# Running the example scripts (and how Kaldi works)

# Overview of this talk

- Will be going through the process of downloading Kaldi and running the Resource Management (RM) example.
- Will digress where necessary to explain how Kaldi works
- This talk covers the UNIX installation process (installation using Visual Studio is described in the documentation)
- The scripts are in bash (Kaldi can work with any type of shell, but the example scripts are done this way as many people are familiar with this shell)

# Downloading and installing Kaldi

- These instructions also at kaldi.sf.net
- Make sure svn (Subversion) is installed
- This is a version control system, like cvs
- Check out Kaldi:

```
svn co https://kaldi.svn.sourceforge.net/svnroot/kaldi
```

- Instructions in kaldi/trunk/INSTALL
- A few simple steps (run install script; configure; make)… takes a while though
- Installs some tools the scripts require (sph2pipe, IRSTLM, OpenFst, …), plus Kaldi
- e-mail me if it doesn't work!

# What's in the repository

- In kaldi/trunk/ (the "current" version):
  - tools/ (Installation scripts to install external tools)
  - src/ (The Kaldi source code)
    - base/, matrix/, util/, feat/, tree/, optimization/, gmm/, transform/, sgmm/, fstext/, hmm/, lm/, decoder/, bin/, fstbin/, gmmbin/, fgmmbin/, sgmmbin/, featbin/
  - egs/
    - rm/s1/ (Resource Management example dir)
    - wsj/s1/ (Wall Street Journal example dir)

# Building and testing Kaldi

- [once tools/ installation done]
- Change directory to src/
- Configure: "./configure.sh"
  - This hand-written script creates a file "kaldi.mk" invoked by Makefiles in subdirectories
- Make: "make –j 4" [takes a while→ parallel]
  - Programs created in subdirectories *bin/
- Test: "make test"
  - Runs unit-tests that test various components
  - Can also type "make valgrind" (uses valgrind to look for memory errors in unit tests)

# Running the example scripts

- We’ll talk about the Resource Management example script.
- Obtain  LDC corpus LDC93S3A
- Scripts need the directory name where you put this.
- cd to egs/rm/s1, see run.sh
- The following slides will describe the steps in run.sh, and what they do.

# Data preparation

- cd data_prep/; ./run.sh /path/to/RM; cd ..
- Things created by this step:

```
# head train_sph.scp
trn_adg04_sr009 /foo/sph2pipe –f wav \
/bar/adg0_4/sr009.sph |
trn_adg04_sr009 /foo/sph2pipe –f wav \
/bar/adg0_4/sr009.sph |
...
```

- G.txt (bigram decoding graph, in OpenFst text format)

# Data preparation cont'd

- Lexicon in text format (a script will convert this to FST format before being used by Kaldi):

```
# head lexicon.txt
A                ax
A42128           ey f ao r t uw w ah n t uw ey td
AAW              ey ey d ah b y uw
...
```

- Utterance to speaker (utt2spk) maps (will be read directly by Kaldi tools… also spk2utt maps.

```
# head train.utt2spk
trn_adg04_sr009 adg0
trn_adg04_sr049 adg0
...
```

KALDI

# Data preparation cont'd

- Transcriptions in text format
- Note: will be converted to integer format using symbol table, before being used by Kaldi

```
# head train_trans.txt
trn_adg04_sr009 SHOW THE GRIDLEY+S TRACK IN BRIGHT
ORANGE WITH HORNE+S IN DIM RED
trn_adg04_sr049 IS DIXON+S LENGTH GREATER THAN
THAT OF RANGER
...
```

KALDI

# Next steps after data_prep/

- steps/prepare_graphs.sh
- First prepares symbol-tables for words and phones (OpenFst format):

```
# head data/words.txt
<eps>   0
A   1
A42128  2
AAW 3
...
# head data/phones.txt
<eps>   0
aa  1
ae  2
...
```

# Next steps after data_prep/

- Next, this script prepares binary-format FSTs, with integer labels only (no inbuilt symbol tables)
- data/G.fst, data/L.fst, data/L_disambig.fst
- G is the grammar, L is the lexicon.
- The lexicon includes silence.
- L_disambig.fst includes “disambiguation symbols” (search online for hbka.pdf and read Mohri’s paper to find out what these are).

# Next steps after data_prep/

- Also prepares files that contain lists of integer id's of silence and non-silence phones
- These are needed for various purposes by the training and testing scripts

```
# cat data/silphones.csl
48
# cat data/nonsilphones.csl
1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16:17:18:19:20:21:22
:23:24:25:26:27:28:29:30:31:32:33:34:35:36:37:38:39:40:4
1:42:43:44:45:46:47
```

# Computing raw MFCC features

```
mfccdir=/big/disk/mfccdir
steps/make_mfcc_train.sh $mfccdir
steps/make_mfcc_test.sh $mfccdir
```

- An example of the actual command that one of these scripts runs is:

```
compute-mfcc-feats –use-energy=false \
     scp:data/train_wav.scp \
     ark,scp:/foo/raw_mfcc.ark,/foo/raw_mfcc.scp
```

- “ark”==archive, “scp”==script file
- Data goes in single large archive file.

# Script and archive files…

```
# head /foo/raw_mfcc.scp
trn_adg04_sr009 /foo/raw_mfcc.ark:16
trn_adg04_sr049 /foo/raw_mfcc.ark:23395
...
# head –c 20 foo/raw_mfcc.ark
trn_adg04_sr009 ^@BFM [binary data...]
```

- Archive format is [key] [object] [key] [object]…
- Archives may contain binary or text data
- Text archives are often line-by-line (depends on text form of the object).

# Script and archive files…

- Script format is [key] [extended-filename]\n [key] [extended-filename]\n …
- In general, extended filenames include “/file/name”, “some command|”, “|some command”, “-”, “/offset/into/file:12343”
- To understand how we deal with scripts and archive, need to understand the “Table” concept…

# The Table concept

- A Table is a collection of objects (of some known type), indexed by a “key”.
- A “key” is a nonempty, space-free string, e.g. “trn_adg04_sr009” (an utterance), “adg04” (a speaker)
- There is no single C++ class corresponding to a table…
- There are three (templated) Table classes:

```
TableWriter
SequentialTableReader
RandomAccessTableReader
```

# The Table concept + templates

- The Table is templated, but not on the type of object it holds.
- It’s templated on a class we call a “Holder” class, which contains a typedef Holder::T that is the actual type the Table holds.
- E.g. “Int32VectorHolder” is a name of a Holder class.
- The Holder class tells the Table code how to read and write objects of that type
- I.e. it has appropriate Read and Write functions

# The Table concept: example

- Suppose in your program you want to read, sequentially, objects of type std::vector<int32>, indexed by key.
- The user would provide a string (an "rspecifier") that tells the Table code how to read the object.

```
std::string rspecifier = "ark:/foo/my.ark"
SequentialTableReader<Int32VectorHolder>
     my_reader(rspecifier);
for(; !my_reader.Done(); my_reader.Next()) {
   std::string key = my_reader.Key();
   const std::vector<int32> &value(my_reader.Value());
   ... do something ...
}
```

# The Table concept: purpose

- The Table code provides a convenient I/O abstraction (without the need for an actual database).
- Normal Kaldi code interacts with sets of objects (indexed by key) in three ways:
  - Writing keys and objects one by one (TableWriter)
  - Reading keys and objects one by one (SequentialTableReader)
  - Accessing objects with random access (RandomAccessTableReader)… this class will tell you whether a key is in a table or not.

# The Table concept: hard cases

- The Table code always ensures correctness (to do this, it may have to read all objects into memory).
- Note: the three Table classes are actually each polymorphic (implementation differs depending if it’s a script-file or archive, and also other factors).
- The implementation of most cases is fairly simple
- There is one tricky situation: accessing an archive via random access.
- The archive may be a pipe. In this case we can’t fseek(), so would have to cache all the objects in memory in case they’re asked for later.
- Next slide will describe how we deal with this.

# The Table concept: options

- We can provide various options in the "rspecifiers" and corresponding "wspecifiers".
- The simplest one is specifying text-mode, e.g. (for writing) "ark,t:foo.ark"
- Others are to make life easier for the Table code (i.e. enable it to cache fewer objects in memory).
- E.g. "ark,s:foo.ark": "s" asserts that the archive is sorted on key (stops us having to read to the end of the archive if key not present).
- Common option when reading is "s,cs": "s" asserts archive is sorted, "cs" that the keys are queried in sorted order.  Avoids object caching.

# Computing MFCCs (cont'd)

```
compute-mfcc-feats –use-energy=false \
     scp:data/train_wav.scp \
     ark,scp:/foo/raw_mfcc.ark,/foo/raw_mfcc.scp
```

- Here, "`scp:data/train_wav.scp`" is an rspecifier that says to interpret "`data/train_wav.scp`" as a script file to read from
- "`ark,scp:/foo/raw_mfcc.ark,/foo/raw_mfcc.scp`" is a wspecifier that says to write a (binary) archive, and also a script file with offsets into that archive (for efficient random access).

# Monophone training

**`steps/train_mono.sh`**

- This script first sets up some variables…

```
dir=exp/mono
# create $dir/train.scp which is a data subset.
feats="ark:add-deltas scp:$dir/train.scp ark:- |"
```

- The variable `$feats` will be used as a command-line argument to programs, treated as an rspecifier.
- The part after "`ark:`" is treated as an extended filename (and since it ends with "`|`", the  command is invoked and we read from the output).
- The program add-deltas writes to "`ark:-`", i.e. it writes an archive on the standard output.

# Monophone training; topology

- Next, the script creates a file `$dir/topo` which specifies phone topologies.

```
<Topology>
<TopologyEntry>
<ForPhones>
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
46 47
</ForPhones>
<State> 0 <PdfClass> 0 <Transition> 0 0.75 <Transition> 1 0.25
</State>
<State> 1 <PdfClass> 1 <Transition> 1 0.75 <Transition> 2 0.25
</State>
<State> 2 <PdfClass> 2 <Transition> 2 0.75 <Transition> 3 0.25
</State>
<State> 3 </State>
</TopologyEntry>
# Next is the topology entry for silence, which we won't show…
</Topology>
```

# Monophone training; initialiation

- The next command does a flat start of the model

```
gmm-init-mono $dir/topo 39  $dir/0.mdl $dir/tree
```

- This program also creates a “trivial” decision tree with no splits
- Note: monophone system is treated as a special case of a context-dependent system, with zero phones of left and right context.
- Note: in the real scripts we redirect the stderr into log files (all logging on stderr).

# Monophone training: creating decoding graphs

- Write an archive containing the fully expanded FST corresponding to the transcription of each utterance.

```
compile-train-graphs $dir/tree $dir/0.mdl data/L.fst \
    "ark:$dir/train.tra” \
    "ark:|gzip -c >$dir/graphs.fsts.gz"
```

- Note: the input file `$dir/train.tra` would contain transcriptions in integer form, e.g.:

```
trn_adg04_sr009 763 843 367 879 417 75 622 974 407 417
227 694
trn_adg04_sr049 436 235 483 362 841 842 611 679
trn_adg04_sr089 763 345 842 30 88 617 881
...
```

KALDI

# Monophone training: initial alignment

```
align-equal-compiled \
"ark:gunzip –c $dir/graphs.fsts.gz|“  "$feats" ark:- | \
  gmm-acc-stats-ali $dir/0.mdl "$feats“ ark:- dir/0.acc;
```

- The first command in this pipe (`align-equal-compiled`) does an equally-spaced alignment of a random path through each FST.
- Its output is an “alignment” for each utterance
- An alignment is a vector of “transition-ids”, one per frame.
- A transition-id is like the index of a p.d.f., but with a bit more information encoded in it (the phone, etc.)
- The program `gmm-acc-stats-ali` accumulates stats for GMM training, given alignments.

```
gmm-est $dir/0.mdl $dir/0.acc $dir/1.mdl;
```

# Monophone training

- On selected iterations of training, re-align training data (Viterbi alignment):

```
gmm-align-compiled –beam=8 --retry-beam=40 \
   $dir/$x.mdl "ark:gunzip -c $dir/graphs.fsts.gz|“ \
  "$feats" ark,t:$dir/cur.ali
```

- Realign almost every iteration during monophone phase
- Typically only about 3-4 times during triphone training.
- Mixing-up is an option to the update program.
- Gaussians allocated according to an overall budget we provide, proportional to $\gamma^{0.2}$,where $\gamma$ is the data count
- E.g.: increase this budget linearly for 15 iterations, then leave it fixed for another 15.

KALDI

# Triphone training

- First stage is to align all the data with monophone model

```
gmm-align --beam=8 --retry-beam=40 exp/mono/tree \
    exp/mono/30.mdl data/L.fst "$feats" \
   ark:data/train.tra ark:exp/tri1/0.ali
```

- Next accumulate stats for training the decision tree:

```
acc-tree-stats  --ci-phones=48 exp/mono/30.mdl \
   "$feats" ark:exp/tri/0.ali exp/tri/treeacc
```

# Triphone training: questions etc.

- Automatically generate sets of phones that will be “questions”, via tree clustering (do binary splitting of phones, and get questions of all sizes).

```
cat data/phones.txt | awk '{print $NF}' | \
   grep -v -w 0 > exp/tri/phones.list
cluster-phones exp/tri/treeacc exp/tri/phones.list \
  exp/tri/questions.txt
compile-questions exp/tri/topo exp/tri/questions.txt \
  exp/tri/questions.qst
```

- Create file that specifies tree “roots”: in this case, one per phone (but could have shared roots).

```
scripts/make_roots.pl --separate data/phones.txt  \
  $silphonelist shared split \
      > exp/tri/roots.txt
```

# Triphone training: building tree

- Build the decision tree

```
build-tree --max-leaves=1500 exp/tri/treeacc \
    exp/tri/roots.txt exp/tri/questions.qst \
    exp/tri/topo exp/tri/tree
```

- Initialize the model for this tree

```
gmm-init-model  exp/tri/tree exp/tri/treeacc \
     exp/tri/topo exp/tri/1.mdl
```

- Convert alignments generated from the monophone system to be consistent with the new tree:

```
convert-ali exp/mono/30.mdl exp/tri/1.mdl \
  exp/tri/tree ark:exp/tri/0.ali ark:exp/tri/cur.ali
```

- Rest of training similar to monophone case.

KALDI

# Decoding: building the graph

- Script to build graph is invoked by:

```
scripts/mkgraph.sh exp/tri/tree exp/tri/30.mdl \
   exp/graph_tri
```

- Compose L (lexicon) with G (grammar), determinize, minimize

```
fsttablecompose data/L_disambig.fst data/G.fst | \
     fstdeterminizestar --use-log=true | \
  fstminimizeencoded  > exp/tri/LG.fst
```

- Get list of disambiguation symbols:

```
grep '#' data/phones_disambig.txt | \
  awk '{print $2}' > $dir/disambig_phones.list
```

- This file now contains “49\n50\n51\n”.

# Decoding: building the graph, cont'd

- Compose (dynamically generated) C with LG:

```
fstcomposecontext \
  --read-disambig-syms=$dir/disambig_phones.list \
  --write-disambig-syms=$dir/disambig_ilabels.list \
   $dir/ilabels < $dir/LG.fst >$dir/CLG.fst
```

- The input symbols of CLG.fst represent context-dependent phones. [note: command above defaults to trigram]
- The file $dir/ilabels contains the information that maps these symbol id's to phonetic-context windows.
- Next command generates the "H" tranducer...
- actually Ha is H without self-loops.

```
make-h-transducer --disambig-syms-out=$dir/tstate.list \
  $dir/ilabels exp/tri/tree exp/tri/model \
   > $dir/Ha.fst
```

# Decoding: building the graph, cont'd

- … the transducer Ha.fst has "transition-ids" as its input symbols and context-dependent phones as output.
- transition-ids are like p.d.f. indexes, but also guarantee to encode the phone, HMM-position etc.
- Compose Ha with CLG, determinize, remove disambiguation symbols , remove epsilons, minimize:

```
fsttablecompose $dir/Ha.fst $dir/CLG2.fst | \
  fstdeterminizestar --use-log=true \
  | fstrmsymbols $dir/tstate.list | fstrmepslocal | \
   fstminimizeencoded > $dir/HCLGa.fst
```

- Add self loops to get final graph:

```
add-self-loops exp/tri/30.mdl \
    < $dir/HCLGa.fst > $dir/HCLG.fst
```

# Decoding: decoding command

- First set up the features variable (shell variable)

```
feats="ark:add-deltas scp:data/test_feb89.scp ark:- |"
```

- Decode:

```
gmm-decode-faster --beam=20.0 --acoustic-scale=0.08333 \
    --word-symbol-table=data/words.txt exp/tri/30.mdl \
   exp/graph_tri/HCLG.fst “$feats” \
   ark,t:exp/decode_tri/test_feb89.tra \
   ark,t:exp/decode_tri/test_feb89.ali
```

- Note: this command outputs state-level traceback
- We can use this to compute transforms
- Decode again with separate command.

# Summary (scripts)

- Have described the simplest path through the scripts
- Have summarized some of Kaldi's I/O mechanisms
- Have given some idea of how training and decoding works in Kaldi