arXiv:1811.05553v1 [cs.CL] 13 Nov 2018

Corpus Phonetics Tutorial

Eleanor Chodroff

Core content written: 2015 | Updated: 2018-11-13

Contents

1 Introduction **5**

2 Kaldi **7**
2.1 Overview . . . 7
2.2 Installation . . . 7
2.3 Familiarization . . . 8
2.4 Training Overview . . . 8
2.5 Training Acoustic Models . . . 11
2.6 Forced Alignment . . . 21

3 FAVE-align **25**
3.1 Overview . . . 25
3.2 Installation . . . 25
3.3 Running the aligner . . . 26

4 Montreal Forced Aligner **27**
4.1 Overview . . . 27
4.2 Setup . . . 27
4.3 Grapheme-to-phoneme models . . . 29
4.4 Running the aligner . . . 29
4.5 Tips and tricks . . . 30

5 Penn Forced Aligner (Legacy) **31**
5.1 Overview . . . 31
5.2 Prerequisites . . . 31
5.3 Modifying the lexicon . . . 32
5.4 Running the aligner . . . 33

6 AutoVOT **35**
6.1 Overview . . . 35
6.2 Recipe . . . 35

7 Other resources **41**

IPR2026-00272
Krisp Technologies EX1037 Page 3

Chapter 1

Introduction

Corpus phonetics has become an increasingly popular method of research in linguistic analysis. With advances in speech technology and computational power, large scale processing of speech data has become a viable technique. A fair number of researchers have exploited these methods, yet these techniques still remain elusive for many. In the words of Mark Liberman, there has been "surprisingly little change in style and scale of [phonetic] research" from 1966 on, implying that the field still relies on small sample sizes of speech data (2009). While "big data" phonetics is not the be-all and end-all of phonetic research, larger sample sizes ensure more statistically sound conclusions about phonetic values in an individual or population. Furthermore, corpus research is not synonymous with big data. Rather, corpus phonetics describes a method of processing speech data with advantages primarily gained in its computational power (relation to big data) and efficiency. The methods and tools developed for corpus phonetics are based on engineering algorithms primarily from automatic speech recognition (ASR), as well as simple programming for data manipulation. This tutorial aims to bring some of these tools to the non-engineer, and specifically to the speech scientist.

Acoustic analysis programs such as Praat, MATLAB, and R (check out the tuneR and multitaper packages) are already capable of large scale phonetic measurement via their respective scripting languages. While the tutorial covers some phonetic processing in Praat, the primary aim is to introduce supplementary tools to phonetic processing. These tools are based on concepts and algorithms from automatic speech recognition, which allow for automatic alignment of phonetic boundaries to the speech signal.

In particular, the tutorial currently covers various tools from the Kaldi Automatic Speech Recognition Toolki, FAVE-align, the Montreal Forced Aligner, Penn Phonetics Lab Forced Aligner, and AutoVOT. (The documentation for the Penn Forced Aligner is marked as "legacy" as this system has essentially been replaced by FAVE-align.) Kaldi is an automatic speech recognition toolkit that provides the infrastructure to build personalized **acoustic models** and **forced alignment** systems. Acoustic models are the statistical representations of each phoneme's acoustic information. The "personalized" component means that this system is capable of modeling any corpus of speech, be it British English, Southern American English, Hungarian, or Korean. It additionally houses many speech processing algorithms, which may be of use to the speech scientist. This tutorial will cover acoustic model training and forced alignment in Kaldi; however, the toolkit as a whole provides exceptional potential for phonetic research. "Forced alignment" is the automatic synchronization of a sequence of phones with an audio file. This process employs **acoustic models** of the sounds of a language, along with a pronunciation lexicon which provides a canonical mapping from orthographic words to sequences of phones. Forced alignment greatly expedites data processing and phonetic measurement. Kaldi, FAVE-align, and the Montreal Forced Aligner are all capable of forced alignment, but with varying degrees of flexibility with respect to the input speech. Finally, AutoVOT is an automatic voice onset time (VOT) measurement tool that demarcates the burst release and vocalic onset of word-initial, prevocalic stop consonants.

Finally, the tutorial assumes basic familiarity with Praat, as well as a Mac operating system, primarily for the default bash/Unix shell in the Terminal application. If using a PC, I recommend downloading Cygwin for

IPR2026-00272
Krisp Technologies EX1037 Page 5

running bash/Unix commands. For AutoVOT and the Penn Forced Aligner, most of the Unix commands are provided in the tutorial itself. While I try to provide as many of the commands as possible, Kaldi requires more fluency in shell scripting. If you have not used the Terminal application before, I recommend looking over some basic Unix commands online (Google is every programmer's best friend). For a list of the most useful commands, I recommend this website. For more details regarding the argument structure, I recommend this website.

Each section covers the prerequisites for each program's installation, as well as a standard recipe for each program. As a good rule of thumb, all prerequisites should be installed prior to installation of the desired program.

Citations for each of the programs can be found below:

- Kaldi

Povey, D., Ghoshal, A., Boulianne, G., Burget, L., Glembek, O., Goel, N., Hannemann, M., Motlicek, P., Qian, Y., Schwartz, P., Silovsky, J., Stemmer, G., & Vesely, K. (2011). The Kaldi speech recognition toolkit. In IEEE 2011 Workshop on ASRU.

```
@INPROCEEDINGS{
         Povey_ASRU2011,
         author = {Povey, Daniel and Ghoshal, Arnab and Boulianne, Gilles and
           Burget, Lukas and Glembek, Ondrej and Goel, Nagendra and
           Hannemann, Mirko and Motlicek, Petr and Qian, Yanmin and
           Schwarz, Petr and Silovsky, Jan and Stemmer, Georg and Vesely, Karel},
       keywords = {ASR, Automatic Speech Recognition, GMM, HTK, SGMM},
          month = dec,
          title = {The Kaldi Speech Recognition Toolkit},
      booktitle = {IEEE 2011 Workshop on Automatic Speech Recognition and Understanding},
           year = {2011},
      publisher = {IEEE Signal Processing Society},
       location = {Hilton Waikoloa Village, Big Island, Hawaii, US},
           note = {IEEE Catalog No.: CFP11SRW-USB},
}
```

- FAVE-align

Rosenfelder, Ingrid; Fruehwald, Josef; Evanini, Keelan; Seyfarth, Scott; Gorman, Kyle; Prichard, Hilary; Yuan, Jiahong; 2014. FAVE (Forced Alignment and Vowel Extraction) Program Suite v1.2.2 10.5281/zenodo.22281

- Montreal Forced Aligner

McAuliffe, Michael, Michaela Socolof, Sarah Mihuc, Michael Wagner, and Morgan Sonderegger (2017). Montreal Forced Aligner [Computer program]. Version 0.9.0, retrieved 17 January 2017 from http://montrealcorpustools.github.io/Montreal-Forced-Aligner/.

- Penn Phonetics Lab Forced Aligner

Yuan, Jiahong., & Liberman, Mark. (2008). Speaker identification on the SCOTUS corpus. In Proceedings of Acoustics, '08.

- AutoVOT

Keshet, J., Sonderegger, M., Knowles, T. (2014). AutoVOT: A tool for automatic measurement of voice onset time using discriminative structured prediction [Computer program]. Version 0.91, retrieved August 2014 from https://github.com/mlml/autovot/.

Chapter 2

Kaldi

2.1 Overview

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.

2.2 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

IPR2026-00272
Krisp Technologies EX1037 Page 7

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
```

2.3 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.

2.4 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

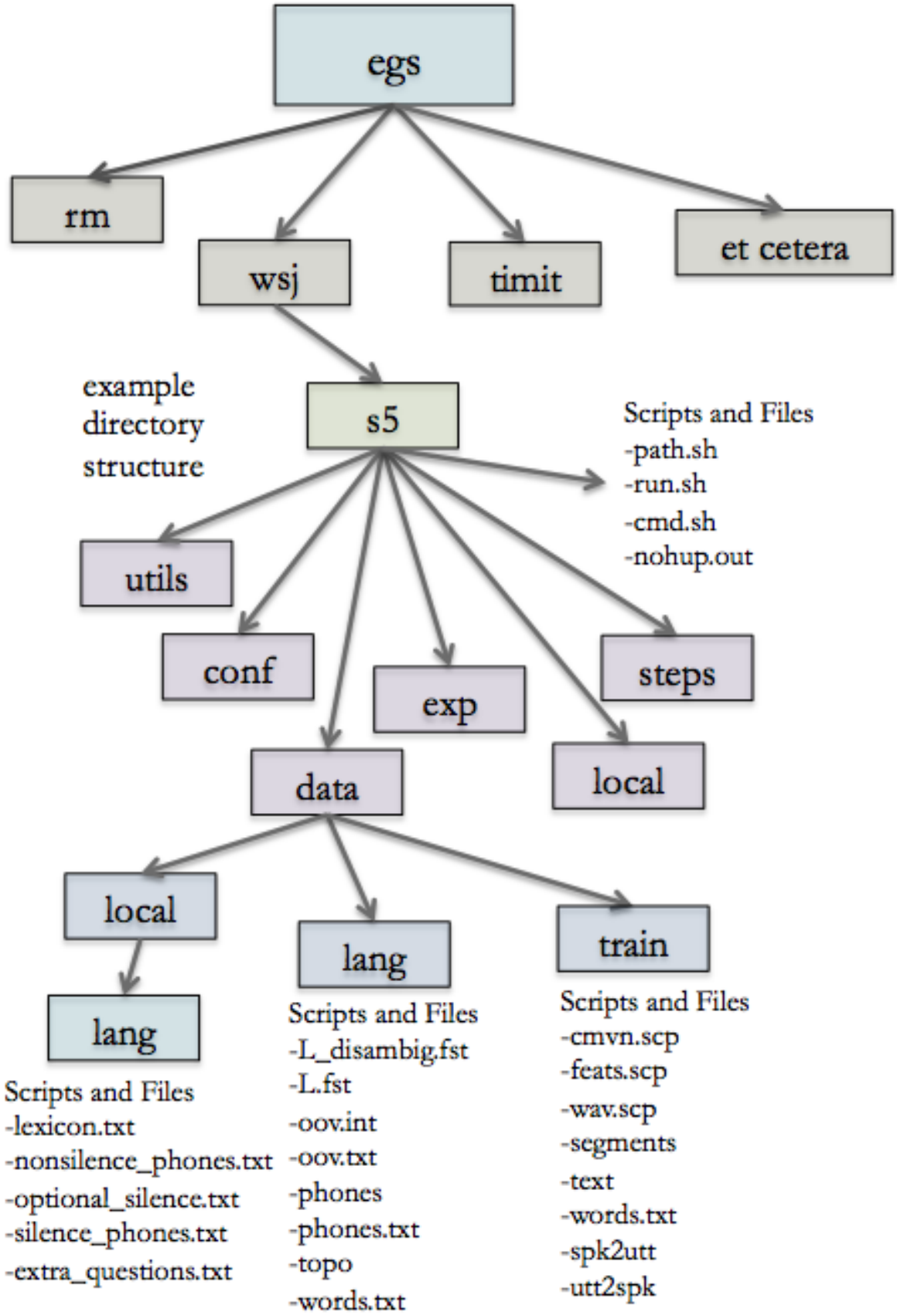

Figure 2.1: Example directory structure

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.

2.5 Training Acoustic Models

2.5.1 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 2.3 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)
```

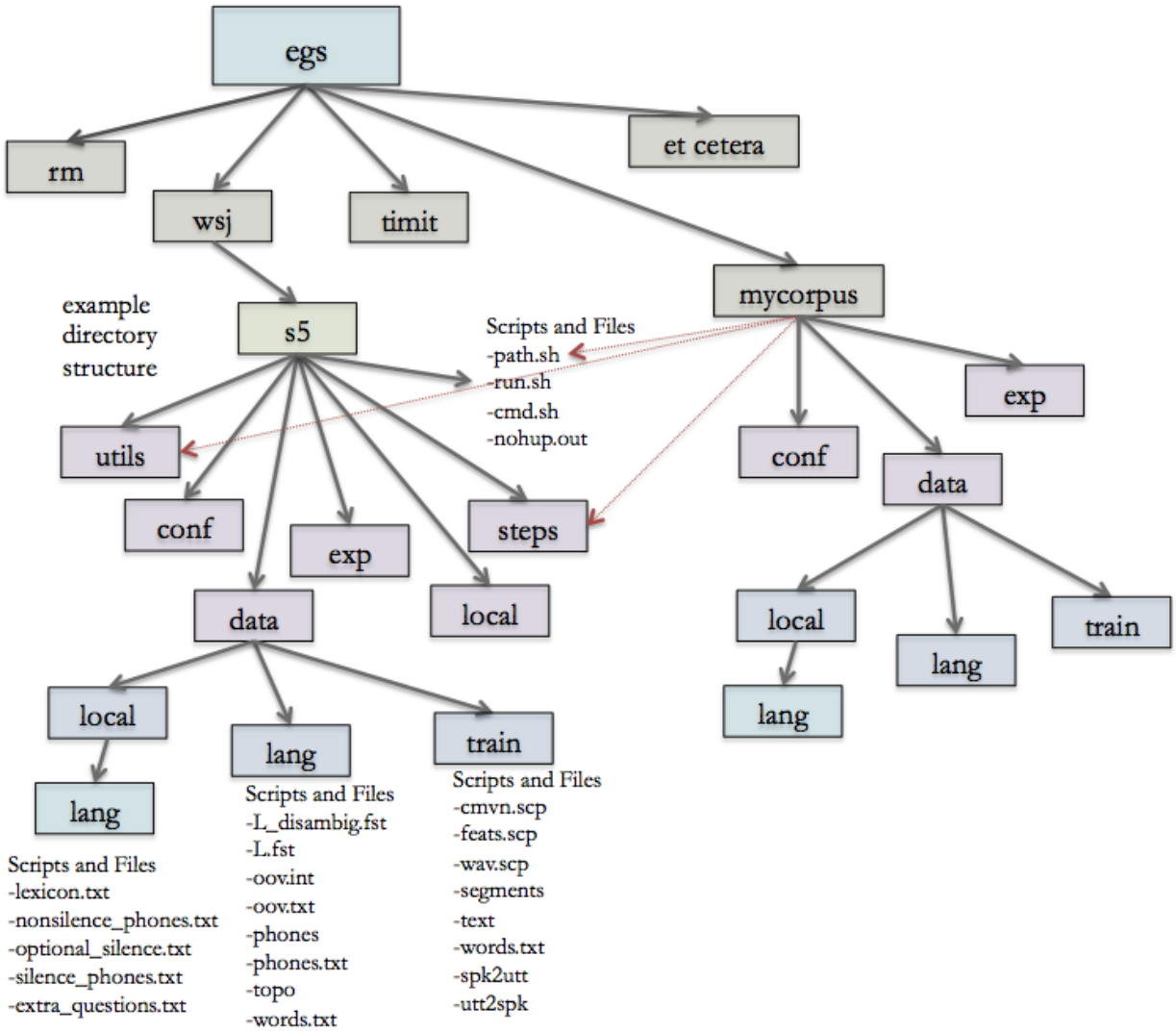

Figure 2.2: Directory structure to replicate

```
# 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 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
```

2.5.2 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`

2.5.2.1 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
```

An example script to accomplish this can be downloaded here: filter_dict.py

`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

2.5.2.2 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

2.5.2.3 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|
```

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
```

2.5.2.5 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

2.5.3 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)

2.5.3.1 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/
```

2.5.3.2 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
```

2.5.3.3 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
```

2.5.3.4 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
```

2.5.3.5 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.

2.5.4 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 2.5.3. 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.

2.5.5 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
```

2.5.6 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
```

2.5.7 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
```

2.5.8 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`
- Source directory for the model: `exp/currentmodel`
```

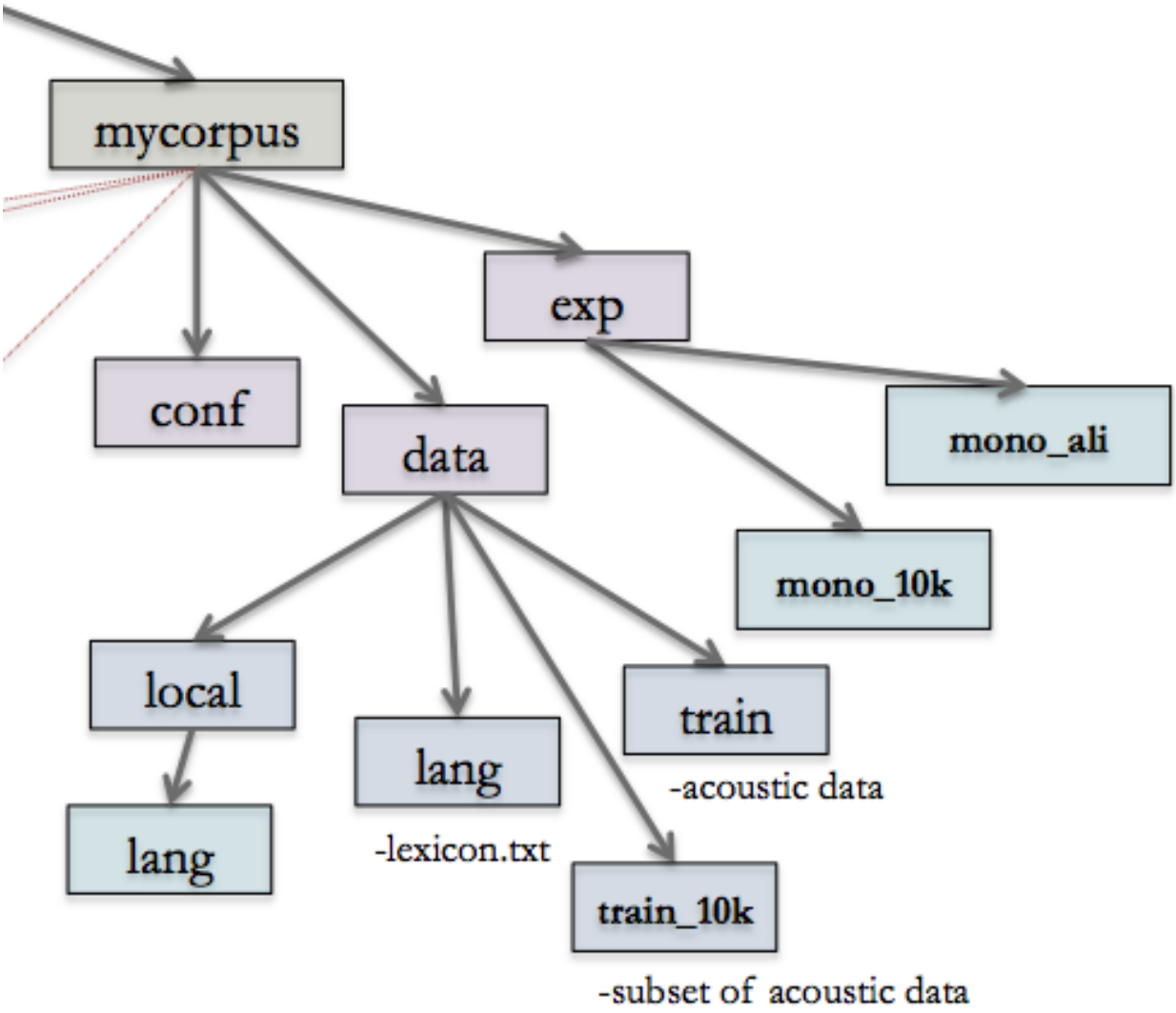

Figure 2.3: Output directory structure

```
- 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:

2.5.9 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;
```

2.6 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 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 2.6.4. Otherwise, you'll need to update the transcript files and audio file specifications.

2.6.1 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 2.5.2). We'll house these in a new directory in `mycorpus/data`.

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

cd mycorpus/data
mkdir alignme
```

2.6.2 Extract MFCC features

Revisit Section 2.5.7 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
```

2.6.3 Align data

Revisit Section 2.5.9 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;
```

2.6.4 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 phone_dur 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.

An example script to accommplish this can be downloaded here: id2phone.R

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**

An example script to accomplish this can be downloaded here: splitAlignments.py

`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.

2.6.5 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`**

An example script to accomplish this can be downloaded here: createWordTextGrids.praat

- **Stack phone and word TextGrids**

stackTextGrids.praat

Chapter 3

FAVE-align

3.1 Overview

What does FAVE do? FAVE is up-to-date implementation of the Penn Forced Aligner covered in section 5. Two programs are included in installation: FAVE-align, which performs forced alignment on American English speech, and FAVE-extract, which performs formant extraction and normalization algorithms. This tutorial focuses on FAVE-align, so future references of FAVE refer to the alignment program only. FAVE takes a sound file and utterance-level transcript (text file) and returns a Praat TextGrid with phone and word alignments. Excellent documentation and instructions for running the aligner can be found on the FAVE-align website. I will cover many of the instructions again, and try to include some additional tips, but provided installation goes smoothly, this aligner tends to work very well right out of the box.

What does it include? As in the Penn Forced Aligner, the system comes with *pre-trained acoustic models* of American English (AE) speech from 25 hours of the SCOTUS corpus, built with the HTK Speech Recognition Toolkit. The section on Kaldi introduces a similar, but alternative system to HTK, in case you'd like to train your own acoustic models from scratch. In addition to the acoustic models, the download also includes a large lexicon of words based on the CMU Pronouncing Dictionary. The dictionary contains over 100,000 words with their standard pronunciation transcriptions in Arpabet. Arpabet is a machine-readable phonetic alphabet of General American English with stress marked on the vowels. While I have, perhaps wrongly, used the Penn Forced Aligner to create phonetic alignments for other languages, those transcriptions had to be forced into Arpabet phones. For a quick alignment, this works fine, but please be advised: the acoustic models of the Penn Forced Aligner are trained on American English, so any speech it is given will be treated like American English. Unless manual adjustments follow the automatic alignment, the PFA alignment would not be ideal for most non-AE phonetic analyses. See the Montreal Forced Aligner (section 4) for pre-trained acoustic models of additional languages and Kaldi if you would like to create custom acoustic models for a particular language or dialect.

3.2 Installation

Please refer to the FAVE website for installation. As of writing, that page currently includes links to OS-specific downloads for each of the prerequisites.

The prerequisites for FAVE are:

- HTK Toolkit Version 3.4.1
  - Unlike p2fa, version 3.4.1 **will** work
  - If installing HTK on a Mac, there are additional prerequisites you'll need. Make sure to check out the FAVE wiki for these instructions.

IPR2026-00272
Krisp Technologies EX1037 Page 25

- Python 2.x
  - Pre-installed on Macs, but you can double-check by opening the terminal and typing python. This will return the version installed. I have been using Version 2.7 and have not had any issues running the aligner. Since you've typed python, the terminal is now running with the assumption of python syntax. You want to exit this to get back to its Unix base; to do this, type `exit()` or `quit()`. If you have an earlier version of Python, you may need to type `exit`.
- SoX
  - Download here: http://sourceforge.net/projects/sox/

To download FAVE, again follow the instructions on the website. FAVE can either be downloaded directly as a .zip or .tar.gz file here or cloned via git using the following command:

```
git clone https://github.com/JoFrhwld/FAVE.git
```

Cloning the system ensures that any new updates are automatically applied to the system.

3.3 Running the aligner

FAVE requires a wav file and a corresponding transcript that must adhere to a precise format, which is described below. Unlike some other alignment systems, the wav file does not need to have a sampling rate of 16 kHz. During alignment, the system will automatically create a temporary file with this sampling rate.

The text file must be tab-delimited (.txt) with the following five columns:

1. Speaker ID
2. Speaker name
3. Onset time (seconds)
4. Offset time (seconds)
5. Transcription of speech between onset and offset times

Some notes I've taken that could be useful:

- The speaker ID can be the same as the speaker name
- You can make up a very large number as a hack for indicating that the offset time should be the end of the file. The aligner will produce a warning, but still complete the alignment.
- FAVE sometimes struggles with spontaneous speech, though I imagine any forced alignment system might struggle with this. When the total number of phones in the transcript exceeded the total utterance time under the assumption that a phone is 30 ms, then the aligner produced overlapping intervals. This resulted in areas of Praat TextGrids with overlapping intervals which are visible but no longer functional, and now somewhat useless. It could be a bug in either Praat or FAVE.
- Some people have asked me how to align multiple files at once. This can be accomplished with a for loop in the shell script that uses regular expression matching to loop through files. Assuming the transcripts and wav files are in the same folder and differ only in their extension (.txt vs .wav), then you could use the following code (make sure to remove the backslashes and run as one line of code):

```
# direct shell to location of FAAValign.py script
cd /Users/Eleanor/FAVE/FAVE-align

# loop through transcript files in a second directory that contains
# the transcript files and like-named wav files
for i in /Users/Eleanor/myCorpus/*.txt; \
do python FAAValign.py "${i/.txt/.wav}" "$i" "${i/.txt/.TextGrid}"; \
done;
```

Chapter 4

Montreal Forced Aligner

4.1 Overview

The Montreal Forced Aligner is a forced alignment system with acoustic models built using the Kaldi ASR toolkit. A major highlight of this system is the availability of pretrained acoustic models and grapheme-to-phoneme models for a wide variety of languages.

The primary website contains excellent documentation, so I'll provide some tips and tricks I've picked up while using it.

A quick link to the installation instructions is located on the primary MFA website. This tutorial is based on Version 1.1.

4.2 Setup

As with any forced alignment system, the Montreal Forced Aligner will time-align a transcript to a corresponding audio file at the phone and word levels provided there exist a set of pretrained acoustic models and a lexicon/dictionary of the words in the transcript with their canonical phonetic pronunciation(s). The phone set used in the dictionary must match the phone set in the acoustic models. The orthography used in the dictionary must also match that in the transcript.

Very generally, the procedure is as follows:

- Prep wav file(s) (16 kHz, single channel)
- Prep transcript(s) (Praat TextGrid or .lab/.txt file)
- Obtain a pronunciation lexicon
- Obtain acoustic models

You will also need to identify or create an **input folder** that contains the wav files and TextGrids/transcripts and an **output folder** for the time-aligned TextGrid to be created.

Please make sure that you have separate input and output folders, and that the output folder is not a subdirectory of the input folder! The MFA deletes everything in the output folder: if it is the same as your input folder, the system will delete your input files.

IPR2026-00272
Krisp Technologies EX1037 Page 27

4.2.1 Wav files

The Montreal Forced Aligner works best and sometimes will only work with wav files that are sampled at 16 kHz and are single channel files. You may need to resample your audio and extract a single channel prior to running the aligner.

prep_audio_mfa.praat

4.2.2 Transcripts

The MFA can take as input either a Praat TextGrid or a `.lab` or `.txt` file. I have worked most extensively with the TextGrid input, so I'll describe those details here. As for `.lab` and `.txt` input, I have only tried running the aligner where the transcript is pasted in as a single line. I think there is a way of providing timestamps at the utterance level, but I can't speak to that yet.

The most straightforward implementation of the aligner with TextGrid input is to paste the transcript into a TextGrid with a single interval tier. The transcript *must* be delimited by boundaries on that tier; however, those boundaries *cannot* be located at either the absolute start or absolute end of the wav file (start boundary != 0, end boundary != total duration). In fact, I've found that the MFA can be very sensitive to the location of the end boundary: it's best to have at least 20 ms, if not 50 ms+ between the final TextGrid boundary and the end of the wav file (see also Section 4.5 on Tips and Tricks).

If you have utterance-level timestamps, you can also add in additional intervals for an alignment that is less likely to "derail". By "derail", I mean that the aligner gets thrown off early on in the wav file and never gets back on track, which yields a fairly misaligned "alignment". By delimiting the temporal span of an utterance, the aligner has a chance to reset at the next utterance, even if the preceding utterance was completely misaligned. Side note: misalignments are more likely to occur if there's additional noise in the wav file (e.g., coughing, background noise) or if the speech and transcript don't match at either the word or phone level (e.g., pronunciation of a word does not match the dictionary/lexicon entry).

Here are few sample Praat scripts I employ for creating TextGrids.

If I don't have timestamps, but I do have a transcript: create_textgrid_mfa_simple.praat

If the transcript has start and end times for each utterance (3 column text file with start time, end time, text): create_textgrid_mfa_timestamps.praat

That last Praat script can also be modified for a transcript with either start or end times, but not both. Make sure to follow the "rules" (which may change) that text-containing intervals be separated by empty intervals and the boundaries do not align with either the absolute start or end of the file.

4.2.3 Pronunciation lexicon

The pronunciation lexicon must be a two column text file with a list of words on the lefthand side and the phonetic pronunciation(s) on the righthand side. Many-to-many mappings between words and pronunciations are permitted. As mentioned above, the phone set must match that used in the acoustic models and the orthography must match that in the transcripts.

There are a few options for obtaining a pronunciation lexicon, outlined below. More details about several of these options are in the sections to come.

- Download the pronunciation lexicon from the MFA website
  - As of writing, there are dictionaries for English, French, and German
- Generate the pronunciation lexicon from the transcripts using a pretrained grapheme-to-phoneme (G2P) model

  - See section 4.3 on Running a G2P model

- Train a G2P model to then generate the pronunciation lexicon
- Create the pronunciation lexicon by hand using the same phone set as the acoustic models

4.2.4 Acoustic models

Pretrained acoustic models for several languages can be downloaded from the Montreal Forced Aligner website.

If you wish to train custom acoustic models on a speech corpus, this can be accomplished using the Kaldi ASR toolkit. A tutorial for training acoustic models can be found here.

4.3 Grapheme-to-phoneme models

If you need a lexicon for the words in your transcript, you might be able to generate one using a grapheme-to-phoneme model. Grapheme-to-phoneme models convert the orthographic representation of a language to its canonical phonetic form after having been trained on examples or conversion rules. Pretrained grapheme-to-phoneme (G2P) models can be found at the Montreal Forced Aligner website. Once you download the one you want, you can follow these instructions:

1. Place grapheme-to-phoneme model in `montreal-forced-aligner/pretrained_models` folder (they can technically go anywhere, but this structure keeps the files organized)
2. Create input and output folders
3. Place transcripts and wav files in input folder. At least in version 1.1, the wav files needed to be present in order to run the grapheme-to-phoneme conversion model on the transcripts
4. Run grapheme-to-phoneme model

```
cd path/to/montreal-forced-aligner/

bin/mfa_generate_dictionary /path/to/model/file.zip /path/to/corpus /path/to/save.txt
```

`bin/mfa_generate_dictionary` takes 3 arguments:

```
1. where is the grapheme-to-phoneme model?
2. where are the wav files and transcripts? (input folder)
3. where should the output go? (output text file)
```

Explicit example (make sure to remove backslashes):

```
cd /Users/Eleanor/montreal-forced-aligner

bin/mfa_generate_dictionary pretrained_models/mandarin_character_g2p.zip \
/Users/Eleanor/Desktop/align_input /Users/Eleanor/Desktop/mandarin_dict.txt
```

4.4 Running the aligner

1. Place acoustic models and dictionary in `montreal-forced-aligner/pretrained_models` folder (they can technically go anywhere, but this structure keeps the files organized)
2. Create input and output folders

3. Place TextGrids and wav files in input folder
4. Run Montreal Forced Aligner

Make sure to change the arguments of `bin/mfa_align`!

```
cd path/to/montreal-forced-aligner/

bin/mfa_align corpus_directory dictionary acoustic_model output_directory
```

`bin/mfa_align` takes 4 arguments:

```
1. where are the wav files and TextGrids? (input folder)
2. where is the dictionary?
3. where are the acoustic models? (you do need the .zip extension)
4. where should the output go? (output folder)
```

Explicit example (make sure to remove backslashes):

```
cd /Users/Eleanor/montreal-forced-aligner

bin/mfa_align /Users/Eleanor/Desktop/align_input pretrained_models/german_dictionary.txt \
pretrained_models/german.zip /Users/Eleanor/Desktop/align_output
```

4.5 Tips and tricks

Acoustic models

You do not need to unzip these. If you do, make sure to call the `.zip` version.

Wav files

I mentioned it above, and will mention it again. Things tend to go more smoothly when the wav file is already 16 kHz and a single channel.

TextGrids

- make sure TextGrid boundaries do not align with either the absolute start or end of the file
- make sure the final TextGrid boundary is at least ~20-50 ms away from the edge (if it still doesn't work, you might want to try increasing that interval)
- sometimes it helps to have an empty interval between every interval containing text
- sometimes it helps to increase the number of intervals present in the file so the aligner becomes less likely to derail

Chapter 5

Penn Forced Aligner (Legacy)

5.1 Overview

What does the Penn Forced Aligner do? The Penn Forced Aligner takes a sound file and the corresponding transcript of speech and returns a Praat TextGrid with phone and word alignments. You can find the website here.

What does it include? The system comes with *pre-trained acoustic models* of American English (AE) speech from the SCOTUS corpus, built with the HTK Speech Recognition Toolkit. The section on Kaldi introduces a similar, but alternative system to HTK, in case you'd like to train your own acoustic models from scratch. In addition to the acoustic models, the download also includes a large lexicon of words based on the CMU Pronouncing Dictionary. The dictionary contains over 100,000 words with their standard pronunciation transcriptions in Arpabet. Arpabet is a machine-readable phonetic alphabet of General American English with stress marked on the vowels. While I have, perhaps wrongly, used the Penn Forced Aligner to create phonetic alignments for other languages, those transcriptions had to be forced into Arpabet phones. For a quick alignment, this works fine, but please be advised: the acoustic models of the Penn Forced Aligner are trained on American English, so any speech it is given will be treated like American English. Unless manual adjustments follow the automatic alignment, the PFA alignment would not be ideal for most non-AE phonetic analyses. Again, see the section on Kaldi if you would like to create acoustic models for a different language or dialect.

5.2 Prerequisites

- HTK Toolkit Version 3.4
  - Version 3.4.1 will **not** work
  - Download here: http://htk.eng.cam.ac.uk/
  - Many run into issues installing this, but detailed installation instructions (and additional tutorial notes) can be found here. **Update** as of November 11, 2018: this website (linguisticmystic.com) no longer works, but covered how to install the Penn Forced Aligner on a Mac. The Penn Forced Aligner is no longer being maintained, and has instead been replaced by FAVE (section 3). The corresponding prerequisites for HTK installation on Mac are now covered on the FAVE wiki).
- Python 2.5 or 2.6
  - Pre-installed on Macs, but you can double-check by opening the terminal and typing python. This will return the version installed. I have been using Version 2.7.6 and have not had any issues running the aligner. Since you've typed python, the terminal is now running with the assumption of python syntax. You want to exit this to get back to its Unix base; to do this, type `exit()` or `quit()`. If you have an earlier version of Python, you may need to type `exit`.

IPR2026-00272
Krisp Technologies EX1037 Page 31

- SoX
  - Download here: http://sourceforge.net/projects/sox/

5.3 Modifying the lexicon

After downloading the Penn Forced Aligner, you should now have a directory named `p2fa`. Before beginning any of the following steps, take note of the parent directory and path, as you will need to direct your terminal to the exact `p2fa` location. For example, my `p2fa` directory is located in `/Users/Eleanor`. I can direct my terminal now to that location and view the contents of that directory by typing the following:

```
cd /Users/Eleanor/
ls
```

Once you've located the `p2fa` directory, you can find the lexicon in `p2fa/model/dict`.

Despite the lack of a `.txt` extension, `dict` is a text file, making it quite easy to add words and non-words alike. You'll want to make sure that all words in your transcript are indeed in the dictionary. This can be done by converting your transcript into a word list and comparing it against the dictionary.

First, you'll need a copy of your transcript in a text file called `fulltranscript.txt`. (Actually, you can call it whatever you want; just make sure to change the name in the code!)

In the terminal, navigate to the location of your transcript and then we can use the long code to create a list of all unique words in the transcript. My `fulltranscript.txt` is located in `/Users/Eleanor/myCorpus`. You'll need to change that part to match your file location.

```
cd /Users/Eleanor/myCorpus
tr ' ' '\n' < fulltranscript.txt | tr '[a-z]' '[A-Z]' | \
sed '/^$/d' | sed '/[.,?!;:]/d' | sort | uniq -c | sed 's/^ *//' | \
sort -r -n > fulltranscript_words.txt
```

The above clearly does a whole slew of functions. It will first take your transcript, separate each word with a new line (`tr ' ' '\n' < fulltranscript.txt`), capitalize all letters (`tr '[a-z]' '[A-Z]'`), delete blank lines (`sed '/^$/d'`), remove punctuation (`sed '/[.,?!;:]/d'`), sort the words (`sort`), remove duplicate words (`uniq -c`), delete blank lines/trailing white space again (`sed 's/^ *//'`), sort again and give you a count of how many times each word appears in the transcript (`sort -r -n > fulltranscript_words.txt`).

The following code will find the word pronunciations in the CMU dictionary. This is accomplished by taking the words and putting them into the regular expression format for locating the beginning of a line (`^`). This results in `tmp.txt`. That file is then compared against the CMU dictionary. If the word is in the dictionary, then the dictionary line is extracted such that you have both the word and its pronunciation. Note that when using the `cut` command, the default cut is tab (`cut -f 2`), but if the delimiter is anything other than tab (space, comma, etc.), it can be specified with `cut -d 'my delimiter' -f 2 myfile.txt`.

```
cut -d ' ' -f 2 fulltranscript_**words**.txt | sed 's/^/^/' | sed 's/$/  /' > tmp.txt
egrep --file=tmp.txt /Users/Eleanor/p2fa/model/dict > fulltranscript_words_pron.txt;
```

You then need to determine which words were skipped in this process, i.e., the words missing from the CMU dictionary. This can be done by comparing the final `fulltranscript_words_pron.txt` against the original `fulltranscript_words.txt`.

```
# extracts and sorts relevant columns and stores in tmp file
sort -k 2 fulltranscript_words.txt > tmp1.txt
sort -k 1 fulltranscript_words_pron.txt > tmp2.txt

# merges the two files to list words, word count, and pron
join -1 2 -2 1 tmp1.txt tmp2.txt > fulltranscript_words_pron2.txt
```

```
# lists words with missing pronunciations
# these are the words you need to add to the dictionary
join -v 1 -1 2 -2 1 tmp1.txt tmp2.txt > fulltranscript_words_missing_pron.txt
```

```
# Extras:
# get count of word types
wc -l < fulltranscript_words.txt

# get count of phone types
wc -l < fulltranscript_words_pron.txt

# get count of phone tokens
cut -d' ' -f 3- < fulltranscript_words_pron.txt | tr ' ' '\n' | sort | uniq -c | \
sed 's/^ *//' | sort -r -n > fulltranscript_phones.txt
```

Lexical entries need to be added in the same format as the rest of the dictionary, which is the word in all caps followed by two spaces and the CMU pronunciation with stress on the vowel. For example:

KLATT K L AE1 T ELEANOR EH1 L AH0 N AO2 R ELEANOR EH1 L AH0 N ER2

Multiple pronunciation variants are fine (this can even be used to test some interesting hypotheses). CMU provides a nice tool for converting standard spellings of words and non-words into the correct pronunciation. This can be found here: http://www.speech.cs.cmu.edu/tools/lextool.html.

After running this tool, you will need to add the stress value to the vowels (1 = primary, 0 = unstressed, 2 = secondary). You can always refer to similar words in the dictionary for an example.

5.4 Running the aligner

The standard implementation of the Penn Forced Aligner will process a single wav file and transcript, returning the phone and word alignment on a Praat TextGrid.

Before beginning this part of the tutorial, make sure that all the words in your transcript are in the included lexicon, `p2fa/model/dict`. If you're not sure or know that they are not, please visit Section 5.3.

Ingredients:
* Wav file of recording
* Corresponding transcript. Example below:

SAY TUTT AGAIN

SAY PAT AGAIN

SAY DOT AGAIN

The transcript should contain the words with spaces between them; these are standardly capitalized, but the aligner will accept lowercase and uppercase letters. Line breaks are fine, as are apostrophes, as long as that spelling is in the dictionary. All punctuation should be removed. Others have suggested adding an "`sp`" or space between words and sentences. This is not necessary. The aligner will determiner whether or not a small space or silence is present.

An important note is that the aligner can derail if there is untranscribed noise or speech in the recording. In my experience, it's not too bad at recovering after a short while, but it's best to avoid that situation. This can be accomplished by giving it smaller portions of the wav file, or ensuring that all noise and extraneous speech is explicitly transcribed. Smaller portions of the wav file can be created manually by extracting the relevant clip(s). Alternatively, a modified version of the script can process relevant sections of speech defined

by their start and end times. Yet another option is to transcribe noise as `{NS}` and silence as `{SP}`. I have not tried this method, so I do not know how robust it is to extraneous speech.

To process a single wav file and transcript, direct the terminal to the directory containing the Penn Forced Aligner `align.py` script with `cd`, then type the second command. The arguments to the align.py script are the locations of the wav file and transcript file. The script creates the aligned TextGrid as output (`subj01.TextGrid`, but you can call it whatever you want).

```
cd ~/p2fa
python align.py /Users/Eleanor/myCorpus/subj01.wav \
/Users/Eleanor/myCorpus/subj01.txt subj01.TextGrid
```

And that's it!

Chapter 6

AutoVOT

6.1 Overview

What does AutoVOT do? AutoVOT takes a sound file and Praat TextGrid marked with the locations of word-initial, prevocalic stop consonants and creates a new tier with boundaries marking the stop consonant burst release and following vocalic onset. With these boundaries, positive VOT can be measured efficiently using a standard acoustic analysis program such as Praat.

What does it include? AutoVOT can be used via Praat directly or via the command line. The system comes with pre-trained acoustic models/classifiers for American English and British English word-initial, prevocalic VOTs. It also offers the option to train your own acoustic models from labeled training data.

As the online tutorial is extremely helpful, I will just offer a few tips from experience. You will first need to visit the online tutorial for the prerequisites and basic installation.

Online tutorial and download: https://github.com/mlml/autovot/

While not mentioned in the online tutorial, another prerequisite is NumPy. This can be downloaded here: http://www.numpy.org/

*Note that this may have been updated since I last used it. The developer of AutoVOT had mentioned he may remove this dependency.

The recipe assumes that you have TextGrids containing the phone and word level transcriptions (e.g., Penn Forced Aligner output)! To run AutoVOT, the word-initial, prevocalic stop consonants need to be located and given a "window of analysis" in the TextGrid. The window of analysis is the interval surrounding the stop consonant that AutoVOT will process. The tutorial will cover one method for accomplishing this.

6.2 Recipe

1. Create a list of word-initial, prevocalic (CV) words in your transcript

There are many ways to accomplish this; the following is just one suggested way using some Unix and the P2FA dictionary. The following bash commands take multiple transcripts with the naming structure `S001.txt`, `S002.txt`, etc. as input, concatenates them, removes punctuation, puts each word on a new line, strips end of line characters, converts lowercase to uppercase, then removes duplicate words. The output of this is returned in the text file `fulltranscript_words.txt`. If you have a single transcript, you can replace `S*[0-9].txt` with the name of your transcript. Before running this code, you must direct the terminal (`cd`) to the directory housing your full transcript (`fulltranscript.txt`).

IPR2026-00272
Krisp Technologies EX1037 Page 35

```
cat S*[0-9].txt | tr -d '[:punct:]' | tr ' ' '\n' |
sed '/^$/d' | tr '[a-z]' '[A-Z]' | sort | uniq > transcript.txt
```

matchText.py

`matchText.py` takes as input the CMU Pronouncing Dictionary with a '.txt' extension. It identifies which of the words in your transcript begin with stop consonants in prevocalic position. These words are stored in the text file `wordList.txt`. You will need to modify the path locations and possibly the regular expression in `matchText.py`.

```
# run matchText.py
python matchText.py
```

2. Find start and end times for words on `wordList.txt` in TextGrids with phone- and word- level boundaries

[findWords.praat]"(https://www.eleanorchodroff.com/tutorial/scripts/findWords.praat){target="_blank"} `findWords.praat` takes as input `wordList.txt` and returns the start and end times of matching words in the audio file. This is stored in the text file `CVWordLocations.txt`. You will need to modify the source and destination paths in `findWords.praat`. Verify that the regular expression in `'Create Strings as file list...'` will work for your setup.

3. Create AutoVOT intervals on a new tier in your Praat TextGrid

makeAutoVOTTextGrids.praat

This script takes as input the text file `CVWordLocations.txt` and the Penn Forced Aligner TextGrids. It adds an interval tier 'vot' and renames the TextGrids to `filename_allauto.TextGrid`. Because all stop consonants are word-initial, the start of the word is assumed to be the start of the stop consonant. The end of the stop consonant is identified using the interval on the phone tier that aligns with the start of the word. Those boundaries are then used to create an AutoVOT check interval on the new vot tier. If the stop consonant begins with PTK, then those boundaries are extended 31 ms in both directions. If the stop consonant begins with BDG, then the boundaries are extended 11 ms in both directions. The extra, odd millisecond is to reduce the chance of placing a boundary where one already exists. Note that the P2FA boundaries can only be placed at 10 ms intervals, so adding time at a factor of 10 ms results in overlapping/identical boundaries. The interval text is then set with the phone name (PTKBDG) after all boundaries have been created. This ensures that overlapping intervals will not be a problem. The output should look something like this:

The different labels in the 'vot' tier will have some consequences when running AutoVOT. The AutoVOT analysis depends on a consistent label for the intervals it needs to check. Since there are six different interval labels [PTKBDG], AutoVOT will need to be run six different times. For me, this is preferable, as I then know exactly which stop consonant was measured. For some, however, this may be too tedious. In that case, I still highly recommend running the voiced and voiceless stop consonants separately.

4. Make a list of your TextGrids and wav files and move them the lists to `autovot/experiments/config`

The code below will generate a textfile located in the correct autovot folder that contains a list of the TextGrids with their path (`PWD` is the command that prints this). When generating these lists, the terminal must be in the directory that contains the TextGrids and wav files, respectively.

```
cd experiment/myTextGrids
ls -d -1 $PWD/*_vot.TextGrid > ~/autovot/experiments/config/ListTextGrids.txt

cd experiment/myWavFiles
ls -d -1 $PWD/*.wav > ~/autovot/experiments/config/ListWavFiles.txt
```

5. Run AutoVOT from the terminal on **one** stop consonant

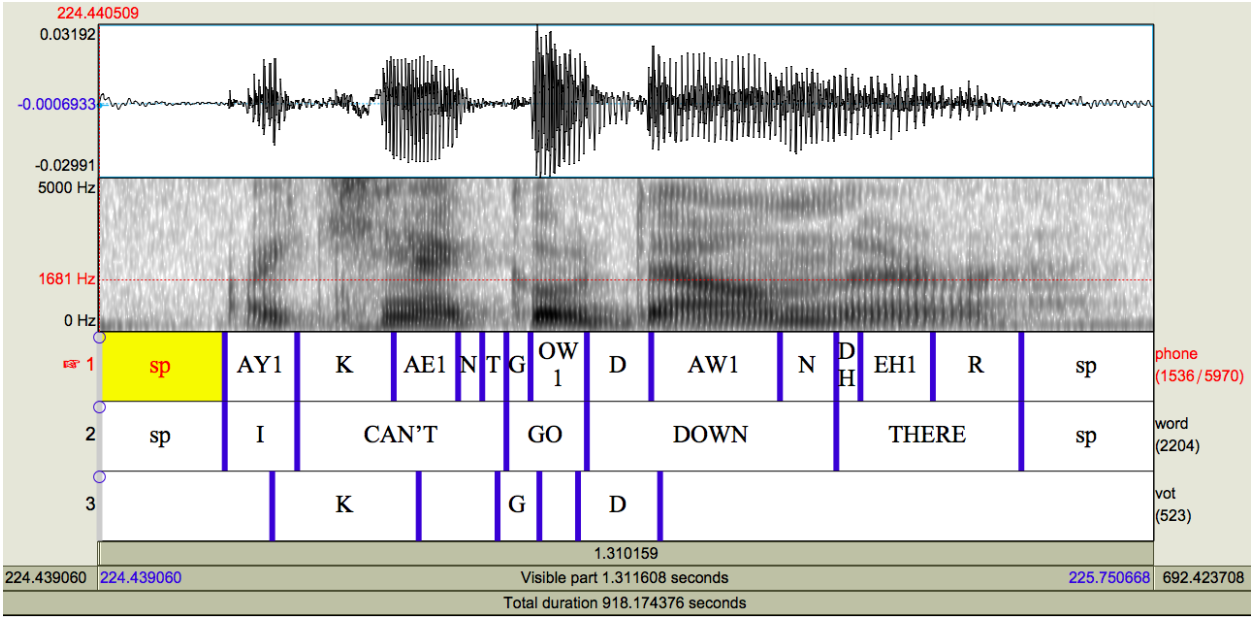

Figure 6.1: Example TextGrid for AutoVOT

Modify the arguments to `auto_vot_decode.py`: `window_mark` should be set to the stop consonant to be analyzed. We recommend setting `-min_vot_length` to 4 (ms) for voiced stops and 15 (ms) for voiceless stops. After each stop consonant, there is a post-processing step. Make sure to do that to avoid overwriting the AutoVOT output (see step 6)!

Example code for a voiceless stop:

```
cd experiments
export PATH=$PATH:/Users/Eleanor/autovot/autovot/bin

auto_vot_decode.py --window_tier vot --window_mark P --min_vot_length 15 \
config/ListWavFiles.txt config/ListTextGrids.txt \
/Users/Eleanor/autovot/autovot/bin/models/vot_predictor.amanda.max_num_instances_1000.model
```

Example code for a voiced stop:

```
cd experiments
export PATH=$PATH:/Users/Eleanor/autovot/autovot/bin

auto_vot_decode.py --window_tier vot --window_mark B --min_vot_length 4 \
config/ListWavFiles.txt config/ListTextGrids.txt \
/Users/Eleanor/autovot/autovot/bin/models/vot_predictor.amanda.max_num_instances_1000.model
```

The path (`export PATH=$PATH` command) should always be set from the `experiments` directory before running AutoVOT. I recommend the above argument specification, but the structure can also be modified to suit your particular dataset. The arguments in the AutoVOT decode command are listed here:

- –window_tier
  - This refers to the TextGrid tier that contains the intervals to check (or windows of analysis). The current procedure has called this tier 'vot'
- –window_mark
  - This refers to the label of the interval to check. The current procedure has six different labels [PTKBDG], so this command will need to be run six times, once for each of these labels. After each run, the output tier will need to be renamed so that it is not overwritten.

- –min_vot_length
  - This refers to the minimum allowed VOT length. I would recommend 15ms for voiceless stops and 4ms for voiced stops, but this can be modified. It should be noted, however, that performance degrades on the voiceless stop measurements if the minimum VOT is too low. (This is why I recommend running AutoVOT separately for the voiced and voiceless stops.)
- Path from experiments to the list of wav files
- Path from experiments to the list of TextGrids
- Path to AutoVOT classifier
  - The default classifier is the one named `amanda`, but there are a few others you can try. While the `amanda` classifier is hidden to the user, the others are located in the `autovot/bin/models` folder. Alternatively, AutoVOT gives you the option to train your own classifier on labeled data. For more information on training, please visit their main website: https://github.com/mlml/autovot/.

6. Rename AutoVOT output tier

autoVOTpostproc.praat

After each stop consonant is processed, run `autoVOTpostproc.praat` to rename the AutoVOT output tier. Otherwise, AutoVOT will overwrite your previous work.

Make sure to change the phone name in the script.

Return to step 5 and repeat until all 6 stop consonants have been processed.You will need to modify the path to the TextGrids, the tier labels, and the new tier name. Once you have completed this, the cycle starts over until you have all six stops.

7. Move all AutoVOT boundaries to one tier

resetBoundaries_stops.praat

After running AutoVOT, you should now have a TextGrid with 6 different output tiers: one for each stop consonant. These tiers can be collapsed into one tier with resetBoundaries_stops.praat. You will need to modify the path directory. In addition, if your AutoVOT output does not occupy tiers 3-8, you will need to modify the tier numbers in the script.

The final product should look like this:

This resembles the previous picture of the TextGrid, but note that the boundaries on the autovot tier are now located at the burst and vocalic onset in the signal.

****If you have manually placed/corrected boundaries, continue on. Otherwise, skip to step 11!****

8. Stack TextGrids with manual boundaries and the AutoVOT boundaries

stackTextGrids.praat

This script takes as input TextGrids with manual boundaries (we have two different types) and the AutoVOT TextGrids (`_stops.TextGrid`). It places all the tiers together in one TextGrid and renames the file `_stacked.TextGrid`.

We have two different TextGrids with manual boundaries in them: `_autovot.TextGrid` and `_check.TextGrid`. Using the `_autovot` TextGrids was meant to eliminate bias from seeing the AutoVOT output. Those TextGrids only contain the window of analysis and not the final measurement. On the other hand, manual adjustments on the `_check` TextGrids were made directly to the AutoVOT output. This was for efficiency. Only the boundaries on the _autovot files were used for comparison to the AutoVOT output.

****If you want to compare AutoVOT and manual measurements, complete step 9; otherwise, continue on to step 10.****

9. Compare manual and automatic boundaries

measureVOT.praat

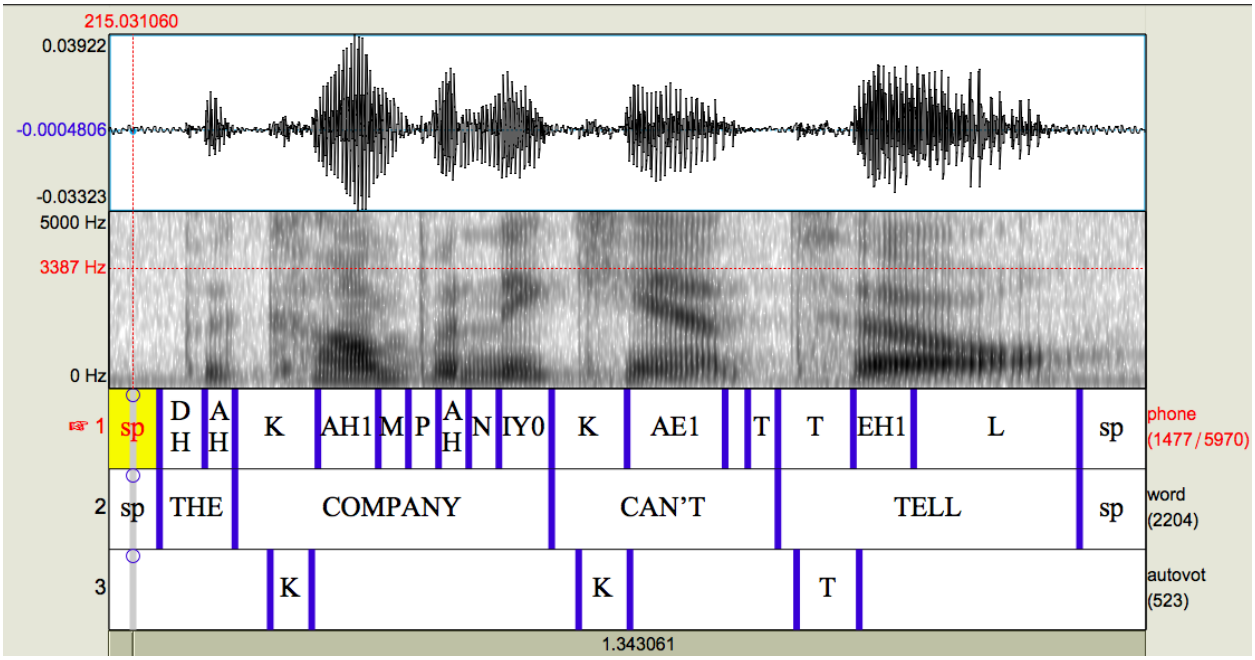

Figure 6.2: Example output TextGrid from AutoVOT

This script takes as input the `_stacked.TextGrid` and creates the file `manualVOTs.txt`.

10. Replace automatic boundaries with manual ones where available

resetBoundaries_stacked.praat

This script takes as input the `_stacked.TextGrids` and creates as output `_stacked2.TextGrid`.

11. Measure VOT and sentence rate

cueAnalysis_new.praat

This script measures the duration of each burst (positive VOT, if you will), following vowel, and word. It also measures the speaking rate, defined as the average word duration per sentence. The speaking rate component relies on there being two 'sp' or silent intervals between each sentence. If your data does not meet this criterion, you can modify the script to fit your data or simply comment out the speaking rate measurement. You will need to modify the path to the wav files and TextGrids.

Chapter 7

Other resources

There are many other freely available phonetics tools and resources. I've listed just a few of them here. If you know of others that should be added, please let me know!

- Montreal Forced Aligner
- Montreal Corpus Tools
- Prosodylab Aligner
- VoiceSauce
- ProsodyPro
- SPAAT
- FAVE
- Phonological CorpusTools
- Korean Phonetic Aligner
- Munich AUtomatic Segmentation System (MAUS)
- Phon
- World Phonotactics Database
- UCLA Database of Sounds
- Praat scripts by Matt Winn
- Praat scripts by Holger Mitterer
- Praat tutorial by Will Styler

IPR2026-00272
Krisp Technologies EX1037 Page 41