Q1

a) $$P(x|c,\theta) = \prod_{i=1}^{N} p(x_j^{(i)}|c,\theta_{jc}) = \prod_{i=1}^{N} \theta_{jc}^{x_j^{(i)}} (1-\theta_{jc})^{(1-x_j^{(i)})}$$

$$\log P(x|c,\theta) = \sum_{i=1}^{N} \log\left(\theta_{jc}^{x_j^{(i)}}(1-\theta_{jc})^{(1-x_j^{(i)})}\right)$$

$$= \sum_{i=1}^{N} \left(x_j^{(i)} \log \theta_{jc} + (1-x_j^{(i)}) \log(1-\theta_{jc})\right)$$

$$\frac{\partial}{\partial \theta_{jc}} \log P(x|c,\theta) = \sum_{i=1}^{N}\left(\frac{x_j^{(i)}}{\theta_{jc}} + \frac{1-x_j^{(i)}}{(-1)(1-\theta_{jc})}\right) = \sum_{i=1}^{N} \frac{x_j^{(i)}}{\theta_{jc}} + \sum_{i=1}^{N} \frac{1-x_j^{(i)}}{\theta_{jc}-1}$$

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

$$= \frac{1}{\theta_{jc}} N_{j,x_j=1} + \frac{1}{\theta_{jc}-1} N_{j,x_j=0}.$$

where $N_{j,x_j=1}$ is the total number of images with pixel x_j classified as 1.

$N_{j,x_j=0}$ is the total number of images with pixel x_j classified as 0.

Setting the derivative to zero:

$$0 = \frac{1}{\hat{\theta}_{jc}} N_{j,x_j=1} + \frac{1}{\hat{\theta}_{jc}-1} N_{j,x_j=0}.$$

$$= \frac{\hat{\theta}_{jc}-1}{\hat{\theta}_{jc}\cancel{(\hat{\theta}_{jc}-1)}} N_{j,x_j=1} + \frac{\hat{\theta}_{jc}}{\hat{\theta}_{jc}\cancel{(\hat{\theta}_{jc}-1)}} N_{j,x_j=0}.$$

$$= \hat{\theta}_{jc}\left(N_{j,x_j=1} + N_{j,x_j=0}\right) - N_{j,x_j=1}$$

$$N_{j,x_j=1} = \hat{\theta}_{jc} \cdot N_j$$

$$\hat{\theta}_{jc} = \frac{N_{j,x_j=1}}{N_j}$$

$$p(t^{(i)}|\pi) = \prod_{j=0}^{9} \pi_j^{t_j^{(i)}}$$

$$\log P(t^{(i)}|\pi) = \sum_{j=0}^{9} \log\left(\pi_j^{t_j^{(i)}}\right)$$

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

$$= \sum_{j=0}^{8} t_j^{(i)} \log \pi_j \ +$$

$$t_9^{(i)} \log \left(1 - \sum_{j=0}^{8} \pi_j\right)$$

$$\frac{\partial}{\partial \pi_j} \log P(t^{(i)}|\pi) = \sum_{j=0}^{8} \frac{t_j^{(i)}}{\pi_j} - t_9^{(i)} \frac{1}{1 - \sum_{j=0}^{8} \pi_j}$$

$$= \frac{\sum_{j=0}^{8} t_j^{(i)}}{\sum_{j=0}^{8} \pi_j} - \frac{t_9^{(i)}}{1 - \sum_{j=0}^{8} \pi_j}$$

Setting derivative to zero:

$$\frac{t_9^{(i)}}{1 - \sum_{j=0}^{8} \pi_j} = \frac{\sum_{j=0}^{8} t_j^{(i)}}{\sum_{j=0}^{8} \pi_j}$$

$$t_9^{(i)} \cdot \sum_{j=0}^{8} \pi_j = \sum_{j=0}^{8} t_j^{(i)} \cdot \left(1 - \sum_{j=0}^{8} \pi_j\right)$$

$$\frac{\sum_{j=0}^{8} \pi_j}{1-\sum_{j=0}^{8} \pi_j} = \frac{\sum_{j=0}^{8} t_j^{(i)}}{t_q^{(i)}}$$

$$t_q^{(i)} \cdot \sum_{j=0}^{8} \pi_j = \sum_{j=0}^{8} t_j^{(i)} - \left(\sum_{j=0}^{8} t_j^{(i)} \cdot \sum_{j=0}^{8} \pi_j\right)$$

$$\sum_{j=0}^{q} t_j^{(i)} \cdot \sum_{j=0}^{8} \pi_j = \sum_{j=0}^{8} t_j^{(i)}$$

$$\sum_{j=0}^{8} \pi_j = \frac{\sum_{j=0}^{8} t_j^{(i)}}{\sum_{j=0}^{q} t_j^{(i)}}$$

$$\pi_q = 1 - \sum_{j=0}^{8} \pi_j = \frac{t_q^{(i)}}{\sum_{j=0}^{q} t_j^{(i)}}$$

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

b) $p(t|x,\theta,\pi) \propto \prod_{i=1}^{N} p(t^{(i)}|\pi)\, p(x^{(i)}|t^{(i)},\theta)$

$$\mathcal{L}(\theta) = \prod_{i=1}^{N} p(t^{(i)}|\pi) \prod_{j=1}^{784} \left(\theta_{jt^{(i)}}\right)^{x_j^{(i)}} \left(1-\theta_{jt^{(i)}}\right)^{(1-x_j^{(i)})}$$

$$\log \mathcal{L}(\theta) = \sum_{i=1}^{N}\left(\log \pi_{t^{(i)}} + \sum_{j=1}^{784}\left(x_j^{(i)} \log \theta_{jt^{(i)}} + (1-x_j^{(i)}) \log(1-\theta_{jt^{(i)}})\right)\right)$$

for a single image:

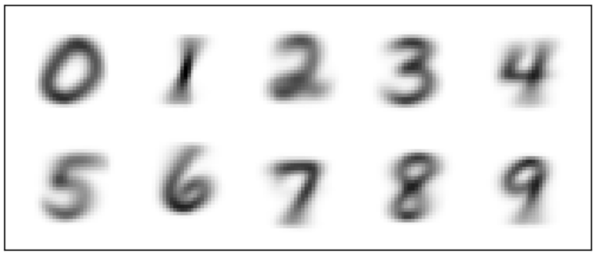
$$\log \mathcal{L}(\theta) = \log \pi_{t^{(i)}} + \sum_{j=1}^{784}\left(x_j^{(i)} \log \theta_{jt^{(i)}} + (1-x_j^{(i)}) \log(1-\theta_{jt^{(i)}})\right)$$

c) ./naive_bayes.py:173: RuntimeWarning: divide by zero encountered in log

```
  sum_img_pixels = np.matmul(images,np.log(theta))
./naive_bayes.py:173: RuntimeWarning: invalid value encountered in matmul
  sum_img_pixels = np.matmul(images,np.log(theta))
Average log-likelihood for MLE is  nan
```

According to the RuntimeWarning, division by zero will happen when calculating the average log-likelihood for MLE, which results in nan.

d)Plotting MLE estimators for 10 classes



e) $\hat{\theta}_{MAP} = \underset{\theta}{\operatorname{argmax}} \log p(\theta) + \log p(D|\theta)$ $\qquad \theta \sim \text{Beta}(3,3)$

$$L(\theta) = \log p(\theta_{jc}) + \sum_{i=1}^{N} \log\left(p(x_j^{(i)} \mid c, \theta_{jc})\right)$$

$$= \log\left(\theta_{jc}^{2}(1-\theta_{jc})^{2}\right) + \sum_{i=1}^{N} \log\left(\theta_{jc}^{x_j^{(i)}}(1-\theta_{jc})^{(1-x_j^{(i)})}\right)$$

$$= 2\log\theta_{jc} + 2\log(1-\theta_{jc}) + \sum_{i=1}^{N} x_j^{(i)} \log\theta_{jc} + \sum_{i=1}^{N}(1-x_j^{(i)})\log(1-\theta_{jc})$$

$$= 2\log\theta_{jc} + 2\log(1-\theta_{jc}) + N_{j,x_j=1}\log\theta_{jc} + N_{j,x_j=0}\log(1-\theta_{jc})$$

$$= \log\theta_{jc}\cdot\left(2+N_{j,x_j=1}\right) + \log(1-\theta_{jc})\cdot(2+N_{j,x_j=0})$$

$$\frac{\partial L(\theta)}{\partial \theta_{jc}} = \frac{2+N_{j,x_j=1}}{\theta_{jc}} + \frac{2+N_{j,x_j=0}}{\theta_{jc}-1}$$

setting the derivative to zero:

$$\frac{2+N_{j,x_j=1}}{\theta_{jc}} = \frac{2+N_{j,x_j=0}}{1-\theta_{jc}}$$

$$2-2\theta_{jc}+N_{j,x_j=1}-\theta_{jc}\cdot N_{j,x_j=1}$$

$$\parallel$$

$$2\theta_{jc} + \theta_{jc}\cdot N_{j,x_j=0}$$

$$2 + N_{j,x_j=1} = 4\theta_{jc} + \theta_{jc} \cdot \left(N_{j,x_j=1} + N_{j,x_j=0}\right)$$

$$2 + N_{j,x_j=1} = \theta_{jc}\left(4 + N_j\right)$$

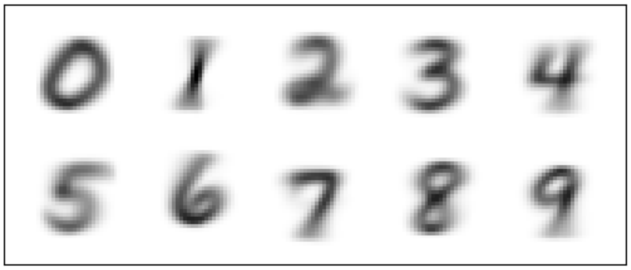
$$\hat{\theta}_{jc} = \frac{N_{j,x_j=1} + 2}{N_j + 4}$$

f)

Average log-likelihood for MAP is -100.39600482438387

```
Training accuracy for MAP is  0.6915833333333333
Test accuracy for MAP is  0.684
```

g)Plotting MAP estimators for 10 classes



Code for Q1:

```
def train_mle_estimator(train_images, train_labels):
    """ Inputs: train_images, train_labels
        Returns the MLE estimators theta_mle and pi_mle"""

    # YOU NEED TO WRITE THIS PART
    #train_images.shape = (60000,784)
    #train_labels.shape = (60000,10)
    num_classes = 10
    num_img = train_images.shape[0]
    pixel_per_img = train_images.shape[1]
    train_data = np.where(train_images>0.5,1.0,0.0)

    num_pixel_in_class = np.matmul(train_labels.T,train_data)
    img_in_class = np.sum(train_labels.T,axis=1)
    img_in_class= img_in_class[:,np.newaxis]
    for i in range(0, num_classes):
        num_pixel_in_class[i] = num_pixel_in_class[i]/img_in_class[i][0]
    theta_mle = num_pixel_in_class.T
    # print("theta_mle",theta_mle.shape)

    img_in_class_vec = np.sum(train_labels,axis=0)
    sum_target_all_img = np.sum(np.sum(train_labels,axis=1),axis=0)
    pi_mle = img_in_class_vec/sum_target_all_img

    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
    num_classes = 10
    num_img = train_images.shape[0]
    pixel_per_img = train_images.shape[1]
    train_data = np.where(train_images>0.5,1.0,0.0)

    num_pixel_in_class = np.matmul(train_labels.T,train_data)
    img_in_class = np.sum(train_labels.T,axis=1)
    img_in_class= img_in_class[:,np.newaxis]
    for i in range(0, num_classes):
        num_pixel_in_class[i] = (2+num_pixel_in_class[i])/(4+img_in_class[i][0])
    theta_map = num_pixel_in_class.T

    img_in_class_vec = np.sum(train_labels,axis=0)
    sum_target_all_img = np.sum(np.sum(train_labels,axis=1),axis=0)
    pi_map = img_in_class_vec/sum_target_all_img

    return theta_map, pi_map


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."""

    # YOU NEED TO WRITE THIS PART
    num_classes = 10
    sum_img_pixels = np.matmul(images,np.log(theta))
    log_like = np.zeros((num_classes,images.shape[0]))

    for c in range(0,num_classes):
        pi_vec = np.full((1,images.shape[0]),np.log(pi[c]))
        log_like[c] = sum_img_pixels.T[c]+pi_vec

    log_like = log_like.T

    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"""

    # YOU NEED TO WRITE THIS PART
    predictions = predict(log_like)
    num_images = log_like.shape[0]
    img_idx = np.array(range(num_images))
    labels_correspondence = labels[img_idx,predictions]
    matched = np.sum(labels_correspondence)
    accuracy = matched/num_images
    return accuracy
```

Q2

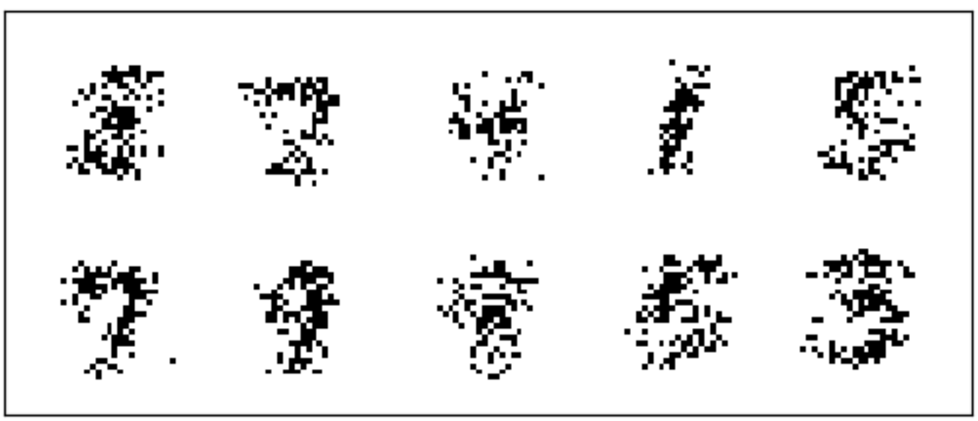
a) True : Naive Assumption : Naive Bayes assumes that features x_i are conditionally independent given the class c.

b) False: $p(x_i, x_j) = \sum_c p(x_i, x_j | c)$

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

Since $p(x_i, x_j) \neq p(x_i)p(x_j)$,
then they are not independent.

c)



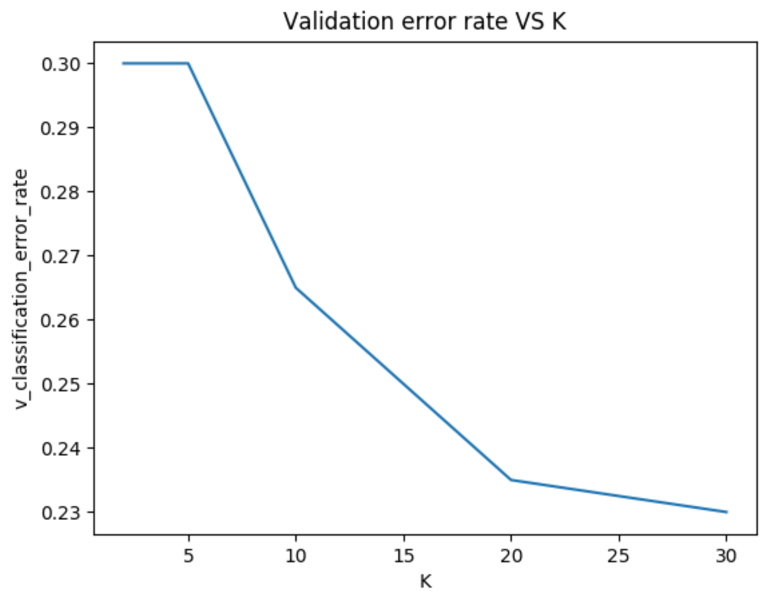
Code for Q2:

```
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
    theta_index = np.random.choice(len(pi),num_images,p=pi)
    sampled_images = np.random.binomial(1,p=theta.T[theta_index])
    return sampled_images
```

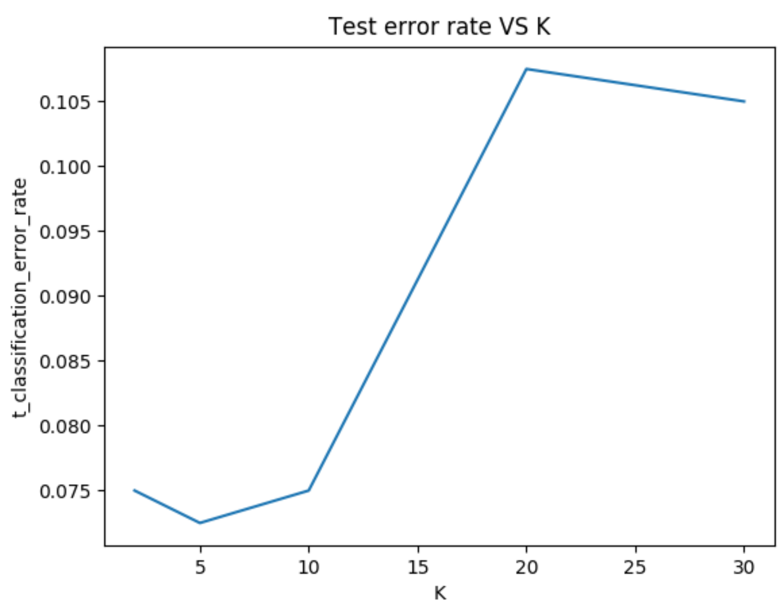
Q3

a)



b) Based on the plot above, I would choose number of eigenvectors as 30 since it has the lowest classification error rate.

c)



Code for Q3:

```
import numpy as np
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

def l2_distance(a, b):
    """Computes the Euclidean distance matrix between a and b.
    """
    print("a",a.shape)
    print("b",b.shape)
    if a.shape[0] != b.shape[0]:
```

```
        raise ValueError("A and B should be of same dimensionality")

    # aa = np.sum(a**2, axis=0)
    aa_sqr = np.square(a)
    bb_sqr = np.square(b)
    aa = np.array(np.sum(aa_sqr, axis=0))[0]
    bb = np.array(np.sum(bb_sqr, axis=0))[0]
    ab = np.dot(a.T, b)

    return np.sqrt(aa[:, np.newaxis] + bb[np.newaxis, :] - 2*ab)

def run_knn(k, train_data, train_labels, valid_data):
    """Uses the supplied training inputs and labels to make
    predictions for validation data using the K-nearest neighbours
    algorithm.

    Note: N_TRAIN is the number of training examples,
          N_VALID is the number of validation examples,
          and M is the number of features per example.

    Inputs:
        k:            The number of neighbours to use for classification
                      of a validation example.
        train_data:   The N_TRAIN x M array of training
                      data.
        train_labels: The N_TRAIN x 1 vector of training labels
                      corresponding to the examples in train_data
                      (must be binary).
        valid_data:   The N_VALID x M array of data to
                      predict classes for.

    Outputs:
        valid_labels: The N_VALID x 1 vector of predicted labels
                      for the validation data.
    """
    print("valid_data 2",valid_data.shape)
    print("train_data 2",train_data.shape)

    dist = l2_distance(valid_data.T, train_data.T)
    nearest = np.argsort(dist, axis=1)[:,:k]

    train_labels = train_labels.reshape(-1)
    valid_labels = train_labels[nearest]

    # note this only works for binary labels
    valid_labels = (np.mean(valid_labels, axis=1) >= 0.5).astype(np.int)
    valid_labels = valid_labels.reshape(-1,1)

    return valid_labels

def OneNN(train_inputs,valid_inputs,test_inputs,train_targets,valid_targets,test_targets):

    k_given = [1]
```

```
    v_classification_rate = 0
    t_classification_rate = 0

    for k in k_given:
        print("valid_data1",valid_inputs.shape)
        print("train_data1",train_inputs.shape)
        valid_outputs = run_knn(k,train_inputs,train_targets,valid_inputs)

        if len(valid_outputs) == len(valid_targets):
            count = 0
            for i in range(0, len(valid_outputs)):
                if valid_outputs[i] == valid_targets[i]:
                    count += 1
            v_classification_rate=count/len(valid_outputs)

    for k in k_given:
        test_outputs = run_knn(k,train_inputs,train_targets,test_inputs)

        if len(test_outputs) == len(test_targets):
            count = 0
            for i in range(0, len(test_outputs)):
                if test_outputs[i] == test_targets[i]:
                    count += 1
            t_classification_rate=count/len(test_outputs)
    return v_classification_rate,t_classification_rate

def PCA(data, num_principal_comp):
    num_images = data.shape[0]
    num_pixels = data.shape[1]
    mean = np.mean(data,axis=0)
    data_centered = data - np.tile(mean,(num_images,1))
    data_cov = np.cov(data_centered.T)
    eigen_values, eigen_vectors = np.linalg.eig(np.mat(data_cov))
    asc_sorted_eigval_index = np.argsort(eigen_values)
    desc_sorted_eigval_index = asc_sorted_eigval_index[:-(num_principal_comp+1):-1]
    eigvec_chosen = eigen_vectors[:,desc_sorted_eigval_index]

    low_d_data = np.matmul(data_centered,eigvec_chosen)
    reconstructed_data = np.matmul(low_d_data, eigvec_chosen.T) + mean
    return reconstructed_data, low_d_data

if __name__ == '__main__':
    inputs_train, inputs_valid, inputs_test, target_train, target_valid, target_test = load_data('./
digits.npz')
    pca_lst = [2,5,10,20,30]
    valid_rates = []
    test_rates = []
    for pca in pca_lst:
        reconstructed_data_train, low_d_data_train = PCA(inputs_train, pca)
        reconstructed_data_valid, low_d_data_valid = PCA(inputs_valid, pca)
        reconstructed_data_test, low_d_data_test = PCA(inputs_test, pca)
        v_rate, t_rate = OneNN(low_d_data_train,
                               low_d_data_valid,
                               low_d_data_test,
```

```
                            target_train,
                            target_valid,
                            target_test)
        valid_rates.append(1-v_rate)
        test_rates.append(1-t_rate)
    plt.plot(pca_lst,valid_rates,label='valid_set')
    plt.xlabel("K")
    plt.ylabel("v_classification_error_rate")
    plt.title("Validation error rate VS K")
    plt.savefig("PCA validation error rate.png")

    plt.clf()
    plt.plot(pca_lst,test_rates,label='test_set')
    plt.xlabel("K")
    plt.ylabel("t_classification_error_rate")
    plt.title("Test error rate VS K")
    plt.savefig("PCA test error rate.png")
    plt.clf()
```