Josh's Kaldi Documentation

This documentation is a work in progress.
Last update: December 1, 2016

Most of what is presented here is stitched together directly from the official Kaldi documentation at http://kaldi-asr.org/doc/. As such, the credit for the content here mainly goes to Dan Povey and the other Kaldi folks who wrote up the original documentation. I've tried here to put the information together in a way that made sense to me, adding in a few of my own interpretations and information from other blogs and some of the seminal papers these techniques are based on.

As such, most of the credit for this goes to the Kaldi team, HOWEVER, if there are errors here, they are most likely my own faults, and not those of the Kaldi folks.

My intention is for this documentation to be read along with one of the `run.sh` scripts from the Kaldi `egs` directory. These scripts are all different, and I don't cover any of the data preparation here. As such, this is a rough guide, but I hope it can be useful.

Feature Extraction

- Mel-Feature Cepstra Coefficient Computation

  - compute-mfcc-feats.cc

    This program requires two command-line arguments: an rspecifier to read the .wav data (indexed by utterance) and a wspecifier to write the computed MFCC features (indexed by utterance).

    In typical usage, we will write the data to one big "archive" file and also write out an "scp" file for easy random access; see Writing an archive and a script file simultaneously for explanation. Shown here is only the option to write to an ark file. The program does not add delta features (for that, see `add-deltas.cc`).

    It accepts an option `--channel` to select the channel (e.g. `--channel=0`, `--channel=1`), which is useful when reading stereo data.

    ```
    compute-mfcc-feats --config=conf/mfcc.conf \
                       scp:exp/make_mfcc/train/wav1.scp \
                       ark:/data/mfcc/raw_mfcc_train.1.ark;
    ```

    The first argument "scp:..." tells it to find filenames (actually via some commands) in the file `exp/make_mfcc/train/wav1.scp`.

    The second argument is usually "ark,scp:...", which tells it to write an archive ('ark' file), and also an index into that archive ('scp' file). This archive file will contain a feature matrix of size `N-frames X N-mfccs`, for each utterance.

    The computation of MFCC features is done by an object of type MFCC, which has a function `Compute()` to compute the features from the waveform.

    The overall MFCC computation is as follows:

    1. Work out the number of frames in the file (typically 25ms frames shifted by 10ms each time).
    2. For each frame:
       (a) Extract the data, do optional dithering, preemphasis and dc offset removal, and multiply it by a windowing function (various options are supported here, e.g. Hamming)

(b) Work out the energy at this point (if using log-energy not C0).
(c) Do FFT and compute the power spectrum
(d) Compute the energy in each mel bin; these are e.g. 23 triangular overlapping bins whose centers are equally spaced in the mel-frequency domain.
(e) Compute the log of the energies and take the cosine transform, keeping as many coefficients as specified (e.g. 13)
(f) Optionally do cepstral liftering; this is just a scaling of the coefficients, which ensures they have a reasonable range.

The lower and upper cutoff of the frequency range covered by the triangular mel bins are controlled by the options `--low-freq` and `--high-freq`, which are usually set close to zero and the Nyquist frequency respectively, e.g. `--low-freq=20` and `--high-freq=7800` for 16kHz sampled speech.

The features differ from HTK features in a number of ways, but almost all of these relate to having different defaults. With the option `--htk-compat=true`, and setting parameters correctly, it is possible to get very close to HTK features. One possibly important option that we do not support is energy max-normalization. This is because we prefer normalization methods that can be applied in a stateless way, and would like to keep the feature computation such that it could in principle be done frame by frame and still give the same results. The program `compute-mfcc-feats.cc` does, however, have an option `--subtract-mean` to subtract the mean of the features. This is done per utterance; there are different ways to do it per speaker (e.g. search for "cmvn", meaning cepstral mean and variance normalization, in the scripts).

– COPY-FEATS.CC

This program will copy features and possibly change their format.

- CEPSTRAL MEAN AND VARIANCE NORMALIZATION

Cepstral mean and variance normalization consists of normalizing the mean and variance of the raw cepstra, usually to give zero-mean, unit-variance cepstra, either on a per-utterance or per-speaker basis. We provide code to support this, and some example scripts, but we do not particularly recommend its use. In general we prefer model-based approaches to mean and variance normalization; e.g., our code for Linear VTLN (LVTLN) also learns a mean offset and the code for Exponential Transform (ET) does a diagonal CMLLR transform that has the same power as cepstral mean and variance normalization (except usually applied to the fully expanded features). For very fast operation, it is possible to apply these approaches using a very tiny model with a phone-based language model, and some of our example scripts demonstrate this. There is also the capability in the feature extraction code to subtract the mean on a per-utterance basis (the `--subtract-mean` option to `compute-mfcc-feats.cc` and `compute-plp-feats.cc`). In order to support per-utterance and per-speaker mean and variance normalization we provide the programs `compute-cmvn-stats.cc` and `apply-cmvn.cc`.

These programs, despite the names, do not care whether the features in question consist of cepstra or anything else; it simply regards them as matrices. Of course, the features supplied to `compute-cmvn-stats.cc` and `apply-cmvn.cc` must have the same dimension.

We note that it would probably be more consistent with the overall design of the feature transformation code, to supply a version of `compute-cmvn-stats.cc` that would write write out the mean and variance normalizing transforms as generic affine transforms (in the same format as CMLLR transforms), so that they could be applied by the program `transform-feats.cc`, and composed as needed with other transforms using `compose-transforms.cc`. If needed we may supply such a program, but because we don't regard mean and variance normalization as an important part of any recipes, we have not done so yet.

- COMPUTE-CMVN-STATS.CC

  The program `compute-cmvn-stats.cc` will, by default, compute the sufficient statistics for mean and variance normalization, as a matrix (the format is not very important; see the code for details), and will write out a table of these statistics indexed by utterance-id. If it is given the `--spk2utt` option, it will write out the statistics on a per-speaker basis instead (warning: before using this option, read Avoiding memory bloat when reading archives in random-access mode, as this option causes the input features to be read in random-access mode).

```
compute-cmvn-stats --spk2utt=ark:data/train.1k/spk2utt \
                   scp:data/train.1k/feats.scp \
                   ark:exp/mono/cmvn.ark;
```

- APPLY-CMVN.CC

  The program `apply-cmvn.cc` reads in features and cepstral mean and variance statistics; the statistics are expected to be indexed per utterance by default, or per speaker if the `--utt2spk` option is applied. It writes out the features after mean and variance normalization.

Monophone Training

- INITIALIZE MONOPHONE MODEL

  - GMM-INIT-MONO.CC

    This program takes in two inputs and returns two outputs. As input, we need (1) a topology file which describes the structure of the HMM phones of the acoustic model (e.g. `data/lang/topo`) and (2) the number of dimensions of each Gaussian mixture component (e.g. 39).

    As output, we get back (1) a GMM-HMM model (e.g. `0.mdl`) and (2) a phonetic decision tree (e.g. `tree`).

    For example:

```
gmm-init-mono data/lang/topo \
              39 \
              exp/mono/0.mdl \
              exp/mono/tree;
```

    In this case, `tree` is phonetic-context decision tree which doesn't have any splits because it is a monophone model.

  - COMPILE-TRAIN-GRAPHS.CC

    Suppose we have built a tree and model. The following command creates an archive (c.f. The Table concept) that contains the graph $HCLG$ corresponding to each of the training transcripts. That is, this program compiles FSTs, one for each training utterance.

```
compile-train-graphs exp/mono/tree \
                     exp/mono/0.mdl \
                     data/L.fst \
```

```
ark:data/train.tra \
ark:exp/mono/graphs.fsts;
```

The input file `train.tra` is a file containing integer versions of the training transcripts, e.g. a typical line might look like the following line, where the first token of the line (`011c0201`) is the utterance id.

```
011c0201 110906 96419 79214 110906 52026 55810 82385 79214 51250
```

The output of the `compile-train-graphs.cc` program is the archive `graphs.fsts`; it contains an FST (in binary form) for each utterance id in `train.tra`. The FSTs in this archive correspond to $HCLG$, except that there are no transition probabilities (by default, `compile-train-graphs.cc` has `--self-loop-scale=0` and `--transition-scale=0`). This is because these graphs will be used for multiple stages of training and the transition probabilities will change, so we add them in later. But the FSTs on the archive will have probabilities that arise from silence probabilities (these probabilities are encoded into L.fst), and if we were using pronunciation probabilities these would be present too. The overall picture for decoding-graph creation is that we are constructing the graph $HCLG = H \circ C \circ L \circ G$.

1. H contains the HMM definitions; its output symbols represent context-dependent phones and its input symbols are transition-ids, which encode the pdf-id and other information (see Integer identifiers used by TransitionModel).
2. C represents the Context-Dependency: its output symbols are phones and its input symbols represent context-dependent phones, i.e. windows of N phones (see Phonetic context windows).
3. L is the Lexicon; its output symbols are words and its input symbols are phones.
4. G represents a Grammar; it is a finite-state acceptor which encodes the language model.

Our graph creation recipe during training time is simpler than during test time, mainly because we do not need the disambiguation symbols. In training time we use the same $HCLG$ formalism as in test time, except that G consists of just a linear acceptor corresponding to the training transcript (of course, this setup is easy to extend to cases where there is uncertainty in the transcripts).

This is the standard recipe. However, there are a lot of details to be filled in. We want to ensure that the output is determinized and minimized, and in order for $HCLG$ to be determinizable we have to insert disambiguation symbols. For details on the disambiguation symbols, see Disambiguation symbols.

We also want to ensure the $HCLG$ is stochastic, as far as possible; in the conventional recipes, this is done (if at all) with the "push-weights" operation. Our approach to ensuring stochasticity is different, and is based on ensuring that no graph creation step "takes away" stochasticity; see Preserving stochasticity and testing it for details.

If we were to summarize our approach on one line (and one line can't capture all the details, obviously), the line would probably as follows, where $asl ==$ "$add\ self\ loops$" and $rds ==$ "$remove\ disambiguation\ symbols$", and H' is H without the self-loops:

$$HCLG = asl(min(rds(det(H' \circ min(det(C \circ min(det(L \circ G))))))))) \quad (1)$$

Weight-pushing is not part of the recipe; instead we aim to ensure that no stage of graph creation will stop the result from being stochastic, as long as G was stochastic. Of course, G will typically not be quite stochastic, because of the way Arpa language models

with backoff are represented in FSTs, but at least our approach ensures that the non-stochasticity "stays put" and does not get worse than it was at the start; this approach avoids the danger of the "push-weights" operation failing or making things worse.

- Train Monophone Model
  - align-equal-compiled.cc

    Given (1) an acoustic model, (2) an rspecifier for graphs, (3) an rspecifier for features, this program will return an wspecifier for alignments. This is the E-Step in the EM training algorithm (i.e. hidden state alignment to the audio input). This program will produce alignments equally spaced for each utterance. This equally spaced alignment only happens at the very first stage in initialization, before any model parameters have been trained. An alignment is a vector of intergers (per utterance).

```
align-equal-compiled 0.mdl \
                     graphs.fsts \
                     scp:train.scp \
                     ark:equal.ali;
```

    If you want to inspect alignments, the program `show-alignments.cc` displays the alignments in a more readable format, with phones.

  - gmm-acc-stats-ali.cc

    There are three inputs for `gmm-acc-stats-ali.cc`: (1) a compiled acoustic model (e.g. `0.mdl`), (2) the features from the training audio files (e.g. MFCC features saved in `train.scp`), and (3) the alignments of hidden states to audio previously computed (e.g. `equal.ali`). The output is a file of accumulated stats for GMM training (e.g. `0.acc`).

```
gmm-acc-stats-ali 0.mdl \
                  scp:train.scp \
                  ark:equal.ali \
                  0.acc;
```

  - gmm-est.cc

    This is the M-Step in the EM Algorithm. Given (1) an acoustic model and (2) a file of accumulated statistics for GMM training, this program will output a new acoustic model which has been updated via Maximum Likelihood Re-Estimation.

```
gmm-est --min-gaussian-occupancy=3 \
        --mix-up=250 \
        exp/mono/0.mdl \
        exp/mono/0.acc \
        exp/mono/1.mdl;
```

    The option `--mix-up` will increase number of mixture components to this overall target.

    When working with a small amount of data, the `--min-gaussian-occupancy` option is important, otherwise you may fail to estimate "rare" phones and later on, they never align properly.

- Align Monophones with Data

  - gmm-align-compiled.cc

    Given (1) an acoustic model, (2) an rspecifier for graphs, (3) an rspecifier for features, this program will return an wspecifier for alignments. This is the E-Step in the EM training algorithm (i.e. hidden state alignment to the audio input). These alignments are between HMM states and feature vectors. Each HMM state has an output Gaussian distribution, and the feature vectors assigned to that distribution are then used to update the Gaussian parameters (i.e. μ and Σ).

```
gmm-align-compiled 1.mdl \
                   ark:graphs.fsts \
                   scp:train.scp \
                   ark:1.ali;
```

Triphone Training

- Build Phonetic Decision Tree for Context-Dependent Triphones

  In this stage we will build a decision tree for each monophone. This follows the "Clustering and Regression Tree" (CART) framework and involves a "greedy" (locally optimal) splitting algorithm.

  The final decision tree will cluster contexts by asking questions like "Is the left phone a vowel?" or "Is the right phone the phone sh"? Models (HMMs) would correspond to the leaves.

  The basic algorithm that is being implemented is a top-down greedy splitting, where we have a number of ways we can split the data by asking about, say, the left phone, the right phone, the central phone, the state we're in, and so on.

  The algorithm we implement is similar to the standard algorithm, see for example the paper "Tree-based State Tying for High Accuracy Acoustic Modeling" by Young, Odell and Woodland. In this algorithm, we split the data up by asking the locally optimal question, i.e. the question that gives the most likelihood increase, supposing we model the data on each side of the split by a single Gaussian.

  Differences from standard implementations include added flexibility about how to configure the tree roots; the ability to ask questions about the HMM-state and the central phone; and the fact that by default in the Kaldi scripts, the questions are automatically generated by a top-down binary clustering of the data, which means that it is not necessary to supply hand-generated questions.

  Regarding the configuration of the tree roots: it's possible to have all the statistics from all the phones in a single shared group that gets split (including questions about the central phone and the HMM-states), or to have individual phones, or HMM-states of phones, be the tree roots that get split, or to have groups of phones be the tree roots.

  For how to configure it using the standard scripts, see Data preparation. In practice we generally let each tree-root correspond to a "real phone", meaning that we group together all word-position-dependent, tone-dependent or stress-dependent versions of each phone into one group that becomes a tree root.

  Traditionally, to build a phonetic decision tree we need to follow a few steps. First, we need to train a monophone system (or use previously built triphone system) to get time alignments for the audio data. Then, given those alignments, we can accumulate statistics to train a single Gaussian per HMM state for each seen triphone.

The sufficient statistics to compute a Gaussian are (count, sum, and sum-squared). This means that the total statistics size (for 39-dim feats) is (38x38x38) * (3 HMM-states) * (39+39+1).

- ACC-TREE-STATS.CC

  This program takes as input (1) an acoustic model, (2) an rspecifier for the acoustic features, (3) an rspecifier for previously made alignments, and the program returns an accumulation of tree statistics.

  After training a monophone system starting from a flat start, we take the monophone alignments and use the function `AccumulateTreeStats()` (called from `acc-tree-stats.cc`) to accumulate statistics for training the tree.

  This program is not limited to reading in monophone alignments; it works from context-dependent alignments too so we can build trees based on e.g. triphone alignments. The statistics for tree building are written to disk as the type `BuildTreeStatsType` (see Statistics for building the tree). The function `AccumulateTreeStats()` takes the values N and P, as explained in the previous section; the command-line programs will set these by default to 3 and 1 respectively, but this can be overridden using the `--context-width` and `--central-position` options. The program `acc-tree-stats.cc` takes a list of context-independent phones (e.g. silence), but this is not required even if there are context-independent phones; it is just a mechanism to reduce the size of the statistics. For context-independent phones, the program will accumulate the corresponding statistics without the keys corresponding to the left and right phones defined (c.f. Event maps).

  In the following command, like elsewhere in Kaldi scripts, `JOB` represents the number of the job being processed. For example, with my four processors on my laptop, I will run one job on each, yielding `1.treeacc`, `2.treeacc`, `3.treeacc`, and `4.treeacc`.

```
acc-tree-stats final.mdl \
               scp:train.scp \
               ark:JOB.ali \
               JOB.treeacc;
```

- SUM-TREE-STATS.CC

  This program will sum statistics for phonetic-context tree building. The program takes as input (1) multiple `*.treeacc` files, and outputs (1) a file of accumulated tree stats (e.g. `treeacc`).

```
sum-tree-stats treeacc \
               *.treeacc;
```

- CLUSTER-PHONES.CC

  This program will cluster phones (or sets of phones) into sets for various purposes. The program takes as input (1) statistics about the tree (e.g. `treeacc`), (2) the sets of phones to be clustered (e.g. `phonesets.int`), and will return as output (1) a set of questions which can be used to culstered phones (e.g. `questions.int`).

  We cluster the phones to get the questions. A question is just a set of phones. Each question would normally be about a phonetic category. However, we just clusters based on acoustic similarity. Tree Clustering yields a hierarchy of sets of all sizes.

```
cluster-phones treeacc \
               phonesets.int \
               questions.int;
```

– COMPILE-QUESTIONS.CC

This program takes (1) an HMM topology (e.g. `topo`), (2) a list of phonemes (e.g. `questions.int`), and returns as output the list of phonemes transformed into a C++ object (written to disk) that contains questions for each "key" in EventMap (e.g. `phonesets.qst`). The program sets up questions about different HMM-states. Here, some options can be set that affect tree-building.

```
compile-questions data/lang/topo \
                  exp/triphones/questions.int \
                  exp/triphones/questions.qst;
```

– BUILD-TREE.CC

When the statistics have been accumulated we use the program `build-tree.cc` to build the tree. This outputs the tree. The program build-tree requires three things as input: (1) The accumulated tree statistics (e.g. `treeacc`), (2) the questions config (e.g. `questions.qst`), and the roots file (e.g. `roots.int`).

The tree statistics would typically come from the program `acc-tree-stats.cc`; the questions configuration class would be output by the `compile-questions.cc` program, which takes in a topology list of phonetic questions (in our scripts, these are automatically obtained from tree-building statistics by the program `cluster-phones.cc`.

The roots file specifies sets of phones that are going to have shared roots in the decision-tree clustering process, and says for each phone set the following two things: (1) "shared" or "not-shared" says whether or not there should be separate roots for each of the pdf-classes (i.e. HMM-states, in the typical case), or if the roots should be shared. If it says "shared" there will be a single tree-root for all HMM states (e.g. all three states, in a normal topology) ; if "not-shared" there would be (e.g.) three tree-roots, one for each pdf-class.

Also, (2) "split" or "not-split" says whether or not the decision tree splitting should actually be done for the roots in question (for silence, we typically don't split). If the line says "split" (the normal case) then we do the decision tree splitting. If it says "not-split" then no splitting is done and the roots are left un-split.

This program builds actually a set of decision trees (one per root). The max-leaves (e.g., 2000) is the number of pdfs. Some post-clustering is done within each tree, after splitting. This shares leaves, but only within each tree (e.g. per phone, not globally).

```
build-tree treeacc \
           roots.int \
           questions.qst \
           topo \
           tree;
```

You can use `draw-tree.cc` to view any decision tree. For example:

```
draw-tree data/lang/phones.txt \
          exp/mono/tree | \
          dot -Tps -Gsize=8,10.5 | \
          ps2pdf - ~/tree.pdf
```

- INITIALIZE TRIPHONE MODEL

– GMM-INIT-MODEL.CC

Initialize a GMM acoustic model (e.g. `1.mdl`) from (1) a decision tree (e.g. `tree`), (2) accumulated tree stats (e.g. `treeacc`), and (3) an HMM model topology (e.g. `topo`).

```
gmm-init-model tree \
               treeacc \
               topo \
               1.mdl;
```

– GMM-MIXUP.CC

Does GMM mixing up (and Gaussian merging). Given as input (1) a GMM acoustic model (e.g. `1.mdl`), (2) per-transition-id occupation counts (e.g. `1.occs`), this program will return a new GMM acoustic model with an increased number of Gaussian components (e.g. `2.mdl`).

```
gmm-mixup --mix-up=$numgauss \
          1.mdl \
          1.occs \
          2.mdl;
```

– CONVERT-ALI.CC

Convert alignments from one decision-tree/model to another. Given (1) an old GMM model (e.g. `monophones_aligned/final.mdl`), some new GMM model (e.g. `triphones_del/2.mdl`), a new decision tree (e.g. `triphones_del/tree`), and an rspecifier for a set of old alignments (e.g. `monophones_aligned/ali.*.gz`, this program will return (1) a set of new alignments (e.g. `triphones_del/ali.*.gz`.

```
convert-ali monophones_aligned/final.mdl \
            triphones_del/2.mdl \
            triphones_del/tree \
            monophones_aligned/ali.*.gz \
            triphones_del/ali.*.gz \
```

– COMPILE-TRAIN-GRAPHS.CC

Creates training graphs (without transition-probabilities, by default). More detail on this program is given above in monophone training.

Given as input (1) a decision tree (e.g. `tree`), (2) an acoustic model (e.g. `2.mdl`), (3) a finite state transducer (FST) for the lexicon (e.g. `L.fst`), (4) an rspecifier for the training data transcriptions (e.g. `text`), this program will return as output (1) an wspecifier for training graphs (e.g. `fsts.*.gz`).

```
compile-train-graphs tree \
                     1.mdl \
                     L.fst  \
                     text \
                     fsts.*.gz;
```

- TRAIN TRIPHONE MODEL

The following three programs used to train a triphone model are identical to those used in monophone training described above. Together, these three steps are iterated recursively, generating alignments (E-Step), and updating model parameters given the newly aligned

feature vectors (M-Step). In general we want to improve our GMM-HMM model in training to make good predictions on future audio input. However, all we have to work with right now is the training data. The kind of Expectation Maximization we're using during training is the Maximum Likelihood Estimation. The likelihood (just like in Bayes Theorem) is the probability of the data given the model. Since we can't change the data, we have to change the model to better account for the data. That's what we're doing here. We use the model to assign itself more accurate data (i.e. feature vectors), and then use that data to update the model.

- GMM-ALIGN-COMPILED.CC
- GMM-ACC-STATS-ALI.CC
- GMM-EST.CC