Duke University

ECE Independent Study

Automatic Speech Recognition using the Kaldi Toolkit

Madeline Briere

supervised by
Dr. Michael Gustafson

February 24, 2018

Contents

1 Abstract

This project explores the current technology available for *Automatic Speech Recognition* (ASR), the process of converting speech from a recorded audio signal to text [11]. The primary goal is to identify a toolkit for use in the construction of a personal assistant system, similar to Amazon's Alexa, but with a smaller and more targeted lexicon meant to increase accuracy. In particular, we explore the Kaldi Speech Recognition Toolkit, written in C++ and licensed under the Apache License v2.0, developed for use by speech recognition researchers [17]. This toolkit was chosen on the grounds of extensibility, minimal restrictive licensing, thorough documentation (including example scripts), and complete speech recognition system recipes. In this project, we explore the ASR process used in Kaldi (including feature extraction, GMMs, decoding graphs, etc.). With this foundation, we walk through three extensions of the Kaldi toolkit: (1) the **Digits** example, using 1500 audio recordings of the digits 0-9, (2) the **VoxForge** example[3] and (3) the **CMU AN4** alphanumeric example[2]. This project demonstrates that Kaldi can be extended in simple and complex situations and is flexible and easy to use in development. Given the results of this analysis, we conclude that Kaldi is a viable choice for future extension.

2 Project Goals

The goal of this project is to develop a prototype system for Automatic Speech Recognition (ASR; the process of converting speech from a recorded audio signal to text [11]) satisfying the following requirements:

1. Easy to develop and extend
2. Lightweight and minimal
3. Accurate and fast (less than 10 second wait time)
4. Maintains a balance between a homegrown and outsourced system

First and foremost, this system must be easy to develop and extend. If we cannot work with the system (because it is incredibly esoteric and/or not well-documented), it is virtually useless. Likewise, if we cannot extend the system with custom data (which is integral to our system design), the system will not work for our purposes.

In addition, this product must be a lightweight and minimal ASR system – we need to maximize accuracy for a small state space of input options (geared towards the client), using a small device and streamlined system. The user should be able to request something within a small state-space of options and receive feedback accordingly with high accuracy. For instance, if the client were using such a system in a car shop, requests such as "purchase a Chevy Malibu A/C Compressor" should be parsed and carried out (e.g., through Amazon's marketplace). We must minimize run time so that the system can be used without inconvenience (for instance, a response time over 10 seconds would be non-viable).

Additionally, the resulting ASR system must maintain a delicate balance between the two ends of the system development spectrum:

1. **A homegrown system**: A system developed from scratch, using other resources only minimally

   Pros: Ownership, intimate understanding of system

   Cons: Less well-tested (more buggy), less extensive functionality

2. **A borrowed system**: Bootstrapping another system to produce the desired functionality

   Pros: Well-tested, more extensive functionality

   Cons: Potential legality issues, lack of ownership

Given all of these requirements, we move forward with our project in search of a usable toolkit. In the previous part of this exploration, we looked into the signal processing side of the system. We broke down the process of Mel-Frequency Cepstral Coefficients (MFCCs) feature extraction and confirmed the viability of this type of processing for ASR [8]. Instead of breaking down all of the mathematical steps as we did before, we now seek a high-level understanding of a full-process system.

We explore the Kaldi Speech Recognition Toolkit [17], a well-documented ASR toolkit written in C++ that uses MFCCs for feature extraction. This system deals with the entire ASR process, from WAV file to text transcription. This toolkit seems the perfect solution to the homegrown vs. outsourced debate. Hence, we hope to come away from the exploration with an early-stage extension of Kaldi that is viable for use in the aforementioned product (i.e., satisfying the stated requirements). Of course, this will still require knowledge of the system, as development in Kaldi is largely the authorship of scripts carrying out the stages of speech recognition.

In order to completely explore Kaldi, we hope to do the following:

1. Outline the layout of Kaldi

    Installation

    Organization

    Sub-components of Kaldi

    Data preparation (using custom data)

    Decoding the results

2. Walk through several examples using the Kaldi Toolkit

    Introductory example: Using 1500 audio files of the digits 0-9

    Advanced example: Using VoxForge dataset/acoustic model (training on more complex data) [3]

    Additional example: Using CMU AN4 census data (to recognize alphanumeric queries) [2]

The final result should be a well-rounded understanding of the Kaldi system.

3 Background

3.1 What is Automatic Speech Recognition?

Automatic Speech Recognition (ASR) is "the process of converting speech from a recorded audio signal to text" [11]. The particular type of ASR we are interested in is the *personal assistant* ASR system. These types of systems are seen across households today, in products like Amazon's Alexa, and must be able to respond quickly, accurately and helpfully to a user. We are interested in the "understanding" component of this system – the part of the assistant that "understands" what the user is saying (by translating the query from speech to text) and searching its resources for a response.

The typical model for ASR can be found in Figure 1. We start with an audio waveform and extract a series of "features," representations of small frames of the speech function. These features, along with a pronunciation dictionary to match features to phones, can be used to generate acoustic models (the likelihood of an observed acoustic signal given a word sequence). The likelihood of an observed word sequence is derived from a language model.

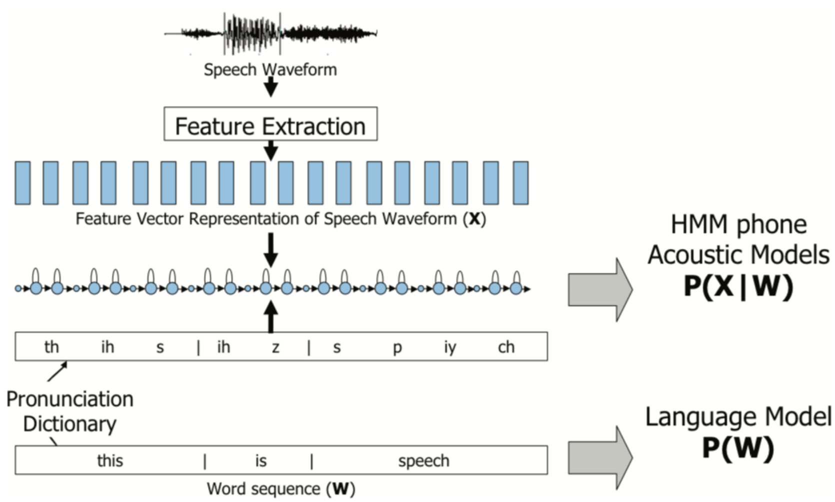

Figure 1: Typical process of Automatic Speech Recognition [12]

Automatic Speech Recognition is a complicated process and will not be completely outlined in this paper. Rather, we will explore the steps involved in interacting with an ASR system like Kaldi as a client. Should you be interested further in the theory, see the prior paper in this series [8] or other papers devoted to the theory of Automatic Speech Recognition (such as *Automatic Speech Recognition* by Gruhn et. al [11]).

3.2 What is Kaldi?

The Kaldi Speech Recognition Toolkit is a toolkit for speech recognition written in C++ and licensed under the Apache License v2.0. It is intended for use by speech recognition researchers. At its inception in 2009, this toolkit was designed for "Low Development Cost, High Quality Speech Recognition." Its founders felt that "a well-engineered, modern, general-purpose speech toolkit with an open license would be an asset to the speech-recognition community" [17]. Since its initial release, Kaldi has been maintained and developed largely by Daniel Povey (Researcher at Microsoft and John Hopkins University).

4 Kaldi: Automatic Speech Recognition Toolkit

4.1 Kaldi Layout

The general layout of the Kaldi Toolkit is displayed in Figure 2. It accepts a set of customizable audio data as input, along with accompanying language and acoustic data (see the *Data Preparation* section).

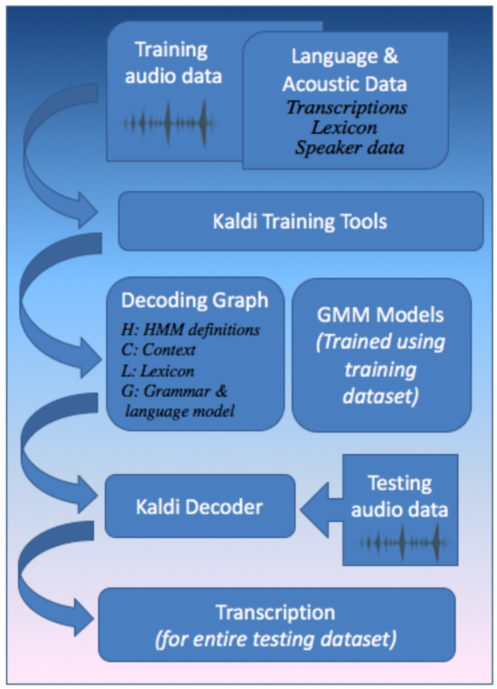

Figure 2: Layout of Kaldi Toolkit (based on NTNU diagram and Kaldi documentation) [17] [9]. Note that this diagram is hugely simplifying – optimizations and adjustments (e.g., using alignments) are not shown.

We may note that the input data is used to generate two main Kaldi components, the *decoding graph* and *final acoustic GMM*.

4.1.1 Decoding Graph

The first central element is a decoding graph (of the HCLG format; see Fig. 2). The **H** represents the Hidden Markov Model (HMM) structure, where an HMM is used to model a Markov Process (a stochastic process satisfying the Markov property of "memorylessness"). In this case, the structure map states to phonemes. The **C** represents contextual information about the phones (i.e., the articulation of a phone may change given surrounding phones). The **L** represents the lexicon, which maps each possible word to a set or several sets of phones.

Finally, the **G** represents the language model (or grammar) which estimates the probability of a given word sequence. Together, these components form a decoding graph which can be used to match a given input vector to a resulting transcription. The decoding graph for our Digits example, for instance, might look something like the network shown in Figure 3.

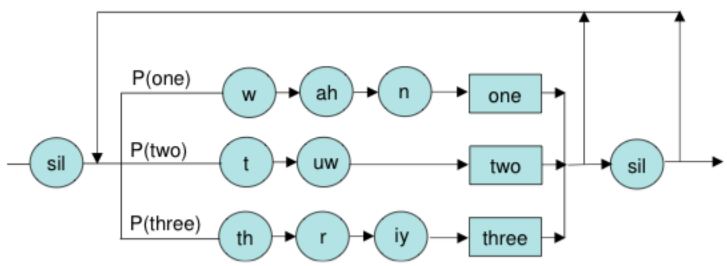

Figure 3: An example decoding graph with the words "one," "two," and "three" in the lexicon [12]

4.1.2 Acoustic GMMs

The second element is a final Gaussian Mixture Model (GMM). A GMM is a probabilistic model, in this case used to represent an acoustic output. Our final result in this process will be a series of GMMs matching to each state in our decoding network. Mapping the HMM structure to this GMM structure is done in the *run* script of each example [17].

It can be noted that we will primarily observe two types of GMM training: triphone and monophone. The first uses contextual information while the latter does not [10].

4.2 Decoding

Together, these pieces (*HCLG.fst* and *final.mdl*) can be fed into the decoder, along with testing features to produce transcriptions [17]. During the *run* process, the system will generate a series of transcriptions, documented in the decoding logs, which can be compared to the expected results manually or via the generated word error rate files.

4.3 Reader Caveat

As users of Kaldi, rather than true developers of Kaldi, we will focus on the start and end points of this flow, rather than the mechanics of the Kaldi training algorithms (if you have background in GMMs, decoding graphs, etc., the Kaldi documentation may be of interest to you [17]). Of primary interest to us are the customizable input (discussed in *Data Preparation*) and the decoding results.

4.4 Organization

The relevant Kaldi directories are organized in the following fashion:

1. *egs*: A series of example scripts allowing you to quickly build ASR systems for over 30 popular speech corpora (documentation is attached for each project)
2. *misc*: Additional tools and supplies, not needed for proper Kaldi functionality
3. *src*: Kaldi source code
4. *tools*: Useful components and external tools

We will be working in the *egs* folder, where all of the Kaldi extensions are housed. We will also use some of the scripts in the *tools* folder, which help with installation.

4.5 Installation

Kaldi is housed on Github, so installation is as easy as cloning the project, using the below command:

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

To retrieve any new updates, users need only pull from this repo and refresh their project.

Actually running Kaldi will require building the project – this can be accomplished by following the README instructions and using the relevant Makefiles.

4.6 Data Preparation

Data preparation is the most relevant component of the Kaldi layout to this analysis. Because we seek to feed in customized data, we must understand the requirements of the system.

In each extension, we have to define:

1. Audio data (training and testing)

2. Acoustic data

   spk2gender: [speakerID] [gender]

   wav.scp: [utteranceID] [file_path]

   text: [utteranceID] [transcription]

   utt2spk: [utteranceID] [speakerID]

   corpus.txt: [transcription]

3. Language data

   lexicon.txt: [word] [phone(s)]

   nonsilence_phones.txt: [phone]

   silence_phones.txt: [phone]

   optional_silence.txt: [phone]

4. (Optional) Configuration

5. (Optional) Language model toolkit

We will see in the examples how such files may be manually or automatically generated.

5 Initial Assessment of Kaldi

An initial assessment of Kaldi (see Figure 4) reveals it to be a viable system for the desired product. Kaldi includes a variety of utility scripts, including functionalities such as feature extraction, data preparation, transition modeling, construction of decoding graphs, and acoustic modelling. Extensions of Kaldi can incorporate custom training and testing data and use the corresponding lexicon. These extensions can still utilize the provided scripts, substituting in various decoding types, language models, etc.

Requirements	✓	✗
Easy to use	- Well-documented - Has extensive support system (Git, Kaldi homepage, help pages) - Many examples (including VoxForge, AMI, and Fisher)	- Requires knowledge of shell coding - Not initially designed for “casual use” (meant to be used by full-time speech recognition researchers)[2]
Extensible	- Can reasonably build off of examples - Built specifically for extension with new datasets/models	- Complex extension requires intimate knowledge of Kaldi system - Commands change frequently
Partly homegrown	- Extensions possible through customized scripting	- Customization leaves room for suboptimal configurations - Potentially buggy
Partly outsourced	- Extensive toolkit for feature extraction, decoding, etc. - Open license (limited legality concerns)	- Less intimate knowledge of system

Figure 4: Assessing the viability of Kaldi (note that speed was not considered in this analysis) [17] [9].

6 Digits Example

6.1 Introduction

The goal for this example is to develop a simple ASR system using the Kaldi toolkit with a small, targeted dataset (about 1500 audio files). We hope to explore some potential issues and the general steps involved in the creation of a personalized ASR system.

In this example, we will use a series of audio files from various speakers, each containing an individual spoken digit from 0 to 9. Note that, in this example, a word is equivalent to a sentence and there is no sentence context (with only one word per file). This corpora is composed of several trials per speaker/digit. The goal is to train the system to recognize new audio files in which the speaker says a single digit from 0 to 9.

6.2 Resources

The tutorial in this example is based upon the "Kaldi for Dummies Tutorial" on the Kaldi site [17]. Our example goes slightly further in depth in some regions (especially the script results) and explores potential issues in the process.

This example requires audio data and, for the sake of time, we outsource the task of collection by using the audio files from the free-spoken-digit-dataset Github repository[13]. The audio files in this repository are collected from three males (Theo, Jackson, and Nicolas), where each individual speaks a single digit per WAV file. Each speaker records 50 files per each digit (0-9), producing 1500 audio files.

6.3 Preparing Audio Data

Audio samples were retrieved from the free-spoken-digit-dataset above, but it must be noted that there were certainly some issues with the given dataset in terms of incorporation into the Kaldi system.

Firstly, the data must be named in the fashion: speaker_digit_iteration.wav. This is done for sorting purposes – sorting by speaker id ends up being much more useful than sorting by digit. The data files in Jakobovski's repository currently have the format digit_speaker_iteration.wav, so this format must be changed with a simple bash script that swaps the first element with the second element. Following this renaming process, we have to sort the audio files into speaker folders. This is accomplished using the *sort.sh* script in the Appendix. The resulting speaker folders must be placed in the data/test or data/train folders.

The next step is to generate the acoustic data. Luckily, because we have so few speakers and a very clear state-space of audio transcriptions, this data can be generated using a bash script (*acoustic.sh* in the Appendix). This script:

1. Organizes the data folder
2. Generates the *spk2gender* file for the test and train folders
3. Generates the *wav.scp* file for the test and train folders, matching utterance IDs to full paths in the directory
4. Generates the *text* file for the test and train folders, matching utterance ID to a transcription
5. Generates the *utt2spk* file for the test and train folders, matching utterance ID to speaker
6. Generates the corpus, *corpus.txt* (all possible text transcriptions in the ASR system)

By supplying the audio files and its accompanying acoustic data, we give the system a way to map new audio files to text transcriptions, given this particular system.

6.4 Language Data

The language data for this example can be manually entered. The lexicon (shown below) is a phonetic transcription of every possible word in the lexicon.

```
!SIL sil
<UNK> spn
eight ey t
five f ay v
four f ao r
nine n ay n
one hh w ah n
one w ah n
seven s eh v ah n
six s ih k s
three th r iy
two t uw
zero z ih r ow
zero z iy r ow
```

The non-silence phones are those phones used that are not categorized as silence phones.

```
ah
ao
ay
```

```
eh
ey
f
hh
ih
iy
k
n
ow
r
s
t
th
uw
w
v
z
```

The list of silence phones, used to represent silence or unknown sounds, is a short one (shown below):

```
sil
spn
```

In the optional silence phones text file, we just put *sil*.

6.5 SRI Language Model (SRILM)

This particular example uses the SRI Language Model (SRILM) Toolkit [5]. SRILM is “a toolkit for building and applying statistical language models (LMs), primarily for use in speech recognition, statistical tagging and segmentation, and machine translation” [5]. Luckily, Kaldi has an *install_srilm.sh* file in the extras folder, which can be run to bypass manual SRILM installation.

6.6 A Note on Sampling Rates

If you choose to use the same data as this example, you may have to re-sample the audio files. The language model used by SRILM in this example expects a 16kHz sampling rate, while the digit audio files are sampled at 8kHz. You can change the SRILM modeling sample rate, or you can re-sample the audio files with a script. See the Appendix for *resample.m*, a simple MATLAB script that re-samples the entire folder of digit audio files using piece-wise cubic hermite interpolation. As a note: It is important to be careful about resampling. Inserting a buffer “0” in between every data point in the audio, for instance, would allow the program to run, but would create interference around the Nyquist frequency and potentially produce erroneous results. Another option for re-sampling is SoX, the “Swiss Army Knife of Audio Manipulation,” to re-sample the audio in the command line [6].

6.7 The "Run" Script

The *run.sh* script in each Kaldi example is used to execute all steps of the process, including data preparation, feature extraction, training and decoding. The script for *Digits* is relatively simple, and shows the general Kaldi process. Let's take a look at the general outline below (note that the unabridged version can be found in the appendix):

```
# General organizational preparation beforehand (not included)

echo "===== PREPARING ACOUSTIC DATA ====="

# Needs to be prepared by hand (or using self written scripts):
#
# spk2gender  [<speaker-id> <gender>]
# wav.scp     [<uterranceID> <full_path_to_audio_file>]
# text           [<uterranceID> <text_transcription>]
# utt2spk     [<uterranceID> <speakerID>]
# corpus.txt  [<text_transcription>]

# Making spk2utt files
utils/utt2spk_to_spk2utt.pl data/train/utt2spk > data/train/spk2utt
utils/utt2spk_to_spk2utt.pl data/test/utt2spk > data/test/spk2utt

echo "===== FEATURES EXTRACTION ====="

# Making feats.scp files
mfccdir=mfcc
steps/make_mfcc.sh --nj $nj --cmd "$train_cmd" data/train ...
    exp/make_mfcc/train $mfccdir
steps/make_mfcc.sh --nj $nj --cmd "$train_cmd" data/test ...
    exp/make_mfcc/test $mfccdir

# Making cmvn.scp files
steps/compute_cmvn_stats.sh data/train exp/make_mfcc/train $mfccdir
steps/compute_cmvn_stats.sh data/test exp/make_mfcc/test $mfccdir

echo "===== PREPARING LANGUAGE DATA ====="

# Needs to be prepared by hand (or using self written scripts):
#
# lexicon.txt           [<word> <phone 1> <phone 2> ...]
# nonsilence_phones.txt    [<phone>]
# silence_phones.txt    [<phone>]
# optional_silence.txt  [<phone>]

# Preparing language data
utils/prepare_lang.sh data/local/dict "<UNK>" data/local/lang ...
    data/lang

echo "===== LANGUAGE MODEL CREATION ====="
echo "===== MAKING lm.arpa ====="

#Check that SRILM installed excluded

```

```
local=data/local
mkdir $local/tmp
ngram-count -order $lm_order -write-vocab ...
     $local/tmp/vocab-full.txt -wbdiscount -text ...
     $local/corpus.txt -lm $local/tmp/lm.arpa

echo "===== MAKING G.fst ====="

lang=data/lang
arpa2fst --disambig-symbol=#0 ...
     --read-symbol-table=$lang/words.txt $local/tmp/lm.arpa ...
     $lang/G.fst

echo "===== MONO TRAINING ====="

steps/train_mono.sh --nj $nj --cmd "$train_cmd" data/train ...
     data/lang exp/mono  || exit 1

echo "===== MONO DECODING ====="

utils/mkgraph.sh --mono data/lang exp/mono exp/mono/graph || ...
     exit 1
steps/decode.sh --config conf/decode.config --nj $nj --cmd ...
     "$decode_cmd" exp/mono/graph data/test exp/mono/decode

echo "===== MONO ALIGNMENT ====="

steps/align_si.sh --nj $nj --cmd "$train_cmd" data/train ...
     data/lang exp/mono exp/mono_ali || exit 1

echo "===== TRI1 (first triphone pass) TRAINING ====="

steps/train_Δs.sh --cmd "$train_cmd" 2000 11000 data/train ...
     data/lang exp/mono_ali exp/tri1 || exit 1

echo
echo "===== TRI1 (first triphone pass) DECODING ====="
echo

utils/mkgraph.sh data/lang exp/tri1 exp/tri1/graph || exit 1
steps/decode.sh --config conf/decode.config --nj $nj --cmd ...
     "$decode_cmd" exp/tri1/graph data/test exp/tri1/decode

echo "===== run.sh script is finished ====="
```

This process can be broken down into a series of steps, starting at data preparation and continuing to training and decoding:

1. **Preparing acoustic data** (using the audio files)
2. **MFCC feature extraction** using train and test data
3. **Preparing language data** (relating to the possible phones seen and the breakdown of words into phones)
4. **Language model creation** (here, using SRILM)

Making *lm.arpa* (the language model, as an ARPA file[1])

Making *G.fst* (converted from *lm.arpa* to a FST file[2])

5. Monophone Speech Recognition: *does not* include any contextual information about the preceding or following phone [10]

   Training

   Decoding

   Alignment

6. Triphone Speech Recognition: *does* include any contextual information about the preceding or following phone

   Training (first pass)

   Decoding (first pass)

We can see a sample output in the Appendix under *Digits run.sh Output*. It is too long to include here.

6.8 Interpreting Script Results

6.8.1 Decoding Logs

One easy way to observe the script's functionality is to look at the decoding logs generated via the script. In the logs, we can see the utterance ID paired to the predicted transcription (seen in Figure 5). In our example log, we can see successful transcriptions (in green) and failed transcription (in red).

6.8.2 Word Error Rates

Another way to assess the script results is to look at the resulting *Word Error Rates*. During the monophone and triphone decoding phases, the script generates a series of *Word Error Rates* (WER). The WER is used to measure the accuracy of the ASR system. The WER is calculated as the minimum edit distance between the output of the ASR system and the reference transcriptions. The relevant edit operations are substitution, deletion and insertion [16]. The expression for WER is shown below in Equation 1.

$$WER = 100 * \frac{min\_dist(decoded(a), t, edit\_op = sub, del, ins)}{num\_words(t)} \tag{1}$$

Because WER is an error-based measurement, the ideal WER would be 0 – indicating no deviation between the ASR output and the reference transcription. We can see the WER in action by altering the input of our script slightly.

[1] An ARPA file uses log probabilities to convey phrase probabilities [14]

[2] An FST file is a binary representation of the finite state transducer/acceptor [20]

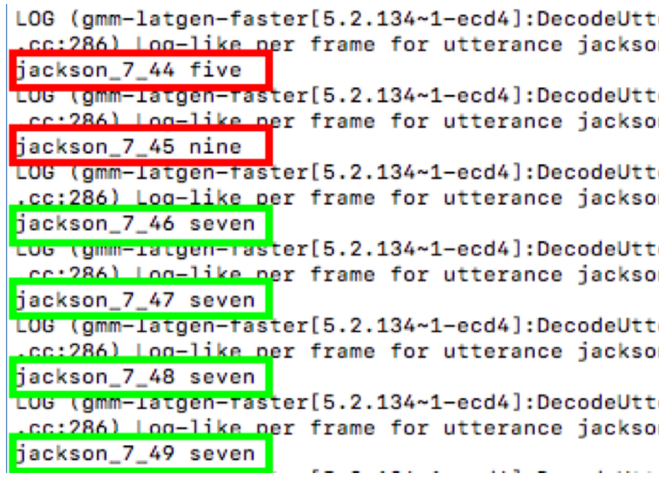

Figure 5: A sample log from the digits example, showing transcriptions for several audio files of the number "7"

In Table 1, we see the WER results given completely overlapping equivalent test and train data. We see minimal WER (.40 percent for monophone, .27 percent for triphone). This is because our system has been trained to handle the test input. We expect a very low error rate for this case.

Meanwhile, in Table 4, we see the results given non-overlapping test and train data (using Theo and Nicolas for training and Jackson for testing). The WER is now much higher (between 7.40 ad 10.80 percent across the monophone and triphone training), indicating a much larger number of deviations. This is because the system has not yet seen Jackson's audio data, and must determine the output based only on the data it has seen before (Theo and Nicolas). It can also be noted that the triphone results are not necessarily better than the monophone results in this case because the words used (e.g., "one", "two", "three") don't have any real context in the audio files. Hence, using contextual information doesn't improve the decoding WER.

Table 1: Results with equivalent test and train data

Table 2: Monophone Training

WER	Percent	Ratio
WER_7	0.40	6/1500
WER_8	0.40	6/1500
WER_9	0.40	6/1500
WER_10	0.40	6/1500
WER_11	0.40	6/1500
WER_12	0.40	6/1500
WER_13	0.40	6/1500
WER_14	0.40	6/1500
WER_15	0.40	6/1500
WER_16	0.40	6/1500
WER_17	0.40	6/1500

Table 3: Triphone Training

WER	Percent	Ratio
WER_7	0.27	4/1500
WER_8	0.27	4/1500
WER_9	0.27	4/1500
WER_10	0.27	4/1500
WER_11	0.27	4/1500
WER_12	0.27	4/1500
WER_13	0.27	4/1500
WER_14	0.27	4/1500
WER_15	0.27	4/1500
WER_16	0.27	4/1500
WER_17	0.27	4/1500

Table 4: Results with non-overlapping train and test data

Table 5: Monophone Training

WER	Percent	Ratio
WER_7	7.40	37/500
WER_8	7.40	37/500
WER_9	7.20	36/500
WER_10	7.60	38/500
WER_11	8.00	40/500
WER_12	8.40	42/500
WER_13	8.40	42/500
WER_14	9.20	46/500
WER_15	9.80	49/500
WER_16	10.20	51/500
WER_17	10.80	54/500
Average	8.58	43/500

Table 6: Triphone Training

WER	Percent	Ratio
WER_7	9.60	48/500
WER_8	9.40	47/500
WER_9	9.40	47/500
WER_10	9.40	47/500
WER_11	9.60	48/500
WER_12	9.40	47/500
WER_13	9.00	45/500
WER_14	9.40	47/500
WER_15	8.80	44/500
WER_16	8.40	42/500
WER_17	8.40	42/500
Average	9.19	46/500

7 VoxForge Example

7.1 Introduction

7.1.1 What is VoxForge?

VoxForge is an open source acoustic model (including a huge speech corpus), initially set up to collect transcribed speech for use with Free and Open Source Speech Recognition Engines (on Linux, Windows and Mac) [3]. VoxForge has similar aims to Kaldi in that it seeks to provide acoustic models and transcribed audio data without restriction, in order to contribute to current speech recognition engines.

7.1.2 VoxForge Dataset

Unlike our simple example using single digits, VoxForge utilizes a more difficult dataset, as characterized by the following features:

1. More complicated syntax, grammar, and lexicon
2. Longer transcriptions per audio file (a paragraph vs. a single word)
3. Massive amounts of total audio data (around 75 hours of speech)
4. Submitted from varied sources, creating more diversity in tone, volume, dialect, etc. and increased potential for errors

Luckily, we do not have to generate or format this dataset ourselves (as we did before).

7.2 Dependencies

The VoxForge Kaldi example has several dependencies which must be installed before executing the *run.sh* script. These can be found in the prep script below:

```
sudo apt-get install flac
sudo apt-get install python-dev
sudo apt-get install swig
sudo apt-get install pip
pip install numpy
extras/install_sequitur.sh
```

The *run.sh* will fail without any of these libraries.

7.3 (Optional) Memory Considerations

It should be noted that the VoxForge dataset in its entirety takes up 25GB of space. If you have enough space on your machine, you may skip this section. If you are working with limited space (on a virtual machine, for example, as will be explored in this section), this exploration may be useful.

This particular study was undertaken using a virtual machine with only 2GB of base memory (slow, indeed). In order to undertake some of the more complex examples, it was necessary to mount additional storage in the VM.

Duke University allocates a CIFS home directory space for each student, so it was possible to mount this directory space to the VM without having to alter the VM or create room for additional storage. The steps to do so looked something like this:

```
sudo apt-get update
sudo apt-get install cifs-utils -y
sudo mkdir /srv/homedir #Create directory for external CIFS storage
sudo mount -t cifs -o username=USER,password=PASSWORD,domain=WIN ...
    //homedir.oit.duke.edu/.../srv/homedir
```

Introducing this type of complexity can add new sources of errors. CIFS does not support the creation of symbolic links, which are used in Kaldi, so a work-around had to be built in order to accommodate the external storage. To get around this, we create an additional folder in the same directory as the CIFS mounted directory and funnel our generated symbolic links into this directory. Symbolic links do not take up enough memory for this to be a problem. We then direct all future symbolic links to this directory.

7.4 Parallelization with Sun GridEngine

7.4.1 Why do I need to do this?

This example is particularly interesting because it is much more complex than introductory examples: it has a lexicon of around 16,000 words and required the use of the Sun Grid Engine for parallelization. This platform let's us split up jobs across multiple CPUs using a queue system.

7.4.2 Assessing Machine Capabilities

Beginning this installation process, it is important to know the capabilities of the machine/cluster with which you are working. The number of CPUs and amount of memory available are of particular importance. To determine this information, type the following command:

```
grep MemTotal /proc/meminfo #Total memory
grep proc /proc/cpuinfo | wc -l #Number of CPUs
```

This information should inform the value of *ram_free* (discussed in the next section) and the variables in the *cmd.sh* script, which dictate the size of jobs passed to the train, decode and make-graph scripts. The number of CPUs should impact how many jobs your program can run at the same time – this is defined in *run.sh* as *numJobs*.

7.4.3 Installation

As mentioned prior, the difficulty in this example lies in dealing with massive amounts of complex data required to run the system. To combat this issue, Kaldi incorporates the Sun GridEngine (SGE) in order to parallelize tasks [4] (the Kaldi site offers guidance on this topic [17]). In this system, a queue management software runs on a master node, while a different set of programs run on worker nodes.

The following command installs GridEngine on the master node:

```
sudo apt-get install gridengine-master gridengine-client
```

We can use automatic configuration (as well as the default cell name), and the "master" hostname should be set to the hostname of the chosen master node (found by running the hostname command in this node).

On the client nodes, we run the following command:

```
sudo apt-get install gridengine-client gridengine-exec
```

We follow similar instructions as before. In our example, because we are working with a single node, we configure the same node as the master and client node.

To confirm success up to this point, we can run the *qstat* and *qhost -q* commands. The first, which is used to check the status of the queues, should return nothing (you have entered no jobs). The second should print two hosts, a global host and your host (entered previously). If your host does not have printed information, something has gone wrong. This is likely a DNS (domain name server) error, as it indicates that a client cannot reach the master at the given hostname. Here are some suggestions for what to do given issues at this point:

1. Explicitly add your master hostname to the /etc/hosts file to ensure DNS resolution. Note: you may need to also list the first name identifier of the hostname [19], as seen in the example below:

```
127.0.0.1       localhost
<IP address>    vcm-id.vm.university.edu vcm-id
```

2. Confirm that SGE_ROOT is correctly defined by printing it in the command line and, if not, set it to /var/lib/gridengine
3. Print the hostname listed in /var/lib/gridengine/default/common/act_qmaster and confirm that it matches your master node hostname
4. Another good test is to use the binaries located in var/lib/gridengine/utilbin/ arch/ – there are a number of programs there such as *gethostbyname* and *gethostbyaddr* – these are used by SGE for DNS lookups

To make the rest of this process easier, we give the current user manager permissions with the following command:

```
sudo qconf -am <your-user-id> #Add yourself as manager
```

Next, we add additional configurations to GridEngine. GridEngine has no default queues, so we must create one from a default template. We add a new queue and open a queue editor, making the alterations listed below:

```
Command: qconf -aq #Add queue command

Old version:
qname               template
shell               /bin/csh

New version:
qname               all.q
shell               /bin/bash
```

We also want to modify the free memory parameter in our configurations so that we can submit and run jobs. We do this by entering the command below (to open an editor) and altering the memory-related variables accordingly.

```
qconf -sc #Modify resource configurations

Original line:
mem_free    mf  MEMORY  ≤   YES    NO   0    0

New lines:
mem_free    mf          MEMORY  ≤   YES    YES  1G   0
gpu         g           INT     ≤   YES    YES  0    10000
ram_free    ram_free    MEMORY  ≤   YES    JOB  1G   0
```

Next, we must configure a parallel environment called *smp* to GridEngine, in order to allow the reservation of CPU slots and the use of the *smp* parallelization method in our queue. To do this, we enter the following editor commands and make the subsequent changes:

```
Command: qconf -ap smp
```

```
Change:
pe_name            smp
slots              9999

Command: qconf -mq all.q
Change:
pe_list            make smp
```

Now that we've properly configured the settings in our GridEngine environment, we must add nodes (to create a network for job completion).

From here, we must set the proper roles for our nodes so that the network functions properly. Note that setting our machine as an execution host spawns an editor, in which we must make a small change to indicate the free RAM and GPU parameters. These value should be informed by our memory considerations (discussed prior):

```
qconf -ah <your-fqdn> #Add your machine as an admin host
qconf -as <your-fdqn> #Add your machine as a submit host
qconf -ae <your-fqdn> #Add your machine as an execution host
#  --> Change: complex_values    ram_free=112G,gpu=1
```

The final step to pull all of this together involves adding our machine to the hostlist (so that the queue can be populated with jobs from the master):

```
Command: qconf -mq all.q #Add machine to hostlist
```

Note that this command will spawn an editor. In this editor, we must add our host, as well as the number of slots allowed (based upon our available CPUs). The file should look something like this:

```
qname                 all.q
hostlist              <host>
...
slots                 30
```

At this point, the SGE system should be properly configured. However, different systems may require additional steps. If the VoxForge script fails on any queue tasks, try any of the steps in the following section.

7.4.4 Debugging SGE

A useful note in debugging is as follows: The SGE platform only starts being used at the stage of MFCC extraction. To expedite debugging, you can comment out the code before this point and run from there (assuming you've successfully run the code prior). This should let you test the queue process in an isolated fashion.

Another useful tool is *qmon*, which allows for graphical interaction with the queues, jobs, host groups, etc. Launching this program allows you to check the queues, monitor the status of jobs, etc.

The UPenn ACG SGE Cluster documentation offers some useful tips for debugging as well [1].

There are also many useful tools in the SGE toolkit that we can use to debug. The *qstat* command, for instance, can be used to monitor jobs:

```
qstat -u '*' #Print all current jobs
qstat -j job-id #Print info about specific job
```

If the job is listed as *r*, that means the job is running – it may just be taking a long time. Meanwhile, *qw* means the job is waiting on the queue – this could either be intentional, or it could mean the system is not properly configured. The status *E* indicates that an error has occurred. Printing the status of this job will provide more information.

The *qhost* command can also be useful to monitor the hosts and queues in the GridEngine. An example output can be seen below:

```
# qhost -q
HOSTNAME  ARCH  NCPU  LOAD  MEMTOT  MEMUSE  SWAPTO  SWAPUS
-------------------------------------------------------------------
global     -     -     -       -      -       -        -
<host> lx26-amd64 2   1.16   1.9G   227.5M   2.0G   6.4M
   all.q               BIP   0/2/2
```

Should these fail, the Kaldi documentation on parallelization is incredibly useful.

7.5 VoxForge Output

Our VoxForge decoding results, summarized and abbreviated in Tables 7 and 8, vary greatly from those we saw in the *Digits* example. Firstly, we see up to 49.34 percent word error rate with monophone training – this is abysmal. The high error rates for this example can be attributed to:

1. A large lexicon (increased state space of options translates to more room for error)

2. Potentially erroneous training data (submitted by VoxForge users)

For this type of example to be useful, more data would be necessary – this much data is fine for demonstration, but having around 20 percent error for *every word* (not every sentence) makes this system difficult to use in practice.

Secondly, because the training/testing data we used actually had context (rather than being single digits), the triphone training is much more successful, coming in around 22.53 percent in the second pass.

This example clearly indicates the potential failures of Kaldi and open-source datasets. When dealing with more complex queries, error skyrockets and the need for data increases. This will need to be considered in development.

Table 7: Monophone Training Results

WER	Percent	Ratio
WER_7	50.36	2249 / 4466
WER_8	48.86	2182 / 4466
WER_9	47.56	2124 / 4466
WER_10	47.27	2111 / 4466
WER_11	47.29	2112 / 4466
WER_12	47.81	2135 / 4466
WER_13	49.08	2192 / 4466
WER_14	50.13	2239 / 4466
WER_15	50.60	2260 / 4466
WER_16	51.28	2290 / 4466
WER_17	52.55	2347 / 4466
Average	49.34	2203/4466

Table 8: Triphone Training Results

Table 9: Pass One

WER	Percent	Ratio
WER_7	27.14	1212 / 4466
WER_8	25.12	1122 / 4466
WER_9	23.67	1057 / 4466
WER_10	22.62	1010 / 4466
WER_11	22.53	1006 / 4466
WER_12	21.99	982 / 4466
WER_13	21.63	966 / 4466
WER_14	21.70	969 / 4466
WER_15	21.65	967 / 4466
WER_16	21.99	982 / 4466
WER_17	22.41	1001 / 4466
Average	22.95	1025 / 4466

Table 10: Pass Two

WER	Percent	Ratio
WER_7	26.87	1200 / 4466
WER_8	24.45	1092 / 4466
WER_9	22.84	1020 / 4466
WER_10	21.99	982 / 4466
WER_11	21.65	967 / 4466
WER_12	21.34	953 / 4466
WER_13	21.70	969 / 4466
WER_14	21.63	966 / 4466
WER_15	21.56	963 / 4466
WER_16	21.85	976 / 4466
WER_17	21.99	1020 / 4466
Average	22.53	1006 / 4466

8 CMU AN4 Example

8.1 Introduction

The CMU AN4 (the Alphanumeric database) is a series of census data recorded at CMU in 1991 [2]. This data will be used to create a system capable of recognizing alphanumeric queries. This example provides insight as to how non-formatted audio and acoustic data can be funneled into Kaldi. We use a hand-written script to retrieve the dataset from the CMU site, renaming and sorting it into training and testing folders. This script also extracts the lexicon, phones, transcriptions, etc. from the /etc files. This script can be found in the appendix (note that it may require some modification for personal use).

8.2 CMU Results

The decoding results for this example, shown in Table 11, are not as promising as initially projected given the relatively large size of the dataset and the small lexicon (131 words). Even in the last pass of triphone decoding, we only get down to a 6.27 percent word error rate. This isn't stellar, considering we only have around one hundred options per word in a sentence.

A potential improvement would be to revise the dataset to use the NATO phonetic alphabet for letters. This system is specifically designed to distinguish between similar letters (such as **M**[ike] and **N**[ovember]). Of course, this would require collection of an entirely new dataset, which would require lots of time and resources.

Another improvement may be to use an error-correcting system, like the one demonstrated in the example below:

```
Service: Please read your serial number.
Client: C O P M N 6 8 D
Service: I'm sorry -- I got the first part [ C O P ]. After ...
    that, did you say M as in Mike or N as in November?
Client: M as in Mike
Service: Ok. Your serial number is C O P M N 6 8 D, is this correct?
Client: Yes
```

Table 11: CMU AN4 Decoding Results

Table 12: Monophone Training

WER	0	5
WER_7	11.64	11.51
WER_8	11.64	11.51
WER_9	11.38	11.77
WER_10	11.64	11.77
WER_11	11.77	12.29
WER_12	12.29	12.68
WER_13	12.03	12.94
WER_14	12.68	13.71
WER_15	13.07	14.62
WER_16	14.10	15.14
WER_17	14.88	15.65
Average	12.46	11.88

Table 14: Triphone Pass 2

WER	0	5
WER_7	16.95	16.43
WER_8	16.04	14.88
WER_9	14.36	13.84
WER_10	13.84	13.58
WER_11	13.45	12.81
WER_12	12.81	12.81
WER_13	12.81	12.42
WER_14	12.68	12.29
WER_15	12.42	12.29
WER_16	12.16	12.03
WER_17	12.03	11.90
Average	12.43	13.21

Table 13: Triphone Pass 1

WER	0	5
WER_7	16.95	15.65
WER_8	15.27	13.97
WER_9	13.7	13.45
WER_10	13.32	12.55
WER_11	12.55	12.03
WER_12	12.03	11.64
WER_13	11.77	11.51
WER_14	11.51	11.64
WER_15	11.64	11.77
WER_16	11.77	11.64
WER_17	11.77	11.38
Average	12.93	12.48

Table 15: Triphone Pass 3

WER	0	5
WER_7	6.99	6.99
WER_8	6.60	6.47
WER_9	6.47	6.60
WER_10	6.60	6.47
WER_11	6.47	6.47
WER_12	6.47	6.47
WER_13	6.47	6.08
WER_14	6.08	6.08
WER_15	6.08	5.95
WER_16	6.08	5.69
WER_17	5.95	5.69
Average	6.39	6.27

9 How Does Kaldi Measure Up?

In Figure 6 below, we can see how the error rates from these Kaldi examples (taken to be the best training system average word error rate) line up with current error rates for on-the-market systems [15] [18]. It should be noted that the comparison is not necessarily between completely similar systems – Google and Cortana are on much larger scales, designed to recognize all possible input (as opposed to our customized systems). Large companies spend billions on data collection, system training, optimization, etc. They devote much more man power and money to these systems than any one individual could. Hence, these "comparisons" should be considered more as standard benchmarks, rather than direct comparisons.

In Figure 7, we can see how the training dataset size must increase with the lexicon size to maintain a reasonable word error rate for the system. This is standard across most ASR systems.

As a final caveat, we must think beyond the word error rate. For instance, a 4.9 percent error rate for Google translates to a 4.9 percent chance of erroneous transcription for *every word* in a sentence (not considering contextual changes). This error will compound over the course of an entire sentence. Hence, an error rate on the scale of 20 percent (seen in CMU AN4), where there is a 20 percent chance of incorrect transcription with each word, virtually guarantees at least one erroneous word transcription for a long sentence.

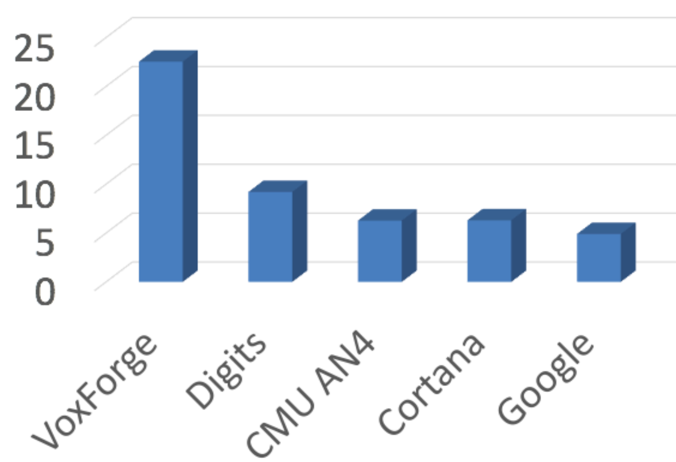

Figure 6: Word Error Rates in Kaldi examples compared to readily available systems [15] [18]

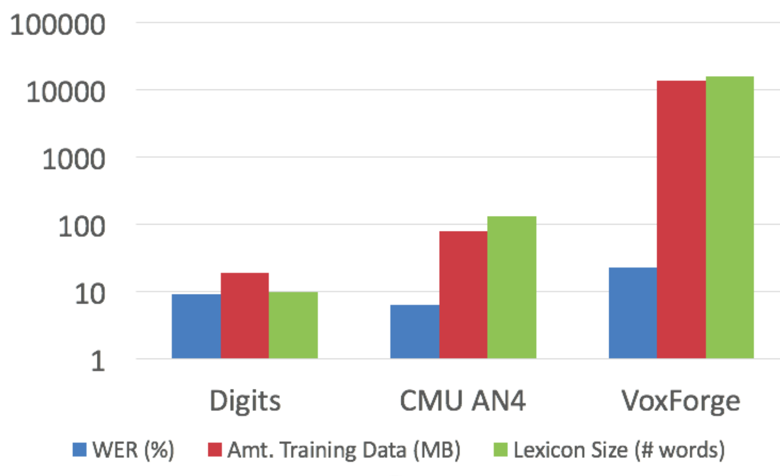

Figure 7: Word Error Rates, lexicon size and dataset size across Kaldi Examples (log scale)

10 Conclusion

Given the flexibility demonstrated by the Kaldi toolkit, it is safe to say that further extensions and explorations will be possible. The ideal case will involve the incorporation of a large, custom training dataset, which we have shown to be possible. Another important extension will be real-time encoding – right now, this system is geared towards static, already-recorded datasets. Our product will require a dynamic system that can accommodate real-time decoding. Such examples are clearly possible, as indicated by the Kaldi Online Decoding Tutorial[17] and the Kaldi GStreamer (a real-time speech recognition server implemented using Kaldi and readily available on Github)[7]. It should be noted that a variety of elements were not considered in this analysis, including speed. Future explorations must confirm that Kaldi real-time decoding is capable of supplying speech-to-text results in under ten seconds (given our custom dataset). Similarly, we identified several potential issues (e.g., a need for a huge dataset to achieve low word error rates) that will need to be addressed. The information we have seen so far, however, indicates that Kaldi is capable of providing accurate and flexible solutions to the problem of speech recognition.

References

[1] ACG's Sun Grid Engine (SGE) Cluster. University of Pennsylvania.

[2] CMU Census Database, 1991. Carnegie Mellon University.

[3] VoxForge: Open Source Acoustic Model and Audio Transcriptions, 2006-2017. VoxForge.

[4] Sun Cluster Data Service for Sun Grid Engine Guide for Solaris OS, 2010. Oracle.

[5] SRILM - The SRI Language Modeling Toolkit, 2011. SRI International.

[6] SoX - Sound eXchange, 2015. PmWiki.

[7] Tanel Alumäe. Kaldi GStreamer server, 2017.

[8] Madeline Briere. Speech Processing and Recognition, 2017.

[9] Berlin Chen. An Introduction to the Kaldi Speech Recognition Toolkit, 2014. National Taiwan Normal University.

[10] Eleanor Chodroff. Corpus Phonetics Tutorial: Kaldi, 2017. Northwestern University.

[11] Gruhn et. al. Automatic Speech Recognition, 2011. Statistical Pronunciation Modeling for Non-Native Speech Processing.

[12] Mirko Hannemann. Weighted Finite State Transducers in Automatic Speech Recognition, 2013. Computer Science Department at Brandeis.

[13] Zohar Jackson. Free Spoken Digit Dataset (FSDD), 2017.

[14] Duane Johnson. Understanding ARPA and Language Models, 2014. WordTree.

[15] Sundar Pichai. Google's I/O Developer Conference, 2017.

[16] Ondřej Plátek. Automatic speech recognition using Kaldi. Institute of Formal and Applied Linguistics.

[17] Daniel Povey, Arnab Ghoshal, Gilles Boulianne, Lukas Burget, Ondrej Glembek, Nagendra Goel, Mirko Hannemann, Petr Motlicek, Yanmin Qian, Petr Schwarz, Jan Silovsky, Georg Stemmer, and Karel Vesely. The Kaldi Speech Recognition Toolkit, 2011.

[18] George Saon. Recent Advances in Conversational Speech Recognition, 2016. IBM.

[19] Sam Skipsey. Reresolve hostname failed: can't resolve hostname with SGE 6.1, 2008. University of Liverpool.

[20] Lyndon White. Kaldi-notes: Some notes on Kaldi. University of Western Australia.

11 Appendix

11.1 Basic audio sorting script, *sort.sh*

```
# Run in free-spoken-digit-dataset directory

declare -a speakers=("jackson", "theo", "nicolas")

for i in {0..49}
do
    for j in {0..9}
    do
        for k in "${speakers[@]}"
        do
            mkdir $k
            folder="recordings"
            utterance="${j}_${k}_${i}.wav"
            newfile="{k}_{j}_${i}.wav"
            utterance="${folder}/${utterance}"
            newfile="${folder}/${k}/${newfile}"
        done
    done
done
```

11.2 Acoustic Data Script, *acoustic.sh*

```
#!/bin/bash
DATA_TEST="data/test"
DATA_TRAIN="data/train"

# TODO: Don't hard code this
declare -a test=("jackson")
declare -a train=("theo" "nicolas")
declare -A map=( [0]="zero" [1]="one" [2]="two" [3]="three" ...
    [4]="four" [5]="five" [6]="six" [7]="seven" [8]="eight" ...
    [9]="nine" )
declare -a arr=( "one" "two" "three" "four" "five" "six" "seven" ...
    "eight" "nine" "zero" )
user="mfb33"

# TODO: Check that in example
# Prompt for delete folder instead of exit

# Organization
if [ -d "$DATA_TEST" ]; then
    echo "Test folder already exists, please remove."
    exit 1
else
    mkdir data/test
fi

if [ -d "$DATA_TRAIN" ]; then
```

```
    echo "Train folder already exists, please remove."
    exit 1
else
    mkdir data/train
fi

# Enter data folder
cd data

# Creation of spk2gender files
touch test/spk2gender
touch train/spk2gender

#TODO: don't hard code

echo "jackson m" >> test/spk2gender
echo "nicolas m
theo m" >> train/spk2gender


# Creation of wav.scp
# <uterranceID> <full_path_to_audio_file>
touch test/wav.scp
touch train/wav.scp

for i in {0..49}
do
    for j in {0..9}
    do
        for k in "${test[@]}"
        do
            folder="recordings"
            end=".wav"
            utterance="${k}_${j}_${i}"
            path="/home/$user/kaldi/egs/digits/digits_audio/test/"
            file="$path${k}/$utterance$end"
            echo "$utterance $file" >> test/wav.scp
        done
        for k in "${train[@]}"
        do
            folder="recordings"
            end=".wav"
            utterance="${k}_${j}_${i}"
            path="/home/$user/kaldi/egs/digits/digits_audio/train/"
            file="$path${k}/$utterance$end"
            echo "$utterance $file" >> train/wav.scp
        done
    done
done


# Generate text
# <uterranceID> <text_transcription>
touch test/text
touch train/text


```

```

for i in {0..49}
do
    for j in {0..9}
    do
        for k in "${train[@]}"
        do
            utterance="${k}_${j}_${i}"
            echo "$utterance ${map[$j]}" >> train/text
        done

        for k in "${test[@]}"
        do
            utterance="${k}_${j}_${i}"
            echo "$utterance ${map[$j]}" >> test/text
        done
    done
done

# Create utt2speak
# <uterranceID> <speakerID>
touch test/utt2spk
touch train/utt2spk

for i in {0..49}
do
    for j in {0..9}
    do
        for k in "${test[@]}"
        do
            utterance="${k}_${j}_${i}"
            echo -e "$utterance $k" >> test/utt2spk
        done
        for k in "${train[@]}"
        do
            utterance="${k}_${j}_${i}"
            echo -e "$utterance $k" >> train/utt2spk
        done
    done
done

# Create corpus
# <text_transcription>
touch local/corpus.txt

for i in "${arr[@]}"
do
    echo $i >> local/corpus.txt
done

# Fix sorting
cd ..
./utils/validate_data_dir.sh data/test
./utils/fix_data_dir.sh data/test
./utils/validate_data_dir.sh data/train
./utils/fix_data_dir.sh data/train
```

11.3 Digits *resample.m* script

```
filename = '';
for name = {'jackson', 'theo', 'nicolas'}
    for i = 0:9
        for j=0:49
            filename = ...
                strcat('free-spoken-digit-dataset/recordings/', ...
                name, '/', name, '_', num2str(i), '_', ...
                num2str(j), '.wav');
            file = char(filename);
            [y,Fs] = audioread(file);
            up = resample(y, 2, 1,    pchip   );
            delete(file);
            audiowrite(file, up, Fs*2);
        end
    end
end
```

11.4 Digits *run.sh* script

```
#!/bin/bash
. ./path.sh || exit 1
. ./cmd.sh || exit 1
nj=1 # number of parallel jobs - 1 is perfect for such a small ...
    data set
lm_order=1 # language model order (n-gram quantity) - 1 is ...
    enough for digits grammar

# Safety mechanism (possible running this script with modified ...
    arguments)
. utils/parse_options.sh || exit 1
[[ $# -ge 1 ]] && { echo "Wrong arguments!"; exit 1; }

# Removing previously created data (from last run.sh execution)
rm -rf exp mfcc data/train/spk2utt data/train/cmvn.scp ...
    data/train/feats.scp data/train/split1 data/test/spk2utt ...
    data/test/cmvn.scp data/test/feats.scp data/test/split1 ...
    data/local/lang data/lang data/local/tmp ...
    data/local/dict/lexiconp.txt

echo
echo "===== PREPARING ACOUSTIC DATA ====="
echo

# Needs to be prepared by hand (or using self written scripts):
#
# spk2gender  [<speaker-id> <gender>]
# wav.scp     [<uterranceID> <full_path_to_audio_file>]
# text           [<uterranceID> <text_transcription>]
# utt2spk     [<uterranceID> <speakerID>]
# corpus.txt  [<text_transcription>]
```

```
# Making spk2utt files
utils/utt2spk_to_spk2utt.pl data/train/utt2spk > data/train/spk2utt
utils/utt2spk_to_spk2utt.pl data/test/utt2spk > data/test/spk2utt

echo
echo "===== FEATURES EXTRACTION ====="
echo

# Making feats.scp files
mfccdir=mfcc
# Uncomment and modify arguments in scripts below if you have ...
    any problems with data sorting
# utils/validate_data_dir.sh data/train     # script for ...
    checking prepared data - here: for data/train directory
# utils/fix_data_dir.sh data/train          # tool for data ...
    proper sorting if needed - here: for data/train directory
steps/make_mfcc.sh --nj $nj --cmd "$train_cmd" data/train ...
    exp/make_mfcc/train $mfccdir
steps/make_mfcc.sh --nj $nj --cmd "$train_cmd" data/test ...
    exp/make_mfcc/test $mfccdir

# Making cmvn.scp files
steps/compute_cmvn_stats.sh data/train exp/make_mfcc/train $mfccdir
steps/compute_cmvn_stats.sh data/test exp/make_mfcc/test $mfccdir

echo
echo "===== PREPARING LANGUAGE DATA ====="
echo

# Needs to be prepared by hand (or using self written scripts):
#
# lexicon.txt           [<word> <phone 1> <phone 2> ...]
# nonsilence_phones.txt    [<phone>]
# silence_phones.txt    [<phone>]
# optional_silence.txt  [<phone>]

# Preparing language data
utils/prepare_lang.sh data/local/dict "<UNK>" data/local/lang ...
    data/lang

echo
echo "===== LANGUAGE MODEL CREATION ====="
echo "===== MAKING lm.arpa ====="
echo

loc=`which ngram-count`;
if [ -z $loc ]; then
    if uname -a | grep 64 >/dev/null; then
         sdir=$KALDI_ROOT/tools/srilm/bin/i686-m64
    else
        sdir=$KALDI_ROOT/tools/srilm/bin/i686
    fi
    if [ -f $sdir/ngram-count ]; then
        echo "Using SRILM language modelling tool from $sdir"
        export PATH=$PATH:$sdir
     else
```

```
        echo "SRILM toolkit is probably not installed.
         Instructions: tools/install_srilm.sh"
        exit 1
     fi
fi

local=data/local
mkdir $local/tmp
ngram-count -order $lm_order -write-vocab ...
    $local/tmp/vocab-full.txt -wbdiscount -text ...
    $local/corpus.txt -lm $local/tmp/lm.arpa

echo
echo "===== MAKING G.fst ====="
echo

lang=data/lang
arpa2fst --disambig-symbol=#0 ...
    --read-symbol-table=$lang/words.txt $local/tmp/lm.arpa ...
    $lang/G.fst

echo
echo "===== MONO TRAINING ====="
echo

steps/train_mono.sh --nj $nj --cmd "$train_cmd" data/train ...
    data/lang exp/mono  || exit 1

echo
echo "===== MONO DECODING ====="
echo

utils/mkgraph.sh --mono data/lang exp/mono exp/mono/graph || ...
    exit 1
steps/decode.sh --config conf/decode.config --nj $nj --cmd ...
    "$decode_cmd" exp/mono/graph data/test exp/mono/decode

echo
echo "===== MONO ALIGNMENT ====="
echo

steps/align_si.sh --nj $nj --cmd "$train_cmd" data/train ...
    data/lang exp/mono exp/mono_ali || exit 1

echo
echo "===== TRI1 (first triphone pass) TRAINING ====="
echo

steps/train_Δs.sh --cmd "$train_cmd" 2000 11000 data/train ...
    data/lang exp/mono_ali exp/tri1 || exit 1

echo
echo "===== TRI1 (first triphone pass) DECODING ====="
echo

utils/mkgraph.sh data/lang exp/tri1 exp/tri1/graph || exit 1
```

```
steps/decode.sh --config conf/decode.config --nj $nj --cmd ...
    "$decode_cmd" exp/tri1/graph data/test exp/tri1/decode

echo
echo "===== run.sh script is finished ====="
echo
```

11.5 Digits *run.sh* Output

```
===== PREPARING ACOUSTIC DATA =====


===== FEATURES EXTRACTION =====

steps/make_mfcc.sh --nj 1 --cmd run.pl data/train ...
    exp/make_mfcc/train mfcc
utils/validate_data_dir.sh: Successfully validated ...
    data-directory data/train
steps/make_mfcc.sh: [info]: no segments file exists: assuming ...
    wav.scp indexed by utterance.
Succeeded creating MFCC features for train
steps/make_mfcc.sh --nj 1 --cmd run.pl data/test ...
    exp/make_mfcc/test mfcc
utils/validate_data_dir.sh: WARNING: you have only one speaker.  ...
    This probably a bad idea.
   Search for the word 'bold' in ...
       http://kaldi-asr.org/doc/data_prep.html
   for more information.
utils/validate_data_dir.sh: Successfully validated ...
    data-directory data/test
steps/make_mfcc.sh: [info]: no segments file exists: assuming ...
    wav.scp indexed by utterance.
Succeeded creating MFCC features for test
steps/compute_cmvn_stats.sh data/train exp/make_mfcc/train mfcc
Succeeded creating CMVN stats for train
steps/compute_cmvn_stats.sh data/test exp/make_mfcc/test mfcc
Succeeded creating CMVN stats for test

===== PREPARING LANGUAGE DATA =====

utils/prepare_lang.sh data/local/dict <UNK> data/local/lang ...
    data/lang
Checking data/local/dict/silence_phones.txt ...
--> reading data/local/dict/silence_phones.txt
--> data/local/dict/silence_phones.txt is OK

Checking data/local/dict/optional_silence.txt ...
--> reading data/local/dict/optional_silence.txt
--> data/local/dict/optional_silence.txt is OK

Checking data/local/dict/nonsilence_phones.txt ...
--> reading data/local/dict/nonsilence_phones.txt
--> data/local/dict/nonsilence_phones.txt is OK

```

```
Checking disjoint: silence_phones.txt, nonsilence_phones.txt
--> disjoint property is OK.

Checking data/local/dict/lexicon.txt
--> reading data/local/dict/lexicon.txt
--> data/local/dict/lexicon.txt is OK

Checking data/local/dict/extra_questions.txt ...
--> data/local/dict/extra_questions.txt is empty (this is OK)
--> SUCCESS [validating dictionary directory data/local/dict]

**Creating data/local/dict/lexiconp.txt from ...
    data/local/dict/lexicon.txt
fstaddselfloops data/lang/phones/wdisambig_phones.int ...
    data/lang/phones/wdisambig_words.int
prepare_lang.sh: validating output directory
utils/validate_lang.pl data/lang
Checking data/lang/phones.txt ...
--> data/lang/phones.txt is OK

Checking words.txt: #0 ...
--> data/lang/words.txt is OK

Checking disjoint: silence.txt, nonsilence.txt, disambig.txt ...
--> silence.txt and nonsilence.txt are disjoint
--> silence.txt and disambig.txt are disjoint
--> disambig.txt and nonsilence.txt are disjoint
--> disjoint property is OK

Checking sumation: silence.txt, nonsilence.txt, disambig.txt ...
--> summation property is OK

Checking data/lang/phones/context_indep.{txt, int, csl} ...
--> 10 entry/entries in data/lang/phones/context_indep.txt
--> data/lang/phones/context_indep.int corresponds to ...
    data/lang/phones/context_indep.txt
--> data/lang/phones/context_indep.csl corresponds to ...
    data/lang/phones/context_indep.txt
--> data/lang/phones/context_indep.{txt, int, csl} are OK

Checking data/lang/phones/nonsilence.{txt, int, csl} ...
--> 80 entry/entries in data/lang/phones/nonsilence.txt
--> data/lang/phones/nonsilence.int corresponds to ...
    data/lang/phones/nonsilence.txt
--> data/lang/phones/nonsilence.csl corresponds to ...
    data/lang/phones/nonsilence.txt
--> data/lang/phones/nonsilence.{txt, int, csl} are OK

Checking data/lang/phones/silence.{txt, int, csl} ...
--> 10 entry/entries in data/lang/phones/silence.txt
--> data/lang/phones/silence.int corresponds to ...
    data/lang/phones/silence.txt
--> data/lang/phones/silence.csl corresponds to ...
    data/lang/phones/silence.txt
--> data/lang/phones/silence.{txt, int, csl} are OK

Checking data/lang/phones/optional_silence.{txt, int, csl} ...
```

```
--> 1 entry/entries in data/lang/phones/optional_silence.txt
--> data/lang/phones/optional_silence.int corresponds to ...
    data/lang/phones/optional_silence.txt
--> data/lang/phones/optional_silence.csl corresponds to ...
    data/lang/phones/optional_silence.txt
--> data/lang/phones/optional_silence.{txt, int, csl} are OK

Checking data/lang/phones/disambig.{txt, int, csl} ...
--> 2 entry/entries in data/lang/phones/disambig.txt
--> data/lang/phones/disambig.int corresponds to ...
    data/lang/phones/disambig.txt
--> data/lang/phones/disambig.csl corresponds to ...
    data/lang/phones/disambig.txt
--> data/lang/phones/disambig.{txt, int, csl} are OK

Checking data/lang/phones/roots.{txt, int} ...
--> 22 entry/entries in data/lang/phones/roots.txt
--> data/lang/phones/roots.int corresponds to ...
    data/lang/phones/roots.txt
--> data/lang/phones/roots.{txt, int} are OK

Checking data/lang/phones/sets.{txt, int} ...
--> 22 entry/entries in data/lang/phones/sets.txt
--> data/lang/phones/sets.int corresponds to ...
    data/lang/phones/sets.txt
--> data/lang/phones/sets.{txt, int} are OK

Checking data/lang/phones/extra_questions.{txt, int} ...
--> 9 entry/entries in data/lang/phones/extra_questions.txt
--> data/lang/phones/extra_questions.int corresponds to ...
    data/lang/phones/extra_questions.txt
--> data/lang/phones/extra_questions.{txt, int} are OK

Checking data/lang/phones/word_boundary.{txt, int} ...
--> 90 entry/entries in data/lang/phones/word_boundary.txt
--> data/lang/phones/word_boundary.int corresponds to ...
    data/lang/phones/word_boundary.txt
--> data/lang/phones/word_boundary.{txt, int} are OK

Checking optional_silence.txt ...
--> reading data/lang/phones/optional_silence.txt
--> data/lang/phones/optional_silence.txt is OK

Checking disambiguation symbols: #0 and #1
--> data/lang/phones/disambig.txt has "#0" and "#1"
--> data/lang/phones/disambig.txt is OK

Checking topo ...

Checking word_boundary.txt: silence.txt, nonsilence.txt, ...
    disambig.txt ...
--> data/lang/phones/word_boundary.txt doesn't include ...
    disambiguation symbols
--> data/lang/phones/word_boundary.txt is the union of ...
    nonsilence.txt and silence.txt
--> data/lang/phones/word_boundary.txt is OK

```

```
Checking word-level disambiguation symbols...
--> data/lang/phones/wdisambig.txt exists (newer prepare_lang.sh)
Checking word_boundary.int and disambig.int
--> generating a 81 word sequence
--> resulting phone sequence from L.fst corresponds to the word ...
    sequence
--> L.fst is OK
--> generating a 79 word sequence
--> resulting phone sequence from L_disambig.fst corresponds to ...
    the word sequence
--> L_disambig.fst is OK

Checking data/lang/oov.{txt, int} ...
--> 1 entry/entries in data/lang/oov.txt
--> data/lang/oov.int corresponds to data/lang/oov.txt
--> data/lang/oov.{txt, int} are OK

--> data/lang/L.fst is olabel sorted
--> data/lang/L_disambig.fst is olabel sorted
--> SUCCESS [validating lang directory data/lang]

===== LANGUAGE MODEL CREATION =====
===== MAKING lm.arpa =====

Using SRILM language modelling tool from ...
    /home/mfb33/kaldi/egs/digits/../../tools/srilm/bin/i686-m64

===== MAKING G.fst =====

arpa2fst --disambig-symbol=#0 ...
    --read-symbol-table=data/lang/words.txt ...
    data/local/tmp/lm.arpa data/lang/G.fst
LOG (arpa2fst[5.2.134¬1-ecd4]:Read():arpa-file-parser.cc:98) ...
    Reading \data\ section.
LOG (arpa2fst[5.2.134¬1-ecd4]:Read():arpa-file-parser.cc:153) ...
    Reading \1-grams: section.
LOG (arpa2fst[5.2.134¬1-ecd4]: RemoveRedundantStates(): ...
    arpa-lm-compiler.cc:359) Reduced num-states from 1 to 1

===== MONO TRAINING =====

steps/train_mono.sh --nj 1 --cmd run.pl data/train data/lang ...
    exp/mono
steps/train_mono.sh: Initializing monophone system.
steps/train_mono.sh: Compiling training graphs
steps/train_mono.sh: Aligning data equally (pass 0)
steps/train_mono.sh: Pass 1
steps/train_mono.sh: Aligning data
steps/train_mono.sh: Pass 2
steps/train_mono.sh: Aligning data
steps/train_mono.sh: Pass 3
...
steps/train_mono.sh: Pass 38
steps/train_mono.sh: Aligning data
steps/train_mono.sh: Pass 39
steps/diagnostic/analyze_alignments.sh --cmd run.pl data/lang ...
    exp/mono
```

```
analyze_phone_length_stats.py: WARNING: optional-silence sil is ...
    seen only 12.8% of the time at utterance begin.  This may ...
    not be optimal.
analyze_phone_length_stats.py: WARNING: optional-silence sil is ...
    seen only 5.3% of the time at utterance end.  This may not ...
    be optimal.
steps/diagnostic/analyze_alignments.sh: see stats in ...
    exp/mono/log/analyze_alignments.log
61 warnings in exp/mono/log/align.*.*.log
2 warnings in exp/mono/log/analyze_alignments.log
228 warnings in exp/mono/log/update.*.log
exp/mono: nj=1 align prob=-76.67 over 0.10h [retry=0.0%, ...
    fail=0.0%] states=70 gauss=1003
steps/train_mono.sh: Done training monophone system in exp/mono

===== MONO DECODING =====

WARNING: the --mono, --left-biphone and --quinphone options are ...
    now deprecated and ignored.
tree-info exp/mono/tree
tree-info exp/mono/tree
fsttablecompose data/lang/L_disambig.fst data/lang/G.fst
fstminimizeencoded
fstpushspecial
fstdeterminizestar --use-log=true
fstisstochastic data/lang/tmp/LG.fst
-0.0338077 -0.0345085
[info]: LG not stochastic.
fstcomposecontext --context-size=1 --central-position=0 ...
    --read-disambig-syms=data/lang/phones/disambig.int ...
    --write-disambig-syms=data/lang/tmp/disambig_ilabels_1_0.int ...
    data/lang/tmp/ilabels_1_0.56333
fstisstochastic data/lang/tmp/CLG_1_0.fst
-0.0338077 -0.0345085
[info]: CLG not stochastic.
make-h-transducer ...
    --disambig-syms-out=exp/mono/graph/disambig_tid.int ...
    --transition-scale=1.0 data/lang/tmp/ilabels_1_0 ...
    exp/mono/tree exp/mono/final.mdl
fstrmepslocal
fstminimizeencoded
fstdeterminizestar --use-log=true
fstrmsymbols exp/mono/graph/disambig_tid.int
fsttablecompose exp/mono/graph/Ha.fst data/lang/tmp/CLG_1_0.fst
fstisstochastic exp/mono/graph/HCLGa.fst
0.000331514 -0.0349441
HCLGa is not stochastic
add-self-loops --self-loop-scale=0.1 --reorder=true ...
    exp/mono/final.mdl
steps/decode.sh --config conf/decode.config --nj 1 --cmd run.pl ...
    exp/mono/graph data/test exp/mono/decode
decode.sh: feature type is Δ
steps/diagnostic/analyze_lats.sh --cmd run.pl exp/mono/graph ...
    exp/mono/decode
analyze_phone_length_stats.py: WARNING: optional-silence sil is ...
    seen only 51.0% of the time at utterance begin.  This may ...
    not be optimal.
```

```
analyze_phone_length_stats.py: WARNING: optional-silence sil is ...
    seen only 34.6% of the time at utterance end.  This may not ...
    be optimal.
steps/diagnostic/analyze_lats.sh: see stats in ...
    exp/mono/decode/log/analyze_alignments.log
Overall, lattice depth (10,50,90-percentile)=(1,1,3) and mean=1.4
steps/diagnostic/analyze_lats.sh: see stats in ...
    exp/mono/decode/log/analyze_lattice_depth_stats.log
exp/mono/decode/wer_10
%WER 7.60 [ 38 / 500, 0 ins, 20 del, 18 sub ]
%SER 7.60 [ 38 / 500 ]
exp/mono/decode/wer_11
%WER 8.00 [ 40 / 500, 0 ins, 21 del, 19 sub ]
%SER 8.00 [ 40 / 500 ]
exp/mono/decode/wer_12
%WER 8.40 [ 42 / 500, 0 ins, 23 del, 19 sub ]
%SER 8.40 [ 42 / 500 ]
exp/mono/decode/wer_13
%WER 8.40 [ 42 / 500, 0 ins, 23 del, 19 sub ]
%SER 8.40 [ 42 / 500 ]
exp/mono/decode/wer_14
%WER 9.20 [ 46 / 500, 0 ins, 27 del, 19 sub ]
%SER 9.20 [ 46 / 500 ]
exp/mono/decode/wer_15
%WER 9.80 [ 49 / 500, 0 ins, 30 del, 19 sub ]
%SER 9.80 [ 49 / 500 ]
exp/mono/decode/wer_16
%WER 10.20 [ 51 / 500, 0 ins, 32 del, 19 sub ]
%SER 10.20 [ 51 / 500 ]
exp/mono/decode/wer_17
%WER 10.80 [ 54 / 500, 0 ins, 35 del, 19 sub ]
%SER 10.80 [ 54 / 500 ]
exp/mono/decode/wer_7
%WER 7.40 [ 37 / 500, 0 ins, 18 del, 19 sub ]
%SER 7.40 [ 37 / 500 ]
exp/mono/decode/wer_8
%WER 7.40 [ 37 / 500, 0 ins, 19 del, 18 sub ]
%SER 7.40 [ 37 / 500 ]
exp/mono/decode/wer_9
%WER 7.20 [ 36 / 500, 0 ins, 19 del, 17 sub ]
%SER 7.20 [ 36 / 500 ]

===== MONO ALIGNMENT =====

steps/align_si.sh --nj 1 --cmd run.pl data/train data/lang ...
    exp/mono exp/mono_ali
steps/align_si.sh: feature type is Δ
steps/align_si.sh: aligning data in data/train using model from ...
    exp/mono, putting alignments in exp/mono_ali
steps/diagnostic/analyze_alignments.sh --cmd run.pl data/lang ...
    exp/mono_ali
analyze_phone_length_stats.py: WARNING: optional-silence sil is ...
    seen only 12.8% of the time at utterance begin.  This may ...
    not be optimal.
analyze_phone_length_stats.py: WARNING: optional-silence sil is ...
    seen only 5.3% of the time at utterance end.  This may not ...
    be optimal.
```

```
steps/diagnostic/analyze_alignments.sh: see stats in ...
    exp/mono_ali/log/analyze_alignments.log
steps/align_si.sh: done aligning data.

===== TRI1 (first triphone pass) TRAINING =====

steps/train_∆s.sh --cmd run.pl 2000 11000 data/train data/lang ...
    exp/mono_ali exp/tri1
steps/train_∆s.sh: accumulating tree stats
steps/train_∆s.sh: getting questions for tree-building, via ...
    clustering
steps/train_∆s.sh: building the tree
WARNING ...
    (gmm-init-model[5.2.134¬1-ecd4]:InitAmGmm():gmm-init-model.cc:55) ...
    Tree has pdf-id 1 with no stats; corresponding phone list: 6 ...
    7 8 9 10
** The warnings above about 'no stats' generally mean you have ...
    phones **
** (or groups of phones) in your phone set that had no ...
    corresponding data. **
** You should probably figure out whether something went wrong, **
** or whether your data just doesn't happen to have examples of ...
    those **
** phones. **
steps/train_∆s.sh: converting alignments from exp/mono_ali to ...
    use current tree
steps/train_∆s.sh: compiling graphs of transcripts
steps/train_∆s.sh: training pass 1
steps/train_∆s.sh: training pass 2
steps/train_∆s.sh: training pass 3
....
steps/train_∆s.sh: training pass 32
steps/train_∆s.sh: training pass 33
steps/train_∆s.sh: training pass 34
steps/diagnostic/analyze_alignments.sh --cmd run.pl data/lang ...
    exp/tri1
analyze_phone_length_stats.py: WARNING: optional-silence sil is ...
    seen only 12.7% of the time at utterance begin.  This may ...
    not be optimal.
analyze_phone_length_stats.py: WARNING: optional-silence sil is ...
    seen only 5.4% of the time at utterance end.  This may not ...
    be optimal.
steps/diagnostic/analyze_alignments.sh: see stats in ...
    exp/tri1/log/analyze_alignments.log
12 warnings in exp/tri1/log/init_model.log
7 warnings in exp/tri1/log/align.*.*.log
1 warnings in exp/tri1/log/questions.log
1 warnings in exp/tri1/log/mixup.log
1 warnings in exp/tri1/log/build_tree.log
832 warnings in exp/tri1/log/update.*.log
2 warnings in exp/tri1/log/analyze_alignments.log
exp/tri1: nj=1 align prob=-73.94 over 0.10h [retry=0.1%, ...
    fail=0.0%] states=105 gauss=1703 tree-impr=6.52
steps/train_∆s.sh: Done training system with ∆+∆-∆ features in ...
    exp/tri1

===== TRI1 (first triphone pass) DECODING =====
```

```
tree-info exp/tri1/tree
tree-info exp/tri1/tree
fstcomposecontext --context-size=3 --central-position=1 ...
    --read-disambig-syms=data/lang/phones/disambig.int ...
    --write-disambig-syms=data/lang/tmp/disambig_ilabels_3_1.int ...
    data/lang/tmp/ilabels_3_1.58581
fstisstochastic data/lang/tmp/CLG_3_1.fst
0 -0.0345085
[info]: CLG not stochastic.
make-h-transducer ...
    --disambig-syms-out=exp/tri1/graph/disambig_tid.int ...
    --transition-scale=1.0 data/lang/tmp/ilabels_3_1 ...
    exp/tri1/tree exp/tri1/final.mdl
fstminimizeencoded
fstdeterminizestar --use-log=true
fstrmepslocal
fsttablecompose exp/tri1/graph/Ha.fst data/lang/tmp/CLG_3_1.fst
fstrmsymbols exp/tri1/graph/disambig_tid.int
fstisstochastic exp/tri1/graph/HCLGa.fst
0.000331514 -0.0788057
HCLGa is not stochastic
add-self-loops --self-loop-scale=0.1 --reorder=true ...
    exp/tri1/final.mdl
steps/decode.sh --config conf/decode.config --nj 1 --cmd run.pl ...
    exp/tri1/graph data/test exp/tri1/decode
decode.sh: feature type is Δ
steps/diagnostic/analyze_lats.sh --cmd run.pl exp/tri1/graph ...
    exp/tri1/decode
analyze_phone_length_stats.py: WARNING: optional-silence sil is ...
    seen only 52.6% of the time at utterance begin.  This may ...
    not be optimal.
analyze_phone_length_stats.py: WARNING: optional-silence sil is ...
    seen only 40.2% of the time at utterance end.  This may not ...
    be optimal.
steps/diagnostic/analyze_lats.sh: see stats in ...
    exp/tri1/decode/log/analyze_alignments.log
Overall, lattice depth (10,50,90-percentile)=(1,1,2) and mean=1.3
steps/diagnostic/analyze_lats.sh: see stats in ...
    exp/tri1/decode/log/analyze_lattice_depth_stats.log
exp/tri1/decode/wer_10
%WER 9.40 [ 47 / 500, 13 ins, 7 del, 27 sub ]
%SER 9.40 [ 47 / 500 ]
exp/tri1/decode/wer_11
%WER 9.60 [ 48 / 500, 13 ins, 8 del, 27 sub ]
%SER 9.60 [ 48 / 500 ]
exp/tri1/decode/wer_12
%WER 9.40 [ 47 / 500, 13 ins, 8 del, 26 sub ]
%SER 9.40 [ 47 / 500 ]
exp/tri1/decode/wer_13
%WER 9.00 [ 45 / 500, 11 ins, 9 del, 25 sub ]
%SER 9.00 [ 45 / 500 ]
exp/tri1/decode/wer_14
%WER 9.40 [ 47 / 500, 10 ins, 11 del, 26 sub ]
%SER 9.40 [ 47 / 500 ]
exp/tri1/decode/wer_15
%WER 8.80 [ 44 / 500, 8 ins, 11 del, 25 sub ]
```

```
%SER 8.80 [ 44 / 500 ]
exp/tri1/decode/wer_16
%WER 8.40 [ 42 / 500, 5 ins, 12 del, 25 sub ]
%SER 8.40 [ 42 / 500 ]
exp/tri1/decode/wer_17
%WER 8.40 [ 42 / 500, 3 ins, 12 del, 27 sub ]
%SER 8.40 [ 42 / 500 ]
exp/tri1/decode/wer_7
%WER 9.60 [ 48 / 500, 15 ins, 5 del, 28 sub ]
%SER 9.40 [ 47 / 500 ]
exp/tri1/decode/wer_8
%WER 9.40 [ 47 / 500, 14 ins, 6 del, 27 sub ]
%SER 9.40 [ 47 / 500 ]
exp/tri1/decode/wer_9
%WER 9.40 [ 47 / 500, 14 ins, 7 del, 26 sub ]
%SER 9.40 [ 47 / 500 ]

===== run.sh script is finished =====
```

11.6 CMU AN4 Data Preparation Script, *prep.sh*

```
# Build organization
## TODO: Remove only if present
echo
echo "---DOWNLOADING CMU ALPHANUMERIC DATA---"
echo

ALPHA_ROOT="/home/mfb33/kaldi/egs/alpha/"
cd ¬/kaldi/egs/
mkdir alpha
cd alpha
mkdir alpha_audio
mkdir alpha_audio/test
mkdir alpha_audio/train
wget ...
    http://www.speech.cs.cmu.edu/databases/an4/an4_raw.bigendian.tar.gz
tar -xvzf an4_raw.bigendian.tar.gz
rm an4_raw.bigendian.tar.gz

# Create audio data
mv an4/wav/an4_clstk/* alpha_audio/train
mv an4/wav/an4test_clstk/* alpha_audio/test

# Convert from RAW to WAV
# 16kHZ sampled, 16 bit
# sox -r 16000 -e unsigned -b 16 -c 1 <RAW_FILE> <TARGET_FILE>
# find alpha_audio/ -maxdepth 3 -type f
for d in $(find alpha_audio/ -maxdepth 3 -type f)
do
  raw="${d}"
  target="${d%%.*}"
  name="${target##*/}" #Everything after
  content="$(cut -d'-' -f1 <<<"${name}")"
  speaker="$(cut -d'-' -f2 <<<"${name}")"
```

```
  suffix="$(cut -d'-' -f3 <<<"${name}")"
  name="${speaker}-${content}-${suffix}"
  path="${target%/*}" #Everything before /
  target="${path}/$name"
  target="${target}.wav"
  sox -r 16000 -e unsigned -b 16 -c 1 $raw $target
  rm $raw
done

echo
echo "---GENERATING ACOUSTIC DATA---"
echo

mkdir data
mkdir data/train
mkdir data/test
mkdir data/local
mkdir data/local/dict


# spk2gender
# TODO: Gender data? (don't really want that)
# <speakerID> <gender>

# wav.scp
# <uterranceID> <full_path_to_audio_file>
rm data/train/wav.scp
touch data/train/wav.scp
for d in $(find alpha_audio/train -maxdepth 2 -type f)
do
    path=$ALPHA_ROOT$d
    name="${d%%.*}"
    name="${name##*/}"
    echo -e "$name $path" >> data/train/wav.scp
done

rm data/test/wav.scp
touch data/test/wav.scp
for d in $(find alpha_audio/test -maxdepth 2 -type f)
do
    path=$ALPHA_ROOT$d
    name="${d%%.*}"
    name="${name##*/}"
    echo -e "$name $path" >> data/test/wav.scp
done

# TEXT
# an4_test.transcription
# an4_train.transcription
rm data/train/text
touch data/train/text
input="an4/etc/an4_train.transcription"
while IFS= read -r line
do
  trans="${line%%</s>*}"
  trans="${trans##*<s>}"
  name="${line%%)*}"
```

```
  name="${name##*(}"
  content="$(cut -d'-' -f1 <<<"${name}")"
  speaker="$(cut -d'-' -f2 <<<"${name}")"
  suffix="$(cut -d'-' -f3 <<<"${name}")"
  name="${speaker}-${content}-${suffix}"
  echo -e "$name $trans" >> data/train/text
done < "$input"

rm data/test/text
touch data/test/text
input="an4/etc/an4_test.transcription"
while IFS= read -r line
do
  trans="${line%%(*}"
  name="${line%%)*}"
  name="${name##*(}"
  content="$(cut -d'-' -f1 <<<"${name}")"
  speaker="$(cut -d'-' -f2 <<<"${name}")"
  suffix="$(cut -d'-' -f3 <<<"${name}")"
  name="${speaker}-${content}-${suffix}"
  echo -e "$name $trans" >> data/test/text
done < "$input"

### utt2spk
# <uterranceID> <speakerID>
rm data/train/utt2spk
touch data/train/utt2spk
for d in $(find alpha_audio/train -maxdepth 1 -type d)
do
    for e in $(find $d -maxdepth 1 -type f)
    do
        folder="${d##*/}"
        name="${e##*/}"
        name="${name%%.*}"
        echo "$name $folder" >> data/train/utt2spk
    done
done

rm data/test/utt2spk
touch data/test/utt2spk
for d in $(find alpha_audio/test -maxdepth 1 -type d)
do
    for e in $(find $d -maxdepth 1 -type f)
    do
        folder="${d##*/}"
        name="${e##*/}"
        name="${name%%.*}"
        echo "$name $folder" >> data/test/utt2spk
    done
done

### corpus.txt
## TODO: Generate more comprehensive corpus
# For now, just copy in
rm data/local/corpus.txt
touch data/local/corpus.txt

```

```
input="an4/etc/an4_train.transcription"
while IFS= read -r line
do
  trans="${line%%</s>*}"
  trans="${trans##*<s>}"
  echo -e "$trans" >> data/local/corpus.txt
done < "$input"

input="an4/etc/an4_test.transcription"
while IFS= read -r line
do
  trans="${line%%(*}"
  echo -e "$trans" >> data/local/corpus.txt
done < "$input"


echo
echo "---GENERATING LANGUAGE DATA---"
echo

## Lexicon <word> <phone 1> <phone 2> ...
rm data/local/dict/lexicon.txt
touch data/local/dict/lexicon.txt
## PROBLEM WITH FORMAT:
# TWENTIETH            T W EH N IY AH TH
# TWENTIETH(2)         T W EH N IY IH TH
# TWENTIETH(3)         T W EH N T IY AH TH
# TWENTIETH(4)
# Can't have duplicates
dict=`cat an4/etc/an4.dic`
echo "$dict" | sed 's/([^()]*)//g' >> data/local/dict/lexicon.txt
echo '<UNK> SIL' >> data/local/dict/lexicon.txt

## Nonsilence_phones.txt
# <phone>
rm data/local/dict/nonsilence_phones.txt
touch data/local/dict/nonsilence_phones.txt
cat an4/etc/an4.phone | grep -v 'SIL' > ...
    data/local/dict/nonsilence_phones.txt

## Silence_phones.txt
rm data/local/dict/silence_phones.txt
touch data/local/dict/silence_phones.txt
echo 'SIL' > data/local/dict/silence_phones.txt

## TODO: OPTIONAL SILENCE?
rm data/local/dict/optional_silence.txt
touch data/local/dict/optional_silence.txt
echo 'SIL' > data/local/dict/optional_silence.txt

## Copy toolkits from wsj
mkdir utils
cp -r ../wsj/s5/utils/* ./utils
mkdir steps
cp -r ../wsj/s5/steps/* ./steps

## Copy scoring script from voxforge
```

```
mkdir local
cp -r ../voxforge/s5/local/score.sh local/score.sh

## Install SRILM (used for this example)
#cd ../..
#cd tools
#./install_srilm.sh
#cd ..
#cd egs/alpha

# Configuration
mkdir conf
touch conf/decode.config
echo "first_beam=10.0
beam=13.0
lattice_beam=6.0" >> conf/decode.config
touch conf/mfcc.conf
echo "--use-energy=false" >> conf/mfcc.conf

## TODO: Choose proper training methods
cp ../digits/cmd.sh ./cmd.sh
cp ../digits/run.sh ./run.sh
cp ../digits/path.sh ./path.sh

# Fix ordering
./utils/fix_data_dir.sh data/test
./utils/fix_data_dir.sh data/train
#   echo "--no-spk-sort means that the script does not require ...
    the utt2spk to be "
#   echo "sorted by the speaker-id in addition to being sorted ...
    by utterance-id."
#   echo "By default, utt2spk is expected to be sorted by both, ...
    which can be "
#   echo "achieved by making the speaker-id prefixes of the ...
    utterance-ids


## ****** See new run script
# Move language model into the tmp folder
```