EN 685.621 HW3 - Noboru Hayashi

Q1:

```
import numpy as np
import pandas as pd
# init parameter
def init(data, k):
    (n, d) = data.shape
    mu_col = data.sum().to_numpy()/n
    sigma_col = data.std().to_numpy()

    m = np.dot(mu_col.reshape((d,1)), np.ones([1, k])) +
np.dot(sigma_col.reshape((d,1)), np.random.rand(1,k))
    sigma = sigma_col.mean() * np.ones([1,k])[0]
    p = np.ones([1,k])/k

    return m, sigma, p[0]
# e step
def e_step(data, m, sigma, p, k, n):
    ret = np.zeros((k,n))

    for i in range(k):
        for j in range(n):
            c = 1/((np.sqrt(2*np.pi)*sigma[i])**2)
            diff = (data.to_numpy()[j] - m[:, i])**2
            e = np.exp(-0.5* diff.sum() / sigma[i]**2)
            ret[i, j] = p[i] * c * e
            denom = ret.sum(axis=0)

    return ret/denom
# m step
def m_step(data, p, k, d):
    mu = np.zeros((d, k))
    for i in range(k):
        for j in range(d):
            mu[j,i] = np.dot(data.to_numpy()[:,j],p[i,:].transpose()) / p[i].sum()

    sigma = np.zeros((1, k))
    for i in range(k):
        nume = 0
        dist = (data.to_numpy() - mu[:, i])**2
        for j in range(len(dist)):
            nume += dist[j].sum() * p[i,j]
```

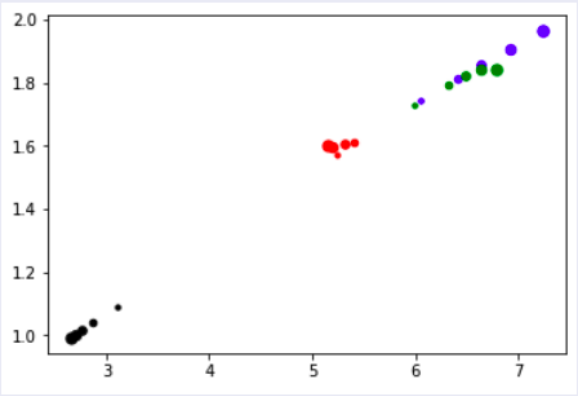
```
            sigma[0,i] = np.sqrt( nume / (p[i].sum() * d ))

    new_p = 1/len(data.to_numpy()) * p.sum(axis=1)
    return mu, sigma, new_p

# use init, e_step, m_step to operate EM algorithm
def em(data, n, k):
    mu, sigma, p = init(data, k)

    for i in range(n):
        p_cond = e_step(data, m, sigma, p, k, len(data.to_numpy()))
        mu, sigma, p = m_step(data, p_cond, k, len(data.to_numpy()[0]))
        # print out mixed mean for every iteration
        print(m)
```

With using the third & fourth features of additional observations from HW2:

```
def em(data, n, k):

    mu, sigma, p = init(data, k)
    num_obs = len(data)
    dim = len(data.columns)

    colors = ['r', 'g', 'b', 'k', 'y']

    for i in range(n):
        p_cond = e_step(data, mu, sigma, p, k, num_obs)
        mu, sigma, p = m_step(data, p_cond, k, dim)
        for j in range(k):
            plt.scatter(mu[0][j], mu[1][j], s=10*i+10, c=colors[j])

    return mu
```

```
em(data, 5, 4)
```

```
array([[5.1550894 , 6.78916637, 7.23674965, 2.6618086 ],
       [1.5997486 , 1.84420641, 1.9633917 , 0.99235302]])
```



Q2:

Agent: An entity that perceives and acts based on a sensed environment

Agent function: A function for an agent that maps from percept histories to actions

Agent program: A program runs on the physical architecture or hardware to produce an agent function

Artificial intelligence: A field of study concerned with designing and building intelligent entities possible to perceive, reason, and act

Autonomy: A property of agents that learn what it can to compensate for partial or incorrect prior knowledge

Goal-based agent: An agent that acts in order to achieve its designated goals

Intelligence: A level of ability for entities or computations to perceive, reason and act.

Learning agent: An agent that learns and improves its performance based on its experience over time

Logical reasoning: Methods of reasoning to have conclusions and corresponding actions when preconditions or environments is perceived

Model-based agent: An agent updates its internal model of current world state over time and acts according to this updated internal model.

Rationality: A rational agent chooses whichever action maximizes the expected value of the performance measure given the percept sequence to date

Reflex agent: An agent that acts solely on its current percept

Utility-based agent: An agent that acts in order to maximize the expected utility of the new state after its action

Q3:
chekcWinPos() function for an agent to act goal-based, which let the agent to either take a winning move or blocking opponent's wins:

```
#Check possibilities for wins in the next move
def checkWinPos(place):
    # 0-1-2
    if place[0] == place[1] and place[0] != 0 and place[2] == 0: return 2
    if place[0] == place[2] and place[0] != 0 and place[1] == 0: return 1
    if place[1] == place[2] and place[1] != 0 and place[0] == 0: return 0
    # 0-3-6
    if place[0] == place[3] and place[0] != 0 and place[6] == 0: return 6
    if place[0] == place[6] and place[0] != 0 and place[3] == 0: return 3
    if place[3] == place[6] and place[3] != 0 and place[0] == 0: return 0
    # 0-4-9
    if place[0] == place[4] and place[0] != 0 and place[8] == 0: return 8
    if place[0] == place[8] and place[0] != 0 and place[4] == 0: return 4
    if place[4] == place[8] and place[4] != 0 and place[0] == 0: return 0
    # 3-4-5
    if place[3] == place[4] and place[3] != 0 and place[5] == 0: return 5
    if place[3] == place[5] and place[3] != 0 and place[4] == 0: return 4
    if place[4] == place[5] and place[4] != 0 and place[3] == 0: return 3
    # 6-7-8
    if place[6] == place[7] and place[6] != 0 and place[8] == 0: return 8
    if place[6] == place[8] and place[6] != 0 and place[7] == 0: return 7
    if place[7] == place[8] and place[7] != 0 and place[6] == 0: return 6
    # 1-4-7
    if place[1] == place[4] and place[1] != 0 and place[7] == 0: return 7
    if place[1] == place[7] and place[1] != 0 and place[4] == 0: return 4
    if place[4] == place[7] and place[4] != 0 and place[1] == 0: return 1
    # 2-5-8
    if place[2] == place[5] and place[2] != 0 and place[8] == 0: return 8
    if place[2] == place[8] and place[2] != 0 and place[5] == 0: return 5
    if place[5] == place[8] and place[5] != 0 and place[2] == 0: return 2
    return None
```

If there's no possibility for wins, the agent will take the open spot with the priority of the center, top-left, bot-left, top-right, bot-right, mid-left, top-mid, bot-mid, mid-right:

```
        F = checkWinPos(place)
        if F != None:
            if F == 0:
                topLeftPress()
            if F == 1:
                midLeftPress()
            if F == 2:
                botLeftPress()
            if F == 3:
                topMidPress()
            if F == 4:
                midMidPress()
            if F == 5:
                botMidPress()
            if F == 6:
                topRightPress()
            if F == 7:
                midRightPress()
            if F == 8:
                botRightPress()
        elif place[4] == 0:
            midMidPress()
        elif place[0] == 0:
            topLeftPress()
        elif place[2] == 0:
            botLeftPress()
        elif place[6] == 0:
            topRightPress()
        elif place[8] == 0:
            botRightPress()
        elif place[1] == 0:
            midLeftPress()
        elif place[3] == 0:
            topMidPress()
        elif place[5] == 0:
            botMidPress()
        elif place[7] == 0:
            midRightPress()
```

Since checkWinPos() function perceives the state of the TicTacToe board, where scanning through 9 spots occurs, the running time for checking the winning position will be O(1). In

addition, an if-else statement for picking the next move also costs constant time. So total running time for an agent to make a move is O(1)

Q4:

Read and Split data function:

```
        def read_and_split(file_name, n, D, classes=[]):
            df = pd.read_csv(file_name)
            train = df.iloc[0:0]
            test = df.iloc[0:0]
            for cls in classes:
                cls_obs = df[df.iloc[:, D]==cls]
                cls_train = cls_obs.sample(frac=0.8, random_state=0)
                cls_test = cls_obs.drop(cls_train.index)
                train = train.append(cls_train, ignore_index=True)
                test = test.append(cls_test, ignore_index=True)

            return train, test
```

Gaussian Kernel

```
def gaussian_kernel(X1, X2, spread):
    row1, col1 = X1.shape
    row2, col2 = X2.shape
    N = col2;
    D = row2;
    K = np.zeros([col1, col2]);
    for i in range(col1):
        for j in range(col2):
            K[i,j] = (1/N)*(1/((np.sqrt(2*np.pi)*spread))**D) * np.exp(-0.5*
np.dot((X1[i,:]- X2[j,:]).transpose(), (X1[i,:]-X2[j,:])) / (spread**2))

    return K
```