Speech REcognition

- a practical guide
Lecture 3

Phonetic Context Dependency
## Steps covered in this lecture

<table>
<thead>
<tr>
<th>Steps covered in this lecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aligning data with monophone system</td>
</tr>
<tr>
<td>Training triphone system</td>
</tr>
</tbody>
</table>

```bash
$ cd ~/kaldi-trunk/rm/s3/

$ # Get alignments from monophone system.
$ steps/align_deltas.sh data/train data/lang exp/mono exp/mono_ali

$ # train tril [first triphone pass]
$ steps/train_deltas.sh data/train data/lang exp/mono_ali exp/tril
```
Phones “sound different” in different contexts.

Most strongly affected by phones immediately before/after.

Simplest model of context dependency is to build separate model per “triphone” context.

For 38 phones, #models required is $38 \times 38 \times 38$

Too many models to train!
Traditional context-dependency tree

- Build a “decision tree” for each “monophone”
- This follows the “Clustering and Regression Tree” (CART) framework
  - Involves a “greedy” (locally optimal) splitting algorithm.
- Ask questions like “Is the left phone a vowel?”
  - Is the right phone the phone “sh”? 
- Models (HMMs) would correspond to the leaves
Square boxes correspond to Hidden Markov Models
Train a monophone system (or use previously built triphone system) to get time alignments for data.

For each seen triphone, accumulate sufficient statistics to train a single Gaussian per HMM state.

Suff. stats for Gaussian are (count, sum, sum-squared).

Total stats size (for 39-dim feats):

\[(38 \times 38 \times 38) \times (3 \text{ HMM-states}) \times (39 + 39 + 1)\]
Building the triphone system

Align more of the training data using the monophone model.

We saw the alignment format last time (sequences of integers)

Note: exp/mono has alignments too, but need it on more data (and with very last model).
Getting data alignments

Alignments go in file “exp/mono.ali.ali”

--beam=8, --retry-beam=40 important options

For Viterbi pruning

If don’t reach end-state with beam=8, retry with beam=40, then give up.

$ cd ~/kaldi-trunk/egs/rm/s3
$ head -1 exp/mono.ali/align.log
gmm-align --transition-scale=1.0 --acoustic-scale=0.1 --self-loop-scale=0.1 --beam=8 --retry-beam=40 exp/mono.ali/tree exp/mono.ali/
Building the triphone system

Rest of lecture will be about this stage.
Getting stats for tree

Double-precision stats.

Max. possible size is $38^3 \times 3 \times (39+39+1) \times 8 = 104M$

Actual size 14M (~15% of triphones seen).
Looking at tree stats

This data is for HMM-state 0, phonetic context (0,10,22), = "<eps>/d/ih" (as in "DID" at start of sentence).

Count of 1st state is 10

Next two lines are data sum, sum-squared.
Looking at tree stats

There are 19268 states with stats (this is 3x #seen triphones).

Object in yellow/orange/red is “Event Vector”

Consider as a set of key-value pairs.

The 0.01 is a variance floor (in here for C++ reasons, although shared among all stats..)
Fire a Linguist

The late Fred Jelinek (founding director of the CLSP)

Reported to have said, “Every time I fire a linguist, the error rate goes down.”

Apparently he insisted this had been taken out of context...

Fred championed the “purely statistical” approach to speech recognition.

In the same spirit, in Kaldi we avoid the use of “meaningful” hand-generated phonetic questions.
We cluster the phones to get questions.

A question is just a set of phones.

Would normally be a phonetic category.

Here, just clusters based on acoustic similarity.

Tree clustering -> hierarchy of sets of all sizes.
Some of the smaller sets look meaningful.

Not all do, e.g. “aa” and “f” are not similar

Questions in next stage (tri2a) are a bit better

Presumably, monophone alignments are poor quality.
This clustering algorithm only used for a small part of the system (getting the questions)...

but useful introduction to Kaldi’s framework for clustering and decision trees.

Abstract C++ interface “ClusterableInterface”

Represents some kind of stats and associated model type, from which we can get an objective function

Stats can be added together.
Clustering routines act on generic objects satisfying this interface.

In our stats, \texttt{Objf()} returns a Gaussian likelihood.

We'll split at the root to maximize the data likelihood, then split each branch...
GaussClusterable represents statistics for a Gaussian distribution.

Contains count, sum and sum-squared of data (as in the tree-stats we saw).
the Add() function is very simple-- just add the stats together.
Getting likelihood

Compute the variance of the Gaussian

Return the expected likelihood, times the count...

Note: we apply a variance floor (otherwise the likelihood can go to infinity).
Clustering code

Note: `phone_sets` and `phone_sets_out` both of type `vector<vector<int32>>`

`phone_sets` is just a single vector containing all the phones, in our example

A mechanism to let you keep some phone sets together through clustering.
stats is of type BuildTreeStatsType

- vector of pair<EventVector, ClusterableInterface*>  
  
- EventVector specifies phone, context, etc.

- pdf_id_list is by default a vector containing just "1"... specifies to use only middle HMM-state's stats to cluster.
Clustering code

Note on coding style:

- Variables for function outputs are passed by pointer, and come after input parameters.
- Style guide (derived from Google style guide) dictates this.
Looking at code of `AutomaticallyObtainQuestions`

Call to `FilterStatsByKey` keeps only stats from HMM-state 1 (middle HMM-state)... configurable via `all_pdf_classes` variable.

Note: `kPdfClass` is an enum that evaluates to -1... this is the EventMap “key” for “pdf-class” which is normally synonymous with HMM-state.
Next statement splits stats up according to the central phone (monophone).

Note: variable \( P \) (c.f. command-line option \(--central-position\)) is the center of context-window of phonemes.

Value of \( P \) is 1 for triphone, 0 for monophone.
Next statement sums up all the stats for each phoneme (over all contexts, HMM-positions).

Type of stats is now `vector<Clusterable*>`

I.e. we don’t have the `EventVector` any more, that specifies context etc.

Phone is just index into vector.
Here is the main call to tree-clustering routine.

```c++
void AutomaticallyObtainQuestions( <snip> ) {
    ...
    ...
    ...
    TreeClusterOptions topts;
    topts.kmeans_cfg.num_tries = 10;  // This is a slow-but-accurate setting,
    // we do it this way since there are typically few phones.

    std::vector<int32> assignments;  // assignment of phones to clusters. dim ==
    summed_stats.size().
    std::vector<int32> clust_assignments;  // Parent of each cluster. Dim == #clusters.
    int32 num_leaves;  // number of leaf-level clusters.
    TreeCluster(summed_stats_per_set,
                summed_stats_per_set.size(),  // max-#clust is all of the points.
                NULL,  // don't need the clusters out.
                &assignments,
                &clust_assignments,
                &num_leaves,
                topts);
    ...
}
```
Compiling the questions

Program “compile-questions” takes lists of phonemes...

Transforms it into a C++ object (written to disk) that contains questions for each “key” in EventMap

Sets up questions about HMM-state (0,1,2)...

Here, some options can be set that affect tree-building.
Set up sets of phones with “shared roots” for trees

In this case, all phones have separate tree root

If phones only differ in stress or tone etc., can be useful to share roots

This way, unseen variants still get a model.
Note: integers correspond to phones; in general, can have lists of integers (to share roots).

**shared** means HMM-states 0,1,2 share a root (can ask questions about HMM-state/pdf-id).

**split** means we build a decision tree for this root (else, leave an un-split stub).
Actually a set of decision trees (one per root)

The max-leaves (e.g., 2000) is number of p.d.f's

Some post-clustering done within each tree, after splitting.

This shares leaves, but only within each tree (e.g. per phone, not globally)
Building the decision tree

**Likelihood improvement from tree splitting** is an important diagnostic.

**More likelihood improvement** is generally better (means context is helping more).
Drawing the decision tree

```bash
$ . path.sh
$ draw-tree data/lang/phones.txt exp/tri1/tree | dot -Tps -Gsize=8,10.5 | ps2pdf ~/tree.pdf
```
Drawing the decision tree
Drawing the decision tree
Drawing the decision tree
Decision tree leaves

- Decision tree leaves are integers (no names!)
- We call these “pdf-ids”.
- They are zero-based (i.e. numbered from zero)
  - Caution: some integer identifiers in Kaldi are one-based (e.g. “transition-ids”)
- This is for compatibility with OpenFst, where zero is “special”.
- Where possible we prefer zero-based indexing.
Decision tree -- differences from “standard” approach

- Can share tree roots among phones
- Same tree root for all the HMM-states of a phone (or phone-set)
  - Ask questions about HMM-state.
- Automatically obtained questions
- Leaves post-clustered after tree splitting
Decision tree object: type “ContextDependency”

Written to file called “tree”

Maps from [window of phones, HMM-state] to integer pdf-id.

E.g. (aa/n/d, 3) -> 1402
Initializing the model

This program reads the tree, tree accumulators, and topology.

It outputs the model file 1.mdl
Write two objects to the model file.

"trans_model" (type: TransitionModel)

"am_gmm" (type: AmDiagGmm)

Some programs that read the model file, only read the **TransitionModel** object.

This makes them model-type independent.
Note: "Output" object opens a generalized filename (works with files, stdin/stdout, piped commands)

The Write functions of Kaldi objects take an ostream, and a bool (for binary/text mode)

Idea is to make code easily refactorable (if they took the "Output" object, too Kaldi-dependent.)
Object **DiagGmm** is a single Gaussian Mixture Model

Parameters stored in “exponential model” form for fast likelihood evaluation

Convert to **DiagGmmNormal** for easy updates etc. (this stores them more conventionally).
Object AmDiagGmm contains a vector of DiagGmm
Indexed by “pdf-id” (remember, this is a zero-based integer index).
This object knows nothing about transitions, topology, etc.
Object **TransitionModel** responsible for storing HMM transition probabilities

Also keeps track of HMM topologies (contains the **HmmTopology** object)

Defines “transition-ids”-- an index that corresponds with transition probs, and useful for other reasons.
Transition parameters

- The decision-trees are at the individual state level, not the whole-HMM level.

- Therefore there may be many more HMMs than decision-tree leaves (combinatorial explosion)

- We tie the transition-model parameters the same way as the decision-tree leaves

  - although if the monophone/center phone is different, we have a different transition-prob.

- Transitions out of a given state are tied like that state.
End of this lecture