EE516 Speech Processing and Recognition **University of Washington**
Winter 2018 **Dept. of Electrical Engineering**

Homework 5. Due Tuesday March 13th, 2018, 11:59pm Electronically

Prof: J. Bilmes <bilmes@ee.washington.edu> *Monday Feb 26th, 2018*
TA: Dhanush Kannangola <dhanush@uw.edu>

All homework is due electronically via the link https://canvas.uw.edu/courses/1188387/assignments. Note that the due dates and times might be in the evening. Please submit a PDF file. Doing your homework by hand and then converting to a PDF file (by say taking high quality photos using a digital camera and then converting that to a PDF file) is fine, as there are many jpg to pdf converters on the web. Some of the problems below will require that you look at some of the lecture slides at our web page (http://j.ee.washington.edu/~bilmes/classes/ee516_winter_2018/).

There are 2 problems, for a maximum total of 380 points spanning 9 pages.

Problem 1. Small Probabilities (30 points)

Lets say that you have a set of probabilities $p_1, p_2, \ldots, p_T$ and you multiply them all together in the following form:

$$p_{\text{prod}} = \prod_{t=1}^{T} p_t \tag{1}$$

then it is the case that p_{prod} is going to get small exponentially fast in T. The reason is that for each t, $0 \leq p \leq 1$ and so assuming that all of the probabilities are less than one, this is an exponentially decreasing sequence. For example, of $p_t = 0.5$ for all t, then $p_{\text{prod}} = 0.5^T$ which is a very small positive number.

When we construct $\alpha_t(j) = p(x_1, \ldots, x_t, Q_t = j)$ in the HMM alpha recursion, we can easily see that as t gets big, the probability value of $\alpha_t(j)$ gets smaller and smaller. In fact, the typical value of $\alpha_t(j)$ gets exponentially smaller in t — that is $\alpha_t(j) \approx O(r^{-t})$ where r is the number of states. As a simple example of this, suppose for the moment that all variables are independent, so that $p(x_1, \ldots, x_t, Q_t = j) = p(Q_t = j) \prod_{\tau=1}^{t} p(x_t)$ (it is very important to realize that this is **not** the case with an HMM, but we are doing this here just as an example). Then if each $p(x_t) = 0.5$ then we see that $\alpha_t(j) \propto 0.5^t$ which gets exponentially small in t, like the first example above. The fact that an HMM is not independent can make this either better or worse, depending on the distribution. I.e., it could be that $x_1, x_2, \ldots$ are such that the probability does not decrease as fast (it's even mathematically possible for $p(x_1, \ldots, x_t, Q_t = j) = p(Q_t = j)$), but it is very common in practice that this decreases very fast (much faster than the additional factor of 0.5 at each time step, as suggested above). This is especially true when the HMM uses Gaussian distributions, as is done often in in speech recognition systems and when the HMM has many states, also common in ASR systems.

Similarly, $\beta_t(j)$ in the beta recursion gets smaller as t goes down from T, that is $\beta_t(j)$ gets exponentially small with $T - t$.

Hence it is not long, even for moderate values of t, before we will run out of numeric precision. The smallest single precision 32-bit IEEE floating point (FP) number is approximately 10^{-33} the smallest double precision 64-bit IEEE floating point is about 10^{-308}. So if $r = 10$, this means that an HMM can not be longer than about 308 frames before running out of numeric precision, and this is when using double precision arithmetic. Hence, we are unable to practically compute $\alpha_t(j)$ and $\beta_t(j)$ for all but very small values of T, unless we do something smarter.

There are many possible solutions and which solution is desirable depends on which operations one wishes to perform. For example, one may wish to compute $p(\bar{x}_{1:T})$, or one may only wish to compute the

posteriors mentioned above (say for parameter learning), or one may wish to compute the Viterbi path (for decision making). Lets consider each in turn.

First, to compute the probability of evidence $p(\bar{x}_{1:T})$, this quantity itself presents a problem. As T gets larger, $p(\bar{x}_{1:T})$ gets (typically exponentially in T) smaller and so would not have enough precision to even store the answer to this query. One solution is to compute $\log p(\bar{x}_{1:T})$ directly, and this is possible using log arithmetic which we describe below.

To compute a quantity like $p(q_t|\bar{x}_{1:T})$ or $p(q_t, q_{t-1}|\bar{x}_{1:T})$ we need both a numerator such as $p(q_t, \bar{x}_{1:T})$ (or $p(q_t, q_{t-1}, \bar{x}_{1:T})$) and a denominator $p(\bar{x}_{1:T})$. Each of these themselves would have values too small to store (i.e., numeric underflow issues would cause them to be truncated to zero in most finite-precision arithmetic systems, including IEEE floating point, which is the standard). If we were able to compute quantities that are proportional to each (say $c_T p(q_t, \bar{x}_{1:T})$ and $c_T p(\bar{x}_{1:T})$) where both numerator and denominator have the same constant c_T that also scales roughly exponentially with T, then we should be able to compute the desired quantities. Note that the proportional constant c_T is a function of T which means that it scales as the sequence gets longer. Moreover, this strategy never actually computes $p(q_t, \bar{x}_{1:T})$ but only quantities that are a known proportionality constant away from it.

Lastly, to compute the Viterbi path, we need to produce a structure that allows us to obtain the same quantity but without needing to produce such small values. This is also possible.

There are two general strategies for dealing with the above problems: 1) scaling the scores, and 2) log arithmetic, and we'll discuss them in this order.

Consider another form of scaled $\alpha_t(j)$ where at each time step in the recursion we form:

$$\bar{\alpha}_t(q) = \frac{\sum_r p(x_t|q)p(q|r)\bar{\alpha}_{t-1}(r)}{d_t} \tag{2}$$

Hence, we see that

$$\bar{\alpha}_t(q) = \frac{\alpha_t(q)}{\prod_{\tau=1}^{t} d_\tau} \tag{3}$$

and if d_t is chosen wisely then $\bar{\alpha}_t(j)$ will have a reasonable scale. One choice of d_t would be the maximum at each step as in

$$d_t = \max_q \left[\sum_r p(x_t|q)p(q|r)\bar{\alpha}_{t-1}(r) \right] \tag{4}$$

and this tends to work well in practice. But other (and even easier) choices for d_t would be $d_t = 1/|\mathsf{D}_Q|$, or

$$d_t = \sum_q \left[\sum_r p(x_t|q)p(q|r)\bar{\alpha}_{t-1}(r) \right] \tag{5}$$

(which of course would then give us $\hat{\alpha}_t(j)$) or even the median of the quantity. In any event, we'll get a resulting version of α that typically would not decrease exponentially in T.

For the backwards recursion, there are similar options. That is, each backwards recursion would produce $\bar{\beta}$ and would be of the form

$$\bar{\beta}_t(q_t) = \frac{\sum_{q_{t+1}} \bar{\beta}_{t+1}(q_{t+1})p(x_{t+1}|q_{t+1})p(q_{t+1}|q_t)}{e_{t+1}} \tag{6}$$

where we could use

$$e_{t+1} = \max_{q_t} \left[\sum_{q_{t+1}} \beta_{t+1}(q_{t+1})p(x_{t+1}|q_{t+1})p(q_{t+1}|q_t) \right] \tag{7}$$

or

$$e_{t+1} = \sum_{q_t} \left[\sum_{q_{t+1}} \beta_{t+1}(q_{t+1}) p(x_{t+1}|q_{t+1}) p(q_{t+1}|q_t) \right] \tag{8}$$

or something else. In each case, we have the relationship

$$\bar{\beta}_t(q_t) = \frac{\beta_t(q_t)}{\prod_{\tau=t+1}^{T} e_\tau} \tag{9}$$

and so we see that $\hat{\beta}_t(q_t)$ is likely going to have a well behaved dynamic range.

With $\bar{\alpha}_t$ and $\bar{\beta}_t$ it is easy to get posteriors. For example,

$$p(q_t|\bar{x}_{1:T}) = \frac{\alpha_t(q_t)\beta_t(q_t)}{\sum_{q_t} \alpha_t(q_t)\beta_t(q_t)} = \frac{\prod_{\tau=1}^{t} d_\tau \bar{\alpha}_t(q_t) \prod_{\tau=t+1}^{T} e_\tau \bar{\beta}_t(q_t)}{\sum_{q_t} \prod_{\tau=1}^{t} d_\tau \bar{\alpha}_t(q_t) \prod_{\tau=t+1}^{T} e_\tau \bar{\beta}_t(q_t)} = \frac{\bar{\alpha}_t(q_t)\bar{\beta}_t(q_t)}{\sum_{q_t} \bar{\alpha}_t(q_t)\bar{\beta}_t(q_t)} \tag{10}$$

It is just as easy to get the edge posterior $p(q_t, q_{t-1}|\bar{x}_{1:T})$.

Scaling of HMM scores is critically important for most if not all HMM systems. For online inference and Kalman-style filtering to be used in streaming applications, scaling the scores is essential to ensure that the unboundedly long sequences that we might encounter will not run out of numeric precision.

We have not yet specified how to compute $p(\bar{x}_{1:T})$. In this case, the value we wish to compute itself is not within the limited range of 64-bit IEEE floating point arithmetic, so the scaling options mentioned above will not work. In this case, therefore, we need a numeric representation that has enough dynamic range to be able to represent $p(\bar{x}_{1:T})$. One possibility is log-arithmetic where rather than working with probabilities $0 \leq p \leq p$ we work with log-probabilities $-\infty \leq p \leq 0$. More generally, since a probability score might actually be a likelihood (e.g., a Gaussian could be used for $p(x_t|q_t)$), rather than scores of the form $0 \leq p < \infty$, we have log scores of the form $-\infty \leq \log p < \infty$.

So let $a = \log p_a$ and $b = \log p_b$ be two log probabilities. In general, there are two operations we may wish to compute with them: 1) multiplication $p_a p_b$ and addition $p_a + p_b$. Multiplication of two log probabilities is particularly easy, since $\log(p_a p_b) = \log p_a + \log p_b = a + b$, so log arithmetic turns multiplication into addition.

Addition of the probabilities $p_a + p_b$ is a bit more tricky however. An obvious way of doing this would be to do convert them back to probabilities from log probabilities, do the addition, and then re-take the log as represented by $a \boxplus b = \log(\exp(a) + \exp(b))$ where $a \boxplus b$ is meant to read "a log-add b". Unfortunately, this does not at all solve the scaling problem, since it might never be the case that we can actually represent p_a. For example, if $a = \log(\alpha_t(q))$ for a large value of t, then just taking $\exp(a)$ would result in zero. In fact, using standard numerical math libraries, $\exp(a) = 0$ roughly whenever $a < -745$, which means once again that when T is large, this approach will fail. We need therefore a smarter strategy for computing a log add.

Now using log arithmetic would not mean that we want to avoid using standard IEEE floating point formats for representing numbers. That is, we still want to be able to store values a and b in standard floating point registers (32 or 64 bit) and use hardware-based floating point operations whenever possible – this is to ensure that our log arithmetic operations are as fast as possible.

Throughout this discussion, we'll assume that the base of the logarithm is e — in general, the base can be arbitrary but we need to be sure that we have optimized routines that, for base b, we can quickly compute $\log_b(p)$ and b^p, and base e is a good bet. On the other hand, base 10 might be easier for debugging (since we always know we're dealing with some power of 10, something that humans might find intuitively more facile than powers of e). One option is to use base 10 for debugging and once done switch to base e if base e is faster. Again, in below, we'll assume base e.

The first thing we need is a representation when $p_a = 0$, the log of which is $-\infty$. While IEEE FP arithmetic does have a special representation of $-\infty$, that usually indicates an error and for some settings

of the FP registers, anytime a $-\infty$ is produced will cause an operating systems trap to indicate the error. Instead, we use a special value ℓ_0 which is our representation of log zero. This value should be smaller than the smallest possible value we are needing to represent, so something like $\ell_0 = -1 \times 10^{30}$ would be reasonable for single precision (since it can still be represented with only 32 bits). What this says then is that we are not able to represent numbers smaller than $e^{-10^{30}}$ (base e). This of course does not mean we'll ever attempt to evaluate $e^{-10^{30}}$ (we can't evaluate anything smaller than about e^{-745} as mentioned above) — what it does mean is that any non-zero probability value has log probability a that satisfies $a \geq -10^{30}$, and if for some reason the log probability falls below this value, it is considered for all intents and purposes as log-zero.

Also, $e^{-\infty} = 0$ but we don't have a representation for real $-\infty$ so we are approximating $-\infty$ as ℓ_0; we could write in some very informal way that $\ell_0 \approx -\infty$ only in that we will treat ℓ_0 as $-\infty$, and correspondingly we will treat $-\ell_0$ as ∞. Hence, we could also write in a very informal way that $-\ell_0 = \tilde{\infty}$, our approximation of ∞. Also, since $-\ell_0 = \tilde{\infty}$, then $\log(-\ell_0)$ is like ∞ but in the log domain, and $-\log(-\ell_0)$ is like $-\infty$ but in the log domain. Thus, any time two log-domain numbers a, b have a positive difference $a - b > 0$ more than $\log(-\ell_0)$ (i.e., $a - b > \log(-\ell_0) = \log(\tilde{\infty})$ then a is essentially "infinitely" larger than b and so any addition of a and b could safely numerically ignore b and say the answer is just a. Similarly, any time two log-domain numbers a, b have a non-positive difference $a - b \leq 0$ less than $-\log(-\ell_0)$ (i.e., $a - b \leq -\log(-\ell_0) = -\log(\tilde{\infty})$ then a is essentially "infinitely" smaller than b and so any addition of a and b could safely numerically ignore a and say the answer is just b.

So given this, how do we compute $a \boxplus b$? Assume that $a \leq b$ and if not, swap a and b so that it is true (we can do this without any loss of generality). Therefore, we can compute the difference so that the following is true $d = a - b \leq 0$. The first thing we check when wanting to compute $a \boxplus b$ is this quantity d. If $d < -\log(-\ell_0)$ then b is significantly larger than a, so much so that numerically $\log(p_b/p_x) > \log(-\ell_0)$ or $p_b/p_a > -\ell_0 = \tilde{\infty}$ since $\ell_0 = -\tilde{\infty}$, our approximation to $-\infty$ as mentioned above. Thus, $d < -\log(-\ell_0)$ essentially means that $p_b > \infty \cdot p_a$. In other words, p_b is so much larger than p_a that it is not worth even adding p_a to p_b and we just say that $a \boxplus b = b$.

If it is not the case that $d < -\log(-\ell_0)$ then we need to do the work to combine both a and b without ignoring either. Our goal therefore is to compute $c = a \boxplus b$. Consider the following derivations:

$$
\begin{aligned}
c &= \log(p_b + p_a) && (11)\\
&= \log(\exp(b) + \exp(a)) && (12)\\
&= \log(\exp(b) + \exp(b + a - b)) && (13)\\
&= \log\big(\exp(b)(1 + \exp(a - b))\big) && (14)\\
&= b + \log\big(1 + \exp(a - b)\big) && (15)
\end{aligned}
$$

Using the last expression in Equation (15) to compute c is beneficial for a number of reasons:

1. It uses only two two rather than three calls to transcendental functions as would Equation (11).

2. It is amenable to integer representation of the log probabilities meaning that a and b can be quantized to integer values

3. Whether a or b are integer valued or not, the operation $\log(1 + \exp(n))$ can be done efficiently via a table lookup via a rounded value of $a - b$. For example, we can compute a table that stores values for $\log(1 + \exp(\alpha_i))$ which can be looked up with integer i. We can set $\alpha_i \leftarrow \frac{-\log(-\ell_0)}{M} i$ where M is the size of the table.

   To create a table lookup, we would create an array v of a given size, lets say R, and we would uniformly space items in the array, for $i = 0, \ldots, R$ with i^{th} value being $v(i) = \log(1 + \exp(-i * \log(-\ell_0)/R))$. Defining an increment of $\epsilon = (R - 1)/(-log(-\ell_0))$, Equation (15) would become $b + v([\epsilon * (a - b)])$ where [] is the round-to-nearest-integer operator.

4. It is numerically much better behaved than the original expression since. For example, if $a \approx b$ and they are both less than -745, we'll get the much more accurate $b + \log(2)$ rather than $\log(0)$.

5. If a happens to be much smaller than b, then we don't loose precision by needlessly evaluating $b = log(exp(b) + 0)$

6. Perhaps most importantly of all, if a or b are less than -745 (which will easily happen), then $c = log(exp(a) + exp(b))$ would at best produce garbage, but this last expression in Equation (15) would not.

It is elucidating moreover to consider the function $f(\alpha) = \log(1 + \exp(\alpha))$ for $\alpha \leq 0$. First, we have that $f'(\alpha) = \frac{\exp(\alpha)}{1+\exp(\alpha)} = \frac{1}{1+\exp(-\alpha)}$ which is a sigmoid function. Since $f''(\alpha) = f'(\alpha)(1 - f'(\alpha)) \geq 0$, for all $\alpha \in \mathbb{R}$, we see therefore that $f(\alpha)$ is everywhere convex. Second, when we look at $f(\alpha)$ for $\alpha \leq 0$, and since $\log(1 + \epsilon) \approx \epsilon$ for $\epsilon \approx 0$, we see it appears as a smooth exponential decaying to the left. Hence, it is possible to approximate this function with fast methods more sophisticated then the table lookup mentioned above.

To summarize, here is our high level recipe for log addition (offered in a way that you still need to go through the analysis above to understand why it works to produce the final algorithm).

1. We are given two log-domain probabilities a and b corresponding to probabilities p_a and p_b.

2. First, sort a and b to ensure that $a \leq b$.

3. Compute d as above.

4. Check the condition on d and if it is true, then we have the answer immediately.

5. If the condition on d is not true, go through Equations (11) through (15).

To conclude, log arithmetic is a useful alternative to the issues that arise when working with HMMs on long sequences.

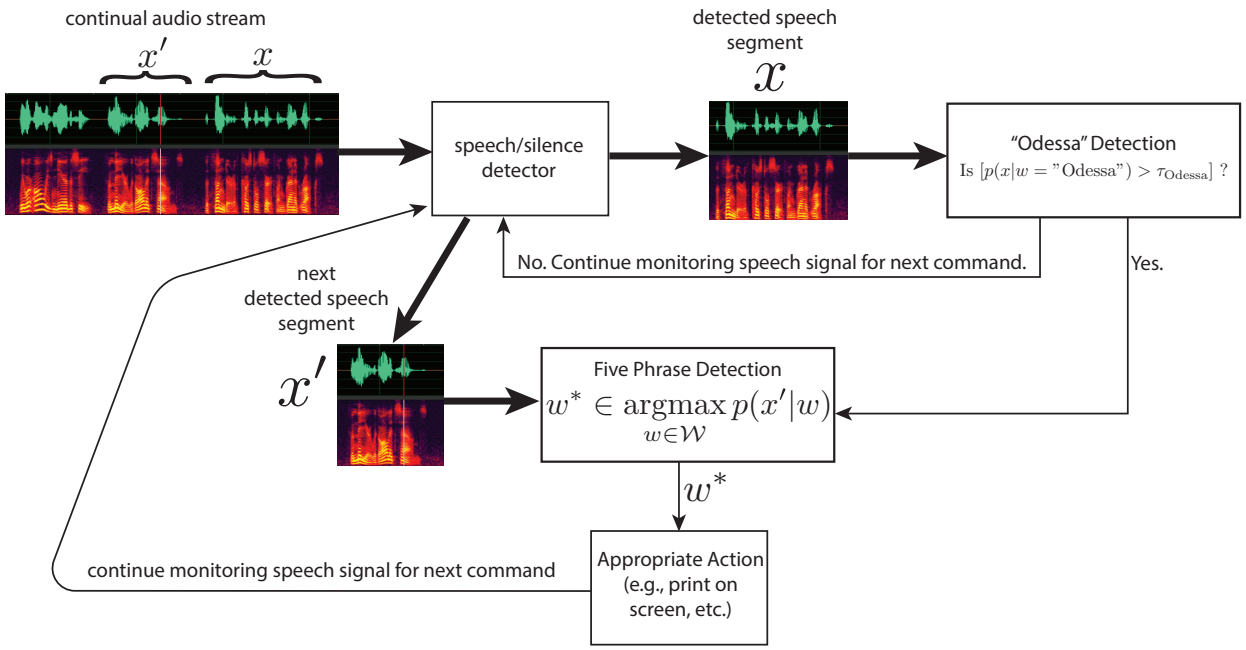
Problem 2. Final Project Problem (350 points)

For the final project, you are to implement isolated phrase HMMs for doing an isolated phrase HMM-based ASR system. This includes the expression of the Markov chain via a transition matrix, the initial state distribution, and the multivariate Gaussian distributions.

The first thing you should do is review lecture slides the previous set of lectures (the ones that involve HMMs). The next thing you should do is read this homework assignment entirely and ensure that you understand things.

The final project will consist of writing in python from scratch a simple speaker dependent, isolated-phrase, whole-word model (i.e., one HMM per phrase), single Gaussian per state, diagonal covariance Gaussian, automatic speech recognition (ASR) system (see further definitions below), using the following six-phrase vocabulary $\mathcal{W} = \{$ "Odessa" , "Turn on the lights", "Turn off the lights", "What time is it", "Play music", "Stop music"$\}$.

The basic model is that your system will continuously monitor audio from your computer and it will use the speech/silence detector you have already written. When the speech/silence detector detects a segment of audio that it thinks is speech, it will send it to the HMM to detect the wake-up phrase, which is "Odessa.". If the score for the wake-up phrase is high enough, then it will expect the next segment of audio to be one of the remaining five commands, i.e., "Turn on the lights", "Turn off the lights", "What time is it", "Play music", and "Stop music". Your system should then indicate in some way which phrase was recognized. Basically, you will produce a separate HMM for each of the phrases to be used as a score. The overall flow of your speech recognizer is described in the following figure.



You will need to use six separate HMMs, one for each phrase, that takes the following form: for each $w \in \mathcal{W}$, we have the probability distribution over a sequence of T MFCC vectors.

$$p(x) = p(x_1, x_2, \ldots, x_T|w) = \sum_{y_1,y_2,\ldots,y_T} p(x_1, x_2, \ldots, x_T, y_1, \ldots, y_T|w) \tag{16}$$

$$= \sum_{y_1,y_2,\ldots,y_T} p(x_1|y_1)p(y_1) \prod_{t=2}^{T} p(x_t|y_t)p(y_t|y_{t-1}) \tag{17}$$

$$= \sum_{y_1=1}^{M_w} \sum_{y_2=1}^{M_w} \cdots \sum_{y_T=1}^{M_w} p(x_1|y_1)p(y_1) \prod_{t=2}^{T} p(x_t|y_t)p(y_t|y_{t-1}) \tag{18}$$

where y_i is a state variable at time t, x_t is an N-dimensional continuous feature vector (MFCCS and their deltas, so $N = 26$) at time t, $w \in \mathcal{W}$ is one of the six phrases, and T is the number of MFCC vectors (i.e., the number of frames in the utterance). Note that T is a constant for a given unknown utterance — i.e., you record some speech, convert it to MFCCS, and then T is the number of resultant frames. For different speech utterances, T will likely be different (i.e., "Turn off the lights" is typically longer than "Stop music", and even two instances of the same word will not be exactly the same length). Lastly, M_w is the number of states in the HMM for word w.

Your HMMs will be over 26-dimensional MFCC feature vectors (the MFCCs and deltas) that you computed a part of homework 4. That is, $x_t \in \mathbb{R}^N$ where $N = 26$.

Here is a description of the other terms from above:

- **Python:** the programming language and system you'll use (it is generally not the fastest language, but for our purposes it will be fine, it also has functions such as `fft`, etc. that you can use). You should start with the python code you've already coded for previous assignments' (e.g., the speech/silence detector was done as part of HW3).

- **speaker dependent:** The system is speaker dependent, meaning that it need not work well on anyone other than you. A speaker independent system is one that works regardless of who is speaking. Note that for the final demo, we will test your system on another person just to see if by chance you've achieved a bit of speaker independence (which would be nice but is not necessary).

- **isolated-word:** An isolated word or isolated phrase ASR system is one where each word or phrase is presented in isolation from other words or phrases, which means that there is silence both before

and after each word or phrase. In fact, in your system, you'll only be recording one phrase at a time, i.e., you'll start recording, the speech detection will detect a segment (which will have a bit of silence before and after the segment and should be included in the segment). You may assume that the stream of speech will consist of phrases separated by silence. Also we will not be testing these in harsh, noisy, or hostile acoustic background environments although you should appreciate that any real system (e.g., Alexa, Google home, etc.) does need to worry about such things.

- **whole-word model:** Our HMMs will not use phones, rather they will use states that are distinct for each phrase. In other words, the states do not have a phonetic interpretation, rather we will just use a separate HMM for each phrase, and there is no sharing of the states or the observation distributions across phrases. In larger speech systems, the states often have some sort of phonetic interpretation and observations are shared over different phrases (but we need not do this here). This will become more clear in the below.

- **single Gaussian per state:** Each state will have a single Gaussian density, i.e., $p(x_t|y_t) = \mathcal{N}(x_t|\mu_{y_t}, C_{y_t})$ where $\mathcal{N}(x_t|\cdot,\cdot)$ is an N-dimensional multivariate Gaussian, and μ_{y_t} is a mean vector, and C_{y_t} is a covariance matrix (which necessarily is positive definite), where y_t is a state index.

- **diagonal covariance Gaussian:** This is a simplification of the Gaussians, namely that C_{y_t} is a diagonal matrix. Since it is also positive definite, this means that C_{y_t} is a diagonal matrix with strictly positive entries along the diagonal and zeros everywhere else (this means, also, that you need **not** allocate storage for all N^2 entries, rather only the N entries along C_{y_t}'s diagonal).

  The distribution $p(x_t|y_t)$ is a Gaussian distribution with a diagonal covariance matrix, so this means that:

$$p(x_t|y_t) = \frac{1}{\sqrt{|2\pi C_{y_t}|}} \exp\left[-\frac{1}{2}(x_t - \mu_{y_t})^\intercal {C_{y_t}}^{-1}(x_t - \mu_{y_t})\right] \tag{19}$$

$$= \frac{1}{(2\pi)^{N/2}\sqrt{|C_{y_t}|}} \exp\left[-\frac{1}{2}(x_t - \mu_{y_t})^\intercal {C_{y_t}}^{-1}(x_t - \mu_{y_t})\right] \tag{20}$$

$$= \frac{1}{(2\pi)^{N/2}\sqrt{\prod_{i=1}^{N} C_{y_t}(i)}} \exp\left[-\frac{1}{2}\sum_{i=1}^{N}\frac{(x_t(i) - \mu_{y_t}(i))^2}{C_{y_t}(i)}\right] \tag{21}$$

  where $|C_{y_t}|$ is the determinant of matrix C_{y_t}, $C_{y_t}(i)$ is the i^{th} diagonal entry in diagonal matrix C_{y_t}, $x_t(i)$ is the i^{th} entry of the N-D vector x_t, and $\mu_{y_t}(i)$ is the i^{th} entry of the N-D mean vector μ_{y_t} (which is a vector index by the state $y_t \in \{1, \ldots, M_w\}$.

  As you can see, the diagonal covariance assumption leads to some nice computational savings for both computing the determinant and the matrix inverse.

  Since you will have six phrases, and M_w is the number of states in the HMM for phrase w, and since you will have one Gaussian per state, you will have total $\sum_{w\in\mathcal{W}} M_w$ number of Gaussians.

  You will need to test out different values for M_w for each w to see what works best, but I'd guess a reasonable starting point would be to set $M_w \approx 10$. Note, again, you need not use the same number of states for each phrase (longer phrases should probably have a few more states). Also remember, that the first state and the last state will probably model silence, presumably your speech recordings used for training data also start/end with silence.

Your project will consist of the following stages:

1. Write code for the data structures and the algorithms of the HMMs in python.

2. Write code that computes the $\alpha_t(y)$ recursion, as expressed in class. Note that sometimes people use the letter q as the state and other times people use the letter y for the state. In our case here, we're using y. This means that the state at time t, y_t is an integer from 1 to M_w. You'll probably want to store, for each HMM, the entire α matrix (which is going to be $M_w \times T$ for HMM w) for each word w.

3. Write code that computes the $\beta_t(y)$ recursion, as expressed in class. You'll probably want to store, for each HMM, the entire β matrix (which is going to be $M_w \times T$ for HMM w) for each word w. Note that you'll need a separate beta matrix for each instance of each word.

4. Note that you'll need a separate α and β matrix for each instance of each word (so if you have a six-phrase vocabulary, and say 10 instances of each word to train on using EM (see below) you'll need to compute 60 α and 60 β matrices, or 120 total matrices). You need not have them all in memory at the same time, during EM training (below), rather you can compute a pair of α and β matrices and keep them around, to compute the needed γ and ξ quantities, before deallocating them and moving on to the next utterance.

5. From either the α or the β matrix, you can compute the quantity $p(x_1, x_2, \ldots, x_T|w) = p(x_{1:T}|w)$.

6. Write code that computes the $\gamma_t(y)$ recursion, as expressed in class. This quantity you probably don't want to store in a matrix as it can be easily computed from the α and the β matrices.

7. Write code that computes the $\xi_t(i, j)$ quantity. This quantity also you probably don't want to store in a matrix as it can be easily computed from the α and the β matrices.

8. Collect a set of training utterances for each word. This can come from an earlier HW. That is, you will have a training set $\mathcal{D}_w$ for each word where

$$\mathcal{D}_w = \left\{ x^1_{1:T^w_1}, x^2_{1:T^w_2}, \ldots \right\} \tag{22}$$

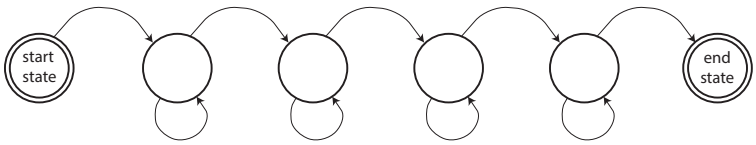
   where T^w_i is the length, in number of frames, of the i^{th} instance of word w. The number of speech utterances $|D_w|$ for word w should probably be about 10-20, but you should experiment with this (you might need to use more to get your system to work better). The full set of training data is referred to as $\mathcal{D} = \cup_w D_w$.

9. Machine learning: Implement the EM algorithm for HMMs, which is described in lecture. The EM algorithm allows you to compute the maximum likelihood estimate of the parameters. The HMM for word w can be written with its parameters as:

$$p(x_{1:T}|w, \lambda_w) \tag{23}$$

   where λ_w are the parameters for HMM w. That is $\lambda_w = (\pi^w, A^w, \{b^w_i(x)\}_i)$ where

   - π is the initial state distribution (a probability distribution over M_w values that in Equation (18) is written as $p(y_1)$),
   - A is a transition matrix which is $M_w \times M_w$ (and that is written as $p(y_t|y_{t-1})$ in Equation (18), recall that $p(y_t|y_{t-1})$ is really shortcut notation, and what we really mean is $p(y_t|y_{t-1}) = p_{Y_t,Y_{t-1}}(Y_t = y_t|Y_{t-1} = y_{t-1}) = p_{Y_t,Y_{t-1}}(Y_t = j|Y_{t-1} = i) = a_{ij}$ when $y_t = j$ and $y_{t-1} = i$, and since a_{ij} is not t-dependent, this is a time-homogeneous Markov chain as discussed in class), You probably want to use a strictly left-to-right sequential order transition matrix which means that $a_{ij} = 0$ whenever $j \notin \{i, i+1\}$ as shown in the following figure which corresponds to $M_w = 4$.



- and $b_i^w(x)$ is the diagonal covariance Gaussian mean vector and covariance matrix for word w's HMM state i (this corresponds to $p(x_t|y_t)$ in Equation (18)).

The EM algorithm estimates:

$$\lambda_w^* \in \operatorname*{argmax}_{\lambda_w} \sum_{j=1}^{|\mathcal{D}_w|} \log p(x^j_{1:T_j^w}|w, \lambda_w) \tag{24}$$

The parameters λ_w^* are the maximum likelihood parameter estimates. The exact equations for how to compute this, based on the α and β matrices, and the γ and ξ quantities are described in lectures 8 and 9.

10. Speech recognition: Speech recognition uses the trained HMMs you computed in the previous steps, and is done as follows. Given a new speech utterance that has been converted to a sequence of MFCC vectors $x_{1:T}$, speech recognition occurs as follows:

$$w^* \in \operatorname*{argmax}_{w \in \mathcal{W}} p(x_{1:T}|w, \lambda_w^*) \tag{25}$$

11. Your python ASR system will run as follows. After prompting the user, you'll start monitoring speech and when a segment is detected, and if the segment is recognized as the wakeup word, will then assume the following segment of audio is a speech segment to be recognized.

What is due:

1. A four page technical writeup describing all of the details of your system, the technical approach you chose, any problems that came up and how did you solve them. Please keep it to four 8.5×11 pages, with a font size no smaller than 10pt.

2. Electronic version of slides (e.g., powerpoint, keynote, etc.) for the five minute talk describing your system.

   It's important that your talk last no more than five minutes, so please practice it to ensure you can get through everything in five minutes.

3. All code for your system in python, and also the speech files you used for training and any other data files that you used. I.e., we should be able to train your system using the HMM EM code that you coded up in python.

4. A short **no-more-than-5-minute** video demonstrating that your system works in your environment (I mentioned in office hours that sometimes changing microphones and/or rooms can hurt the performance of a speech recognition system). Your video can be taken by a smart phone by, say, a friend. Then you can upload it to a private link in youtube and you need to include the youtube URL to the video.

   The video must not be longer than five minutes long, but it can be shorter.