For this tutorial, we'll be following the Kaldi tutorial for building TIDIGITS. I have stored audio data in data/tidigits. This data is stored in NIST wave file format (which is different than the Microsoft WAV format). The filename of each file gives the transcription, of the format ##...#a.wav or ##...#b.wav. So z12a.wav contains the phrase "zero one two" and 34oa.wav contains "three four oh".

Kaldi is a WFST-based speech recognizer – it builds four different WFST/WFSAs:

H: maps multiple HMM states (a.k.a. transition-ids in Kaldi-speak) to context-dependent triphones
C: maps triphone sequences to monophones
L: maps monophone sequences to words
G: FSA grammar (can be built from an n-gram grammar).

For more detail, see the Kaldi documentation page (kaldi.sourceforge.net) and particularly http://kaldi.sourceforge.net/graph.html

Your job today is to build a digit recognizer, following the script that comes with Kaldi.

Step 0: unpack the tidigits data

```
cd /vagrant/protected
tar -xzf tidigits.tgz
```

Step 1: cd ~/kaldi-trunk/egs/tidigits/s5

Step 2: Look in run.sh . This gives a complete set of commands to go through in building a complete recognizer. While you can run all of these commands just by running ./run.sh, you may find it more instructive to cut and paste commands to see what is going on.

Here's a blow-by-blow of what is going on:

a) Set up some environment variables. If you had a cluster you can submit jobs on the cluster, but for now we'll just run everything locally.

```
. ./cmd.sh
```

b) Set a variable to track where the data is stored:

```
tidigits=/vagrant/protected/tidigits
```

c) Set up some files that are needed. Run this script, then read it to see what's going on:

```
local/tidigits_data_prep.sh $tidigits || exit 1;
```

This creates training and test lists, for example
data/train/spk2utt – lists all of the utterances for a speaker in the training set
data/train/text – gives the transcription for each utterance
data/train/utt2spk – lists the speaker for each utterance
data/train/wav.scp – a script that will create input (via a linux pipe) for each waveform
data/local/data/train.flist – list of training files
data/local/data/train_sph.scp – mapping of each utterance to the corresponding file

There are similar files for the test set.

d) Create the language model:

```
local/tidigits_prepare_lang.sh  || exit 1;
```

Notice that the dictionary is hard-coded in the script, which is a bit unusual for this setup. This creates the fsts for the lexicon and grammar. Read through the script and see what has been created. You should be able to use your OpenFST sleuthing skills.

e) Validate the language models:

```
utils/validate_lang.pl data/lang/
```

As the script says, this will have some errors because the system expects there to be disambiguation symbols. In general, you need to have disambiguation symbols when you have one word be a prefix of another (cat and cats in the same lexicon would need to have cat being pronounced "k ae t #1") or a homophone of another word (red: "r eh d #1", read: "r eh d #2"). If you don't have these then the models become nondeterministic.

f) Create the MFCCs from the wavefile lists. --nj refers to the number of parallel jobs; on a laptop you'll want to reduce this to 2 or 4.

```
mfccdir=mfcc for x in test train; do
        steps/make_mfcc.sh --cmd "$train_cmd" --nj 2 \
                data/$x exp/make_mfcc/$x $mfccdir || exit 1;
        steps/compute_cmvn_stats.sh data/$x exp/make_mfcc/$x $mfccdir || exit 1;
        done
```

This will take a while. Look through the MFCC creation script in the meantime. Note that it calls on a config file in conf/mfcc.conf. Also, let's talk about what cmvn does: this stands for Cepstral Mean and Variance normalization. Many systems often find it useful to subtract the mean cepstral vector and divide each dimension by the variance – this means that the resulting vectors have mean vector<0> and covariance matrix <diagonal<1>> (or, variance vector<1>). The test set will be normalized by a different vector – this can help reduce the mismatches between training and test sets.

g) OK, now that you've read that, an error cropped up, and one file didn't generate any data. So fix it using:

```
utils/fix_data_dir.sh data/test
```

h) The Kaldi system bootstraps by starting with a smaller training set, which you create by doing

```
utils/subset_data_dir.sh data/train 1000 data/train_1k
```

i) The first thing the system does is to build a monophone model from a "flat start" – that is assume that the monophones are all equally likely. It builds from the train_1k dataset. (Remember to adjust the number of jobs.)

```
steps/train_mono.sh  --nj 2 --cmd "$train_cmd" \
        data/train_1k data/lang exp/mono0a
```

While this is running, open up the train_mono.sh script and take a look at it. The first critical step is "gmm-init-mono", which initializes the monophones. The model file is binary, but you can convert it to text by doing the following:

```
~/kaldi/kaldi-trunk/src/bin/copy-transition-model --binary=false
exp/mono0a/0.mdl exp/mono0a/0.txt
```

See if you can piece together what is happening in here. It's a bit complex. The key idea is that multiple passes of EM training are done over the data.

Compare this to the trained model (after it finishes):

```
~/kaldi/kaldi-trunk/src/bin/copy-transition-model --binary=false
exp/mono0a/final.mdl exp/mono0a/final.txt
```

You can also look at the alignments that are created:

```
../../../src/bin/show-alignments data/lang/phones.txt exp/mono0a/final.mdl \
"ark:gunzip -c exp/mono0a/ali.2.gz|" | head -4 | less
```

j) We can then decode the test set using this initial model:

```
utils/mkgraph.sh --mono data/lang exp/mono0a exp/mono0a/graph && \
steps/decode.sh --nj 2 --cmd "$decode_cmd" \
exp/mono0a/graph data/test exp/mono0a/decode
```

k) Once that is done, we can align the entire training set using the “small” model

```
steps/align_si.sh --nj 4 --cmd "$train_cmd" \
data/train data/lang exp/mono0a exp/mono0a_ali
```
l) This alignment is used to train triphones:

```
steps/train_deltas.sh --cmd "$train_cmd" \
300 3000 data/train data/lang exp/mono0a_ali exp/tri1
```

Again, it might be useful to look through the script to see what’s going on. There’s alternate passes of EM training and alignment.

m) Finally, after training, the test set can be decoded using the trigram model:

```
utils/mkgraph.sh data/lang exp/tri1 exp/tri1/graph
steps/decode.sh --nj 2 --cmd "$decode_cmd" \
exp/tri1/graph data/test exp/tri1/decode
```

What the decode does (for both the monophone and triphone cases) is to run the decoding several times with different language model weights. The acoustic model scores, since they have quite a few frames in them, tend to be on a different scale than the language model scores (which are over words). So we scale the probabilities in the language model by multiplying the log probability by a weight in order to bring it into line with the range of the acoustic model scores.

n) Now check out the results – the system computes both a Word Error Rate (WER) and Sentence Error Rate (SER). The easiest way to find the best language model scale is to use a utility script:

```
for x in exp/*/decode*; do [ -d $x ] && grep SER $x/wer_* | utils/best_wer.sh; done
```

which gives:

```
%WER exp/mono0a/decode/wer_20:%SER 2.95 [ 257 / 8699 ]
%WER exp/tri1/decode/wer_35:%SER 1.47 [ 128 / 8699 ]
```

So you can see that this builds a relatively accurate digit recognizer (only 1.47% of digit strings are wrong).