Kaldi Tutorial

Eleanor Chodroff

Written: 2015-07-15 | Last updated: 2018-11-05

Contents

Preface

This website provides a tutorial on how to build acoustic models for automatic speech recognition, forced phonetic alignment, and related applications using the Kaldi Speech Recognition Toolkit.

Acknowledgements

I would like to thank Jack Godfrey, Sanjeev Khudanpur, Paul Smolensky, Yenda Trmal, and Colin Wilson who were integral in creating this tutorial. I am grateful to Jack Godfrey for creating the opportunity for me to learn Kaldi, and to Yenda Trmal and Sanjeev Khudanpur for taking almost an entire day to teach me how to use Kaldi. Yenda Trmal and Paul Smolensky graciously provided comments and revisions on previous drafts of this tutorial. I am also very grateful to Colin Wilson for introducing me to coding and training me as an "apprentice". All remaining errors are my own.

Chapter 1

Introduction

What is Kaldi? Kaldi is a state-of-the-art automatic speech recognition (ASR) toolkit, containing almost any algorithm currently used in ASR systems. It also contains recipes for training your own acoustic models on commonly used speech corpora such as the Wall Street Journal Corpus, TIMIT, and more. These recipes can also serve as a template for training acoustic models on your own speech data.

What are acoustic models? Acoustic models are the statistical representations of a phoneme's acoustic information. A phoneme here represents a member of the set of speech sounds in a language. N.B., this use of the term 'phoneme' only loosely corresponds to the linguistic use of the term 'phoneme'.

The acoustic models are created by training the models on acoustic features from labeled data, such as the Wall Street Journal Corpus, TIMIT, or any other transcribed speech corpus. There are many ways these can be trained, and the tutorial will try to cover some of the more standard methods. Acoustic models are necessary not only for automatic speech recognition, but also for forced alignment.

Kaldi provides tremendous flexibility and power in training your own acoustic models and forced alignment system. The following tutorial covers a general recipe for training on your own data. This part of the tutorial assumes more familiarity with the terminal; you will also be much better off if you can program basic text manipulations.

Please also refer to the Kaldi website for thorough documentation.

Chapter 2

Installation

Please refer to http://www.kaldi-asr.org/doc/install.html for more details.

1. Prerequisites

- Git

Git is a version control system that let's developers update source code and easily redistribute updates to the users. Git can be installed via homebrew or following instructions here.

- Subversion (svn)

Subversion is also a version control system that keeps track of individual changes while developing the source code. Some of the example scripts still depend on this package.

2. Downloading

It is recommended that Kaldi be installed on a machine with good computing power. Following the instructions for downloading Kaldi on this page: http://kaldi-asr.org/doc/install.html, first direct the terminal to where you would like to install Kaldi, and then type the following:

```
git clone https://github.com/kaldi-asr/kaldi.git kaldi --origin upstream
```

3. Installation

Locate the file `INSTALL` in the downloaded package and follow the instructions there. In short, you'll need to follow the install instructions in `kaldi/tools` and then in `kaldi/src`. The most typical installation should involve the following code, but read the `INSTALL` file just in case:

```
cd kaldi/tools
extras/check_dependencies.sh
make

cd kaldi/src
./configure
make depend
make
```

Chapter 3

Familiarization

This section serves as a cursory overview of Kaldi's directory structure. The top-level directories are `egs`, `src`, `tools`, `misc`, and `windows`. The directories we will be using are `egs` and `src`.

`egs` stands for 'examples' and contains example training recipes for most major speech corpora. Training recipes are available for the Wall Street Journal Corpus (`wsj`), TIMIT (`timit`), Resource Management (`rm`), and many others. Under each of these directories are usually a few different versions (`s3`, `s4`, `s5`, etc.) The highest number, usually `s5`, is the most current version and should be used for any new development or training. The older versions are kept for archival purposes only.

`src` stands for 'source' or 'source code' and contains most of the source code for programs that the training recipes call.

For each training recipe directory, there is a standard sub-directory structure. This is best exemplified in the Resource Management directory (`egs/rm/s5`). The top directory contains the run script (`run.sh`), as well as two other required scripts (`cmd.sh` and `path.sh`). The sub-directories are `conf` (configuration), `data`, `exp` (experiments), `local`, `steps`, and `utils` (utilities). The directories we will primarily be using are `data` and `exp`. The `data` directory will eventually house information relevant to your own data such as transcripts, dictionaries, etc. The `exp` directory will eventually contain the output of the training and alignment scripts, or the acoustic models.

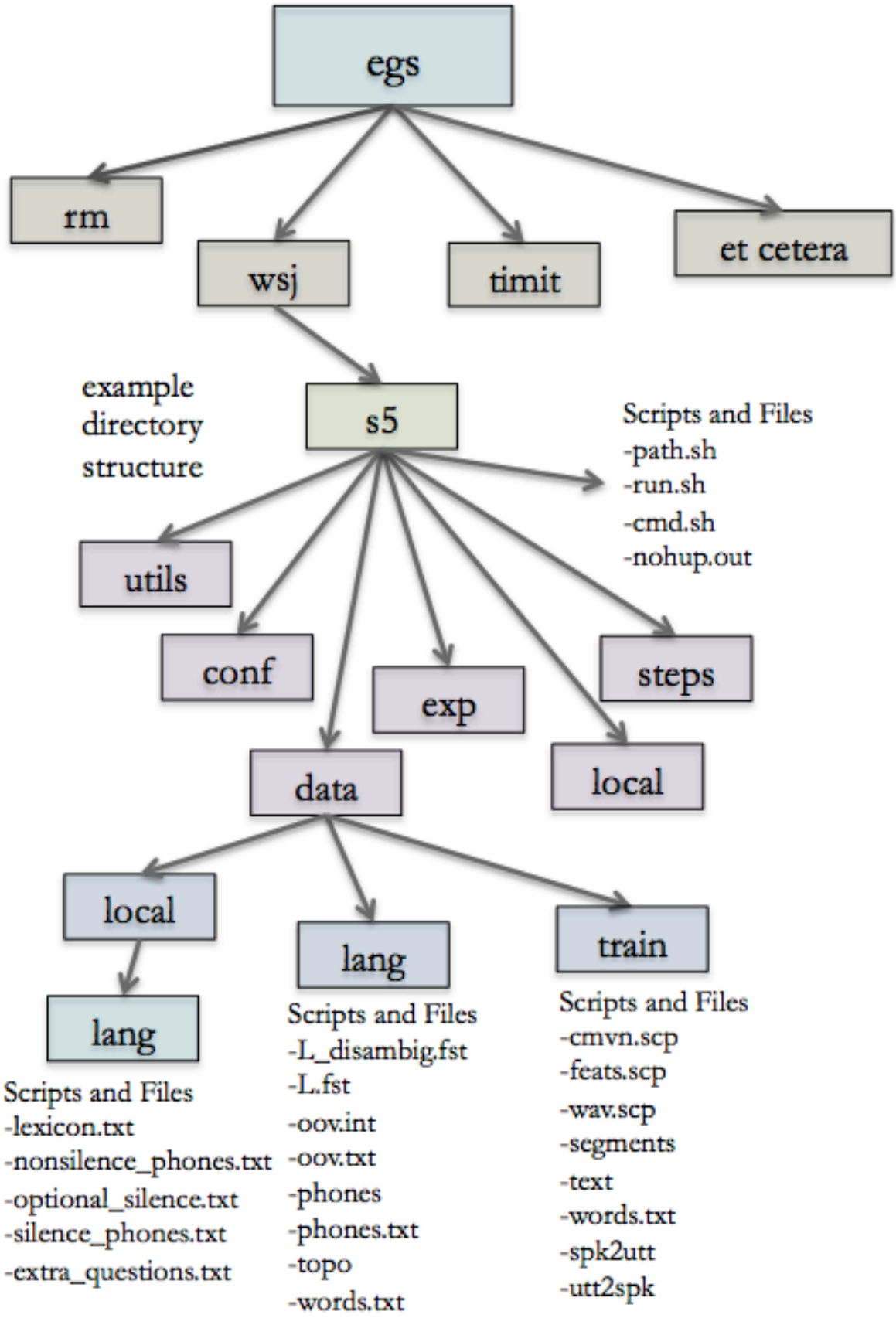

Figure 3.1: Example directory structure

Chapter 4

Training Overview

Before diving into the scripts, it is essential to understand the basic procedure for training acoustic models. Given the audience and purpose of the tutorial, this section will focus on the process as opposed to the computation (see Jurafsky and Martin 2008, Young 1996, among many others). The procedure can be laid out as follows:

1. Obtain a written transcript of the speech data

For a more precise alignment, utterance (~sentence) level start and end times are helpful, but not necessary.

2. Format transcripts for Kaldi

Kaldi requires various formats of the transcripts for acoustic model training. You'll need the start and end times of each utterance, the speaker ID of each utterance, and a list of all words and phonemes present in the transcript.

3. Extract acoustic features from the audio

Mel Frequency Cepstral Coefficients (MFCC) are the most commonly used features, but Perceptual Linear Prediction (PLP) features and other features are also an option. These features serve as the basis for the acoustic models.

4. Train monophone models

A monophone model is an acoustic model that does not include any contextual information about the preceding or following phone. It is used as a building block for the triphone models, which do make use of contextual information.

*Note: from this point forward, we will be assuming a Gaussian Mixture Model/Hidden Markov Model (GMM/HMM) framework. This is in contrast to a deep neural network (DNN) system.

5. Align audio with the acoustic models

The parameters of the acoustic model are estimated in acoustic training steps; however, the process can be better optimized by cycling through training and alignment phases. This is also known as Viterbi training (related, but more computationally expensive procedures include the Forward-Backward algorithm and Expectation Maximization). By aligning the audio to the reference transcript with the most current acoustic model, additional training algorithms can then use this output to improve or refine the parameters of the model. Therefore, each training step will be followed by an alignment step where the audio and text can be realigned.

6. Train triphone models

While monophone models simply represent the acoustic parameters of a single phoneme, we know that phonemes will vary considerably depending on their particular context. The triphone models represent a

phoneme variant in the context of two other (left and right) phonemes.

At this point, we'll also need to deal with the fact that not all triphone units are present (or will ever be present) in the dataset. There are (# of phonemes)^3 possible triphone models, but only a subset of those will actually occur in the data. Furthermore, the unit must also occur multiple times in the data to gather sufficient statistics for the data. A phonetic decision tree groups these triphones into a smaller amount of acoustically distinct units, thereby reducing the number of parameters and making the problem computationally feasible.

7. Re-align audio with the acoustic models & re-train triphone models

Repeat steps 5 and 6 with additional triphone training algorithms for more refined models. These typically include delta+delta-delta training, LDA-MLLT, and SAT. The alignment algorithms include speaker independent alignments and FMLLR.

- **Training Algorithms**

Delta+delta-delta training computes delta and double-delta features, or dynamic coefficients, to supplement the MFCC features. Delta and delta-delta features are numerical estimates of the first and second order derivatives of the signal (features). As such, the computation is usually performed on a larger window of feature vectors. While a window of two feature vectors would probably work, it would be a very crude approximation (similar to how a delta-difference is a very crude approximation of the derivative). Delta features are computed on the window of the original features; the delta-delta are then computed on the window of the delta-features.

LDA-MLLT stands for Linear Discriminant Analysis – Maximum Likelihood Linear Transform. The Linear Discriminant Analysis takes the feature vectors and builds HMM states, but with a reduced feature space for all data. The Maximum Likelihood Linear Transform takes the reduced feature space from the LDA and derives a unique transformation for each speaker. MLLT is therefore a step towards speaker normalization, as it minimizes differences among speakers.

SAT stands for Speaker Adaptive Training. SAT also performs speaker and noise normalization by adapting to each specific speaker with a particular data transform. This results in more homogenous or standardized data, allowing the model to use its parameters on estimating variance due to the phoneme, as opposed to the speaker or recording environment.

- **Alignment Algorithms**

The actual alignment algorithm will always be the same; the different scripts accept different types of acoustic model input.

Speaker independent alignment, as it sounds, will exclude speaker-specific information in the alignment process.

fMLLR stands for Feature Space Maximum Likelihood Linear Regression. After SAT training, the acoustic model is no longer trained on the original features, but on speaker-normalized features. For alignment, we essentially have to remove the speaker identity from the features by estimating the speaker identity (with the inverse of the fMLLR matrix), then removing it from the model (by multiplying the inverse matrix with the feature vector). These quasi-speaker-independent acoustic models can then be used in the alignment process.

Chapter 5

Training Acoustic Models

5.1 Prepare directories

Create a directory to house your training data and models:

```
cd kaldi/egs
mkdir mycorpus
```

The goal of the next few sections is to recreate the directory structure laid out in Section 3 on Familiarization. The structure we'll be building in this section starts at the node `mycorpus`:

In the following sections, we'll fill these directories in. For now, let's just create them.

Enter your new directory and make soft links to the following directories in the `wsj` directory to access necessary scripts: `steps`, `utils`, and `src`. In addition to the directories, you will also need a copy of the `path.sh` script in your `mycorpus` directory. **You will likely need to edit `path.sh` to make sure the KALDI-ROOT path is correct**. Make sure that the number of double dot levels takes you from your primary Kaldi directory (KALDI-ROOT) down to your working directory. For example, there are three levels between `kaldi` and `wsj/s5`, but only two levels between `kaldi` and `mycorpus`.

```
cd mycorpus
ln -s ../wsj/s5/steps .
ln -s ../wsj/s5/utils .
ln -s ../../src .

cp ../wsj/s5/path.sh .
```

Since the mycorpus directory is a level higher than `wsj/s5`, we need to edit the `path.sh` file.

```
vim path.sh

# Press i to insert; esc to exit insert mode;
# ':wq' to write and quit; ':q' to quit normally;
# ':q!' to quit forcibly (without saving)

# Change the path line in path.sh from:
export KALDI_ROOT='pwd'/../../..
# to:
export KALDI_ROOT='pwd'/../..
```

Finally, you will need to create the following directories in `mycorpus`: `exp`, `conf`, `data`. Within `data`, create

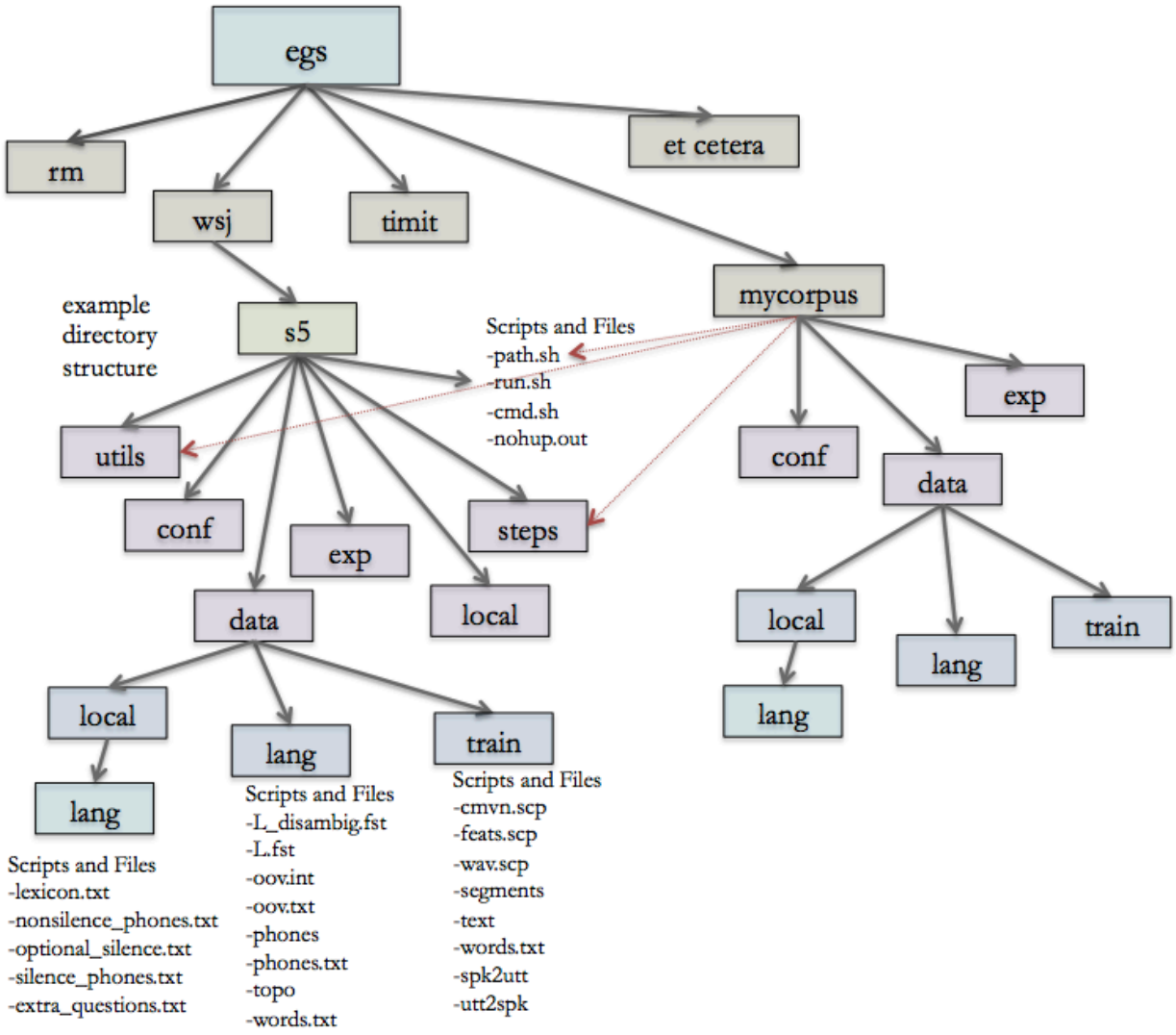

Figure 5.1: Directory structure to replicate

the following directories: `train`, `lang`, `local` and `local/lang`. The next few steps in the tutorial will explain how to fill these directories in.

```
cd mycorpus
mkdir exp
mkdir conf
mkdir data

cd data
mkdir train
mkdir lang
mkdir local

cd local
mkdir lang
```

5.2 Create files for data/train

The files in `data/train` contain information regarding the specifics of the audio files, transcripts, and speakers. Specifically, it will contain the following files:

- `text`
- `segments`
- `wav.scp`
- `utt2spk`
- `spk2utt`

5.2.1 text

The `text` file is essentially the utterance-by-utterance transcript of the corpus. This is a text file with the following format:

utt_id WORD1 WORD2 WORD3 WORD4 …

utt_id = utterance ID

Example text file:

110236_20091006_82330_F_0001 I'M WORRIED ABOUT THAT
110236_20091006_82330_F_0002 AT LEAST NOW WE HAVE THE BENEFIT
110236_20091006_82330_F_0003 DID YOU EVER GO ON STRIKE
…
120958_20100126_97016_M_0285 SOMETIMES LESS IS BETTER
120958_20100126_97016_M_0286 YOU MUST LOVE TO COOK

Once you've created `text`, the lexicon will also need to be reduced to only the words present in the corpus. This will ensure that there are no extraneous phones that we are "training."

The following code makes a list of words in the corpus and stores it in a file called `words.txt`. Note that when using the `cut` command, the default cut is delimited by tab (`cut -f 2`), but if the delimiter is anything other than tab, it can be specified as such: `cut -d 'my delimiter' -f 2- text`. `words.txt` will serve as input to a script, `filter_dict.py`, that downsizes the lexicon to only the words in the corpus.

```
cut -d ' ' -f 2- text | sed 's/ /\n/g' | sort -u > words.txt
```

Click on filter_dict.py to download

`filter_dict.py` takes `words.txt` and `lexicon.txt` as input and removes words from the lexicon that are not in the corpus. Remember that `lexicon.txt` should be in `/data/local/lang`. You will need to modify the path to `lexicon.txt` within the script `filter_dict.py`. You may also need to change the specified delimiter (tab, comma, space, etc.) within the file. `filter_dict.py` returns a modified `lexicon.txt`.

```
cd mycorpus
python filter_dict.py
```

One more modification needs to be made to the lexicon and that is adding the pseudo-word <oov> as an entry. <oov> stands for 'out of vocabulary'. Even though we ensured that all words present are indeed in the dictionary, the system requires that this option be present. At the top of your lexicon, add <oov> <oov>.

<oov> <oov>
A AH0
A EY1

5.2.2 segments

The `segments` file contains the start and end time for each utterance in an audio file. This is a text file with the following format:

utt_id file_id start_time end_time

utt_id = utterance ID
file_id = file ID
start_time = start time in seconds
end_time = end time in seconds

Example segments file:

110236_20091006_82330_F_001 110236_20091006_82330_F 0.0 3.44
110236_20091006_82330_F_002 110236_20091006_82330_F 4.60 8.54
110236_20091006_82330_F_003 110236_20091006_82330_F 9.45 12.05
110236_20091006_82330_F_004 110236_20091006_82330_F 13.29 16.13
110236_20091006_82330_F_005 110236_20091006_82330_F 17.27 20.36
110236_20091006_82330_F_006 110236_20091006_82330_F 22.06 25.46
110236_20091006_82330_F_007 110236_20091006_82330_F 25.86 27.56
110236_20091006_82330_F_008 110236_20091006_82330_F 28.26 31.24
...
120958_20100126_97016_M_282 120958_20100126_97016_M 915.62 919.67
120958_20100126_97016_M_283 120958_20100126_97016_M 920.51 922.69
120958_20100126_97016_M_284 120958_20100126_97016_M 922.88 924.27
120958_20100126_97016_M_285 120958_20100126_97016_M 925.35 927.88
120958_20100126_97016_M_286 120958_20100126_97016_M 928.31 930.51

5.2.3 wav.scp

`wav.scp` contains the location for each of the audio files. If your audio files are already in wav format, use the following template:

file_id path/file

Example `wav.scp` file:

110236_20091006_82330_F path/110236_20091006_82330_F.wav
111138_20091215_82636_F path/111138_20091215_82636_F.wav

111138_20091217_82636_F path/111138_20091217_82636_F.wav
...
120947_20100125_59427_F path/120947_20100125_59427_F.wav
120953_20100125_79293_F path/120953_20100125_79293_F.wav
120958_20100126_97016_M path/120958_20100126_97016_M.wav

If your audio files are in a different format (sphere, mp3, flac, speex), you will have to convert them to wav format. Instead of having to convert the files manually and storing multiple copies of the data, you can let Kaldi convert the files on-the-fly. The tool `sox` will come in handy in many of these cases. As an example of sphere (suffix `.sph`) to wav, you can use the following template; make sure to change `path` to the actual path where files are located. Also, don't forget the pipe (`|`).

```
file_id path/sph2pipe -f wav -p -c 1 path/file |
```

For an example using `sox`, this following code will convert the second channel of an 128kbit/s 44.1kHz joint-stereo mp3 file to a 8kHz mono wav file (which will be processed by Kaldi to generate the features):

```
file_id path/sox audio.mp3 -t wav -r 8000 -c 1 - remix 2|
```

5.2.4 utt2spk

`utt2spk` contains the mapping of each utterance to its corresponding speaker. As a side note, engineers will often conflate the term speaker with recording session, such that each recording session is a different "speaker". Therefore, the concept of "speaker" does not have to be related to a person – it can be a room, accent, gender, or anything that could influence the recording. When speaker normalization is performed then, the normalization may actually be removing effects due to the recording quality or particular accent type. This definition of "speaker" then is left up to the modeler.

`utt2spk` is a text file with the following format:

utt_id spkr

utt_id = utterance ID
spkr = speaker ID

Example `utt2spk` file:

110236_20091006_82330_F_0001 110236
110236_20091006_82330_F_0002 110236
110236_20091006_82330_F_0003 110236
110236_20091006_82330_F_0004 110236
...
120958_20100126_97016_M_0284 120958
120958_20100126_97016_M_0285 120958
120958_20100126_97016_M_0286 120958

Since the speaker ID in the first portion of our utterance IDs, we were able to use the following code to create the `utt2spk` file:

```
# this should be interpreted as one line of code
cat data/train/segments | cut -f 1 -d ' ' | \
perl -ane 'chomp; @F = split "_", $_; print $_ . " " . @F[0] . "\n";' > data/train/utt2spk
```

5.2.5 spk2utt

`spk2utt` is a file that contains the speaker to utterance mapping. This information is already contained in `utt2spk`, but in the wrong format. The following line of code will automatically create the spk2utt file and

simultaneously verify that all data files are present and in the correct format:

```
utils/fix_data_dir.sh data/train
```

While `spk2utt` has already been created, you can verify that it has the following format:

spkr utt_id1 utt_id2 utt_id3

5.3 Create files for `data/local/lang`

`data/local/lang` is the directory that contains language data specific to the your own corpus. For example, the lexicon only contains words and their pronunciations that are present in the corpus. This directory will contain the following:

- `lexicon.txt`
- `nonsilence_phones.txt`
- `optional_silence.txt`
- `silence_phones.txt`
- `extra_questions.txt` (optional)

5.3.1 lexicon.txt

You will need a pronunciation lexicon of the language you are working on. A good English lexicon is the CMU dictionary, which you can find here. The lexicon should list each word on its own line, capitalized, followed by its phonemic pronunciation

WORD W ER D
LEXICON L EH K S IH K AH N

The pronunciation alphabet must be based on the same phonemes you wish to use for your acoustic models. You must also include lexical entries for each "silence" or "out of vocabulary" phone model you wish to train.

Once you've created the lexicon, move it to `data/local/lang/`.

```
cp lexicon.txt kaldi-trunk/egs/mycorpus/data/local/lang/
```

5.3.2 nonsilence_phones.txt

As the name indicates, this file contains a list of all the phones that are not silence. Edit `phones.txt` so that *like phones* are on the same line. For example, `AA0`, `AA1`, and `AA2` would go on the same line; `K` would go on a different line. Then save this as `nonsilence_phones.txt`.

```
# this should be interpreted as one line of code
cut -d ' ' -f 2- lexicon.txt |  \
sed 's/ /\n/g' | \
sort -u > nonsilence_phones.txt
```

5.3.3 silence_phones.txt

`silence_phones.txt` will contain a 'SIL' (silence) and 'oov' (out of vocabulary) model. optional_silence.txt will just contain a 'SIL' model. This can be created with the following code:

```
echo -e 'SIL'\\n'oov' > silence_phones.txt
```

5.3.4 optional_silence.txt

optional_silence.txt will simply contain a 'SIL' model. Use the following code to create that file.

```
echo 'SIL' > optional_silence.txt
```

5.3.5 extra_questions.txt

A Kaldi script will generate a basic extra_questions.txt file for you, but in data/lang/phones. This file "asks questions" about a phone's contextual information by dividing the phones into two different sets. An algorithm then determines whether it is at all helpful to model that particular context. The standard extra_questions.txt will contain the most common "questions." An example would be whether the phone is word-initial vs word-final. If you do have extra questions that are not in the standard extra_questions.txt file, they would need to be added here.

5.4 Create files for data/lang

Now that we have all the files in data/local/lang, we can use a script to generate all of the files in data/lang.

```
cd mycorpus
utils/prepare_lang.sh data/local/lang 'OOV' data/local/ data/lang

# where the underlying argument structure is:
utils/prepare_lang.sh <dict-src-dir> <oov-dict-entry> <tmp-dir> <lang-dir>
```

The second argument refers to lexical entry (word) for a "spoken noise" or "out of vocabulary" phone. Make sure that this entry and its corresponding phone (oov) are entered in lexicon.txt and the phone is listed in silence_phones.txt.

Note that some older versions of Kaldi allowed the source and tmp directories to refer to the same location. These must now point to different directories.

The new files located in data/lang are L.fst, L_disambig.fst, oov.int, oov.txt, phones.txt, topo, words.txt, and phones. phones is a directory containing many additional files, including the extra_questions.txt file mentioned in section 5.3. It is worth taking a look at this file to see how the model may be learning more about a phoneme's contextual information. You should notice fairly logical and linguistically motivated divisions among the phones.

5.5 Set the parallelization wrapper

Training can be computationally expensive; however, if you have multiple processors/cores or even multiple machines, there are ways to speed it up significantly. Both training and alignment can be made more efficient by splitting the dataset into smaller chunks and processing them in parallel. The number of jobs or splits in the dataset will be specified later in the training and alignment steps. Kaldi provides a wrapper to implement this parallelization so that each of the computational steps can take advantage of the multiple processors. Kaldi's wrapper scripts are run.pl, queue.pl, and slurm.pl, along with a few others we won't discuss here. The applicable script and parameters will then be specified in a file called cmd.sh located at the top level of your corpus' training directory.

- run.pl allows you to run the tasks on a local machine (e.g., your personal computer).
- queue.pl allows you to allocate jobs on machines using Sun Grid Engine (see also Grid Computing).

- `slurm.pl` allows you to allocate jobs on machines using another grid engine software, called SLURM.

The parallelization can be specified separately for training and decoding (alignment of new audio) in the file `cmd.sh`. The following code provides an example using parameters specific to the Johns Hopkins CLSP cluster. If you are training on a personal computer or do not have a grid engine, you can set `train_cmd` and `decode_cmd` to `"run.pl"`.

As a side note, `vim` is a text editor that operates within the Unix shell. The commented portion of text provides the crucial commands you'll need to know to insert, change modes, write, and quit the editor. Finally, `cmd.sh` will automatically be created by typing `vim cmd.sh`.

```
cd mycorpus
vim cmd.sh

# Press i to insert; esc to exit insert mode;
# ':wq' to write and quit; ':q' to quit normally;
# ':q!' to quit forcibly (without saving)

# Insert the following text in cmd.sh
train_cmd="queue.pl"
decode_cmd="queue.pl  --mem 2G"
```

Please see http://www.kaldi-asr.org/doc/queue.html for how to correctly configure this.

Once you've quite vim, then run the file:

```
cd mycorpus
. ./cmd.sh
```

5.6 Create files for `conf`

The directory `conf` requires one file `mfcc.conf`, which contains the parameters for MFCC feature extraction. The text file includes the following information:

–use-energy=false
–sample-frequency=16000

The sampling frequency should be modified to reflect your audio data. This file can be created manually or within the shell with the following code:

```
# Create mfcc.conf by opening it in a text editor like vim
cd mycorpus/conf
vim mfcc.conf

# Press i to insert; esc to exit insert mode;
# ':wq' to write and quit; ':q' to quit normally;
# ':q!' to quit forcibly (without saving)

# Insert the following text in mfcc.conf

--use-energy=false
--sample-frequency=16000
```

5.7 Extract MFCC features

The following code will extract the MFCC acoustic features and compute the cepstral mean and variance normalization (CMVN) stats. After each process, it also fixes the data files to ensure that they are still in the correct format.

The `--nj` option is for the number of jobs to be sent out. This number is currently set to 16 jobs, which means that the data will be divided into 16 sections. It is good to note that Kaldi keeps data from the same speakers together, so you do not want more splits than the number of speakers you have.

```
cd mycorpus

mfccdir=mfcc
x=data/train
steps/make_mfcc.sh --cmd "$train_cmd" --nj 16 $x exp/make_mfcc/$x $mfccdir
steps/compute_cmvn_stats.sh $x exp/make_mfcc/$x $mfccdir
```

5.8 Monophone training and alignment

- **Take subset of data for monophone training**

The monophone models are the first part of the training procedure. We will only train a subset of the data mainly for efficiency. Reasonable monophone models can be obtained with little data, and these models are mainly used to bootstrap training for later models.

The listed argument options for this script indicate that we will take the first part of the dataset, followed by the location the data currently resides in, followed by the number of data points we will take (10,000), followed by the destination directory for the training data.

```
cd mycorpus
utils/subset_data_dir.sh --first data/train 10000 data/train_10k
```

- **Train monophones**

Each of the training scripts takes a similar baseline argument structure with optional arguments preceding those. The one exception is the first monophone training pass. Since a model does not yet exist, there is no source directory specifically for the model. The required arguments are always:

```
- Location of the acoustic data: `data/train`
- Location of the lexicon: `data/lang`
- Source directory for the model: `exp/lastmodel`
- Destination directory for the model: `exp/currentmodel`
```

The argument `--cmd "$train_cmd"` designates which machine should handle the processing. Recall from above that we specified this variable in the file `cmd.sh`. The argument `--nj` should be familiar at this point and stands for the number of jobs. Since this is only a subset of the data, we have reduced the number of jobs from 16 to 10. Boost silence is included as standard protocol for this training.

```
steps/train_mono.sh --boost-silence 1.25 --nj 10 --cmd "$train_cmd" \
data/train_10k data/lang exp/mono_10k
```

- **Align monophones**

Just like the training scripts, the alignment scripts also adhere to the same argument structure. The required arguments are always:

```
- Location of the acoustic data: `data/train`
- Location of the lexicon: `data/lang`
```

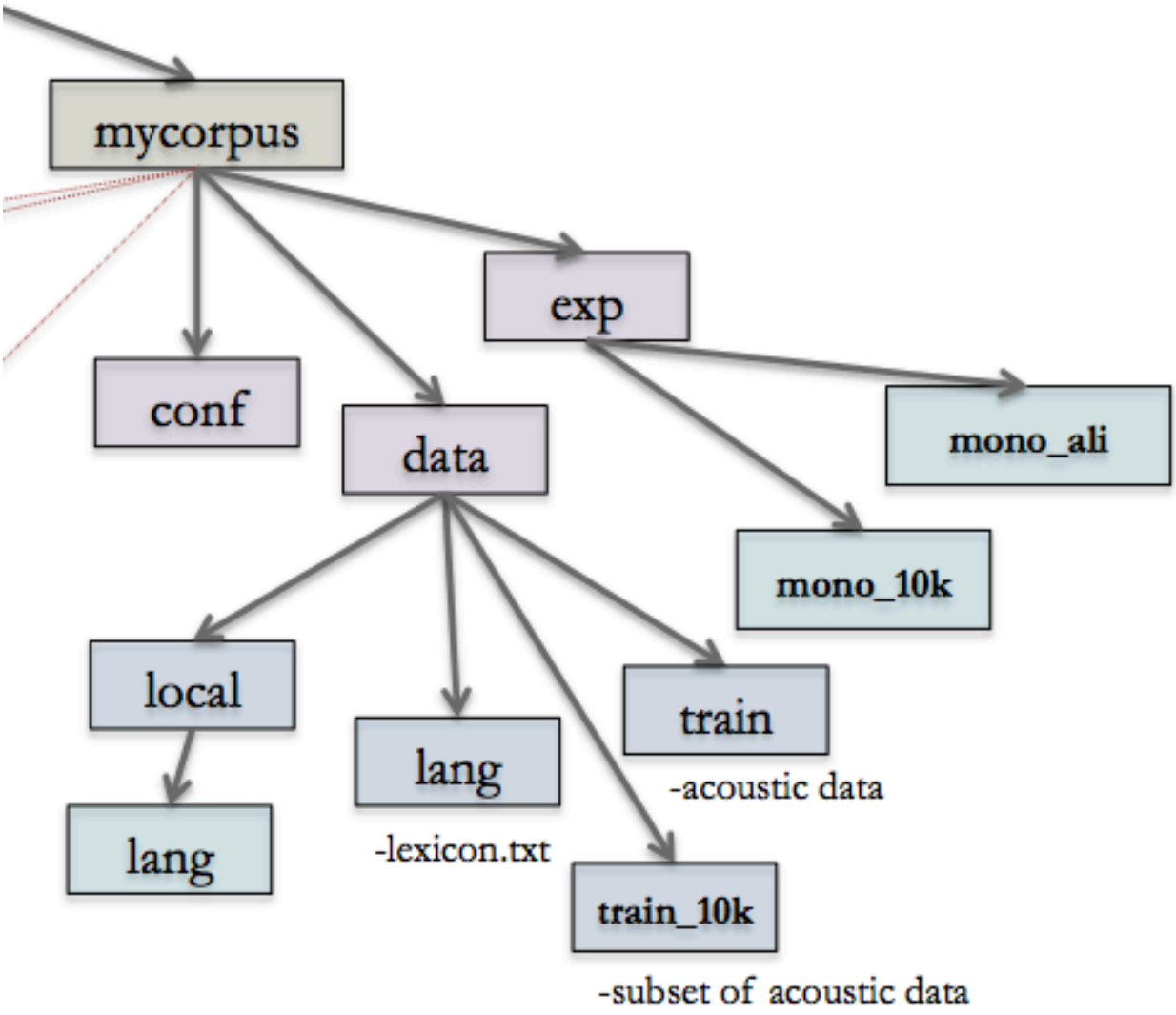

Figure 5.2: Output directory structure

```
- Source directory for the model: `exp/currentmodel`
- Destination directory for the alignment: `exp/currentmodel_ali`
```

```
steps/align_si.sh --boost-silence 1.25 --nj 16 --cmd "$train_cmd" \
data/train data/lang exp/mono_10k exp/mono_ali || exit 1;
```

The directory structure should now look something like this:

5.9 Triphone training and alignment

- **Train delta-based triphones**

Training the triphone model includes additional arguments for the number of leaves, or HMM states, on the decision tree and the number of Gaussians. In this command, we specify 2000 HMM states and 10000 Gaussians. As an example of what this means, assume there are 50 phonemes in our lexicon. We could have

one HMM state per phoneme, but we know that phonemes will vary considerably depending on if they are at the beginning, middle or end of a word. We would therefore want *at least* three different HMM states for each phoneme. This brings us to a minimum of 150 HMM states to model just that variation. With 2000 HMM states, the model can decide if it may be better to allocate a unique HMM state to more refined allophones of the original phone. This phoneme splitting is decided by the phonetic questions in `questions.txt` and `extra_questions.txt`. The allophones are also referred to as subphones, senones, HMM states, or leaves.

The exact number of leaves and Gaussians is often decided based on heuristics. The numbers will largely depend on the amount of data, number of phonetic questions, and goal of the model. There is also the constraint that the number of Gaussians should always exceed the number of leaves. As you'll see, these numbers increase as we refine our model with further training algorithms.

```
steps/train_deltas.sh --boost-silence 1.25 --cmd "$train_cmd" \
2000 10000 data/train data/lang exp/mono_ali exp/tri1 || exit 1;
```

- **Align delta-based triphones**

```
steps/align_si.sh --nj 24 --cmd "$train_cmd" \
data/train data/lang exp/tri1 exp/tri1_ali || exit 1;
```

- **Train delta + delta-delta triphones**

```
steps/train_deltas.sh --cmd "$train_cmd" \
2500 15000 data/train data/lang exp/tri1_ali exp/tri2a || exit 1;
```

- **Align delta + delta-delta triphones**

```
steps/align_si.sh  --nj 24 --cmd "$train_cmd" \
--use-graphs true data/train data/lang exp/tri2a exp/tri2a_ali  || exit 1;
```

- **Train LDA-MLLT triphones**

```
steps/train_lda_mllt.sh --cmd "$train_cmd" \
3500 20000 data/train data/lang exp/tri2a_ali exp/tri3a || exit 1;
```

- **Align LDA-MLLT triphones with FMLLR**

```
steps/align_fmllr.sh --nj 32 --cmd "$train_cmd" \
data/train data/lang exp/tri3a exp/tri3a_ali || exit 1;
```

- **Train SAT triphones**

```
steps/train_sat.sh  --cmd "$train_cmd" \
4200 40000 data/train data/lang exp/tri3a_ali exp/tri4a || exit 1;
```

- **Align SAT triphones with FMLLR**

```
steps/align_fmllr.sh  --cmd "$train_cmd" \
data/train data/lang exp/tri4a exp/tri4a_ali || exit 1;
```

Chapter 6

Forced Alignment

Once acoustic models have been created, Kaldi can also perform forced alignment on audio accompanied by a word-level transcript. Note that the Montreal Forced Aligner{target = "_blank"} is a forced alignment system based on Kaldi-trained acoustic models for several world languages. You could also considering checking out FAVE for aligning American English speech.

Otherwise, if the audio to be aligned is the same as the audio used in the acoustic models, then the alignments can be extracted directly from the alignment files. If you have new audio and transcripts, then the transcript files will need to be updated before alignment.

The full procedure will convert output from the model alignment into Praat TextGrids containing the phone-level transcript.

If the data to be aligned is the same as the training data, skip to Section 6.4. Otherwise, you'll need to update the transcript files and audio file specifications.

6.1 Prepare alignment files

To extract alignments for new transcripts and audio, you'll need to create new versions of the files in the directory `data/train`. As a reminder, these files are `text`, `segments`, `wav.scp`, `utt2spk`, and `spk2utt` (see Section 5.2). We'll house these in a new directory in `mycorpus/data`.

```
# create text, segments, wav.scp, utt2spk, and spk2utt

cd mycorpus/data
mkdir alignme
```

6.2 Extract MFCC features

Revisit Section 5.7 on MFCC feature extraction for reference. You'll need to replace `data/train` with the the new directory, `data/alignme`.

```
cd mycorpus

mfccdir=mfcc
for x in data/alignme
do
    steps/make_mfcc.sh --cmd "$train_cmd" --nj 16 $x exp/make_mfcc/$x $mfccdir
```

```
    utils/fix_data_dir.sh data/alignme
    steps/compute_cmvn_stats.sh $x exp/make_mfcc/$x $mfccdir
    utils/fix_data_dir.sh data/alignme
done
```

6.3 Align data

Revisit Section 5.9 on triphone training and alignment for reference. Select the acoustic model and corresponding alignment process you'd like to use. You'll need to replace `data/train` with the the new directory, `data/alignme`. As an example:

```
cd mycorpus
steps/align_si.sh --cmd "$train_cmd" data/alignme data/lang exp/tri4a exp/tri4a_alignme || exit 1;
```

6.4 Extract alignment

- **Obtain CTM output from alignment files**

CTM stands for time-marked conversation file and contains a time-aligned phoneme transcription of the utterances. Its format is:

utt_id channel_num start_time end_time phone_id

To obtain these, you will need to decide which acoustic models to use. The following code will extract the CTM output from the alignment files in the directory `tri4a_alignme`, using the acoustic models in `tri4a`:

```
cd mycorpus

for i in  exp/tri4a_alignme/ali.*.gz;
do src/bin/ali-to-phones --ctm-output exp/tri4a/final.mdl ark:"gunzip -c $i|" -> ${i%.gz}.ctm;
done;
```

- **Concatenate CTM files**

```
cd mycorpus/exp/tri4a_alignme
cat *.ctm > merged_alignment.txt
```

- **Convert time marks and phone IDs**

The CTM output reports start and end times relative to the utterance, as opposed to the file. You will need the `segments` file located in either `data/train` or `data/alignme` to convert the utterance times into file times.

The output also reports the phone ID, as opposed to the phone itself. You will need the `phones.txt` file located in `data/lang` to convert the phone IDs into phone symbols.

Click on id2phone.R to download

After obtaining the `segments` and `phones.txt` files, run `id2phone.R` to convert phone IDs to phones characters and map utterance times to file times. You will need to modify the file locations and possibly the regular expression to obtain the filename from the utterance name. Recall that the CTM output lists the utterance ID whereas the segments file lists the file ID. (If you named things logically, the file ID should be a subset of the utterance ID.)

`id2phone.R` returns a modified version of `merged_alignment.txt` called `final_ali.txt`

- **Split `final_ali.txt` by file**

Click on splitAlignments.py to download

`final_ali.txt` contains the phone transcript for all files together. This can be split into unique files by running `splitAlignments.py`. You will need to modify the location of `final_ali.txt` in this script.

```
python splitAlignments.py
```

- **Create word alignments from phone endings**

First we'll need to use the [B I E S] suffixes on the phones in order to group phones together into word-level units.

Run phons2pron.py to complete this step. Note that I have utf-8 character encoding on this script. If necessary, this can be updated to reflect the character encoding that best matches your files.

Second, we'll need to match the phone pronunciation to the corresponding lexical entry using `lexicon.txt`.

Run pron2words.py to complete this step.

6.5 Create Praat TextGrids

- **Append header to each of the text files for Praat**

Praat requires that a text file have a header. Once we append the header, then we can convert these text files into TextGrids. The following code requires a text file containing the header:

file_utt file id ali startinutt dur phone start_utt end_utt start end

It also requires a `tmp` directory for processing. I put this on my Desktop.

```
cd ~/Desktop
mkdir tmp

header="/Users/Eleanor/Desktop/header.txt"

# direct the terminal to the directory with the newly split session files
# ensure that the RegEx below will capture only the session files
# otherwise change this or move the other .txt files to a different folder

cd mycorpus/forcedalignment
for i in *.txt;
do
    cat "$header" "$i" > /Users/Eleanor/Desktop/tmp/xx.$$
    mv /Users/Eleanor/Desktop/tmp/xx.$$ "$i"
done;
```

- **Make Praat TextGrids of phone alignments from `.txt` files**

createtextgrid.praat will read in the new phone transcripts and corresponding audio files to create a TextGrid for that file. You will need to modify the locations of the phone transcripts and audio files.

- **Make Praat TextGrids for word alignments from `word_alignment.txt`**

Click on createWordTextGrids.praat to download

- **Stack phone and word TextGrids**

stackTextGrids.praat