HMM Report

Implementation

Models

I designed both the sensor and the transition model as 16x16 matrices to allow for computation. My sensor model is implemented as a 4x4x16 three-dimensional array. The first dimension represents the color (0 for red, 1 for blue, 2 for green, 3 for yellow). The rest can be thought of as a 16x16 matrix. I designed it this way so that I could use a single nested for loop for simplicity sake. This allows me to return a single sensor model which made my filtering code simpler to write. The sensor models represent the probability of seeing a given color at a location on the board. For my transition model, each place in the matrix represents the probability of moving from one state to another. For instance trans_model[1][1] would represent the probability of the robot not moving at all at index 1. The robot always tries to move in all four possible directions, but if there is an obstruction in one of the directions the robot does not move giving the starting state a transition probability. In comparison, trans_model[1][0] would represent the probability of the robot moving from index 1 to 0. Excerpts for both of these models can be seen below.

Sensor model

```
    def generate_sensor_models(self):
```

```
        """ generates sensor model as   a 3d matrix
            to save computation time and so that i
            can return the entire model"""
        possible_states = self.generate_intial()
        #print(possible_states)
        sensor_model  = np.zeros((4, 16, 16))
        colors = ['R', 'B', 'G', 'Y']
        for state in possible_states:
            for color in colors:
                if self.maze.get_color(state[0], state[1]
) == color:
                    sensor_model[colors.index(color), sel
f.maze.index(state[0], state[1]), self.maze.index(state[0
], state[1])] = .88
                else:
                    sensor_model[colors.index(color), sel
f.maze.index(state[0], state[1]), self.maze.index(state[0
], state[1])] = .04
        #print(sensor_model[0,self.maze.index(2,3),self.m
aze.index(2,3)])
        return sensor_model
```

Transition model

```
  def get_transition_model(self):
        """generate a transition model"""
        possible_states = self.generate_intial() # get th
```

```
e possible states
        trans_model = np.zeros((16, 16)) # 16X16 matrix
        for state in possible_states: # go through all po
ssible states

            possible_moves = self.get_successors(state)
            num_moves = len(possible_moves)
            for new_state in possible_moves:
                # add the probability of that state movin
g to the next state to the transition matrix
                # will add higher than .25 probablity if
it stays on the same state for some of the possible moves
                trans_model[self.maze.index(state[0], sta
te[1]), self.maze.index(new_state[0], new_state[1])] = po
ssible_moves.count(new_state) * .25
        return trans_model
```

Filtering

I recursively implemented filtering by cutting the first number off a sensor evidence array until its length is zero. For each iteration of the recursion, I read the first character out of the sensor evidence array and set the guest_model equal to my sensor model for the character read out (so R for red, etc.) The transition matrix represents the probability of moving from one state to the next state, but I wanted the probability of going “into” a given state, so I transposed my transition matrix for the next calculation. I then implemented the code below which I inferred from the textbook.

```
     def solver(self, forward, sensor_evidence, count=0):
 # solves using HMM filtering
        print("Step " + str(count) + ":")
        print_solution(normalize(forward))

        if len(sensor_evidence) == 0: ## base case for re
cursion
            #print(forward)
            return forward

        # pick sensor model picks the sensor model based
on what the reading from the sensor is
        reading  =  sensor_evidence[0]
        if reading == 'R':
            guessed_model = self.sensor_model[0, :, :]
        if reading == 'B':
            guessed_model = self.sensor_model[1, :, :]
        if reading == 'G':
            guessed_model = self.sensor_model[2, :, :]
        if reading == 'Y':
            guessed_model = self.sensor_model[3, :, :]


        tranposed_model = np.transpose(self.transition_mo
del)
        temp = np.matmul(tranposed_model, forward)
```

```
        forward = np.matmul(guessed_model, temp)
        return self.solver(forward, sensor_evidence[1:], count+1) # recurse
```

I then normalize and display the probability distribution in a 4X4 grid where each number represents the probability of the robot being there at the given time step

Testing

To test I created 3 mazes as below:

maze1:

```
##R#
#BGY
#R#Y
#B##
```

maze2:

```
##B#
#YRY
#B#B
##GR
```

maze3:
```
```

```
##B#
#YBG
#R#Y
#RRG
```

I then made up a few path in these mazes and changed some
 of the sensor readings according to the probabilities:
For maze 1:
1. ['B', 'R', 'B', 'G', 'Y']
2. ['Y', 'Y', 'G', 'R', 'G', 'B', 'R', 'B']
3. ['B', 'G', 'Y', 'G', 'R', 'G', 'B', 'R', 'B', 'G', 'Y'
, 'Y']

For maze 2:
1. ['B', 'R', 'Y', 'B', 'R', 'G']
2. ['G', 'R', 'B', 'Y', 'R', 'Y', 'B']

For maze 3:
1. ['Y','B', 'G', 'Y', 'G','R','R','R' ,'Y','B','B']

Here is the algorithm ran for maze 1 and path 1:
```
Solver ran on the correct path:
['B', 'R', 'B', 'G', 'Y']
Step 0:
0.062   0.062   0.062   0.062
```

```
0.062   0.062   0.062   0.062
0.062   0.062   0.062   0.062
0.062   0.062   0.062   0.062
Step 1:
0.000   0.000   0.020   0.000
0.000   0.449   0.020   0.020
0.000   0.020   0.000   0.020
0.000   0.449   0.000   0.000
Step 2:
0.000   0.000   0.071   0.000
0.000   0.037   0.020   0.003
0.000   0.812   0.000   0.003
0.000   0.054   0.000   0.000
Step 3:
0.000   0.000   0.005   0.000
0.000   0.459   0.003   0.001
0.000   0.039   0.000   0.000
0.000   0.493   0.000   0.000
Step 4:
0.000   0.000   0.001   0.000
0.000   0.069   0.744   0.000
0.000   0.075   0.000   0.000
0.000   0.110   0.000   0.000
Step 5:
0.000   0.000   0.038   0.000
0.000   0.049   0.041   0.834
0.000   0.017   0.000   0.001
0.000   0.021   0.000   0.000
```

```
Solver ran on the sensor reading simulated path :
['B', 'R', 'R', 'G', 'Y']
Step 0:
0.062   0.062   0.062   0.062
0.062   0.062   0.062   0.062
0.062   0.062   0.062   0.062
0.062   0.062   0.062   0.062
Step 1:
0.000   0.000   0.020   0.000
0.000   0.449   0.020   0.020
0.000   0.020   0.000   0.020
0.000   0.449   0.000   0.000
Step 2:
0.000   0.000   0.071   0.000
0.000   0.037   0.020   0.003
0.000   0.812   0.000   0.003
0.000   0.054   0.000   0.000
Step 3:
0.000   0.000   0.114   0.000
0.000   0.020   0.003   0.001
0.000   0.841   0.000   0.000
0.000   0.022   0.000   0.000
Step 4:
0.000   0.000   0.050   0.000
0.000   0.128   0.439   0.001
0.000   0.250   0.000   0.000
0.000   0.132   0.000   0.000
Step 5:
```

0.000   0.000   0.044   0.000
0.000   0.071   0.047   0.730
0.000   0.057   0.000   0.002
0.000   0.049   0.000   0.000
```

We can see that the algorithm predicts the correct location with an 83.4% probability when we have a correct path. When we adjust for sensor error the probability drops to 73%. This is explained by the fact that one of the sensor color on the path was wrong.

Here is the algorithm ran for maze 1 and path 3:
This path is quite a bit more complex as it backtrack.
```
Solver ran on the correct path:
['B', 'G', 'Y', 'G', 'R', 'G', 'B', 'R', 'B', 'G', 'Y', 'Y']
Step 0:
0.062   0.062   0.062   0.062
0.062   0.062   0.062   0.062
0.062   0.062   0.062   0.062
0.062   0.062   0.062   0.062
Step 1:
0.000   0.000   0.020   0.000
0.000   0.449   0.020   0.020
0.000   0.020   0.000   0.020
0.000   0.449   0.000   0.000
Step 2:

```
0.000   0.000   0.006   0.000
0.000   0.064   0.763   0.006
0.000   0.064   0.000   0.006
0.000   0.093   0.000   0.000
Step 3:
0.000   0.000   0.037   0.000
0.000   0.046   0.040   0.823
0.000   0.014   0.000   0.023
0.000   0.016   0.000   0.000
Step 4:
0.000   0.000   0.006   0.000
0.000   0.006   0.872   0.072
0.000   0.004   0.000   0.037
0.000   0.003   0.000   0.000
Step 5:
0.000   0.000   0.850   0.000
0.000   0.039   0.041   0.046
0.000   0.015   0.000   0.008
0.000   0.001   0.000   0.000
Step 6:
0.000   0.000   0.106   0.000
0.000   0.005   0.877   0.006
0.000   0.003   0.000   0.003
0.000   0.001   0.000   0.000
Step 7:
0.000   0.000   0.052   0.000
0.000   0.859   0.044   0.039
0.000   0.001   0.000   0.001
```

```
0.000   0.005   0.000   0.000
Step 8:
0.000   0.000   0.167   0.000
0.000   0.067   0.038   0.005
0.000   0.721   0.000   0.002
0.000   0.001   0.000   0.000
Step 9:
0.000   0.000   0.014   0.000
0.000   0.518   0.007   0.001
0.000   0.040   0.000   0.000
0.000   0.419   0.000   0.000
Step 10:
0.000   0.000   0.003   0.000
0.000   0.071   0.775   0.001
0.000   0.066   0.000   0.000
0.000   0.085   0.000   0.000
Step 11:
0.000   0.000   0.039   0.000
0.000   0.048   0.042   0.840
0.000   0.014   0.000   0.001
0.000   0.016   0.000   0.000
Step 12:
0.000   0.000   0.003   0.000
0.000   0.003   0.017   0.655
0.000   0.002   0.000   0.321
0.000   0.001   0.000   0.000
Solver ran on the sensor reading simulated path :
['B', 'G', 'Y', 'G', 'R', 'Y', 'B', 'R', 'B', 'G', 'R', '
```

```
Y']
Step 0:
0.062   0.062   0.062   0.062
0.062   0.062   0.062   0.062
0.062   0.062   0.062   0.062
0.062   0.062   0.062   0.062
Step 1:
0.000   0.000   0.020   0.000
0.000   0.449   0.020   0.020
0.000   0.020   0.000   0.020
0.000   0.449   0.000   0.000
Step 2:
0.000   0.000   0.006   0.000
0.000   0.064   0.763   0.006
0.000   0.064   0.000   0.006
0.000   0.093   0.000   0.000
Step 3:
0.000   0.000   0.037   0.000
0.000   0.046   0.040   0.823
0.000   0.014   0.000   0.023
0.000   0.016   0.000   0.000
Step 4:
0.000   0.000   0.006   0.000
0.000   0.006   0.872   0.072
0.000   0.004   0.000   0.037
0.000   0.003   0.000   0.000
Step 5:
0.000   0.000   0.850   0.000
```

```
0.000   0.039   0.041   0.046
0.000   0.015   0.000   0.008
0.000   0.001   0.000   0.000
Step 6:
0.000   0.000   0.308   0.000
0.000   0.016   0.116   0.368
0.000   0.008   0.000   0.182
0.000   0.002   0.000   0.000
Step 7:
0.000   0.000   0.137   0.000
0.000   0.453   0.107   0.136
0.000   0.005   0.000   0.121
0.000   0.042   0.000   0.000
Step 8:
0.000   0.000   0.448   0.000
0.000   0.040   0.033   0.020
0.000   0.435   0.000   0.020
0.000   0.005   0.000   0.000
Step 9:
0.000   0.000   0.055   0.000
0.000   0.483   0.022   0.004
0.000   0.037   0.000   0.003
0.000   0.397   0.000   0.000
Step 10:
0.000   0.000   0.012   0.000
0.000   0.065   0.783   0.002
0.000   0.060   0.000   0.001
0.000   0.078   0.000   0.000
```

````
Step 11:
0.000   0.000   0.674   0.000
0.000   0.036   0.032   0.030
0.000   0.216   0.000   0.000
0.000   0.011   0.000   0.000
Step 12:
0.000   0.000   0.314   0.000
0.000   0.049   0.118   0.307
0.000   0.073   0.000   0.101
0.000   0.038   0.000   0.000
```
````

We can see that the algorithm predicts the location pretty well with an 65.5% probability on the adjacent cell to the correct one and a 32.1% probability on the correct one when we have a correct path. This can be explained by the fact that they are both yellow. When we adjust for sensor error the probability drops to 30.7% and 10.1%. This is explained by the fact that 2 of the sensor colors on the path were wrong.
````