Lab 2: HMM's and You

EECS E6870: Speech Recognition

Due: October 17, 2012 at 6pm

SECTION 1

Overview

Hidden Markov Models are a fundamental technology underlying almost all of today's speech recognition systems. They are simple and elegant, and yet stunningly powerful. Indeed, they are often pointed to as evidence of *intelligent design* as it is deemed inconceivable that they evolved spontaneously from simpler probabilistic models such as multinomial or Poisson distributions.

The goal of this assignment is for you, the student, to implement the basic algorithms in an HMM/GMM-based speech recognition system, including algorithms for both training and decoding. For simplicity, we will use individual Gaussians to model the output distributions of HMM arcs rather than mixtures of Gaussians, and the HMM's we use will not contain "skip" arcs (*i.e.*, all arcs have output distributions). For this lab, we will be working with isolated digit utterances (as in Lab 1) as well as continuous digit strings.

The lab consists of the following parts, all of which are required:

- **Part 1: Implement the Viterbi algorithm** — Given a trained model, write the algorithm for finding the most likely word sequence given an utterance.
- **Part 2: Implement Gaussian training** — Implement the algorithm for reestimating the means and variances of a Gaussian.
- **Part 3: Implement the Forward-Backward algorithm** — Write the forward and backward algorithms needed for training HMM's.
- **Part 4: Train various models from scratch, and evaluate them on various digit test sets** — Use the code developed in the earlier parts to run a bunch of experiments.

All of the files needed for the lab can be found in the directory `/user1/faculty/stanchen/e6870/lab2/`. Before starting the lab, please read the file `lab2.txt`; this includes all of the questions you will have to answer while doing the lab. Questions about the lab can be posted on Courseworks (`https://courseworks.columbia.edu/`); a discussion topic will be created for each lab.

Please make liberal use of the Courseworks discussion group for this lab, as judging from the previous times we've given the course, it's nontrivial.

SECTION 2

Part 1: Implement the Viterbi algorithm

In this part, you will be implementing the interesting parts of a simple HMM *decoder*, *i.e.*, the program that computes the most likely word sequence given an utterance. We have pre-trained the observation probabilities of an HMM on data consisting of isolated digits, and this is the model you will be decoding with.

2.1 Background

As discussed in the slides for Lecture 5 (10/8), we can use an HMM to model each word in a vocabulary. For the lab, we use the same HMM topology and parametrization as in the slides. That is, for each word, we compute how many phonemes its canonical pronunciation has, and use three times that many states plus a final state. We line up the states linearly, and each (non-final) state has an arc to itself and to the next state. We place output distributions on arcs (as in the slides) rather than states (as in the readings), and each output distribution is modeled using a GMM. For each state, the GMM on each of its outgoing arcs is taken to be the same GMM.

For example, consider the word `TWO` whose canonical pronunciation can be written as the two phonemes `T UW`. Its HMM contains $2 \times 3 = 6$ states plus a final state. Intuitively, the GMM attached to the outgoing arcs of the first state can be thought of as modeling the feature vectors associated with the first third of the phoneme `T`, the second state's GMM models the second third of a `T`, etc. The topology of the HMM can be thought of as accommodating one or more feature vectors representing the first third of a `T`, followed by one or more feature vectors representing the second third of a `T`, etc.

Now, we can use word HMM's to perform isolated word recognition in the same way that recognition is performed with DTW. Instead of a training example (or *template*) for each word, we have a word HMM. Instead of computing an ad hoc distance between each word and the test example, we compute the probability that each word HMM assigns the test example, and select the word that assigns the highest probability. We can compute the total probability an HMM assigns to a sequence of feature vectors efficiently using dynamic programming.

However, this approach doesn't scale well, and we can do better using an alternate approach. Instead of scoring each word HMM separately, we can build one big HMM consisting of each word HMM glued together in parallel, and keep track of which word each part of the big HMM belongs to. Then, we can use the Viterbi algorithm on this big HMM, and do backtracing to recover the highest scoring path. By seeing which word HMM the best path belongs to, we know what the best word is. In theory, doing dynamic programming on this big HMM takes about the same amount of time as the corresponding computation on all of the individual word HMM's, but this approach lends itself better to pruning, which can vastly accelerate computation. Furthermore, this approach can easily be extended beyond isolated word recognition.

For example, consider the case of wanting to recognize continuous digit strings rather than single digits. We can do this by taking the big HMM we would use for single digits, and simply adding an arc from the final state back to the start state. The HMM can now accept digit strings consisting of multiple digits, and we can use the Viterbi algorithm to find the best word sequence in the same way as we did for isolated digits. In this case, the best path may loop through the machine several times, producing several words of output. We use the "one big HMM" framework in this lab.

2.2 The Decoder

For this part of the lab, we supply much of a decoder for you, and you have to fill in one key part. Here's an outline of what the decoder does:

- Load in the big HMM graph to use for decoding as well as the associated GMM's.
- For each acoustic signal to decode:
  - Use the front end from Lab 1 to produce feature vectors.
  - Perform Viterbi on the big HMM with the feature vectors (*).
  - Recover the best word sequence via backtracing.

You have to implement the part with the (*); we do everything else.

Now, here's an overview of the main data structures you need to deal with.

2.3 The Big HMM Graph

We store the big HMM in a structure of type `Graph`. This class is documented at

```
http://www.ee.columbia.edu/~stanchen/fall09/e6870/classlib/classGraph.html
```

Make sure to check out the example for iterating through the outgoing arcs of a state, as you'll need this and it's kind of weird. (The reason for this weirdness has to do with possible future implementations of the class and efficiency.) There is no easy way to find the incoming arcs for a state, but you don't need this for the lab. (This holds also for the Forward-Backward algorithm, even if it's not immediately obvious how.)

The arc class `Arc` is documented at

```
http://www.ee.columbia.edu/~stanchen/fall09/e6870/classlib/classArc.html
```

Note that the transition log prob (usually denoted a_{ij}) for an arc `arc` can be accessed via `arc.get_log_prob()`, and the observation log prob (usually denoted $b_{ij}(t)$ or some such) can be gotten at through `arc.get_gmm()`. Specifically, `arc.get_gmm()` returns an integer index for a GMM in a set of GMM's, which are held in an object of type `GmmSet`:

```
http://www.ee.columbia.edu/~stanchen/fall09/e6870/classlib/classGmmSet.html
```

A `GmmSet` corresponds to an *acoustic model*. It holds a set of GMM's, one for each primitive acoustic unit, numbered from 0 upwards. To compute the observation probability for an arc at a frame, one would compute the probability of the corresponding GMM for the associated feature vector. However, we have precomputed the log prob of each GMM for each frame and stuck it in the matrix `gmmProbs`, so all you have to do is look up the observation probabilities from here. Incidentally, arcs can also hold a word label, *e.g.*, `arc.get_word()`; this information is needed to recover the word sequence associated with an HMM path.

Aside: in addition to having a start state (the state that all legal paths much start in), an HMM can also have final states, states that legal paths must end in. In our implementation, a graph can have multiple final states, and each final state can have a "final probability" that is multiplied in when computing the probability of a path ending there. However, you don't have to worry about this, because we supply all of the code dealing with final states.

2.4 The Dynamic Programming Chart

When doing dynamic programming as required by all three main HMM tasks (likelihood computation, Viterbi, Forward-Backward), you need to fill in a matrix of values, *e.g.*, forward probabilities or backtrace pointers or the such. This matrix of values is sometimes referred to as a *dynamic programming chart*, and the tuple of values you need to compute at each location in the chart is sometimes called a *cell* of the chart. Appropriately, we define a structure named `VitCell` that can store the values you need in a cell for Viterbi decoding, and allocate a matrix of cells for you in a variable named `chart` for you to fill in for the Viterbi computation. The `VitCell` class is documented at:

```
http://www.ee.columbia.edu/~stanchen/fall09/e6870/classlib/classVitCell.html
```

For Viterbi, you should fill in the log probability and the backtrace arc ID for each cell in the chart.

One thing to note is that instead of storing probabilities or likelihoods directly, we will be storing log probabilities (base e). This is because if we store probabilities directly, we may cause numerical underflow. For more information, read Section 9.12 (p. 153) in Holmes!!! **We're not kidding: if you don't understand the concepts in section 9.12 before starting, you're going to be in a world of pain!!** (The readings can be found on the web site.) For example, we would initialize the log prob for the start state at frame 0 to the logprob value 0, since $\ln 1 = 0$. If we want to set a value to correspond to the probability 0, we would set it to the logprob $-\infty$, or to the constant `g_zeroLogProb` that we have provided which is pretty close. Hint: you may be tempted to convert log probabilities into regular probabilities to make things clearer in your mind, but resist the temptation! The reason we use log probabilities is because converting to a regular probability may result in an underflow.

2.5 What You're Supposed to Do

To prepare for the exercise, create the relevant subdirectory and copy over the needed files:

```
mkdir -p ~/e6870/lab2/
chmod 0700 ~/e6870/
cd ~/e6870/lab2/
cp -d /user1/faculty/stanchen/e6870/lab2/* .
```

Be sure to use the `-d` flag with `cp` (so the symbolic links are copied over correctly).

If working in C++, your job in this part is to fill in the section between the markers `BEGIN_LAB` and `END_LAB` in the file `lab2_vit.C`. Read this file to see what input and output structures need to be accessed. To compile this program, type `make lab2_vit`, which produces the executable `lab2_vit`. (This links in several files, including `util.C`, which includes I/O routines and the `GmmSet` and `Graph` classes; `front_end.o`, which holds the compiled front end from Lab 1; and `main.C`, which holds a minimal `main()` function.) If working in Java, the corresponding file to edit is `Lab2Vit.java`; compile via `make Lab2Vit`, which produces the file `Lab2Vit.class`. A summary of what this program does is provided in Section 2.2.

For testing, we have created a decoding graph and trained some GMM's appropriate for isolated digit recognition. The big HMM or decoding graph used in this run consists of an

(optional) silence HMM followed by each digit HMM in parallel followed by another (optional) silence HMM. To run `lab2_vit` (or `Lab2Vit.class` if `lab2_vit` is absent) on a single isolated digit utterance, run the script

```
lab2_p1a.sh
```

You can examine this script to see what arguments are being provided to `lab2_vit`. In addition to outputting the decoded word sequence to standard output, it also outputs the contents of the dynamic programming chart (first log probs, then arc ID's) to `p1a.chart`. Each row corresponds to a different frame, and each column corresponds to a different state. The target output can be found in `p1a.chart.ref`. You should try to match the target output more or less exactly, modulo arithmetic error. In a later part of the lab, you will be using this decoder to decode with models trained using the code you develop in the following sections.

The instructions in `lab2.txt` will ask you to run the script `lab2_p1b.sh`, which runs the decoder on a test set of ten single digit utterances.

SECTION 3

Part 2: Implement Gaussian training

In this part of the lab, you will implement the statistics collection needed for reestimating a Gaussian distribution as well as the actual reestimation algorithm. This will involve filling in a couple methods of the class `GmmStats` in `gmm_util.C` or `GmmStats.java`. All of the Gaussian "mixtures" in this lab will only contain a single Gaussian. Note that we use diagonal-covariance Gaussians here as is common practice, so we need not store a full covariance matrix for each Gaussian, but only the covariances along the diagonal.

In the typical training scenario, you have a training set consisting of audio utterances and the corresponding reference transcripts. You iterate over the training set, where in each training iteration, you first reset all training counts to 0. Then, for each utterance, you run the Forward-Backward algorithm; this gives you a posterior count for each arc at each frame. Given the identity of the GMM/Gaussian on that arc, the posterior count of the arc, and the feature vector for that frame, you have enough information to update the training statistics for that GMM/Gaussian. (This corresponds to the method `GmmStats::add_gmm_count()`.) At the end of the iteration, you use your training statistics to update all GMM/Gaussian means and variances (and mixture weights if we were using GMM's with more than one component). (This corresponds to the method `GmmStats::reestimate()`.) As far as where to find the update equations, you can look at equations (9.50) and (9.51) on p. 152 of Holmes or equations (8.55) and (8.56) on p. 396 of HAH. Hint: remember to use the *new* mean when reestimating the variance. Hint: remember that we are using diagonal-covariance Gaussians, so you only need to reestimate covariances along the diagonal of the covariance matrix. Hint: remember that variances are σ^2, not σ!

Since you won't be implementing the Forward-Backward algorithm until the next part, instead of using the Forward-Backward algorithm in this part of the lab, we use the "Viterbi" version of this algorithm. That is, instead of considering all possible paths in the HMM to compute the posterior count of each arc at each frame, we use Viterbi decoding to find the single most likely path in the HMM. Then, we take the posterior count of each arc in this path to be 1 for the corresponding frame. In fact, this is also a valid training update in the sense that training set likelihood is guaranteed to (weakly) increase. This algorithm is an

instance of the *generalized* EM algorithm. In practice, Viterbi EM training is often used in later stages of training rather than full Forward-Backward, as it is less expensive computationally, and because once models become very sharp, there isn't a huge difference between the two algorithms since FB posteriors will usually be very sharp.

As before, your job in this part is to fill in the sections between the markers `BEGIN_LAB` and `END_LAB` in the appropriate file (`gmm_util.C` or `GmmStats.java`). Read the file to see what input and output structures need to be accessed. To compile this program, type `make lab2_train` for C++ or `make Lab2Train` for Java.

What this program does is first load an acoustic model/set of GMM's. Then, for each input audio utterance, it loads a representation of the best HMM path (or *alignment*) for that utterance computed using Viterbi decoding with some preexisting model. In particular, we represent an alignment by listing the corresponding GMM/Gaussian for each frame in the path. Then, the method `add_gmm_count()` is called for each frame with the corresponding GMM/Gaussian index and feature vector (and posterior count of 1.0). At the end of a training iteration, the method `reestimate()` is called. Finally, the reestimated Gaussian parameters are written to an output file. For now, don't worry about exactly how these GMM's fit into our overall acoustic model; we'll discuss this in the next part.

For testing, we have created a small training set consisting of about twenty isolated digit utterances. The corresponding alignments were created using a model trained using FB on the same training set. The initial model we'll use is one where all the Gaussian means and variances have been set to 0 and 1, respectively. To run one iteration of training on this data, type

```
lab2_p2a.sh
```

You can examine this script to see what arguments are being provided to `lab2_train`. Ignore the output log prob value (which will just be 0). This script will output GMM parameters to the file `p2a.gmm`. The beginning of the file contains a bunch of stuff you can ignore; the actual Gaussian means and variances are at the end of the file. For example, you can do

```
tail -n 20 p2a.gmm
```

to see the last twenty lines of the file. Each line corresponds to a single Gaussian and for each dimension lists the mean and variance. The target output can be found in `p2a.gmm.ref`. You should try to match the target output exactly, modulo arithmetic error. Your Gaussian reestimation code from this part will be used in the Forward-Backward algorithm you will be implementing in the next part of the lab.

The instructions in `lab2.txt` will ask you to run the script `lab2_p2b.sh`, which runs the trainer on the same data set but with a different input alignment.

SECTION 4

Part 3: Implement the Forward-Backward algorithm

4.1 The Plan

OK, now we're ready to tackle Forward-Backward training. The goal of Forward-Backward training is to create good models for decoding (*i.e.*, Part 1). Let's get specific about all the pa-

rameters in our model that need to be trained. In this lab, our decoding vocabulary consists of twelve words (`ONE` to `NINE`, `ZERO`, `OH`, and silence) consisting of a total of 34 phonemes. We use 3 HMM states for each phoneme (not including final states), giving us 102 states total. Each of these states has two outgoing arcs that we need to assign probabilities to, and a GMM/Gaussian that we need to estimate. For simplicity, we're going to ignore transition probabilities in this part (by setting them all to 1). In practice, transition probabilities have minimal effect on performance. To estimate the parameters of our 102 Gaussians, we'll be using the Gaussian training code completed in the last part.

To see the Gaussian parameters used in Part 1, look at the file `p018k7.22.20.gmm`. Do `tail -n 102 p018k7.22.20.gmm` to see the means and variances of all 102 Gaussians. The mapping from words to Gaussian indices is arbitrary; *e.g.*, the word `EIGHT` is assigned Gaussians 0 through 5, and the silence word is assigned Gaussians 99 to 101.

Now, we can use the Forward-Backward algorithm to estimate these parameters. For example, we can initialize all parameters arbitrarily (*e.g.*, 0 means, 1 variances). Then, we can iterate over our training data, reestimating our parameters after each iteration such that the likelihood our model assigns to the training data is guaranteed to increase (or at least not decrease) over the last iteration. (We can also do this using the Viterbi EM algorithm as in the last section, except there's a chicken and egg problem: how do we come up with a model to generate the initial alignment required for training?)

What we do in each iteration is the following:

- Initialize all counts to 0.
- Loop through each utterance in the training data:
  - Use a front end to produce feature vectors.
  - Load an HMM created by splicing together the word HMM's for each word in the reference transcript for the utterance.
  - Run Forward-Backward on this HMM and the acoustic feature vectors to update the counts.
- Use the counts to reestimate our parameters.

Now, the Forward-Backward algorithm can be decomposed into two parts: computing the posterior probabilities of each arc a at each frame t, usually denoted something like $\gamma(a,t)$; and using these $\gamma(a,t)$ values to update the counts you need to update. You've already done the latter in the last section, so all we need to do here is figure out how to compute $\gamma(a,t)$, the posterior count/probability for each arc at each frame.

4.2 What You're Supposed to Do

In this part, you will be implementing the forward and backward passes of the Forward-Backward algorithm to calculate the posterior probability of each arc at each frame. In particular, this corresponds to filling in sections in the file `lab2_fb.C` or `Lab2Fb.java`.

For this lab, you will only need to calculate the posterior counts of *arcs* and not *states*, since both transition probabilities and observations are located on arcs in this lab. This convention agrees with the slides from Lecture 4, but not the readings, so be careful. More precisely, you need to calculate the values $c(S \xrightarrow{x} S', t)$ (which are another name for the $\gamma(a,t)$ values) (see around slide 83, say, of lecture 4).

Specifically, you need to write code that fills in the forward (log) probability for each cell in the dynamic programming chart. The cell class, `FbCell`, is documented at

```
http://www.ee.columbia.edu/~stanchen/fall09/e6870/classlib/classFbCell.html
```

Then, we provide code to compute the total (log) probability of the utterance, and code that initializes the backward (log) probability for each state for the last frame in the chart. Then, you need to fill in code that does the rest of the backward algorithm, and that computes the posterior counts. Once you have the posterior count $c(S \xrightarrow{x} S', t)$ for an arc at a frame, you must record this count in an object of type `GmmCount` and append this to the vector `gmmCountList`; an example of how to do this is given in the code. The class `GmmCount` is documented at

```
http://www.ee.columbia.edu/~stanchen/fall09/e6870/classlib/classGmmCount.html
```

After the forward-backward function completes for an utterance, `gmmStats.add_gmm_count()` will be called for each element in `gmmCountList` to update the Gaussian training statistics (except that posterior counts below 0.001 are not forwarded for efficiency's sake). After iterating through the whole training set, `gmmStats.reestimate()` will be called to update the GMM parameters.

Hint: you'll probably want to use the function `add_log_probs()` (accessed as `asr_lib.add_log_probs()` in Java) documented at

```
http://www.ee.columbia.edu/~stanchen/fall09/e6870/classlib/util_8H.html
```

Read section 9.12.2 (p. 154) of the Holmes in the assigned readings to see what this is doing. **Note:** you should place a *regular* probability in each `GmmCount` object, not a *log* probability!

To compile, type `make lab2_fb` for C++ and `make Lab2Fb` for Java. To run this trainer on a single isolated digit utterance, run

```
lab2_p3a.sh
```

This script outputs to `p3a_chart.dat` the matrix of forward probabilities, the matrix of backward probabilities, and the matrix of posterior counts for each GMM at each frame. Each line in each matrix corresponds to a frame; for the F+B probs, each column maps to a different state in the graph, and for the posteriors, each column maps to a different GMM. The target output can be found at `p3a_chart.ref`. You should try to match the target posteriors exactly, modulo arithmetic error. (You can have different values for some forward and backward entries than the target, and still have a correct implementation as long as the posteriors match.) To visualize the posterior probs for a given GMM over an utterance, you can run `octave` and type something like

```
load p3a_chart.dat
plot(utt2a_post(:,100))
```

to see the posteriors for the GMM with index 99 (the Gaussian associated with the first state of the silence word). (This requires that you are running X Windows and that you run "`ssh`" with the "`-X`" flag.) (Matlab won't work for this unless you munge the data file some first.)

To test that training is working correctly, you can run your trainer for a single iteration on a training set of about twenty single digit utterances by typing

```
lab2_p3b.sh
```

This outputs the reestimated GMM's to `p3b.gmm`. The target output can be found at `p3b.gmm.ref`.

Once you think things are working, run the script

```
lab2_p3c.sh
```

This script reestimates Gaussian parameters by performing twenty iterations of the forward-backward algorithm and outputs the average logprob/frame (as computed in the forward algorithm) at each iteration. The training set is twenty single digit utterances. If your implementation of the FB algorithm is correct, this logprob should always be (weakly) increasing and should look like it's converging. This script will output trained Gaussian parameters to the file `p3c.gmm`. Finally, decode some (isolated digit) test data with this acoustic model by running the script

```
lab2_p3d.sh
```

This should match the output of `lab2_p1b.sh` from Part 1, as the model provided for that part was trained on the same training set.

SECTION 5

Part 4: Train various models from scratch, and evaluate them on various digit test sets

In this section, we run our trainer/decoder on some larger data sets and look at continuous digit data (consisting of multiple connected digits per utterance) in addition to isolated digits. First, if you're running C++, it's a good idea to recompile your programs with optimization, like so:

```
make clean
OPTFLAGS=-O2 make lab2_fb lab2_vit
```

If you're running Java, too bad.

First, let us see how our HMM/GMM system compares to the DTW system we developed in Lab 1 on isolated digits. We created a test set consisting of 11 isolated digits from each of 56 test speakers, and ran DTW using a single template for each digit from a pool of 56 training

speakers (using a different training speaker for each test speaker). This yielded an error rate of 18.8%.

Run the following script:

```
lab2_p4a.sh | tee p4a.out
```

This first trains a model on 100 isolated digit utterances (with five iterations of FB), and then decodes the same test set as above; then, trains a model on 300 utterances and decodes; then, trains a model on 1000 utterances and decodes. After decoding, the script runs `p018h1.calc-wer.sh` to compute the word-error rate of the decoded output. See how the word-error rate varies according to training set size. The trained models are saved in various files beginning with the prefix `p4a`.

Next, run the following script:

```
lab2_p4b.sh | tee p4b.out
```

This takes the 300-utterance model output by `lab2_p4a.sh` and decodes connected digit string data (rather than isolated digits) with this model. It also trains a model on 300 connected digit sequences and decodes the same test set.

Finally, we include an experiment to look at the effect of transition probabilities. Run the following script:

```
lab2_p4c.sh | tee p4c.out
```

We use the model trained on 100 isolated digit utterances produced by `lab2_p4a.sh` for decoding, except that we also include trained transition probabilities, by embedding them in the big HMM used for decoding.

SECTION 6

What is to be handed in

You should have a copy of the ASCII file `lab2.txt` in your current directory. Fill in all of the fields in this file and submit this file using the provided script.