HW #4 example solutions

Due: Never

Question 1

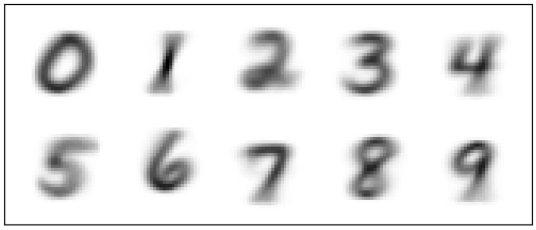
(a) Let N_c represent the number of examples of the cth class, and let N_{cd} be the number of times $x_d = 1$ in the cth class. Letting the data subset $\{x_d\} = \{x_{id}\}_{i=1}^{10k}$ represent all instances of the dth dimension of $\mathbf{x}$, we have

$$\begin{aligned} p(\theta_{cd}|c,\{x_d\}) &\propto p(\{x_d\}|c,\theta_{cd})p(\theta_{cd}|c) \\ &\propto \theta_{cd}(1-\theta_{cd})\prod_{i=1}^{10k}\theta_{cd}^{x_{id}}(1-\theta_{cd})^{1-x_{id}} \\ &\propto \theta_{cd}^{N_{cd}+1}(1-\theta_{cd})^{N_c-N_{cd}+1} \\ &\propto \beta(2+N_{cd}, 2+N_c-N_{cd}) \end{aligned}$$

Since the mode of a $\beta(a,b)$ distribution is $(a-1)/(a+b-2)$, we have

$$\hat{\theta} = \frac{1+N_{cd}}{2+N_c}.$$

(b)
```
import numpy as np
import data
import math
import random
import scipy
from scipy import optimize

# Load data
N_data, train_images, train_labels, test_images, test_labels = data.load_mnist()
N = 10000 # Number of data points in training set
train_images = train_images[:N,:] # 10k x 784 array
train_labels = train_labels[:N,:] # 10k x 10 array
train_images = np.ndarray.round(train_images) # Binarize the data

# Fit theta
Ncd = np.matmul(np.transpose(train_images),train_labels) # 784 x 10 array
Nc = train_labels.sum(axis=0)
thetaHat = (1+Ncd)/(2+Nc)  # 784 x 10 array
data.save_images(np.transpose(thetaHat),'ques1') # Plot thetaHat
```



(c)

$$
\begin{aligned}
p(c|\mathbf{x}_i, \boldsymbol{\theta}_c, \boldsymbol{\pi}) &\propto p(\mathbf{x}_i|c, \boldsymbol{\theta}_c) p(c|\boldsymbol{\pi}) \\
&\propto \cdots \text{equation (1) on assignment sheet} \cdots \\
&\propto \left(\prod_{d=1}^{784} \theta_{cd}^{x_{id}} (1-\theta_{cd})^{1-x_{id}} \right) \left(\prod_{c'=0}^{9} \pi_{c'}^{I(c'=c)} \right) \\
\therefore \log p(c|\mathbf{x}_i, \boldsymbol{\theta}_c, \boldsymbol{\pi}) &= \log \pi_c + \sum_{d=1}^{784} [x_{id} \log \theta_{cd} + (1-x_{id}) \log(1-\theta_{cd})] + \text{const}
\end{aligned}
$$

(d) Since $\pi_c = \frac{1}{10} \; \forall \; c$, absorb $\log \pi_c$ into the constant. Define

$$
\mathcal{L} = \sum_{d=1}^{784} [x_{id} \log \theta_{cd} + (1-x_{id}) \log(1-\theta_{cd})] .
$$

Continuing from the Python code in part (b),

```
# Proxy for predictive log-likelihood
logPtrain = np.matmul(train_images,np.log(thetaHat)) + \
            np.matmul(1-train_images,np.log(1-thetaHat)) # 10k x 10 array
avLtrain = np.mean(np.sum(logPtrain*train_labels,axis=1))
logPtest = np.matmul(test_images,np.log(thetaHat)) + \
           np.matmul(1-test_images,np.log(1-thetaHat)) # 10k x 10 array
avLtest = np.mean(np.sum(logPtest*test_labels,axis=1))
print(round(avLtrain,2), round(avLtest,2))

# Predictive accuracy
M = len(test_images) # Number of data points in test set
# 10k x 1 vector indicating whether a prediction was correct (1) or not (0):
accsTrain = train_labels[np.arange(N),logPtrain.argmax(1)]
accTrain = sum(accsTrain)/N
accsTest = test_labels[np.arange(M),logPtest.argmax(1)]
accTest = sum(accsTest)/M
print (round(accTrain*100,2),round(accTest*100,2))
```

Results:

	Train	Test
Average $\mathcal{L}$	-170.1	-172.6
Predictive accuracy	83.98%	84.14%

Examples of equally acceptable (full marks) approaches:

- Keeping the normalizing constants in parts (c) and (d) should lead to $\mathcal{L}$ values much closer to 0
- Maximizing predictive log-likelihood across classes (this isn't ideal) increases $\mathcal{L}$ by a tiny amount

Question 2

(a) False in the real world: eg black pixels tend to occur near other black pixels.
True under the assumptions of an NBC.

(b) False in the real world, for similar reasons to before.
False under the assumptions of an NBC, because $p(x_i, x_j|\boldsymbol{\theta}, \boldsymbol{\pi}) \neq p(x_i|\boldsymbol{\theta}, \boldsymbol{\pi})p(x_j|\boldsymbol{\theta}, \boldsymbol{\pi})$. Note:

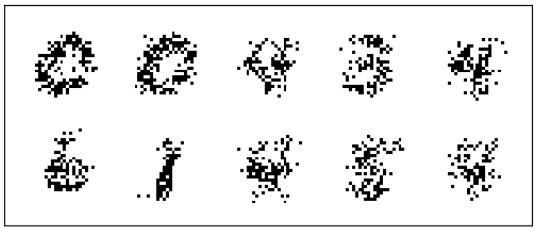
$$p(x_i, x_j|\boldsymbol{\theta}, \boldsymbol{\pi}) = \sum_c p(c|\boldsymbol{\pi})p(x_i, x_j|c, \boldsymbol{\theta}, \boldsymbol{\pi})$$

whereas

$$p(x_i|\boldsymbol{\theta}, \boldsymbol{\pi})p(x_j|\boldsymbol{\theta}, \boldsymbol{\pi}) = \left(\sum_c p(c|\boldsymbol{\pi})p(x_i|c, \boldsymbol{\theta}, \boldsymbol{\pi})\right)\left(\sum_c p(c|\boldsymbol{\pi})p(x_j|c, \boldsymbol{\theta}, \boldsymbol{\pi})\right)$$

(c) Continuing the code from question 1:

```
c = (np.floor(np.random.rand(10)*10)).astype(int) # Pick the classes
xt = np.random.rand(10,784) # Prepare to sample 10 images
thresh = np.asmatrix(thetaHat[:,c].T) # Set thresholds
sample10 = 1*(thresh > np.asmatrix(xt)).T # Complete the sampling
data.save_images(np.transpose(sample10),'ques2c') # Plot the images
```



(d) Using the hint provided, we can marginalize over c:

$$\begin{aligned}
p(\mathbf{x}_{\text{bottom}}|\mathbf{x}_{\text{top}}, \boldsymbol{\theta}, \boldsymbol{\pi}) &= \sum_{c=0}^{9} p(\mathbf{x}_{\text{bottom}}|c, \boldsymbol{\theta})p(c|\mathbf{x}_{\text{top}}, \boldsymbol{\theta}, \boldsymbol{\pi}) \\
&= \sum_{c=0}^{9} \left[\prod_{d\in\text{bottom}} p(x_d|c, \boldsymbol{\theta}\right] \left[\frac{p(\mathbf{x}_{\text{top}}|c, \boldsymbol{\theta})p(c|\boldsymbol{\pi})}{p(\mathbf{x}_{\text{top}}|\boldsymbol{\theta}, \boldsymbol{\pi})}\right] \\
&= \frac{1}{\sum_{c=0}^{9} \pi_c \prod_{d\in\text{top}} \theta_{cd}^{x_d}(1-\theta_{cd})^{1-x_d}} \sum_{c=0}^{9} \pi_c \prod_{d=1}^{784} \theta_{cd}^{x_d}(1-\theta_{cd})^{1-x_d} \\
&= \frac{1}{\sum_{c=0}^{9} \prod_{d\in\text{top}} \theta_{cd}^{x_d}(1-\theta_{cd})^{1-x_d}} \sum_{c=0}^{9} \prod_{d=1}^{784} \theta_{cd}^{x_d}(1-\theta_{cd})^{1-x_d} \quad \text{if } \pi_c = 0.1 \ \forall\, c
\end{aligned}$$

Alternatively, we could have started directly with the expression for conditional probability:

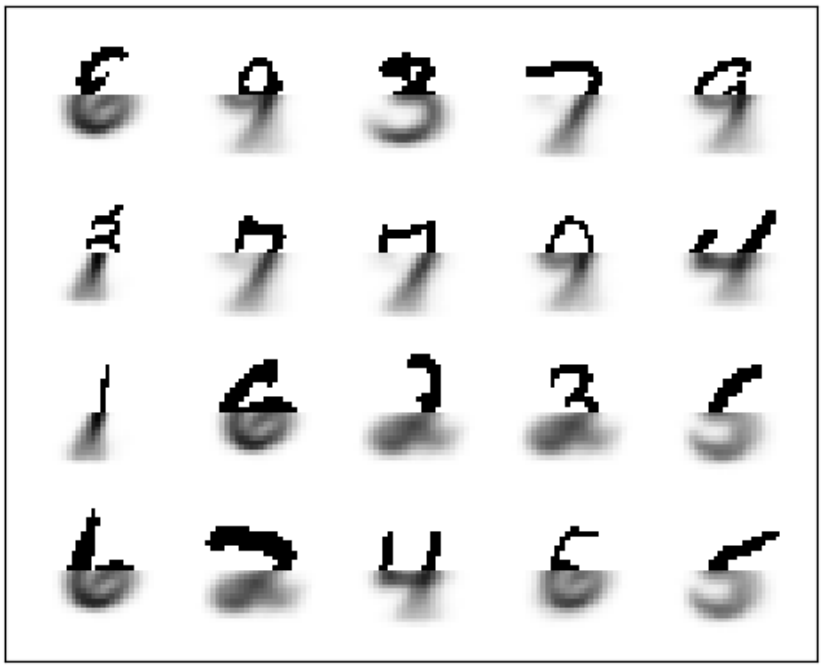
$$p(\mathbf{x}_{\text{bottom}}|\mathbf{x}_{\text{top}}, \boldsymbol{\theta}, \boldsymbol{\pi}) = \frac{p(\mathbf{x}_{\text{all}}|\boldsymbol{\theta}, \boldsymbol{\pi})}{p(\mathbf{x}_{\text{top}}|\boldsymbol{\theta}, \boldsymbol{\pi})}$$

which arrives at the same answer.

(e) Similarly:

$$p(x_{i\in\text{bottom}} = 1|\mathbf{x}_{\text{top}}, \boldsymbol{\theta}, \boldsymbol{\pi}) = \frac{1}{\sum_{c=0}^{9} \prod_{d\in\text{top}} \theta_{cd}^{x_d}(1-\theta_{cd})^{1-x_d}} \sum_{c=0}^{9} \left(\prod_{d\in\text{top}} \theta_{cd}^{x_d}(1-\theta_{cd})^{1-x_d}\right) \theta_{cd}$$

(f)
```
x = train_images[np.random.choice(N,20),:] # 20 x 784 array

# Denominator in part (e)
pi_dtop = np.zeros((10,20)) # The pi product occurs in num and denom
thetaHat = thetaHat.T
dtop = range(392) # Pixels in top half
for c in range(10):
    for i in range(20):
        fac = thetaHat[c,dtop]**x[i,dtop] * (1-thetaHat[c,dtop])**(1-x[i,dtop])
        pi_dtop[c,i] = np.prod(fac)
denr = pi_dtop.sum(axis=0) # 1x20

# Numerator in part (e)
dbott = range(392,784)
numr = np.zeros((392,20))
for d in dbott:
    for c in range(10):
        numr[d-392,:] += pi_dtop[c,:]*thetaHat[c,d] # (1x20)

x[:,dbott] = np.transpose(np.divide(numr,denr))
data.save_images(x,'ques2f') # Plot the images
```



Question 3

(a) $10 \times 28^2 = 7840$.

(b) $9 \times 28^2 = 7056$. e.g. following Murphy section 8.3.7, we could set $\mathbf{w}_c = \mathbf{0}$.

(c)

$$\nabla_{\mathbf{w}} \log p(c|\mathbf{x}, \mathbf{w}) = \nabla_{\mathbf{w}} \log \frac{\exp(\mathbf{w}_c^T \mathbf{x})}{\sum_{c'=0}^{9} \exp(\mathbf{w}_{c'}^T \mathbf{x})} = \nabla_{\mathbf{w}} \left[\mathbf{w}_c^T \mathbf{x} - \log \left(\sum_{c'=0}^{9} \exp(\mathbf{w}_{c'}^T \mathbf{x}) \right) \right]$$

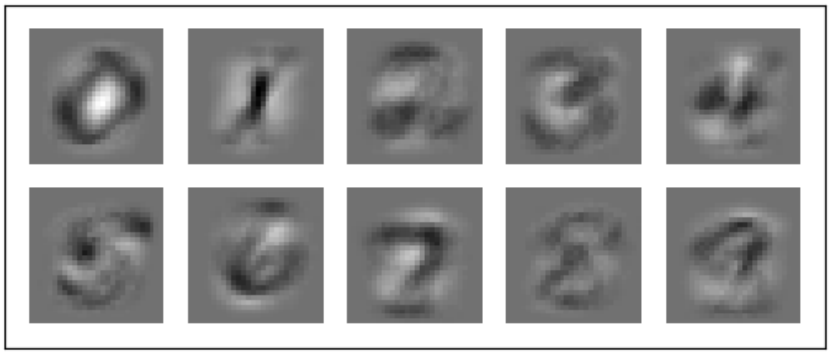
The answer is a 7840×1 column vector $\mathbf{g} = \nabla_{\mathbf{w}} \log p(c|\mathbf{x}, \mathbf{w})$. It comprises a concatenation of ten 784×1 vectors, $\mathbf{g}_k$, where $k \in \{0, \ldots, 9\}$, and

$$\mathbf{g}_k = \begin{cases} \mathbf{x} \left(1 - \frac{\exp(\mathbf{w}_k^T \mathbf{x})}{\sum_{c'=0}^{9} \exp(\mathbf{w}_{c'}^T \mathbf{x})} \right) & \text{if } k = c \\ \frac{-\exp(\mathbf{w}_k^T \mathbf{x}) \mathbf{x}}{\sum_{c'=0}^{9} \exp(\mathbf{w}_{c'}^T \mathbf{x})} & \text{otherwise} \end{cases}$$

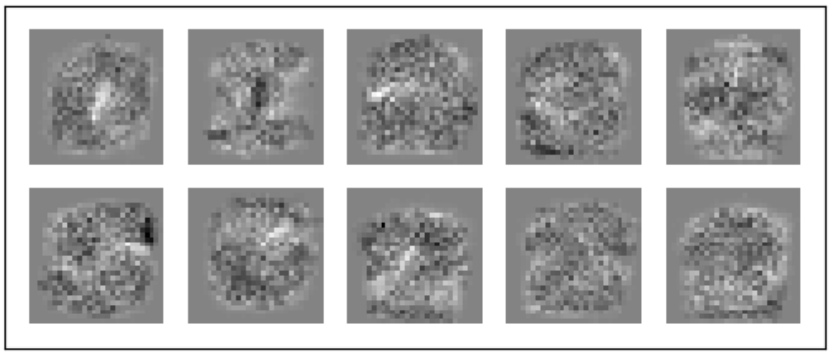
(d) The code below is preceded by the initializations in the solution to 1(b) above. The weights are shaped into a matrix, so that the 7840×1 column vector answer $\mathbf{g} = \nabla_{\mathbf{w}} \log p(c|\mathbf{x}, \mathbf{w})$ is transformed into a 784×10 matrix G.

```
def smax(x): # Improves numerical stability
    e_x = np.exp(x - np.max(x))
    return e_x / e_x.sum()

def logp (W,C,X): # See Murphy (8.35) or Bishop (4.108)
    N = len(C)
    WX = np.matmul(X,W) # 10k x 10
    Cvec = np.argmax(C,axis=1)
    logsumE = np.zeros(N)
    for n in range(N):
        logsumE[n] = scipy.misc.logsumexp(WX[n,:]) # second of two terms
    F = -np.sum(logsumE) # starting point
    for i in range(N):
        F += np.matmul(X[i,:],W[:,int(Cvec[i])])
    return F

def grad_logp (W,C,X): # See slide 40 of Lecture 5
    N = len(C)
    K = int(W.shape[1])
    G = np.zeros_like(W) # DxK
    WX = np.matmul(X,W)  # NxK
    for n in range(N):
        YNJ = smax(WX[n,:])
        for j in range(K):
            x = X[n:n+1,:].T # Dx1
            G[:,j:j+1] -= (YNJ[j] - C[n,j]) * x
    return G
```

```
D = train_images.shape[1] # Number of dimensions
K = train_labels.shape[1] # Number of classes
W = np.zeros((D,K))       # Weight matrix
for t in range(1000):
   oldW = W
   g = grad_logp(W,train_labels,train_images)
   alpha = 0.1/(t+1)
   W = W + alpha*g # Gradient descent
data.save_images(W.T,'ques3d')
```



In fact, with a better optimizer such as `scipy.optimize.fmin_cg`, the results become blurrier:



The data have been overfit to the training set, but these kinds of plots are acceptable because the question does ask to maximize the training set's likelihood, and the above parameters give a training-set classification accuracy of nearly 100%. The performance on the test set is poor in this case.

(e) The code continues from the previous part as follows:

```
def like_single (W,X):  # Choose the highest w_c.T * x for each datum
    N = len(X)
    Y = [0]*N
    for i in range(N):
        wcTx = np.matmul(W.T,X[i,:]) # Kx1
        Y[i] = np.argmax(wcTx)
    return Y

avLtrain = logp (W,train_labels,train_images)/N # proxy for avg pred. LL
avLtest = logp (W,train_labels,test_images)/M
print(round(avLtrain,2), round(avLtest,2))
```

```
Ytrain = like_single(W,train_images)
Ytest = like_single(W,test_images)
accsTrain = train_labels[np.arange(N),Ytrain]
accTrain = sum(accsTrain)/N
accsTest = test_labels[np.arange(M),Ytest]
accTest = sum(accsTest)/M
print (round(accTrain*100,2),round(accTest*100,2))
```

Results for just 1000 iterations:

	Train	Test
Average $\mathcal{L}$	-23	-1063
Predictve accuracy	90.4%	89.0%

The accuracies are higher than those from the Naïve Bayes classifier in question 1. Logistic regression learns over all pixels simultaneously, rather than independently, so it's able to extract more information from the patterns available.

The above proxies for average log-likelihoods are not useful to compare to those in question 1 in absolute terms. However, in relative terms there is an important difference: the significant drop from training set to test set in the case of logistic regression only. An advantage of the Naïve Bayes classifier is its robustness to overfitting.

Through the use of a validation set to control overfitting (not required for full marks in this question), the maximum test-set performance for this logistic regression classifier is about 91%.

Question 4

(a) There are $784 \times 10 + 9 = 7849$ parameters.

(b) There are $K!$ ways to permute. If $K = 30$ as per upcoming part (d), this is $\sim 2.7 \times 10^{32}$ ways.

(c) Several representations are acceptable here. One is to express the gradient w.r.t. θ_{cd}, but others are fine if clear.

$$\nabla_{\boldsymbol{\theta}} \log p(\mathbf{x}|\boldsymbol{\theta}, \boldsymbol{\pi}) = \nabla_{\boldsymbol{\theta}} \log \sum_{c=1}^{k} \text{Cat}(c|\boldsymbol{\pi}) \prod_{d=1}^{784} \text{Ber}(x_d|\theta_{cd}) = \nabla_{\boldsymbol{\theta}} \log \sum_{c=1}^{k} \pi_c \prod_{d=1}^{784} \theta_{cd}^{x_d} (1 - \theta_{cd})^{1-x_d}$$

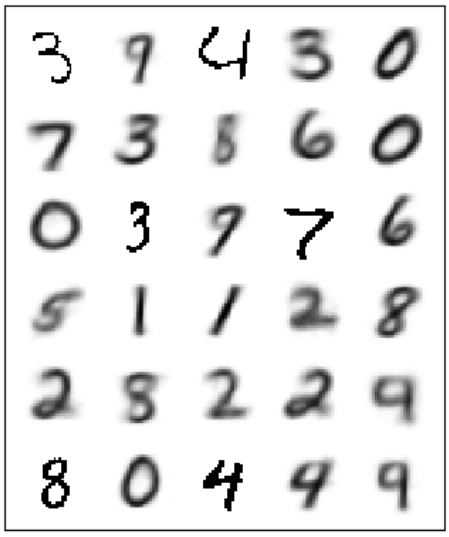
$$\nabla_{\theta_{cd}} \log p(\mathbf{x}|\boldsymbol{\theta}, \boldsymbol{\pi}) = \frac{\pi_c (-1)^{1-x_d}}{p(\mathbf{x}|\boldsymbol{\theta}, \boldsymbol{\pi})} \prod_{d' \neq d} \theta_{cd'}^{x_{d'}} (1 - \theta_{cd'})^{1-x_{d'}}$$

(d) Gradient-descent approaches can work here but have a few pitfalls. If you are able to correctly code up the derivatives etc, then well done — even if the results are poor.

Considering an alternative, EM: the code on the next page is intended to be run after the data-loading code in question 1(b).

```
K = 30 # Number of components
D = train_images.shape[1] # dimensionality of data
mu = np.random.rand(K,D)
X = train_images # N x D
for ctr in range(100):
    oldmu = mu

    # E step
    muT = mu.T # DxK
    logp = np.zeros((N,K))
    for d in range(D):
        a = np.reshape(X[:,d],(N,1))
        b = np.reshape(muT[d,:],(1,K))
        b = np.maximum(1e-3,np.minimum(b,0.999)) # Avoids log(0)
        logp += a*np.log(b) + (1-a)*np.log(1-b)
    logGamma = logp
    for n in range(N):
        logGamma[n,:] -= scipy.misc.logsumexp(logp[n,:])
    gamma = np.exp(logGamma)

    Nk = np.sum(gamma,axis=0) # 1xK, Bishop (9.57)

    # M step
    mu = np.matmul(gamma.T,X) / Nk[:,np.newaxis] # KxD, Bishop (9.59)
    # no update to pi_k, as these are fixed

    muChange = np.sum(np.sum(np.power(mu-oldmu,2)))
    print (ctr,muChange,mu[2,2])

data.save_images(mu,'ques4d')
```

The cluster means are quite different from those of the supervised model. They span the numbers from 0 to 9 at least once, but now for some digits a few different ways of writing it are represented: curly 2's vs flat-bottomed 2's, diagonal zeros vs circular zeros, and so on. One shortcoming is that six cluster centres appear to have simply collapsed on a single data point (see the six binary images among the 30 on the next page).



(e) The below code is meant to follow that of 4(d).

```
x = train_images[np.random.choice(N,20),:] # 20 x 784 array
thetaHat = mu
# Now run all but the first line of 2(f)'s code
```

The images below provide better predictions than before. With more components to choose from, the idiosyncrasies of digits can be better accommodated.