DD2434 Machine Learning, Advanced Course Assignment 2

Thomas Gaddy

January 30, 2018

I. Graphical Models

2.1 Dependencies in a Directed Graphical Model

Consider the Directed Acyclic Graph (DAG) of a DGM shown in Figure 1

Question 1: *Which pairs of variables, not including X, are dependent conditioned on X?*

A simple way to approach this problem is to use the *Bayes Balls Algorithm* in order to determine which pairs of variables are D-separated. Based on this we see that (A, B), (C, E), and (D, F) are all dependent when conditioned on X.

(A, B): The path from A to B meet head-to-head at the observed node X. The path is therefore *unblocked* and A and B are dependent conditioned on X.

(C, E): There is a direct causal link between these two variables and they are therefore dependent, regardless of whether we condition on X or not.

(D, F): Similarly to C and E, this pair of variables is independent regardless of whether we condition on X or not.

All other pairs of variables are conditionally independent given X because the arrows on the path between them either meet head to head at an unobserved node or head-to-tail or tail-to-tail at an observed node.

Question 2: *Which pairs of variables, not including X, are dependent, not conditioned on X?*

We can approach this problem similarly to question 1. Doing so we see that there are seven pairs of variables that are dependent when not conditioned on X: (A, F), (A, E), (B, F), (B, E), (C, E), (F, D), (E, F).

(A, F): The path from A to F can be seen as a chain. Since we do not condition on X this chain is unblocked and the variables are dependent.

(A, E): Same justification as for (A, F)

(B, F): Same justification as for (A, F)

(B, E): Same justification as for (A, F)

(C, E): As explained above, these variables are dependent regardless of whether we condition on X

(F, D): These variables are always dependent

(E, F): The path from E to F can be seen as a "fork" structure. Since we do not condition on X this fork is unblocked and the variables are dependent.

All other pairs of variables are independent when not conditioned at X because the path between them meets head-to-head at an unobserved node.

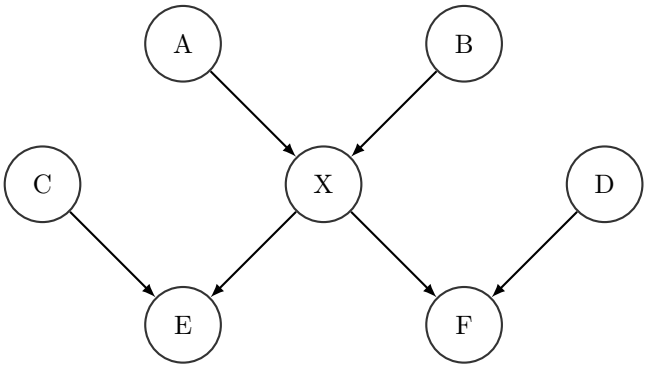

Figure 1: The DAG.

2.2 The Sum-HMM

Here we consider a generative model based on a standard HMM. There are 2K tables in a casino, $t_1^1, \ldots, t_K^1, t_1^2, \ldots, t_K^2$ each of which is equipped with a single dice which may be biased, i.e. any categorical distribution $p(X_k|Z_k = t_k^i)$ on $1, \ldots, 6$. There are N players $p_1, \ldots, p_N$. Each player p_n visits K tables. When transitioning between tables, there is a $1/4$ probability that the player moves to the table in the same group and a $3/4$ probability that the player moves to the table in the other group. At table k player n throws the dice; its outcome X_k^n is observed with probability p. We observe the sum $S^n = \sum_{k=1}^K X_k^n$ of all the outcomes for the player. So, for N players we have $S^1, X^1, \ldots, S^N, X^N$, where X^n is the sequence of outcomes for player n with the unobserved outcomes censored.

Question 3: *Implement the Sum-HMM, i.e., write your own code for it*

See appendix.

Question 4: *Provide data generated using at least three different sets of categorical dice distributions that provide reasonable tests for the correctness of your program.*

In order to better demonstrate the correctness of my program, for the first two cases cases I used only two categorical distributions, one for group 1 and another for group 2.

In set 1, group 1 had a uniform distribution while group 2 had a biased die that rolled 6 with $\frac{1}{2}$ probability and all other values with an equal probability. There were 200 total tables with 10 players and a 50% chance that a roll would be observed. We see in figure 2 that the observed data follows the generative distributions. In addition we see that group one and two had a roughly equal proportion of the total rolls (as expected) and that a player switched groups roughly 75% of the time.

In set 2, group 1 had an equal chance of rolling a 1, 3, or 5 with zero probability for other rolls while group 2 had a uniform distribution. The histograms from figure 3 reflect this. Also included in figure 3 are the data for the hidden throws only in each group. We see that the hidden throws follow the same distribution as all throws.

In set 3, every table had its own randomly generated distribution. In figure 4 we see this leads to a roughly uniform distribution over all outcomes.

Question 5: *Motivate your test and why the result of it indicates correctness*

The three tests above demonstrate correctness of the program. The data from each test followed the expected distributions in all cases. Additionally, the program allows us to use unique categorical distributions for each die or to have a group 1 and group 2 distribution. The program was also able to handle multiple combinations of players and tables. It was also demonstrated that a player switched groups roughly 75% of the time, which follows from the assignment description. Additionally, because the transition probabilities are the same, it is expected that there should be a roughly equal number of rolls in each group, which is corroborated by the pie chart given in figure 2. Finally we note that the hidden throws have the same distribution as the overall throws. This is important because the probability that a throw is observed is always equal to p and should therefore not result in different distributions between observed and unobserved throws within a group.

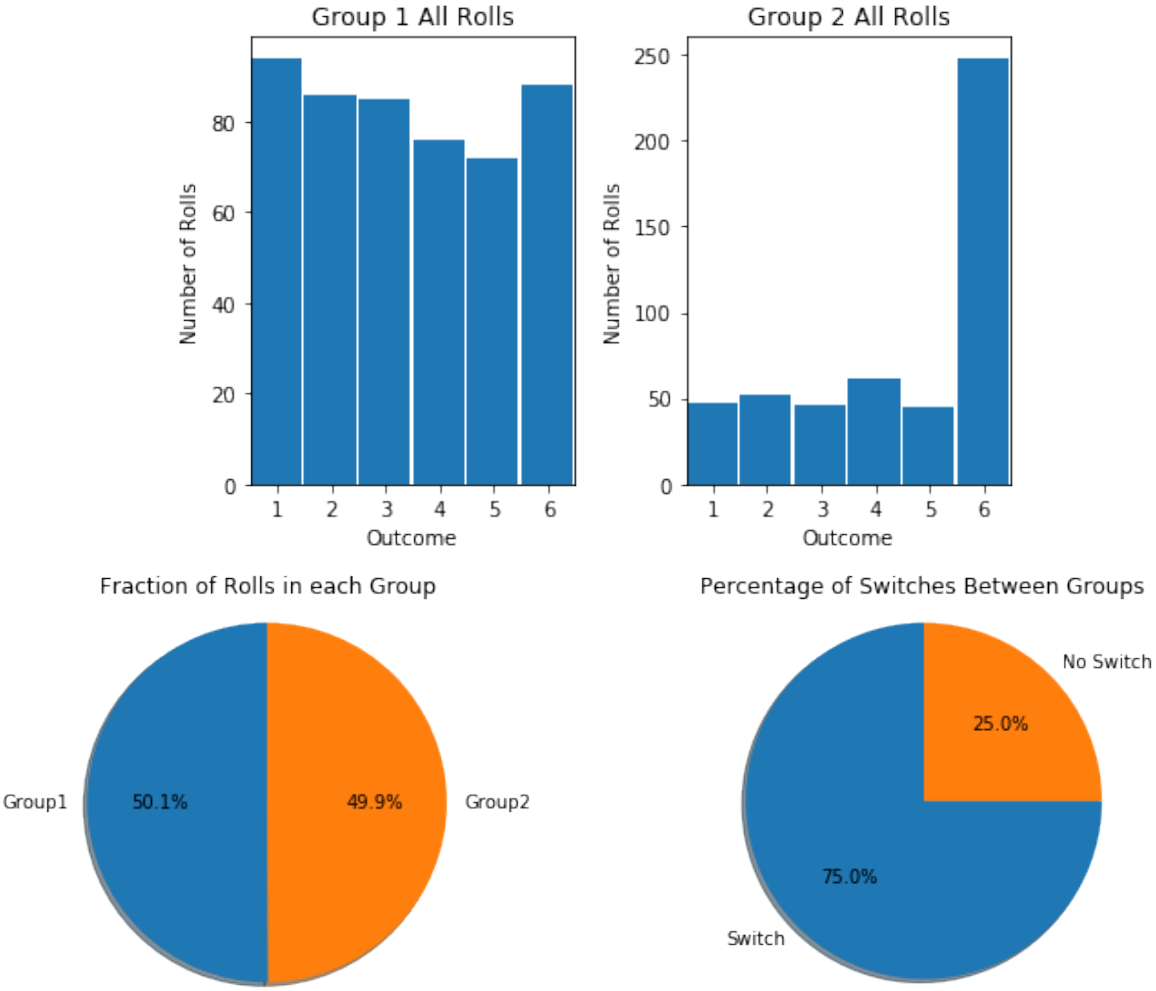

Figure 2: Set 1. Group 1 distribution: $[\frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}]$. Group 2 distribution: $[\frac{1}{10}, \frac{1}{10}, \frac{1}{10}, \frac{1}{10}, \frac{1}{10}, \frac{1}{2}]$. $K = 100, N = 10, p = 0.5$.

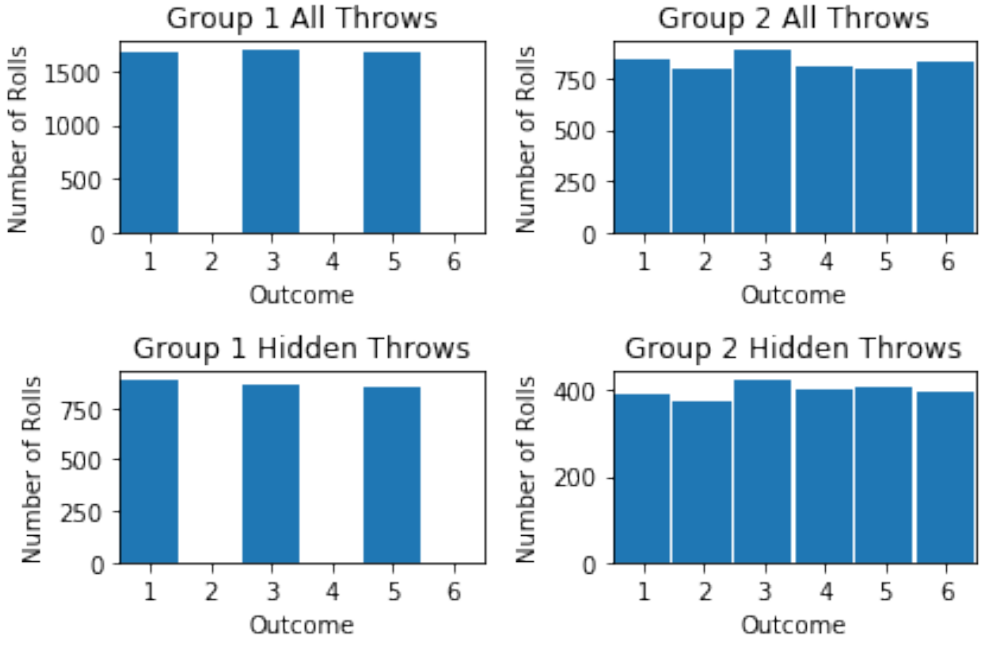

Figure 3: Set 2. Group 1 distribution: $[\frac{1}{3}, 0, \frac{1}{3}, 0, \frac{1}{3}, 0]$. Group 2 distribution: $[\frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}]$. $K = 1, N = 10000, p = 0.5$.

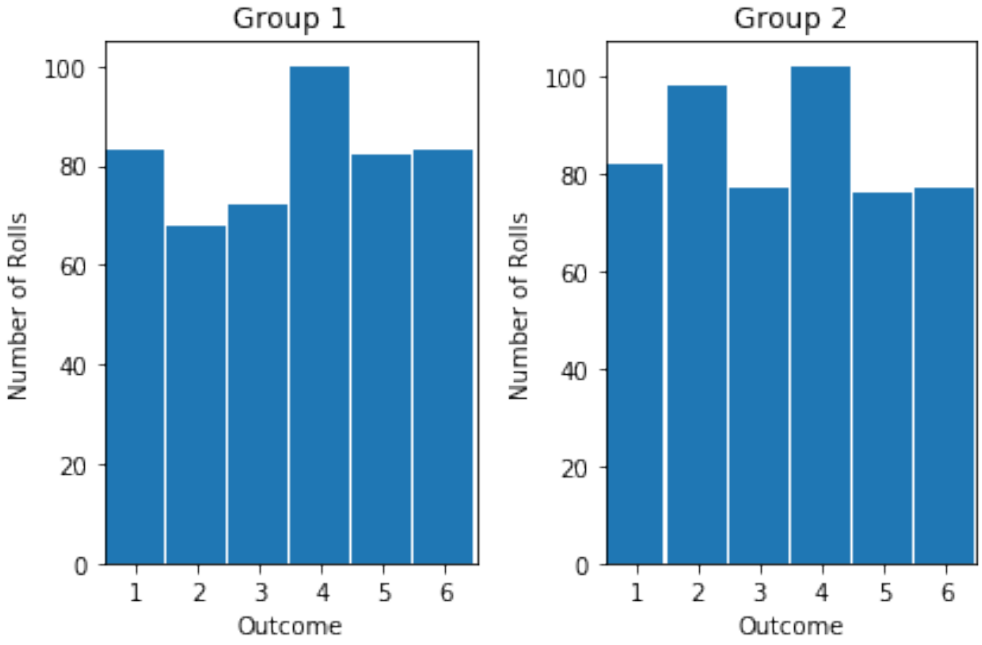

Figure 4: Set 3. Every table had its own distribution. $K = 1000, N = 1, p = 0.5$.

Question 6: *Give polynomial time dynamic programming algorithm for computing* $p(X_k^n = s, Z_k = t_k^i | s^n, x^n)$. *Hint: a dice outcome is an integer between 1 and 6, so a sum* s^n *is an integer between K and 6K and, moreover, if a partial sum is associated with a state* t_k^i*, it is an integer between k and 6k.*

We can begin by rewriting $p(X_k^n = s, Z_k = t_k^i | s^n, x^n)$ based on conditional probability rules,

$$p(X_k^n = s, Z_k = t_k^i | s^n, x^n) = \frac{p(X_k^n = s, Z_k = t_k^i, s^n, x^n)}{p(s^n, x^n)}. \tag{1}$$

Using the sum rule, the right hand side can be rewritten as,

$$\frac{\sum_{Z_K} p(Z_K, X_k^n = s, Z_k = t_k^i, s^n, x^n)}{\sum_{Z_K} p(Z_K, s^n, x^n)} \tag{2}$$

The denominator and numerator now have a similar form to the "alpha" variable typically defined as the joint probability of an observation sequence up to a time point and the hidden state at that time point for the forward pass in a "normal" HMM, $\alpha(Z_n) = p(x_{1:n}, Z_n)$. Using this as inspiration, we can define our own alpha variable for the "Sum-HMM" as follows,

$$\alpha(Z_k) = p(Z_k, s_k^n, x_{1:k}^n) \tag{3}$$

where Z_k is the hidden state at table k, s_k^n is the partial sum at state k for player n, and $x_{1:k}^n$ is the observation sequence up to state k for player n. In this case the observation sequences may include both observed and unobserved rolls. Have defined the alpha variable as in (3), we see that the denominator of (2) is the sum of the alpha values across both states at the final table, $\sum_{Z_K} \alpha(Z_K)$. The numerator is also similar in form to an alpha variable, with the exception that in this case when we perform the α-recursion, we explicitly set the observation equal to s and the hidden state to t_k^i such that we now have the joint probability. For simplicity, we can denote this as, $\alpha(Z_K, X_k^n = s, Z_k = t_k^i)$. We can then rewrite (2) as,

$$\frac{\sum_{Z_K} \alpha(Z_K, X_k^n = s, Z_k = t_k^i)}{\sum_{Z_K} \alpha(Z_K)} \tag{4}$$

Having defined (4) we recognize that if we perform two alpha passes, one in which set the desired observation value and hidden state at table k and one "normal" alpha pass, we can sum up the values at the final table to give us the desired probability. Now we must define a dynamic programming algorithm for the alpha variable.
The first step is rather trivial; we can simply state that if X_k^n is observed and not equal to s, then the probability is zero.

If X_k^n is equal to s or not observed we proceed with the alpha pass. We can begin with the normal alpha pass at the final table $\alpha(Z_K)$.

$$\alpha(Z_K) = \sum_{X_K^n} p(X_K|Z_K) \sum_{Z_{K-1}} p(Z_{K-1}, (s^n - X_K^n), x_{1:K-1}^n) p(Z_K|Z_{K-1}) \tag{5}$$

Basically, this equation states that in order to calculate the alpha variable at table K we first consider all possible die rolls at state K (if X_K^n is unobserved we consider all possible rolls 1-6, if observed we only consider that value). For each possible die roll we multiply the emission probability times the sum across previous states of $\alpha(Z_{K-1})$ times the transition probability, where $\alpha(Z_{K-1}) = p(Z_{k-1}, (s^n - X_K^n), x_{1:K-1}^n)$. $s^n - X_K^n$ represents the necessary previous partial sum, s_{K-1}^n, in order to reach the observed partial sum if we roll X_K^n. We can also be sure to only consider the rolls that give a value of s_{K-1}^n between $(K-1)$ and $6(K-1)$. We can now continue this recursion all the way back to table 1, where we must have a base case, $\alpha(Z_1)$.

$$\alpha(Z_1) = \sum_{X_1^n} p(Z_1) p(X_1^n|Z_1)$$

It is perhaps easier to visualize the progress of the algorithm in which we imagine a large three dimensional matrix of alpha values with dimensions (K x 6K x 2), where for each table k we have alpha values for all possible partial sums across both states. We begin with two alpha values, one for each final state at the observed final sum, and recurse backwards through the matrix in order to populate the matrix with alpha values based on the previous state. Once we reach the base case (table 1), we can calculate probabilities and fill the matrix with numerical values.

We should also note that when we consider the probability of an observation sequence x^n, we must also account for the probability of the sequence of unobserved versus observed rolls. For example if the probability of observing a roll is p and we have a three roll sequence in which all rolls are unobserved (e.g. the sequence is ? ? ?), we should multiply each possible observation sequence by $(1-p)^3$. However, since we will be performing the alpha pass twice this factor will be canceled out and need not be considered in the actual implementation of the algorithm.
We can now formalize this algorithm as follows:

Algorithm 1 Sum-HMM Dynamic Programming Smoothing

1: **if** X_k^n is observed and not equal to s **then**
2: $\quad p(X_k^n = s, Z_k = t_k^i | s^n, x^n) = 0$
3: **else**
4: $\quad$ **for** `k=K to 1` **do**
5: $\quad\quad$ **if** $k = 1$ **then**
6: $\quad\quad\quad \alpha(Z_1) = \sum_{X_1^n} p(Z_1) p(X_1^n|Z_1) \mathbb{I}(X_1^n = s_1^n)$
7: $\quad\quad$ **else**
8: $\quad\quad\quad \alpha(Z_k) = p(X_k|Z_k) \sum_{Z_{k-1}} p(Z_{k-1}, (s^n - X_k^n), x_{1:k-1}^n) p(Z_K|Z_{K-1}) \mathbb{I}(k-1 \leq s^n - X_k^n \leq 6(k-1))$

The initial probabilities for the first state Z_1 are each 0.5. Both alpha passes are the exact same, with the exception that for alpha pass for the numerator in (4) once we get to the desired table k we only consider $X_k^n = s$ and $Z_k = t_k^i$. Additionally, we must take into account that when X_k is unobserved then we must sum over all possible observations in line 8 as in equation 5.

Question 7: *Implement this DP algorithm, test it, in particular for varying values of p, and, finally motivate your tests and why the result of it indicates correctness.*

This algorithm was successfully implemented (see appendix). I focused on analyzing only one player, since it was already established that the implementation could handle multiple players. In this implementation, we assume that the observation, transition, and emission probabilities are all known. In order to check for correctness, three separate tests were performed. The first test was on a very simple observation sequence consisting of only 5 rolls generated by two fair die with a 0.5 probability of observation, as shown below.

Since the observed sum is 14, we know that the two unobserved rolls must sum to $14-5=9$. This leaves two possibilities for the combination of rolls, $3,6$ or $4,5$. The probability of either combination is equal. Additionally, the probability of the rolls being generated by either die are equal, and for a given combination, the rolls could have happened in either order. From these observations, we can know the joint probability of any roll and the hidden state given the observed sum and the observation sequence without having to run the algorithm. We know that for tables 1, 2, and 5 the probability of an observation not being 1, 2, and 2, respectively is 0. The probability that an observation is one those rolls and is in

$ObservedSum = 14$				
1	2	?	?	2

Table 1: Observations generated by two fair die with p=0.5

Expected and Calculated Probabilities		
$p(Z_k = t_k^i, X_k^n = s \mid s^n, x^n)$	Expected	Calculated
$p(Z_3 = t_3^1, X_3^1 = 3 \mid s^n, x^n)$	0.125	0.125
$p(Z_4 = t_4^2, X_4^2 = 6 \mid s^n, x^n)$	0.125	0.125
$p(Z_4 = t_4^2, X_4^1 = 2 \mid s^n, x^n)$	0	0
$p(Z_5 = t_5^2, X_5^2 = 3 \mid s^n, x^n)$	0	0
$p(Z_5 = t_5^2, X_5^1 = 2 \mid s^n, x^n)$	0.5	0.5
$p(Z_5 = t_5^2, X_5^2 = 2 \mid s^n, x^n)$	0.5	0.5

Table 2: Probabilities for Observation sequence in Table 1

either state is 0.5. At tables 3 and 4 the probability of rolling a 1 or 2 is 0, whereas for each table there should be an equal probability of rolling a 3 or above in either state. We can therefore say that $p(Z_k = t_k^i, X_k^n = s \mid s^n, x^n) = .125$ for $i = 1, 2, k = 3, 4, s = 3, 4, 5, 6$. Indeed, when we run the algorithm, we get these results.

Having validated the algorithm for a very simple test, we can increase the complexity slightly by analyzing a longer observation sequence with $p = 0.75$. Additionally, we will change the dice distributions such that the group 1 die is still fair but the group 2 die only rolls 6s. This gives us the observation sequence show in table 3. Since the observed sum is 48, we know the three unobserved rolls must sum to 16. This problem is slightly more complex, so the probabilities are not calculated explicitly by hand. However, we can make some observations; namely that the probability of any roll not equal to 6 while being in state 2 should be 0, and it is much more likely that we were in state 2 when rolling a 6 than in state 1.

As a sanity check, we can perform one more test that is slightly more complex. Having already established the ability of the algorithm to correctly calculate probabilities for relatively simple die distributions, we will now have more complex distributions for both dice, with group 1 being slightly more biased towards smaller rolls while group 2 is more biased towards larger rolls. The observation probability was also decreased to 0.3, such that we have a larger space of possibilities for the unobserved rolls. This demonstrates that the algorithm is not simply using a naive approach and explicitly enumerating all possibilities, but rather is using a dynamic-programming approach. The observation sequence is given in table 5.

Table 6 demonstrates that the algorithm is able to handle a larger space of rolls. Though these tests are certainly not exhaustive, they show that the algorithm is behaving as expected. In general, rolls that are impossible based on the observed sum or that conflict with an observed roll are assigned a probability zero. Rolls that are observed and could have only been produced by one of the die are assigned a probability of 1 for that state. A die that has a probability distributions more biased toward certain rolls will have higher probability for those rolls than the other die.

2.3 Simple VI

Here we consider the model defined by Equations (10.21)-(10.23) in *Pattern Recognition and Machine Learning*. We are concerned with the variational inference (VI) algorithm for this model covered during the lectures and in the book.

Question 8: *Implement the VI algorithm for the variational distribution in Equation (10.24) in Bishop.*

See appendix.

Question 9: *Describe the exact posterior.*

$ObservedSum = 48$									
6	6	1	?	6	6	1	?	?	6

Table 3: Group 1 distribution: $[\frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}]$. Group 2 distribution: $[0, 0, 0, 0, 0, 1]$. p=0.75

Calculated Probabilities	
$p(Z_k = t_k^i, X_k^n = s \mid s^n, x^n)$	Calculated
$p(Z_1 = t_1^1, X_1^1 = 6 \mid s^n, x^n)$	0.304
$p(Z_2 = t_2^2, X_2^1 = 6 \mid s^n, x^n)$	0.895
$p(Z_3 = t_3^1, X_3^1 = 1 \mid s^n, x^n)$	1
$p(Z_9 = t_9^1, X_9^1 = 6 \mid s^n, x^n)$	0.067
$p(Z_9 = t_9^2, X_9^1 = 6 \mid s^n, x^n)$	0.173
$p(Z_9 = t_9^1, X_9^1 = 4 \mid s^n, x^n)$	0.694

Table 4: Probabilities for Observation sequence in Table 3

$ObservedSum = 48$									
?	?	1	6	?	?	?	5	4	?

Table 5: Group 1 distribution: $[0.2, 0.2, 0.3, 0.1, 0.1, 0.1]$. Group 2 distribution: $[0.1, 0.1, 0.1, 0.3, 0.2, 0.2]$. p=0.3

The prior distribution over the mean μ and precision τ is given by a Gaussian-gamma distribution,

$$(\mu, \tau) \sim \mathcal{NG}(\mu_0, \lambda_0, \alpha_0, \beta_0)$$

The exact posterior distribution is therefore also given by a Gaussian-gamma distribution [2],

$$p(\mu, \tau | \mathbf{X}) \sim \mathcal{NG}\left(\frac{\lambda_0\mu_0 + n\bar{x}}{\lambda_0 + n}, \lambda_0 + n, \alpha_0 + \frac{n}{2}, \beta_0 + \frac{1}{2}\left(\sum_{i=1}^{n}(x_i - \bar{x})^2 + \frac{\lambda_0 n(\bar{x} - \mu_0)^2}{\lambda_0 + n}\right)\right)$$

Question 10: *Compare the variational distribution with the exact posterior. Run the inference for a couple of interesting cases and describe the difference.*

The true mean and variance were always 0 and 1, respectively, for the generated data. The plots are inspired by figure 10.4 from *Pattern Recognition and Machine Learning* with the true posterior distribution having green contours and the factors of the approximating distribution shown in blue until convergence, at which point the contours are red [1].

In case 1 only 10 data points were generated. The hyper-parameter values were set to very small positive numbers, leading to a broad and uninformative prior. Despite this, we see that the variational inference converges relatively quickly to the solution. However, the shape of the approximated posterior is not exactly the same as the exact posterior, with the former being more "egg-shaped" while the latter is more triangular. We also see that the posterior for both cases is fairly close to the true parameter values, showing that even with a relatively small number of data points the posterior can be centered near the true generating parameters, albeit with quite a bit of uncertainty.

In case 2 slightly more data points were generated. The primary difference here is that the prior was much stronger. This is especially noticeable in the first panel where compared to figure 5, the initial factorized approximation already has a fairly well-defined shape. Again, VI converged quickly to a solution. We see that the shapes of the posteriors are much more similar in this case and more compact; however, they are also much further away from the true parameter values than

Calculated Probabilities	
$p(Z_k = t_k^i, X_k^n = s \mid s^n, x^n)$	Calculated
$p(Z_3 = t_3^1, X_3^1 = 1 \mid s^n, x^n)$	0.743
$p(Z_{10} = t_{10}^1, X_{10}^1 = 2 \mid s^n, x^n)$	0.058
$p(Z_{10} = t_{10}^2, X_{10}^1 = 4 \mid s^n, x^n)$	0.140
$p(Z_6 = t_6^1, X_6^1 = 3 \mid s^n, x^n)$	0.120
$p(Z_6 = t_6^2, X_6^1 = 4 \mid s^n, x^n)$	0.161
$p(Z_8 = t_8^2, X_8^1 = 4 \mid s^n, x^n)$	0

Table 6: Probabilities for Observation sequence in Table 5

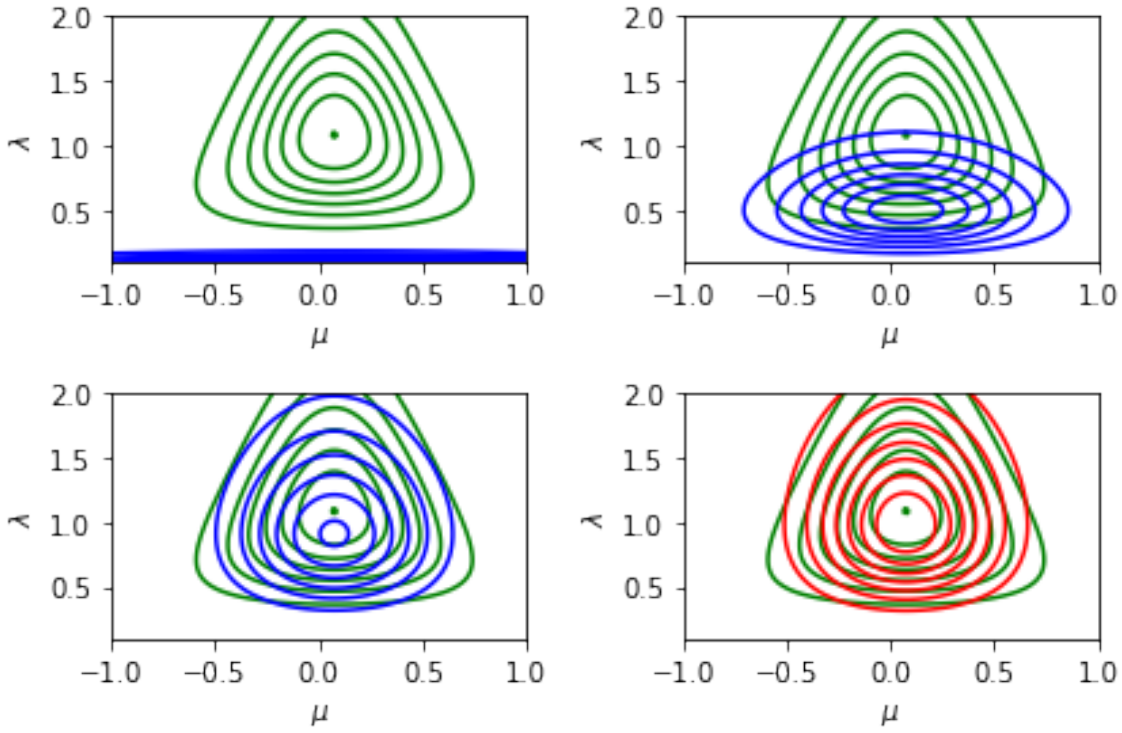

Figure 5: Case 1. 10 data points, uninformative prior.

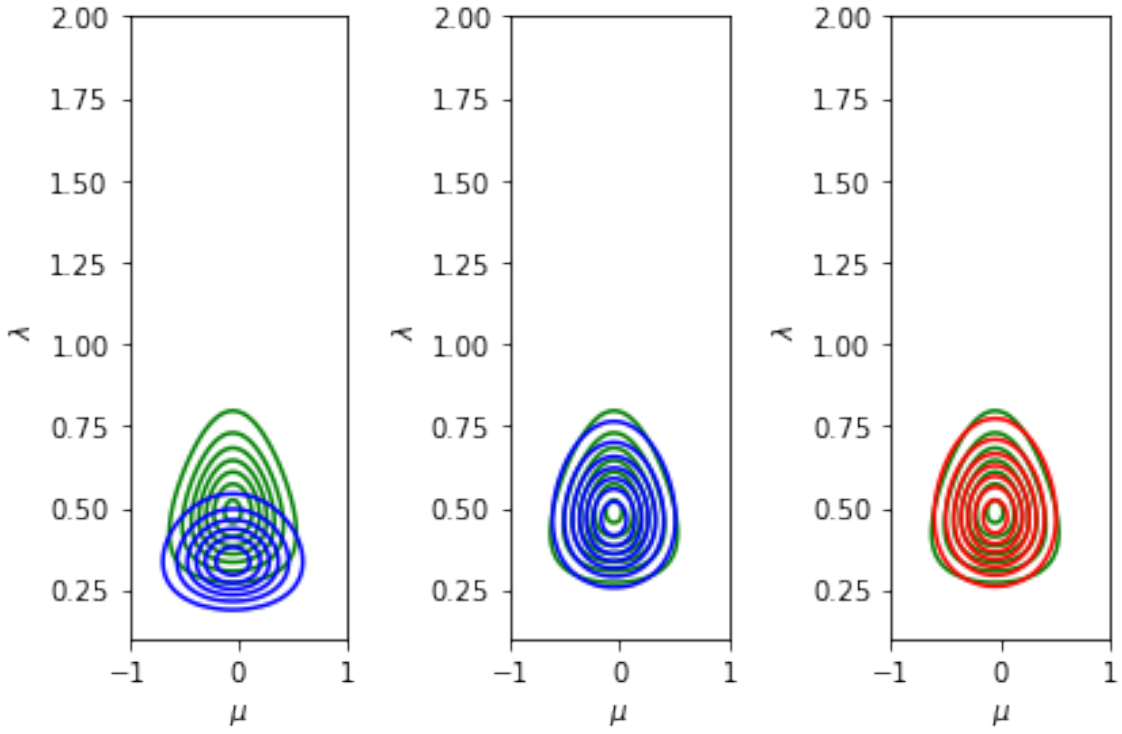

Figure 6: Case 2. 25 data points, strong prior.

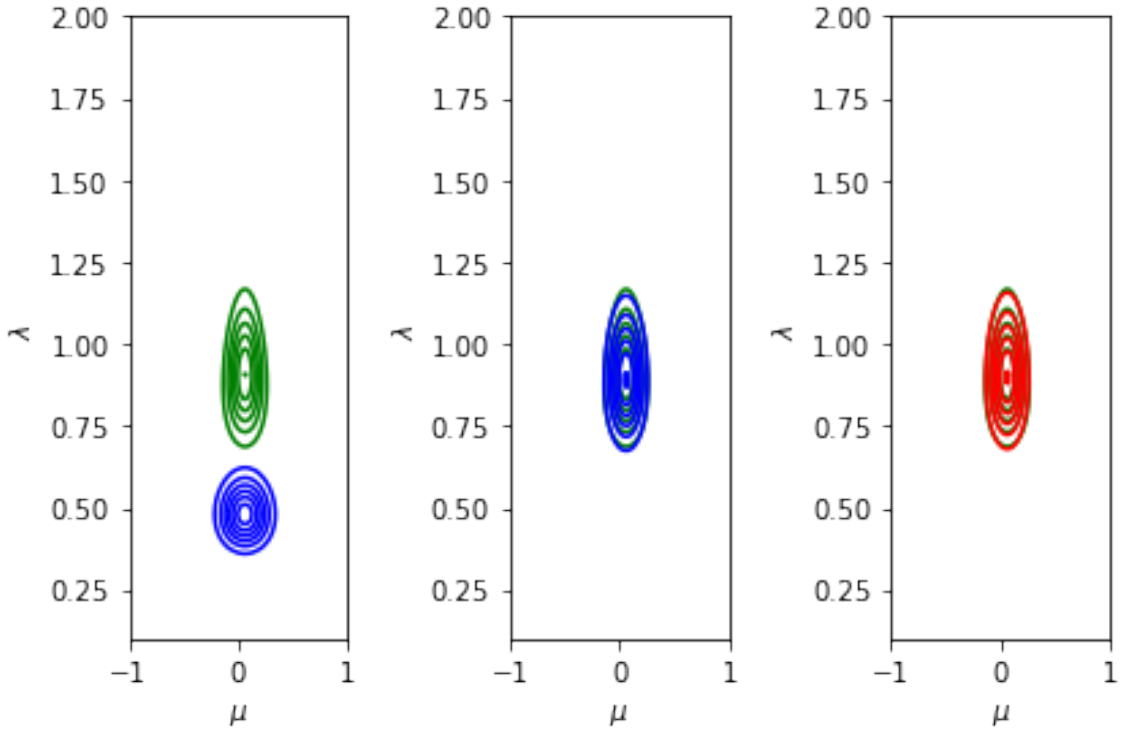

Figure 7: Case 3. 100 data points, strong prior.

in figure 5. This demonstrates that the still relatively small number of data points is not enough to overwhelm the strong prior.

Case 3 is similar to case 2 in that a very strong prior was used; however, in this case many more data points were generated (100 vs. 25). The result is that posteriors are much closer to the true parameter values than in figure 6. In contrast to case 2, here there are enough data points to overwhelm the strong prior. Again, variational inference quickly converged to a solution. Here there is almost no discernible difference between the exact posterior and the optimal factorized approximation. More data points result in a more compact factorized posterior that is also closer to the true posterior.

2.4 Sampling tables for the Sum-HMM

2.5 Expectation-Maximization (EM)

2.6 Variational Inference

Use Variational Inference in order to obtain a variational distribution,

$$q(\mu_1, \ldots, \mu_R, \xi_1, \ldots, \xi_C) = \prod_r q(\mu_r) \prod_c q(\xi_c)$$

that approximates $p(\mu_1, \ldots, \mu_R, \xi_1, \ldots, \xi_C | S)$.

Question 15: *Present the algorithm written down in a formal manner (using both text and mathematical notation but not pseudo code*

We have that each element of the matrix S is given by $S_{rc} = X_r + Y_c$, where $X_r \sim \mathcal{N}(\mu_r, \lambda_r^{-1})$ and $Y_c \sim \mathcal{N}(\xi_c, \tau_c^{-1})$. The variances λ_r^{-1} and τ_c^{-1} are known while the priors for the means are $\mathcal{N}(\mu, \lambda^{-1})$ and $\mathcal{N}(\xi, \tau^{-1})$, respectively. All hyper-parameters are known. We know that the sum of two normally distributed variables is itself normally distributed with the mean being the sum of the two means and the variance being the sum of the two variances, giving us $S_{rc} \sim \mathcal{N}(\mu_r + \xi_c, \lambda_r^{-1} + \tau_c^{-1})$.

If we have partitions Z_j, the optimal solution for the approximating distributions is given by,

$$\log q_j^*(Z_j|X) = \mathbb{E}_{i\neq j}[\log p(Z, X)] + C \tag{6}$$

where X represents the data and Z are all latent factors and parameters. So we need the joint probability distribution over the factors and data. In this case this is given by,

$$p(S, \mu_{1:R}, \xi_{1:C}) = p(S|\mu_{1:R}, \xi_{1:C})p(\mu_{1:R})p(\xi_{1:C}) \tag{7}$$

where

$$\begin{aligned} p(S|\mu_{1:R}, \xi_{1:C}) &= \prod_{c=1}^{C}\prod_{r=1}^{R}\mathcal{N}(\mu_r + \xi_c, \lambda_r^{-1} + \tau_c^{-1}) \\ p(\mu_{1:R}) &= \prod_{r=1}^{R}\mathcal{N}(\mu, \lambda^{-1}) \\ p(\xi_{1:C}) &= \prod_{c=1}^{C}\mathcal{N}(\xi, \tau^{-1}) \end{aligned}$$

Now that we have defined our priors and likelihood, for each of the $r \times c$ partitions we can write down the best approximating distributions. We want the expectation as functions of variables not in the current partition. Eventually we want an EM-type algorithm in which we initialize all the parameters and iterate through a series of steps updating each parameter. This algorithm is guaranteed to converge on a local maximum.

We can begin by considering the update equation for μ_1. We have,

$$\log q_{\mu_1}^*(\mu_1) = \mathbb{E}_{\mu\neq 1,\xi_{1:C}}[\log p(S|\mu_{1:R}, \xi_{1:C}) + \log p(\mu_{1:R}) + \log p(\xi_{1:C})] + C \tag{8}$$

We are only interested in the functional dependence on μ_1, we can therefore absorb terms not dependent on μ_1 into the normalization constant. It is immediately obvious that none of the ξ terms are dependent on μ_1 and therefore can be eliminated.

$$\begin{aligned} \log q_{\mu_1}^*(\mu_1) &= \mathbb{E}_{\mu\neq 1,\xi_{1:C}}\left[\log p(S|\mu_{1:R}, \xi_{1:C}) + \log p(\mu_{1:R})\right] + C \\ \log q_{\mu_1}^*(\mu_1) &= \mathbb{E}_{\mu\neq 1,\xi_{1:C}}\left[\log \prod^{C}\prod^{R}\mathcal{N}(\mu_r + \xi_c, \lambda_r^{-1} + \tau_c^{-1})\right] + \mathbb{E}_{\mu\neq 1,\xi_{1:C}}\left[\log \prod^{R}\mathcal{N}(\mu, \lambda^{-1})\right] + C \end{aligned}$$

Since the variances of both the row and column distributions are known, we can use a dummy variable $\kappa_{rc} = (\lambda_r^{-1} + \tau_c^{-1})^{-1}$ representing the precision for a particular cell of the matrix in order to clean up the math. Substituting in the probability distributions we have,

$$\log q_{\mu_1}^*(\mu_1) = \mathbb{E}_{\mu\neq 1,\xi_{1:C}}\left[\log \prod^{C}\prod^{R}\sqrt{\frac{\kappa_{rc}}{2\pi}}e^{\frac{-(S_{rc}-\mu_r-\xi_c)^2\kappa_{rc}}{2}}\right] + \mathbb{E}_{\mu\neq 1,\xi_{1:C}}\left[\log \prod^{R}\sqrt{\frac{\lambda}{2\pi}}e^{\frac{-(\mu_r-\mu)^2\lambda}{2}}\right] + C$$

Once we translate the log of products to the sum of logs we can again absorb terms not dependent on μ_1 in to the normalization constant. Thus getting rid of any constants and all other μ terms by ignoring the summation over all other r values. We also see that we will have a quadratic polynomial in μ_1, meaning that the distribution of q_{μ_1} is a Gaussian. It is important

to note that we have not assumed this form for the distribution and instead it merely "fell out" as we performed the math.

$$\log q^*_{\mu_1}(\mu_1) = \mathbb{E}_{\xi_{1:C}}\left[\sum^{C} \frac{-(S_{1c}-\mu_1-\xi_c)^2\kappa_{1c}}{2}\right] - \frac{(\mu_1-\mu)^2\lambda}{2} + C$$

$$\log q^*_{\mu_1}(\mu_1) = -\frac{1}{2}\left(\mathbb{E}_{\xi_{1:C}}\left[\sum^{c}(S_{1c}^2 - 2S_{1c}\mu_1 - 2S_{1c}\xi_c + 2\mu_1\xi_c + \xi_c^2 + \mu_1^2)\kappa_{1c}\right] + (\mu_1^2 - 2\mu_1\mu + \mu^2)\lambda\right)$$

$$\log q^*_{\mu_1}(\mu_1) = -\frac{1}{2}\left(\mathbb{E}_{\xi_{1:C}}\left[\sum^{c}(-2S_{1c}\mu_1 + 2\mu_1\xi_c + \mu_1^2)\kappa_{1c}\right] + (\mu_1^2 - 2\mu_1\mu)\lambda\right)$$

$$\log q^*_{\mu_1}(\mu_1) = -\frac{1}{2}\left(\mathbb{E}_{\xi_{1:C}}\sum^{C} 2\mu_1\xi_c\kappa_{1c} - \sum^{C} 2S_{1c}\mu_1\kappa_{1c} + \sum^{C}\mu_1^2\kappa_{1c} + \mu_1^2\lambda - 2\mu_1\mu\lambda\right)$$

$$\log q^*_{\mu_1}(\mu_1) = -\frac{1}{2}\left(\mu_1^2\left(\sum^{C}\kappa_{1c} + \lambda\right) - 2\mu_1\left[\sum^{C} S_{1c}\kappa_{1c} + \mu\lambda - \sum^{C}\kappa_{1c}\mathbb{E}_{\xi_c}(\xi_c)\right]\right)$$

$$\log q^*_{\mu_1}(\mu_1) = -\frac{1}{2}\left(\mu_1^2\left(\sum^{C}\kappa_{1c} + \lambda\right) - 2\mu_1\left[\sum^{C}(S_{1c} - \mathbb{E}_{\xi_c}(\xi_c))\kappa_{1c} + \mu\lambda\right]\right)$$

The expected value of ξ_c with respect to ξ_c is simply ξ, which is known since it is stated that all hyper-parameters are known. We also now clearly see the form of $q(\mu_1)$, which is distributed as,

$$q^*(\mu_1) \sim \mathcal{N}(\mu_1|\alpha_1, \beta_1^{-1})$$
$$\beta_1 = \sum^{C}\kappa_{1c} + \lambda$$
$$\alpha_1 = \frac{\sum^{C}(S_{1c}-\xi)\kappa_{1c} + \mu\lambda}{\beta_1}$$

From here we can generalize to all other μ_r distributions, simply noting that κ_{1c} would be replaced with κ_{rc} and S_{1c} would be replaced with S_{rc} for that row. We also note that all ξ_c would be similarly distributed,

$$q^*(\xi_c) \sim \mathcal{N}(\xi_c|\gamma_c, \delta_c^{-1})$$
$$\delta_c = \sum^{R}\kappa_{rc} + \tau$$
$$\gamma_c = \frac{\sum^{R}(S_{rc}-\mu)\kappa_{rc} + \xi\tau}{\delta_c}$$

Since all hyper-parameters and each variance for both the column and row distributions are known, this actually does not end up being an EM-type iterative algorithm and should reach an optimal solution after one update.

2.7 Variational Inference

Here we again consider a casino model in which there are $2K$ tables in a casino, $t_1^1, \ldots, t_K^1, t_1^2, \ldots, t_K^2$ and N players $p_1, \ldots, p_N$. Tables as well as players are equipped with a Gaussian; $t_k^i \sim \mathcal{N}(\mu_k^1, \lambda_k^{-1}$ where λ_k^{-1} is known and μ_k^i has a prior $\mathcal{N}(\mu, \lambda^{-1}$, and $p_n \sim \mathcal{N}(\xi_n, \tau_n^{-1})$ where τ_n^{-1} is known and ξ_n has a prior distribution $\mathcal{N}(\xi, \tau^{-1})$. All hyper-parameters are known.

The transition probabilities between tables are the same as in 2.2. At table k the player samples from both her own and the table's Gaussian leading to the observation of the sum S_k^n. For each player n we observe $S^n = S_1^n, \ldots, S_K^n$, and we also have the overall observation for all N players, $S^1, \ldots, S^N$.

Here we will use Variational Inference to obtain a variational distribution,

$$q(\mu_{1:K}^1, \mu_{1:K}^2, \xi_{1:N}) = \prod_k q(\mu_k^1)\prod_k q(\mu_k^2)\prod_n q(\xi_n)$$

that approximates $p(\mu_{1:K}^1, \mu_{1:K}^2, \xi_{1:N}|S^1, \ldots, S^N)$. We will first target,

$$q(\mu_{1:K}^1, \mu_{1:K}^2, \xi_{1:N}, Z) = q(Z)\prod_k q(\mu_k^1)\prod_k q(\mu_k^2)\prod_n q(\xi_n),$$

where $Z = Z_1, \ldots, Z_K$ is the table sequence. Then the marginalization will be performed in order to approximate the full posterior.

Question 16: *Present the algorithm written down in a formal manner (using both text and mathematical notation but not pseudo code*

We begin similarly to question 15. The joint probability distribution can be written as,

$$p(\mu^1_{1:K}, \mu^2_{1:K}, \xi_{1:N}, Z, S^{1:N}) = p(S^{1:N}|\mu^1_{1:K}, \mu^2_{1:K}, \xi_{1:N}, Z)p(\xi_{1:N})p(\mu^1_{1:K}|Z)p(\mu^2_{1:K}|Z)p(Z)$$

where

$$\begin{aligned}
p(\xi_{1:N}) &= \prod_n \mathcal{N}(\xi, \tau^{-1}) \\
p(\mu^1_{1:K}|Z) &= \prod_k \mathcal{N}(\mu, \lambda)^{\mathbb{I}(Z_k=1)} \\
p(\mu^2_{1:K}|Z) &= \prod_k \mathcal{N}(\mu, \lambda)^{\mathbb{I}(Z_k=2)} \\
p(Z) &= \prod_{k=1}^{K-1} p(Z_{k+1} = z_{k+1}|Z_k = z_k)p(Z_1 = z_1) \\
p(S^{1:N}|\mu^1_{1:K}, \mu^2_{1:K}, \xi_{1:N}, Z) &= \prod_k \prod_n \mathcal{N}(\mu^1_k + \xi_n, \lambda_k^{-1} + \tau_n^{-1})^{\mathbb{I}(Z_k=1)} \mathcal{N}(\mu^2_k + \xi_n, \lambda_k^{-1} + \tau_n^{-1})^{\mathbb{I}(Z_k=2)}
\end{aligned}$$

We define a new variable $\kappa_{kn} = (\lambda_k^{-1} + \tau_n^{-1})^{-1}$. We can begin by finding the best approximating distribution for μ^1_1. We have,

$$\log q^*(\mu^1_1) = \mathbb{E}_{\mu^1 \neq 1, \mu^2_{1:k}, \xi_{1:N}, Z}[\log p(S^{1:N}|\mu^1_{1:K}, \mu^2_{1:K}, \xi_{1:N}, Z) + \log p(\mu^1_{1:K}|Z) + \log p(\mu^2_{1:K}|Z) + \log p(Z) + logp(\xi_{1:K})]$$

As in 2.6, we can ignore terms that are constant with respect to μ^1_1, giving us,

$$\begin{aligned}
\log q^*(\mu^1_1) &= \mathbb{E}_{\xi_{1:N}, Z}\left[\sum_n \mathbb{I}(Z_1 = 1)\frac{-(S^n_1 - \mu^1_1 - \xi_n)^2\kappa_{1n}}{2}\right] - \frac{(\mu^1_1 - \mu)^2\lambda}{2} + C \\
\log q^*(\mu^1_1) &= -\frac{1}{2}\left(\mathbb{E}_{\xi_{1:N}, Z}\left[\sum_n \mathbb{I}(Z_1 = 1)(-2S^n_1\mu^1_1 + 2\mu^1_1\xi_n + (\mu^1_1)^2)\kappa_{1n}\right] + (\mu^1_1)^2\lambda - 2\mu^1_1\mu\lambda\right) + C \\
\log q^*(\mu^1_1) &= -\frac{1}{2}\left(p(Z_1 = 1)\left(\sum_n 2\mu^1_1\mathbb{E}_n(\xi_n)\kappa_{1n} - \sum_n 2S^n_1\mu^1_1\kappa_{1n} + \sum_n \mu^2_1\kappa_{1n}\right) + (\mu^1_1)^2\lambda - 2\mu^1_1\mu\lambda\right) + C \\
\log q^*(\mu^1_1) &= -\frac{1}{2}\left((\mu^1_1)^2\left[p(Z_1 = 1)\sum_n \kappa_{1n} + \lambda\right] - 2\mu^1_1\left[p(Z_1 = 1)\sum_n (S^n_1 - \mathbb{E}_n(\xi_n))\kappa_{1n} + \mu\lambda\right]\right) + C
\end{aligned}$$

We see that this expression is quadratic in μ^1_1 and we therefore expect the approximating distribution to be Gaussian. Generalizing to all μ^1_k we have,

$$\begin{aligned}
q^*(\mu^1_k) &\sim \mathcal{N}(\mu^1_k|\alpha_k, \beta_k^{-1}) \\
\beta_k &= p(Z_k = 1)\left(\sum_n \kappa_{kn} + \lambda\right) \\
\alpha_k &= \frac{p(Z_k = 1)\left(\sum_n (S^n_k - \xi)\kappa_{kn} + \mu\lambda\right)}{\beta_k}
\end{aligned}$$

By examining the form of this distribution, we note that $q^*(\mu^2_k)$ should be the same with the exception that $p(Z_k = 1)$ is replaced with $p(Z_k = 2)$. We can perform a similar derivation with respect to finding the best approximating distribution for ξ_1,

$$q^*(\xi_1) = \mathbb{E}_{\mu^1_{1:K},\mu^2_{1:K},Z}[\log p(S^{1:N}|\mu^1_{1:K},\mu^2_{1:K},\xi_{1:N},Z) + \log p(\xi_{1:N})] + C$$

$$q^*(\xi_1) = \mathbb{E}_{\mu^1_{1:K},\mu^2_{1:K},Z}\left[\sum_k \frac{\mathbb{I}(Z_k=1)^-(S^1_k-\mu^1_k-\xi_1)^2\kappa_{k1}}{2} + \sum_k \frac{\mathbb{I}(Z_k=2)^-(S^1_k-\mu^2_k-\xi_1)^2\kappa_{k1}}{2}\right] - \frac{(\xi_1-\xi)^2\tau}{2} + C$$

$$q^*(\xi_1) = -\frac{1}{2}\left(\mathbb{E}\left[\sum_k \mathbb{I}(Z_k=1)(-2S^1_k\xi_1 + 2\mu^1_k\xi_1 + \xi_1^2)\kappa_{k1} + \sum_k \mathbb{I}(Z_k=2)(-2S^1_k\xi_1 + 2\mu^2_k\xi_1 + \xi_1^2)\kappa_{k1}\right] + \xi_1^2\tau - 2\xi_1\xi\tau\right) + C$$

$$q^*(\xi_1) = -\frac{1}{2}\left(\mathbb{E}\left[\xi_1^2\left(\sum_k \mathbb{I}(Z_k=1)\kappa_{k1} + \sum_k \mathbb{I}(Z_k=2)\kappa_{k1} + \tau\right)\right.\right.$$
$$\left.\left. - 2\xi_1\left(\sum_k \mathbb{I}(Z_k=1)(S^1_k-\mu^1_k)\kappa_{k1} + \sum_k \mathbb{I}(Z_k=2)(S^1_k-\mu^2_k)\kappa_{k1} + \xi\tau\right)\right]\right) + C$$

$$q^*(\xi_1) = -\frac{1}{2}\left(\xi_1^2\left(\sum_k \kappa_{k1} + \tau\right) - 2\xi_1\left(\sum_k (S^1_k-\mu)\kappa_{k1} + \xi\tau\right)\right) + C$$

Here we have used the fact that in any given state the sum of the probabilities that $Z_k = 1$ and $Z_k = 2$ is equal to 1. Again we notice the form of the approximating distribution to be Gaussian. Generalizing to all ξ_n we have,

$$q^*(\xi_n) \sim \mathcal{N}(\xi_n|\gamma_n,\delta_n^{-1})$$
$$\delta_n = \sum_k \kappa_{kn} + \tau$$
$$\gamma_n = \frac{\sum_k (S^n_k-\mu)\kappa_{kn} + \xi\tau}{\delta_n}$$

Finally, we can consider the approximating distribution with respect to Z.

$$q^*(Z) = \mathbb{E}_{\mu^{1:2}_{1:K},\xi_{1:N}}[\log p(S^{1:N}|\mu^{1:2}_{1:K},\xi_{1:N},Z) + \log p(\mu^1_{1:K}|Z) + \log p(\mu^2_{1:K}|Z) + \log p(Z)] + C$$

$$q^*(Z) = \mathbb{E}\left[\sum_k\sum_n \mathbb{I}(Z_k=1)\frac{-(S^n_k-\mu^1_k-\xi_n)^2\kappa_{kn}}{2} + \sum_k\sum_n \mathbb{I}(Z_k=2)\frac{-(S^n_k-\mu^2_k-\xi_n)^2\kappa_{kn}}{2}\right.$$
$$\left. + \sum_k \mathbb{I}(Z_k=1)\frac{-(\mu^1_k-\mu)^2\lambda}{2} + \sum_k \mathbb{I}(Z_k=2)\frac{-(\mu^2_k-\mu)^2\lambda}{2} + \sum_{k=1}^{k-1}\log\left(p(Z_1=z_1)p(Z_{K+1}=z_{k+1}|Z_k=z_k)\right)\right] + C$$

Now, once we take the expectations with respect to all other variables, we see that all the terms except for those from $p(Z)$ will cancel out since the expectation of $\mu^{1:2}_k$ is μ and the expectation with respect to μ and ξ of S^n_k is simply $\mu+\xi$. Therefore, we see that the best approximating distribution for Z is merely the true distribution $p(Z)$. This also makes intuitive sense since we would not expect the distribution for Z to depend on the other variables. We can therefore easily marginalize over the table sequence since this will leave us with one and we are left a variational distribution $\prod_k q(\mu^1_k)\prod_k q(\mu^2_k)\prod_n q(\xi_n)$ to approximate $p(\mu^{1:2}_{1:K},\xi_{1:N}|S^{1:N})$. Similar to 2.6, we are not exactly left with an iterative procedure since all variances and hyper-parameters are known. Plugging in appropriate values and performing one update should leave us with an optimal solution.

A Import useful packages

```
import pylab as pb
import numpy as np
import random
from matplotlib import pyplot as plt
from numpy.random import multinomial
from scipy.stats import bernoulli
from math import pi
import itertools as it
import scipy.stats as stats
from scipy.special import gamma
```

B Sum HMM

B.1 Problem 3 — Sum HMM Implementation

```
#K = Length of sequence
K = 10
#N = number of players
N = 1
#p = Probability observation is observed
p = 0.3

group1Probs = [0.2, 0.2, 0.3, 0.1, 0.1, 0.1]
group2Probs = [0.1, 0.1, 0.1, 0.3, 0.2, 0.2]

#Dice through
def roll(groupNum=0):
    diceRoll = 0
    #Diceroll if player is at first group of tables

    #Random Distribution (default)
    if (groupNum == 0):
        dist = np.random.randn(6)**2
        dist = dist/dist.sum()
        x = np.random.multinomial(1, dist, size=1)
        diceRoll = np.flatnonzero(x) + 1


    if (groupNum == 1):
        x = np.random.multinomial(1, group1Probs, size=1)
        diceRoll = np.flatnonzero(x) + 1
    #Diceroll if player is at second group of tables
    elif (groupNum == 2):
        x = np.random.multinomial(1, group2Probs, size=1)
        diceRoll = np.flatnonzero(x) + 1

    return diceRoll

class Player(object):
    #Initialize player with empty vector of hiddenthrows and array of
    #all throws (hidden throws == 0)
    def __init__(self, tables):
        self.hiddenThrows = []
        self.Throws = np.zeros(tables)
        self.allThrows = np.zeros(tables)
        self.sum = 0
        #Randomly starts in either group of tables
        self.groupNum = bernoulli.rvs(0.5) + 1

        #Hidden throws for group 1 and 2
        self.hidden1 = []
        self.hidden2 = []

        #True state sequence
        self.sequence = np.zeros(tables)

    #Dice throw. Throw is observed according to probability p
    def Throw(self, throwNum, random=0):
        observed = bernoulli.rvs(p)
```

```
        #Roll dice according to which group of tables player is at
        if (random):
            throw = roll()
        else:
            throw = roll(self.groupNum)
        self.sequence[throwNum] = self.groupNum

        #Add roll to the player sum and to obersevation sequence
        self.sum += throw
        self.allThrows[throwNum] = throw

        if (observed):
            self.Throws[throwNum] = throw
        else:
            self.hiddenThrows.append(throw)
            if (self.groupNum == 1):
                self.hidden1.append(throw)
            elif (self.groupNum == 2):
                self.hidden2.append(throw)

    #Return the sum of all throws
    def SumThrows(self):
        return self.sum


players = []
group1 = np.zeros(0)
group2 = np.zeros(0)
hidden1 = np.zeros(0)
hidden2 = np.zeros(0)
#Loop through each player
for i in range(N):
    player = Player(K)
    players.append(player)
    #Loop through each table
    switches = 0
    for j in range(K):
        player.Throw(j)
        #Change the table group. 25% chance of staying in same group,
        #75% chance of switching group
        switch = bernoulli.rvs(.75)
        if (switch):
            player.groupNum = 3 - player.groupNum
            switches += 1


    print ("sum", player.sum)
    print ("Observation Sequence", player.Throws)
    print ("All throws", player.allThrows)
    print ("Sequence", player.sequence)
    group1 = np.append(group1, player.allThrows[np.where(player.sequence == 1)])
    group2 = np.append(group2, player.allThrows[np.where(player.sequence == 2)])
    hidden1 = np.append(hidden1, player.hidden1)
    hidden2 = np.append(hidden2, player.hidden2)

noswitch = (N*K - switches)
```

B.2 Problem 4 — Sum HMM Testing and Graphs

```
plt.subplot(1,2,1)
n, bins, patches = plt.hist(group1, bins=np.arange(1,8) - 0.5, width=0.95)
plt.xlim(1-0.5, 6+0.5)
plt.xticks(range(1,7))
plt.xlabel("Outcome")
plt.ylabel("Number of Rolls")
plt.title("Group 1 All Rolls")
plt.tight_layout()

plt.subplot(1,2,2)
n, bins, patches = plt.hist(group2, bins=np.arange(1,8) - 0.5, width=0.95)
plt.xlim(1-0.5, 6+0.5)
plt.xticks(range(1,7))
plt.xlabel("Outcome")
plt.ylabel("Number of Rolls")
plt.title("Group 2 All Rolls")
plt.tight_layout()

print(len(group1))
print(len(group2))

labels = 'Group1','Group2'
sizes = [len(group1), len(group2)]
fig1, ax1 = plt.subplots()
ax1.pie(sizes, labels=labels, autopct='%1.1f%%',
        shadow=True, startangle=90)
ax1.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.
plt.title('Fraction of Rolls in each Group')
plt.show()


labels = 'Switch', 'No Switch'
sizes = [switches, noswitch]
fig1, ax1 = plt.subplots()
ax1.pie(sizes, labels=labels, autopct='%1.1f%%',
        shadow=True, startangle=90)
ax1.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.
plt.title('Percentage of Switches Between Groups')
plt.show()
```

B.3 Problem 7 — DP Algorithm for Computing $p(Z_k = t_k^i, X_k^n = s|s^n, x^n)$

def alphaVar(table, group, partialSum, obSeq):

```
    #Define transition probabilities based on current state
    if (group == 1):
        transition = [.25, .75]
        groupProbs = group1Probs
    else:
        transition = [.75, .25]
        groupProbs = group2Probs

    #Base Case
    if (table == 0):
        #If unobserved return the probability of the partial sum at that state
        if (obSeq[table] == 0):
            return 0.5*groupProbs[int(partialSum)-1]
        else:
```

```
            #Ensure that observed roll at first state is equal to the partial sum
            if (obSeq[table]==partialSum):
                return 0.5*groupProbs[int(obSeq[table]-1)]
            #If observed roll is not the same as first partial sum, probability is zero
            else:
                return 0
    else:
        #Roll at table is unobserved
        if (obSeq[table] == 0):
            alpha = 0
            #Loop through all possible rolls
            for i in range(6):
                #Calculate previous sum based on current roll and partial sum
                prevSum = partialSum - (i+1)
                #Check to make sure the previous partial sum is possible
                if ((prevSum < (table)) or (prevSum > (6*table))):
                    continue
                #Recurse
                else:
                    #Emission probability for that roll
                    emissionProb = groupProbs[i]
                    alpha += emissionProb*alphaVar(table-1, 1, prevSum, obSeq)*transition[0]
                    alpha += emissionProb*alphaVar(table-1, 2, prevSum, obSeq)*transition[1]
            return alpha
        #Roll at table is observed
        else:
            alpha = 0
            #Extract observed roll and calculate previous sum
            roll = int(obSeq[table])
            prevSum = partialSum - (roll)
            #Make sure previous sum is possible, if not we can return 0 and prune that path
            if ((prevSum < (table)) or (prevSum > (6*table))):
                return alpha
            #Recurse
            emissionProb = groupProbs[roll - 1]
            alpha += emissionProb*alphaVar(table-1, 1, prevSum, obSeq)*transition[0]
            alpha += emissionProb*alphaVar(table-1, 2, prevSum, obSeq)*transition[1]
            return alpha


def alphaPass(table, group, partialSum, obSeq, obs, state, k):
    if (group == 1):
        transition = [.25, .75]
        groupProbs = group1Probs
    else:
        transition = [.75, .25]
        groupProbs = group2Probs

    #Make sure to only consider when Z_k=t_k^i
    if (table == k):
        if ((state == 1 and group == 2) or (state == 2 and group == 1)):
            return 0

    if (table == 0):
        if (obSeq[table] == 0):
            if (table==k and partialSum != obs):
                return 0
            else:
```

```
                return 0.5*groupProbs[int(partialSum)-1]
        else:
            if (obSeq[table]==partialSum):
                return 0.5*groupProbs[int(obSeq[table]-1)]
            else:
                return 0
    else:
        if (table == k):
            prevSum = partialSum - obs
            alpha = 0
            if ((prevSum < (table)) or (prevSum > (6*table))):
                return alpha
            if (state == 1):
                emissionProbs = group1Probs[obs-1]
            else:
                emissionProbs = group2Probs[obs-1]
            alpha += emissionProbs*alphaPass(table-1, 1, prevSum, obSeq, obs, state, k)*...
            transition[0]
            alpha += emissionProbs*alphaPass(table-1, 2, prevSum, obSeq, obs, state, k)*...
            transition[1]
            return alpha
        else:
            if (obSeq[table] == 0):
                alpha = 0
                for i in range(6):
                    prevSum = partialSum - (i+1)
                    #Check to make sure the previous partial sum is possible
                    if ((prevSum < table) or (prevSum > (6*table))):
                        continue
                    else:
                        emissionProb = groupProbs[i]
                        alpha += emissionProb*alphaPass(...
                                table-1, 1, prevSum, obSeq, obs, state, k)*transition[0]
                        alpha += emissionProb*alphaPass(...
                                table-1, 2, prevSum, obSeq, obs, state, k)*transition[1]
                return alpha

            else:
                alpha = 0
                roll = int(obSeq[table])
                prevSum = partialSum - (roll)
                if ((prevSum < table) or (prevSum > (6*table))):
                    return alpha
                emissionProb = groupProbs[roll - 1]
                alpha += emissionProb*alphaPass(table-1, 1, prevSum, obSeq, obs, state, k)*
                        transition[0]
                alpha += emissionProb*alphaPass(table-1, 2, prevSum, obSeq, obs, state, k)*
                        transition[1]
                return alpha




def calcProb(player, state, obs, k):
    if (player.Throws[k] != 0 and player.Throws[k] != obs):
        return 0
    else:
        num = 0
```

```
        denom = 0
        #print(alphaVar(len(player.Throws)-1, 1, player.sum, player.Throws))
        #print(alphaVar(len(player.Throws)-1, 2, player.sum, player.Throws))
        denom += alphaVar(len(player.Throws)-1, 1, player.sum, player.Throws)
        denom += alphaVar(len(player.Throws)-1, 2, player.sum, player.Throws)
        #print(alphaPass(len(player.Throws)-1, 1, player.sum, player.Throws, obs, state, k))
        #print(alphaPass(len(player.Throws)-1, 2, player.sum, player.Throws, obs, state, k))
        num += alphaPass(len(player.Throws)-1, 1, player.sum, player.Throws, obs, state, k)
        num += alphaPass(len(player.Throws)-1, 2, player.sum, player.Throws, obs, state, k)
        return num/denom
```

C Variational Inference

```
#Parameters of normal distribution
u0 = 0
tau = 1

#Generate Data
N = 10
X = np.random.normal(u0, 1/tau, N)



xbar = np.mean(X)
sumSq = np.sum((X-xbar)**2)
sumx = np.sum(X)
sumxsq = np.sum(X**2)



#Initial guesses (small positive values to give broad prior distributions indicating ignorance
a_n = .1
b_n = .1
mu_n = 0.1
lambda_n = .1

#Exact posterior parameter vals
aN = a_n + N/2
bN = b_n + 1/2*sumSq + (lambda_n*N*(xbar - mu_n)**2)/(2*(lambda_n + N))
muN = (lambda_n*mu_n + N*xbar)/(lambda_n + N)
lambdaN = lambda_n + N



muVals = np.linspace(-1,1,100)
lamVals = np.linspace(0.1,2,100)



def plot_pdf(pdf, X, Y, color):
    plt.contour(X, Y, pdf, colors=color)
    plt.xlabel("$\mu$")
    plt.ylabel("$\lambda$")

def gaussian_gamma(mu, lam, a, b, u, l):
    pdf_gamma = stats.gamma.pdf(l, a=a, scale=1/b)
    pdf_norm = stats.norm.pdf(u, loc=mu, scale=((l*lam)**(-0.5)))
```

```
    return pdf_gamma*pdf_norm

def gauss_gam_pdf(mu, lam, a, b, X, Y):
    return [[gaussian_gamma(mu, lam, a, b, u, l) for u in X] for l in Y]

def gaussian_gammaVI(mu, lam, a, b, u, l):
    pdf_gamma = stats.gamma.pdf(l, a=a, scale=1/b)
    pdf_norm = stats.norm.pdf(u, loc=mu, scale=((lam)**(-0.5)))
    return pdf_gamma*pdf_norm

def gauss_gam_pdfVI(mu, lam, a, b, X, Y):
    return [[gaussian_gammaVI(mu, lam, a, b, u, l) for u in X] for l in Y]

posterior_pdf = gauss_gam_pdf(muN, lambdaN, aN, bN, muVals, lamVals)

lambda_nt = lambda_n
mu_nt = mu_n
a_nt = a_n
b_nt = b_n

it = 0
maxIt = 10
while (it < maxIt):
    oldLq = .5*np.log(1/lambda_nt) + np.log(gamma(a_nt)) - a_nt*np.log(b_nt)

    VI_pdf = gauss_gam_pdfVI(mu_nt, lambda_nt, a_nt, b_nt, muVals, lamVals)

    mu_nt = (lambda_n*mu_n + N*xbar) / (lambda_n + N)
    a_nt = a_n + (N + 1)/2
    b_nt = b_n + 0.5*((lambda_n + N)*((1/lambda_nt) + mu_nt**2) -
            2*mu_nt*(lambda_n*mu_n + sumx) + sumxsq + lambda_n*mu_n**2)
    lambda_nt = (lambda_n + N)*(a_nt/b_nt)

    newLq = .5*np.log(1/lambda_nt) + np.log(gamma(a_nt)) - a_nt*np.log(b_nt)
    if (abs(newLq - oldLq) < .1):
        plt.subplot(2,2,it)
        plot_pdf(posterior_pdf, muVals, lamVals, "green")
        plot_pdf(VI_pdf, muVals, lamVals, "red")
        plt.tight_layout()
        break
    if (it > 0):
        plt.subplot(2,2,it)
        plot_pdf(posterior_pdf, muVals, lamVals, "green")
        plot_pdf(VI_pdf, muVals, lamVals, "blue")
        plt.tight_layout()

        #plt.show()

    it += 1
```

References

[1] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[2] R. Dearden, N. Friedman, and S. Russel. Bayesian q-learning. *AAAI Proceedings*, 1998.

[3] K. P. Murphy. *Machine learning: a probabilistic perspective*. Cambridge, MA, 2012.