HW4 — ISYE6416

Ashish Dhiman — ashish.dhiman9@gatech.edu

May 7, 2023

Contents

1 Question 1: M-step of GMM

We have to maximize

$$Q(\theta|\theta_k) = \sum_{i=1}^{n}\sum_{c=1}^{C} p_{i,c} \log \pi_c + \sum_{i=1}^{n}\sum_{c=1}^{C} p_{i,c} \log \phi(x_i|\mu_c, \Sigma_c)$$

wrt $\theta^{k+1} = \{\pi_c, \mu_c, \Sigma_c\}$, subject to $\sum \pi_c = 1$

Hence we can use the Lagrange approach, with the Lagrange function given as:

$$L(\lambda) = \sum_{i=1}^{n}\sum_{c=1}^{C} p_{i,c} \log \pi_c + \sum_{i=1}^{n}\sum_{c=1}^{C} p_{i,c} \log \phi(x_i|\mu_c, \Sigma_c) - \lambda(\sum_{c=1}^{C} \pi_c - 1)$$

First order optimality condition:

$$\begin{aligned} \nabla_\lambda Q &= 0 \\ \nabla_{\pi_c} Q &= 0 \\ \nabla_{\mu_c} Q &= 0 \\ \nabla_{\sigma_c} Q &= 0 \end{aligned} \tag{1}$$

With

$$\nabla_\lambda Q = \sum_{c=1}^{C} \pi_c = 1$$

With

$$\begin{aligned} \nabla_{\pi_c} Q = \nabla_{\pi_c}(\sum_{i=1}^{n}\sum_{c=1}^{C} p_{i,c} \log \pi_c - \lambda(\sum_{c=1}^{C} \pi_c - 1)) \\ = \frac{\sum_{i=1}^{n} p_{i,c}}{\pi_c} - \lambda = 1 \implies \pi_c = \frac{\sum_{i=1}^{n} p_{i,c}}{\lambda} \\ \text{But from constraint } \sum_{c=1}^{C} \pi_c = 1 \\ \implies \sum_{c=1}^{C} \pi_c = \sum_{c=1}^{C} \frac{\sum_{i=1}^{n} p_{i,c}}{\lambda} = 1 \\ \lambda = \sum_{c=1}^{C}\sum_{i=1}^{n} p_{i,c} = \sum_{i=1}^{n}\sum_{c=1}^{C} p_{i,c} = n \\ \implies \pi_c = \frac{\sum_{i=1}^{n} p_{i,c}}{n} \end{aligned} \tag{2}$$

Similarly for μ_c

$$\nabla_{\mu_c} Q = 0$$

$$\log \phi(x_i|\mu_c, \Sigma_c) \propto -\frac{1}{2} log(det(\Sigma_c)) - \frac{1}{2}((x_i - \mu)^T \Sigma_c^{-1}(x_i - \mu_c))$$

$$\propto -log(det(\Sigma_c)) - ((x_i - \mu)^T \Sigma^{-1}(x_i - \mu_c))$$

$$\nabla_{\mu_c} Q = -\nabla_{\mu_c} \big(\sum_{i=1}^{n} \sum_{c=1}^{C} p_{i,c}(log(det(\Sigma_c)) + ((x_i - \mu)^T \Sigma^{-1}(x_i - \mu_c))) \big)$$

$$= \sum_{i=1}^{n} (p_{i,c} * (2\Sigma_c^{-1}(x_i - \mu_c))) = 0 \tag{3}$$

$$= \sum_{i=1}^{n} (p_{i,c} * (x_i - \mu_c)) = 0$$

$$\implies \sum_{i=1}^{n} p_{i,c} * x_i = (\sum_{i=1}^{n} p_{i,c}) \mu_c$$

$$\implies \mu_c = \frac{\sum_{i=1}^{n} p_{i,c} * x_i}{\sum_{i=1}^{n} p_{i,c}}$$

Similarly for Σ_c

$$\nabla_{\Sigma_c^{-1}} Q = 0$$

$$\nabla_{\Sigma_c^{-1}} Q = -\nabla_{\Sigma_c^{-1}} \big(\sum_{i=1}^{n} \sum_{c=1}^{C} p_{i,c}(log(det(\Sigma_c)) + ((x_i - \mu)^T \Sigma_c^{-1}(x_i - \mu_c))) \big)$$

$$= \sum_{i=1}^{n} (p_{i,c} * \big(\Sigma_c^T + ((x_i - \mu_c)(x_i - \mu_c)^T)^T \big)) = 0 \tag{4}$$

$$\sum_{i=1}^{n} (p_{i,c}) \Sigma_c^T = \sum_{i=1}^{n} (p_{i,c} * (x_i - \mu_c)(x_i - \mu_c)^T)^T$$

$$\implies \Sigma_c = \frac{\sum_{i=1}^{n} (p_{i,c} * (x_i - \mu_c)(x_i - \mu_c)^T}{\sum_{i=1}^{n} p_{i,c}}$$

2 Question 2: GMM on MNIST data

Initialisation
The given pictures below present the initialisation $\mu_0, \mu_1, \Sigma_0, \Sigma_1$

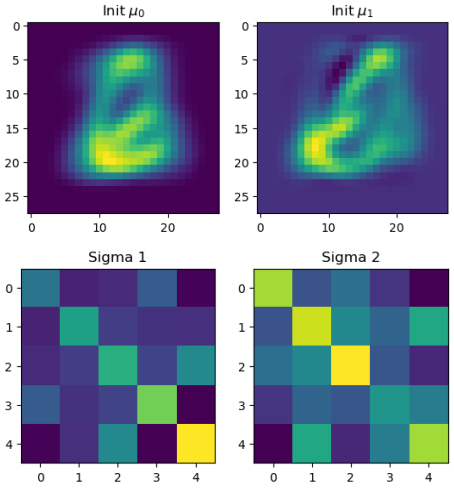


Figure 1: Initialised GMM parameters

2.1 Convergence of EM

The convergence of likelihood after running EM algorithm is given below:

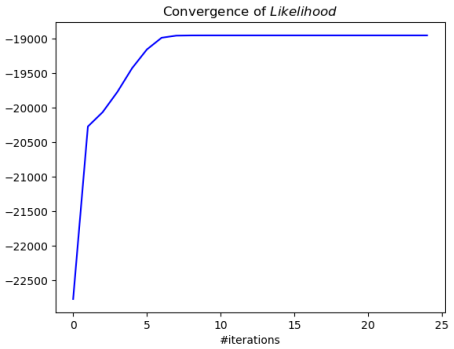


Figure 2: Initialised GMM parameters

In conjunction to likelihood, the μ_0, μ_1 also show conevergence

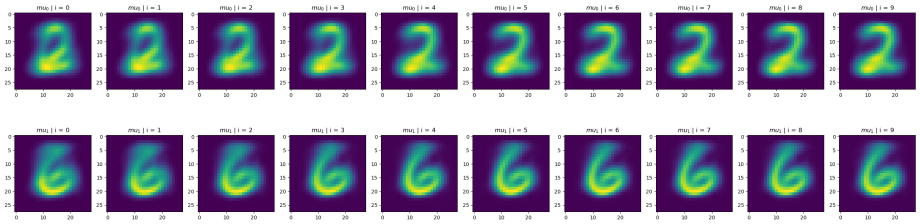


Figure 3: Initialised GMM parameters

2.2 Fitted GMM

For the fitted GMM, the model parameters are given below:

Fitted π

$$\pi = \begin{bmatrix} 0.49313392 \\ 0.50686608 \end{bmatrix}$$

Fitted μ

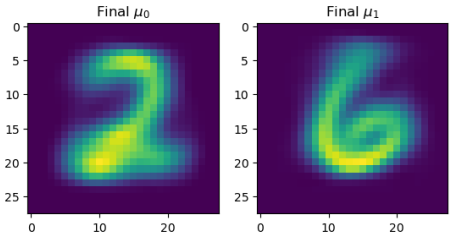


Figure 4: Fitted μ

We can see that the cluster centroids above correspond to average images in the cluster.

Fitted Σ

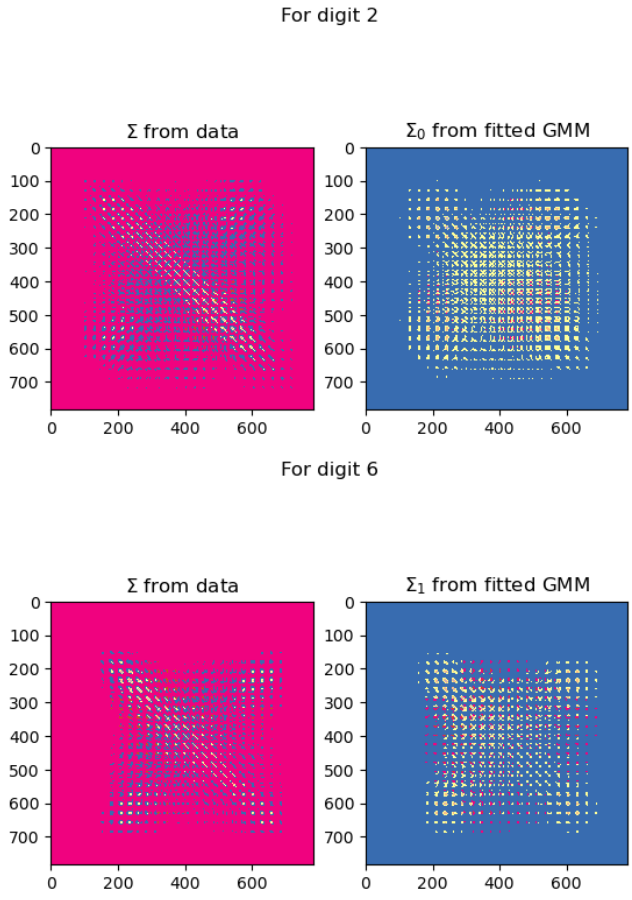


Figure 5: Sigma from data vs Sigma from GMM

We can see that the Sigmas from fitted GMM also closely resemble, the Sigmas from data. This resemblance is further apparent at the pca reduced level.

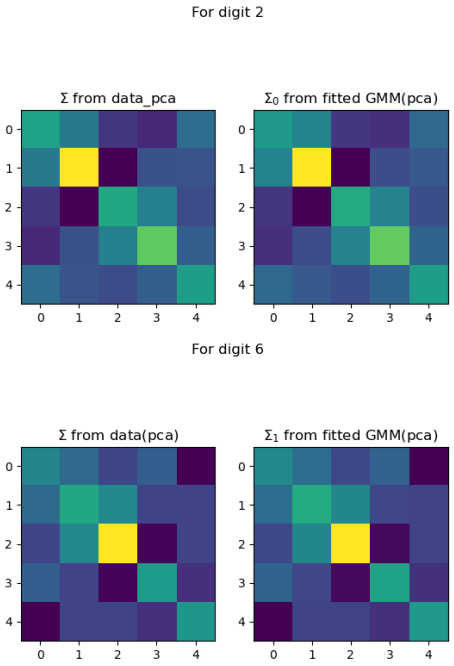


Figure 6: Sigma from pca data vs Sigma from GMM(pca)

2.3 Mis-classfication rate

The confusion matrix for fitted GMM vs K-means

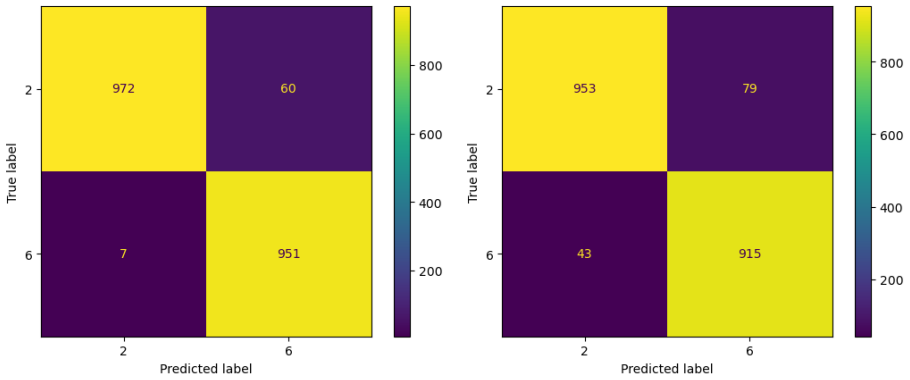


Figure 7: confusion matrix for GMM (left) vs K-means(right)

Hence we see mis-classification rate is

$$error_{gmm} = \frac{60+7}{1990} = 3.37\%$$

$$error_{k-means} = \frac{79+43}{1990} = 6.13\%$$

Hence GMM presents better performance.

3 Question 3: Low rank GMM

Implementation of low rak pdf function:

```
def ad_pdf(X,mu_c,sigma_c,r=5):
#     print ("$\mu$",mu_c.shape)
    _,D = sigma_c.shape
    #eigen decompose sigma
    u, s, vh = np.linalg.svd(sigma_c) #eigen and svd same for
    symmetric square matrix
    #pick top r eigen vectors
    u_r = u[:,:r]
    s_r = s[:r]
    #calculate pdf
    det_s = s_r.prod()
    term0 = X-mu_c
    term1 = (term0 @ u_r)
    term2 = term1 @ np.diag(s_r**(-0.5))
    exp_term =  -0.5 * np.linalg.norm(term2,axis=1)**2.0
    log_pdf = ((-D/2) * np.log(2*np.pi)) + (-0.5 * np.log(det_s)) +
     exp_term
    assert log_pdf.shape[0]==X.shape[0], "Error in pdf"
    return (np.exp(log_pdf))
```

Centroid Initialisation

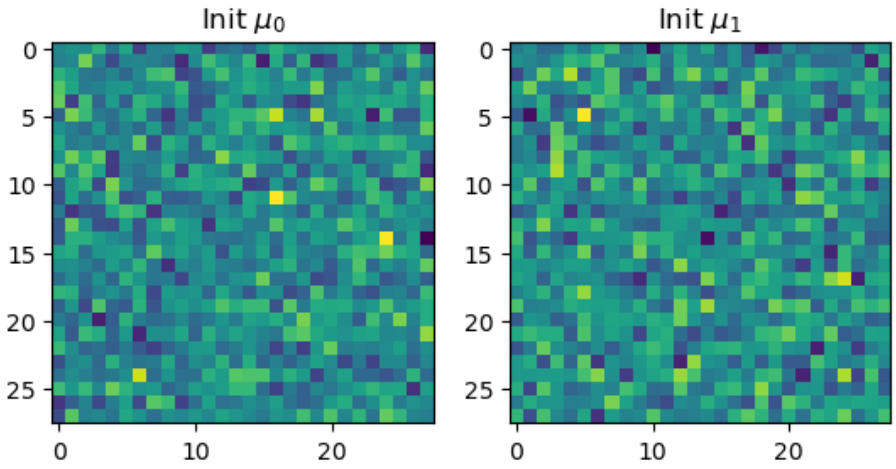


Figure 8: Initialisation

3.1 part a: EM with low rank approximation

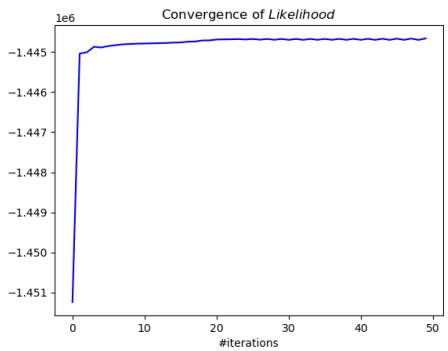


Figure 9: Convergence of Likelihood

Fitted GMM centroids:

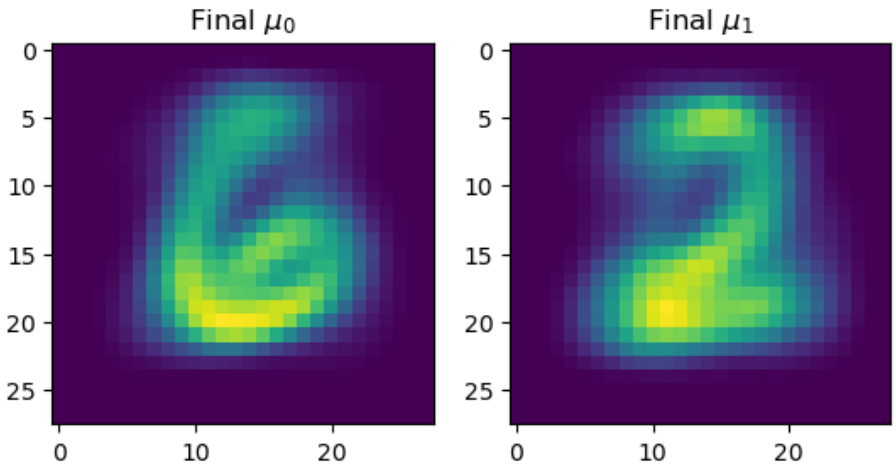


Figure 10: Convergence of Likelihood

Fitted GMM Sigma:

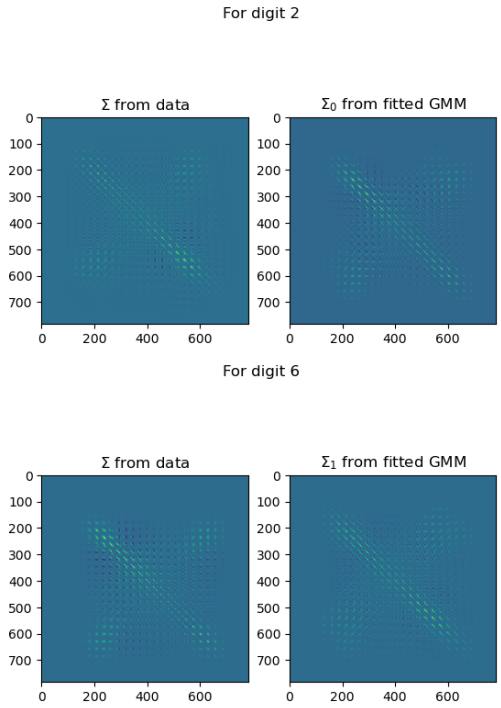


Figure 11: Confusion Matrix

3.2 part b: mis-classification rate

The confusion matrix for GMM is given below:

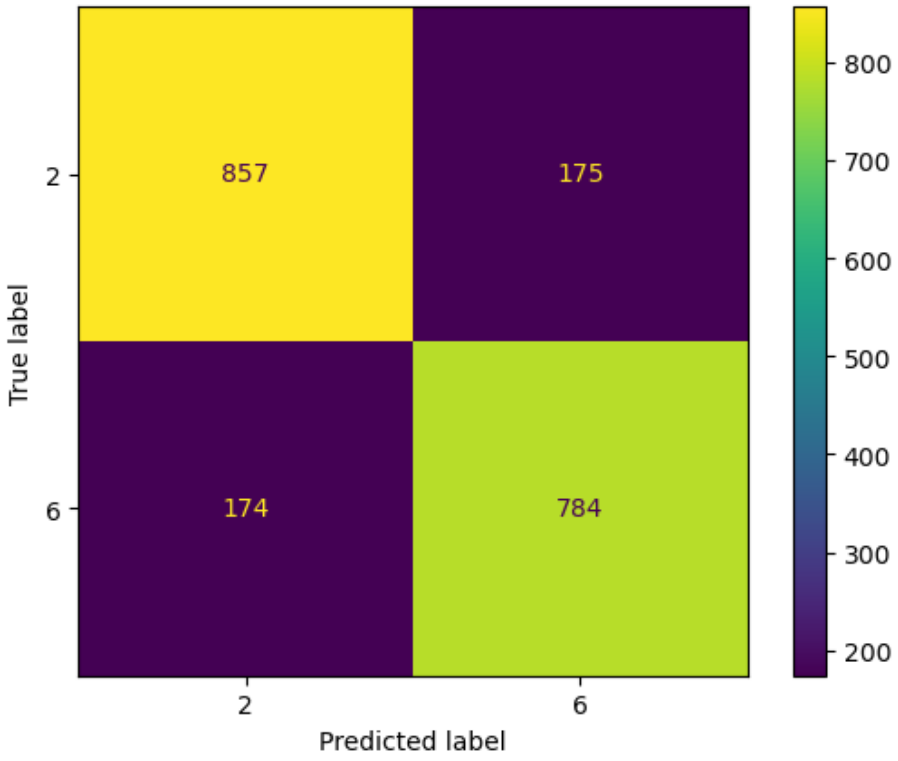


Figure 12: Confusion Matrix

Hence we see mis-classification rate is

$$error_{gmm-low-rank} = \frac{175 + 174}{1990} = 17.54\%$$

Thus we see that the GMM in this case performs a little worse than one with GMM on PCA.

4 Question 4: HMM

For the given problem we have the following state transition matrix

$$A = \begin{bmatrix} 0.95 & 0.05 \\ 0.1 & 0.9 \end{bmatrix}$$

We map state 0 and 1 to F and L respectively.
Similarly the emission matrix is:

$$E = \begin{bmatrix} 1/6 & 1/6 & 1/6 & 1/6 & 1/6 & 1/6 \\ 1/10 & 1/10 & 1/10 & 1/10 & 1/10 & 1/2 \end{bmatrix}$$

where 1 to 6 roll of dice are mapped to 0 to 5.

4.1 part a: Most likely state decoding

The implementation of forward backward recursion is

```
def fwd_backward(o_list,o_dict_repo,A,f0):
    #fwd pass
    f_list=[]
    for i,y_t in enumerate(o_list):
        o_t = o_dict_repo[y_t]
        if i ==0:
            f_i = o_t @ A.T @ f0
        else:
            f_i = o_t @ A.T @ f_list[-1]
        #normalize f_i
        f_i = f_i/f_i.sum()
        f_list.append(f_i)
    #bkwd pass
    b_list=[]
    for i,y_t in enumerate(o_list.copy()[::-1]):
        o_t = o_dict_repo[y_t]
        if i ==0:
            b_i = A @ o_t @ np.array([1,1]).reshape(2,1)
        else:
            b_i = A @ o_t @ b_i
        #normalize f_i
        b_i = b_i/b_i.sum()
        b_list.append(b_i)
    return (f_list,b_list)
```

```
alpha,beta = fwd_backward([2,6,6,6,4,1],o_dict_repo,A,pi)
gamma_list = []
for i in range(len(alpha)):
    gamma = alpha[i]*beta[i]
    gamma = gamma/gamma.sum()
    gamma_list.append(gamma)
```

```
gamma_list[2]
% array([[0.15636004],
%        [0.84363996]])
```

Hence we get $P(S_3 = L|\text{“266641”}) = 0.84363996$

4.2 part b: Viterbi

The implementation of Viterbi is

```
o_list = [2,3,6,6,4,1]
T = len(o_list)
K = A.shape[0]
path_dict = {}
#fill in trellis
def get_path_len(k1,k2,ot):
    path_len = -(np.log(A[k1,k2])+np.log(B[k2,ot-1]))
    return path_len

start = -1
path_dict[-1,0]=-np.log(pi).reshape(1,K)
for t in range(0,T):
#     print (o_list[t])
    path_dict[t,t+1]=np.array([get_path_len(k1,k2,o_list[t])\
                       for k1,k2 in itertools.product(range(0,K),
    range(0,K))]).reshape(K,K)

sink = 99
path_dict[6,sink]=np.zeros(shape=(K,1))

#traverse through time and find shortest path
short_dist_prev = np.zeros(shape=2)
short_path_list=[]
for (t1,t2),possible_paths in path_dict.items():
    #find shortest path to each state
    cum_path = possible_paths + short_dist_prev.reshape(2,1)
    short_dist_i,short_path_i = cum_path.min(axis=0),cum_path.
    argmin(axis=0)
    print("
    ----------------------------------------------------------------
    ")
    print (f"Between (t1,t2)={t1,t2})")
    print (f"shortest path to states:",short_dist_i)
    print()
    short_dist_prev = short_dist_i
    short_path_list.append(short_path_i)
```

```
------------------------------------------------------------
Between (t1,t2)=(-1, 0))
shortest path to states: [0.69314718 0.69314718]

------------------------------------------------------------
Between (t1,t2)=(0, 1))
shortest path to states: [2.53619994 3.10109279]

------------------------------------------------------------
Between (t1,t2)=(1, 2))
shortest path to states: [4.37925271 5.5090384 ]

------------------------------------------------------------
Between (t1,t2)=(2, 3))
shortest path to states: [6.22230547 6.30754609]

------------------------------------------------------------
Between (t1,t2)=(3, 4))
shortest path to states: [8.06535824 7.10605379]

------------------------------------------------------------
Between (t1,t2)=(4, 5))
shortest path to states: [9.908411   9.5139994]

------------------------------------------------------------
Between (t1,t2)=(5, 6))
shortest path to states: [11.75146376 11.92194501]

------------------------------------------------------------
Between (t1,t2)=(6, 99))
```

```
shortest path to states: [11.75146376]
```

Hence basis above we backtrack the shortest path as

```
backtrack shortest path
shortest path b/w time (6, 99) = 0
shortest path b/w time (5, 6) = 0
shortest path b/w time (4, 5) = 0
shortest path b/w time (3, 4) = 0
shortest path b/w time (2, 3) = 0
shortest path b/w time (1, 2) = 0
shortest path b/w time (0, 1) = 0
shortest path b/w (-1, 0) = 0
```

Here -1 is start and and 99 is sink. Since 0 is mapped to "F", the most likely sequence is **FFFFFF**

5 Question 5: Trellis for HMM

The trellis and the shortest path are given in next page.

We arrive at two shortest paths with $S_0 = H, S_1 = H, S_2 = H, S_3 = S$ and $S_0 = H, S_1 = H, S_2 = S, S_3 = S$, where 'H' = Healthy and 'S' = Sick. The two shortest path share the same length of 4.97449625.

Given the matrix B is symmetric, i.e $P(120 < B < 140|H) = P(120 < B < 140|S) = 0.3$, arriving at two shortest paths is rationalized.

The edge weights are arrived using trellis code from previous question:

```
o_list = [0,1,2]
T = len(o_list)
K = A.shape[0]
path_dict = {}
#fill in trellis
def get_path_len(k1,k2,ot):
    path_len = -(np.log(A[k1,k2])+np.log(B[k2,ot]))
    return path_len

start = -1
path_dict[-1,0]=-np.log(pi).reshape(1,K)
for t in range(0,T):
#     print (o_list[t])
    path_dict[t,t+1]=np.array([get_path_len(k1,k2,o_list[t])\
                       for k1,k2 in itertools.product(range(0,K),
    range(0,K))]).reshape(K,K)

sink = 99
path_dict[6,sink]=np.zeros(shape=(K,1))
```

```
print (path_dict)
{(-1, 0): array([[0.69314718, 0.69314718]]),
 (0,
  1): array([[0.73396918, 3.91202301],
        [2.12026354, 2.52572864]]),
 (1,
  2): array([[1.42711636, 2.81341072],
        [2.81341072, 1.42711636]]),
 (2,
  3): array([[2.52572864, 2.12026354],
        [3.91202301, 0.73396918]]),
 (6,
  99): array([[0.],
        [0.]])}
```

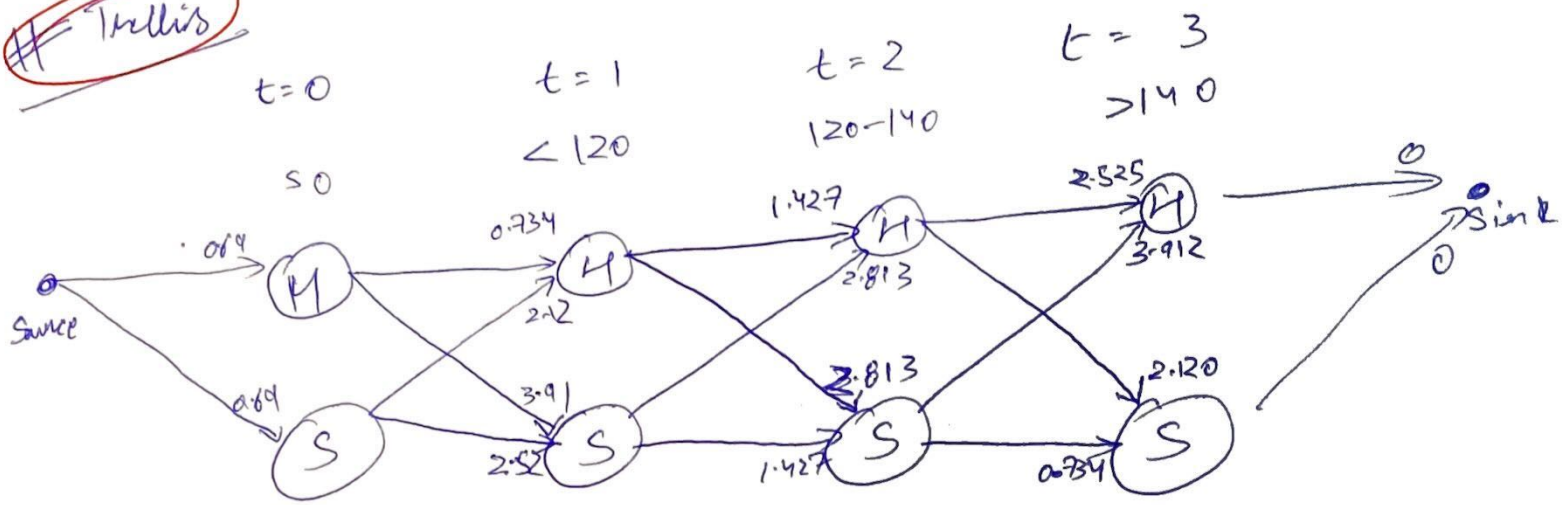
Question 5:

$$A = \begin{array}{c} \\ H \\ S \end{array} \begin{array}{c} \begin{array}{cc} H & S \end{array} \\ \begin{bmatrix} 0.8 & 0.2 \\ 0.2 & 0.8 \end{bmatrix} \end{array}$$

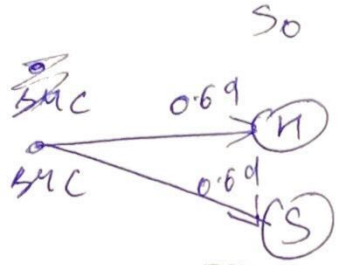
$$B = \begin{array}{c} \\ H \\ S \end{array} \begin{array}{c} \begin{array}{ccc} B<120 & 120\text{to}140 & >140 \end{array} \\ \begin{bmatrix} 0.6 & 0.3 & 0.1 \\ 0.1 & 0.3 & 0.6 \end{bmatrix} \end{array}$$

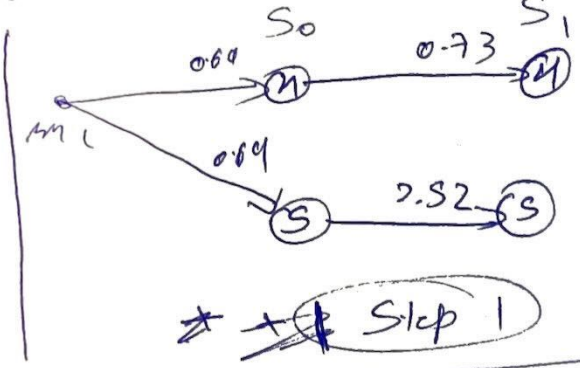
$$O = [\,<120,\ 120\text{to}140,\ >140\,]$$

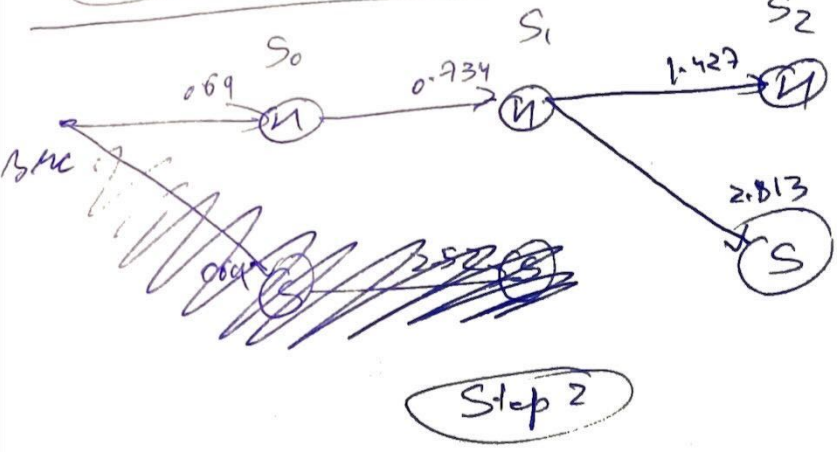
Trellis

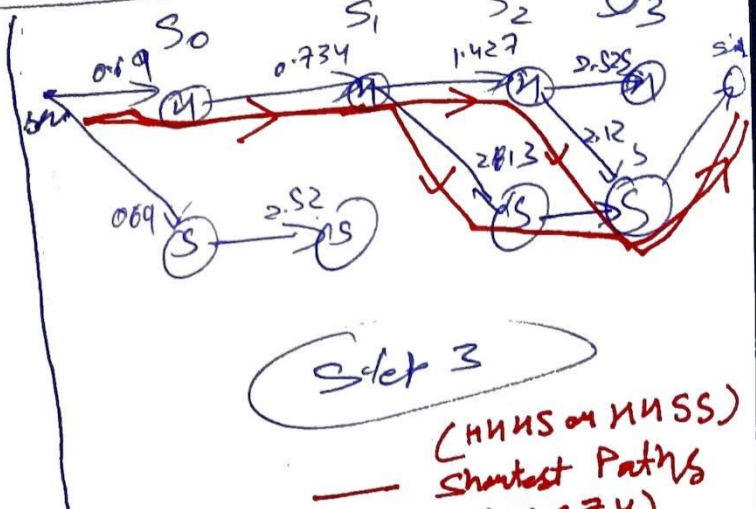


Part b: path elimination









(HHHS or HHSS)
Shortest Paths
(4.974)