INF 552 MACHINE LEARNING FOR DATA SCIENCE

HOMEWORK 7

Team member:

SANJAY MALLASAMUDRAM SANTHANAM – 3124715393

Part 1:

Language used:

Python 3, JUPYTER NOTEBOOK

Data structures used:

Numpy array to store data

Code-level optimization:

Since the observations values are continuous in the steps of 0.1, it is not possible to store the probability values for all observations. Rather it is calculated on run time when needed. This saves computation time as well as storage space.

Challenges:

Calculating emission matrix was difficult. There are various possible interpretations for the problem and it is easy to be misguided away from the problem. One particular challenge was calculating emission probabilities. Understanding how it was uniform distribution was confusing. I initially thought it to be Gaussian and on further keen reading, I realized it is uniformly distributed with distance values incrementing in 0.1. Also dealing with hidden library was a problem as enumerating the full emission matrix resulted in memory overflow error and all the existing libraries required a matrix input, and there was no way to use a function to calculate emission probabilities.

Part 2:

Library used: hidden_markov

Although the implementation is good , for this homework, it was not suited. It could take only a simple emission matrix and transmission matrix as input. Also It required observations to be of string data type, whereas in this problem it is numeric. Although we can enumerate all distance values and convert it to character data type, the resulting matrix size was huge and it couldn't fit into memory. Thus there is no pre defined library that can be used to solve the problem. On the other hand, this library works well for simple discrete observations ,not for continuous emissions . There are other HMM implementations even for continuous observations (ex. Scikit learn has Gaussian HMM for Gaussian emissions) but it is only limited to a few distributions and not for all

Part 3:

HMM is used to identify human gaits using camera images.The structural component of a human can be identified using HMM.

HMM is also used to analyse biological sequences. It is used to solve various sequence analysis problems like similarity search, , gene annotation, pairwise& multiple sequence alignments and classification.

In [1]:
```
#Done by SANJAY MALLASAMUDRAM SANTHANAM ; USC ID:3124715393
import pandas as pd
import numpy as np
import math
from operator import truediv,add
import copy
import os
from PIL import Image
np.set_printoptions(precision=100)
```

In [2]:
```
#read file
f=open('C:/Users/Lenovo/Desktop/hmm-data.txt')
```

In [3]:
```
#read file line by line
data=f.readlines()
print(data)
```

```
['Grid-World:\n', '\n', '1 1 1 1 1 1 1 1 1 1\n', '1 1 1 1 1 1 1 1 1 1\n', '1 1 0 0 0 0 0 1 1 1\n',
'1 1 0 1 1 1 0 1 1 1\n', '1 1 0 1 1 1 0 1 1 1\n', '1 1 0 1 1 1 0 1 1 1\n', '1 1 0 1 1 1 0 1 1 1
\n', '1 1 1 1 1 1 1 1 1 1\n', '1 1 1 1 1 1 1 1 1 1\n', '1 1 1 1 1 1 1 1 1 1\n', '\n', '\n', 'Tower
Locations:\n', '\n', 'Tower 1: 0 0\n', 'Tower 2: 0 9\n', 'Tower 3: 9 0\n', 'Tower 4: 9 9\n', '\n',
'\n', 'Noisy Distances to Towers 1, 2, 3 and 4 Respectively for 11 Time-Steps:\n', '\n', '6.3 5.9
5.5 6.7\n', '5.6 7.2  4.4 6.8\n', '7.6 9.4  4.3 5.4\n', '9.5 10.0 3.7 6.6\n', '6.0 10.7 2.8 5.8
\n', '9.3 10.2 2.6 5.4\n', '8.0 13.1 1.9 9.4\n', '6.4 8.2  3.9 8.8\n', '5.0 10.3 3.6 7.2\n', '3.8
9.8  4.4 8.8\n', '3.3 7.6  4.3 8.5']
```

In [4]:
```
#save world as grid 2d array
grid=[]
for i in range(0,10):
    #strip line of white spaces at the end and split string
    temp=data[2+i].strip().split(' ')
    #convert string to int
    grid.append(list(map(int,temp)))
#convert to numpy array
grid=np.asarray(grid)
print(grid)
```

```
[[1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1]
 [1 1 0 0 0 0 0 1 1 1]
 [1 1 0 1 1 1 0 1 1 1]
 [1 1 0 1 1 1 0 1 1 1]
 [1 1 0 1 1 1 0 1 1 1]
 [1 1 0 1 1 1 0 1 1 1]
 [1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1]]
```

In [5]:
```python
#used to store tower locations
tower=[]
for i in range(4):
    #strip spaces at the end and split string
    temp=data[16+i].strip().split()
    tower.append([int(temp[-2]),int(temp[-1])])
#number of towers
n_t=len(tower)
#convert to numpy array
tower=np.asarray(tower)
print(tower)
```

```
[[0 0]
 [0 9]
 [9 0]
 [9 9]]
```

In [6]:
```python
#noisy distance observations of 4 towers at 11 timesteps
dist=[]
ts=11
for i in range(ts):
    #strip spaces at the end and split string
    dist.append(list(map(float,data[24+i].strip().split())))
#convert to numpy array
dist=np.asarray(dist)
print(dist)
#number of timesteps
time_=len(dist)
```

```
[[ 6.3  5.9  5.5  6.7]
 [ 5.6  7.2  4.4  6.8]
 [ 7.6  9.4  4.3  5.4]
 [ 9.5 10.   3.7  6.6]
 [ 6.  10.7  2.8  5.8]
 [ 9.3 10.2  2.6  5.4]
 [ 8.  13.1  1.9  9.4]
 [ 6.4  8.2  3.9  8.8]
 [ 5.  10.3  3.6  7.2]
 [ 3.8  9.8  4.4  8.8]
 [ 3.3  7.6  4.3  8.5]]
```

In [7]:
```python
#function that returns neighbour of a given cell
def neighbour(x,y):
    neig=[]
    #check left and right cells if they are valid and are free
    for i in range(x-1,x+2,2):
        j=y
        if(i>=0 and i<10 and j>=0 and j<10 and grid[i][j]==1):
            neig.append([i,j])
    #check top and bottom cells if they are valid and are free
    for j in range(y-1,y+2,2):
        i=x
        if(i>=0 and i<10 and j>=0 and j<10 and grid[i][j]==1):
            neig.append([i,j])
    return neig
```

In [8]:
```python
neighbour(3,5)
```

Out[8]:
```
[[4, 5], [3, 4]]
```

In [9]:
```python
#returns euclедian distance between 2 points
def euc(x1,y1,x2,y2):
    return np.sqrt(np.power((x1-x2),2)+np.power((y1-y2),2))
```

In [10]:
```
#dimension of square grid
l=len(grid)
#tower coordinates
tower = [[0,0],[0,9],[9,0],[9,9]]
#returns emission probabilities of all grids for a given timestamp observation
def calc_emission(obs):
    emission=np.ones((l,l))
    #iterate all grid cells
    for i in range(l):
        for j in range(l):
            #for each grid cell, check if observed noisy dist is in range with actual distance.

            #if cell is an obstacle cell, then assign 0 as observation couldnt have happened here.
            if(grid[i][j]==0):
                emission[i][j]=0
            else:
                for t in range(n_t):
                    #original distance
                    d=euc(tower[t][0],tower[t][1],i,j)
                    #print(d,0.7*obs[t],1.3*obs[t])
                    #if observed noisy dist to tower t is within acceptable range, then its valid
 observation
                    #It is possible that the robot could have made the observation from this cell.
So multiply the
                    #uniform distribution probability
                    if(0.7*d<=np.round(obs[t],1)<=1.3*d):
                        #uniform distribution pdf= 1/(b-a).
                        emission[i][j]*=1/((0.6*d)*10+1)
                    #if observed distance to tower t in outside acceptable range, then its invalid
observation.
                    #so the robot cannot have made the observation in the grid. so assign 0
                    else:
                        emission[i][j]=0

    return emission
```

In [11]:
```
"""
import sys
for k in range(11):
    print("Time:",k)
    for i in range(10):
        for j in range(10):
            em=calc_emission(dist[k])
            sys.stdout.write(str(int(em[i][j] and grid[i][j]))+' ')
        print("\n")
"""
```

Out[11]:
```
'\nimport sys\nfor k in range(11):\n    print("Time:",k)\n    for i in range(10):\n        for j i
n range(10):\n            em=calc_emission(dist[k])\n            sys.stdout.write(str(int(em[i][j]
and grid[i][j]))+\' \')\n        print("\n")\n'
```

In [12]:
```
#T1 stores probability values of robot being in cell [i,j] at timestep t as T1[i,j,t]
T1=np.zeros((l,l,time_),dtype='float64')
#T2 stores the most probable cell from which transition might have  from previous time step given
 observations from time 1 to t-1
#i.e. stores most probable previous cell
T2=np.full((l,l,time_),-1)
```

In [13]:
```
#number of free cells
n_free=np.count_nonzero(grid)
```

In [14]:
```
#calculate emission matrix for 1st timestep
emission_0=calc_emission(dist[0])
for i in range(l):
    for j in range(l):
        #initial probability for all cells is assumed to be constant 1/number of free cells.
        T1[i,j,0]=float(1/n_free)*emission_0[i][j]
```

In [ ]:

In [15]:
```
#iterate forward through each time step
for t in range(1,time_):
    #calculate emission probability
    emission=calc_emission(dist[t])
    #iterate all cells
    for i in range(l):
        for j in range(l):
            #get all neighbours of cell
            neig=neighbour(i,j)
            #for each neighbour
            for n in neig:
                #find max of (prob of robot being in neighbour cell at t-1*prob of transition from
neighbour to current cell*
                #emission prob of current observation at time t in current cell)
                val=T1[n[0]][n[1]][t-1]*1/float(len(neighbour(n[0],n[1])))*emission[i][j]
                if (T1[i,j,t]<val):
                    T1[i,j,t]=val
                    #store 2d cell coordinates(x,y) of predecessor as an integer 10*x+y
                    T2[i,j,t]=n[0]*10+n[1]
```

In [16]:
```
#z stores most probable cells at all timesteps
z=np.full((time_,2),-1)
#find most probable final cell after tinal timestep
z[time_-1,:]=np.unravel_index(T1[:,:,time_-1].argmax(), T1[:,:,time_-1].shape)
```

In [17]:
```
#iterate backwards finding which state lead to current state using store values
for i in range(time_-1,0,-1):
    #reverse map stored integer predecessor to 2d cell coordinates
    z[i-1,:]=[T2[z[i][0],z[i][1],i]/10,T2[z[i][0],z[i][1],i]%10]
```

In [18]:
```
#most probable path
print("Most probable path of robot",z)
```

```
Most probable path of robot [[5 3]
 [6 3]
 [7 3]
 [8 3]
 [8 2]
 [7 2]
 [7 1]
 [6 1]
 [5 1]
 [4 1]
 [3 1]]
```

In [19]:
```
import numpy as np
from hidden_markov import hmm

# States
states = ('Classroom', 'Starbucks')

# list of possible observations
possible_observation = ('Blackboard','Coffee' )

# The observations that we observe and feed to the model
obs1 = ('Coffee', 'Blackboard','Blackboard','Coffee')
obs2 = ('Blackboard', 'Coffee','Coffee')


# Number of observation sequece 1 and 2
quantities_observations = [1,2]

observation_tuple = []
observation_tuple.extend( [obs1,obs2] )

# Input parameters as Numpy matrices
start_probability = np.matrix( '0.2 0.8 ')
transition_probability = np.matrix('0.9 0.1 ;  0.3 0.7 ')
emission_probability = np.matrix( '0.2 0.8 ; 0.6 0.4 ' )
```

In [20]:
```
#Hidden markov model
test = hmm(states,possible_observation,start_probability,transition_probability,emission_probabili
ty)
```

In [21]:
```
#Output of the Viterbi algorithm
test.viterbi(obs1)
```

Out[21]:
```
['Starbucks', 'Starbucks', 'Starbucks', 'Starbucks']
```

In [ ]:
```

```