CS76 Assignment 6: HMM - Blind Robot with Color Sensor

Himadri Narasimhamurthy | 3/3/19

Introduction

We use a Hidden Markov Model (HMM) to solve the problem of locating a blind robot within a maze where the robot has a partially accurate sensor informing it of which color tile it is on. The HMM works by keeping track of the probability distribution of the robot's location at all the tiles on the map and implements forward-backward smoothing, the viterbi algorithm for choosing the next best state, and a matrix multiplication method using numpy for filtering.

To run the test, you can run the file HMMSolution.py and change the maze given!

The Map Class

The Map class (**Map.py**) was where I set up my HMM problem. The class takes in a .map file in a format where floors are of colors r, g, b, y and walls are # characters.

Here is one possible maze:

```
    #ybbbg
    ##gy##
    #bbg##
    ###y##
    rygygg
    ry##gy
```

Now when the map class reads in the maze file, it borrows from the code from Mazeworld and joins the lines into a list of tiles i.e. for the above example [#, y, b, b, b, g, #, #, g, y, #, #, #, b, b, g, ...] and the index is also borrowed from the mazeworld code. It also contains code to help run the simulation in methods create rand initial and create rand moves.

Now, we have the same functions used to get whether or not we are at a wall or at a tile - this function is useful when we are calculating the number of potential floor tiles we have in order to finalize our distribution:

```
    if x < 0 or x >= self.width:
        return False
    if y < 0 or y >= self.height:
        return False
    if self.map[self.index(x, y)] == "#":
        return False

    return True
```

The maze class also contains the method to populate our first distribution. It works simply and basically fills the maze map values with our original distribution rather than the tiles. The value for the original distribution for any tile which isn't a wall should be 1/num of floor tiles.

HMM

The HMM class is where the majority of the work is done for the updating of the distributions, the filtering, and the retrieval of the best sequence. I will begin with the structures which are initialized in the init namely the transition matrix and the color matrix.

The transition matrix is of maze area * maze area and the color matrix is of size len colors * maze area. The transition matrix increments the distribution values and saves the increment in [new move index][old move index] if the new move index is a floor tile.

```
    #we create the transition matrix
    self.trans = [[0 for i in range(len(self.maze.dist))] for j in range(len(self.maz
e.dist))]

    for x in range(self.maze.width):
        for y in range(self.maze.height):
            if self.maze.is_floor(x, y):
                for neighbor in self.moves:
                    if self.maze.is_floor(x + neighbor[0], y + neighbor[1]):
                        new_ind = self.maze.index(x + neighbor[0], y + neighbor[1])
                        self.trans[new_ind][self.maze.index(x, y)] += 1/len(self.maze
.colors)
                    else:
                        self.trans[self.maze.index(x, y)][self.maze.index(x, y)] += 1
/len(self.maze.colors)
```

For the color matrix, we create a matrix of maze area size for each color in the color list and within it, we set the probability of the color being equal to the color which we are on i.e. if we are calculating for the color r and we have an r next to a g, our color matrix for r would say {r: [0.88 0.04]}

```
    #now we make a colors matrix - dictionary
    self.colormatrix = {}
    for color in self.maze.colors:

        color_matrix = [0 for i in range(len(self.maze.dist))]

        for x in range(self.maze.width):
            for y in range(self.maze.height):
                if self.maze.is_floor(x, y):
                    coords = self.maze.index(x, y)
                    if self.maze.map[coords] == color:
                        color_matrix[coords] = self.accuracy
                    else:
                        color_matrix[coords] = (1 - self.accuracy) / 3
        self.colormatrix[color] = color_matrix
```

The way to run the HMM comes from the run() method which initializes a random start location and a random move sequence in order to run the forward-backwards smoothing and calculate the states. Within run, we call helper functions to locate_bot which loops through the random move list and assigns the actual coordinate location for each move and get-colors which returns the colors that the sensor would return to the robot for each move in the move sequence, taking into accound the inaccuracy of the sensor.

```
    # runs the robot's movement for 5 steps.
def run(self):
    i = self.maze.create_rand_initial()
    m = self.maze.create_rand_moves(6)

    self.initial_loc = i
    self.move_seq = m

    self.robotlocs = self.locate_bot()
    self.color_sequence = self.get_colors()

    self.smoothing()
```

Forward Backward Smoothing

The next part of the HMM model is to run the forward-backward smoothing in order to properly compute the probability distribution over all states at any point in the observation sequence. In order to write this part of my code I consulted both the textbook as well as the following link (https://en.wikipedia.org/wiki/Forward–backward_algorithm).

We first append the original distribution to our state list and then move to forward. The first part is the forward step. We begin by getting the forward row vector by dotting the transition matrix with the states that have been created and then appending the result to forwardstates.

```
    #forward step
    for i in range(len(self.color_sequence)):

        #dot the trans model with each state and then filter
        state = np.dot(self.trans, self.forwardstates[i])
        self.forwardstates.append(state)
```

Now we implement filtering through matrix multiplication. In order to filter the forwardstate, we multiply the color distribution saved in the color matrix dictionary with the forward state and then normalize the distribution.

```
        #filtering using matrix multiplication
        m1 = self.colormatrix[self.color_sequence[i]]
        m2 = self.forwardstates[i+1]
        self.forwardstates[i+1] = np.multiply(m1, m2)
```

Next we have the backwards step where we again loop through the number of colors returned by the sensor, i.e. the number of moves made, in order to get the column from the back (or index -i-1). We then do the same method of multiplication as we did to filter the forwardstate and append that result to bprobs.

```
    #backwards step
    for i in range(len(self.color_sequence)):
        #get negative value as column index
        c = self.color_sequence[-i - 1]

        #calculate in the same way as forwards filtering but with back sequences
        b = np.multiply(self.colormatrix[c], self.bprobs[i])
        self.bprobs.append(b)
```

Finally, we call the function to calculate backwards and forwards row and column vectors in smoothing(). Here, we will loop through the range of moves and get the proper back value by dotting it with transition matrix and then multiply the back value and the forward state to resave it in forward states. We then normalize the dirstribution.

```
#smooth the two row and col matrices
.
.
.
        #dot the trans and back probabilities to get the proper back prob
        back = np.dot(self.trans, self.bprobs[cind])

        #save the states by multiplying back and forward
        self.forwardstates[rind] = np.multiply(back, self.forwardstates[rind])
.
.
.
```

Finally, I step back through the state matrix in order to get the highest probability from each matrix and save it as our final state sequence using the method get_sequence().

HMM Solution

After getting the best possible sequence with the hmm.get*sequence(), we need to output the results properly and I do so by looping through my state list and writing each state distribution into a string using the print*sol function.

The output displays as such for each step:

```
    Step 5 : move: (-1, 0)
     Robot is at (2, 2) and the color sensor says b



      0.0000(#)   0.0003(y)   0.2180(b)   0.0526(b)   0.0028(b)   0.0000(g)
      0.0000(#)   0.0000(#)   0.0121(g)   0.0150(y)   0.0000(#)   0.0000(#)
      0.0000(#)   0.0478(b)   *0.6047(b)*  0.0109(g)   0.0000(#)   0.0000(#)
      0.0000(#)   0.0000(#)   0.0000(#)   0.0017(y)   0.0000(#)   0.0000(#)
      0.0000(r)   0.0015(y)   0.0029(g)   0.0038(y)   0.0070(g)   0.0071(g)
      0.0000(r)   0.0000(y)   0.0000(#)   0.0000(#)   0.0069(g)   0.0048(y)
```