Hidden Markov Model

Ping-Jung Liu November 2nd 2017

1. Introduction

Hidden Markov Model is a standard method of integrating state informations and sensor observations. For example, judging the weather by counting umbrellas, or calculating the probability of thief if alarm activated. In this project, I will use hidden markov model, HMM for the rest of this report, to observe mazes. Note I used observe instead of solve; because this time, the program will not solve a maze on its own. Rather, the program will calculate the probability distribution of the robot's position at each step, given a list of sensor readings. The floors of the mazes in this project will be of four colors: Red, Green, Yellow, Blue. The robot will return an observation after each step; a example sensor reading: ["r", "y", "b", "r"]. In each step, the robot will attempt to move in any direction with 0.25 probability, because there are four directions available. If the robot hits a wall or maze boundary, it will remain at the current location. After the step, the sensor attached to the robot will return the color of the current location. But note the result will only be 88% times correct, which means if the robot is on red not, the sensor has 88% probability of returning red, and 4% of returning any of the other three colors.

Specifically, I implemented filtering and foward-backward smoothing to calculate the probability distribution. I built a transition matrix of states, locations in this case, and sensor models for all four colors to help carry out the algorithms. The functionalities of all the features will be demonstrated by reasonings and interesting test results in later sections. In the end, I wil provide a brief review on a research paper about HMM in the field of speech recognition.

2. State Transitions Matrix

The most important component of the project is the state transition model, T, a matrix that indicates Prob(Xt+1|Xt), the probability of moving to a state from current state. For example, T11 contains the probability of moving to position 1 from position 1 in the maze problem.

Code of build_transition as follow:

```
def build_transition(self):

    # initialize an all 0s matrix of size num_state * num_state
    matrix = self.build_matrix(self.maze.num_state(), self.maze.num_state())

    # loop through all locations
    for x in range(0, self.maze.width):
        for y in range(0, self.maze.height):
            state_num = self.maze.state_num(x, y)
            # ignore walls
            if state_num == -1:
                continue

            # find the neighbors
            neighbors = self.maze.find_neighbors(x, y)
            wall_num = 5 - len(neighbors)
            # P of staying at current location
            matrix[state_num][state_num] = wall_num/4

            # P of moving to neighbor locations
            for neighbor in neighbors:
                if not neighbor == (x, y):
                    matrix[state_num][self.maze.state_num(neighbor[0], neighbor[1])] = 0.25

    return matrix
```

Given the comments, the code should be extremely clear. I will demonstrate the functionality of this function with an simple example, maze1.

```
rb
gr

[[0.5, 0.25, 0.25, 0],
[0.25, 0.5, 0, 0.25],
[0.25, 0, 0.5, 0.25],
```

```
[0, 0.25, 0.25, 0.5]]
```

Let's briefly examine the result. There are four states in this problem; four possible locations. Consider the first row, the transition model of the upper left postion. The probabilities of trying to move up and left are both 0.25, so the probability of staying would be 0.25 + 0.25 = 0.5. The probabilities of moving to the right, b, and moving down, g are also 0.25, which are indicated by the second and third element in the first row. And there is no way to reach bottom right, so the probability is 0.

3. Sensor Model

As suggested by Professor Belkom and the book, sensor models shuold be diagonal matrices for computation simplicity. Since there are four different colors in the maze, I created four diagonal matrices, O, to represent the sensor models. Oii = 0.88 if the ith state is the right color, Oii = 0.04 if not. It would be easier to see an example.

```
rb
gr

red:
[[0.88, 0, 0, 0],
[0, 0.04, 0, 0],
[0, 0, 0.04, 0],
[0, 0, 0, 0.88]]

green:
[[0.04, 0, 0, 0],
[0, 0.04, 0, 0],
[0, 0, 0.88, 0],
[0, 0, 0, 0.04]]

yellow:
[[0.04, 0, 0, 0],
[0, 0.04, 0, 0],
[0, 0, 0.04, 0],
[0, 0, 0, 0.04]]

blue:
[[0.04, 0, 0, 0],
[0, 0.88, 0, 0],
[0, 0, 0.04, 0],
[0, 0, 0, 0.04]]
```

There are no yellow floor in the maze, so the yellow model consists of only 0.04. Position 0, upper left, and position 3, bottom right, are both red, so RED00 and RED33 are both 0.88.

4. Filtering

I implemented a filtering function using the forward method described in the book to calculate the probability distribution of each step. The formula is as follow: f(1:t+1) = norm(O(t+1) * T' * f(1:t)). f1:t is the probability distributions for the robot locations from time 1 to time t. O is the sensor model. T' is the transpose of the transition matrix. Provided a list of sensor readings of each attempted step, the function will loop through all steps and save the probability distribution of every step into a result list. Code as follow:

```
# norm(sensor_model * transition' * result)
def filter(self):

    trans_transition = self.transpose(self.transition)

    for i in range(1, len(self.sensor_reading)):
        self.result[i] = self.m_times_vec(trans_transition, self.result[i - 1])

        if self.sensor_reading[i] == "r":
            self.result[i] = self.m_times_vec(self.sensor[0], self.result[i])
        elif self.sensor_reading[i] == "g":
            self.result[i] = self.m_times_vec(self.sensor[1], self.result[i])
        elif self.sensor_reading[i] == "y":
            self.result[i] = self.m_times_vec(self.sensor[2], self.result[i])
```

```
        elif self.sensor_reading[i] == "b":
            self.result[i] = self.m_times_vec(self.sensor[3], self.result[i])

    for v in self.result:
        v = self.normalize(v)

    # round to 1/1000
    self.clean(self.result)
    return self.result
```

There are several helper functions I wrote to make matrices calculations easier. Transpose(M) returns M transposed. m-times-vec(m, v) returns the result of matrix m times vector v. v-times-v(v1, v2), though not used in this function, returns the multiplications of the elements of corresponding index. Other than the helper functions, the code completely follows the formula and should be clear to read. Testing results will be shown in later section.

5. Forward Backward Smoothing (Extra Credit)

In our specific problem, the precisions of sensors are in fact not ideal with a 88% correct probability. The forward-backward algoithm provides a better way to deal with lists of noisy observations. Its main idea is to use future observations to help calculate the probability distribution of past states. As described in the book, the backward formula is as follow: b(k+1:t) = T * O(k+1:t) * b(k+2:t), while the forward formula remains the same as the one described in filtering section. Closely following the pseudocode on page 576, the codes of forward_backward as follow:

```
# forward-backward smoothing
def fb_smoothing(self, filter_result):

    # initialize back and result lists
    back = []
    for i in range(0, self.maze.num_state()):
        back.append(1)
    result = []
    for i in range(0, len(filter_result)):
        result.append(0)

    # follow the formula, starts from the end
    t = len(self.sensor_reading) - 1

    while t >= 1:
        # v_times_v multiply the elements of corresponding index
        result[t] = self.v_times_v(filter_result[t], back)
        result[t] = self.normalize(result[t])
        back = self.b(back, t - 1)
        t = t - 1

    result = result[1:len(result)]
    self.clean(result)
    return result
```

The function "b" is the backward function; it follows the formula I just described:

```
# transition * sensor_model * result
def b(self, back, t):

    temp = []

    if self.sensor_reading[t] == "r":
        temp = self.m_times_vec(self.sensor[0], back)
    elif self.sensor_reading[t] == "g":
        temp = self.m_times_vec(self.sensor[1], back)
    elif self.sensor_reading[t] == "y":
        temp = self.m_times_vec(self.sensor[2], back)
    elif self.sensor_reading[t] == "b":
        temp = self.m_times_vec(self.sensor[3], back)

    result = self.m_times_vec(self.transition, temp)
    return result
```

6. Testing

Since the functionalities of build transition and build sensor model have been shown in previous sections, this section will focus on the results of filtering and forward-backward smoothing. Though I designed a program to randomly picked steps and exactly iminating the robots, I found it easier to test if I gave it a pre-generated list of sensor readings and actual locations.

First, I tested filtering with maze2.maz.
sensor reading = ["r", "y", "b", "y", "g"] actual locations = [(1, 3), (0, 3), (0, 2), (1, 2), (0, 2), (0, 1)] (including the start location)
results:

```
step: 0
observed color: X
actual position: (1, 3)

rgyg
yb#r
g##g
bbgy

0.077 0.077 0.077 0.077

0.077 0.077 ##### 0.077

0.077 ##### ##### 0.077

0.077 0.077 0.077 0.077

-------------------------------------
step: 1
observed color: r
actual position: (0, 3)

rgyg
yb#r
g##g
bbgy

0.4  0.019 0.019 0.019

0.019 0.019 ##### 0.4

0.019 ##### ##### 0.019

0.019 0.019 0.019 0.019

-------------------------------------
step: 2
observed color: y
actual position: (0, 2)

rgyg
yb#r
g##g
bbgy

0.051 0.028 0.097 0.028

0.603 0.005 ##### 0.051

0.005 ##### ##### 0.028

0.005 0.005 0.005 0.097

-------------------------------------
step: 3
```

```
observed color: b
actual position: (1, 2)

rgyg
yb#r
g##g
bbgy

0.041 0.01  0.014 0.012

0.037 0.775 ##### 0.009

0.034 ##### ##### 0.012

0.022 0.022 0.007 0.013

-------------------------------------
step: 4
observed color: y
actual position: (0, 2)

rgyg
yb#r
g##g
bbgy

0.006 0.035 0.044 0.002

0.796 0.066 ##### 0.002

0.006 ##### ##### 0.002

0.004 0.003 0.002 0.038

-------------------------------------
step: 5
observed color: g
actual position: (0, 1)

rgyg
yb#r
g##g
bbgy

0.032 0.121 0.005 0.04

0.033 0.036 ##### 0.001

0.661 ##### ##### 0.035

0.001 0.001 0.037 0.003

-------------------------------------
```

The forward filtering method did a decent job at calculating the probability distribution. Then I used forward-backward smoothing to calculate the same maze with identical inputs.

```
step: 0
observed color: X
actual position: (1, 3)

rgyg
yb#r
g##g
bbgy
```

```
NNNAAA
-------------------------------------
step: 1
observed color: r
actual position: (0, 3)

rgyg
yb#r
g##g
bbgy

0.911 0.023 0.002 0.002

0.024 0.004 ##### 0.029

0.003 ##### ##### 0.002

0.002 0.001 0.001 0.002

-------------------------------------
step: 2
observed color: y
actual position: (0, 2)

rgyg
yb#r
g##g
bbgy

0.064 0.012 0.058 0.009

0.78  0.008 ##### 0.002

0.007 ##### ##### 0.009

0.001 0.001 0.002 0.055

-------------------------------------
step: 3
observed color: b
actual position: (1, 2)

rgyg
yb#r
g##g
bbgy

0.005 0.007 0.003 0.002

0.025 0.937 ##### 0.001

0.006 ##### ##### 0.002

0.007 0.007 0.002 0.002

-------------------------------------
step: 4
observed color: y
actual position: (0, 2)

rgyg
yb#r
g##g
bbgy

0.006 0.033 0.076 0.002
```

```
0.746 0.062 ##### 0.001

0.006 ##### ##### 0.002

0.001 0.001 0.002 0.066

-------------------------------------
step: 5
observed color: g
actual position: (0, 1)

rgyg
yb#r
g##g
bbgy

0.032 0.121 0.005 0.04

0.033 0.036 ##### 0.001

0.658 ##### ##### 0.035

0.001 0.001 0.037 0.003

-------------------------------------
```

We can see forward-backward smoothing also produce logical calculations, and performed tremendously better than forward filtering on step 1 and step 3.

If you would like to see the program calculate the possible locations of a real random robot, please refer to the README for instructions. I believe the provided two test cases are enough to show the two functions truly worked.

7. Literature Review (Extra Credit)

In addition to the project itself, I reviewed the paper “Hidden Markov Models for Speech Recognition by B. H. Juang and L. R. Rabiner” Technometrics, Vol. 33, No. 3. (Aug., 1991), pp. 251-272. Natural language processing and speech recognition has been an essential field of AI for a long time. I chose this paper first because HMM is the most widely used techniques for speech recognition, second because it provides in depth but understandable explanations to how HMM is applied.

Just like other HMM problems, speech recognition could be separated into three components: evaluation, estimation, and decoding. A forward induction procedure makes the evaluation of P(O|model) linear, thus, allows us to score how well an unknown observation sequence matches a model efficiently. The estimation problem involves assigning right parameters for model given an observation sequence, in other words, training. Maximizing log likelihood is a traditional but effective method to approach this problem. HMM then provides a straightforward training procedure given the goal of maximizing P(O|model). The decoding problem calculates the most likely state sequence given the observation sequence O. The Viterbi algorithm is the most sounded methods of finding state sequence with highest probability of resulting in an observation sequence.

The strengths of HMM’s application in speech recognition includes its versatility and ease of implementation. But the area required further research in modeling criteria, incorporating nonspectral features prior linguistic knowledge. HMM systems could achieve recognition rates up to 95% in some tasks and it is expected that HMM-based speech recognition systems could become a part of our daily lives in the near future.