WHERE'S CROC
LAB ASSIGNMENT

Group 6

Choi, Inho

Oh, Jaehee

Osipov, George

Takhchi, Medhi

Part 1 HMM – Theoretical Overview

A theoretical overview of the Hidden Markov Models, including an explanation of the current state estimation algorithm.

HMM Theoretical Overview

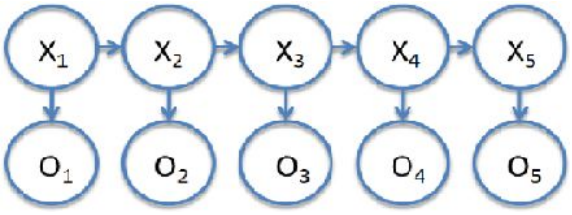
CURRENT STATE ESTIMATION ALGORITHM

Hidden Markov Model

HMM (Hidden Markov Model) is a temporal probabilistic model in which the observed system is assumed to evolve stochastically, the states of the system are hidden from us and the information is obtained from some observations which are conditioned upon system's current state.

In this model, the state of the process is described by a single discrete random variable. The values of the variable are the possible states of the world. In case when we have two or more state variables, we can still describe the system by HMM using a "megavariable" (combining multiple variables into one). Restricted structure of HMM allows for a simple matrix implementation of basic algorithms.

To define HMM framework we must describe the transition model and the sensor model.



In the picture above, X_t is a hidden variable at time t. In this example, we consider the simplest possible model - first-order Markov chain: here probability distribution of X_t depends only on the X_{t-1}:
$P(X_t \mid X_{t-1} X_{t-2} \dots X_1) = P(X_t \mid X_{t-1})$

Transition in HMM is described by a stochastic matrix T. The evolution of the system from X_t to X_{t+1} can be formally described like this:
$X_{t+1} = TX_t$
(this is a property specific to first-order Markov chain; state vectors are assumed to be column vectors, matrix T is assumed to have normalized columns).

The process is believed to be stationary, i.e. T does not change over time. Generally speaking, the rules that govern the process do not change - the distribution of $P(X_t \mid X_{t-1} X_{t-2} \dots)$ is stationary.

O_t is an observation obtained at time t (the arrow from X_t means that it's determined by X_t). Observations are conditioned only upon the current state of the system: O_t is independent of all variables but X_t.

Let us consider an example. Suppose we want to analyze a sequence of DNA (which in our case can be viewed as a string consisting of 4 letters: A, C, T, G - one for each amino acid). Suppose we have processed a big body of data (sequences with corresponding labels) and have come up with a transition matrix T with size 4x4. Here, entry T[1,3] tells us, what is the probability of seeing A after T

(the order ACTG is respected; matrix has normalized columns). We are designing a way to analyze a new sequence of amino acids. We test amino acids sequentially. Our hidden state is the type of currently tested amino acid. Our source of information are some tests that report numbers. We concatenate these numbers together in a vector - this is our observation in this model. Observation is likely not to be precise: each number in vector O_t will let us calculate some probability based on the parameters of our tests (for example, mean and standard deviation). This will be the probability that this observation was obtained given that the state was equal to one of the amino acids, e.g. $P(O_t \mid X_t = \text{'C'})$. This model is useful for our purpose of analyzing DNA. In the next segment, we will discuss inference methods more precisely.

State estimation in HMM

As we discussed previously, current state of the system is hidden from us. Frequently, our goal is to estimate the state given current knowledge (all evidence to the date). This way the rational agent is able to keep track of the process. We need a useful filtering algorithm to maintain the current state and update it rather than go back over the entire history for at each step. In the latter case the cost of each update would increase with time. So we want a function f such that:

$$P(X_{t+1} \mid e_{1:t+1}) = f(\, e_{t+1}, P(X_t \mid e_{1:t}) \,)$$

To clarify f, let us first consider, what our prediction for the state X_{t+1} would be, given the observations $e_{1:t}$: $P(X_{t+1} \mid e_{1:t})$
Now consider an update from our prediction with the new evidence e_{t+1}:

$$P(X_{t+1} \mid e_{1:t})\, P(e_{t+1} \mid X_{t+1}) = P(X_{t+1} \mid e_{1:t})\, P(e_{t+1} \mid X_{t+1}\, e_{1:t}) = \alpha\, P(X_{t+1} \mid e_{1:t+1})$$

The first equality is due to observation model - the assumption that e_{t+1} is independent of everything but X_{t+1}, this we can add it to condition without changing anything.
Second equality is due to Bayes rule. α here denotes a normalization factor.

f is propagated forward, modified by each transition and updated by each observation:

$$f_{1:t+1} = \alpha\, \text{FORWARD}(f_{1:t}, e_{t+1}),$$

where FORWARD performs the recursive update described above. The base case for the recursion is $f_{1:0} = P(X_0)$.

We illustrate how the state estimation algorithm works in terms of our DNA example. Suppose our transition matrix looks like this:

0.1	0.2	0.4	0.1
0.2	0.2	0.4	0.2
0.3	0.4	0.1	0.6
0.4	0.2	0.1	0.1

First amino acid is equally likely to be any of them. So $P(X_0)$ looks like this:

0.25
0.25
0.25
0.25

By definition, $f_{1:0} = P(X_0)$.

Let's calculate our prediction for the next state $P(X_1)$ (we have no observations yet):

$$P(X_1) = T\ P(X_0)$$

0.20
0.25
0.35
0.20

Suppose our observation yields this probability vector $P(e_1 \mid X_1)$:

0.1
0.2
0.4
0.3

Now we can estimate current state X_1 based on the prediction and evidence. We simply multiply them, normalize the result and get:

0.05
0.05
0.7
0.2

Thus, we conclude that the second amino acid in the sequence is C with probability of 70%, T with probability 20% and A and G with probabilities 5% each.

We save current state vector and continue in this manner.

[References : *Russell, Stuart Jonathan; Norvig, Peter* Artificial intelligence : a modern approach 3. ed.: Boston: Pearson Education, cop. 2010]

Where's Croc
LAB ASSIGNMENT

Part 2 Our HMM Implementation

Including a discussion of the observable variables you utilized, the parameters of your transition and emission matrices, and your initial state.

Methods used in our code

OUR HMM IMPLEMENTATION

States:

In our implementation the state represents the position of crocodile - it is a vector S of dimensions (1 x numWaterholes), in which S[i] corresponds to the probability of croc being in i^{th} waterhole at the time of our turn.

Initial state :

Before starting the game, we have no information about croc. That is why our initial vector is uniform over all waterholes: initial vector has 1/40 probability at each index.
We save the initial state to memory at "moveInfo$mem[1]" so we can retrieve it in future.

Using Hidden Markov Model

Ranger's inference in the game of WheresCroc is well described by a hidden Markov model: croc's position is the hidden state, the readings from sensors are the observations conditioned upon the state.

Transition

The transition from a waterhole to another is ensured by a transition matrix which is just the adjacency matrix with normalized rows (because moves to each neighbor as well as staying in place are equally possible at each turn).

```
getAdjacencyMatrix=function(edges) {
        adj = matrix(0, nrow=numWaterpools, ncol=numWaterpools)
        # mark edges in one direction
        adj[edges] = 1
        # mark edges in other direction (symmetrically)
        adj = adj + t(adj)
        # mark each waterpool as adjacent to itself
        for (i in 1:numWaterpools) {
                adj[i,i] = 1
        }
        return (adj)
}

transitionMatrix=function(adj) {
        for (i in 1:numWaterpools) {
                adj[i,] = adj[i,] / sum(adj[i,])
        }
        return (adj)
}
```

We normalize rows because our state vector is a row vector and the transition is modeled by multiplying this vector by matrix.

Observation

Observations are the readings of salinity, phosphate and nitrogen from croc's sensors at his current location.

Observations yield a probability vector E of size (1 x numWaterholes). E[i] is the product of the probabilities that each of 3 readings was taken at i^{th} waterhole,because they are independent of one another. These probabilities are calculated with the command dnorm: given a reading, the mean and the standard deviation, it returns the desired probability.

```
processObservation=function(readings,probs) {
        p = rep(1, numWaterpools)
        p = p * dnorm(readings[1], mean=probs$salinity[,1], sd=(probs$salinity[,2]))
        p = p * dnorm(readings[2], mean=probs$phosphate[,1], sd=(probs$phosphate[,2]))
        p = p * dnorm(readings[3], mean=probs$nitrogen[,1], sd=(probs$nitrogen[,2]))
        return (p)
}
```

Tourist positions are also part of our observation. We will cover how we make you of them in the next part.

State estimation

We incorporate our knowledge about tourist positions into our state vector: when a tourist is eaten, we know for sure that croc is at the position where the event occurred (this position is indicated by a negative number in "positions" vector). If a tourist is alive, we can be sure that her current position is not occupied by the croc, so we can set probability of croc being there to 0.

We estimate current state vector by combining information from readings and our prediction of croc's position based on the previous state (which is stored in "moveInfo$mem[1]"). To get the prediction, we multiply previous state vector by transition matrix. After that, we multiply the result by observation probability vector and normalize the result.

```
# Calculate current state vector

# if a tourist was eaten, we can determine where the crocodile is
eatenTourists = which(positions < 0)
if (length(eatenTourists) > 0) {
        stateVector = rep(0, numWaterpools)
        stateVector[-eatenTourists[1]] = 1
} else {
        stateVector = moveInfo$mem[[1]]
        stateVector = stateVector %*% moveInfo$mem[[2]]
        if (!is.na(positions[1]) && positions[1] > 0) {
                stateVector[positions[1]] = 0
        }
        if (!is.na(positions[2]) && positions[2] > 0) {
                stateVector[positions[2]] = 0
        }
        stateVector = stateVector * processObservation(readings,probs)
        stateVector = stateVector / sum(stateVector)
}
```

Where's Croc
LAB ASSIGNMENT

Part 3 Additional Strategies

Including your route finding algorithm for moving between locations and your choice of locations to move to.

Methods used in our code

ADDITIONAL STRATEGIES

Apart from Hidden Markov Model techniques, we've employed other strategies in our implementation of hmmWC. We used them to decide on our next action.

Minimizing distance between ranger and croc

We start with an observation that the problem of catching the crocodile is equivalent to minimizing the distance between ranger and croc. This sentence should be taken with scrutiny, especially for edge cases - when the crocodile and ranger are close to each other (in a more general case, it is obvious that minimizing distance is the way to go - you cannot catch the croc until that distance is more than 0). Let us consider 2 cases:

- when the distance between ranger and croc is equal to 1
- when the distance between ranger and croc is equal to 2

In the first case, the best action is, obviously, to move to croc's position and search it. This does not interfere with our observation as long as we follow each one-step transition with a search.

In the second case, we cannot catch croc immediately, so there are at least 2 more turns left in the game. According to our strategy, we perform a two-step action and result at the same waterhole with croc. Croc moves one step at a time, so whatever her next move is, we can perform a (transition + search) action to the waterhole where she lands. So it takes us exactly 2 moves to transition in this case, which is the minimal number of step. Therefore, our strategy yields an optimal result in this case as well.

We could have built up a rigorous mathematical proof by induction based on these 2 cases.

Choosing next action

At each turn, ranger has the probability distribution for croc's position (state vector in Hidden Markov Model). We can use this information to choose an action that would minimize the *expected distance* from ranger to croc. To do this, we consider the position of ranger after performing each action and count the expected distance from there to croc. After considering each possible action, we choose one that minimizes expected distance.

It still remains unclear, how do we compute the expected distance. We need to know the distance between each node to all other nodes. Suppose we are given this information in a form of a matrix of size (numWaterholes x numWaterholes). Let us denote this matrix by D. D[**i**,**j**] gives us the number of steps it takes reach **j** from **i**. Obviously, this matrix is symmetric.

We will discuss the derivation of matrix D in the next section and focus on choosing the next action. Given the state vector and D, we can compute the result of each action: we simply take the index of the waterhole where ranger ends up as the result of a given action (call it **x**), take corresponding row from D (D[x,]) and consider its dot product with the state vector, i.e. the probability distribution for croc's position.

$$E(d) = \sum_{i=1}^{N} P(C = i \mid S_{t-1}\, O_t) * d(x_a, i)$$

where:
N - number of waterholes
d - distance between ranger and crocodile
C - crocodile's actual position
S_{t-1} - state vector at time *(t-1)*
O_t - observation (salinity, nitrogen and phosphate measurements) at time *t*
x_a - ranger's position as a result of an action *a*

Derivation of matrix of distances

To derive the matrix of distances we use the powers of adjacency matrix.

The adjacency matrix A is a square matrix of size (numWaterholes x numWaterholes), containing 1-s for those pairs of [i,j] that are connected with an edge, all other terms being 0-s. This matrix is symmetric in the case of undirected graph (which is precisely the case in "Where is Croc").

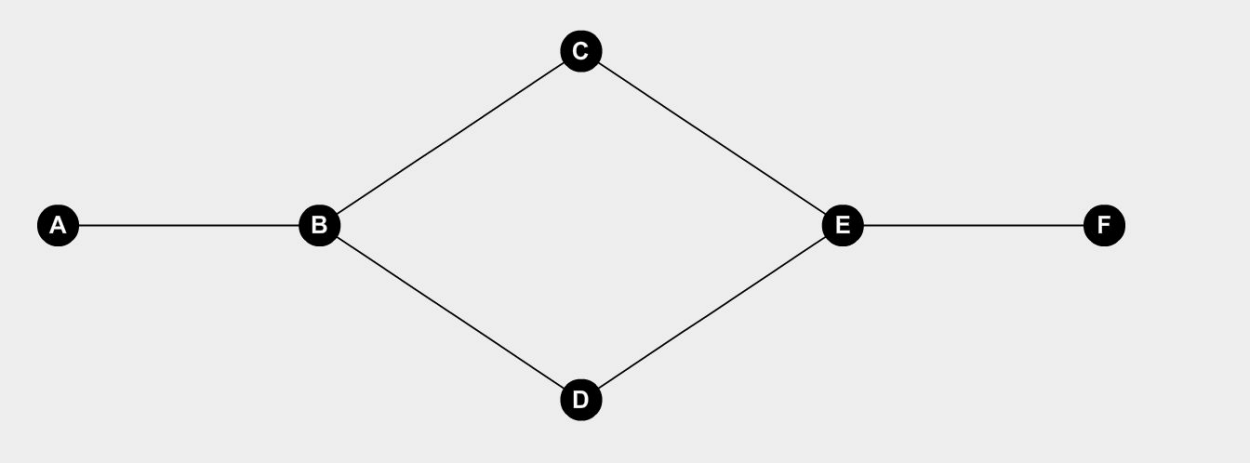
We can generalize our notion of adjacency matrix by considering it an adjacency matrix of degree 1: here A[**i**,**j**] > 0 iff **j** is reachable from **i** in a walk of length 1.
Using this generalized notion, we introduce adjacency matrix of degree **n**. Our claim is that A^n is the adjacency matrix of degree **n**, where A^n[**i**,**j**] > 0 iff **j** is reachable from **i** in a walk of length **n**.

We prove our claim by induction:

1) The base case (n = 1) is evident from the definition of adjacency matrix.
2) Suppose we are given a matrix A^{n-1}. Let us consider an entry of $A^n = A^{n-1} * A$:
A^n[i,j] = A^{n-1}[i,] * A[,j] ← taking the dot product of i^{th} row of A^n by j^{th} row of A
A^n[i,j] > 0 iff there is an index **k** such that A^{n-1}[i,k] > 0 and A[k,j] > 0.
If A^{n-1}[i,k] > 0, there must be a way of length **n-1** to get from i to k.
A[k,j] > 0 means that there is an edge between k and j.
Subsequently, if these two quantities happen to be positive, the must be a way of length **n** to get from i to j through k. On the contrary, if there is no such node **k** through which we can get to j, then there is no way to get from i to j in **n** steps, and the corresponding entry will be 0.

∎

Let us consider an example to explain the way we generate the matrix of distances in more details.



Our example graph consists of 6 nodes: A, B, C, D, E, F.
It also contains following edges: (AB), (BC), (BD), (CE), (DE), (EF)

Adjacency matrix for this graph would look like this:

1	A	B	C	D	E	F
A	1	1	0	0	0	0
B	1	1	1	1	0	0
C	0	1	1	0	1	0
D	0	1	0	1	1	0
E	0	0	1	1	1	1
F	0	0	0	0	1	1

1-s are included on the diagonal as each node is reachable from itself.

At this point, our matrix of distances coincides with the adjacency matrix, with the only difference that entries on the main diagonal are 0 for the distance matrix.
Now let's see what do the powers of the adjacency matrix look like:

2	A	B	C	D	E	F
A	2	2	1	1	0	0
B	2	4	2	2	2	0
C	1	2	3	2	2	1

D	1	2	2	3	2	1
E	0	1	2	2	4	2
F	0	0	1	1	2	2

3	A	B	C	D	E	F
A	4	6	3	3	2	0
B	6	10	8	8	6	2
C	3	8	7	6	8	3
D	3	8	6	7	8	3
E	2	6	8	8	10	6
F	0	2	3	3	6	4

4	A	B	C	D	E	F
A	10	16	11	11	8	1
B	16	32	24	24	24	8
C	11	24	23	22	24	11
D	11	24	22	23	24	11
E	8	24	24	24	32	16
F	1	8	11	11	16	10

In each of these matrices, we've highlighted the entries that were 0 in the previous matrix. The number in the upper left corner - the power of adjacency matrix - corresponds to the number of steps it takes to reach node j from i for each highlighted element [i,j].

As a result, we get this matrix of distances:

	A	B	C	D	E	F
A	0	1	2	2	3	4
B	1	0	1	1	2	3

C	2	1	0	0	1	2
D	2	1	0	0	1	2
E	3	2	1	1	0	1
F	4	3	2	2	1	0

It can be easily verified that the distance matrix is generated correctly.