AI Lab Report No 2 - Group 16

Pan, Jiaqi; Bhargavan, Sudarsan; Hechehouche, Hacene

October 14, 2017

Introduction

This report is turned in as an observation for the second laboratory assignment which was tasked with guiding a **Ranger** through a graph of **Water Holes** in order to find a croc.

The Ranger has to traverse through a graph which maps all the waterholes as vertices, there are 3 attributes that are given which aids the ranger in finding the croc, also there are two Swedish backpackers who have the risk of getting eaten by the croc, but if they get eaten then, the location of the croc is easily identified.

The main objective of this lab assignment is to discuss the advantages and limitations of the **Hidden Markov Models** by performing multiple simulations of **Find the Croc Program**, and also to implement and optimize **Hidden Markov Models**, & various other strategies to find the most turn effective solution.

Tools Used

- R Language
- R Markdown
- R Studio

Hidden Markov Model Theory:

Hidden Markov Model (HMM) is a statistical Markov model in which the system being modeled is assumed to be a Markov process with unobserved (i.e. hidden) states [2]

Properties

A Hidden Markov Model (HMM) consists of two (in our case discrete) random variables O and Y, which change their state sequentially. The variable Y with states $\{y_1, y_2, \ldots, y_n\}$ is called the “hidden variable”, since its state is not directly observable. The state of Y changes sequentially.

Markov Property. This means, that the state change probability of Y only depends on its current state and does not change in time. Formally we write: P(Y(t+1)= y_i | Y(0)…Y(t)) = P(Y(t+1)= y_i | Y(t)) = P(Y(2)= y_i|Y(1)).

The variable O with states $\{O_1, O_2 \ldots, O_m\}$ is called the "observable variable", since its state can be directly observed. O does not have a Markov Property, but its state probability depends statically on the current state of Y.

Formally, an HMM is defined as a tuple M=(n,m,P,A,B), where n is the number of hidden states, m is the number of observable states, P is an n-dimensional vector containing initial hidden state probabilities, A is the n*n-dimensional "transition matrix" containing the transition probabilities such that A[i,j] = P(Y(t) = y_i| Y(t-1) = y_i) and B is the m*n-dimensional "emission matrix" containing the observation probabilities such that B[i,j] = P(O = O_i | Y = y_j)

Applications

There are three main Applications for Hidden Markov Model [1]:

Evaluation: Given a sequence O of observations and a model M, what is the probability P(O|M) that sequence O was generated by model M. The Evaluation problem can be efficiently solved using the Forward algorithm

Decoding: Given a sequence O of observations and a model M, find the most likely sequence of hidden variables to generate these observations. The Decoding problem can be efficiently solved using the Viterbi algorithm.

Learning: Given a sequence O of observations, find what model is the most likely one to generate this sequence. The Learning problem can be efficiently solved using the Baum-Welch algorithm.

Example of Hidden Markov Model [3]

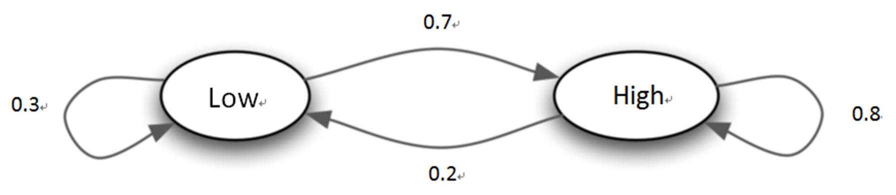

Figure 1: transition diagram

• Two states : 'Low' and 'High' atmospheric pressure.

• Two observations : 'Rain' and 'Dry'.

•Transition probabilities: P('Low'|'Low')=0.3 ,P('High'|'Low')=0.7 , P('Low'|'High')=0.2, P('High'|'High')=0.8

Or transition probabilities can be represented in transition matrix:

$$T = \begin{bmatrix} 0.3 & 0.7 \\ 0.2 & 0.8 \end{bmatrix}$$

•Observation probabilities: P('Rain'|'Low')=0.6, P('Dry'|'Low')=0.4, P('Rain'|'High')=0.3, P('Dry'|'High')=0.7 .

Or observation probabilities can be represented in emission matrix:

E = $\begin{bmatrix} 0.6 & 0.4 \\ 0.3 & 0.7 \end{bmatrix}$ or E_rain = $\begin{bmatrix} 0.6 & 0 \\ 0 & 0.3 \end{bmatrix}$ and E_dry = $\begin{bmatrix} 0.4 & 0 \\ 0 & 0.7 \end{bmatrix}$

• Initial probabilities: P('Low')=0.4 , P('High')=0.6 .

Or represented in matrix: S_0 = [0.4, 0.6].

Current State Estimation – Forward Algorithm

Forward Algorithm is used together with HMM to calculate current state. We want to calculate the probability distribution for the state variable at a certain time: St, given the distribution over the initial state: S0, and sequence of observations up to that point, O1:t.

We use a dynamic programming algorithm to iteratively calculate the probability distribution for the state variable at times 1,2,...,t given the distribution of the previous state variable and the current observation.

Let $S_{i,t}$ represent the state i at time t with N possible states and O_t represent the observation at time t. Because many state paths can give rise to the same observationO_t, we must add the probabilities for all possible paths to obtain the full probability of$S_{i,t}$. In equation (1) the forward algorithm computes $f(S_{i,t})$ under HMM.

$$f(S_{i,t}) = \sum_{k=1}^{N}[f(S_{k,t-1})P(S_{i,t}|S_{k,t-1})]P(O_t|S_{i,t}) \tag{1}$$

$$P(S_{i,t})\ \alpha\ f(S_{i,t}) \tag{2}$$

And since $f(S_{i,t})$ is proportional to $P(S_{i,t})$ as shown in equation (2), we can calculate for each i ≤ N, the value of $f(S_{i,t})$ and normalize it to get value of $S_{i,t}$ for each i at time t.

Example of Calculating Current State:

Continuing with our "Low and High, Rain and Dry" example, suppose we want to calculate current state under a sequence of observations in our example, {'Dry','Rain'}, and under the initial state {0.4, 0.6} of being dry or rain.

We can calculate the current state by using equation (1)(2)

$$f(S_{low,1}) = (0.4 * 0.3 + 0.6 * 0.2) * 0.4 = 0.096 \rightarrow P(S_{low,1}) = 0.15$$

$$f(S_{high,1}) = (0.4 * 0.7 + 0.6 * 0.8) * 0.7 = 0.532 \rightarrow P(S_{high,1}) = 0.85$$

$$f(S_{low,2}) = (0.15 * 0.3 + 0.85 * 0.2) * 0.6 = 0.129 \rightarrow P(S_{low,2}) = 0.35$$

$$f(S_{high,2}) = (0.15 * 0.7 + 0.85 * 0.8) * 0.3 = 0.236 \rightarrow P(S_{low,2}) = 0.65$$

We can also use the below matrix multiplication (equation (3)(4)), which is equivalent to the equation (1)(2):

$$S'_t = S_{t-1} * T \qquad (3)$$

$$S_t \alpha S'_t * E \qquad (4)$$

Initial state to first state:

[0.4, 0.6] * $\begin{bmatrix} 0.3 & 0.7 \\ 0.2 & 0.8 \end{bmatrix}$ = [0.24, 0.76]

[0.24, 0.76] * $\begin{bmatrix} 0.4 & 0 \\ 0 & 0.7 \end{bmatrix}$ = [0.096, 0.532]   [0.15, 0.85]

First state to second state:

[0.15, 0.85] * $\begin{bmatrix} 0.3 & 0.7 \\ 0.2 & 0.8 \end{bmatrix}$ = [0.215, 0.785]

[0.215, 0.785] * $\begin{bmatrix} 0.6 & 0 \\ 0 & 0.3 \end{bmatrix}$ = [0.129, 0.236]   [0.35, 0.65]

We find the current state is [0.35, 0.65] which means the atmospheric pressure on the second day is 35% probability of being low and 65% of probability of being high.

Resources

[1] Lawrence R. Rabiner (February 1989). “A tutorial on Hidden Markov Models and selected applications in speech recognition”. Proceedings of the IEEE 77 (2): 257-286. doi:10.1109/5.18626

[2] Stanford university, Machine Learning course lectures.

[3] https://www.cse.buffalo.edu/

[4] Lecture slides

Hidden Markov Model Implementation:

We use the Hidden Markov model to estimate the current state of the system given current knowledge. Since we are only interested in croc's current position, there is no need to apply smoothing to estimate prior states. So we only implement forward algorithm of HMM and always estimate the current state of the croc's position.

Since we can get emissions from the croc and decide its current state before the ranger move, and also since without readings from the croc, prediction can be very imprecise, we don't use the model to do prediction.

Observable variables:

We utilized the following observable variables:

1. The distribution of the water conditions in all of the 40 waterholes: probs

   It gives the distribution of three measures: phosphate, salinity and nitrogen respectively in each waterhole. Because the water conditions of one waterhole measured in different time vary, we cannot give a specific value for a measure in each waterhole. But we know that the phosphate, salinity and nitrogen values all obey the normal distribution with a known mean value and standard deviation, and their distributions in each waterhole vary. So we can find for a given value which waterhole it is more likely to appear and the probability of it appears in each waterhole.

2. The water condition measured by the censor on the croc: readings

   It gives real time values of phosphate, salinity and nitrogen of the waterhole where the croc stays. We compare these values to the phosphate, salinity and nitrogen's distributions of each waterhole to decide the probability that these values are obtained from each waterhole.

3. The position of Swedish backpackers: positions

   If the Swedish backpackers are alive, we know the croc is not in the same waterholes as they are. If one of the Swedish backpacker died, we know the croc is in the waterhole where he died.

4. The connectivity of waterholes: edges

   All the paths between waterholes are given so that we can find out which waterhole the croc can move to and the probability of the transition in the next step provided with its current position. Also, we can find out the shortest path from ranger's current position to one of waterholes as destination.

Initial state:

We define *state* as a vector in which each value is the probability of the croc staying in the corresponding waterhole. Here, we use a 1 * 40 row vector *'state'* to describe the state information. For example, if *state[3] = 0.2*, it means the probability of the croc stays in waterhole No.3 is 0.2.

Assume the very first position of the croc before it making any move is P1. We define the *initial state* as the state that describes P1. We can simply start with setting the initial state uniformly, meaning all the 40 values in state vector equal 1/40 because the probabilities that the croc being in different waterholes are the same.

However, we notice that before the croc making the first move, it gives out information of the water condition of its current waterhole, so we can use this observation to adjust the initial state. We do the modification of initial state with the help of emission matrix: we use the uniform initial state to multiply the emission matrix column by column to get a new 1 * 40 vector, and then we normalize the vector making the sum of the 40 values equals 1. This is the same as what we do in the step 2 of forward algorithm in HMM: using emissions to adjust the state. After doing this modification of a uniform probability distribution, we get a more precise estimation of the probabilities of the croc being in each waterhole at the beginning and thus getting a more accurate initial state to describe the croc's initial position.

Transition matrices:

The transition matrix is obtained through the connectivity information of waterholes. We use a 40 * 40 matrix *tranM* to represent the transition probability from each one of the waterholes to all the 40 waterholes. We use the row ID to represent the current position and the column ID to represent the next position to move to. For example, *tranM*[2,3] = 0.2 means the probability of moving from waterhole No.2 to waterhole No.3 is 0.2. The sum of probabilities that move from one waterhole to all the 40 waterholes is 1. So in our case, the sum of each row of *tranM* equals to 1.

To calculate each value in the matrix, we first find out the adjacent nodes of a particularly node N use the provided function *getOptions(N, edges).* Where 'N' is the node we inspect and 'edges' is the connectivity information of waterholes. The function returns a vector of N's neighboring nodes. We set the corresponding positions of neighboring nodes in row N in *tranM* to 1. Since the croc may stay in the current waterhole, we also set *tranM[N,N]* to 1. We loop through all rows and inspect each node and set the corresponding columns of the possible transitions to 1. Because the probabilities of transit from current waterhole to all neighbors (including itself) are the same, we normalize by the row to get the probability of each transitions. After calculated in the beginning, the transition matrix will not change when the croc moves.

Example:

in row 10, *tranM*[10,1] = *tranM*[10,4] = *tranM*[10,10] = *tranM*[10,30] = 1, there are 4 possible transitions from waterhole No.10 in total. After normalization, *tranM*[10,1] = *tranM*[10,4] = *tranM*[10,10] = *tranM*[10,30] = 0.25, meaning the probabilities of transit from node 10 to node 1,4,10 or 30 are all equal to 0.25.

Emission matrices:

We have the distributions of phosphate, salinity and nitrogen respectively in each waterhole, which means we have the probability density function (p.d.f.) for these attributes in each waterhole. Then we input the phosphate value *PH* (which we get from vector *'readings*') of croc to the p.d.f. of phosphate of waterhole *W* to get if the croc is in waterhole *W*, the probability that it obtain a phosphate value *PH*. We use the function dnorm() to approximate this emission probability:

phosphateP = P (obtain the phosphate of value PH | *Croc is in water hole W) =*

dnorm(PH, mean=probs$phosphate[W,1], sd=probs$phosphate[W,2])

Then we apply this dnorm() function to all these three attributes and get values of salinityP, phosphateP, and nitrogenP in waterhole W. Since the three measures are independent, we multiply the three values to get if the croc is in water hole *W*, the probability that it obtain its salinity, phosphate and nitrogen values as values in *readings* vector simultaneously.

P (Croc's sensor gives its three readings like these simultaneously | *Croc is in water hole W)*

*= salinityP * phosphateP * nitrogenP*

Now, we get the emission probability from waterhole W. We apply these steps to all these 40 waterholes using a loop to get a 1*40 vector of probabilities that Croc's sensor takes on these values given that it is in each of the waterhole.

Steps of HMM implementation:

1. Obtain the initial state in the beginning or we get our current state from moveInfo$mem.
2. Use the state to multiply the transition matrix. Here we use matrix multiplication and what we get is a transitional state vector.
3. Multiply the transitional state vector with emission vector column by column.
4. Inspect the position information of Swedish backpackers. We modify the state that if they are not dead, we set the corresponding position of state vector where swedes are to 1. If one of them dead, we set the corresponding position of state vector where he died to 0.
5. Normalize the state vector and then store it in moveInfo$mem.
6. Check the largest value in the state vector, the position holding this largest value is the position that has the largest possibility to see the croc. We extract this position.
7. Implement a path finding algorithm to find out the shortest path from ranger's current position to the most promising position to see the croc. And then decide next steps for the ranger.

Additional Strategies:

Breath First Search

It is a search algorithm that was invented to get the **Shortest Path** out of a maze under *Certain Conditions*, as opposed to *Depth First Search* which was invented to find any path out of a maze. It is used for traversing *graph* or *tree* data structures. It beings at an arbitrary vertex, exploring neighboring vertices before moving to the next level of neighbors. Worst-case performance of *Breath First Search* is similar to that of *Depth First Search* and is given by

$$O(|V| + |E|)$$

It is a linear search algorithm where *O* is the sum of the *Number of Vertices* and the *Number of Edges*. It uses a *Queue data-structure* as opposed to the *Stack data-structure* used by *Depth First Search*. This enables *Breath First Search* to search all adjacent vertices before moving on to the next level neighbors.

It tracks all the *Visited Vertices* by *Marking* them, which prevents *Infinite Loops*. The procedure is as follows:

Initialization:

Un-mark all the vertices.

En-queue the starting vertex.

Traversal:

While the queue is not empty.

De-queue the vertex.

Mark it.

En-queue all adjacent unmarked vertices.

Stop when is goal is marked.

Why Breath First Search?

In un-weighted graphs BFS can construct a Shortest Path from vertex u to v. It guarantees to compute Smallest number of moves to solve a puzzle in a given State Space

Since we keep track of visited nodes in this un-directed graph, we avoid Infinite loops.

Since we have an un-weighted graph, there is no need to use other more complicated path finding algorithm such as Dijkstra Algorithm

Implementation of BFS:

Here we've used the BREATH FIRST SEARCH ALGORITHM as it guarantees to compute the Smallest Number of Moves to aid the Ranger reach the Croc

BFS gives us a list of Vertices that are to be Traversed to reach the Waterhole Where The Croc Currently Is.

BFS uses a Queue Data structure so whatever vertex is first en-queued first is de-queued first

Moving strategies Discussions

We discussed three different strategies on using BFS to help Move & Search, viz;

- **Move and Search in each turn:**

Case 1: Here we first move to the adjacent waterhole and search for the croc, this is helpful only when the croc is close. When the croc is far away, searching only takes up an extra turn and is useless.

The croc's estimated position is fairly precise (About 80% accurate), so it is better to move faster when the croc is far away.

The average of 1000 runs was found to be 7.336.

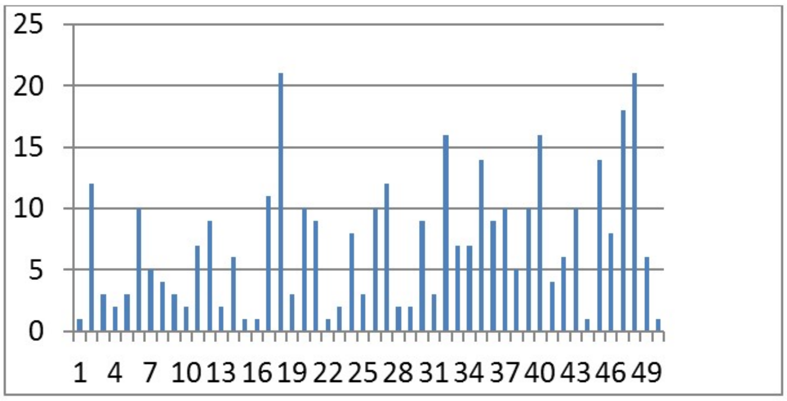

Figure 2: Graph of 50 Runs in case 1

- **When the croc is close then, Move and Search or else Move Twice:**

Case 2: Here we move and search in each turn if the croc's estimated position is less than or equal to 2 edges away from the ranger. If that is not the case then we move twice in each turn.

The average of 1000 runs was found to be 4.864.

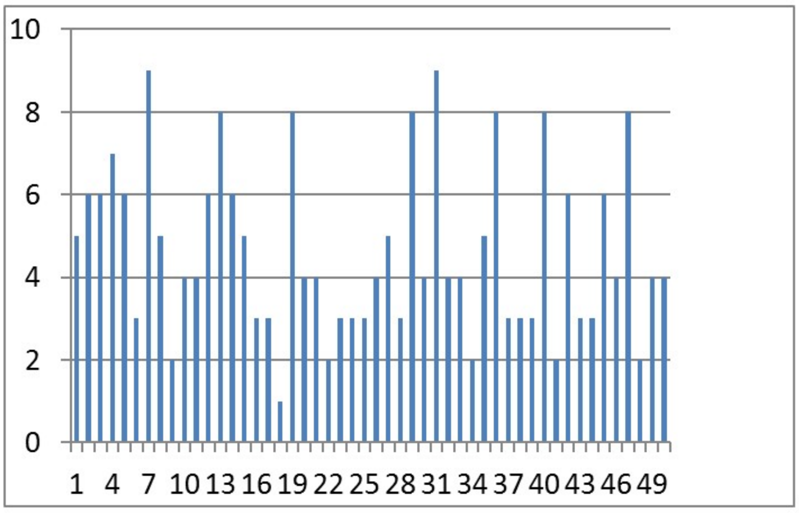

Figure 3: Graph of 50 Runs in case 2

- **When the croc and ranger are next to each other, Move and Search or else Move Twice:**

Case 3: Here we move and search in each turn if the croc's estimated position is the adjacent vertex of the ranger's position. If that is not the case then we move twice in each turn. When compared with the previous case the improvement is small.

The average of 1000 runs was found to be 4.411.

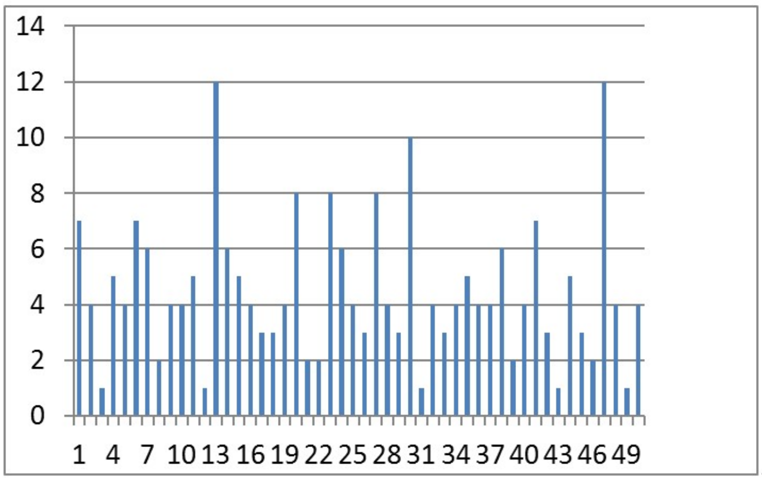

Figure 4: Graph of 50 Runs in case 3

Conclusion:

It is evident from the averages that **Moving & Searching when the ranger and the croc are next to each other, else moving twice** was the best fit to be implemented to quickly reach the croc with minimal turns usage.