Nate Pierce
CIS479 - P2
HMM

Description:

As can be seen, the filtering and prediction functions work as expected for the first iteration of the HMM. Checking the values below against the example output and the sum of all elements in the matrix confirm this. However, after the first prediction, the probability matrix sums begin to diverge. Significant time was invested in attempting to fix the diverging probabilities.

The general trend in element probabilities matches the output, and the program does correctly identify the same element with highest position probability after all movement and sensing, but the probability values are all scaled as mentioned.

I suspect there is an issue in the normalization part of the filtering function, since the normalization is responsible for driving the sum of all elements in the filtering matrix to 1.0 (and therefore also the prediction matrix). I went through the lecture slides examples and the methods of calculation for the normalization factor are the same - the sum of the products of each element in the conditional probability matrix and the corresponding element in the prior matrix.

Inspecting the filtering function, you will find two for loops - one for determining the normalization factor, and the second which assigns the probability to each element in the matrix. Checking the validity of the prior matrix is easy - simply check the maze of the prediction probability matrix directly before the filtering is performed (or the initial state, in the case of the first iteration). The evidence maze is perhaps a little more tricky - but the fact that the first evidence filter results in *exactly* the correct filtered matrix for the first iteration inspires my confidence of the correctness of the second half of the filtering function.

Also - the transitional probability functionality is integrated in the prediction function, in case you are looking for that function and can't find it.

Screenshot:
https://drive.google.com/file/d/1ICmy8AuCiWgH8sAtDeeROQ0qL63l_n7K/view?usp=sharing

Output:

```
Initial Location Probabilities
2.63   2.63   2.63   2.63   2.63   2.63   2.63
2.63   #   2.63   2.63   #   2.63   2.63
2.63   2.63   2.63   2.63   2.63   2.63   2.63
2.63   #   2.63   2.63   #   2.63   2.63
2.63   2.63   2.63   2.63   2.63   2.63   2.63
2.63   2.63   2.63   2.63   2.63   2.63   2.63
Location before move is  [4, 2]

Filtering after evidence  [0, 0, 0, 0]
```

```
Filtering sum  0.9999999999999997
0.47   0.47   2.01   2.01   0.47   2.01   0.47
0.47   #   2.01   2.01   #   2.01   2.01
2.01   0.47   8.53   8.53   0.47   8.53   2.01
0.47   #   2.01   2.01   #   2.01   2.01
2.01   2.01   8.53   8.53   2.01   8.53   2.01
0.47   2.01   2.01   2.01   2.01   2.01   0.47

Prediction after Action N
Prediction sum is  0.9999999999999999
0.85   0.63   3.46   3.46   0.78   3.30   2.23
1.70   #   7.22   7.22   #   7.22   2.01
0.63   1.43   2.50   2.50   2.08   1.85   2.66
1.70   #   7.22   7.22   #   7.22   2.01
0.78   4.26   2.66   2.66   4.91   2.01   1.43
0.25   0.25   0.40   0.40   0.40   0.25   0.25
Location before move is  [3, 2]

Filtering after evidence  [1, 0, 0, 0]

Filtering sum  1.5488506002606848
0.86   0.03   0.66   0.66   0.03   0.63   0.10
1.72   #   31.06   1.37   #   31.06   0.38
2.69   0.06   2.02   2.02   0.09   1.49   0.50
1.72   #   31.06   1.37   #   31.06   0.38
3.35   0.81   2.14   2.14   0.93   1.62   0.27
0.25   0.05   0.08   0.08   0.08   0.05   0.01

Prediction after Action N
Prediction sum is  1.548850600260684
2.15   0.17   25.44   1.69   0.16   25.36   0.46
2.50   #   4.86   4.86   #   4.34   3.55
1.65   0.52   25.06   1.31   0.43   24.91   0.50
3.02   #   4.96   4.96   #   4.44   3.36
0.62   1.23   0.36   0.37   1.18   0.16   0.20
0.03   0.03   0.01   0.02   0.01   0.01   0.01
Location before move is  [2, 2]

Filtering after evidence  [0, 0, 0, 0]

Filtering sum  2.417628929663512
0.39   0.03   19.39   1.29   0.03   19.33   0.08
0.45   #   3.70   3.70   #   3.31   2.70
1.26   0.09   81.17   4.24   0.08   80.69   0.38
0.54   #   3.78   3.78   #   3.38   2.56
0.47   0.94   1.15   1.19   0.90   0.51   0.15
0.01   0.02   0.01   0.01   0.01   0.01   0.00

Prediction after Action W
Prediction sum is  4.23657431001689
0.42   15.83   3.37   16.04   16.50   2.35   15.75
0.52   #   15.98   3.52   #   14.81   2.69
1.18   65.96   4.21   65.75   67.95   1.04   65.08
0.61   #   14.28   3.57   #   12.88   2.76
1.18   1.40   2.09   2.02   1.46   1.18   0.67
0.07   0.11   0.14   0.13   0.10   0.06   0.02
Location before move is  [2, 1]

Filtering after evidence  [0, 1, 0, 1]

Filtering sum  10.476538567664988
0.11   94.48   3.77   17.95   98.50   2.63   4.15
0.01   #   0.79   0.17   #   0.73   0.13
0.06   393.73   0.88   13.80   405.64   0.22   3.21
```

```
0.01   #   0.71   0.18   #   0.64   0.14
0.06   1.56   0.44   0.42   1.63   0.25   0.03
0.02   0.12   0.16   0.15   0.12   0.07   0.01

Prediction after Action W
Prediction sum is  17.800533129761874
75.68   12.55   90.40   83.63   26.32   82.45   2.53
0.02   #   1.24   3.81   #   0.98   1.32
315.03   40.13   326.17   325.25   51.78   327.22   0.20
0.02   #   0.84   1.99   #   0.66   0.83
1.30   0.57   1.68   1.69   0.71   1.40   0.21
0.12   0.30   0.26   0.27   0.34   0.12   0.06
Location before move is  [2, 0]

Filtering after evidence  [1, 0, 0, 0]

Filtering sum  23.032237037152417
76.60   0.56   17.15   15.87   1.18   15.65   0.11
0.02   #   5.31   0.72   #   4.20   0.25
1355.08   1.79   263.06   262.32   2.31   263.91   0.04
0.02   #   3.60   0.38   #   2.86   0.16
5.59   0.11   1.35   1.36   0.14   1.13   0.04
0.12   0.06   0.05   0.05   0.06   0.02   0.00
```

Source Code (latest code in the submitted .py file):

```python
# -*- coding: utf-8 -*-
"""
Created on Fri Feb 26 12:37:01 2021

CIS479 - Program 2
@author: Nate Pierce UMID 94712233
"""

import copy
import math
import numpy as np

maze_data = [[0,0,0,0,0,0,0],
             [0,'#',0,0,'#',0,0],
             [0,0,0,0,0,0,0],
             [0,'#',0,0,'#',0,0],
             [0,0,0,0,0,0,0],
             [0,0,0,0,0,0,0]]
```

```
"""
0 is open space
# is an obstacle
"""

# Constants
ROWS = 6
COLS = 7
OBSTACLES = 4
OPEN_SPOTS = (ROWS*COLS)-OBSTACLES
LOC = [4,2] # location in reality. The robot is ignorant of this value.
Mutable.

  # Uses Bayes Rule to filter the evidence determined by evidence() and the
prior Sum of elements must be 1.
def filtering(maze, evi):
    prior_maze = copy.deepcopy(maze) # Priors are the element values in the
maze that is passed to the function
    evidence_maze = evidence(maze, evi) # update maze with P(Zw,n,e,s,t|St)
for each element

    y = [] # used to troubleshoot - check if sum of all elements is 1
    summation = 0

    # find the sum total of all probabilities. Used when finding the norm.
    for i in range(0, ROWS):
        for j in range(0, COLS):
            if maze[i][j] != '#':
                summation += evidence_maze[i][j]
    summation = summation/OPEN_SPOTS

    # Calculate probability.
    for i in range(0, ROWS):
        for j in range(0, COLS):
            num = 0
            norm = 0

            if maze[i][j] != '#':
                num = evidence_maze[i][j]*prior_maze[i][j]
                norm = num/summation
                maze[i][j] = norm
```

```
                y.append(norm)
    c = sum(y)
    print('Filtering sum ', c)
    return maze

# Takes the maze and evidence - checks if evidence matches reality and
assigns probability accordingly for each element in the maze, i.e.
# the function generates P(Zw,n,e,s,t|St) for every state (element ij in
matrix) in the maze.
def evidence(maze, evi):
    for k in range(0, ROWS):
        for m in range(0, COLS):
            if maze[k][m] != '#':
                look = checkSurrounding([k, m])
                prod = 0
                arr = []
                for n in range(0, len(look)):
                        if look[n] == evi[n] and look[n] == 0: # an open
spot is sensed and there is an open spot
                            t = 0.85
                        if look[n] == evi[n] and look[n] == 1: # a barrier
is sensed and there is a barrier
                            t = 0.8
                        if look[n] != evi[n] and look[n] == 0: # a barrier
is sensed and there is an open spot
                            t = 0.15
                        if look[n] != evi[n] and look[n] == 1: # an open
spot is sensed and there is a barrier
                            t = 0.2
                        arr.append(t)
                        n += 1
                prod = math.prod(arr)
                maze[k][m] = prod
            else: continue
    return maze

# The maze passed into this function contains the element values P(St+1|Zt)
determined during the filtering i.e. the transpition probabilities.
# Sum of elements must equal 1.
def prediction(maze, move):
    f = copy.deepcopy(maze) # Filtered maze. Elements are P(Si|Zi)
    x = move.index(1) # extract move command
```

```
    arr = []
    for i in range(0, ROWS):
        for j in range(0, COLS):
            if f[i][j] != '#':

                # Initialize transition probabilities - must be reset for
every element in maze!
                p = 0 # running total
                a = 0 # west
                b = 0 # north
                c = 0 # east
                d = 0 # south
                e = 0 # stationary probability

                if x == 0: # command to move west
                    if j - 1 > -1:
                        if f[i][j-1] != '#': a = f[i][j-1]*0.8
                        else: e += 0.8
                    else: e += 0.8
                    if i - 1 > -1:
                        if f[i-1][j] != '#': b = f[i-1][j]*0.1
                        else: e += 0.1
                    else: e += 0.1
                    if j + 1 < COLS:
                        if f[i][j+1] != '#': c = f[i][j+1]*0.8
                        else: e += 0.0
                    else: e += 0.0
                    if i + 1 < ROWS:
                        if f[i+1][j] != '#': d = f[i+1][j]*0.1
                        else: e += 0.0
                    else: e += 0.0
                    e = e*f[i][j]
                    p = a + b + c + d + e
                    maze[i][j] = p
                    arr.append(p)


                if x == 1: # command to move north
                    if j - 1 > -1:
                        if f[i][j-1] != '#': a = f[i][j-1]*0.1
                        else: e += 0.1
                    else: e += 0.1
```

```
                    if i - 1 > -1:
                        if f[i-1][j] != '#': b = f[i-1][j]*0.0
                        else: e += 0.8
                    else: e += 0.8
                    if j + 1 < COLS:
                        if f[i][j+1] != '#': c = f[i][j+1]*0.1
                        else: e += 0.1
                    else: e += 0.1
                    if i + 1 < ROWS:
                        if f[i+1][j] != '#': d = f[i+1][j]*0.8
                        else: e += 0.0
                    else: e += 0.0
                    e = e*f[i][j]
                    p = a + b + c + d + e
                    maze[i][j] = p
                    arr.append(p)
    t = sum(arr)
    print("Prediction sum is ", t)



# Given a maze state, generates a probability matrix depending on the move
command.
def transitional(maze, move):
    global LOC
    x = move.index(1)
    look = checkSurrounding(LOC)

    # initialize array of zeroes
    for i in maze:
        c = 0
        for j in i:
            if j == '#':
                c += 1
                continue
            else:
                i[c] = 0.0
                c += 1
    i = LOC[0]
    j = LOC[1]

    # Generate transition matrix depending on move command
```

```
    if x == 0: # command to move west
        if look[0] == 1: # barrier exists to the west
            t = [0.0, 0.1, 0.8, 0.1, 0.8] # [W, N, E, S, Stay at Current
Location]
            if look[1] == 0: maze[i-1][j] = t[1] # north
            if look[2] == 0: maze[i][j+1] = t[2] # east
            if look[3] == 0: maze[i+1][j] = t[3] # south
            maze[i][j] = t[4] # current location
        else:
            t = [0.8, 0.1, 0.8, 0.1, 0.0] # high probability of moving west
            if look[0] == 0: maze[i][j-1] = t[0] # west
            if look[1] == 0: maze[i-1][j] = t[1] # north
            if look[2] == 0: maze[i][j+1] = t[2] # east
            if look[3] == 0: maze[i+1][j] = t[3] # south
            maze[i][j] = t[4] # current location


    if x == 1: # command to move north
        if look[1] == 1: # barrier exists to the north
            t = [0.1, 0.0, 0.1, 0.8, 0.8] # [W, N, E, S, Stay at Current
Location]
            if look[0] == 0: maze[i][j-1] = t[0] # west
            if look[2] == 0: maze[i][j+1] = t[2] # east
            if look[3] == 0: maze[i+1][j] = t[3] # south
            maze[i][j] = t[4] # current location
        else:
            t = [0.1, 0.8, 0.1, 0.8, 0.0] # high probability of moving
north
            if look[0] == 0: maze[i][j-1] = t[0] # west
            if look[1] == 0: maze[i-1][j] = t[1] # north
            if look[2] == 0: maze[i][j+1] = t[2] # east
            if look[3] == 0: maze[i+1][j] = t[3] # south
            maze[i][j] = t[4] # current location

    return maze


# Generates an array of arrays with all possible combinations of no
barrier/barrier states surrounding a given space in the maze
def combinations():
    combs = []
    for W in range(0,2):
```

```
        for N in range(0,2):
            for E in range(0,2):
                for S in range(0,2):
                    combs.append((W,N,E,S))
    return combs

# Takes maze data and uses actual location to sense surroundings. Returns a
list of of integers 0 (did not see barrier)
# and 1 (did see barrier) e.g. [1,0,0,0] barrier detected to west [W, N, E,
S]
def checkSurrounding(loc):
    sense = []
    i = loc[0]
    j = loc[1]

    # look west for object
    if j - 1 == -1: # we're outside the maze
        blockWest = 1
        sense.append(blockWest)
    else: # we're within the maze
        if  maze_data[i][j-1] == '#':
            blockWest = 1
            sense.append(blockWest)
        else:
            blockWest = 0
            sense.append(blockWest)

    # look north for object
    if i - 1 == -1:  # we're outside the maze
        blockNorth = 1
        sense.append(blockNorth)
    else: # we're within the maze
        if  maze_data[i-1][j] == '#':
            blockNorth = 1
            sense.append(blockNorth)
        else:
            blockNorth = 0
            sense.append(blockNorth)

    # look east for object
    if j + 1 == COLS: # # we're outside the maze
        blockEast = 1
```

```
        sense.append(blockEast)
    else: # we're within the maze
        if  maze_data[i][j+1] == '#':
            blockEast = 1
            sense.append(blockEast)
        else:
            blockEast = 0
            sense.append(blockEast)

    # look south for object
    if i + 1 == ROWS:  # we're outside the maze
        blockSouth = 1
        sense.append(blockSouth)
    else: # we're within the maze
        if  maze_data[i+1][j] == '#':
            blockSouth = 1
            sense.append(blockSouth)
        else:
            blockSouth = 0
            sense.append(blockSouth)

    return sense

def move(move):
    global LOC
    x = move.index(1) #
    i = LOC[0]
    j = LOC[1]

    # movement command is west
    if x == 0:
        j = j - 1

    # movement command is north
    if x == 1:
        i = i - 1

    LOC = [i, j]


# Auxilliary function that finds the real sensory state for each element in
the 2D maze [W, N, E, S]
```

```
def findSpots():
    arr = []
    for i in range(0, ROWS):
        l = []
        for j in range(0, COLS):
            location = [i, j]
            t = checkSurrounding(location)
            l.append(t)
        arr.append(l)

    return arr

# Helper function that counts the number of spots in the maze with matching
probablities to prob passed. Returns the integer count.
def countProbs(maze, prob):
    count = 0
    prob = np.round(prob, 3)
    for i in range(0, ROWS):
        for j in range(0, COLS):
            if maze[i][j] != '#':
                elem = round(maze[i][j], 3)
                if elem == prob:
                    count +=1
    return count


# Returns a count of the number of states within the maze that matches the
passed state.
def countSpots(state):
    l = findSpots()
    count = 0
    for i in l:
        for j in i:
            if state == j:
                count += 1
    return count

# Helper function to convert a state in probability form to state in
barrier/no barrier form
def convertState(prob_state):
    for i in prob_state:
        if i == 0.85: i = 0
```

```
        if i == 0.8: i = 1
        if i == 0.2: i = 1
        if i == 0.15: i = 0
    return prob_state


def printState(state):
    for i in state:
        for j in i:
            if j == '#':
                print('#', end="    ")
            else:
                j = j*100
                print('{0:.2f}'.format(j), end="    ")
        print()

# Generate and return the initial state. The prior is contained within each
element of state.
def initial_state(maze):
    for i in range(0, ROWS):
        for j in range(0, COLS):
            if maze[i][j] == '#':
                continue
            else:
                maze[i][j] = 1/OPEN_SPOTS
    return maze

if __name__ == '__main__':
    maze = copy.deepcopy(maze_data) # maze_data should be immutable -
deepcopy allows different locations for maze and maze_data
    start_prior = 1/OPEN_SPOTS # initialize probability matrix with prior
    move_sequence = [[0,1,0,0], [0,1,0,0], [1,0,0,0], [1,0,0,0]]
    evidence_sequence = [[0,0,0,0], [1,0,0,0], [0,0,0,0], [0,1,0,1],
[1,0,0,0]]
    maze = initial_state(maze)


    print('\nInitial Location Probabilities')
    printState(maze)
    for i in range(0, len(evidence_sequence)):

        print('Location before move is ', LOC)
```

```
        # filtering
        print('\nFiltering after evidence ', evidence_sequence[i] )
        filtering(maze, evidence_sequence[i])
        printState(maze)

        # prediction
        if i < len(move_sequence):
            direction = move_sequence[i].index(1)
            if direction == 0:
                direction = 'W'
            else: direction = 'N'

            print('\nPrediction after Action', direction)
            prediction(maze, move_sequence[i])
            printState(maze)

            # move
            move(move_sequence[i])

    # last evidence
    print('\nFiltering after evidence [1,0,0,0]')
    printState(filtering(maze, [1,0,0,0]))
```