1(a) $\hat{\theta}_{mle}$: First find the log likelihood:

$$\ell(\theta) = \sum_{i=1}^{N} \log(x^{(i)}, c^{(i)} \mid \theta, \pi)$$

$$= \sum_{i=1}^{N} \log\left[ p(c^{(i)}|\pi) \prod_{j=1}^{784} p(x_j^{(i)} \mid c^{(i)}, \theta_{j c^{(i)}}) \right]$$

$$= \sum_{i=1}^{N} \left[ \log \pi_{c^{(i)}} + \sum_{j=1}^{784} x_j^{(i)} \log \theta_{j c^{(i)}} + (1 - x_j^{(i)}) \log(1 - \theta_{j c^{(i)}}) \right]$$

$$= \sum_{i=1}^{N} \left[ \log \pi_{c^{(i)}} + \sum_{k=0}^{9} \sum_{j=1}^{784} (c^{(i)} = k) \left[ x_j^{(i)} \log \theta_{jk} + (1 - x_j^{(i)}) \log(1 - \theta_{jk}) \right] \right]$$

Taking derivative wrt $\theta_{jk}$ and set it to 0:

$$\frac{\partial \ell}{\partial \theta_{jk}} = \sum_{i=1}^{N} \mathbb{1}(c^{(i)} = k) \left( \frac{x_j^{(i)}}{\theta_{jk}} - \frac{1 - x_j^{(i)}}{1 - \theta_{jk}} \right) = 0$$

$$\sum_{i=1}^{N} \mathbb{1}(c^{(i)} = k) \left( x_j^{(i)} - x_j^{(i)} \theta_{jk} - \theta_{jk} + x_j^{(i)} \theta_{jk} \right) = 0$$

$$\hat{\theta}_{jk} = \frac{\sum_{i=1}^{N} \mathbb{1}(c^{(i)} = k)\, x_j^{(i)}}{\sum_{i=1}^{N} \mathbb{1}(c^{(i)} = k)}$$

Therefore: $$\hat{\theta}_{MLE} = \frac{\sum_{i=1}^{N} \mathbb{1}(c^{(i)} = k) \cdot x_j^{(i)}}{\sum_{i=1}^{N} \mathbb{1}(c^{(i)} = k)} \quad \text{for } k = 0, \ldots, 9 \text{ and } j = 1, \ldots, 784$$

$\hat{\pi}_{MLE}$: First find the log likelihood

$$\ell(\pi) = \sum_{i=1}^{N} \log \prod_{j=0}^{9} \pi_j^{t_j^{(i)}} = \sum_{i=1}^{N} \sum_{j=0}^{9} t_j^{(i)} \log \pi_j$$

$$= \sum_{i=1}^{N} \left[ \sum_{j=0}^{8} t_j^{(i)} \log \pi_j + t_9^{(i)} \log\left(1 - \sum_{k=0}^{8} \pi_k\right) \right]$$

Taking the derivative and setting it to 0:

$$\frac{\partial \ell}{\partial \pi_j} = \sum_{i=1}^{N} \frac{t_j^{(i)}}{\pi_j} - \frac{t_9^{(i)}}{1 - \sum_{k=0}^{8} \pi_k} = \sum_{i=1}^{N} \frac{t_j^{(i)}}{\pi_j} - \frac{t_9^{(i)}}{\pi_9} = 0$$

$$\sum_{i=1}^{N} \frac{t_j^{(i)}}{\pi_j} = \sum_{i=1}^{N} \frac{t_9^{(i)}}{\pi_9}$$

$$\frac{\hat{\pi}_j}{\hat{\pi}_9} = \sum_{i=1}^{N} \frac{t_j^{(i)}}{t_9^{(i)}} \quad (*)$$

since we know $\hat{\pi}_1 + \hat{\pi}_2 + \ldots + \hat{\pi}_9 = 1$

we also know $\frac{\hat{\pi}_1}{\hat{\pi}_9} + \frac{\hat{\pi}_2}{\hat{\pi}_9} + \ldots + \frac{\hat{\pi}_9}{\hat{\pi}_9} = \frac{1}{\hat{\pi}_9}$

so: $$\hat{\pi}_9 = \sum_{i=1}^{N} \frac{t_9^{(i)}}{\sum_{j=0}^{9} t_j^{(i)}}$$

However, since we chose to isolate for $\hat{\pi}_9$ arbitrarily, meaning we could have chosen $\hat{\pi}_j$ for $j = 0, \ldots, 8$ and get the same we can say with loss of generality:

$$\hat{\pi}_j = \sum_{i=1}^{N} \frac{t_9^{(i)}}{\sum_{j=0}^{9} t_j^{(i)}} = \frac{\sum_{i=1}^{N} t_j^{(i)}}{N}$$

so:

$$\hat{\pi}_{MLE} = \frac{\sum_{i=1}^{N} t_j^{(i)}}{N} \quad \text{for } j = 0, \ldots, 9$$

For part (a), the following is my implementation:

```
def train_mle_estimator(train_images, train_labels):
    """ Inputs: train_images, train_labels
        Returns the MLE estimators theta_mle and pi_mle"""
    labels_sum = np.sum(train_labels, axis=0)
    pi_mle = labels_sum/train_labels.shape[0]
    theta_mle = np.dot(train_images.T,train_labels)/labels_sum
    return theta_mle, pi_mle
```

The $\hat{\pi}$ is calculated by taking the total number of each class in the dataset and divide it by N because from our derivation, if we move the summation over N into the denominator, we can observe that is just the total number of images.

The $\hat{\theta}$ is calculated by first taking the dot product between the images and labels as suggested by the formula we derived from above, the hint also suggests that it should represent the number of $j^{th}$ pixel appears in class c over # of class c in dataset. In which we can perform with numpy in one single line.

1 (b) By Bayes Rule:

$$\log p(t \mid x, \theta, \pi) = \log \frac{p(c \mid \theta, \pi)\, p(x \mid c, \theta, \pi)}{\sum_{k=0}^{9} p(c_k \mid \theta, \pi)\, p(x \mid c_k, \theta, \pi)}$$

$$= \log \left[ \frac{p(c \mid \pi) \prod_{j=1}^{784} p(x_j \mid c, \theta, \pi)}{\sum_{k=0}^{9} p(c_k \mid \theta, \pi)\, p(x \mid c_k, \theta, \pi)} \right]$$

$$= \log \left[ \frac{\pi_c \prod_{j=1}^{784} \theta_{jc}^{x_j} (1-\theta_{jc})^{1-x_j}}{\sum_{k=0}^{9} \pi_{c_k} \prod_{j=1}^{784} \theta_{jc_k}^{x_j} (1-\theta_{jc_k})^{1-x_j}} \right]$$

$$= \log \pi_c + \sum_{j=1}^{784} \left( x_j \log \theta_{jc} + (1-x_j) \log (1-\theta_{jc}) \right) - \log \sum_{k=0}^{9} \pi_{c_k} \prod_{j=1}^{784} \theta_{jc_k}^{x_j} (1-\theta_{jc_k})^{1-x_j}$$

$$= \log \pi_c + \sum_{j=1}^{784} \left( x_j \log \theta_{jc} + (1-x_j) \log (1-\theta_{jc}) \right) - \log \sum_{k=0}^{9} \exp \left( \log \pi_{c_k} \prod_{j=1}^{784} \theta_{jc_k}^{x_j} (1-\theta_{jc_k})^{1-x_j} \right)$$

$$= \log \pi_c + \sum_{j=1}^{784} \left( x_j \log \theta_{jc} + (1-x_j) \log (1-\theta_{jc}) \right) - \log \sum_{k=0}^{9} \exp \left[ \log \pi_{c_k} + \sum_{j=1}^{784} x_j \log \theta_{jc_k} + (1-x_j) \log (1-\theta_{jc_k}) \right]$$

Thus we can calculate above with numpy operations.

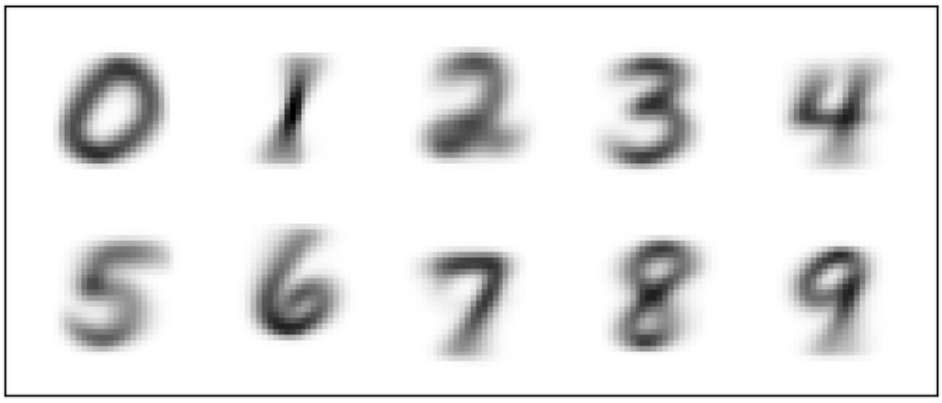
The following is my log likelihood implementation:

```
def log_likelihood(images, theta, pi):
    """ Inputs: images, theta, pi
        Returns the matrix 'log_like' of loglikehoods over the input images where
    log_like[i,c] = log p (c |x^(i), theta, pi) using the estimators theta and pi.
    log_like is a matrix of num of images x num of classes (60000 x 10)
    Note that log likelihood is not only for c^(i), it is for all possible c's."""
    numerator = np.log(pi) + np.dot(images, np.log(theta)) + np.dot(1. - images, np.log(1. - theta))
    denominator = logsumexp(numerator, axis=1)
    log_like = (numerator.transpose() - denominator).transpose()
    return log_like
```

I used logsumexp function from the scipy library to help calculate the “denominator”. I noticed that since the denominator uses the numerator, I calculated the numerator first then deduct the denominator for each column of numerator as suggested from the derivation above.

1 (c):
The average log likelihood is nan, I believe that this is due to high number of 0s in my log likelihood function since most of theta are 0s which is again due to the fact there is only one entry with 1 in the labels. Therefore, when taking log of an array with lots of 0s, we would get lots of nan. Thus averaging as nan.

1 (d):
MLE $\hat{\theta}$ as 10 images:



1 (e) $\hat{\theta}_{MAP}$: First find the log likelihood

we know for MAP: $p(\theta \mid x, c, \pi) \propto p(\theta)\, p(x, c \mid \theta, \pi)$

we also know $\theta \sim Beta(3,3)$

$$l(\theta) = \log p(\theta)\, p(x, c \mid \theta, \pi)$$

$$= \log(\theta^2(1-\theta)^2) + \sum_{i=1}^{N} \log\left( p(c^{(i)} \mid \pi) \prod_{j=1}^{784} \theta_{j c^{(i)}}^{x_j^{(i)}} (1-\theta_{j c^{(i)}})^{1-x_j^{(i)}} \right)$$

$$= 2\theta + 2(1-\theta) + \sum_{i=1}^{N} \left[ \log \pi_{c^{(i)}} + \sum_{j=1}^{784} x_j^{(i)} \log \theta_{j c^{(i)}} + (1-x_j^{(i)}) \log(1-\theta_{j c^{(i)}}) \right]$$

$$= 2\theta + 2(1-\theta) + \sum_{i=1}^{N} \left[ \log \pi_{c^{(i)}} + \sum_{k=0}^{9} \sum_{j=1}^{784} \mathbb{1}(c^{(i)}=k) \left( x_j^{(i)} \log \theta_{jk} + (1-x_j^{(i)}) \log(1-\theta_{jk}) \right) \right]$$

Taking derivative wrt $\theta_{jk}$ and setting it to 0:

$$\frac{\partial l}{\partial \theta_{jk}} = \frac{2}{\theta_{jk}} - \frac{2}{1-\theta_{jk}} + \sum_{i=1}^{N} \mathbb{1}(c^{(i)}=k) \left( \frac{x_j^{(i)}}{\theta_{jk}} - \frac{1-x_j^{(i)}}{1-\theta_{jk}} \right) = 0$$

$$2 - 2\theta_{jk} - 2\theta_{jk} + \sum_{i=1}^{N} \mathbb{1}(c^{(i)}=k) \left( x_j^{(i)} - x_j^{(i)}\theta_{jk} - \theta_{jk} + x_j^{(i)}\theta_{jk} \right) = 0$$

$$4\theta_{jk} + \sum_{i=1}^{N} \mathbb{1}(c^{(i)}=k)\, \theta_{jk} = 2 + \sum_{i=1}^{N} \mathbb{1}(c^{(i)}=k)\, x_j^{(i)}$$

$$\hat{\theta}_{jk} = \frac{2 + \sum_{i=1}^{N} \mathbb{1}(c^{(i)}=k)\, x_j^{(i)}}{4 + \sum_{i=1}^{N} \mathbb{1}(c^{(i)}=k)}$$

Therefore we can say:

$$\hat{\theta}_{map} = \frac{2 + \sum_{i=1}^{N} \mathbb{1}(c^{(i)}=k)\, x_j^{(i)}}{4 + \sum_{i=1}^{N} \mathbb{1}(c^{(i)}=k)} \quad \text{for } k = 0, \ldots, 9 \text{ and } j = 1, \ldots, 784$$

Here is my implementation of the map estimators:

```
def train_map_estimator(train_images, train_labels):
    """ Inputs: train_images, train_labels
        Returns the MAP estimators theta_map and pi_map"""
    labels_sum = np.sum(train_labels, axis=0)
    pi_map = labels_sum/train_labels.shape[0]
    theta_map = (2.+np.dot(train_images.T,train_labels))/(4.+labels_sum)
    return theta_map, pi_map
```

Since $\hat{\pi}$ is the same for MAP, I copied the method from (a) but added the "shift" (2 and 4 in the numerator and denominator) for $\hat{\theta}$ as suggested in the above derivation.

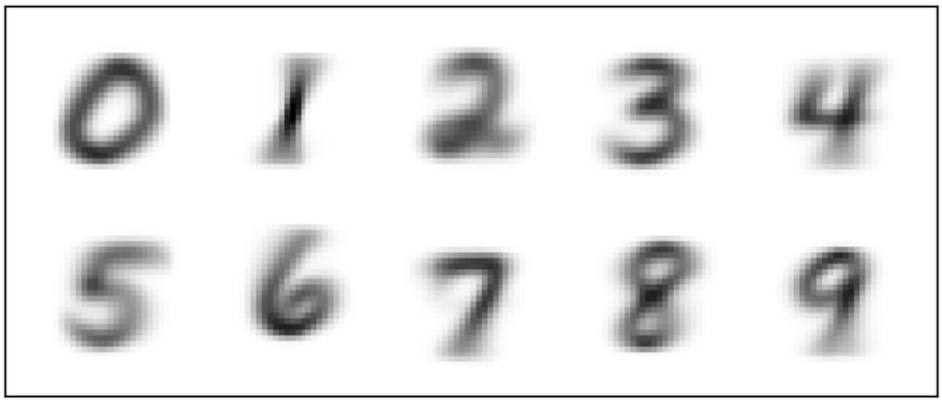
1 (f):
Here is my implementation for calculating the prediction and accuracy:

```
def predict(log_like):
    """ Inputs: matrix of log likelihoods
    Returns the predictions based on log likelihood values"""
    predictions = (log_like == log_like.max(axis=1, keepdims=1)).astype(int)
    return predictions
def accuracy(log_like, labels):
    """ Inputs: matrix of log likelihoods and 1-of-K labels
    Returns the accuracy based on predictions from log likelihood values"""
    pred = predict(log_like)
    accuracy = np.mean(pred == labels)
    return accuracy
```

The average log likelihood and accuracy results are as follows:

```
Average log-likelihood for MAP is  -3.357063137860285
Training accuracy for MAP is  0.9670433333333334
Test accuracy for MAP is  0.9632
```

1 (g):
MAP $\hat{\theta}$ as 10 images:



Notice there is no visible difference compare to MLE $\hat{\theta}$

2 (a):
True, since by our assumption of a Bayes model, each pixels are sampled independently.

2 (b):
False, because once we marginalized over c, the joint probability of $x_i$ and $x_j$ does not equal to the product of probabilities of $x_i$ and $x_j$. The derivation is as follows:

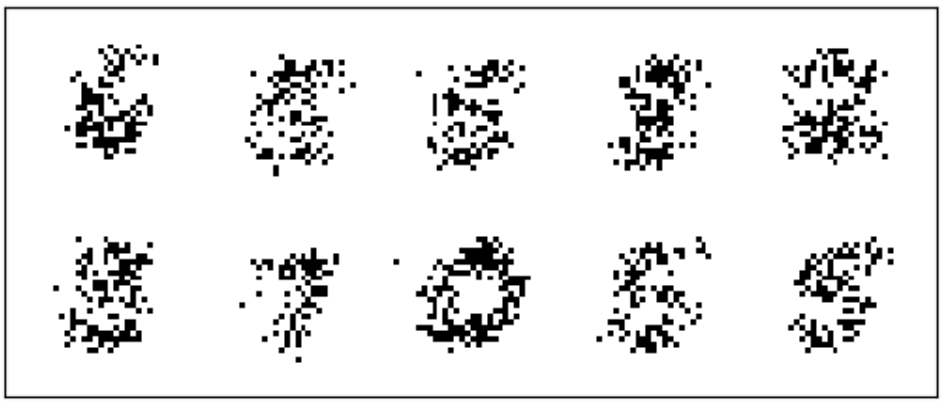
$$p(x_i, x_j) = \sum_{c'} p(x_i|c')p(x_j|c')$$

$$p(x_i)p(x_j) = \sum_{c'} p(x_i|c') \sum_{c'} p(x_j|c')$$

2 (c):
Here is my image sampler implementation:

```
def image_sampler(theta, pi, num_images):
    """ Inputs: parameters theta and pi, and number of images to sample
    Returns the sampled images"""
    sampled_images = np.zeros((num_images, 784))
    for i in range(num_images):
        choice = np.random.choice(a=10,p=pi)
        print(choice)
        sampled_images[i] = np.random.binomial(1, theta[:,choice]).reshape((1, 784))
    return sampled_images
```

and here are my results:



I printed out the choices which are:

6 5 5 8 2
3 7 0 5 5

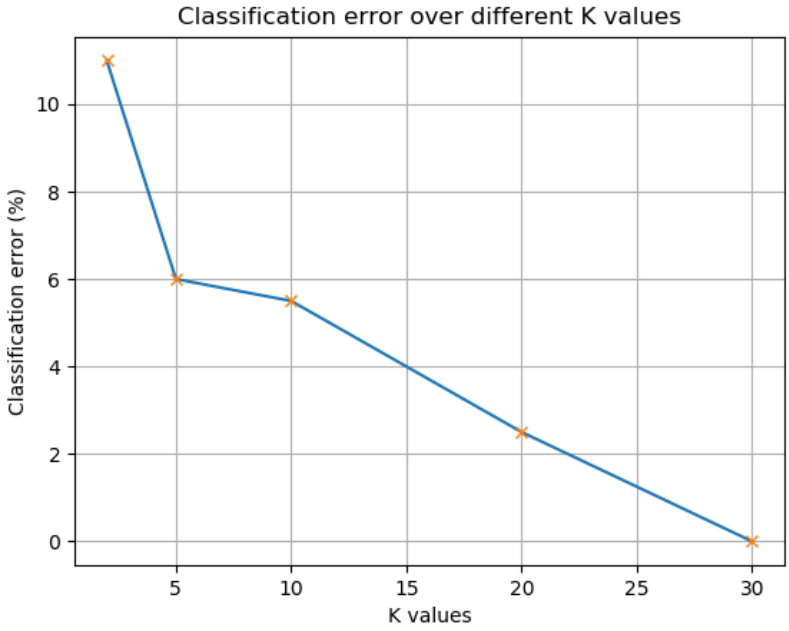
It is actually very close to what a normal writing of those choices looks like.

3 (a):

```
import scipy
import numpy as np
import matplotlib.pyplot as plt
from utils import load_data, run_knn

def main():
    train, valid, test, trainTarget, validTarget, testTarget = load_data("digits.npz")
    kRange = [2,5,10,20,30]
    m = np.mean(train , axis=0)
    train_centered = train - np.tile(m, (train.shape[0], 1))
    C = np.cov(train_centered.T)
    U,S,V = np.linalg.svd(C)
    plot_data = []
    for k in kRange:
        train_recon = train_centered.dot(U[:,:k])

        knn_prediction = run_knn(1, train_recon,trainTarget, np.dot(valid, U[:, :k]))

        acc = np.mean(knn_prediction == validTarget)
        print("validation accuracy with k={}: {}%".format(k,acc*100))
        plot_data.append(1-acc)
    # Test data performance
    knn_prediction = run_knn(1, train_recon,trainTarget, np.dot(test, U[:, :30]))
    acc = np.mean(knn_prediction == testTarget)
    print("test accuracy with k={}:".format(k), acc*100, "%")
    # Plotting
    plt.plot(kRange, np.array(plot_data)*100)
    plt.plot(kRange, np.array(plot_data)*100, 'x')
    plt.title("Classification error over different K values")
    plt.xlabel("K values")
    plt.ylabel("Classification error (%)")
        plt.grid()
        plt.savefig("q3.png")
    plt.show()

if __name__ == '__main__':
    main()
```

Note that I took code for knn from A2 starter and also used code from pca.py from tutorial on PCA.

Here is my classification graph:




We can see that with higher K values, the classification error for validation is significantly lower, to a point that is seemingly overfitting the data. However, in later parts, the accuracy on test data is actually relatively high.

3 (b):
Given the above graph, I chose to use K = 30 since it gives the lowest classification error thus theoretically it should predicts the best. Another reason for picking K = 30 is also because I believe that the first 30 eigenvectors captures the most important variance of the data. In fact with a 0 classification error, the first 30 captures all 100% of the variance in data.

3 (c):
I have achieved a very higher accuracy with K = 30:

```
test accuracy with k=30: 97.5 %
```