Project 2

Noah Hall, Noah Schrick, Jordan White

October 12th, 2021

1 Part A: Hidden Markov Model (HMM)

1.1 Problem Overview

Hidden Markov Models are temporal probabilistic models which describe a state with a single discrete value. HMMs provide several conditional probability tables. This includes a table which describes the initial value probability for the first state, and a table which describes the value estimation for each state given its prior states, and a table which describes the probability of observing each possible value of each evidence variable given the value of the current state.

1.2 Domains

1.2.1 Sleeping in class

For the 'Sleeping in Class' domain, each state has two possible values, "True" and "False" which describe whether the student got enough sleep on night 't'. The prior probability matrix for the first state provides the probability that a student gets enough sleep at time slice 0. The transition matrix provides the probability of a given student getting enough sleep on night 't' given the amount of sleep they got on night 't-1'. The observation variables provided to us are whether a student has red eyes, and whether a student is sleeping in class. The probability of a student having red eyes or sleeping in class is given with respect to the possible prior state values.

1.2.2 Weather

In the 'Weather' domain, each state has two possible values, "Cold" and "Hot". The prior probability matrix describes the probability of the first day in the sequence being hot or cold. The transition matrix provides the probability of day 't' being hot or cold given the value of day 't-1'. The observation values are the number of ice cream scoops ordered, and the possible values are 1, 2, and 3. The probability of each of the possible scoop values is given with respect to the value of the current day

1.3 Algorithms

1.3.1 Country Dance Algorithm

The purpose of the Country Dance algorithm is to provide a state estimation for any given state in an HMM. The Country Dance algorithm takes all evidence into account, including evidence prior to and subsequent to the query variable. It does this using a forward pass and a backward pass through the states in the HMM.

To perform the forward pass, the algorithm computes a new value probability estimation using the evidence values from the current time slice 't' and the probability state estimation from the previous time slice 't-1'. In this way, the forward pass calculation is independent of state values from two or more time slices prior, and it is independent of any evidence values from outside the current time slice.

To perform the backward pass, calculates the probability of viewing some evidence values in time slice 't' given the state estimation in time slice 't-1'. In this way, the algorithm moves backwards using the values estimated by the forward pass, and calculates an inference value for each time step. An element wise multiplication is then carried out for each time slice using the values calculated from the forward and backward passes for that time slice.

1.3.2 Fixed Lag Smoothing

The Fixed Lag Smoothing algorithm is intended to smooth over a set time lag of d steps. The Fixed Lag Smoothing algorithm takes in the current evidence, the current Hidden Markov Model, and the length of the lag for smoothing. It then returns the new smoothed estimate given the observation of a new time step. Something that can be noted is the final return of the Fixed Lag Smoothing algorithm is just af x b.

To perform the algorithm we begin by adding the evidence of the current time slice(et) to the end of our list of evidences (et-d:t). Next we initialize a variable to hold the probability of P(et—Xt). We now check if the time slice we are currently at is greater than value of steps performed for Fixed Lag Smoothing(d). If the current time slice is greater than d then we will perform a forward pass. After the forward pass we will remove the first value in et-d:t. Now we will initialize a matrix(Ot-d) to contain P(et-—Xt-d). Next using a variable that was initialized as the identity matrix(B) we perform B = ((Ot-d)-1)(T(transition matrix)-1)(BTOt). In the case the earlier check comparing the current time slice to d was false then we would simply perform B = BTOt. Lastly we will increment the time slice by 1 and check if the new time slice is greater than d + 1, since we don' want to ruin the case of if the prior time slice was equal to d. after checking if the new time slice is greater then we will return the normalized(fxB1), otherwise we will return null. It is important to note that for this algorithm some of the variables are global in order for their values or lists to be updated regardless of what the algorithm returns.

This algorithm is a smoothing algorithm, which means that we compute the posterior distribution over the past states given all evidence up to the present state.Smoothing is considered because it is able to incorporate evidence more and provide a better estimate of the state than what was available at the time.

1.3.3 Viterbi

The Viterbi algorithm aims to return the most likely sequence of states given a set of evidence. This is most used when a set of observations was collected, but when states were not. This has application in deciphering missing pieces of historical data. For instance, in our second domain that we tested, historians did not record if the day was hot or cold. However, Bob's journal includes notes about how many ice cream cones they ate that day. Using this evidence, we can try to decipher if the missing days were hot or cold. To implement the Viterbi algorithm, we first generated a random set of evidence from the possible evidence variables in a domain. For the first time step, we utilized the prior probability table and obtained a value for each state in the network in conjunction with the observation table for each state. For all time steps beyond this, we also needed to include the probability of a future state based on a current state through the transition table. This would visually form an "X", since we need to test each state in the network transitioning to every other possible state. We would then obtain a final set of values by utilizing the transition probabilities and the probability of a state given evidence. The maximum value was assigned to the state, which would then feed into the next time step. This would repeat until we analyzed all of our observations. To retrieve the most likely sequence of states, we would then need to work backwards through our generated network. For each time step, we would select the state that had the highest probability. In our case, we stored each state in a list, and after identifying the most probable state in the first time step, we returned the entirety of the list.

To verify the Viterbi algorithm results, a pre-determined set of evidence varibles were used, and the problem was worked out by hand. Since the domain only had two states

1.4 Results

As you can see in figure 1, the country dance algorithm produces a probability estimation of about 72.7% for state 1 being True, 27.56% for state 2, and about 10.45% for the final state in the 'Sleeping in Class' HMM (the HMM is provided in exercise 17 of the book). We performed hand calculation for the first two states and these values matched up perfectly. This calculation took about 0.0003845 seconds to complete, or 384.5 nanoseconds, averaged over 50 runs.

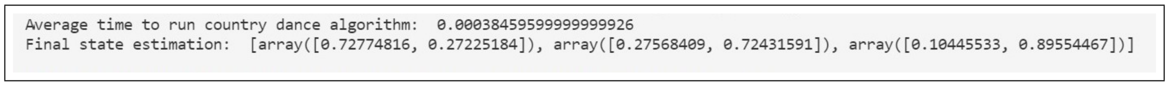
Meanwhile, in figure 3, we see that the fixed lag smoothing algorithm calculates the probability of the final state being True in the Exercise 17 HMM as 30.66%. This was surprising because it is very different from the country dance value of 10.45%. The reason for this may be because of how different these two algorithms are, since fixed lag smoothing does not involve the backward pass that country dance uses.

In figure 2, we see that the country dance algorithm computes the probability of the final state being True (or 'Hot') as 55.35%. As shown, the algorithm performs this calculation in about 0.00018904 seconds, or 189.04 nanoseconds on average, when averaged over 50 runs. This is marginally less time than the 'Sleeping in Class' HMM took, which makes sense because the weather HMM had one less evidence variable.

In figure 4 and 5, we see that fixed lag smoothing took an average of 1938 and 778 nanoseconds to run, respectively. This is much longer than country dance, which makes sense, because fixed lag smoothing requires more passes through the evidence variables than country dance does (three passes in our case).

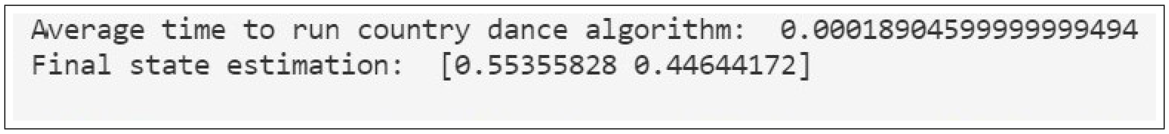
In figure 6 and 7 we see that the Viterbi algorithm has a very fast runtime and the based on the list output we can see the states determined at each time slice. when we compare this to the probabilities determined from the Forward-Backward and Fixed Lag Smoothing algorithm we can see that the probabilities generated from those algorithms correlates with what the expected output given from the Viterbi algorithm.

1.5 Reference



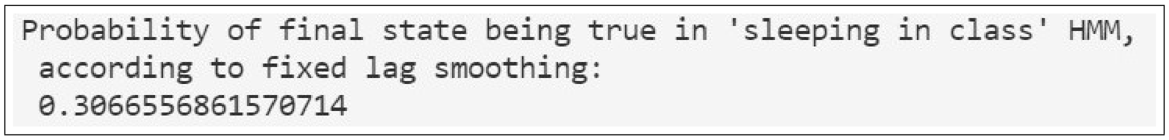
```
Average time to run country dance algorithm:  0.00038459599999999926
Final state estimation:  [array([0.72774816, 0.27225184]), array([0.27568409, 0.72431591]), array([0.10445533, 0.89554467])]
```

Figure 1: Time take to run the country dance algorithm, and the probability estimations for each value of each state in the "Sleeping in Class" HMM.



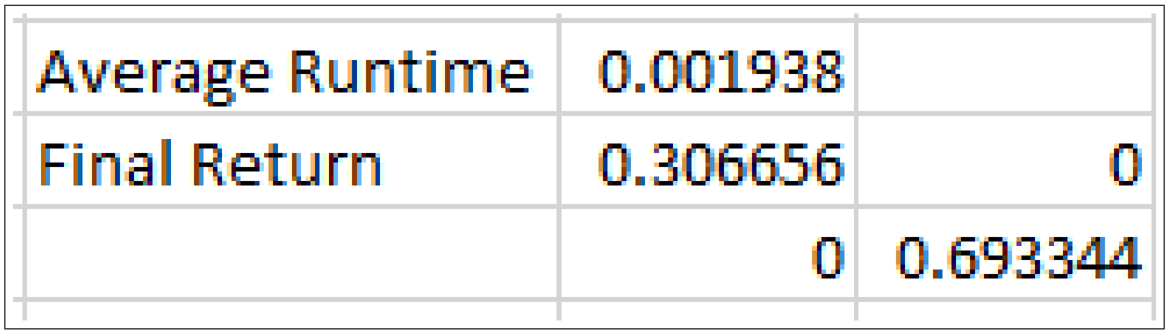
```
Average time to run country dance algorithm:  0.00018904599999999494
Final state estimation:  [0.55355828 0.44644172]
```

Figure 2: Time take to run the country dance algorithm, and the probability estimation for the final state value in the "Weather" HMM.



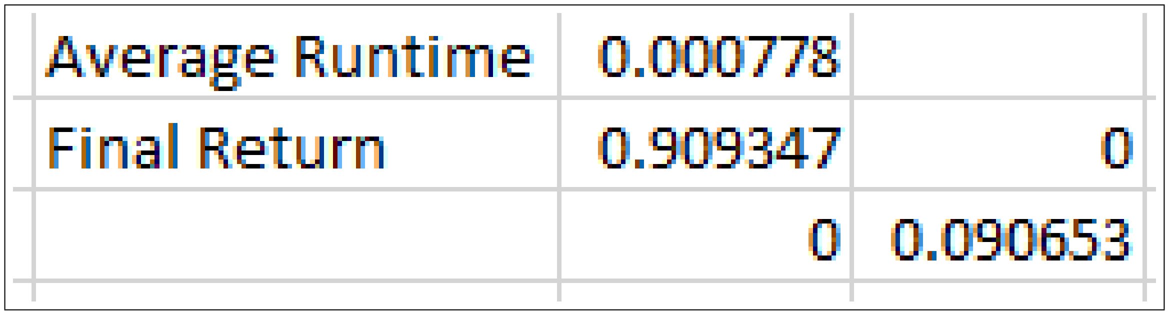
```
Probability of final state being true in 'sleeping in class' HMM,
 according to fixed lag smoothing:
 0.3066556861570714
```

Figure 3: The probability of the state being 'True', as calculated by fixed lag smoothing for the 'sleeping in class' HMM.



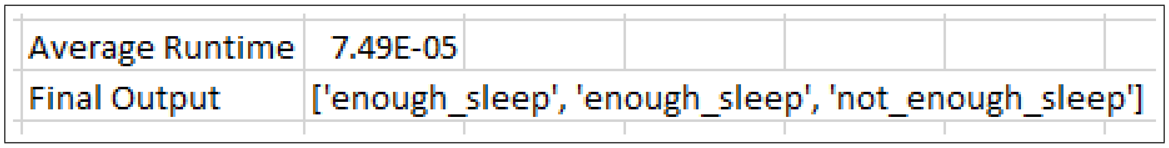
Average Runtime	0.001938	
Final Return	0.306656	0
	0	0.693344

Figure 4: The average time and an example output from running the Fixed Lag Smoothing Algorithm for 3 evidence pairs using the sleeping in class domain, the index 0,0 of the return indicates the True state and 1,1 the False state



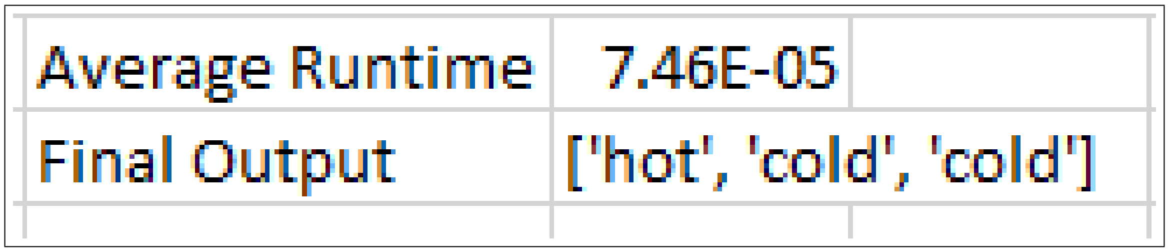
Average Runtime	0.000778	
Final Return	0.909347	0
	0	0.090653

Figure 5: The average time and an example output from running the Fixed Lag Smoothing Algorithm for 3 evidence pairs using the weather domain, the index 0,0 of the return indicates the True state and 1,1 the False state



Average Runtime	7.49E-05
Final Output	['enough_sleep', 'enough_sleep', 'not_enough_sleep']

Figure 6: The average time and an example output from running the Viterbi Algorithm for 3 evidence pairs using the sleeping in class domain



Average Runtime	7.46E-05
Final Output	['hot', 'cold', 'cold']

Figure 7: The average time and an example output from running the Viterbi Algorithm for 3 evidence pairs using the weather domain

2 Part B: Dynamic Bayesian Network

2.1 Problem Overview

Dynamic Bayesian Networks are Bayesian Networks that relate variables over adjacent time steps. The idea of using Dynamic Bayesian Networks is because at any time T can be determined by using internal regressors and immediate prior values.

2.2 Domains

2.2.1 Robot localization

The first domain involves robot localization where a robot is placed randomly into a maze and must determine its location. The maze generated will also contain obstacles which are spaces that the robot cannot pass through. When the robot runs into an obstacle it must change direction. The robot has a preference to moving forward(the direction it is facing) but if noise is added it will occasionally change direction. The robot has multiple parameters such as what moves are possible for the robot to make and also determine the direction the robot is facing using North, South, East, West.

2.2.2 Pac-Man

The second domain is based off of the old video game Pac-Man. In this domain we utilize the robot player from the previous domain but now we initialize moving obstacles that act as ghosts with the goal for Pac-Man to be able to judge the probability that any given move could result in running into a ghost.

2.3 Particle Filtering

Particle Filtering is a filtering type algorithm that takes in a Dynamic Bayesian Network, new incoming evidence, and the amount of samples. The output is a A vector of weighted samples(S) of size N (number of samples). The Particle Filtering algorithm involves sampling the current slice variables in topological order.

The process of the algorithm begins with starting a for loop with the initialized variable(i) incrementing to the number of samples. Next we sample from P(Xi — X0) and insert it into S[i]. After that we use a vector of size N to hold P(e—S[i]). After we exit the for loop we perform a weighted sample with replacement using N, S, and W and store it into S and return S.

2.4 Results

The results observed from the Particle Filtering algorithm demonstrate that To obtain results, we utilized a run script. This run script would exhaustively run through incrementing all noise variables. Action noise would be incremented from 0 to 0.1, observation noise would be incremented from 0 to 0.1, and action bias would be incremented from 0 to 0.1. Each combination of values were tested. This entire process was repeated by changing the value of N (the number of samples taken) from 1, to 10, to 100, and finally to 1000.

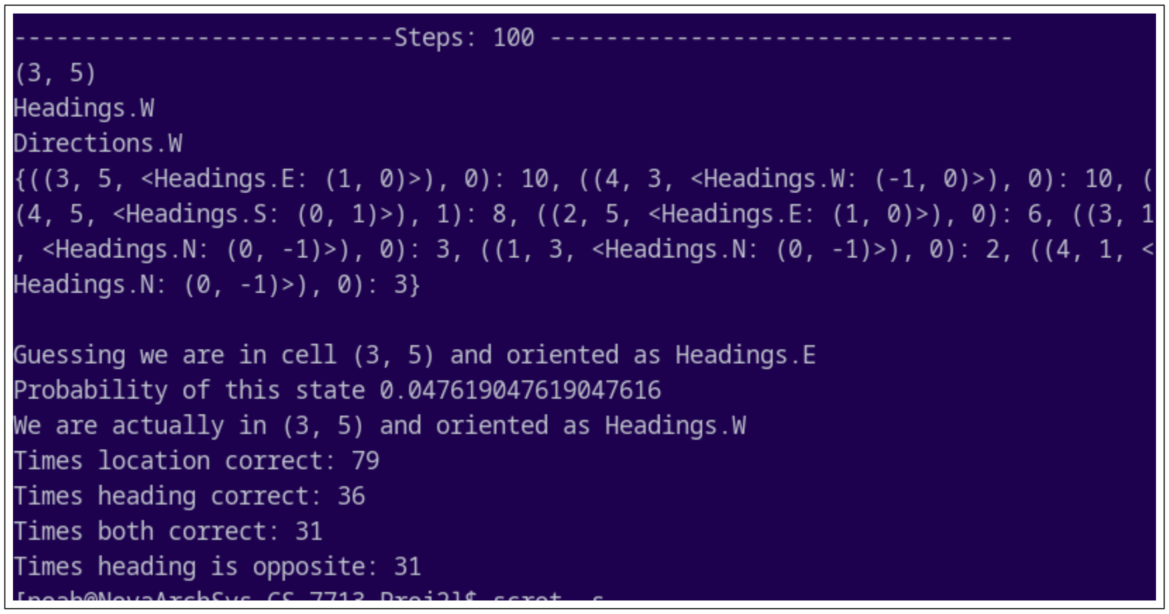
The results for this DBN, though expansive, are unfortunately not representative of the expected result. When sampling values as part of the particle filtering algorithm, we made a poor decision in terms of how to sample. When we sample, we do so randomly through numpy's random.choice() function. As a result, we do not intelligently choose samples. Despite changing noise values in the system, since we are just randomly choosing a sample, the noise variables do not make an effect on the outcome of the results.

To correct this, we would need better sampling logic. Ideally we would use a non-uniform distribution based on the weights. In the project code, we are storing weights, and we are normalizing through the weights and number of samples. In addition, we are properly utilizing the transition tables provided as well as the observation table. Meaning, most of the leg-work has already been done in regards to make a weighted-sampling function. To continue from this point, we could make a probability list based on the weights of each sample, and use numpy's choice function based on the probability distribution rather than randomly, which allows us to have a greater likelihood of sampling a more likely cell and heading.

Regardless of the poor sampling choice, we are receiving fairly decent results. We are able to correctly predict the robot's location with an accuracy of 50-60 percent, the robot's heading 20-30 percent, and both heading and location correct 10-20 percent of the time. In some cases, our robot's location can be correctly predicted up to 80 percent of the time. While these are not indicative of the optimal results, we are still performing better than if we were to arbitrarily guess a random cell and heading.

Refer to the attached data file for the full set of results.

2.5 Reference



```
---------------------------Steps: 100 ---------------------------------
(3, 5)
Headings.W
Directions.W
{((3, 5, <Headings.E: (1, 0)>), 0): 10, ((4, 3, <Headings.W: (-1, 0)>), 0): 10, (
(4, 5, <Headings.S: (0, 1)>), 1): 8, ((2, 5, <Headings.E: (1, 0)>), 0): 6, ((3, 1
, <Headings.N: (0, -1)>), 0): 3, ((1, 3, <Headings.N: (0, -1)>), 0): 2, ((4, 1, <
Headings.N: (0, -1)>), 0): 3}

Guessing we are in cell (3, 5) and oriented as Headings.E
Probability of this state 0.047619047619047616
We are actually in (3, 5) and oriented as Headings.W
Times location correct: 79
Times heading correct: 36
Times both correct: 31
Times heading is opposite: 31
[noah@NovaArchSys CS 7713 Proj2]$ scrot -s
```

Figure 8: Statistics for DBN.