CSC311 A3, Winter 2019
Timothy Lee (leetim13)

1. a)

$$\begin{aligned}
L(\theta) &= p(x, c|\theta, \pi) \\
&= p(c|\theta, \pi)p(x|c, \theta, \pi) \\
&= p(c|\pi)\prod_{j=1}^{784} p(x_j|c, \theta_{jc}) \\
&= \pi_c \prod_{j=1}^{784} \theta_{jc}^{x_j}(1-\theta_{jc})^{1-x_j}
\end{aligned}$$

Taking logarithms for MLE,

$$\begin{aligned}
l(\theta) &= \log(L(\theta)) \\
&= \sum_{j=1}^{N}(\log \pi_j^{t_j^{(i)}} + \sum_{j=1}^{784}(x_j^{(i)} \log\theta_{jc} + (1-x_j^{(i)})\log(1-\theta_{jc}))
\end{aligned}$$

Calculating partial derivative to find $argmax$,

$$\frac{\partial l(\theta_{jc})}{\partial \theta_{jc}} = \sum_{i=1}^{N}(\frac{x_j^{(i)}}{\theta_{jc}} - \frac{1-x_j^{(i)}}{1-\theta_{jc}}))$$

Setting derivative to 0,

$$\sum_{i=1}^{N}\mathbb{I}(c^{(i)} = c)\theta_{jc} = \sum_{i=1}^{N}\mathbb{I}(c^{(i)} = c)x_j^{(i)}$$

Therefore,

$$\theta_{jc}^{MLE} = \frac{\sum_{i=1}^{N}\mathbb{I}(C^{(i)} = c)x_j^{(i)}}{\sum_{i=1}^{N}\mathbb{I}(c^{i} = c)}$$

Note that the indicator function $\mathbb{I}(c^{(i)} = c)x_j^{(i)} = \mathbb{I}(c^{(i)} = c \textbf{ and } x_j^{(i)} = 1)$. In other words, it is use to denote the number of times $j-th$ pixel $= 1$ in the $c-th$ class.

Now, we will find the MLE estimate for prior π.

First (by hint), we know that $L(\theta) = p(t^i|\pi) = \prod_{j=0}^{9} \pi_j^{t_j^{(i)}}$, where $\sum_{j=0}^{9} \pi_j = 1$. Applying logarithms,

$$l(\theta) = t_j^{(i)} \sum_{j=0}^{9} \log(\pi_j)$$

Since (by hint) we can denote $\pi_9 = 1 - \sum_{i=0}^{8} \pi_i$. We can re-write $l(\theta)$ as

$$l(\theta) = \sum_{i=0}^{N} (t_9^{(i)} \log(1 - \sum_{j=0}^{8} \pi_j)^{t_9^{(i)}} + \sum_{j=0}^{8} t_j^{(i)} \log(\pi_j)$$

Taking derivative with respect to π_j:

$$\frac{\partial l(\theta)}{\partial \pi} = \sum_{i=0}^{N} (t_j^{(i)} \frac{1}{\pi_j} + t_9^{(i)} - \frac{1}{1 - \sum_{j=0}^{8} \pi_j}), \text{ for } j = 0, .., 8$$

Setting derivative to 0, we have

$$\sum_{i=0}^{N} \frac{t_j^{(i)}}{\pi_j} = \sum_{i=0}^{N} \frac{t_9^{(i)}}{\pi_9}$$

Then,

$$\begin{aligned} \frac{\hat{\pi_j}}{\hat{\pi_9}} &= \frac{\sum_{i=0}^{N} t_j^{(i)}}{\sum_{i=0}^{N} t_9^{(i)}} \\ \hat{\pi_j} &= \frac{\sum_{i=0}^{N} t_j^{(i)}}{\sum_{i=0}^{N} t_9^{(i)}} \cdot \hat{\pi_9} \end{aligned}$$

Since $\pi_9 = 1 - \sum_{j=0}^{8} \pi_j$ (by hint) and $\sum_{j=0}^{9} t_j^{(i)} = 1$ for each class label $t^{(i)}$ (by definition of 1-of-10 encoded class), we can conclude that

$$\begin{aligned} \hat{\pi_9} &= 1 - \sum_{j=0}^{8} \hat{\pi_j} \\ &= 1 - \sum_{j=0}^{8} \frac{\sum_{i=0}^{N} t_j^{(i)}}{\sum_{i=0}^{N} t_9^{(i)}} \cdot \hat{\pi_9} \qquad \text{(by definition of } \hat{\pi_j} \text{ above)} \\ &= 1 - \frac{\sum_{i=0}^{N} (1 - t_9^{(i)})}{\sum_{i=0}^{N} t_9^{(i)}} \cdot \hat{\pi_9} \qquad \text{(since } \sum_{j=0}^{9} t_j^{(i)} = 1 \text{)} \\ &= \frac{\sum_{i=0}^{N} t_9^{(i)}}{N} \qquad \text{(by re-arranging)} \end{aligned}$$

Therefore,

$$
\begin{aligned}
\pi_j^{MLE} &= \frac{\sum_{i=0}^{N} t_j^{(i)}}{\sum_{i=0}^{N} t_9^{(i)}} \cdot \frac{\sum_{i=0}^{N} t_9^{(i)}}{N} \qquad \text{(by substituion)} \\
&= \frac{\sum_{i=0}^{N} t_j^{(i)}}{N} \\
&= \frac{\sum_{i=0}^{N} \mathbb{I}(t_c^{(i)} = t_c)}{N}
\end{aligned}
$$

Again, the indicator function is used to denote the number of times $j-th$ pixel $= 1$ in the $c-th$ class and N is the total number of training samples.

b) We can denote $L(\theta)$ as

$$p(c|x,\theta,\pi) = \frac{p(c|\pi)p(x|c,\theta)}{p(x|\theta,\pi)}$$

Then, the log-likelihood, $l(\theta)$ would be

$$
\begin{aligned}
\log p(c|x,\theta,\pi) &= \log(p(c|\pi)) + \log(p(x|c,\theta)) - \log(p(x|\theta,\pi)) \\
&= \log(p(c|\pi)) + \log(\prod_{j=1}^{784} p(x_j|c,\theta_{jc}) + \log(\sum_{j=0}^{9} p(x|c=j,\theta,\pi)p(c=j))
\end{aligned}
$$

Hence, $\log p(t|x,\theta,\pi)$ can be written as

$$= \sum_{j=0}^{9} t_j^{(i)} \log \pi_j + \sum_{j=1}^{784} (x_j \log \theta_{jc} + (1-x_j)\log(1-\theta_{jc}) - \log(\sum_{j=0}^{9} \pi_j \prod_{j=1}^{784} \theta_{jc}^{x_j}(1-\theta_{jc})^{(1-x_j)})$$

c) We encounter a divide by 0 error when performing logarithm.

```
C:/Users/hwtim/Desktop/311a3/naive_bayes.py:129: RuntimeWarning: divide by zero encountered in log
  print(np.log(theta).shape)
```

Figure 1: Error Message

d)

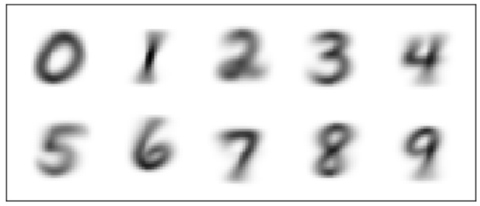

Figure 2: MLE estimator

e) For derving the *Maximum A posterior Proability* (*MAP*) given $\theta_{jc} \sim \beta(3,3)$, we will first derive the pdf of this prior beta distribution,

$$f(\theta_{jc}) = \frac{1}{B(3,3)} \theta_{jc}^{(3-1)} (1-\theta_{jc})^{(3-1)}$$

Ignoring beta normalizing constant (as per question) and simplifying,

$$f(\theta_{jc}) = \theta_{jc}^{2}(1-\theta_{jc})^{2}$$

Now applying Baye's Theorem,

$$\begin{aligned} p(\theta_{jc}|x_j, c, \pi) &\propto p(\theta_{jc}) p(x_j, c|\theta_{jc}, \pi) \\ &\propto \theta_{jc}^2 (1-\theta_{jc})^2 \pi_c^i \prod_{i=1}^{784} \theta_{jc}^{x_{ij}} (1-\theta_{jc})^{(1-x_{ij})} \end{aligned}$$

Taking logarithms,

$$l(\theta) = 2\log(\theta_{jc}) + 2\log(1-\theta_{jc}) + \sum_{i=1}^{784} x_{ij} \log\theta_{jc} + (1-\theta_{jc}) \log(1-x_{ij}) + k$$

, where k is a constant factor.
Again, I will use $\mathbb{I}(C^i = c)$ to denote $x_j = 1$ in the c-th class.

Taking derivatives to find *argmax* of θ,

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

Setting derivative to 0,

$$-2\theta_{jc} + (2 - 2\theta_{jc}) + \sum_{i=1}^{N} \mathbb{I}(C^i = c)(x_j^{(i)}(1-\theta_{jc}) - (1-x_j^{(i)})\theta_{jc}) = 0$$

Re-arranging,

$$-4\theta_{jc} + 2 + \sum_{i=1}^{N} \mathbb{I}(C^i = c)(x_j^{(i)} - x_j^{(i)}\theta_{jc} - \theta_{jc} + x_j^{(i)}\theta_{jc}) = 0$$

Finally, we get,

$$\theta_{jc}^{MAP} = \frac{\sum_{i=1}^{N} \mathbb{I}(C^i = c)x_j^{(i)} + 2}{\sum_{i=1}^{N} \mathbb{I}(C^i = c) + 4}$$

f)

```
Average log-likelihood for MLE is  nan
Average log-likelihood for MAP is  -3.3570631378602687
Training accuracy for MAP is  0.8352166666666667
Test accuracy for MAP is  0.816
```

Figure 3: Accuracy and average log-likelihood

g)

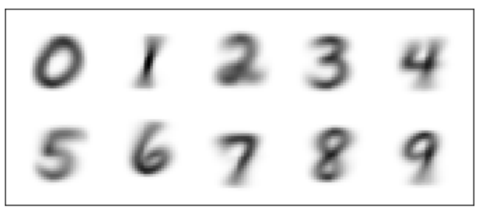

Figure 4: Theta MAP

```python
from __future__ import absolute_import
from __future__ import print_function
from future.standard_library import install_aliases
install_aliases()
import numpy as np
import os
import gzip
import struct
import array
import matplotlib.pyplot as plt
import matplotlib.image
from urllib.request import urlretrieve
from scipy.special import logsumexp

def download(url, filename):
    if not os.path.exists('data'):
        os.makedirs('data')
    out_file = os.path.join('data', filename)
    if not os.path.isfile(out_file):
        urlretrieve(url, out_file)

def mnist():
    base_url = 'http://yann.lecun.com/exdb/mnist/'

    def parse_labels(filename):
        with gzip.open(filename, 'rb') as fh:
            magic, num_data = struct.unpack(">II", fh.read(8))
            return np.array(array.array("B", fh.read()), dtype=np.uint8)

    def parse_images(filename):
        with gzip.open(filename, 'rb') as fh:
            magic, num_data, rows, cols = struct.unpack(">IIII", fh.read(16))
            return np.array(array.array("B", fh.read()), dtype=np.uint8).reshape(num_data, rows, cols)

    for filename in ['train-images-idx3-ubyte.gz',
                     'train-labels-idx1-ubyte.gz',
                     't10k-images-idx3-ubyte.gz',
                     't10k-labels-idx1-ubyte.gz']:
        download(base_url + filename, filename)

    train_images = parse_images('data/train-images-idx3-ubyte.gz')
    train_labels = parse_labels('data/train-labels-idx1-ubyte.gz')
    test_images = parse_images('data/t10k-images-idx3-ubyte.gz')
    test_labels = parse_labels('data/t10k-labels-idx1-ubyte.gz')
    return train_images, train_labels, test_images[:1000], test_labels[:1000]


def load_mnist():
    partial_flatten = lambda x: np.reshape(x, (x.shape[0], np.prod(x.shape[1:])))
    one_hot = lambda x, k: np.array(x[:, None] == np.arange(k)[None, :], dtype=int)
    train_images, train_labels, test_images, test_labels = mnist()
    train_images = (partial_flatten(train_images) / 255.0 > .5).astype(float)
    test_images = (partial_flatten(test_images) / 255.0 > .5).astype(float)
    train_labels = one_hot(train_labels, 10)
    test_labels = one_hot(test_labels, 10)
    N_data = train_images.shape[0]
    return N_data, train_images, train_labels, test_images, test_labels
```

Figure 5: Code Part 1

```python
def plot_images(images, ax, ims_per_row=5, padding=5, digit_dimensions=(28, 28),
                cmap=matplotlib.cm.binary, vmin=None, vmax=None):
    """Images should be a (N_images x pixels) matrix."""
    N_images = images.shape[0]
    N_rows = np.int32(np.ceil(float(N_images) / ims_per_row))
    pad_value = np.min(images.ravel())
    concat_images = np.full(((digit_dimensions[0] + padding) * N_rows + padding,
                             (digit_dimensions[1] + padding) * ims_per_row + padding), pad_value)
    for i in range(N_images):
        cur_image = np.reshape(images[i, :], digit_dimensions)
        row_ix = i // ims_per_row
        col_ix = i % ims_per_row
        row_start = padding + (padding + digit_dimensions[0]) * row_ix
        col_start = padding + (padding + digit_dimensions[1]) * col_ix
        concat_images[row_start: row_start + digit_dimensions[0],
                      col_start: col_start + digit_dimensions[1]] = cur_image
        cax = ax.matshow(concat_images, cmap=cmap, vmin=vmin, vmax=vmax)
        plt.xticks(np.array([]))
        plt.yticks(np.array([]))
    return cax


def save_images(images, filename, **kwargs):
    fig = plt.figure(1)
    fig.clf()
    ax = fig.add_subplot(111)
    plot_images(images, ax, **kwargs)
    fig.patch.set_visible(False)
    ax.patch.set_visible(False)
    plt.savefig(filename)


def train_mle_estimator(train_images, train_labels):
    """ Inputs: train_images, train_labels
        Returns the MLE estimators theta_mle and pi_mle"""
    theta = np.matmul(np.transpose(train_images),train_labels) # 784 x 10 array
    sum_labels = train_labels.sum(axis=0) #sum of indicator function
    theta_mle = theta/sum_labels

    N = len(train_images)
    pi_mle = np.ones(N)
    pi_mle = pi_mle.dot(train_labels) * 1/N
    return theta_mle, pi_mle


def train_map_estimator(train_images, train_labels):
    """ Inputs: train_images, train_labels
        Returns the MAP estimators theta_map and pi_map"""

    # YOU NEED TO WRITE THIS PART
    N = len(train_images)
    theta = np.matmul(np.transpose(train_images),train_labels) # 784 x 10 array
    sum_labels = train_labels.sum(axis=0) #sum of indicator function
    pi_map = sum_labels/N
    theta_map = (theta + 2)/(4 + sum_labels) #using beta_prior
    return theta_map, pi_map
```

Figure 6: Code Part 2

```python
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
    Note that log likelihood is not only for c^(i), it is for all possible c's."""
    N_pi = len(pi)
    N = len(images)
    shape =(N, N_pi)
    log_like = np.zeros(shape)
    log_p_x = logsumexp(np.log(pi) + np.dot(images, np.log(theta)) + np.dot((1. - images), np.log(1. - theta)), axis=1)
    for c in range(N_pi):
        log_like[:, c] = (np.dot(images, np.log((theta.T)[c])) + np.dot((1. - images), np.log(1. - (theta.T)[c]))
        - log_p_x) + np.log(pi[c]) #by derivation
    return log_like


def predict(log_like):
    """ Inputs: matrix of log likelihoods
    Returns the predictions based on log likelihood values"""

    # YOU NEED TO WRITE THIS PART
    predictions = np.argmax(log_like, axis =1)
    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"""
    N = len(labels)
    one_of_k = labels[np.arange(N),log_like.argmax(1)]
    accuracy = sum(one_of_k)/N
    return accuracy


def image_sampler(theta, pi, num_images):
    """ Inputs: parameters theta and pi, and number of images to sample
    Returns the sampled images"""

    # YOU NEED TO WRITE THIS PART
    l = len(theta)
    shape = num_images, l
    sampled_images = np.zeros(shape) #new array of given shape and type, filled with zeros.
    for i in range(num_images):
        c = np.random.choice(10, p=pi) #by hint
        sampled_images[i] = np.random.binomial(1, p=theta[:,c])  #by hint
    save_images(sampled_images, "samples.png")
    return sampled_images
```

Figure 7: Code Part 3

```python
def main():
    N_data, train_images, train_labels, test_images, test_labels = load_mnist()

    # Fit MLE and MAP estimators
    theta_mle, pi_mle = train_mle_estimator(train_images, train_labels)
    theta_map, pi_map = train_map_estimator(train_images, train_labels)

    # Find the log likelihood of each data point
    loglike_train_mle = log_likelihood(train_images, theta_mle, pi_mle)
    loglike_train_map = log_likelihood(train_images, theta_map, pi_map)


    avg_loglike_mle = np.sum(loglike_train_mle * train_labels) / N_data
    avg_loglike_map = np.sum(loglike_train_map * train_labels) / N_data
#
    print("Average log-likelihood for MLE is ", avg_loglike_mle)
    print("Average log-likelihood for MAP is ", avg_loglike_map)
#
    train_accuracy_map = accuracy(loglike_train_map, train_labels)
    loglike_test_map = log_likelihood(test_images, theta_map, pi_map)
    test_accuracy_map = accuracy(loglike_test_map, test_labels)
#
    print("Training accuracy for MAP is ", train_accuracy_map)
    print("Test accuracy for MAP is ", test_accuracy_map)

    # Plot MLE and MAP estimators
#     print(theta_mle.T.shape)
    save_images(theta_mle.T, 'mle.png')
    save_images(theta_map.T, 'map.png')

    # Sample 10 images
    sampled_images = image_sampler(theta_map, pi_map, 10)
    save_images(sampled_images, 'sampled_images.png')


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

Figure 8: Code Part 4

2. a) (Trivially) True, under the given assumptions of the naiive bayes model where we assume the independence of x_i, x_j, which might not be the case in the real world.

   b) False, we will show that x_i and x_j are not independent by showing $p(x_i, x_j) \neq p(x_i)p(x_j)$
   When marginalizing over c, we have

$$p(x_i, x_j) = \sum p(x_i, x_j|c) = \sum p(x_i|c)p(x_j|c)$$

   , while

$$p(x_i)p(x_j) = \sum p(x_i|c) \sum p(x_j|c)$$

   c)

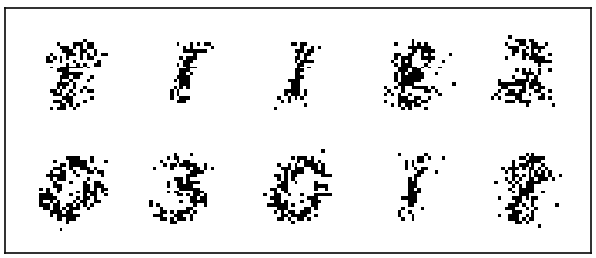

Figure 9: Random Image Samples

```python
def image_sampler(theta, pi, num_images):
    """ Inputs: parameters theta and pi, and number of images to sample
    Returns the sampled images"""

    # YOU NEED TO WRITE THIS PART
    l = len(theta)
    shape = num_images, l
    sampled_images = np.zeros(shape) #new array of given shape and type, filled with zeros.
    for i in range(num_images):
        c = np.random.choice(10, p=pi) #by hint
        sampled_images[i] = np.random.binomial(1, p=theta[:,c])  #by hint
    save_images(sampled_images, "samples.png")
    return sampled_images

```

Figure 10: Code for image sampler

3. First, I will provide some intuition of PCA in terms of graphs and images (as described, but not required by the question).

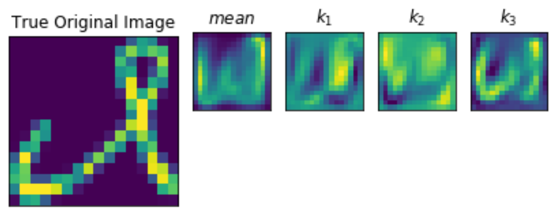

Figure 11: First 3 PCA of digit '2'

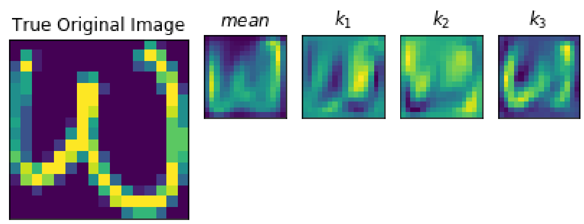

Figure 12: First 3 PCA of digit '3'

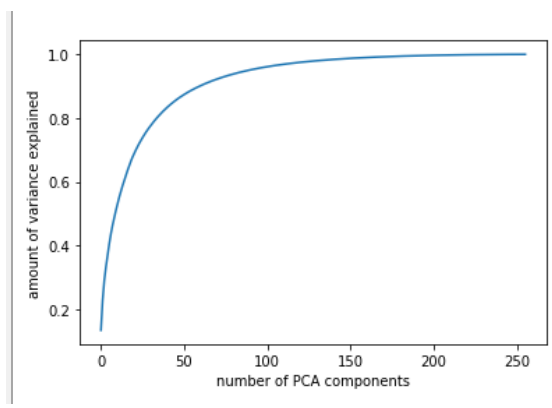

Figure 13: Graph of number of PCA components vs variance explained

a)

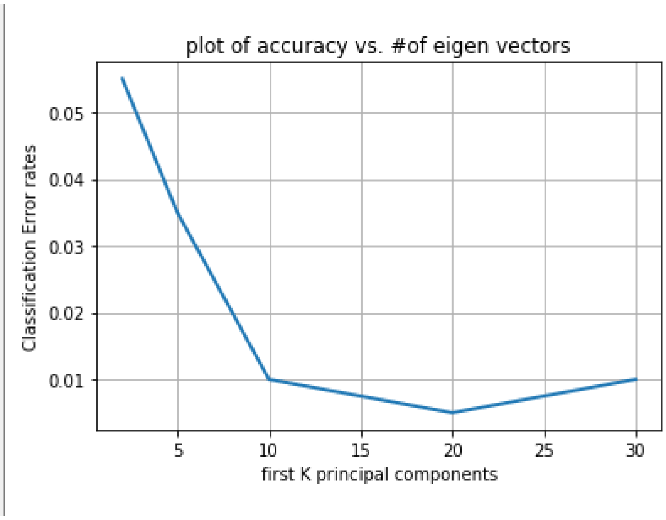

Figure 14: Graph of validation set classification error rates versus number of eigenvectors, K

b) Based on the graph, we could conclude that $K = 20$ seems like a reasonable number of eigen vector to choose from, since the classification error rate on the validation set is the lowest (global minimum), implying a higher classification accuracy. We can deduce that as K continues to increase from 2, the classification error reduces up until $K = 20$, where the error appears to go up again. This is most likely due to overfitting of our training data (which causes the performance of the training data set to be almost perfect, but the performance to be a lot worse in our validation data).

c) As argued before, $K = 20$ seems like a logical choice, which has about an error rate of around 1%. This is the optimum/lowest error rate achieved in the validation set for K in which we avoid over-fitting.

Figure 15: Error for K=20

```python
import numpy as np
from sklearn.decomposition import PCA #used to build intuition
import seaborn as sns
import matplotlib.pyplot as plt

def load_data(filename, load2=True, load3=True):
  """Loads data for 2's and 3's
  Inputs:
    filename: Name of the file.
    load2: If True, load data for 2's.
    load3: If True, load data for 3's.
  """
  assert (load2 or load3), "Atleast one dataset must be loaded."
  data = np.load(filename)
  if load2 and load3:
    inputs_train = np.hstack((data['train2'], data['train3']))
    inputs_valid = np.hstack((data['valid2'], data['valid3']))
    inputs_test = np.hstack((data['test2'], data['test3']))
    target_train = np.hstack((np.zeros((1, data['train2'].shape[1])), np.ones((1, data['train3'].shape[1]))))
    target_valid = np.hstack((np.zeros((1, data['valid2'].shape[1])), np.ones((1, data['valid3'].shape[1]))))
    target_test = np.hstack((np.zeros((1, data['test2'].shape[1])), np.ones((1, data['test3'].shape[1]))))
  else:
    if load2:
      inputs_train = data['train2']
      target_train = np.zeros((1, data['train2'].shape[1]))
      inputs_valid = data['valid2']
      target_valid = np.zeros((1, data['valid2'].shape[1]))
      inputs_test = data['test2']
      target_test = np.zeros((1, data['test2'].shape[1]))
    else:
      inputs_train = data['train3']
      target_train = np.zeros((1, data['train3'].shape[1]))
      inputs_valid = data['valid3']
      target_valid = np.zeros((1, data['valid3'].shape[1]))
      inputs_test = data['test3']
      target_test = np.zeros((1, data['test3'].shape[1]))

  return inputs_train.T, inputs_valid.T, inputs_test.T, target_train.T, target_valid.T, target_test.T

#Intution of displaying PCA compoenents vs images
def show(g, imshape, i, j, x, title=None):
        ax = fig.add_subplot(g[i, j], xticks=[], yticks=[])
        ax.imshow(x.reshape(imshape), interpolation='nearest')
        if title:
            ax.set_title(title, fontsize=12)
```

Figure 16: Code

```python
def plot_pca_components(x, coefficients=None, mean=0, components=None,
                        imshape=(16, 16), n_components=8, fontsize=12,
                        show_mean=True):
    '''Building intuition by viewing top k PCA
    '''
    if coefficients is None:
        coefficients = x
    if components is None:
        components = np.eye(len(coefficients), len(x))
    mean = np.zeros_like(x) + mean
    fig = plt.figure(figsize=(1.2 * (5 + n_components), 1.2 * 2))
    g = plt.GridSpec(2, 4 + bool(show_mean) + n_components, hspace=0.3)
    show(g, imshape, slice(2), slice(2), x, "True Original Image")
    approx = mean.copy()
    counter = 2
    if show_mean:
        show(g, imshape, 0, 2, np.zeros_like(x) + mean, r'$mean$')
        show(g, imshape,1, 2, approx)
        counter += 1
    for i in range(n_components):
        approx = approx + coefficients[i] * components[i]
        show(g, imshape,0, i + counter, components[i], r'$k_{0}$'.format(i + 1))
        show(g, imshape,1, i + counter, approx,
            r"${0:.2f} \cdot c_{1}$".format(coefficients[i], i + 1))
        if show_mean or i > 0:
            plt.gca().text(0, 1.05, '$+$', ha='right', va='bottom',
                           transform=plt.gca().transAxes, fontsize=fontsize)
    show(g, imshape, slice(2), slice(-2, None), approx, "Approx")
    return fig

def plot_digits(data):
    fig, axes = plt.subplots(10, 10, figsize=(10, 4),
                             subplot_kw={'xticks':[], 'yticks':[]},
                             gridspec_kw=dict(hspace=0.1, wspace=0.1))
    for i, ax in enumerate(axes.flat):
        ax.imshow(data[i].reshape(16, 16), #reshape into 16x16 in order to be displayed
                  cmap='binary', interpolation='nearest')
```

Figure 17: Code

```
def first_k_components(training_data, k):
    '''
    Plot of first k_components vs eigen values
    '''
    mean = np.mean(training_data, axis =0)
    num_repeated = training_data.shape[0], 1
    centered_data = training_data - np.tile(mean, num_repeated) #subtract the mean of training data
    data_T = centered_data.T
    covariance_matrix = np.cov(data_T)
    eigen_values_cov , eigen_vectors_cov = np.linalg.eig(covariance_matrix )
    eigen_values_cov = eigen_values_cov[:: -1]
    eigen_vectors_cov = eigen_vectors_cov[:: -1]
    plt.figure()
    length = np.arange(0.0 , len(eigen_values_cov ) , 1) #[0,1,...,256]
    plt.plot(length, eigen_values_cov )
    plt.xlabel ("number of eigenvectors")
    plt.ylabel ("accuracy")
    plt.title ("plot of eigenvalues of covarience")
    plt.grid ( True )
    eigen_values = eigen_values_cov[:k]
    eigen_vectors =  eigen_vectors_cov[:k ,:]
#    print(eigen_values.shape) #(10,)
#    print(eigen_vectors.shape) #(10, 256)
    return eigen_values, eigen_vectors , mean

def one_nn_classifier(train_data, train_labels, valid_data, k=1) :
    N = len(valid_data)
    shape = N, 1
    valid_labels = np.zeros(shape) #initialize empty
    train_data_N = len(train_data)

    for i in range (N):
        min_index = -1
        min_value = np.inf
        for j in range (train_data_N):
            euclidean_distance = np.linalg.norm(valid_data[i]- train_data[j])
            if euclidean_distance < min_value :
                min_value = euclidean_distance
                min_index = j
        valid_labels[i] = train_labels[min_index]
    return valid_labels
```

Figure 18: Code

```python

def extract_eigen_features(training_data):
    mean = np.mean(training_data , axis =0)
    num_repeated = training_data.shape[0], 1
    centered_data = training_data - np.tile(mean, num_repeated)
    data_T = centered_data.T
    covariance_matrix = np.cov(data_T)
    eigen_values_cov , eigen_vectors_cov = np.linalg.eig(covariance_matrix )
    #    print(eigen_values_cov.shape) #(10,)
#     print(eigen_vectors_cov.shape) #(10, 256)
    sorted_eigen_values = eigen_values_cov.argsort()[:: -1] #vector of sorted eigen values asc
    eigen_values = eigen_values_cov[sorted_eigen_values]
    eigen_vectors = eigen_vectors_cov [:,sorted_eigen_values]
    return eigen_values , eigen_vectors , mean

def accuracy (prediction_value, target_value):
    return np.mean(target_value==prediction_value)

def train_model_pca(given_K , inputs_train , inputs_valid , target_train , target_valid):
    accuracy_list = []
    eigen_values , eigen_vectors , mean = extract_eigen_features(inputs_train)
    for k in given_K :
        code_vectors = eigen_vectors[: ,: k]
#           print(top_k_vector)
#           top_k_value = value [: k]
        num_repeated_training = (inputs_train.shape[0] , 1)
#           print(num_repeated_training.shape) #600
        centered_training_data = inputs_train - np.tile(mean, num_repeated_training )
        num_repeated_valid = (inputs_valid.shape[0] , 1)
#           print(num_repeated_valid.shape) #200
        centered_valid_data = inputs_valid - np .tile(mean, num_repeated_valid)

        #projection onto the low-dimensional space
        low_dim_space_valid = np.dot(centered_valid_data,  code_vectors)
        low_dim_space_train = np.dot(centered_training_data,  code_vectors)

        #using 1-NN classifier on K dimensional features
        low_dim_space_target = one_nn_classifier(low_dim_space_train , target_train , low_dim_space_valid )
        accuracy_ = accuracy(low_dim_space_target, target_valid)
        error = 1 - accuracy_
        accuracy_list.append(error)
    plt.figure()
    plt.grid(True)
    plt.plot(given_K , accuracy_list)
    plt.xlabel('first K principal components')
    plt.ylabel('Classification Error rates')
    plt.title ('plot of accuracy vs. #of eigen vectors')
    return accuracy_list

```

Figure 19: Code

```python
if __name__ == '__main__':
    inputs_train, inputs_valid, inputs_test, target_train, target_valid, target_test = load_data("digits.npz")
#     print(inputs_train.shape)
#     print(inputs_valid.shape)
#     print(target_train.shape)
#     print(target_valid.shape)

    given_K = [2 , 5, 10 , 20 , 30]
#     view_eig_vector_images (10 , top_k_vector , mean )
    accuracy_list = train_model_pca(given_K, inputs_train , inputs_valid , target_train , target_valid)

#     print (accuracy_k)
    best_K = 20 #after selection
    accuracy_list = train_model_pca([best_K], inputs_train , inputs_test , target_train , target_test)
    print ("Error of K=" + str(best_K) + " = " + str(accuracy_list[0]))

    pca = PCA().fit(inputs_train) #only used to build intuition
    plt.plot(np.cumsum(pca.explained_variance_ratio_))
    plt.xlabel('number of PCA components')
    plt.ylabel('amount of variance explained')
    pca = PCA(n_components=20)
    Xproj = pca.fit_transform(inputs_train)
    fig = plot_pca_components(inputs_train[155], Xproj[155],
                              pca.mean_, pca.components_)
    plot_digits(inputs_train)
    plt.show
```

Figure 20: Code