GMTK Tutorial on Dynamic Graphical Model Training with Gaussian Mixture Unaries, using TIMIT

Richard Rogers <rprogers@uw.edu>
and
Jeff A. Bilmes <bilmes@uw.edu>

September 13, 2014

1 Introduction

The Graphical Models Toolkit (GMTK) is an open source, publicly available software system that allows a researcher or scientist to rapidly prototype, test, learn, and then use statistical models under the framework of dynamic graphical models (DGMs) and dynamic Bayesian networks (DBNs). GMTK can be used for applications and research in speech and language processing, bioinformatics, activity recognition, econometrics, or any type of offline or streaming time series or sequential data application. GMTK source code can be downloaded from http://melodi.ee.washington.edu/gmtk/.

This tutorial document serves to introduce GMTK by describing a standard methodology for training a model based on data. It demonstrates how to train a sequential model that uses Gaussian mixture unary distributions, and begins to demonstrate the flexibility of this framework.

Details on hidden Markov models (HMMs), dynamic Bayesian networks (DBNs), and dynamic graphical models (DGMs), and documentation for GMTK can be found via [4]. The basic GMTK design and work flow is described in Section 14.2 in the GMTK manual.

The tutorial uses speech data (namely the TIMIT corpus [6]), in particular, it describes the process of building a phone recognition system for the TIMIT data set. The methodology presented here, however, is applicable to any type of sequential data. Moreover, the trained model can be used either for offline decoding, or for online decoding using a program such as `gmtkOnline`.

The tutorial covers the following steps:

1. Background and introduction to dynamic graphical models and the speech data that the tutorial uses.
2. Section 5 defines the DGM that the tutorial uses to learn how to recognize speech and explains how to implement and train the model using GMTK.
3. Applying the learned model and evaluating its accuracy are covered in Section 6.
4. Section 7 presents an alternative DGM that takes longer to train but achieves higher accuracy.
5. Section 8 is an exercise in which you modify the speech decoding model defined in Section 6 to include penalties or rewards for hypothesizing more phonemes.
6. Section 9 is another exercise in which you extend your modified speech decoding model to include a time limit on how long a hypothesized phoneme can last.

1.1 Requirements

This tutorial requires that you have a compiled and working version of GMTK, that you are in some form of Unix-like environment (which could be Linux, OSX, or Windows with Cygwin), and that your `TKSRC` environment variable is currently set to the directory where the GMTK executables live (see below). The tutorial also requires a compiled and working version of HTK in order to score results using HTK's `HResults` command (see below).

It is important to look at the file `README.txt` for information on building GMTK and some initial environment setup required to run the tutorial scripts (including some tips for compiling HTK).

GMTK should compile and work in most Unix/POSIX environments that provide a Bourne shell, make, and C/C++ compiler. Linux, OS X, and Windows (requires Cygwin) are specifically supported using GCC 4.5 or later or the Apple Clang compilers. The scripts that implement this tutorial also make use of the unix command `awk`, which should be present on all of the supported platforms.

To compute the accuracy (see Section 6.3) between a reference string and a hypothesized string, you'll need the `HResults` program from HTK [12] that computes things like Levenshtein distance. You can get it from `http://htk.eng.cam.ac.uk`. See `README.txt` for some platform-specific advice on compiling HTK. The `HResults` program must be found under your `PATH` environment variable. Note that only the `HResults` command from HTK is used in this tutorial, nothing else (and we are currently working on a GMTK-specific scoring script but it is not yet ready).

1.2 Environment Setup

The tutorial scripts look for the GMTK executables under the `TKSRC` environment variable, so for Bourne shells

```
$ export TKSRC=/path/to/gmtk/executables
```

or for C shells (such as csh, tcsh),

```
$ setenv TKSRC /path/to/gmtk/executables
```

Note that when you compile GMTK, it compiles the main GMTK executables in the subdirectory `tksrc`, so the path you want is something like `/path/to/gmtk/tksrc`, and there are a few other observation-file manipulation utilities that also do simple signal processing (such as mean subtraction/variance normalization) in `/path/to/gmtk/featureFileIO`.

If you also did a `make install` with the default configuration, the path to the GMTK executables will be `/usr/local/bin`, so set `TKSRC` to that (or whatever directory you installed the GMTK executables to). Alternatively, you can run `make prefix=/path/to/where/you/want install` and then all GMTK programs will be installed in `/path/to/where/you/want/bin` which is what you should add to your path and set the `TKSRC` variable to.

Several of the scripts will launch a small number of worker processes to take advantage of parallel processing, assuming you have a multicore CPU. Set the `NUMPROCS` environment variable to the number of parallel processes to launch. The default is 2.

2 Quick mathematical background

This section briefly describes the task mathematically and introduces some notation. Full details can be found in [4].

The basic goal of this tutorial is to show how GMTK can automatically learn to map from a sequence of T observations, $\bar{x}_{1:T} = (\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_T)$ to a sequence of T labels $y_{1:T} = (y_1, y_2, \ldots, y_T)$ based on training data. The training data is represented as $\mathcal{D} = \left\{(\bar{x}^{(i)}_{1:T_i}, y^{(i)}_{1:T_i})\right\}_{i=1}^{D}$ consisting of D pairs of sequences, input sequences $\bar{x}^{(i)}_{1:T_i}$ and output sequences $y^{(i)}_{1:T_i}$ where the i^{th} sequence has length T_i. For example, in this tutorial, T_i (the utterance or segment length) will typically be different for each training and testing utterance — T_i is the number of time frames (or time slices) of speech. In TIMIT, the frame rate is 100Hz, and each $\bar{x}^{(i)}_t \in \mathbb{R}^N$ is an N-dimensional known vector of real-valued features. Such vectors are called "observed" random variables, since while they are random, their values are known and hence observed.

A hidden Markov model is a probabilistic model over such pairs of sequences. That is, an HMM offers a probability distribution over $\Pr(y_{1:T}, \bar{x}_{1:T})$ for any length T. To be an HMM, this distribution must factorize in the following way:

$$\Pr(y_{1:T}, \bar{x}_{1:T}) = \Pr(y_1)\Pr(x_1|y_1)\prod_{t=2}^{T}\Pr(y_t|y_{t-1})\Pr(\bar{x}_t|y_t) \tag{1}$$

where $\Pr(y_1)$ is the initial state distribution, $\Pr(y_t|y_{t-1})$ is a transition probability matrix (shared over all time so that $\Pr(Y_t = i|Y_{t-1} = j) = a_{ij}$ for some matrix $[a_{ij}]_{ij}$), and $\Pr(\bar{x}_t|y_t)$ are the unary distributions.

For a Gaussian model, $\Pr(\bar{x}_t|y_t)$ is a Gaussian mixture (possibly a different mixture for every possible value of y_t). For a hybrid DGM/DNN model, the unary $\Pr(\bar{x}_t|y_t)$ takes on a different form that best can be described using the mechanism of virtual evidence [2].

What is described above, however, is just an HMM, where $\bar{x}_t$ is a known (observed) vector, and y_t is hidden or unknown. In fact, there is a bit more subtlety than this, as the hidden or observed status of the y_t variables depends on if we are training a model, or decoding (testing) a model[1]. During training, indeed y_t (or some aspect of y_t) is known. During decoding, however, y_t is unknown and must be inferred via the process:

$$y^*_{1:T} \in \operatorname*{argmax}_{y_{1:T}} \Pr(\bar{x}_{1:T}, y_{1:T}) \tag{2}$$

While this might look difficult to do, dynamic programming (implemented in GMTK) makes this process very fast.

What we have described so far is an HMM, but the tutorial below in fact is a DGM, which means that there is structure to the hidden variables. This means that it is no longer accurate to think of y_t as a single integer describing a phone identity. Instead, y_t itself is a vector of values, some element of which might be the phone and some elements of which describe random sequencing processes that help to form the mapping between the observed sequence $\bar{x}_{1:T}$ and the phone sequence. Then $y_{1:T}$ is a sequence of vectors.

In order to make this distinction, lets break the hidden state into two sequences variables, $y_{1:T}$ which for the purposes of this discussion we may assume to be a sequence of integer-valued phone variables, and $h_{1:T}$ which is a sequence of vector-valued variables, where the elements of the vectors are integers. An example of an $h_{1:T}$ is given in Section 6.1. The precise form and meaning of $h_{1:T}$ is the choice of the DGM model designer (we have made a particular choice for you in the below), but in general there is often many different ways of producing the same underlying statistical model — a design goal is to, from amongst this equivalence class of models, produce one that is attractive in some way (e.g., uses few memory or computational resources). In any event, once this is done, we have produced a model for the probabilistic quantity:

$$\Pr(\bar{x}_{1:T}, h_{1:T}, y_{1:T}). \tag{3}$$

Viterbi decoding then takes the form:

$$(h^*_{1:T}, y^*_{1:T}) \in \operatorname*{argmax}_{h_{1:T}, y_{1:T}} \Pr(\bar{x}_{1:T}, h_{1:T}, y_{1:T}) \tag{4}$$

The answer we seek, namely the phone sequence, resides solely within $y^*_{1:T}$ and so we then ignore the other decoded hidden variables $h^*_{1:T}$ — this is the nature of Viterbi decoding.

3 Background on the TIMIT speech corpus

In this section, we outline a set of generally accepted best practices for work on TIMIT, based on a literature survey and by contacting several researchers.

TIMIT is a standard speech database widely used in the field of automatic speech recognition — even given its age, it is still common for new models to be tested, debugged, and proven on TIMIT before graduating to larger data sets. TIMIT has most of the aspects of hierarchical sequential streaming classification tasks, in that there are objects (Phones) which one may assume is comprised of subparts (subphones). For the purposes of an application such as human activity recognition, one would map a "Phone" to an "Activity" and a subphone to a sub-part of that activity.

The TIMIT speech database [6] consists of a total of 6300 sentences. Our models are trained on the 3696 sentences from the TIMIT training set that are not labeled SA. The development set (400 sentences) and test set (192 sentences) are drawn from the "test" section of TIMIT. The three sets are divided by speaker so that no two sets contain utterances by the same speaker.

[1] The term "decoding" comes from communications theory, where a message that encoded and then transmitted over a channel is "decoded" on the receiving end. Speech recognition has long adopted the same terminology, as speech recognition technology was inspired by communications theory models, where the "speech" is considered to be the noisy signal needing to be decoded and received over a noisy channel. In the machine learning community, this process is often called MPE (most probable explanation) inference.

Following Lee and Hon [9] (who are the first to define the standard way of using TIMIT), the set of 60 TIMIT phone labels are collapsed to a smaller set of 48 phones for training. In addition to those 48 phones, we also include (for training only) two more phones: First, we retain the glottal stop 'q' (i.e., the stop within the word "button") as a training label, but ignore it during evaluation (it is removed entirely by Lee and Hon for training and testing). We also employ a "dummy phone" that is used to label phones with a duration of two frames or less, but is not allowed during the decoding phase so it does not show up in transcriptions.

The process in use requires the training of a Gaussian Mixture Model (GMM) based Dynamic Graphical Model (DGM) with TIMIT Data using the Expectation Maximization (EM) algorithm [1]. The phone sequence for each sentence corresponds to the labels in use for the training. We use forced Viterbi alignment to estimate the frame level labels. The HTK Speech Recognition Toolkit scores the decoded results using its `HResults` command.

The input features are Mel-frequency cepstrum coefficients(MFCCs) over 25.6 ms windows, plus the first-order and second-order temporal differences, giving 39 total features per 10 ms frame (this means that $\bar{x}_t \in \mathbb{R}^{39}$). The Graphical Model is trained with input features (mean subtracted, and variance normalized) from the TIMIT training set in use. There are up to 64 components of the GMM with the stopping criterion being that the log likelihood ratio between successive EM iterations should be less than 0.002.

The decoded phones are scored using the HResults tool of HTK. The glottal stop 'q' phone is ignored while scoring. So the scoring command that implements the phone mapping in Table 1 is:

```
HResults \
  -I timit_ref_transcription_39Phones \
  -e ih ix  -e ax ah  -e aa ao   -e n en \
  -e l el   -e sh zh  -e cl vcl  -e cl epi \
  -e "???" sil   -e "???" q \
  phone_list_39 decoded.mlf
```

where `timit_ref_transcription_39Phones` is the reference label file containing the true phone label sequence for each utterance, `decoded.mlf` contains the decoded phone labels produced by gmtkViterbi and converted to HTK's MLF file format, and `phone_list_39` contains the set of 39 phones used for evaluation. More details on the vocal tract meaning of these phones (e.g., that "vcl" is a generic voiced closure) is given in the TIMIT distribution [6] (and a good introduction on phonetics in general is [5]), but it is sufficient to do this tutorial without knowing the meaning and form of each phone, and instead to just think of phones as "high level states."

Note that `HResults` is a scoring command that is part of HTK [12] and that does string-string alignment using Levenshtein distance algorithm (see `http://en.wikipedia.org/wiki/Levenshtein_distance`).

Before scoring, the 48 remaining phones are mapped to a set of 39 phones as detailed by Lee and Hon [9]. Also, before decoding, some phones are mapped to a single phone thereby bringing down the number of phones when the results are scored. The mapping is as given in Table 1. Note that Lee and Hon [9] have a slightly different mapping — they map $\{vcl, epi, cl\} \rightarrow \{sil\}$. We map $\{vcl, epi\} \rightarrow \{cl\}$.

original	mapped	original	mapped
ix	ih	ah	ax
ao	aa	en	n
el	l	zh	sh
vcl	cl	epi	cl

Table 1: The phone mapping done after decoding but before scoring.

4 Two types of sequential labels

It is typical in sequential pattern recognition problems for there to be a lot of repetition in both the training data, and also in the decoded sequence. To make this idea precise, suppose that $y_{1:T}$ is a sequence of labels corresponding to observations $\bar{x}_{1:T}$. A typical example set of labels might take the following form:

$$(y_1, y_2, y_3, \ldots, y_T) = (\text{'c','c','c','a','a','a','a','a','a','t','t','t'}) \tag{5}$$

which means that $T = 12$, and $y_i =$ 'c' for $i \in \{1, 2, 3\}$, $y_i =$ 'a' for $i \in \{6, \ldots, 9\}$, and $y_i =$ 't' for $i \in \{10, \ldots, 12\}$.

For training a sequential model, there are two forms that this information might come in. In **frame labeling**, one has precisely the information given above, namely one knows the precise label at each frame of every training sample. In **sequence labeling**, on the other hand, the labels only consist of the unique sequence where the repetitions have been stripped out, and where the points of transition between different labels are unknown. For example, the sequence labeling variant of the above labeling would consist only of the three labels $('c', 'a', 't')$.

While one might think that it is always desirable to have frame-level labeling, sequence level labeling is in fact more common, and this is for several reasons. First, it is very costly, time-consuming, and error prone to produce frame-level labelings since the labels contain more information than the corresponding sequence labelings. Typically, humans (or some very computationally expensive process) needs to be employed to produce such labelings. Secondly, the points of transition between two successive distinct labels are often ambiguous at best, or entirely arbitrary at worst, and when a human (or a process) is forced to make a crisp decision at the transition points, this can in fact introduce error. Sequence labeling, by leaving the transition point unspecified (and left to be determined by the training system) can in fact be better since they offer an expression of the inherent uncertainty in the labels, and they can also produce more accurately performing trained models. Sequence level labels, however, mean that the training process is a bit slower since there are more hidden variables (it is unknown where the transitions lie, and so they must be hypothesized at training time).

Fortunately, it is often cost-effective to train a sequence model using only sequence labels, and this is made possible by the EM algorithm [1], which GMTK uses. In this tutorial, we describe how to obtain GMM boot models using both labeling scenarios, using frame labels in §5, and then sequence labels in §7.

Clearly, given a frame labeling one can easily derive a sequence labeling just by removing successive duplicates in the frame labeling. Thus, the easier to acquire form of labeling is easily derived from the more difficult to acquire form of labeling. TIMIT is a corpus that happens to have true frame labeling information (done by humans who have analyzed the speech), but this is relatively rare both in the speech recognition world and also rare in all sequential processing data sets. A third form of labeling, intermediary between frame and sequence labeling is described in [11] that is almost as easy to acquire as sequence labeling but it has empirically shown to perform better in practice.

5 Frame Label GMM-based TIMIT Boot Model

The process starts with training a GMM from the TIMIT training data. The training data consists of a sequence $\bar{x}_{1:T}$ of MFCC [10] observations — MFCCs are a standard vector time-series feature representation for speech, but it should be noted that any real-valued observed time series could be used for this kind of model. For each spoken sentence i (or what we will call a "segment"), we have a sequence of features $\bar{x}^{(i)}_{1:T_i}$, where T_i is the length of the i^{th} segment.

In the frame label case, each time frame of the training data is labeled with the phone being spoken (or silence). This means that the training data consists of sequences $\bar{x}_{1:T}$ and $y_{1:T}$ where everything about $y_{1:T}$ is known and observed.

On the other hand, it is typical for each phone to be represented by some number of sub-parts or sub-states, say a beginning, middle, and end. Even in the frame-label case, the whereabouts of the sub-parts of a phone are unknown, so in this section what we actually describe is sort of a hybrid between frame labels (at the high-level phone level) and sequence label (at the sub-phone level). That is, we know that each phone comprises a beginning, middle, and end, but we don't know where the transitions between beginning and middle, and middle and end reside.

Our model does this by representing each phone via three "sub-states," corresponding to the beginning, middle, and end of the phone. Since the states are not observed in the training data, the EM training algorithm has to hypothesize different transition points during training in order to determine (after training) the most likely ones. This is exactly what the EM algorithm is good at – there is some missing information (i.e., the transition points) and EM produces a maximum-likelihood estimate of the parameters under such missing values.

In GMTK, the graphical structure of a model is defined in a "structure file" (that typically uses `.str` as the extension) and the numerical parameters of the model (i.e., the things that will be trained based on training data) are defined in a "master file."

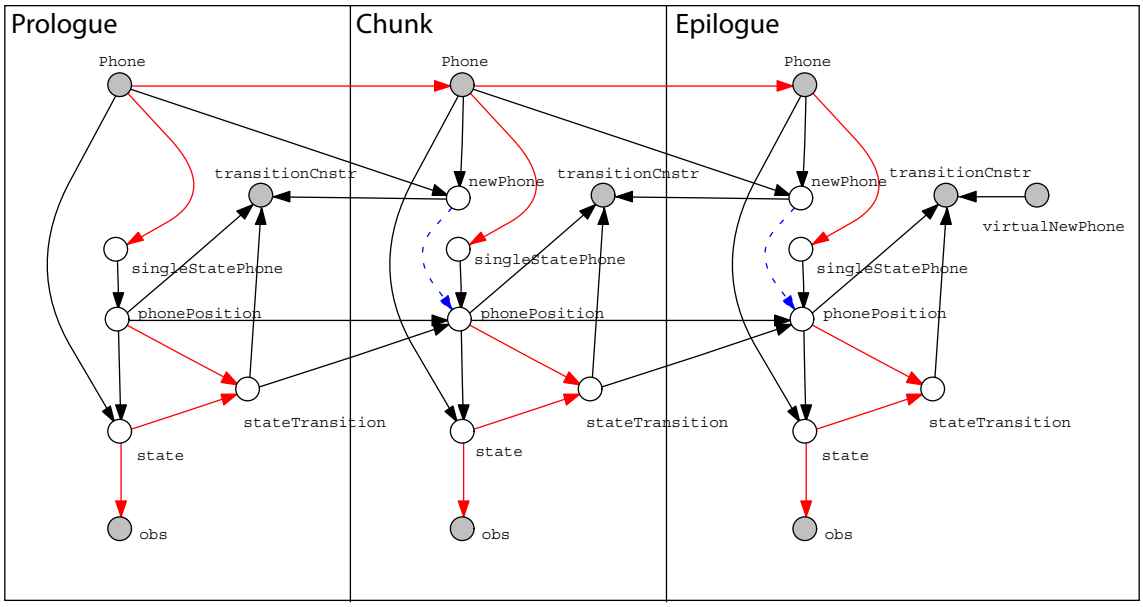

Figure 1: The graph structure to train the GMM-based boot model that uses frame label information (i.e., at each frame, the phone identity is known). While the model only shows three time frames, there are three sections to this model, the prologue (left), chunk (middle, which gets repeated and unrolled as desired so that the unrolled model has length T), and the epilogue (right). If we unroll k times of $k \geq 0$, the model will thus have length $k + 3$.

5.1 Structure File

See Section 15 of the GMTK manual for the definition of the structure file language. The structure file for the training version of the frame label GMM-based boot model is in the `timit_frame_label_gmm_train.str` file. Figure 1 illustrates the model, where hidden variables are unshaded and observed variables are shaded gray. Deterministic dependencies are indicated by solid arrows, random dependencies are indicated by red arrows, and switching dependencies are indicated by dotted arrows.

The model expands the three variables used in Equation (3) in the following way. At each time frame, t there is a variable $\texttt{Phone}_t$ that corresponds to the phone variable y_t. Also, at each t, there is a set of variables ($\texttt{newPhone}_t$, $\texttt{transitionCnstr}_t$, $\texttt{singleStatePhone}_t$, $\texttt{phonePosition}_t$, $\texttt{stateTransition}_t$, and $\texttt{state}_t$) that corresponds to the vector variable h_t. Lastly, there is an observation $\texttt{obs}_t$ at each time corresponding to $\bar{x}_t$.

More specifically, the variables used in the model are defined as follows:

`Phone` is the non-negative integer observed phone variable, that corresponds to the label y_t for each time $t \in \{1, 2, \ldots, T\}$. These variables take on the integer value that defines the phone at each time frame. There are 50 total phones used in the training data, valued from 0 to 49. After an utterance is decoded by a decoding model, the set of 50 phones is mapped down to a set of 39 phones for accuracy scoring (see Table 1).

`newPhone` is a binary indicator random variable that is one whenever the current and previous phone are different, and is zero otherwise. While `newPhone` is officially a "random" variable of its parents, since it is determined with probability one from its parents, and since its parents are always observed, this variable is de facto observed.

`transitionCnstr` this is a transition constraint variable that is always observed to be the value one. The effect of this variable and its parents is to ensure that: 1) any random hypothesis that makes a transition (i.e., $\texttt{stateTransition} = 1$) out of the end of phone (i.e., $\texttt{phonePosition} = 2$) when it is not the end of a phone (i.e., $\texttt{newPhone} = 0$) is given a score of zero; and 2) any random hypothesis that makes a transition (i.e., $\texttt{stateTransition} = 1$) out of **not** the end of phone (i.e., $\texttt{phonePosition} < 2$) when it **is** the end of a phone (i.e., $\texttt{newPhone} = 1$) is also given a score of zero. The way this works is via a CPT that has the

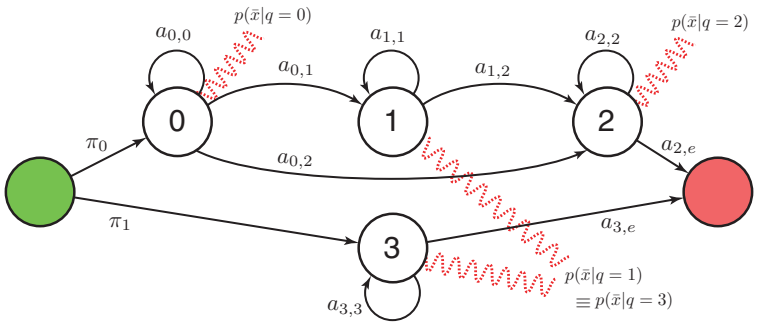

Figure 2: The basic stochastic finite state automata (SFSA) structure of the phone model in the tutorial. Each phone is represented by a stochastic sequencer that must follow this structure. Each phone has the same structure, but may have different parameters. The green state is the start state, and the red state is the end state. A phone may either have three parts (begin, middle, or end) represented by states 0, 1, and 2 respectively, or it may have only one part (middle) represented by state 3. If it has three parts, it may skip the middle state. A phone might also have only one state (middle), in which case it goes through state 3. Each state specifies a Gaussian mixture distribution, given by $p(\bar{x}|q=j)$ where j is the state value, and this is shown in the figure by the wavy red lines. Note that while state 1 and state 3 are distinct states, they both share the same Gaussian mixture, representing the middle of a phone, which means that the two distributions $p(\bar{x}|q=3)$ and $p(\bar{x}|q=1)$ are constrained to be always the same during training and decoding — this is indicated in the figure by the two wavy red lines from these two states to the same distribution.

form:

$$\begin{aligned}\Pr(\texttt{transitionCnstr}=1|&\texttt{newPhone}=i,\\ &\texttt{phonePosition}=j,\\ &\texttt{stateTransition}=k)=f(i,j,k)\end{aligned} \tag{6}$$

where $f(i,j,k)$ is an appropriately 0/1-valued function. Based on Figure 2, this function is [2]:

$$f(i,j,k)=\begin{cases}1 & \text{if its a new phone, and an between-phone transition is being hypothesized}\\ 1 & \text{if its not a new phone, and a within-phone transition is being hypothesized}\\ 0 & \text{else}\end{cases} \tag{8}$$

$$=\begin{cases}1 & \text{if } (i=1)\wedge(k=1)\wedge(j\in\{2,3\})\\ 1 & \text{if } (i=0)\wedge\Big((k=0)\vee\big((k=1)\wedge(j\in\{0,1\})\big)\vee\big((k=2)\wedge(j=0)\big)\Big)\\ 0 & \text{else}\end{cases} \tag{9}$$

`singleStatePhone` is a hidden binary indicator variable that indicates if the current phone is to be considered a one state phone or a three state phone. In general, when one has a high-level sequential object (a phone in this case), one needs to specify how it is decomposed into a set of substates. This is a design decision and different applications will have different designs. For the case, say of activity recognition, the way a given activity is decomposed will depend on the activity (e.g., running will be different than sitting).

In the present case, phone recognition, all phones take the same stochastic structure but we let the EM algorithm decide the training of the parameters of that structure. The underlying structure is given in Figure 2 — each phone has its own distinct set of parameters (corresponding to π_0, π_1, $a_{0,0}$, $a_{0,1}$, $a_{0,2}$, $a_{1,1}$, $a_{1,2}$, $a_{2,2}$, $a_{2,e}$,

[2] The actual decision tree `stateTransCnstr` in the file `PARAMS/timit_frame_label_gmm.mtr` implements the logically identical Equation (7) on the right.

$$f(i,j,k)=\begin{cases}1 & \text{if } (i=0)\wedge(j=0)\\ 1 & \text{if } (i=0)\wedge(j=1)\wedge(k\in\{0,1\})\\ 1 & \text{if } (i=0)\wedge(j\in\{2,3\})\wedge(k=0)\\ 1 & \text{if } (i=1)\wedge(j\in\{2,3\})\wedge(k=1)\\ 0 & \text{else}\end{cases} \tag{7}$$

$a_{3,3}$, and $a_{3,e}$) that govern the probabilities of a sequence through each phone. It is important to realize that while Figure 2 shows a single stochastic finite-state automata (SFSA), the GMTK TIMIT tutorial implements this SFSA via the logic of multiple random variables, some of which have a deterministic and some of which have a stochastic relationship with their parents.

`singleStatePhone` in this case is a binary indicator that, when one, says that the phone go through state 3 in Figure 2, and when zero, says that the phone go through state 0, 1, and 2 in Figure 2. Hence, `singleStatePhone` corresponds to the probabilities π_0 and π_1 in Figure 2. Since these probabilities will be phone-dependent, however, `singleStatePhone` has as its parent the `Phone` variable.

`phonePosition` is the "phone position," sub-state variable, taking on values 0, 1, or 2 representing the "beginning", "middle", and "end" of the current phone — also, state 3 representing an alternate "middle" of a short version of the phone (as done in Figure 2).

Note that from `phonePosition` alone, one can't define the phone, only what section of some phone we are in. The pair (`phone`,`phonePosition`) defines both the phone ID and sub-phone. How `phonePosition` evolves over time is determined by its parents.

In this model, there are constraints about how `phonePosition` may evolve. First, at the beginning of a phone, either `phonePosition` $= 0$ (for a three part phone) or `phonePosition` $= 3$ (for a one part phone).

In the three-state case (when `singleStatePhone` $= 0$), `phonePosition` may either sequence through all three parts of a phone (beginning, middle, end) or may skip the middle part and sequence through only two parts of a phone (beginning, end).

In the one-state case (when `singleStatePhone` $= 1$), `phonePosition` only is at the middle of a phone. Note that in this case, `singleStatePhone` $= 1$ represents the beginning, middle, and end of a phone since it has only one subphone — hence one can jump into the phone right at `singleStatePhone` $= 1$ and jump out of the phone from `singleStatePhone` $= 1$.

The end of a phone is when `phonePosition` $= 2$ (for a three part phone) or `phonePosition` $= 3$ (for a one part phone) and `stateTransition` $= 1$, and in such case, `phonePosition` may jump to the "beginning" (value 0 or 3) of the next phone. The way this is done, however, is with the parent `newPhone`. That is, when $\texttt{newPhone}_t = 1$, then $\texttt{phonePosition}_t$ takes the value 0 if `singleStatePhone` $= 0$, or takes the value 3 if `singleStatePhone` $= 1$. This is a random event since `singleStatePhone` is a random function of its parent (`Phone`).

`stateTransition` is a ternary state transition indicator variable.

When `stateTransition` $= 0$, the `state` variable must remain the same in the next frame and there is no movement in the model from one to the next frame. This is done by making sure that `phonePosition` does not advance. That is, when `stateTransition` $= 0$, then $\Pr(\texttt{phonePosition}_t = j | \texttt{phonePosition}_{t-1} = j) = 1$ for any j.

When $\texttt{stateTransition}_t = 1$, the `state` variable must in the next frame either increment by one, or alternatively move to the beginning of the next phone — what `phonePosition` does is dependent on its current value as well. For example, when $\texttt{phonePosition}_{t-1} = 2$ or $\texttt{phonePosition}_{t-1} = 3$ then this event would trigger a phone transition. When $\texttt{phonePosition}_{t-1} = 0$ or $\texttt{phonePosition}_{t-1} = 1$ then this would just cause an advance to the next phone position (i.e., beginning to middle, or middle to end).

When `stateTransition` $= 2$, then a hypothesis is advanced that `phonePosition` should advance by two positions, which only makes sense (and is only allowed) when `phonePosition` $= 0$. The only reason, in fact, that `stateTransition` has `phonePosition` as a parent is so that it knows to allow a value of `stateTransition` $= 2$ only when `phonePosition` $= 0$.

Also, in this frame label training model, if any hypothesis is advanced that is not consistent with `newPhone`, then that hypothesis is annihilated by `transitionCnstr` as mentioned above.

`state` is the state variable, representing the 2-tuple `phone` $\times$ `phonePosition` possible values. This mechanism allows each pair (P,PP) to have its own unique integer that then can easily select a distribution for the observation. That is, $\Pr(\bar{x}_t | h_t)$ is via $\Pr(\mathrm{O}_t | \mathrm{S}_t)$.

Note that while `phonePosition` has four possible values (since it has two representations for the middle of a phone), there is only one representation of the middle of the phone by the `state` variable. This is done to enforce that the Gaussian mixture for the middle of a phone is shared between the case of a single-subphone or a multi-subphone phone, as shown in Figure 2.

`obs` is the observed MFCC data for each frame, and corresponds to $\bar{x}_t$. Note that each frame is really a vector (rather than a scalar) of observed real-valued data.

At the start of training, there is one component per Gaussian mixture, but after training there will be at most 64 Gaussian components per mixture in this model. For notational conciseness we identify in the following $\texttt{obs}_t \equiv \bar{x}_t$ and $\texttt{state}_t \equiv q_t$. This means that $\Pr(\bar{x}_t|q_t) = \sum_{i=1}^{64} \alpha_i \mathcal{N}(\bar{x}_t|\mu_{q_t,i}, \Sigma_{q_t,i})$ where $\mu_{q_t,i}$ is a mean vector for state q_t and mixture component i, and $\Sigma_{q_t,i}$ is a covariance matrix for state q_t and mixture component i, $\mathcal{N}(\bar{x}|\mu, \Sigma)$ is a multivariate Gaussian with mean μ and covariance matrix Σ, and where $(\alpha_1, \alpha_2, \ldots, \alpha_{64})$ is a vector of mixture coefficients with $0 \leq \alpha_i \leq 1$ for all i, and $\sum_i \alpha_i = 1$.

The way the GMTK splitting procedure works is that each Gaussian component is split into two Gaussian components that have very slightly different mean values. Then, further EM iterations allow these new Gaussians to adjust based on the data.

Note that most of GMTK's ASCII input files are (for better or for worse) run through the C preprocessor `cpp` before GMTK reads them. While this is very useful as it allows us to use C's macro preprocessor to define macros, it also can cause some problems as GMTK's graph language is of course not C (for example, apostrophes sometimes confuse `cpp`, so best to avoid things like contractions, even in comments). Some of the parameters required in the structure file above are given as macros defined in the `PARAMS/timit_frame_label_gmm.h` file.

5.2 Master File

The numerical parameters of the training version of the frame label GMM boot model are defined in the master file called `timit_frame_label_gmm.mtr` and which can be found in the directory `PARAMS`. Section 16.4 of the the GMTK manual describes the overall format of the master file. Section 17 of the the GMTK manual details the specific format of each type of parameter.

5.3 Triangulation

GMTK requires that graphical models be triangulated before it can perform inference or training on them. This is accomplished with the `gmtkTriangulate` command.

The `tricommand` script will triangulate all the graphical models used in the tutorial. If you change a structure file, you will need to re-run the `gmtkTriangulate` command with the `-reSection` argument to synchronize the triangulation with the new model structure. The details of what is happening during triangulation are beyond the scope of this tutorial but the process of graph triangulation for static graphs is described in Section 6.8.3 of the GMTK manual, and for dynamic graphs in Section 13.3 of the GMTK manual.

5.4 Training the Boot Model

The GMM boot model is trained with the `gmtkEMtrain` command. See Section 18.5 of the GMTK manual for a full description of the command. The `frame_boot_train_command` script implements external parallel EM training. It is intended to run on a single machine with a multicore CPU. Set the `NUMPROCS` environment variable to specify the number actual of parallel `gmtkEMtrain` processes to run (the default is 2, but it can be set to 1 if you want to run in serial).

The way parallelism works is that, for each EM epoch, multiple `gmtkEMtrain` jobs are run each of which produces a separate accumulator file (a file that maintains partial training results from the EM algorithm), and then there is a final merge step that merges the separate accumulator files. In the current script setup, the number of such accumulator files does not change when you change `NUMPROCS`, rather what changes only is the number of simultaneous jobs that are run. Note that it would be possible to modify the script so that the number of accumulator files is the same as `NUMPROCS` (in fact, when `NUMPROCS` is one, GMTK does not even need to use an accumulator file as the information can be maintained entirely internally).

5.4.1 Generating Accumulator Files

The `generateAccumulators` function in the `frame_boot_train_command` script invokes `gmtkEMtrain` to produce EM accumulator files for the each section of the training data. The actual GMTK command line is

```
$TKSRC/gmtkEMtrain  \
        $OFARGS \
        -trrng $TRRNG \
        -inputMasterFile $MASTERFILE \
        -inputTrainable $3 \
        -strF timit\_frame\_label\_gmm\_train.str \
        -maxEmIters 1 \
        -storeAccFile accumulators/acc_${p}_${1}_${2}comp.data \
        -checkTriFileCards F \
        -varFloor 1e-5 \
        -floorVarOnRead T \
        -dirichletPriors $DIRICHLET \
        -meanCloneSTDfrac 0.25 \
        -covarCloneSTDfrac 0.0
```

A few of the key arguments are:

`$OFARGS` are the `-ofX` arguments specifying the training MFCC and phone label observation files (see Section 17.9 of the the GMTK manual for more details). In this case, we use

- **`-of1 LISTS/train_mfcc_noSA.pfile`** to read in the MFCC features of the TIMIT development set
- **`-trans1 A@normalize`** normalizes the MFCCs to have mean 0 and variance 1 by applying an affine transform. The `normalize` file contains the parameters for the transform, and was generated from the training data by the `obs-stats` program. Note that the parameters for the normalization computed from the training data are also applied to the observed MFCCs during decoding.
- **`-iswp1`** requests endian swapping, since the `.pfile` files are big-endian and the CPU is little-endian.
- **`-of2 LISTS/train_labs_noSA.pfile`** contains the frame-level phone labels for the TIMIT development set.

`-trrng` specifies the range of training utterances the `gmtkEMtrain` process will handle. The utterances are split up approximately equally between the `$NUMPROCS` processes. The `TRRNG` variable holds a range of training utterance numbers in the form `X:Y` which each `gmtkEMtrain` process will be responsible for processing.

`-inputMasterFile` specifies the numerical parameters of the model (see Section 5.2). This will be `PARAMS/timit_frame_labe` for the frame label training model, or `PARAMS/timit_seq_label_gmm.mtr` for the sequence label training model.

`-inputTrainable` specifies a special master file containing the learned parameters for the current EM iteration (see Section 18.5.1 of the the GMTK manual).

`-strF timit_frame_label_gmm_train.str` defines the graphical structure in Figure 1.

`-maxEmIters` only do 1 EM iteration per invocation in external parallel training.

`-dirichletPriors` causes one instance of the `gmtkEMtrain` to process to add a small constant to its accumulator for the `stateTransition` variable to avoid getting a warning about seeing 0 counts for the illegal cases ruled out by the `transitionCnstr` constraints.

`-storeAccFile` specifies the file name to store the accumulators computed for the processed utterances. The `${p}` in the name is the process number (0–`$NUMPROCS` $-$ 1). The `$NUMPROCS` accumulator files will be merged in the `accumulateAndTrain` function.

`CPPARGS`

5.4.2 Merging Accumulator Files

The `mergeAccumulators` function in the `frame_boot_train_command` script invokes `gmtkEMtrain` a final time per EM iteration to merge the accumulator files and update the parameters.

```
$TKSRC/gmtkEMtrain  \
      $OFARGS \
      -trrng nil $5 $6 $7 $8 \
      -inputMasterFile $MASTERFILE \
      -inputTrainable $3 \
      -outputTrainable $4 \
      -strF $STRFILE \
      -maxEmIters 1 \
      -loadAccFile accumulators/acc_@D_${1}_${2}comp.data \
      -loadAccRange $ACCRANGE \
      -llStoreFile llstore/llstore.${1}_${2} \
      -checkTriFileCards F \
      -varFloor 1e-5 \
      -dirichletPriors T \
      -floorVarOnRead T \
      -meanCloneSTDfrac 0.25 \
      -covarCloneSTDfrac 0.0
```

This command has a few new arguments:

`-trrng` is empty, since this invocation is just merging the accumulator files rather than processing any training data.

`$5--$8` are optional `-mcsr` and `-mcvr` arguments to implement splitting and vanishing. See the description of the `-mcsr` argument in Section 18.5of the the GMTK manual.

`-outputTrainable` is a special master file specifying where to store the updated parameters for the next iteration (see Section 18.5of the the GMTK manual).

`-loadAccFile` specifies the accumulator files to merge.

`-loadAccRange` The `@D` in the accumulator file name is replaced by each integer in the `$ACCRANGE` range, which is `0:$((NUMPROCS-1))`.

`-llStoreFile` stores the log likelihood of the current iteration. This can be used to evaluate stopping criteria.

-meanCloneSTDfrac when Gaussians are split (or cloned), the mean of the new Gaussian is itself a random quantity governed by a separate Gaussian distribution. This option controls the parameters of this separate Gaussian distribution. See Section 18.5 of the GMTK manual for full details.

-covarCloneSTDfrac A similar option but for the variance of a separate Gaussian distribution. See Section 18.5 of the GMTK manual for full details.

5.4.3 Training to Convergence

The `trainToConvergence` function in the `frame_boot_train_command` script simply iterates calling the previous 2 functions to implement an EM iteration. It checks the ratio of the previous and current log likelihoods to stop when the desired level of convergence is reached.

5.4.4 Splitting and Vanishing

The `frame_boot_train_command` script uses the functions described in Sections 5.4.1 through 5.4.3 in this tutorial to implement the splitting and vanishing schedule in the description of the `-mcsr` option in Section 18.5 of the the GMTK manual. The appropriate `-mcsr` and `-mcvr` options are passed to the `mergeAccumulators` function to allow the learned Gaussian components to split or vanish as they are updated in an iteration. Note that no splitting/vanishing is performed in `generateAccumulators`.

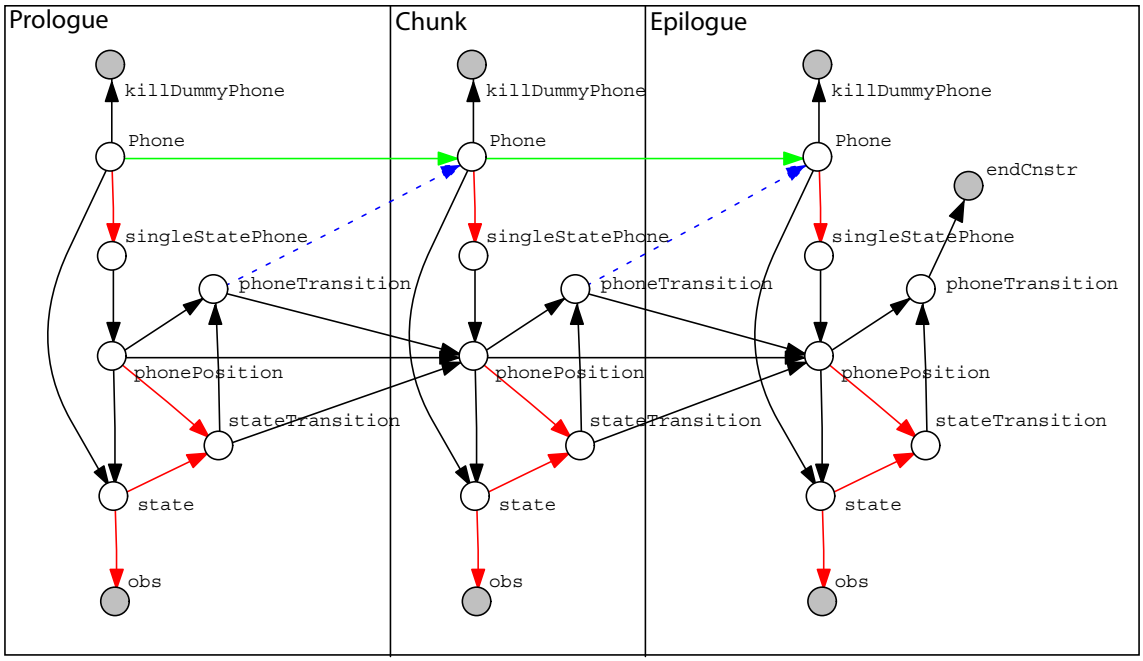

Figure 3: GMTK's TIMIT phone decoding structure. Note that this same code structure is used to decode parameters both by the frame label model of Figure 1 and also the sequence label model of Figure 4.

6 Evaluation: Viterbi decoding, MPE inference, or recognition

As described in Section 14.2 of the GMTK manual, it is typical to use different structure files for training a model, and decoding (e.g., recognition, Viterbi decoding, or MPE inference), and the present case is no different. Sometimes, there will be multiple training models, each one of which produces some subset of the parameters that are ultimately gathered together and used as the "trained" model. In the present TIMIT case, there is one training model that produces parameters that are used for a trained (decoding) model.

The GMTK `gmtkViterbi` command performs Viterbi decoding on a model. See Section 18.7 of the the GMTK manual for a full description of the command. The `frame_boot_vit_command` script invokes `$NUMPROCS` `gmtkViterbi` processes in parallel to produce a Viterbi decoding of the TIMIT test set using a model that utilizes parameters trained via the frame label GMM training model. A similar `seq_boot_vit_command` script does the Viterbi decoding utilizing the parameters trained via the sequence label GMM training model described in Section 7. The decoding process may take several minutes to complete. You can observe its progress by running `tail -f` on one of the log files in the `logs` directory. The decoding logs are named `logs/decode_X:Y.log`, where `X` and `Y` represent the range of utterances each `gmtkViterbi` instance is processing.

The reason decoding requires a slightly different model structure is that the phone labels are now hidden during evaluation (where they are to be inferred) but observed during training (where they are part of the training labels). The file `timit_gmm_decode.str` contains the decoding structure which is illustrated in Figure 3. Note that this decoding model is also used to decode the sequence label trained model described in Section 7.

6.1 Decoding Model

The TIMIT phone decoding model is shown in Figure 3. As mentioned above, it works slightly differently than the training model of Figure 1, although it has some commonalities. It does use the same underlying SFSA strategy for a phone model, as shown in Figure 2. Now, however, the `Phone` variable is hidden and random and is what needs to be hypothesized. The variable `Phone` corresponds to the variable y_t in Equation (4), and the other hidden variables in Figure 3 correspond to h_t in Equation (4).

We describe each of the variables in Figure 3 in turn:

`killDummyPhone` is a variable that is used to eliminate any hypotheses containing the "dummy phone." This

forces the decoding to pick the most likely non-dummy phone. Dummy phone was an extra phone created during training, and any instance of a phone in the frame label case that lasted only one or two frames was mapped to the dummy phone.

`Phone` is once again the variable that specifies what the current phone is in the current frame and it corresponds to y_t in Equation (4). In this decoding model, however, `Phone` is hidden and the job of decoding is to find the most probable sequence of assignments to the `Phone` variables at each time. At time zero, phone takes on an initial state distribution. At subsequent times, `Phone` is governed by its parents.

If $\texttt{phoneTransition}_{t-1} = 0$ then $\texttt{Phone}_{t-1} = \texttt{Phone}_t$.

If $\texttt{phoneTransition}_{t-1} = 1$, this means a new phone should be hypothesized and $\texttt{Phone}_t$ is randomly determined conditioned on whatever the previous phone is $\texttt{Phone}_{t-1}$. The phone-to-phone transition matrix `PhoneBigram` is used for this purpose which was trained by the training model.

`phonePosition` is the position in the phone and takes on values similar to what was done in Figure 1, namely four values, 0,1,2 for the beginning, middle, and end of a phone in the three-subphone case, or 3 (middle) in the one-subphone case.

If $\texttt{stateTransition}_{t-1} = 0$ then `phonePosition` keeps its previous value.

If $\texttt{stateTransition}_{t-1} = 1$ and $\texttt{phoneTransition}_{t-1} = 1$, then `phonePosition` is initialized to a new value (either 0 or 3) depending on the outcome of $\texttt{singleStatePhone}_t$ (in this case, $\texttt{phonePosition}_t \leftarrow 3$ if $\texttt{singleStatePhone}_t = 1$ and $\texttt{phonePosition}_t \leftarrow 0$ if $\texttt{singleStatePhone}_t = 0$.

The rest of the time, `phoneTransition` evolves as it should, similar to Figure 1 and as shown in Figure 2.

`singleStatePhone` is an indicator which is used to determine if a new phone (when it is instantiated) is of the single subphone variety (going through only state 3 in Figure 2) or of the multi-subphone variety (going through states 0-2 in Figure 2).

`singleStatePhone` uses `Phone` as a parent since it needs to know the Phone specific probabilities for being either of the single-state or the multi-state variety.

`phoneTransition` is triggered only when we make a state transition out of the last sub-state of a phone. In particular, $\texttt{phoneTransition}_t = 1$ if and only if: 1) $\texttt{stateTransition}_t = 1$ and 2) when either $\texttt{phonePosition}_t = 2$ or $\texttt{phonePosition}_t = 3$.

Note that `phoneTransition` does not need to also use `Phone` as a parent since the structure, and number of subphones, for all phones are the same (e.g., all phones have the structure given in Figure 2). If it was the case that we wanted to use a different number of (or a different structure over) subphones for each phone, then there would need to be this additional edge (see, for example, Figure 3 in [3], where words take the place of phones, and phones take the place of subphones).

`state` behaves exactly the same as in Figure 1.

`stateTransition` behaves exactly the same as in Figure 1.

`obs` behaves exactly the same as in Figure 1.

`endCnstr` is an observed child, set to $= 1$, and its job is to ensure that we must make a phone transition in the last frame. I.e., all hypotheses that do not do this are annihilated.

6.2 Decoding Command

The `frame_boot_vit_command` and `seq_boot_vit_command` scripts invoke the `gmtkViterbi` command as follows to perform the Viterbi decoding:

```
$TKSRC/gmtkViterbi \
      -of1 LISTS/development.pfile -iswp1 $ISWP -trans1 A@normalize \
      -dcdrng $DCDRNG \
      -strF $STRFILE \
      -inputMasterFile $MASTERFILE \
```

```
-inputTrainable $TRAINEDFILE \
-pVitTrigger "phoneTransition(0)==1" \
-cVitTrigger "phoneTransition(0)==1" \
-eVitTrigger "phoneTransition(0)==1" \
-vitValsFile decoded/64comp_${zp}_${DCDRNG}.vit \
-cppCom '-DDECODING'
```

A few key arguments are:

-of1 specifies the MFCC data to decode, in this case the TIMIT development data set. The tutorial also includes the "core" and "full" TIMIT test sets in the `LISTS` subdirectory (see [8] for descriptions of these data sets). Note that if you change the test data set, you must also change the first argument to `gmtk2mlf` in `scorecommand`.

-dcdrng is the range of utterances in the test set to decode. The decoded set is split approximately evenly across the `NUMPROCS` processes. The argument `-dcdrng X:Y` instructs `gmtkViterbi` to decode utterances X through Y. The `NUMPROCS` separate results files are combined to evaluate the decoding accuracy (see Section 6.3).

-inputTrainable specifies the final learned parameters resulting from the training process. It will be either `learnedParams/64comp-frame.gmp` or `learnedParams/64comp-seq.gmp` depending on which training model you used.

-[p|c|e]VitTrigger enables Viterbi output for only frames where the phone changed (`phoneTransition` $=1$). The `pVitTrigger` applies to the prologue frames, and the `c` and `e` apply to the chunk and epilogue frames (see Section 1). Without the triggers, `gmtkViterbi` would give the Viterbi output for every input frame. The triggers are thus both a form of run-length compression for the Viterbi output, and necessary because the decoding accuracy is evaluated on the phone sequence, rather than individual frames (see Section 6.3).

-vitValsFile specifies the file to store the Viterbi decoding output.

-cppCom '-DDECODING' enables (via the C-preprocessor) some sections in the `PARAMS/timit_frame_label_gmm.mtr` that are not needed during training.

You may want to use the `frame_boot_vit_command` as a basis for implementing your solutions to the exercises in Sections 8 and 9. The script is well commented, so you may wish to read through it.

6.3 Accuracy Evaluation

The GMM boot model was trained on a set of 50 phones, however the accuracy is evaluated on just 39 phones. The `scorecommand` script maps the set of 50 phones produced by the decoding model down to the 39 phones used for evaluation. It uses the `HResults` command from the HTK [12] package to compute the percent correct and the phone accuracy rate:

$$\%\text{Correct} = \frac{H}{N} \times 100\%, \quad \text{and} \quad \text{Accuracy} = \frac{N-S-D-I}{N} \times 100\% = \frac{H-I}{N} \times 100\% \tag{10}$$

where N is the number of phone labels in the reference phone sequence label file, H is the number of phone labels that are correct, S is the number of phone label substitution errors, I is the number of phone label insertion errors, and D is the number of phone label deletion errors. A deletion is a label that is in the reference label string but that was not found in the corresponding aligned hypothesized string. An insertion is a label that is found in the aligned hypothesized string but that is not found in the reference string. A substitution is when one label in the reference string is substituted with another label in the hypothesized string. Note that these are all based on a string to string alignment, produced via the Levenshtein procedure (for more information, see [7]). Note that $N = H + S + D$ since the number of reference labels N is the combined number of correctly identified H by the hypothesized labels, the number incorrectly identified S by the hypothesized string, and the number unaccounted for D within the hypothesized string — a large number of extra I in the hypothesized string would indicate that one should not hypothesize so many phones.

The `HResults` command output calls accuracy "word accuracy" even though in the present case it is actually a phone recognition task, so what it is reporting is phone accuracy.

After running either `frame_boot_vit_command` or `seq_boot_vit_command` scripts to do Viterbi decoding, running the `scorecommand` converts the GMTK Viterbi output to the format used by the `HResults` program and invokes it to report the decoding accuracy. After training the frame label model and decoding, `scorecommand` should produce

```
====================== HTK Results Analysis =======================
  Date: Fri Sep 12 23:06:20 2014
  Ref : timit_ref_transcription_39Phones
  Rec : decoded/decoded.mlf
------------------------ Overall Results --------------------------
SENT: %Correct=0.00 [H=0, S=400, N=400]
WORD: %Corr=67.04, Acc=65.02 [H=9490, D=2026, S=2640, I=286, N=14156]
====================================================================
```

7 Sequence Label GMM-based TIMIT Boot Model

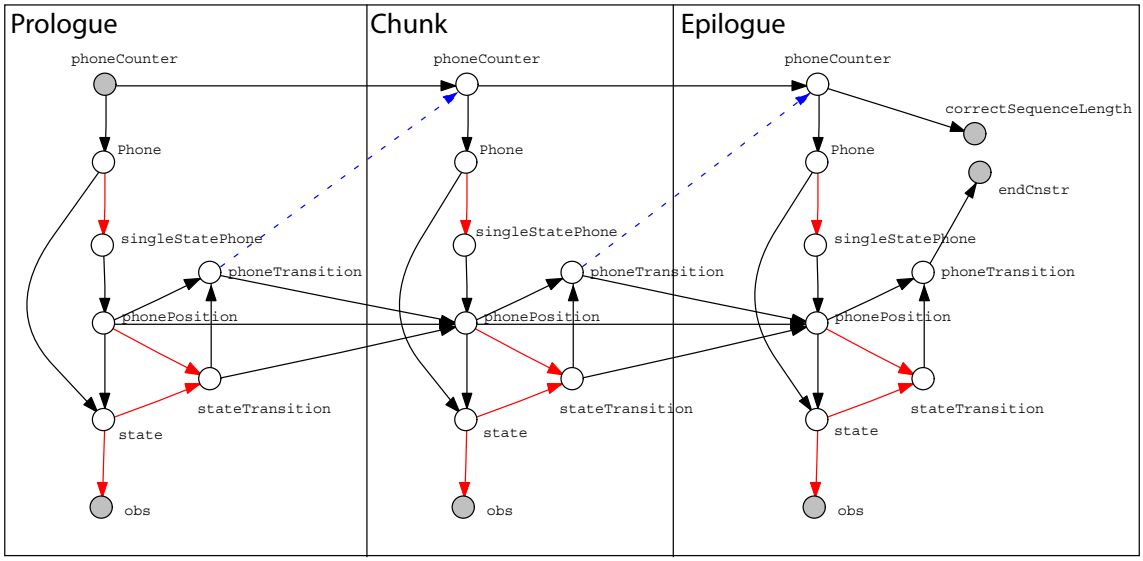

Figure 4: Phone Sequence Boot Model. Unlike Figure 1, this model uses only the sequence of unique phones to train (as described in Section 4) rather than every frame being labeled. Since more information is hidden here relative to Figure 1, this model will run slower.

The TIMIT corpus provides a phone label for each frame of the training utterances. As discussed in Section 4, humans often disagree about the boundaries where a phone ends and the next begins. It is also expensive to generate frame-level phone labels for a large training corpus.

Thus it is common to instead provide just the sequence of phones for a training utterance and allow the model to "decide" the phone boundaries for itself. The model defined in `timit_sequence_label_gmm_train.str` and shown in Figure 4 works in this fashion.

It is similar to the GMM boot model described in Section5, with a few different variables:

`phoneCounter` is a counter tracking the position within the known phone sequence of the utterance. For example, if there is a sequence of phones of the form "[c] [a] [t]" then `phoneCounter` will range from 0 to 2.

`phoneCounter`$_t$ is incremented by one if and only if the model hypothesizes a phone transition, and that is indicated by `phoneTransition`$_{t-1} = 1$.

`Phone` is like its compatriot variable of the same name in Figure 1 except that rather than being observed, it is hidden (like its other compatriot variable of the same name in Figure 3). The reason is that the actual phone at a given

Model	Correct	Phone Accuracy	H	D	S	I
Frame	67.04%	65.02%	9490	2026	2640	286
Sequence	69.86%	66.91%	9890	1684	2582	418
Pruned sequence	70.90%	67.77%	10037	1526	2593	443

Table 2: Accuracy of decoding based on frame label vs. sequence label training.

frame t is not known, only the sequence of phones is known and GMTK needs, during training, to hypothesize the different actual valid instantiations of phones to time frames. The mapping from `phoneCounter` to `Phone`, however, is deterministic as the identity of each phone at each position (e.g., in "[c] [a] [t]", it is known that when the counter is 0, the phone is "[c]", when the counter is 1 the phone is "[a]" and so on). GMTK uses a deterministic CPT mapping for this purpose. Note, however, that this mapping changes from utterance to utterance since each training utterance will have a different sequence of phones (and this is done with an iterated decision tree, described in Section 7.1).

`phoneTransition` just as in Figure 3, `phoneTransition` is triggered only when we make a state transition out of the last sub-state of a phone, and in fact the variable here has exactly the same purpose as in Figure 3.

The difference here, however, is what happens when $\texttt{phoneTransition}_t = 1$. In the present case, $\texttt{phoneTransition}_t = 1$ triggers an increment of `phoneCounter`, which means we move on to the next phone in the sequence of phones.

Note once again that, as we described in the case of Figure 3, `phoneTransition` does not have `Phone` as a parent, and the reason for this is that the structure of each phone is identical (and given in Figure 2). If such a model were to be used for words (or any object where the type of structure is dependent on the current object), then one would need to add such an edge.

`endCnstr` enforces the constraint that the hypothesized phone sequence length matches the known length of the current training utterance. It also ensures that we make a transition out of the last phone. That is, the only way of explaining the event that $\texttt{endCnstr}_T = 1$ is if $\texttt{phoneTransition}_T = 1$ and if $\texttt{phoneCounter} = M$ where M is the number of distinct phones in the current utterance (e.g., 3 in the case of "[c] [a] [t]").

The other variables in Figure 4 not mentioned in this list function exactly as they do in Figure 3.

Note that the sequence label training in Figure 4 will be slower than the frame label training of Figure 1, perhaps quite a bit so. For applications such as speech recognition, training time is less important than decode time, since training can be done offline (when a user of the model is not around). Testing/decoding (as in Figure 3) should be as fast as possible so any user of the model does not get frustrated.

7.1 Iterated Decision Tree

The `seq_boot_train_command` script trains phone sequence model. It uses the `PARAMS/timit_seq_label_gmm.mtr` master file, which implements the deterministic CPTs for the `Phone` and `CorrectSequenceLength` variables with iterated decision trees. The `PARAMS/segmentLenth.dts` and `PARAMS/labelIndex.dts` files each contain a decision tree for each training utterance, specifying the phone sequence length and the true phone sequence respectively.

7.2 Evaluation

The same decoding model (Figure 3) can be used to decode speech using the parameters learnt either with frame label training (Figure 1) or sequence label training (Figure 4). Running the `seq_boot_train_command` script trains the sequence label model (Figure 4). Once the parameters for the model have been learned, the `seq_boot_vit_command` script runs the decoding model (Figure 3) using the parameters learned by the sequence label training model. Then the `scorecommand` script evaluates the accuracy of the decoding. Table 2 compares the accuracy of the frame vs sequence training.

Because the sequence model takes many hours to train, we also tried a variation where we learned the common parameters of the frame and sequence models using the much faster frame model, then "boot strapped" the sequence model by starting from those values and training with aggressive pruning. The results for this model are given in the pruned sequence row in Table 2.

8 Exercise: Phone Insertion Penalties

It is sometimes the case that too many phones are hypothesized for an utterance, and it is other times the case that too few phones are hypothesized. Given an already trained model, there is an easy mechanism where one can impose a penalty (a phone insertion penalty) in the case of too many (or grant a reward in the case of too few) phone insertions. This can be done, moreover, without performing any additional training, and by varying the penalty/reward one can sweep through the trade-off between too many phone insertions and too many phone deletions.

Section 15.1.3 of the GMTK manual describes how a random variable score can be modified by a "weight" construct, and Section 15.1.5 of the document describes the idea of "switching" weights, where a weight is applied only of a certain condition is true. Lastly, Section 15.1.5.1 describes the idea of "word insertion penalties", where if $\beta < 0$, we incur an additional penalty every time a new word is hypothesized (inserted).

We can use the same idea to quickly express phone insertion penalties, and this can be done very quickly by changing the CPT for `phoneTransition` in Figure 3 to include a penalty. This is done by changing appropriate instances of the `Phone` variable in the structure file to the following:

```
variable : Phone {
   symboltable: collection("phone_symbol_table");
   type: discrete hidden cardinality NUM_PHONES;
   switchingparents: phoneTransition(-1) using mapping("internal:copyParent");
   conditionalparents: Phone(-1) using DeterministicCPT("CopyPreviousPhone") |
   weight: scale 3.6 penalty 0
         | scale 3.6 penalty -100;
}
```

The penalty value here is -100 and the more negative that number becomes, the greater the penalty and the fewer phone insertions there will be. Alternatively, one can make the penalty value positive, in which case it will express a phone insertion reward. Thus, a negative penalty value is a "penalty" (due to GMTK's representation of scores as log probabilities), while a positive penalty value is a "reward.". Note that there is no possible switching parent for the first `Phone` in the prologue, as it is the very first frame. You can simply set the unswitched penalty to the desired value.

Exercise 1 *Using the switching weight construct above, modify the file `timit_gmm_decode.str` to add appropriate phone insertion penalties. Try a range of penalty and reward values and investigate the behavior of the accuracy of the decoding model. What's the optimal penalty value to get the best accuracy?*

9 Exercise: Phone Length Limits

Using a simple modification of the decode model in Figure 3, it is possible to express a hard constraint on the upper limit of a phone duration. That is, the model can annihilate any hypotheses that correspond to a phone more than a set number of frames, say K. Doing this also corresponds to effecting a phone insertion reward, since if the phones are forced to be short, more total phones will be decoded.

A strategy for doing this is to add an extra phone length counter variable (`phoneLengthCounter`) to the model in Figure 3. Every time a new phone is hypothesized (e.g., those cases when $\texttt{phoneTransition}_t = 1$), the counter is reset to zero. Otherwise, the counter increments by one at each frame.

In order to annihilate hypotheses corresponding to long phones, we add one more variable to each frame, `lengthCnstr`, that is binary, and observed to be one. `lengthCnstr` is set up so that it's CPT takes the following form:

$$\Pr(\texttt{lengthCnstr}_t = 1 | \texttt{phoneLengthCounter}_t = j) = \begin{cases} 1 & \text{if } j \leq K \\ 0 & \text{else} \end{cases} \tag{11}$$

Adding this constraint to the decoding model thus has the desired effect.

Exercise 2 *Starting from the model you modified in Section 8, add the necessary random variables and deterministic CPTs and decision trees to add a phone length constraint.*

After your model is working, evaluate the performance of the model for different K. How does making K smaller compare with increasing the phone insertion reward of Section 8?

References

[1] Jeff Bilmes. A gentle tutorial of the EM algorithm and its application to parameter estimation for Gaussian mixture and hidden Markov models. Technical Report TR-97-021, ICSI, 1997.

[2] Jeff Bilmes. On soft evidence in bayesian networks. Technical Report UWEETR-2004-0016, University of Washington, Dept. of Electrical Engineering, 2004. https://www.ee.washington.edu/techsite/papers/refer/UWEETR-2004-0016.html.

[3] Jeff Bilmes and Chris Bartels. Graphical model architectures for speech recognition. *IEEE Signal Processing Magazine*, 22(5):89–100, September 2005.

[4] Jeff A. Bilmes. *Dynamic Graphical Models and the Graphical Models Toolkit (GMTK)*. University of Washington, Seattle, 2014.

[5] John Ellery Clark, Colin Yallop, and Janet Fletcher. An introduction to phonetics and phonology, 3rd Edition. 2007.

[6] John S Garofolo, Lori F Lamel, William M Fisher, Jonathon G Fiscus, and David S Pallett. Darpa timit acoustic-phonetic continous speech corpus cd-rom. nist speech disc 1-1.1. *NASA STI/Recon Technical Report N*, 93:27403, 1993. Available at http://catalog.ldc.upenn.edu/LDC93S1.

[7] D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge Univ Press, 1997.

[8] Andrew K. Halberstadt and James R. Glass. Heterogeneous acoustic measurements for phonetic classification, 1997.

[9] K. Lee and H. Hon. Speaker-independent phone recognition using hidden markov models. *Computer Science Department. Paper 1769*, 1988.

[10] Paul Mermelstein. Distance measures for speech recognition, psychological and instrumental. *Pattern recognition and artificial intelligence*, 116:374–388, 1976. Introduced MFCCs, and defined the term.

[11] Amar Subramanya and Jeff Bilmes. Soft-supervised learning for text classification. In *Empirical Methods in Natural Language Processing (EMNLP)*, Honolulu, Hawaii, October 2008.

[12] Steve Young, Gunnar Evermann, Mark Gales, Thomas Hain, Dan Kershaw, Xunying Liu, Gareth Moore, Julian Odell, Dave Ollason, Dan Povey, et al. The htk book (for htk version 3.4). *Cambridge university engineering department*, 2(2):2–3, 2006.