```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True
```

Question 1

(a)

(a) Naïve Bayes defines the joint probability of the each data point x and its class label c as follows: $p(\boldsymbol{x}, c|\boldsymbol{\theta}, \boldsymbol{\pi}) = p(c|\boldsymbol{\theta}, \boldsymbol{\pi})p(\boldsymbol{x}|c, \boldsymbol{\theta}, \boldsymbol{\pi}) = p(c|\boldsymbol{\pi}) \prod_{j=1}^{784} p(x_j|c, \theta_{jc})$, and knwoing that $p(x_j|c, \theta_{jc}) = \theta_{jc}^{x_j}(1-\theta_{jc})^{(1-x_j)}$, and $p(c|\boldsymbol{\pi}) = \pi_c$, we have:
$L(\theta) = p(x, c|\theta, \pi) = \prod_{i=1}^{N} \pi_c^i \prod_{j=1}^{784} \theta_{jc}^{x_j}(1-\theta_{jc})^{(1-x_j)}$, in this way we could derive log-likelihood $l(\theta)$:
$l(\theta) = \frac{\partial L(\theta)}{\partial \theta} = \sum_{i=1}^{N}\left(\log \pi_c^i + \sum_{j=1}^{784}\left(x_j^i \log \theta_{cj} + \left(1-x_j^i\right)\log\left(1-\theta_{cj}\right)\right)\right)$
Thus, $\hat{\theta}_{MLE} = \underset{\theta}{\operatorname{argmax}} l(\theta) \quad \text{s.t. with respect to} \quad \sum_{c=0}^{9} \pi_c = 1$.

Take derivative with respect to θ_{jc}, we have $\frac{\partial l(\theta)}{\partial \theta_{jc}} = \sum_{i=1}^{N} \mathbb{1}\left(C^{(i)} = c\right)\left(\frac{x_j^i}{\theta_{jc}} - \frac{1-x_j^i}{1-\theta_{jc}}\right) = 0$
$\Longrightarrow \sum_{i=1}^{N} \mathbb{1}\left(C^{(i)} = c\right)\left(x_j^i(1-\theta_{jc}) - \theta_{jc}(1-x_j^i)\right) = 0$
$\Longrightarrow \sum_{i=1}^{N} \mathbb{1}\left(C^{(i)} = c\right)\theta_{jc} = \sum_{i=1}^{N} \mathbb{1}\left(C^{(i)} = c\right)x_j^i$
$\Longrightarrow \hat{\theta}_{MLE} = \frac{\sum_{i=1}^{N} \mathbb{1}\left(C^{(i)}=c \wedge x_j^i=1\right)}{\sum_{n=1}^{N} \mathbb{1}\left(C^{(i)}=c\right)}$.

Now derive MLE for π. By using Lagrange multiplier, we have
$\frac{\partial l(\theta)}{\partial \pi_j} + \lambda \frac{\partial \sum \pi_j}{\partial \pi_j} = 0 \Longrightarrow \lambda = -\sum_{n=1}^{N} I\left(t^{(n)} = j\right)\frac{1}{\pi_j} \Longrightarrow \hat{\pi}_j = \frac{-\sum_{n=1}^{N} I\left(t^{(n)}=j\right)}{\lambda}$.
Apply constraint $\sum_{c=0}^{9} \pi_c = 1$, we have $\lambda = -N$. In this way, we have
$\hat{\pi}_j = \frac{\sum_{n=1}^{N} I\left(t^{(n)}=j\right)}{N}$

(b) (c)

(b) $p(t|\mathbf{x}, \boldsymbol{\theta}, \boldsymbol{\pi}) = \frac{p(\mathbf{x}, c|\boldsymbol{\theta}, \pi)}{p(\mathbf{x}|\boldsymbol{\theta}, \pi)} = \frac{p(\mathbf{x}, c|\boldsymbol{\theta}, \pi)}{\sum_{c=0}^{9} p(\mathbf{x}, c|\boldsymbol{\theta}, \pi)} = \frac{\pi_c \prod_{j=1}^{784} \theta_{jc}^{x}(1-\theta_{jc})^{1-x_j}}{\sum_{c=0}^{9} \pi_c \prod_{j=1}^{784} \theta_{jc}^{x_j}(1-\theta_{jc})^{1-x_j}}$.
$\Longrightarrow log(p(\mathbf{t}|\mathbf{x}, \boldsymbol{\theta}, \boldsymbol{\pi})) = log(\pi_c \prod_{j=1}^{784}(\theta_{jc}^{x}(1-\theta_{jc})^{1-x_j})) - log(\sum_{c=0}^{9} \pi_c \prod_{j=1}^{784}(\theta_{jc}(1-\theta_{jc})^{1-x_j})^{t_c})$
$= \log(\pi_c) + \sum_{j=1}^{784} x_j \log(\theta_{cj}) + (1-x_j)\log(1-\theta_{cj})$
$\quad - \sum_{c=0}^{9}\left(\log(\pi_c) + \sum_{j=1}^{784} x_j \log(\theta_{jc}) + (1-x_j)\log(1-\theta_{jc})\right)$
$= -\sum_{i=0, i\neq c}^{9}\left(\log(\pi_i) + \sum_{j=1}^{784}(x_j)\log(\theta_{ij}) + \log(1-\theta_{ij})(1-x_j)\right)$

Step 1 load data

```
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
from IPython.display import display, Math, Latex
```

```
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


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)

if __name__ == '__main__':
    N_data, train_images, train_labels, test_images, test_labels = load_mnist()
    print(N_data)
    print(train_images.shape)
    print(train_labels.shape)
    print(test_images.shape)
    print(test_labels.shape)
```

```
60000
(60000, 784)
(60000, 10)
(1000, 784)
(1000, 10)
```

Step 2: According to question a, complete train_mle_estimator function

```
def train_mle_estimator(train_images, train_labels):
    """ Inputs: train_images, train_labels
        Returns the MLE estimators theta_mle and pi_mle"""
    N, D = train_images.shape
    theta_mle = np.zeros((10, D))
    pi_mle = np.zeros(10)

    for i in range(10):
      pi_mle[i] = sum(train_labels[:, i]) / N

    for c in range(10):
        for d in range(D):
            theta_mle[c][d] = (1 + (np.sum(np.where((train_images[:, d] == 1) & (train_labels[:, c] == 1),
                 1, 0))))/ (2 +(np.sum(train_labels[:, c])))


    return theta_mle, pi_mle
```

Step3: Write log-likehood function according to the equation derived

```
def log_likelihood_with_loop(images, theta, pi, train_labels):
    N, D = images.shape
    log_like = np.zeros((N, 10))

    for i in range(N):
        for c in range(10):
            if train_labels[i][c] != 1:
                log_like[i][c] -= np.log(pi[c])
                for d in range(D):
                    log_like[i][c] -= images[i][d] * np.log(theta[c][d])
                    log_like[i][c] -= np.log(1 - theta[c][d]) * (1 - images[i][d])
    return log_like
```

Problem 1:

since there are so many layers of loop in this function, it would drastically slow down the running speed of the program.

```
def log_likelihood(images, theta, pi, train_labels):
    """ 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 = images.shape[0]
    C, D = theta.shape

    log_like = np.zeros((N, C))

    for i in range(N):
        bern = np.where(images[i] > 0.5, theta, 1- theta)
        temp = np.log(pi) +np.sum(np.log(bern), axis = 1)
        log_like[i] = temp - logsumexp(temp)
    return log_like

def average_log_likelihood(images, theta, pi, train_labels):
    log_like = log_likelihood(images, theta, pi, train_labels)
    average_log_like = np.average(log_like, axis = 0)
    return average_log_like

if __name__ == '__main__':
    N_data, train_images, train_labels, test_images, test_labels = load_mnist()
    theta_mle, pi_mle = train_mle_estimator(train_images, train_labels)
    avg_loglike_mle = average_log_likelihood(train_images, theta_mle, pi_mle, train_labels)
    print(avg_loglike_mle)
```

```
[-109.28662699 -181.68961943  -72.47438253  -72.9793892   -73.12914282
  -55.01175952  -94.88549624 -105.08574599  -59.50921323  -81.3683617 ]
```

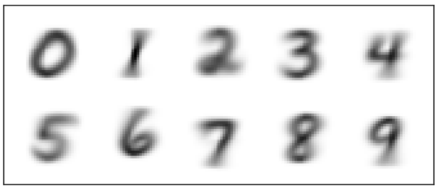
Problem 2: Notice that if we don't handle the situation for log0, the system would report "RuntimeWarning: divide by zero encountered in log", and the result would be -inf. In this case, we should make some minor edit on the log-likelihood function to avoid the sitation of log0.

(d)

Now we plot the MLE estimator $\hat{\theta}$ as 10 separate greyscale images, one for each class.

```
if __name__ == '__main__':
  N_data, train_images, train_labels, test_images, test_labels = load_mnist()
  theta_mle, pi_mle = train_mle_estimator(train_images, train_labels)

  fig = plt.figure(1)
  fig.clf()
  ax = fig.add_subplot(111)
  plot_images(theta_mle, ax)
```

```
<matplotlib.image.AxesImage at 0x7efc7a9f4668>
```



(e)

(e) Since $P(\boldsymbol{\theta}|\boldsymbol{x}, c, \pi) \propto P(\boldsymbol{\theta})P(\boldsymbol{x}, c|\boldsymbol{\theta}, \boldsymbol{\pi})$ where $\theta_{jc} \sim \text{Beta}(3,3)$, we could calculate $l(\theta)$ as:

$l(\boldsymbol{\theta}) = \log P(\boldsymbol{\theta}) + \log P(\boldsymbol{x}, c|\boldsymbol{\theta}, \boldsymbol{\pi}) + c$, where c is constant

$= \log\left(\theta_{jc}\left(1-\theta_{jc}\right)\right) + \log \pi_c^i + \sum_{j=1}^{784}\left(x_j^i \log\theta_{jc} + \left(1-x_j^i\right)\log\left(1-\theta_{cd}\right)\right) + c.$

Similar to the analysis in question a, by taking derivative with respect to θ_{jc} we have:

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

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

$$\Longrightarrow \sum_{i=1}^{N} \mathbb{1}\left(C^{(i)} = c\right)\theta_{jc} + 2\theta_{jc} = \sum_{i=1}^{N} \mathbb{1}\left(C^{(i)} = c\right)x_j^i + 1$$

$$\Longrightarrow \hat{\theta}_{MLE} = \frac{1+\sum_{i=1}^{N}\mathbb{1}\left(C^{(i)}=c \wedge x_j^i=1\right)}{2+\sum_{n=1}^{N}\mathbb{1}\left(C^{(i)}=c\right)}$$

```
def train_map_estimator(train_images, train_labels):
    """ Inputs: train_images, train_labels
        Returns the MAP estimators theta_map and pi_map"""
    N, D = train_images.shape
    theta_mle = np.zeros((10, D))
    pi_mle = np.zeros(10)

    for i in range(10):
      pi_mle[i] = sum(train_labels[:, i]) / N

    for c in range(10):
        for d in range(D):
            theta_mle[c][d] = (1 + (np.sum(np.where((train_images[:, d] == 1) & (train_labels[:, c] == 1),
                 1, 0))))/ (2 + (np.sum(train_labels[:, c])))

    return theta_mle, pi_mle
```

(f)

Similar to question c, derive average log-likelihood per data point and and the accuracy on both the training and test set.

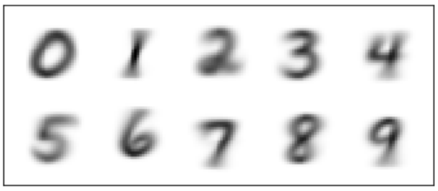
```
def predict(log_like):
    """ Inputs: matrix of log likelihoods
    Returns the predictions based on log likelihood values"""
    return np.argmax(log_like, axis=1)


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"""
    prediction = predict(log_like)
    return np.mean(prediction == np.argmax(labels, axis = 1))

if __name__ == '__main__':
    N_data, train_images, train_labels, test_images, test_labels = load_mnist()
    theta_map, pi_map = train_map_estimator(train_images, train_labels)
```

```
    theta_map, pi_map = train_map_estimator(train_images, train_labels)
    loglike_train_map = log_likelihood(train_images, theta_map, pi_map, train_labels)
    avg_loglike_map = np.sum(loglike_train_map * train_labels) / N_data
    print("Average log-likelihood for MAP is ", avg_loglike_map)
    print("LOADED TRAIN LABEL ", train_labels.shape)
    train_accuracy_map = accuracy(loglike_train_map, train_labels)
    loglike_test_map = log_likelihood(test_images, theta_map, pi_map, test_labels)
    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)
```

```
Average log-likelihood for MAP is  -3.3558565594023526
LOADED TRAIN LABEL  (60000, 10)
Training accuracy for MAP is  0.8357666666666667
Test accuracy for MAP is  0.816
```

(g)

```
if __name__ == '__main__':
  N_data, train_images, train_labels, test_images, test_labels = load_mnist()
  theta_map, pi_map = train_map_estimator(train_images, train_labels)

  fig = plt.figure(1)
  fig.clf()
  ax = fig.add_subplot(111)
  plot_images(theta_map, ax)
```



Question 2

(a)

True

Reason:

Accoding to the Naïve Bayes model assumption, this statement is true.

(b)

False

Reason:

Because $p\left(x_i, x_j\right) = \sum_c p\left(x_i, x_j|c\right) = \sum_c p\left(x_i|c\right) p\left(x_j|c\right)$ and $p\left(x_i\right) p(x_j) = \sum_c p\left(x_i|c\right) \sum_c p(x_j|c)$, we could state that $p\left(x_i, x_j\right) \neq p\left(x_i\right) p(x_j)$. Therefore, the statement that any two pixels x_i and x_j where $i \neq j$ are independent after marginalizing over c is False.

(c)

```
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.ndarray(shape = (10, 784))
    c = np.random.choice(10, 10, p=pi)
    print("random c: ", c)
    for i in range(num_images):
        sampled_images[i] = np.random.binomial(n=1, p=theta[c[i]])
    return sampled_images

if __name__ == '__main__':
    N_data, train_images, train_labels, test_images, test_labels = load_mnist()
    theta_map, pi_map = train_map_estimator(train_images, train_labels)
    sampled_images = image_sampler(theta_map, pi_map, 10)

```

```
    fig = plt.figure(1)
    fig.clf()
    ax = fig.add_subplot(111)
    plot_images(sampled_images, ax)
```

```
random c:  [6 2 3 0 6 4 1 3 9 7]
```

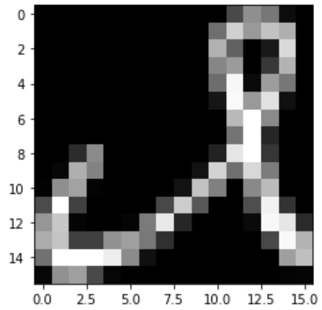
Question 3

(a)

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.decomposition import PCA
from sklearn.neighbors import KNeighborsClassifier
from scipy.spatial import distance
```

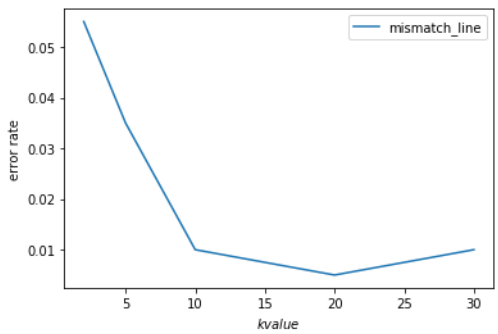
```
def load_data(filename, load2=True, load3=True):
  """Loads data for 2's and 3's
  Inputs:
    filename: Name of the file.
    load2: If True, load data for 2's.
    load3: If True, load data for 3's.
  """
  assert (load2 or load3), "Atleast one dataset must be loaded."
  data = np.load(filename)
  # print(data['train2'].shape)

  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

if __name__ == '__main__':
    path = '/content/drive/My Drive/Colab Notebooks/digits.npz'
    inputs_train, inputs_valid, inputs_test, target_train, target_valid, target_test = load_data(path)
    a = np.reshape(inputs_train[0], (16, 16))
    plt.imshow(a, cmap = "gray")
    plt.show()
```

```
(600, 256)
(400, 256)
```



```
inputs_target = np.hstack((np.zeros((300)), np.ones((300))))
valid_target = np.hstack((np.zeros((100)), np.ones((100))))

knn = KNeighborsClassifier(n_neighbors=1)

inputs_train_mean = np.mean(inputs_train, axis=0)
inputs_train_centered = inputs_train - np.tile(inputs_train_mean, (inputs_train.shape[0], 1))
inputs_valid_centered = inputs_valid - np.tile(inputs_train_mean, (inputs_valid.shape[0], 1))

cov_matrix = np.cov(inputs_train_centered.T)
eigen_value, eigen_vector = np.linalg.eig(cov_matrix)
k_list = [2, 5, 10, 20, 30]
mismatch_list = []

for item in k_list:
    eigen_matrix = eigen_vector[:, 0: item]
    inputs_train_recon = np.matmul(inputs_train_centered, eigen_matrix)
    inputs_valid_recon = np.matmul(inputs_valid_centered, eigen_matrix)
    knn.fit(inputs_train_recon, inputs_target)
    inputs_valid_target_pred = knn.predict(inputs_valid_recon)

    counter = 0
    for i in range(200):
        if inputs_valid_target_pred[i] != 0 and i <= 99:
            counter += 1
        elif inputs_valid_target_pred[i] != 1 and i >= 100:
            counter += 1
    mismatch_list.append(counter / 200)

    print("WHEN K VALUE IS ", item, " THE CLASSIFICATION ERROR RATE IS ", counter / 200)


plt.plot(k_list, mismatch_list, label="mismatch_line")
plt.legend(loc="upper right")
plt.xlabel(r"$k value$")
plt.ylabel("error rate")
plt.show()
```

```
(600, 2)
WHEN K VALUE IS  2  THE CLASSIFICATION ERROR RATE IS  0.055
(600, 5)
WHEN K VALUE IS  5  THE CLASSIFICATION ERROR RATE IS  0.035
(600, 10)
WHEN K VALUE IS  10  THE CLASSIFICATION ERROR RATE IS  0.01
(600, 20)
WHEN K VALUE IS  20  THE CLASSIFICATION ERROR RATE IS  0.005
(600, 30)
WHEN K VALUE IS  30  THE CLASSIFICATION ERROR RATE IS  0.01
```



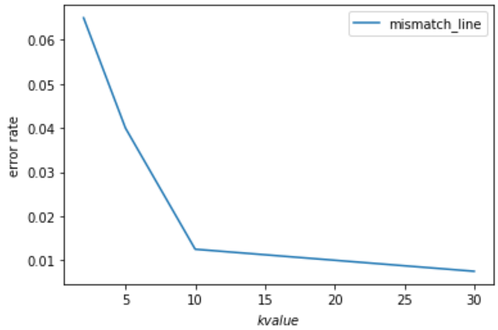
(b)

I would choose K value at 20. Observe from the data above we notice that, when $K = 20$, the classification error reaches the lowerst point, which is 0.005. When K exceeds 20 the model would overfit, and it would learn some error into the system causing the classification rate rising. Therefore, I would choose model with K value at 20.

(c)

```
test_target = np.hstack((np.zeros((200)), np.ones((200))))


inputs_train_mean = np.mean(inputs_train, axis = 0)
inputs_train_centered = inputs_train - np.tile(inputs_train_mean, (inputs_train.shape[0], 1))
inputs_test_centered = inputs_test - np.tile(inputs_train_mean, (inputs_test.shape[0], 1))
cov_matrix = np.cov(inputs_train_centered.T)
eigen_value, eigen_vector = np.linalg.eig(cov_matrix)

mismatch_list = []
for item in k_list:
    eigen_matrix = eigen_vector[:, 0: item]
    inputs_train_recon = np.matmul(inputs_train_centered, eigen_matrix)
    inputs_test_recon = np.matmul(inputs_test_centered, eigen_matrix)

    knn.fit(inputs_train_recon, inputs_target)
    inputs_test_target_pred = knn.predict(inputs_test_recon)

    counter = 0
    for i in range(400):
        if inputs_test_target_pred[i] != test_target[i]:
            counter += 1
    mismatch_list.append(counter / 400)
    print("WHEN K VALUE IS ", item, " THE CLASSIFICATION ERROR RATE IS ", counter / 400)


plt.plot(k_list, mismatch_list, label="mismatch_line")
plt.legend(loc="upper right")
plt.xlabel(r"$k value$")
plt.ylabel("error rate")
plt.show()
```

```
WHEN K VALUE IS  2  THE CLASSIFICATION ERROR RATE IS  0.065
WHEN K VALUE IS  5  THE CLASSIFICATION ERROR RATE IS  0.04
WHEN K VALUE IS  10  THE CLASSIFICATION ERROR RATE IS  0.0125
WHEN K VALUE IS  20  THE CLASSIFICATION ERROR RATE IS  0.01
WHEN K VALUE IS  30  THE CLASSIFICATION ERROR RATE IS  0.0075
```



In this way, we choose $K = 30$, where the classification error rate is the lowest, at 0.0075