Programming Assignment 7: Hidden Markov Models

1. Group member

   a) Hao Yang, USCID: 5284267375. (Contributions: All)

2. Part 1: Implementation

   1. Programming language: Python 3.7

   2. Libraries used: numpy, math

   3. Algorithm: HMM Viterbi algorithm.

   4. Script description:

      1. Data processing: read txt files row by row, record content in row 3 to row 12 as grid map, and row 25 to row 35 as observation set.

```
grid = []
obs = []
with open("hmm-data.txt",'r') as file:
    file_lines = file.readlines()
    for line in file_lines[2:12]:
        grid.append(list(map(float, line.split(' '))))
    for line2 in file_lines[24:35]:
        row = line2.strip('\n').split(' ')
        if '' in row:
            row.remove('')
        row = list(map(float,row))
        obs.append(list(map(float, row)))
```

Get the list of grid map and observation set:

```
grid: [[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, ...]
  0: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
  1: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
  2: [1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0]
  3: [1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0]
  4: [1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0]
  5: [1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0]
  6: [1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0]
  7: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
  8: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
  9: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
  __len__: 10
```

```
obs: [[6.3, 5.9, 5.5, 6.7],
  00: [6.3, 5.9, 5.5, 6.7]
  01: [5.6, 7.2, 4.4, 6.8]
  02: [7.6, 9.4, 4.3, 5.4]
  03: [9.5, 10.0, 3.7, 6.6]
  04: [6.0, 10.7, 2.8, 5.8]
  05: [9.3, 10.2, 2.6, 5.4]
  06: [8.0, 13.1, 1.9, 9.4]
  07: [6.4, 8.2, 3.9, 8.8]
  08: [5.0, 10.3, 3.6, 7.2]
  09: [3.8, 9.8, 4.4, 8.8]
  10: [3.3, 7.6, 4.3, 8.5]
  __len__: 11
```

2. Get matrix A, B, P for building hidden Markov model:

   Run the defined function:

```
P,A,B =  gotABpai(grid,obs)
```

   Got size details about grid and length of observation set.

```
def gotABpai(grid,obs):
    size = len(grid)    # size of map grid : 10
    observeRange = 1.3 * ((size - 1) * (2**0.5))
    lenV = int(observeRange * 10)  # lenV = 165.
    size2 = len(obs[0])  # size of observe value
    lenQ = size ** 2   # N = 100
```

   Step 1: Got matrix P:

```
P = gotPai(grid,obs,lenQ,size2)
```

   For all points in grid map, having same possibility of being initial point if it's 1 in grid. This is a matrix with size of 1x100.

```
def gotPai(grid,obs,lenQ,size2):   # got pai
    P = np.zeros((1,lenQ))
    for i in range(lenQ):
        x = i//10
        y = i-x*10
        realDistance = distances((x,y))
        realRange = []
        for j in range(size2):
            realRange.append([math.ceil(7*realDistance[j])/10,math.floor(13*realDistance[j])/10])
        qqq = 0
        for j in range(size2):
            if obs[0][j]>=realRange[j][0] and obs[0][j]<=realRange[j][1]:
                qqq += 1
        if qqq == 4 and grid[x][y] == 1:
            P[0,i]+=1
    ppp = np.sum(P)
    for i in range(lenQ):
        if P[0,i] == 1:
            P[0,i] = 1/ppp
    return P
```

Step 2: Got matrix A:

```
A = gotA(grid,obs)
```

Because robot can only move from one point to its neighboring point randomly, so for each point (x, y), only (x±1, y) and (x, y±1) have same possibility in transmat. Others are all 0. Especially, points in grid map with value 0 have no possibility to move.

```
def gotA(grid,obs):   # got A
    size = len(grid) ** 2
    A = np.zeros((size,size))
    for i in range(10):
        for j in range(10):
            if grid[i][j] == 1:
                ppp = 0
                if i-1 >= 0 and grid[i-1][j]==1: # up
                    ppp+=1
                    A[i*10+j,i*10+j-10] = 1
                if i+1 <= 9 and grid[i+1][j]==1: # down
                    ppp+=1
                    A[i*10+j,i*10+j+10] = 1
                if j-1 >= 0 and grid[i][j-1]==1: # left
                    ppp+=1
                    A[i*10+j,i*10+j-1] = 1
                if j+1 <= 9 and grid[i][j+1]==1: # down
                    ppp+=1
                    A[i*10+j,i*10+j+1] = 1
                A[i*10+j,:] = A[i*10+j]/ppp
    return A
```

A is a matrix with size of 100 x 100.

```
A: array([[0.        , 0.5
  [0:100] : [array([0. , 0.
  dtype: dtype('float64')
  max: 0.5
  min: 0.0
  shape: (100, 100)
  size: 10000
```

Step 3: Got matrix B:

```
B = gotB(grid,obs,lenV)
```

Because the longest distance in grid map with size 10x10 is $9\sqrt{2}$. So the

possible value range of observation set is [0, 16.5] with one decimal place. And because in this assignment we record all distances to 4 towers, so it's a 4-dimensional distance set. Therefore, it has totally 165^4 possible value, and B will be a matrix with size of 100 x 165^4. It needs 56.5G memory to load it. It's impossible for me. However, because we know that the possibility to observe a certain value is same for all value in a range. So we only need to know how many possible observation value it can get, call it 'pro', and determine if one position can have this observation value. If it can, the possibility to get that is 1/n, if not, it's zero.

```
def gotB(grid,obs,lenV):     # got B
    row = len(grid)**2
    B = np.zeros((row,1))
    for i in range(row):
        x = i//10
        y = i-x*10
        realDistance = distances((x,y))
        realRange = []
        pro = 1
        for j in range(4):
            realRange.append([math.ceil(7*realDistance[j])/10,math.floor(13*realDistance[j])/10])
            pro *= (realRange[j][1]-realRange[j][0]) * 10 + 1
        B[i,0] = 1/round(pro)
    return B
```

Function of determining if a position can get a certain observation value:

```
def ifIn(distanceList,i): # distanceList-the observation value, i- the index of point:0-99
    x = i//10
    y = i-x*10
    dist = distances((x,y))
    realRange = []
    for j in range(4):
        realRange.append([math.ceil(7*dist[j])/10,math.floor(13*dist[j])/10])
    for k in range(len(distanceList)):
        if distanceList[k]<realRange[k][0] or distanceList[k]>realRange[k][1]:
            return False
    return True
```

B is a matrix with size of 100 x 1:

```
B: array([[4.34971727e-06],
 [0:100] : [array([4.34971727e-06]),
 dtype: dtype('float64')
  max: 4.349717268377555e-06
  min: 3.525471531817381e-07
 shape: (100, 1)
```

So far, we already got necessary matrix A, B, P to work HMM.

Step 5: implement Viterbi algorithm:

```
trajectory = HMM(P,A,B,grid,obs)
```

Initialize the matrix of recording hidden state:

d is for record:

$$\delta_t(i) = \max_{i_1,i_2,\ldots i_{t-1}} P(i_t = i, i_1, i_2, \ldots i_{t-1}, o_t, o_{t-1}, \ldots o_1|\lambda),\ i = 1, 2, \ldots N$$

$$\begin{aligned}\delta_{t+1}(i) &= \max_{i_1,i_2,\ldots i_t} P(i_{t+1} = i, i_1, i_2, \ldots i_t, o_{t+1}, o_t, \ldots o_1|\lambda) \\ &= \max_{1\le j\le N} [\delta_t(j)a_{ji}]b_i(o_{t+1})\end{aligned}$$

v is for record:

$$\Psi_t(i) = arg \max_{1\le j\le N} [\delta_{t-1}(j)a_{ji}]$$

For each observation value, calculate the most possibility to get to certain point with the result get from observation value set. Record index in v.

```
def HMM(P,A,B,grid,obs):
    Step = len(obs) # len of observations, 11
    d = np.zeros((1,100))
    v = np.zeros((1,100))
    for i in range(100):
        if grid[i//10][i-(i//10)*10] == 1 and ifIn(obs[0],i):
            d[0,i] = (1/87) * B[i,0]
    for loop in range(1,Step):   # 11 time-step
        d_next = np.zeros((1,100))
        v2 = np.zeros((1,100))
        for i in range(100):
            record = np.multiply(d[loop-1,:],A[:,i].T)
            if ifIn(obs[loop],i):
                d_next[0,i] = record.max() * B[i,0]
                if record.max() != 0:
                    v2[0,i] = record.argmax()
        d = np.vstack((d,d_next))
        v = np.vstack((v,v2))
```

After recording all d and v, track back the trajectory through matrix v, then

return the trajectory:

```
trajectory = []
trajectory.append(d[-1].argmax())
for epoch in range(Step-1,0,-1):
    trajectory.append(int(v[epoch,trajectory[10-epoch]]))
for i in range(len(trajectory)):
    x = trajectory[i]//10
    y = trajectory[i]- 10*x
    trajectory[i] = (x,y)
return trajectory
```

5. Optimizations
    1. For implementing HMM, need to compute matrix A, B, P first, but because the moving of robot has many restriction in this assignment, these three matrix have lots of value zero, I only record value and location of value in matrix to save many memories.
6. Challenges faced
    1. Because the longest distance in grid map with size 10x10 is $9\sqrt{2}$. So the possible value range of observation set is [0, 16.5] with one decimal place. And because in this assignment we record all distances to 4 towers, so it's a 4-dimensional distance set. Therefore, it has totally 165^4 possible value, and B will be a matrix with size of 100 x 165^4. It needs 56.5G memory to load it. It's impossible for me. However, because we know that the possibility to observe a certain value is same for all value in a range. So we only need to know how many possible observation value it can get, call it 'pro', and determine if one position can have this observation value. If it can, the possibility to get that is 1/n, if not, it's zero.

7. Result

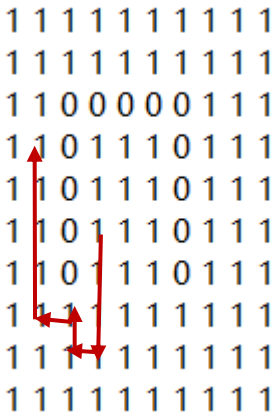
    1. Run code below:

```
print("The most likely trajectory of robot:")
for ele in trajectory[::-1]:
    print(ele)
```

Get the result of trajectory:

```
The most likely trajectory of robot:
(5, 3)
(6, 3)
(7, 3)
(8, 3)
(8, 2)
(7, 2)
(7, 1)
(6, 1)
(5, 1)
(4, 1)
(3, 1)
```

So the trajectory looks like this:



3. Part 2: Software familiarization

    1. Library function: hmmlearn.hmm

    2. How to use:

        1. Prepared matrix A, B, P for implementation.(I already have it in my implementation).

2. Initialize the model:

```
import numpy as np
from hmmlearn import hmm
```

```
model = hmm.MultinomialHMM(n_components=n_states)
model.startprob_= p
model.transmat_= a
model.emissionprob_= b
```

3. Choose algorithm and implement directly to get result:

```
logprob, h = model.decode(o, algorithm="viterbi")
```

3. Problem:

In this assignment, the possible observation value is an interval based on the real state value, so in real situation, there are infinite possible value can be observed from same state value, even though in this assignment is finite because it limited with one decimal place in the interval. However, it still has 165 possible value for one distance value, and because it records distances for four towers, so totally have 165^4 possible observation value. In this assignment the length of state set is 100 different location. So the emission projection matrix B should be with size of 100 x 165^4, it needs 56.5Gb memory to record. But the hmm library set to be used only with all three matrix, so I can't implement this assignment with this library even though I got all information. But I learned how to use this library.

4. Part 3: Applications

HMM is a popular algorithm about solving problems of time series or state

series. It’s broadly used in situations of states prediction and possibility computation. For example, weather prediction programs will use it to do prediction with air humidity as the observation set and future weather as state set to predict short term weather based on air humidity right now.