone [sil_disambig_sym]] >lexicon.fst.txt`

Creates a lexicon FST that transduces phones to words, and may allow optional silence. Note: ordinarily, each line of `lexicon.txt` is: `w fstcompile ord phone1 phone2 ... phoneN` ; if the `--pron-probs` option is used, each line is: `word pronunciation-probability phone1 phone2 ... phoneN` . The probability 'prob' will typically be between zero and one, and note that it's generally helpful to normalize so the largest one for each word is 1.0, but this is your responsibility. The silence disambiguation symbol, e.g. something like #5, is used only when creating a lexicon with disambiguation symbols, e.g. L_disambig.fst, and was introduced to fix a particular case of non-determinism of decoding graphs.

Compiling the Lexicon FST: `L.fst`

The `lexicon.fst.txt` , is then compiled (`fstcompile`), using the isymbols and osymbols generated from the `lexicon.txt` , plus the silence phoneme (sil) added in the make `lexicon.txt.fst` step. It's edges are then sorted by output label using `fstarcsort` . (not sure why this is required.)

L_disambig.fst = L.fst

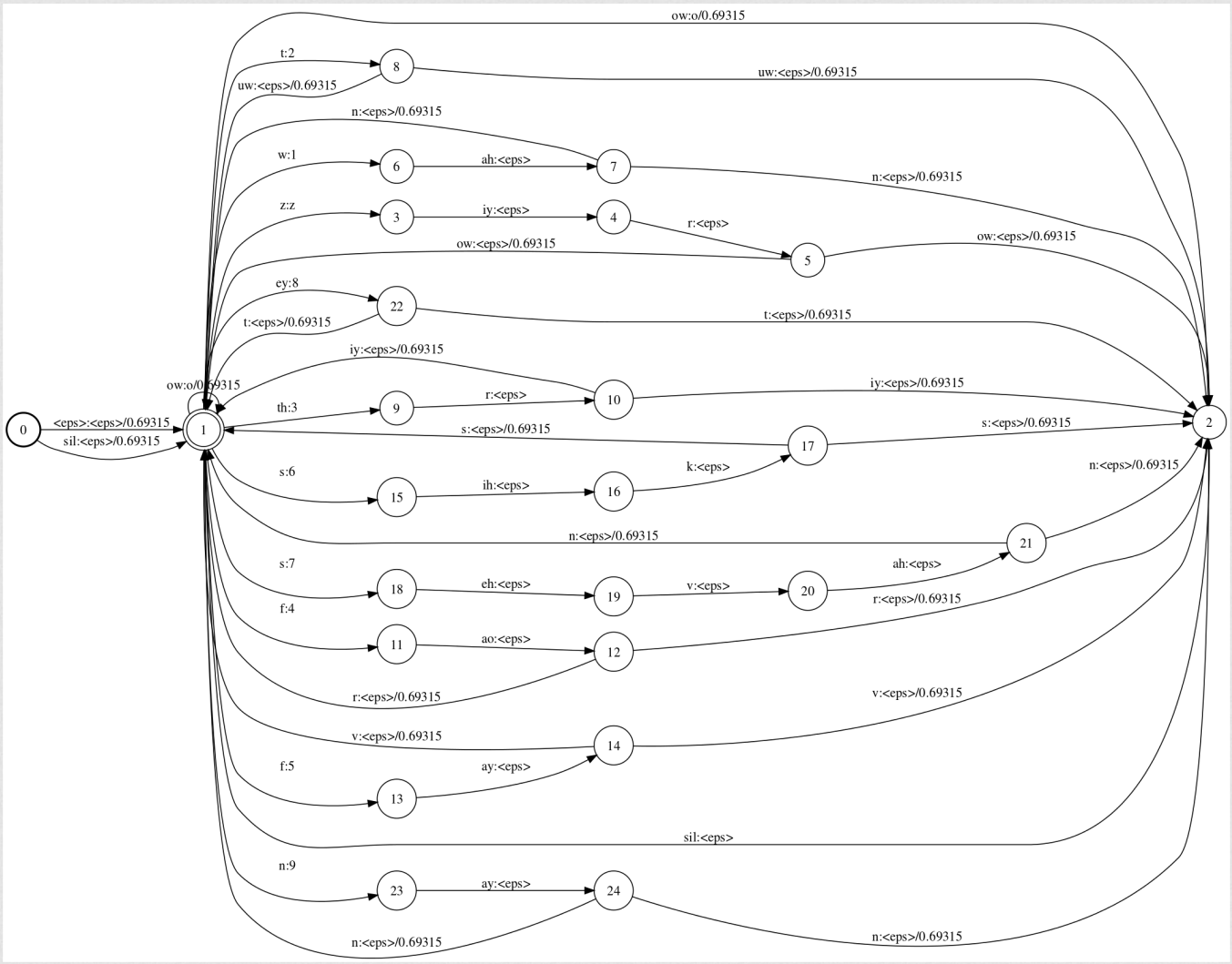
To quote the TIDIGITs recipe:

> *in this setup there are no "disambiguation symbols" because the lexicon contains no homophones; and there is no '#0' symbol in the LM [(Language Model)] because it's not a backoff LM, so L_disambig.fst is the same as L.fst*

So `L.fst` is copied to `L_disambig.fst`

For more information on disambiguation read the documentation page.

The final lexicon FST



0 is the initial state (as always), and 1 is the only final state. Notice there is only one path leaving state 2 and that goes back to 1 via 'sil'. Notice also that all states which have a transition to 2, have a identical transition to 1.

The Grammar

The Lexicon defined how Phonemes make up words. The Grammar defines how words make up a sentence. The grammar is a weighed FSA. It is expressed as a weighted FST in the example script – a FSA can be considered as a FST with input and output symbols the same.

The States

As our sentences are made up of digit sequences of length between 1 and 7, this could be re-represented as a WFSA with 8 states, 7 of which are optionally terminal, and all of which have all digits going to

It is simpler, and more real world useful, however to take the assumption that digit sequences can be of any length (greater than 1).

This can be modelled as a FSA, with just one state, which is both initial and terminal, and all edges connect to it. This is done in the example recipe

The Transitions

As a FSA each edge has one label, but as it is expressed as a FST this label is put on both input and output.

The Weights

We want to relate the weight to the probability of that transition happening. After a digit has been said there are 12

possible future actions:

A digit from 1-9 is said
o, pronounced Oh is said
z, pronounced zero is said
nothing further is said, as the sentence has ended.

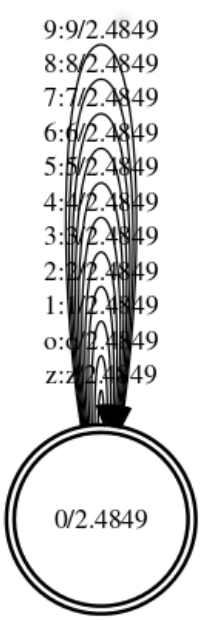
It is reasonable to assume each of there 12 options is equally likely. So they each have a probability of 1/12th.

In there circumstances it is normal to work with negative log probabilities for numerical stability. `-ln(1/2)=2.48490664979...`. This can just be put in to the final like of each column in the FST. The example recipe uses an inline perl to calculate it on the fly (but is is not expressed in any more digits).

Compile and Arc Sort

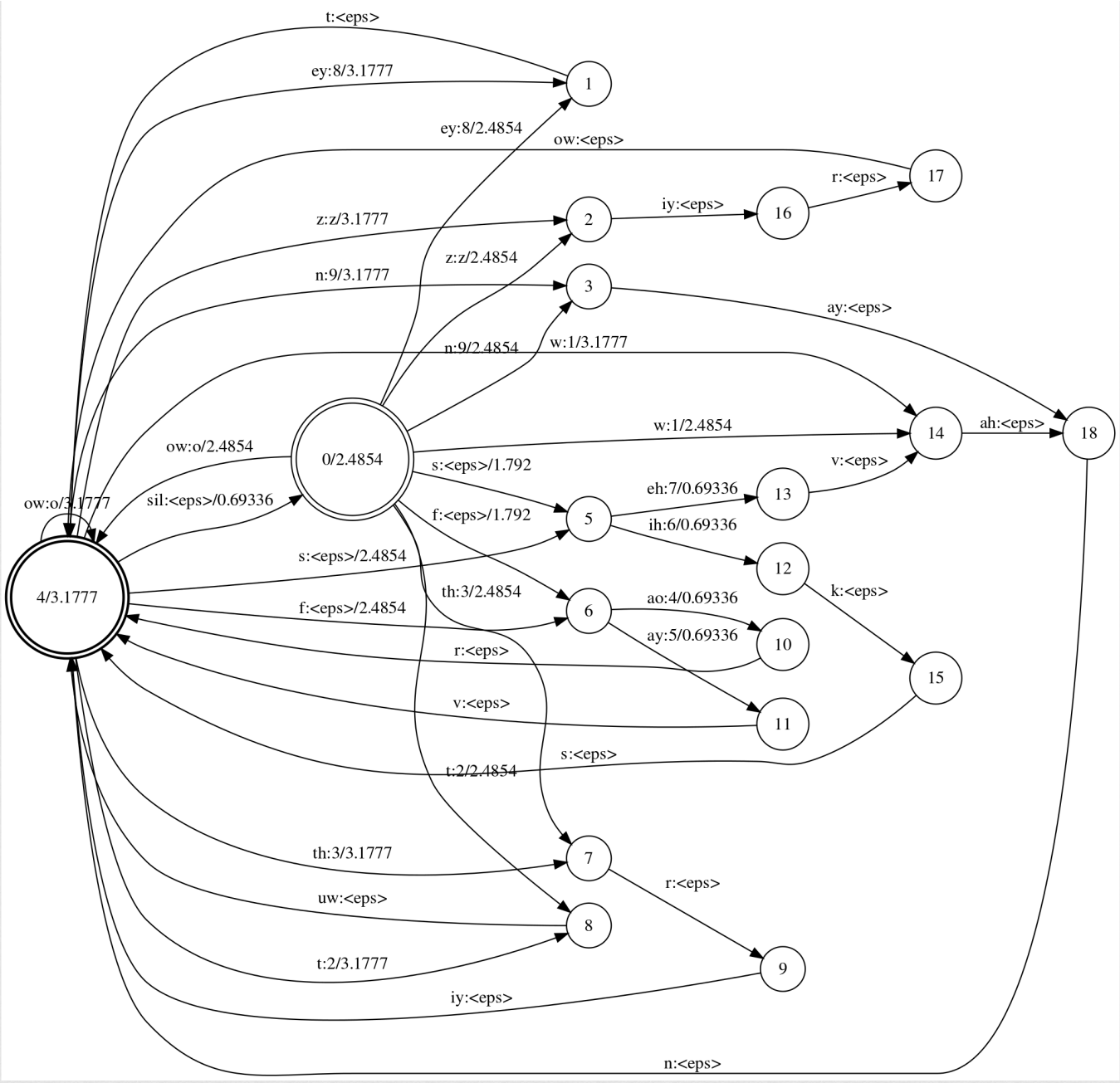
The FST is compiled and arc sored just as for the lexicon. The example calls this `G.fst`

The final grammar FST



The Final Grammar Composed with Lexicon

The great beauty of working with FSTs in this way is they are compose-able. There is no need to compose them in this step – that will be done later when they are also composed with the HMM; but so that you can see what is going on, below is the grammar composed with the lexicon.



0 is the initial state. 0 and 4 are the final states. This FST maps phones (from the lexicon) to strings of words which are allowed by the Grammar. However, since the Grammar is so permissive (no restrictions at all on order of words), this looks very similar to the Lexicon FST. It is in fact equivelent to the Kleene closure of the Lexicon FST.

HMM Topology

One could say this was really part of the next step of training. However it is covered in the sample script for this section at `/local/tidigits_prepare_lang.sh` .

The actual action to be taken is very simple. Understanding why takes some knowledge of HMMs.

The HMM Topology defines how the HMM that is going to be created for the Phones works. In most cases the 3 state Bakis model is used.

To get a idea what is really going on under the hood, read this page of the documentation.

In short, topo files define instructions for how to build Hidden Markov Models (HMMs) – what states are linked to others.

The topo file is expressed in a almost-XML language (not quiet XML as not all opened tags have close tags, only ones that have other elements nested inside them.). Kaldi uses this, and will eventually at some point internally produce a WFST that is the HMM. Which you might find in literature referred to as H, to go with the lexicon L and the grammar G.

In practice

All that is required is to copy the template 3 state Bakis from `conf/topo.protp`, and use `sed` to replace NONSILENCEPHONES, and SILENCEPHONES, with space separated lists of the integer representation of the nonsilent and silent phones respectively.

Validating Everything has been done correctly so far

This step is actually carried out in `run.sh` rather than in the `local/tidigits_prepare_lang.sh`.

`util/validate_lang.pl` takes a single argument – the path to the lang folder. It then validates that all things have been set up correctly. However there are some warnings for the TIDIGITs setup.

To quote `run.sh`:

> *`utils/validate_lang.pl data/lang/` Note; this actually does report errors, and exits with status 1, but we've checked them and seen that they don't matter (this setup doesn't have any disambiguation symbols, and the script doesn't like that).*

Kaldi-notes

Some notes on Kaldi

Controlled remote vs local execution: `cmd.sh`

Kaldi is designed to work with SunGrid clusters. It also work with other clusters. We want to run it locally, it can do that too. This can be done by making sure cmd.sh sets the variables as follows:

```
export train_cmd=run.pl
export decode_cmd=run.pl
```

rather than making references to `queue.pl`.

Training (and testing), will still be split into multiple jobs, each handling different subsets of the data.

Training Recognizer

This section is covered by this section of the kaldi tutorial.

The majority of the steps covered in this page, are triggered by the script `run.sh`

Training

Done using the script `steps/train_mono.sh`, However very similar steps are used in the other training scripts from in `steps` (such as `steps/train_deltas`).

Usage:

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

`training-data-dir` is the path to the training data directory prepared earlier
`lang-dir` is the path the directory containing all the language model files, also prepared earlier
`exp-dir` is a path for the training to store all of its outputs. It will be created if it does not exist.

Configuration / Options

The `train_mono` script takes many configuration options. They can be set by passing them as flags to script: as so: `--<option-name> <value>`. Or by putting them all into a config bash script, and adding the flag `--config <path>`. They could also be set by editing the defaults in `steps/train_mono.sh`, but there is no good reason to do this.

`nj` : Number of Jobs to run in parallel. (default `4`)
`cmd` : Job dispatcher script (default `run.pl`)
`scale_opts` : takes a string (wrap it in quotes) to control scaling options (default `"--transition-scale=1.0 --acoustic-scale=0.1 --self-loop-scale=0.1"`)
  `transition-scale` (default `1.0`)
  `acoustic-scale` (default `0.1`)
  `self-loop-scale` (default `0.1`)

`num_iters` Number of iterations of training (default `40`)
`max_iter_inc` maximum amount to increase the number of Gaussians by (default `30`)
`totgauss` Target number of Gaussians (default `1000`)
`careful` passed on to `gmm-align-compiled`. To quote its documentation: "If true, do 'careful' alignment, which is better at detecting alignment failure (involves loop to start of decoding graph)." (default `false`)
`boost_silence` Factor by which to boost silence likelihoods in alignment. (Default `1.0`)

`realign_iters` iterations in which to perform realignment (default `"1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 23 26 29 32 35 38"`)
`power` exponent to determine number of Gaussians from occurrence counts (default `0.25`)
`cmvn_opts` options will be passed on to cmvn – like scale_opts. (default `""`)
`stage`: This is used to allow you to skip some steps, if the program crashed partway though. The stage variable sets the stage to start at. The stages are discussed in the next section (default `-4`)

What is the parallelism of Jobs in the Training step

During training, the training set can be (and is in the example) split up (the actual spitting is explained in the data preparation step), and each different process (Job), trains on a different subset of utterances, which each iteration are then merged.

Initialisation Stages

Initialise GMM (Stage -3)

Uses `/kaldi-trunk/src/gmmbin/gmm-init-mono`. Call that with the `--help` option for more info

This defines (amongst other things), how many GMMs there are initially.

Compile Training Graphs (Stage -2)

uses `/kaldi-trunk/source/bin/compile-training-graphs`. Call that with the `--help` option for more info.

See this section of the documentation.

Align Data Equally (Stage -1)

Creates an equally spaced alignment. As a starting point for further alignment stages. Uses `/kaldi-trunk/source/bin/align-equal-compiled`. Call that with the `--help` option for more info.

Estimate Gaussians (Stage 0)

Do the maximum likelihood estimation of GMM-based acoustic model. Uses `/kaldi-trunk/src/gmmbin/gmm-est`. Call that with the `--help` option for more info.

The script notes:

> *In the following steps, the `--min-gaussian-occupancy=3` option is important, otherwise we fail to est[imate] "rare" phones and later on, they never align properly.*

Training (Stage = Iterations completed)

Every Iteration a number of steps are carried out.

Realign

If this iteration is one of the `realign_iters` then:

Boost Silence
Silence is boosted using `/kaldi-trunk/src/gmmbin/gmm-boost-silence`, Call that with the `--help` option for more info. Notably it does not necessarily boost the silence phone (but it does in this training case), it can boost any phone. It does this by modifying the GMM weights, to make silence more probable.

Align
Features are aligned given the GMM models. Uses `kaldi-kaldi/src/gmmbin/gmm-align-compiled`. Call that with the `--help` option for more info.

Reestimate the GMM model.

First accumulate stats which are used in the next step. This is done using `/kaldi-trunk/src/gmmbin/gmm-acc-states-ali`. Call that with the `--help` option for more info.

Then redo the GMM-based acoustic model. This is done with `/kaldi-trunk/src/gmmbin/gmm-est`, but using very different arguments. Again call that with the `--help` option for more info.

Merge GMMs

All the different GMMs from the partitioned training dataset are then merged, using `gmm-acc-sum` to produce a model (a `.mdl` file). The model can be examined using `/kaldi-trunk/src/gmmbin/gmm-info` to get some very basic information about the number of Gaussians etc.

Finally, increase the number of Gaussians (capped by `max_iter_inc`), so that by the time all the iterations (`num_iters`) are all complete, it will approach the target total number of Gaussians (`totgauss`) – assuming `max_iter_inc` did not block it.

Making of the Decoding Graph

As showing earlier, the Grammar (G) can be composed with the Lexicon (L), to get a phoneme to word mapping.

To increase the power of the phones, they could be expanded to add context. For example making the 'ay' phone in 'n-ay-n' (nine) different from the one in 'm-ay-n' (mine). This can be done with a Context dependency FST, which can be scaled with the number of phones to take into account in the context. This is roughly equivelent to making use of n-grams on the phonetic level. Using 3 (i.e. one to each side) context is referred to a triphones.

The Context Dependency can be expressed as a FST, referred to as C.

This blog post presents the details of the creation quiet well. It will be a bit of revision from the data preparation step.

Usage of `util/mkgraph.sh`

The final graph is created using `util/mkgraph.sh` To quote the introduction to that script:

> *…creates a fully expanded decoding graph (HCLG) that represents all the language-model, pronunciation dictionary (lexicon), context-dependency, and HMM structure in our model. The output is a Finite State Transducer that has word-ids on the output, and pdf-ids on the input (these are indexes that resolve to Gaussian Mixture Models).*

It also creates the aforementioned Context Dependency Graph.

Usage:

```
utils/mkgraph.sh [options] <lang-dir> <model-dir> <graphdir>
```

`lang-dir` is, as before, the path to the directory containing all the language model files, also prepared earlier
`model-dir` is the `exp-dir` from the previous train-mono step, which now contained the trained model.
`graph-dir` is the directory to place the final graph in. In the example script this is made as a graph subdirectory under the `exp-dir`. If it does not exist, it will be created

Context Options

There are 3 options for defining how many phones are used to create the context. These are passed as the options to the `utils/mkgraph.sh` script

`--mono` for monophone i.e. one phone i.e. no context (Used in `steps/train_mono.sh`)
no flag (default) for triphone i.e. 3 phones i.e. one phone to each side for context
`--quinphone` for quinphone i.e. 5 phones i.e. 2 phones to each side for context

It would not be hard to extend the mkgraph script to create contexts of any length. The section of the mkgraph script responsible for this, makes use of `/kaldi-trunk/src/fstbin/fstcomposecontext`, take a look at its `--help` for more information.

Kaldi-notes

Some notes on Kaldi

Evaluation – using the model to recognise speech.

Decoding

We already created a decoding graph in the training step. Using this graph to decode the utterances is done using `steps/decode.sh`. This script only works only for certain feature types – conveniently all the feature types we use in TIDIGITS. (Similar decoding functions also exist in steps, for other feature types)

Usage for `steps/decode.sh`

Usage:

```
steps/decode.sh [options] <graph-dir> <test-data-dir> <decode-dir>
```

`test-data-dir` is the path to the training data directory prepared earlier
`graph-dir` is the path to the directory containing the graphs generated in the previous step
`decode-dir` is a path to store all of its outputs – including the results of the evaluations. It will be created if it does not exist.

Configuration / Options

The `decode.sh` script takes many configuration options, these should be familiar from the `train_mono.sh` script options above. They can be set by passing them as flags to script: as so: `--<option-name> <value>`. Or by putting them all into a config bash script, and adding the flag `--config <path>`. They could also be set by editing the defaults in `steps/decode.sh`, but there is no good reason to do this.

`nj` : Number of Jobs to run in parallel. (default `4`)
`cmd` : Job dispatcher script (default `run.pl`)

`iter` : Iteration of model to test. Training step above actually stores a copy of the model for each iteration. This option can be used to go back and test that (default final trained model). Overridden by `model` option.

`model` : which model to use, given by path. If given this overrides the `iter` (default determined by value of `iter`)

`transform-dir` directory path to find fMLLR transforms (Not useful for TIDIGITS). (default: N/A only used if fMLLR transformed were done on features.)

`scoring-opts` options to local/score.sh. Can be used to set min and max Language Model Weight for rescoring to be done. (default: "")

`num-threads` number of threads to use, (default 1).

`parallel-opts <opts>` option string to be supplied to the parallel executer (in our running locally case `utils/run.pl`

e.g. '-pe smp 4' if you supply `--num-threads 4`

`stage` : This is used to allow you to skip some steps, as above. However decode only has 2 stages. If stage is greater than 0 will skip decoding and just do scoring. (default `0`)

Options passed on to `kaldi-trunk/src/gmmbin/gmm-latgen-faster` :

`acwt` acoustic scale applied to acoustic likelihoods applied in lattice generation (default 0.083333). It affects the pruning of the lattice (low enough likelihood will be pruned).

`max_active` (default 7000)
`beam` decoding beam (default 13.0)
`lattice_beam` lattice generation beam (default 6.0)

]##What is the parallelism of Jobs in the Decoding step During decoding, the test set can be (and is in the example) split up (the actual spitting was doing in the data preparation step), and each different process (Job), decodes different subset of utterances, into lattices (see below). When scoring happens (see below), all the different lattices are evaluated to get the transcriptions.

Lattices

From the Kaldi documentation "A lattice is a representation of the alternative word-sequences that are "sufficiently likely" for a particular utterance."

This blog post gives an introduction to the Latices in Kaldi quiet well, relating them to the other FSTs.

Kaldi creates and uses these latices during the decoding step. However, interpreting them can be hard, because all the command line programs for working with them use Kaldi's special table IO, describing how this works in detail is beyond the scope of this introduction. The command line programs in question can be found in `/kaldi-trunk/src/latbibin`

The Latices are output during the decoding into `<decode-dir>`. Into a numbered gzipped file. E.g. `lat.10.gz`. The number corresponds to the Job number (because the data has been distributed to multiple processes). Each contains a single binary file. Each of these achieves contains many latices - one for each utterance.

Commands to work with them take the general form of:

```
<lattice-command> [options] "ark:gunzip -c <path to lat.N.gz>|" ark,t:<outputpath>
```

Each of the lattice commands do take the `--help` option which will cause them to give the other options.

Example Converting a lattice to a FST Diagram

For example, Consider the lattice gzipped at `exp/mono0a/decode/lat.1.gz`

Running:

```
lattice-to-fst "ark:gunzip -c exp/mono0a/decode/lat.1.gz|" ark,t:1.fsts
utils/int2sym.pl -f 3 data/lang/words.txt 1.fsts > 1.txt.fsts
```

(Assuming that `/kaldi-trunk/src/latbin` is in your path)

Will fill `1.fsts` with a collection of text form FSTs, one for each utterance space separated. Ones with multiple terminal states have multiple different "reasonably likely" phrases possible. The output labels on the transitions are words (Which we restored using int2sym). The weights are the negative log likelihood of that transition (or that final state)

As shown below:

```
ad_16a
0 1 1 3 14.7788
1 2 6 8 5.0416
2 2.61209

ad_174o2o8a
0 1 1 3 12.5118
0 11 o 2 9.44585
1 2 7 9 9.34774
1 16 o 2 6.57278
```

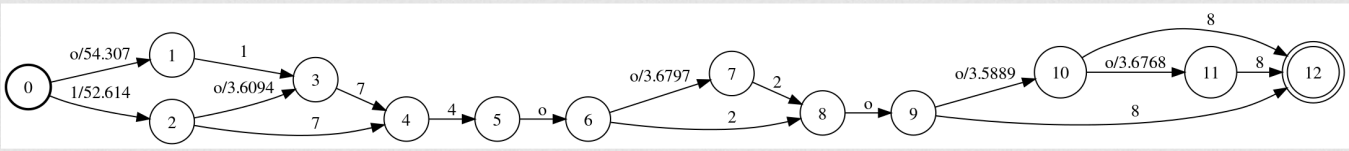
```
2 3 4 6 2.08985
3 4 o 2 10.2191
4 5 2 4 4.91992
4 9 o 2 3.20784
5 6 o 2 3.84306
6 7 o 2 3.90951
6 13 8 10 7.07031
7 8 8 10 6.74935
7 14 o 2 3.79537
8 2.61209
9 10 2 4 5.3914
10 6 o 2 3.84306
11 12 1 3 4.75861
12 2 7 9 9.34774
13 2.61209
14 15 8 10 6.63099
15 2.61209
16 17 7 9 6.38392
17 3 4 6 2.08985
```

Then we grab one particular FST off of it. (in this case just using Awk to grab some lines – most sophisticated approaches exist). Compile it. Project it only along the input labels (cos they are the words it will guess at), Minimise the number of states to get a simpler but equivelent model (easier to read) and finally draw it as an FSA.

```
cat 1.txt.fsts | awk "6<NR && NR<30" |\
   fstcompile --isymbols=data/lang/words.txt --keep_isymbols| \
   fstproject | fstminimize| \
   fstdraw --portrait --acceptor | dot -Tsvg > 1.2.svg
```

The Result of this, being a FSA that will accept (/generate) the likely matches for the utterance `ad_174o2o8a`. The utterance actually said "174o2o8", which is accepted by the path through states "0,2,4,5,6,8,9,12"

Note: that when the confidence in the path being correct is very high no weight is shown.



Notice that the lattice has a lot of paths allowing 'o' to be followed by another 'o'.

Drawing Phone Lattices

Much like we can draw lattices at the word level, we can go down to draw them at the phone level.

```
lattice-to-phone-lattice exp/mono0a/final.mdl "ark:gunzip -c exp/mono0a/decode/lat.1.gz|" ark,t:1.ph.lats
lattice-copy --write-compact=false ark:1.ph.lats ark,t:1.ph.fsts
utils/int2sym.pl -f 4 data/lang/phones.txt 1.ph.fsts > 1.ph.txt.wfsts
cat 1.ph.txt.wfsts | awk 'BEGIN{FS = " "}{ if (NF>=4) {print $1," ", $2," ",$3," ",$4;} else {print $1;};}' > 1.ph.txt.fsts
```

Notice that in the first step the model (`final.mdl`) was also used. The output of the first step is in the Compact Lattice form which is not amenable to being worked with by scripts like int2sym. The second set expands it, making it a FST. Third step is simply substituting the phone symbols into the output. It is worth looking perhaps at `1.ph.txt.fsts`, notices that the weights are only at start word phones. It is also however hard to read as it have hundred of empty string states ('). Notice also there are 2 weights (this is the Graph Weight and the Acoustic Weight). As there are 2 weights, this is not in a valid format for OpenFST. Thus the four line (the Awk Script) removed them all.

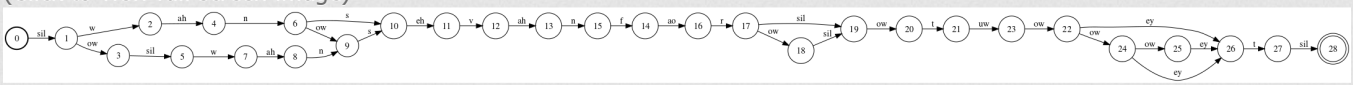
With that done we now have something that looks like a collection txt.fst, however it is still very filled with epsilon states.

Now to draw it up. Capturing the utterance `ad_174o2o8a` again, we will draw it:

```
cat 1.ph.txt.fsts| awk "199<NR && NR<1246" | \
    fstcompile --osymbols=data/lang/phones.txt --keep_osymbols | \
    fstproject --project_output | \
    fstrmepsilon | fstdeterminize | fstminimize  | \
    fstdraw --portrait --acceptor | dot -Tsvg > 1.ph.svg
```

So the steps being again, grabbing the lines we want, compiling it. Projecting it (this time on the output space), removing epsilons (matchers for empty strings), determining, and minimising to make it more readable. Then drawing it.

(Click to view full screen image)



Scoring

Viewing Results

As the final step of `steps/decoding.sh` the results are recorded.

The can be found in `<decode-dir>` under filenames called `wer_<N>` where `N` is the Language Model Scale.

Example:

```
compute-wer --text --mode=present ark:data/test/text ark,p:-
%WER 1.63 [ 670 / 41220, 420 ins, 111 del, 139 sub ]
%SER 4.70 [ 590 / 12547 ]
Scored 12547 sentences, 0 not present in hyp.
```

The Wikipedia entry on Word Error Rate (WER), is a reasonable introduction, if you are not familiar with it.

The Sentence Error Rate (SER), is actually the utterance error rate. Of all the utterances in the test set, it is the portion that had zero errors. Both error rates only consider the most likely hypothesis in the lattice.

`utils/best_wer.pl` will take as input any number of the `wer_<N>` files, and will output the best WER from amongst them.

How scoring is done

Scoring is done by `local/score.sh`. his program takes a `--min-lmwt`, `--max-lmwt` for the minimal and maximum language model weight. It outputs the `wer_N` files for each of this different weights. The Language Model Weight, is the trade off (vs the Arctic model weight) as to which is more important, matching the language model, or matching the sounds.

The scoring program works by opening all the lattice files, and getting them to output a transcription of the best guess at the words in all of the utterances they contain. The best guess is done with `<kalid-trunk>/src/latticebin/lattice-best-path` the language model weight is passed to it, as `-lm-scale`.

The WER is calculated using `<kalid-trunk>/src/bin/computer-wer`, which takes two transcription files – the best guess output in the previous step, and the correct labels. The program outputs the portion that match.