1 Problem 1

1.1 Part a

We are asked to show that to minimize the forwards Divergence $D_{KL}(p\|q)$, we should set $q_1(x) = p(x)$ and $q_2(y) = p(y)$. To do so, we first see

$$D_{KL}(p\|q) = \sum_{x,y} p(x,y) log\left(\frac{p(x,y)}{q_1(x)q_2(y)}\right) \tag{1}$$

with the additional constraints:

$$\sum_x q_1(x) = 1 \tag{2}$$

$$\sum_y q_2(y) = 1 \tag{3}$$

Therefore, with these two constraints in hand we can develop a Lagrangian, i.e.,

$$L(q_1(x), q_2(y)) = \sum_{x,y} p(x,y) log\left(\frac{p(x,y)}{q_1(x)q_2(y)}\right) + \lambda_1\left(1 - \sum_x q_1(x)\right) + \lambda_2\left(1 - \sum_y q_2(y)\right) \tag{4}$$

$$= \sum_{x,y} p(x,y) log p(x,y) - \sum_x p(x) log\ q_1(x) - \sum_y p(y) log\ q_2(y) + \lambda_1\left(1 - \sum_x q_1(x)\right) + \lambda_2\left(1 - \sum_y q_2(y)\right) \tag{5}$$

To solve for the optimal λ_1, we take $\partial L/\partial q_1(x)$ at some point x, which is

$$\frac{\partial L}{\partial q_1(x)} = \frac{p(x)}{q_1(x)} - \lambda_1 = 0 \tag{6}$$

$$\Rightarrow \frac{p(x)}{\lambda_1} = q_1(x) \tag{7}$$

This will hold for each term in $q(x)$. Analogously, $\frac{p(y)}{\lambda_2} = q_2(y)$. Using these definitions above, we get, for L,

$$\sum_{x,y} p(x,y) log p(x,y) + \sum_x p(x) log \frac{p(x)}{\lambda_1} + \sum_y p(y) log \frac{p(y)}{\lambda_2} + \lambda_1 - 1 + \lambda_2 - 1 \tag{8}$$

Taking $\partial L/\partial \lambda_1$, we get

$$\frac{\partial L}{\partial \lambda_1} = -\frac{1}{\lambda_1}\sum_x p(x) + 1 = 0 \tag{9}$$

$$\Rightarrow \lambda_1 = \sum_x p(x) = 1 \tag{10}$$

and therefore

$$q_1(x) = \frac{p(x)}{1} = p(x) \tag{11}$$

Using the exact same logic (just flipped), it is easy to see

$$q_2(y) = \frac{p(y)}{1} = p(y) \tag{12}$$

and the proof is complete.

1.2 Part b

In this section we are asked to consider the reverse divergence $D_{KL}(q\|p)$ and to find its three distinct minima. To start, we see that

$$D_{KL}(q\|p) = \sum_{i\in X, j\in Y} q_1(x_i)q_2(y_j) log \frac{q_1(x_i)q_2(y_j)}{p(x_i, y_j)} \tag{13}$$

With the table of conditional probabilities defined in the homework assignment. As an initial note, it is east to see, at the points where $p(x_i, y_j) = 0$, it must be the case that either $q_1(x_1)$ or $q_2(y_j)$ is zero. This is because, if one of q_1 or q_2 is not zero and p is, the above expression will have a $1 * log(\infty)$ term in it, which will send the entire expression to infinity. Therefore, the domain of $q_1(x)$ and $q_2(y)$ is limited. The domain for which these are limited to one of the three shown below:

$$q_1(x) = \begin{bmatrix} a \\ 1-a \\ 0 \\ 0 \end{bmatrix}, q_2(y) = \begin{bmatrix} b \\ 1-b \\ 0 \\ 0 \end{bmatrix} \tag{14}$$

$$q_1(x) =, q_2(y) = \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}^T \tag{15}$$

$$q_1(x) = q_2(y) = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}^T \tag{16}$$

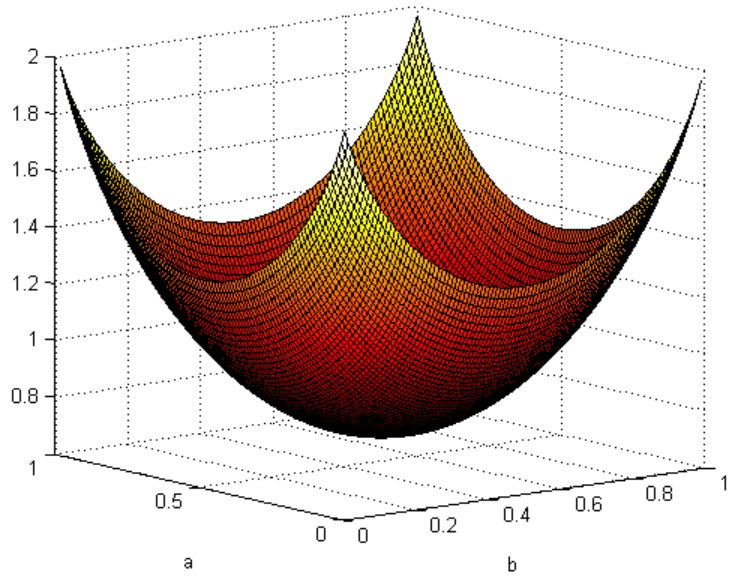
This is the only way to avoid the cross terms (which send our divergence to infinity) is to be in one of the three forms above. Clearly, the last two forms are local minima. However, the first set of values can be optimized over. To ease the amount of notation and spare many partial derivatives, I merely conducted a grid search in MatLab. This was done with the following code:

```
Min=20;
p=0;

q1L = 0:0.01:1;
q2L = 0:0.01:1;

for i=1:length(q1L)
    q11 = q1L(i);
    j=1-q11;
    q1 = [q11 j]';
    for k = 1:length(q2L)
        q21 = q2L(k);
        m=1-q21;
        q2 = [q21 m]';
        Total(i,k) = sum(sum((q1*q2').*log(q1*q2'*8)));
        if Total(i,k)<=Min
           Min=Total(i,k)
           q1
           q2
        end
    end
end
```

With the following results:

```
q1 =
    0.5000
    0.5000
q2 =
    0.5000
    0.5000
```

and the plot below, which shows the value of $D_{KL}(q\|p)$ (z axis) versus the various values of a, b. At these three minimum points, we can evaluate $D_{KL}(q\|p)$ as:



$$q_1(x) = q_2(y) = \begin{bmatrix} 0.5 \\ 0.5 \\ 0 \\ 0 \end{bmatrix} \Rightarrow D_{KL}(q\|p) = 4\,(0.25\ log(0.25/0.125)) = 0.6931 \tag{17}$$

$$q_1(x) =, q_2(y) = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \Rightarrow D_{KL}(q\|p) = log(1/0.25) = 1.3862 \tag{18}$$

$$q_1(x) = q_2(y) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \Rightarrow D_{KL}(q\|p) = log(1/0.25) = 1.3862 \tag{19}$$

1.3 Part c

In this subsection, we are asked to consider the case when we set $q(x, y) = p(x)p(y)$. From the table, it is easy to see (by adding rows/columns) that $p(x_i) = p(y_j) = 1/4\ \forall i \in X, j \in Y$. Therefore, there will be many points, as explained above, where $p(x, y) = 0$ and $q(x, y) \neq 0$. Therefore, the entire divergence expression will get sent to ∞ and $D_{KL}(q\|p) = \infty$.

2 Problem 2

2.1 Part a

In this section we are asked to derive the full conditionals $p(x_1|x_2)$ and $p(x_2|x_1)$. To derive this, I will only derive one of the conditionals, and the same proof can be applied reversing the two variables x_2 and x_1. Since the variance is symmetric and the means are the same, nothing will change. To start, I will generalize, for ease:

$$\mu = \begin{bmatrix}\mu_1 \\ \mu_2\end{bmatrix}, \Sigma = \begin{bmatrix}\Sigma_{aa} & \Sigma_{ab} \\ \Sigma_{ba} & \Sigma_{bb}\end{bmatrix} \tag{20}$$

Using Bayes Rule, we see

$$p(x_1|x_2) = \frac{p(x,y)}{p(x_2)} \tag{21}$$

Given the formula for a multi-variable Gaussian distribution, this is equivalent to

$$\frac{p(x,y)}{p(x_2)} = \frac{\frac{1}{2\pi\sqrt{|\Sigma|}} exp\{-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\}}{\frac{1}{\sqrt{2\pi\Sigma_{bb}}} exp\{-\frac{1}{2}(x-\mu_2)^T\Sigma_{bb}^{-1}(x-\mu_2)\}} \tag{22}$$

where $|\Sigma|$ is the determinant of Σ. To start, I will consider the coefficients, i.e.,

$$\frac{p(x,y)}{p(x_2)} = \frac{\frac{1}{2\pi\sqrt{|\Sigma|}}}{\frac{1}{\sqrt{2\pi\Sigma_{bb}}}} = \frac{\sqrt{2\pi\Sigma_{bb}}}{2\pi\sqrt{|\Sigma|}} = \frac{\sqrt{\Sigma_{bb}}}{\sqrt{2\pi(\Sigma_{aa}\Sigma_{bb} - \Sigma_{ab}\Sigma_{ba})}} \tag{23}$$

Multiplying both side of the fraction by $1/\sqrt{\Sigma_{bb}}$, we get

$$\frac{1}{\sqrt{2\pi}\sqrt{\Sigma_{aa} - \Sigma_{ab}\Sigma_{bb}^{-1}\Sigma_{ba}}} \tag{24}$$

Now, we have the coefficient set. This would seem to suggest $\Sigma_{a|b} = \Sigma_{aa} - \Sigma_{ab}\Sigma_{bb}^{-1}\Sigma_{ba}$. From this point on, I will assume that $\Sigma_{a|b}$ is defined by the above. Either way, we now can compute the two exponential terms. From the properties of exponents, the exponent term can be simplified to

$$exp\left\{-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu) + \frac{1}{2}(x-\mu_2)^T\Sigma_{bb}^{-1}(x-\mu_2)\right\} \tag{25}$$

To start, we expand the transpose terms, where $\bar{x}_1 = x - \mu_1$ and $\bar{x}_2 = x - \mu_2$ to get

$$exp\left\{-\frac{1}{2}\left(\begin{bmatrix}\bar{x}_1 & \bar{x}_2\end{bmatrix}\frac{1}{\Sigma_{a|b}\Sigma_{bb}}\begin{bmatrix}\Sigma_{bb} & -\Sigma_{ba} \\ -\Sigma_{ab} & \Sigma_{aa}\end{bmatrix}\begin{bmatrix}\bar{x}_1 \\ \bar{x}_2\end{bmatrix} - \bar{x}_2\Sigma_{bb}^{-1}\bar{x}_2\right)\right\} \tag{26}$$

$$=exp\left\{-\frac{1}{2}\left(\frac{\bar{x}_1\bar{x}_1}{\Sigma_{a|b}} - \frac{\bar{x}_1\Sigma_{ab}\bar{x}_2}{\Sigma_{a|b}\Sigma_{bb}} + \frac{\bar{x}_2\Sigma_{aa}\bar{x}_2}{\Sigma_{a|b}\Sigma_{bb}} - \frac{\bar{x}_2\bar{x}_2}{\Sigma_{bb}}\right)\right\} \tag{27}$$

$$=exp\left\{-\frac{1}{2\Sigma_{a|b}}\left(\bar{x}_1\bar{x}_1 - \frac{\bar{x}_1\Sigma_{ab}\bar{x}_2}{\Sigma_{bb}} + \frac{\bar{x}_2(\Sigma_{aa} - \Sigma_{a|b})\bar{x}_2}{\Sigma_{bb}}\right)\right\} \tag{28}$$

From the definition of $\Sigma_{a|b}$, we know

$$\Sigma_{aa} - \Sigma_{a|b} = \frac{\Sigma_{ab}\Sigma_{ab}}{\Sigma_{bb}} \tag{29}$$

and therefore the above is

$$exp\left\{-\frac{1}{2\Sigma_{a|b}}\left(\bar{x}_1\bar{x}_1-\frac{\bar{x}_1\Sigma_{ab}\bar{x}_2}{\Sigma_{bb}}+\frac{\bar{x}_2\Sigma_{ab}\Sigma_{ab}\bar{x}_2}{\Sigma_{bb}^2}\right)\right\} \tag{30}$$

$$=exp\left\{-\frac{1}{2\Sigma_{a|b}}\left((\bar{x}_1-\frac{\Sigma_{ab}}{\Sigma_{bb}}\bar{x}_2)^T(\bar{x}_1-\frac{\Sigma_{ab}}{\Sigma_{bb}}\bar{x}_2)\right)\right\} \tag{31}$$

and therefore we can define:

$$\Sigma_{a|b}=\Sigma_{aa}-\Sigma_{ab}\Sigma_{bb}^{-1}\Sigma_{ba} \tag{32}$$

$$\mu_{a|b}=\mu_1+\frac{\Sigma_{ab}}{\Sigma_{bb}}\bar{x}_2 \tag{33}$$

which is the definition from class and the proof is complete.

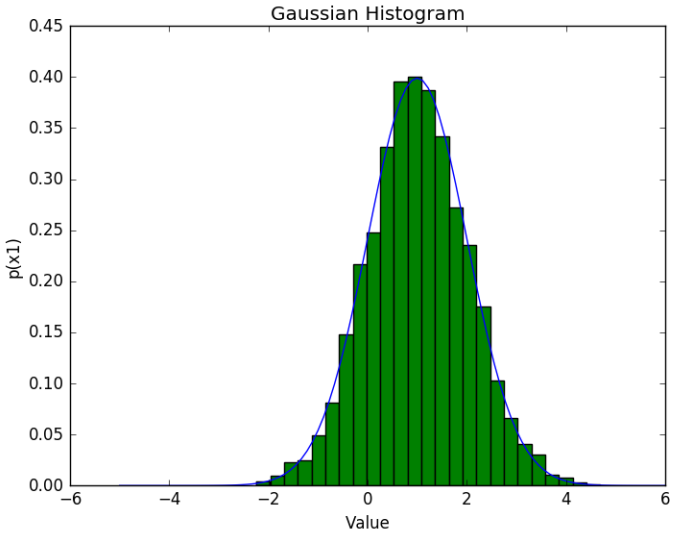
2.2 Part b

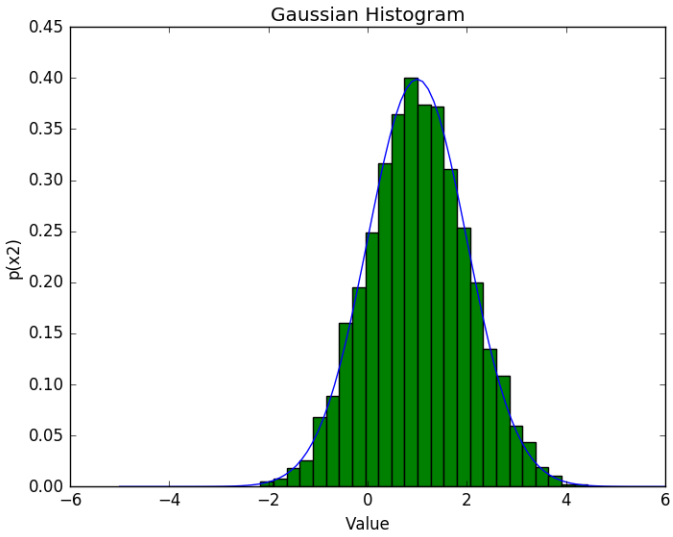
In this section we are asked to implement code which implements Gibbs Sampling for estimating. The code for this is shown below:

```
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.mlab as mlab
import math
from scipy.stats import gaussian_kde
from numpy.random import normal


mu = np.array([1,1])
sigma = np.matrix([[1, 0.5],[0.5,1]])


s_ab = sigma[0,0]-sigma[0,1]*sigma[1,0]/sigma[1,1]
s_ba = sigma[1,1]-sigma[0,1]*sigma[1,0]/sigma[0,0]




CSig = np.array([s_ab,s_ba])
n=0
x1 = 0
Max = 5001
X1 = np.zeros((Max,1))
X2 = np.zeros((Max,1))


while n<Max:

    #Sample x2
    Mu2  = mu[1]+sigma[0,1]/sigma[0,0]*(x1-mu[0])
```

```
    x2   = np.random.normal(Mu2,np.sqrt(s_ba),1)
    X2[n] = x2

    #Sample x1
    Mu1   = mu[0]+sigma[0,1]/sigma[1,1]*(x2-mu[1])
    x1   = np.random.normal(Mu1,np.sqrt(s_ab),1)
    X1[n]=x1


    n=n+1

print "------------"

x = np.linspace(-5,6,100)

plt.plot(x,mlab.normpdf(x,mu[0],np.sqrt(sigma[0,0])))
plt.hist(X1,bins=25,normed=True)
plt.title("Gaussian Histogram")
plt.xlabel("Value")
plt.ylabel("p(x1)")
plt.show()


plt.plot(x,mlab.normpdf(x,mu[1],np.sqrt(sigma[1,1])))
plt.hist(X2,bins=25,normed=True)
plt.title("Gaussian Histogram")
plt.xlabel("Value")
plt.ylabel("p(x2)")
plt.show()
```

With the following deliverables:





Which show the comparison of the sampled marginal pdfs (histogram) versus the true marginal (line).

3 Problem 3

3.1 Part a

To find the most probable sequence, I used the following theory in my code.

3.1.1 Prior Probability

The prior probability is defined as the unconditional probability of the state sequence in the row, i.e. $P(Z)$. To do this, from probability laws, this can be defined as:

$$P(Z) = P(z_1, z_2, z_3, z_4) \tag{34}$$

$$= P(z_1) \prod_{i=2}^{4} P(z_t|z_{t-1}) \tag{35}$$

$$= \pi_0(z_1) \prod_{i=2}^{4} A(z_t|z_{t-1}) \tag{36}$$

Which was done for each sequence.

3.1.2 Likelihood

The likelihood is defined as the conditional probability of the observation given the state sequence, i.e., $P(X|Z)$. Using the law of probability, this is defined as

$$P(X|Z) = P(x_1, x_2, x_3, x_4|z_1, z_2, z_3, z_4) \tag{37}$$

$$= \prod_{i=i}^{4} P(x_i|z_i) \tag{38}$$

$$= \prod_{i=i}^{4} \phi(x_t|z_t) \tag{39}$$

3.1.3 Posterior Probability

The posterior probability is conditional probability of the state sequence in the row given the sequence of observations, i.e. $P(Z|X)$. From Bayes Rule, we know

$$P(Z|X) = \frac{P(X|Z)P(Z)}{P(X)} = \frac{P(X|Z)P(Z)}{\sum_z P(X|Z)P(Z)} \tag{40}$$

$$= \frac{\pi_0(z_1)\phi(x_1|z_1) \prod_{i=2}^{4} \phi(x_t|z_t)A(z_t|z_{t-1})}{\sum_z \left(\pi_0(z_1)\phi(x_1|z_1) \prod_{i=2}^{4} \phi(x_t|z_t)A(z_t|z_{t-1}) \right)} \tag{41}$$

where the sum over z is the sum over every possible sequence.

```
import numpy as np
import itertools
import operator
```

```
pi = np.array([0.5,0.3,0.2])
A  = np.matrix([[0.5, 0.2,0.3],[0.2,0.4,0.4],[0.4,0.1,0.5]])
phi = np.matrix([[0.8,0.2],[0.1,0.9],[0.5,0.5]])

obs  = np.array([0,1,0,1])


##FIND MOST PROBABLE SEQUENCE

x = [0,1,2]
It = [p for p in itertools.product(x, repeat=4)]
Total={}
Total_Prob=0

for i in It:
    string = str(i)
    Prob = pi[i[0]]*phi[i[0],obs[0]]
    Uncond_Prob = pi[i[0]]
    Likelihood = phi[i[0],obs[0]]
    for j in range(1,4):
        prev = i[j-1]
        current = i[j]
        Prob = A[prev,current]*phi[current,obs[j]]*Prob
        Uncond_Prob = A[prev,current]*Uncond_Prob
        Likelihood  = Likelihood*phi[current,obs[j]]

    Total_Prob=Total_Prob+Prob
    Total[string] = (Prob,Uncond_Prob,Likelihood)

Result = {key:value[0]/Total_Prob for key, value in Total.items()}



sorted_Total = sorted(Result.items(), key=operator.itemgetter(1),reverse=True)
print sorted_Total[0:3]
```

With the following output:

Table 1: My caption

Most Probable State	Prior Probability	Likelihood	Posterior Probability
0222	0.0375	0.100	0.0735
0122	0.0200	0.180	0.0706
0201	0.0120	0.288	0.0678

3.2 Part b

Part b was done using entirely Python. You can see how I found $\alpha, \beta, \gamma, \xi$ and the rest of the parameters in the code, but it was the same strategy outlined in the textbook. I have additionally added comments in the code to easier understand my process. Below is my code:

```
from __future__ import division
import numpy as np
import itertools
import operator
import matplotlib.pyplot as plt

# Generate the data according to the specification in the homework description
# for part (b)

A0 = np.array([[0.5, 0.2, 0.3], [0.2, 0.4, 0.4], [0.4, 0.1, 0.5]])
phi0 = np.array([[0.8, 0.2], [0.1, 0.9], [0.5, 0.5]])
pi0 = np.array([0.5, 0.3, 0.2])

X = []

for _ in xrange(5000):
z = [np.random.choice([0,1,2], p=pi0)]
for _ in range(3):
z.append(np.random.choice([0,1,2], p=A0[z[-1]]))
x = [np.random.choice([0,1], p=phi0[zi]) for zi in z]
X.append(x)

# TODO: Implement Baum-Welch for estimating the parameters of the HMM


p     = 0
Alpha = []
sum   = np.zeros((3,1))
Beta  = []
Xi    = []
Gamma = []
Num = np.zeros((3,3))
Den = np.zeros((3,3))
Nump = 0
Denp = 0

#------------INITIALIZE PARAMETERS---------------#

Ai = np.random.uniform(0,1,(3,3))
row_sums = Ai.sum(axis=1)
Ai        = Ai / row_sums[:, np.newaxis]
```

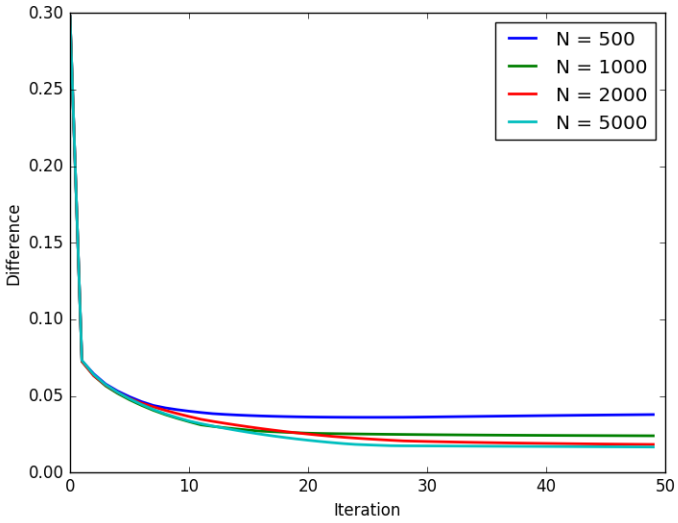
```
phii = np.random.uniform(0,1,(3,2))
row_sums = phii.sum(axis=1)
phii        = phii / row_sums[:, np.newaxis]


pii = np.random.uniform(0,1,(1,3))
pii = pii/np.sum(pii)
pii = pii[0]


#Number of observation points
N  = np.array([500,1000,2000,5000])
#Initialize error arrays
Error =[]
Errors=[]
#Number of iterations of EM array
Finish = 50

x_seq = list(itertools.product([0,1], repeat=4))
z_seq = list(itertools.product([0,1,2], repeat=4))

for j in range(0,4):

    Total = N[j]
#Initial parameters
    A  = Ai
    phi = phii
    pi  = pii
    Error=[]
    for n in range(0,Finish):
        print"--------------------------------------------------"
        print n
        print Total
        Num = np.zeros((3,3))
        Den = np.zeros((3,3))
        Nump = 0
        Denp = 0
        Num_phi = np.zeros((3,2))
        Den_phi = np.zeros((3,1))
        Total_Prob = 0
        Est_Prob=0
        Diff   = 0


        for p in range(0,Total):
            Alpha=[]
            Beta=[]
            Gamma=[]
```

```
        Xi=[]
        obs = X[p]
        PX2   = 0
        for i in z_seq:
            Prob2 = pi[i[0]]*phi[i[0],obs[0]]
            for j in range(1,4):
                prev = i[j-1]
                current = i[j]
                Prob2 *= A[prev,current]*phi[current,obs[j]]

            PX2+=Prob2

    #--------------Initialize Alpha, Beta-------------------#
        alpha = np.reshape(pi*phi[:,obs[0]],(3,1))
        Alpha.append(alpha)

        beta       = np.ones((3,1))
        Beta.append(beta)

    #---------------FORWARD PROCEDURE-----------------------
        for i in range(1,4):
            for j in range(0,3):
                sum[j]     = np.dot(np.transpose(alpha),A[:,j])

            alpha = np.transpose(phi[:,obs[i]]*np.transpose(sum))
            Alpha.append(alpha)

    #---------------BACKWARD PROCEDURE-------------------------
        for i in reversed(range(0,3)):
            sum  = np.zeros((3,1))
            for k in range(0,3):
                for j in range(0,3):
                    sum[k]+= beta[j]*A[k,j]*phi[j,obs[i+1]]

            beta = np.reshape(sum,(3,1))
            Beta.append(beta)

        Beta= Beta[::-1]

    #---------------TEMPORARY VALUES-------------------------

        for i in range(0,4):
            gamma = np.transpose(Alpha[i])*np.transpose(Beta[i])
            gamma = gamma/PX2
            Gamma.append(gamma)

            if i!=3:
                xi = np.zeros((3,3))
```

```
                    for k in range(0,3):
                        for j in range(0,3):
                            xi[k,j] = Alpha[i][k]*A[k,j]*Beta[i+1][j]*phi[j,obs[i+1]]
                    Xi.append(xi/PX2)

    #-----SOLVE FOR NUMERATOR AND DENOMINATOR TERMS OF PARAMETERS-------
            Nump += Gamma[0][0]
            Denp += np.sum(Gamma[0][0])

            for i in range(0,3):
                for j in range(0,3):
                    for m in range(0,3):
                        Num[i,j]+=Xi[m][i,j]
                        Den[i,j]+=Xi[m][i,0]
                        Den[i,j]+=Xi[m][i,1]
                        Den[i,j]+=Xi[m][i,2]

            for i in range(0,4):
                if obs[i]==0:
                    Num_phi[0,0] += Gamma[i][0][0]
                    Num_phi[1,0] += Gamma[i][0][1]
                    Num_phi[2,0] += Gamma[i][0][2]
                else:
                    Num_phi[0,1] += Gamma[i][0][0]
                    Num_phi[1,1] += Gamma[i][0][1]
                    Num_phi[2,1] += Gamma[i][0][2]

            Phi_Sum   = np.sum(Gamma,axis=0)

            Den_phi[0] += Phi_Sum[0][0]
            Den_phi[1] += Phi_Sum[0][1]
            Den_phi[2] += Phi_Sum[0][2]

#----------FIND P(X) AND P(X')-----------
        x_seq = list(itertools.product([0,1], repeat=4))
        z_seq = list(itertools.product([0,1,2], repeat=4))

        Diff = 0
        for x in x_seq:
            PXP=0
            PX=0
            for i in z_seq:
                Prob = pi0[i[0]]*phi0[i[0],x[0]]
                Prob2 = pi[i[0]]*phi[i[0],x[0]]
                for j in range(1,4):
                    prev = i[j-1]
                    current = i[j]
                    Prob *= A0[prev,current]*phi0[current,x[j]]
```

```
                    Prob2 *= A[prev,current]*phi[current,x[j]]

                PXP+=Prob
                PX+=Prob2
            Diff  +=abs(PXP-PX)/2

        Error.append(Diff)
#-----------SOLVE FOR NEW PARAMETERS------------
        A   = np.divide(Num,Den)
        phi = np.divide(Num_phi,Den_phi)
        pi  = np.divide(Nump,Denp)

        print Diff
    Errors.append(Error)

### Plot the results

n=range(0,Finish)

for i in xrange(0, 4):
  plt.plot(n, Errors[i], label = "N = " + str(N[i]), linewidth=2)

plt.xlabel("Iteration" )
plt.ylabel("Difference")
plt.legend(loc='upper right')
plt.show()
```

with the output: You can see the convergence of the algorithm in the plot.



4 Problem 4

Shown below is my code for Kaggle. I have included extensive comments in the code to explain my logic. Basically, I used pre-processing along with recursive least-squares.

```
import numpy as np
import pandas
import sklearn
import cPickle
from collections import Counter
import math
import scipy
import scipy.ndimage
from scipy.stats import mode
from matplotlib import pyplot
import matplotlib as mpl


#Import Data
noisy = pandas.read_csv("train_noised.csv",header=None)
clean = pandas.read_csv("train_clean.csv",header=None)
test = pandas.read_csv("test_noised.csv",header=None)


(r,c) = np.shape(noisy)

#Take away the ID row
del noisy[0]
del clean[0]
del test[0]

#Clean up the arrays
cleaned = np.array(clean[:][1:])
cleaned = cleaned.astype(float)

Noisy_image_plot = np.array(noisy[:][1:])
Noisy_image_plot = Noisy_image_plot.astype(float)

image = np.array(noisy[:][1:])
image = image.astype(float)

#Plotting Function
def show_image(Image_filtered):
    """
    Render a given numpy.uint8 2D array of pixel data.
    """
    fig = pyplot.figure()
    ax = fig.add_subplot(1,1,1)
    imgplot = ax.imshow(Image_filtered, cmap=mpl.cm.Greys)
```

```
    imgplot.set_interpolation('nearest')
    ax.xaxis.set_ticks_position('top')
    ax.yaxis.set_ticks_position('left')
    pyplot.show()


#Patch Function
def get_patches(X,size):
    m,n = X.shape
    X = np.pad(X, ((2, 2), (2, 2)), 'constant')
    patches = np.zeros((m*n, size**2))
    for i in range(m):
        for j in range(n):
            patches[i*n+j] = X[i:i+size,j:j+size].reshape(size**2)
    return patches

#This is my pre-processing step. I go through each and every pixel and find all of the pixels t
Image_filtered = image
Limit=200

for i in range(0,image.shape[0]):
    Digit=Image_filtered[i,:]

    for j in range(0,len(Digit)):

A = Digit[np.minimum(j+1,image.shape[0]-1)]
B = Digit[np.maximum(j-1,0)]
C = Digit[np.minimum(j+28,image.shape[1]-1)]
D = Digit[np.maximum(j-28,0)]
E = Digit[np.minimum(j+27,image.shape[1]-1)]
F = Digit[np.maximum(j-27,0)]
G = Digit[np.minimum(j+29,image.shape[1]-1)]
H = Digit[np.maximum(j-29,0)]
if (A<Limit and B<Limit and C<Limit and D<Limit and E<Limit and F<Limit and G<Limit and H<Limi
Image_filtered[i,j]=0
elif (C==255 and D==255):
Image_filtered[i,j]=255



#Define linear regresstion parameters
t   = np.reshape(cleaned,((r-1)*(c-1)))
Theta= get_patches(Image_filtered,5)

#Define my local kernel, which multiplies by each 5x5 patch point. This kernel weights points r
tau = 2
T   = 2
```

```
kernel = np.matrix([[T,T,T,T,T],[T,1,1,1,T],[T,1,0,1,T],[T,1,1,1,T],[T,T,T,T,T]])
kernel = np.reshape(kernel,(1,25))

#Local weighting matrix
rk   = np.square(np.array(kernel)/(2.0*tau**2))

for i in range(0,25):
    rk[0][i] = math.exp(-rk[0][i])


#Locally weight each pixel
Theta =rk*Theta
#Regularization parameter
lam = 1e-8
num = Theta.shape[1]
Reg = lam*np.identity(num)

#Solve for w
w = np.dot(np.dot(np.linalg.inv(np.dot(np.transpose(Theta),Theta)+Reg),np.transpose(Theta)),t)

#Solve for my estimated digit
Image_Clean   = np.dot(Theta,w)

#Cutoff any negative values or any above 255
Image_Clean   = np.maximum(Image_Clean,0)
Image_Clean   = np.minimum(Image_Clean,255)

#I now filter the image once more. I use the same principles as explained above, but with a lo
Limit2=130

Image_Reshape = np.reshape(Image_Clean,(r-1,c-1))


for i in range(0,image.shape[0]):
Digit=Image_Reshape[i,:]

for j in range(0,len(Digit)):

A = Digit[np.minimum(j+1,image.shape[0]-1)]
B = Digit[np.maximum(j-1,0)]
C = Digit[np.minimum(j+28,image.shape[1]-1)]
D = Digit[np.maximum(j-28,0)]
E = Digit[np.minimum(j+27,image.shape[1]-1)]
F = Digit[np.maximum(j-27,0)]
G = Digit[np.minimum(j+29,image.shape[1]-1)]
H = Digit[np.maximum(j-29,0)]
Num_pix = int(A>Limit2)+int(B>Limit2)+int(C>Limit2)+int(D>Limit2)+int(E>Limit2)+int(F>Limit2)+
```

```
if (A<Limit2 and B<Limit2 and C<Limit2 and D<Limit2 and E<Limit2 and F<Limit2 and G<Limit2 and
Image_Reshape[i,j]=0
elif (C<10 and D<10):
Image_Reshape[i,j]=0
elif Image_Reshape[i,j]<10:
Image_Reshape[i,j]=0
elif Num_pix>5:
Image_Reshape[i,j]=255
for j in range(0,len(Digit)):

A = Digit[np.minimum(j+1,image.shape[0]-1)]
B = Digit[np.maximum(j-1,0)]
C = Digit[np.minimum(j+28,image.shape[1]-1)]
D = Digit[np.maximum(j-28,0)]
E = Digit[np.minimum(j+27,image.shape[1]-1)]
F = Digit[np.maximum(j-27,0)]
G = Digit[np.minimum(j+29,image.shape[1]-1)]
H = Digit[np.maximum(j-29,0)]
if (C==255 and D==255):
Image_Reshape[i,j]=255




Image_Final =np.reshape(Image_Reshape,(r-1*c-1))

#Find the Error
Diff = (Image_Final-t)
Error =np.linalg.norm(Diff,2)/np.sqrt(r*c)

print Error




#------------------------------TEST DATA----------------------------------#

#This was done in the exact same manner described above
test_image = np.array(test[:][1:])
test_image = test_image.astype(float)
test_image = test_image
(l,w2) = np.shape(test_image)

for i in range(0,test_image.shape[0]):
Digit=test_image[i,:]

for j in range(0,len(Digit)):
```

```
A = Digit[np.minimum(j+1,image.shape[0]-1)]
B = Digit[np.maximum(j-1,0)]
C = Digit[np.minimum(j+28,image.shape[1]-1)]
D = Digit[np.maximum(j-28,0)]
E = Digit[np.minimum(j+27,image.shape[1]-1)]
F = Digit[np.maximum(j-27,0)]
G = Digit[np.minimum(j+29,image.shape[1]-1)]
H = Digit[np.maximum(j-29,0)]
if (A<Limit and B<Limit and C<Limit and D<Limit and E<Limit and F<200 and G<Limit and H<Limit)
test_image[i,j]=0
elif (C==255 and D==255):
Image_filtered[i,j]=255

Theta_T = get_patches(test_image,5)
#Use the same local weights
Theta_T= rk*Theta_T


#Use the optimized w
Test_digits   = np.dot(Theta_T,np.transpose(w))

#Cut off the image again
Test_digits   = np.maximum(Test_digits,0)
Test_digits   = np.minimum(Test_digits,255)



Test_Reshape = np.reshape(Test_digits,(l,w2))

for i in range(0,test_image.shape[0]):
Digit=Test_Reshape[i,:]

for j in range(0,len(Digit)):

A = Digit[np.minimum(j+1,image.shape[0]-1)]
B = Digit[np.maximum(j-1,0)]
C = Digit[np.minimum(j+28,image.shape[1]-1)]
D = Digit[np.maximum(j-28,0)]
E = Digit[np.minimum(j+27,image.shape[1]-1)]
F = Digit[np.maximum(j-27,0)]
G = Digit[np.minimum(j+29,image.shape[1]-1)]
H = Digit[np.maximum(j-29,0)]
Num_pix = int(A>Limit2)+int(B>Limit2)+int(C>Limit2)+int(D>Limit2)+int(E>Limit2)+int(F>Limit2)+
if (A<Limit2 and B<Limit2 and C<Limit2 and D<Limit2 and E<Limit2 and F<Limit2 and G<Limit2 and
Test_Reshape[i,j]=0
elif (C<10 and D<10):
Test_Reshape[i,j]=0
elif Test_Reshape[i,j]<10:
```

```
Test_Reshape[i,j]=0
elif Num_pix>5:
Test_Reshape[i,j]=255
for j in range(0,len(Digit)):

A = Digit[np.minimum(j+1,image.shape[0]-1)]
B = Digit[np.maximum(j-1,0)]
C = Digit[np.minimum(j+28,image.shape[1]-1)]
D = Digit[np.maximum(j-28,0)]
E = Digit[np.minimum(j+27,image.shape[1]-1)]
F = Digit[np.maximum(j-27,0)]
G = Digit[np.minimum(j+29,image.shape[1]-1)]
H = Digit[np.maximum(j-29,0)]
if (C==255 and D==255):
Test_Reshape[i,j]=255

Image_Final =np.reshape(Test_Reshape,(l*w2))

#Create Column for the Kaggle Submission
Names=[]

for i in range(0,l):
    for j in range(0,w2):
        row = str(i)
        col = str(j)
        Names.append(row+'_'+col)

#Write to a Dataframe
df = pandas.DataFrame(Image_Final, Names,columns=['Val'])

df.to_csv('erwaner_test_data2.csv')
```

5 Problem 5

5.1 Part a

For this part, we are asked to find the whitening matrix D such that

$$\tilde{X} = DX \tag{42}$$

such that

$$\frac{1}{N}\tilde{X}\tilde{X}^T = I \tag{43}$$

With these two equations, it is easy to see

$$\frac{1}{N}DXX^TD = I \tag{44}$$

Define $A = XX^T$ and V as the matrix of eigenvalues of A. From the properties of A, V,

$$V^TAV = \Lambda \tag{45}$$

where Λ is the diagonal matrix of eigenvalues. From this above equation, it is easy to see:

$$\Lambda^{-1/2}V^TAV\Lambda^{-1/2} = I \tag{46}$$

To normalize this equation, we add in a normalization term $\sqrt{N}$ such that

$$\sqrt{N}\Lambda^{-1/2}V^TAV\Lambda^{-1/2}\sqrt{N} = N \tag{47}$$

and therefore we define $D = \sqrt{N}\Lambda^{-1/2}V^T$.

5.2 Part b

In this section we are asked to implement code which recreated the actual signal S. To do this, I used the following code:

```
from __future__ import division
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.mlab as mlab
import math

# Generate the data according to the specification in the homework description

N = 10000

# Here's an estimate of gamma for you
G = lambda x: np.log(np.cosh(x))
gamma = np.mean(G(np.random.randn(10**6)))

s1 = np.sin((np.arange(N)+1)/200)
s2 = np.mod((np.arange(N)+1)/200, 2) - 1
```

```
S = np.concatenate((s1.reshape((1,N)), s2.reshape((1,N))), 0)

A = np.array([[1,2],[-2,1]])

X = A.dot(S)

# TODO: Implement ICA using a 2x2 rotation matrix on a whitened version of X

L,V = np.linalg.eig(np.dot(X,np.transpose(X)))
Lam = np.diag(np.matrix(L).A1)

D   = np.sqrt(N)*np.dot(np.sqrt(np.linalg.inv(Lam)),np.transpose(V))

Xtil = np.dot(D,X)

Gam  = np.mean(np.log(np.cosh(np.random.normal(0,1,10**6))))

Theta = np.linspace(0,math.pi/2,50)
Total_Cost=[]


for i in range(0,len(Theta)):
    theta=Theta[i]
    W = np.array([[math.cos(theta),math.sin(-theta)],[math.sin(theta),math.cos(theta)]])
    Cost = 0
    for j in range(0,len(W)):
        G     = np.mean(np.log(np.cosh(np.dot(W[j],Xtil))))
        Cost += (G-Gam)**2


    Total_Cost.append(Cost)
    if Cost == np.min(Total_Cost):
        Theta_Min = theta
        W_Min     = W

Y   = np.dot(W,Xtil)


print np.shape(Y)
print np.shape(Y[1,:])



plt.plot(range(0,N),Y[0,:],'r',range(0,N),Y[1,:],'g')
plt.ylabel('Y Value')
plt.xlabel('Iteration')
plt.show()
```

```
plt.plot(range(0,N),S[0,:],'r',range(0,N),S[1,:],'g')
plt.ylabel('S Value')
plt.xlabel('Iteration')
plt.show()


plt.plot(Theta,Total_Cost)
plt.ylabel('J(y)')
plt.xlabel('Theta')
plt.show()
```

With the following deliverables: