1. a) θ: $L(\theta) = P(X_i, c \mid \theta, \pi) = p(c \mid \pi) \prod_{j=1}^{784} p(x_j^{(i)} \mid c, \theta_{jc})$

$$= \pi_c \prod_{j=1}^{784} \theta_{jc}^{x_j^{(i)}} (1-\theta_{jc})^{(1-x_j^{(i)})}$$

Log-Likelihood:

$$l(\theta) = \sum_{i=1}^{N} \log P(c^{(i)}) = \sum_{i=1}^{N} \left[\log(P(c^{(i)}) + \sum_{j=1}^{D} \log P(x_j^{(i)} \mid c^{(i)})\right]$$

$$= \sum_{i=1}^{N} \log P(c^{(i)}) + \sum_{j=1}^{D} \sum_{i=1}^{N} \log P(x_j^{(i)} \mid c^{(i)})$$

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

Set derivative to 0:

$$\frac{\partial l(\theta)}{\partial \theta_{jc}} = 0$$

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

$$\hat{\theta}_{jc} = \frac{\sum_{i=1}^{N} \mathbb{I}(c^{(i)} = c \ \& \ x_j^{(i)} = 1)}{\sum_{i=1}^{N} \mathbb{I}(c^{(i)} = c)}$$

π: Let $\pi_9 = 1 - \sum_{j=0}^{8} \pi_j$

$$l(\pi) = \sum_{i=0}^{N} \sum_{j=0}^{9} \log(\pi_j^{t_j^{(i)}}) = \sum_{i=1}^{N} \sum_{j=0}^{8} \log(\pi_j^{t_j^{(i)}}) + \log(\pi_9^{t_9^{(i)}})$$

Set derivative to 0:

$$\frac{\partial l_\pi}{\partial \pi_j} = 0$$

$$\sum_{i=1}^{N} \frac{t_j^{(i)}}{\pi_j} - \frac{\sum_{i=1}^{N} t_9^{(i)}}{1 - \sum_{j=0}^{8} \pi_j} = 0, \text{ Since } \sum_{j=0}^{9} \pi_j = 1$$

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

b) $p(t \mid x, \theta, \pi) p(x, \theta, \pi) = p(x, \theta, \pi, t)$

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

$$= \frac{\pi_t \prod_{d=1}^{784} \theta_{td}^{x_d} (1-\theta_{td})^{1-x_d}}{\sum_{t=0}^{9} \pi_t \prod_{d=1}^{784} \theta_{td}^{x_d} (1-\theta_{td})^{1-x_d}}$$

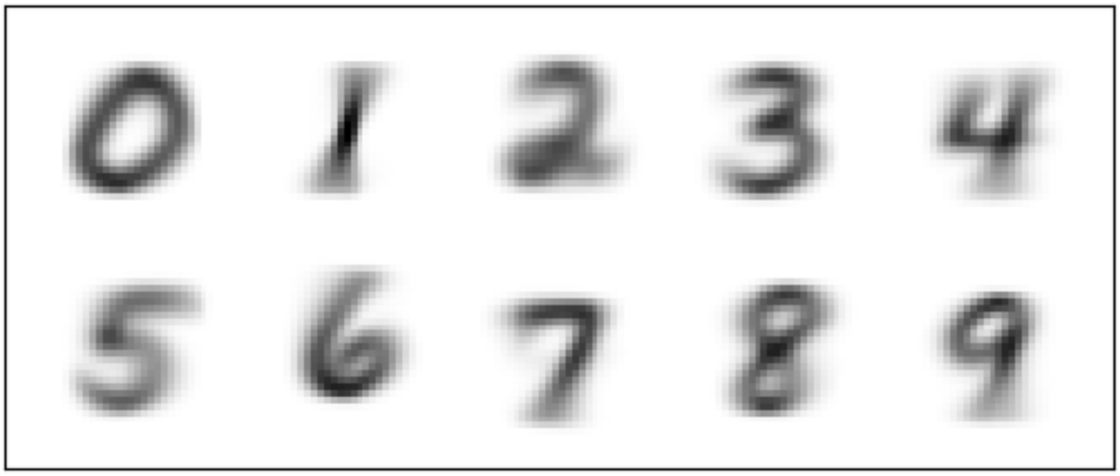
$$\log(p(t \mid x, \theta, \pi)) = \log(\pi_t) + \sum_{d=1}^{784} x_d \log(\theta_{td}) + (1-x_d) \log(1-\theta_{td})$$

$$- \sum_{t=0}^{9} (\log(\pi_t) + \sum_{d=1}^{784} x_d \log(\theta_{td}) + (1-x_d) \log(1-\theta_{td}))$$

$$= - \sum_{i=0, i \neq t}^{9} (\log(\pi_i) + \sum_{d=1}^{784} x_d \log \theta_{id} + \log(1-\theta_{id})(1-x_d))$$

c) The average log likelihood now is nan, since the θ we computed previously is all 0, we can not get a normal log-likelihood value.

d)



e) $p(\theta|x,t,\pi) \propto P(\theta) P(x,t|\theta,\pi)$, $\theta_{cd} \sim Beta(3,3)$

$$l(\theta) = \log P(\theta) + \log P(x,t|\theta,\pi) + C$$

$$= \log(\theta_{td}(1-\theta_{td})) + \log \pi_t^i + \sum_{d=1}^{784} (x_d^i \log \theta_{td} + (1-x_d^i)\log(1-\theta_{td})) + C$$

Set derivative to 0:

$$\frac{\partial l}{\partial \theta_{cd}} = \left(\frac{1}{\theta_{td}} - \frac{1}{1-\theta_{td}}\right) + \sum_{i=1}^{N} \mathbb{1}(t^i = t)\left(\frac{x_d^i}{\theta_{td}} - \frac{1-x_d^i}{1-\theta_{td}}\right) = 0$$

$$= (1-\theta_{td}) - \theta_{td} + \sum_{i=1}^{N} \mathbb{1}(t^i = t)(x_d^i(1-\theta_{td}) - (1-x_d^i)\theta_{cd}) = 0$$

$$\hat{\theta}_{td}^{MAP} = \frac{\sum_{i=1}^{N} \mathbb{1}(t^i = t) x_d^i + 2}{\sum_{i=1}^{N} \mathbb{1}(t^i = t) + 4}$$

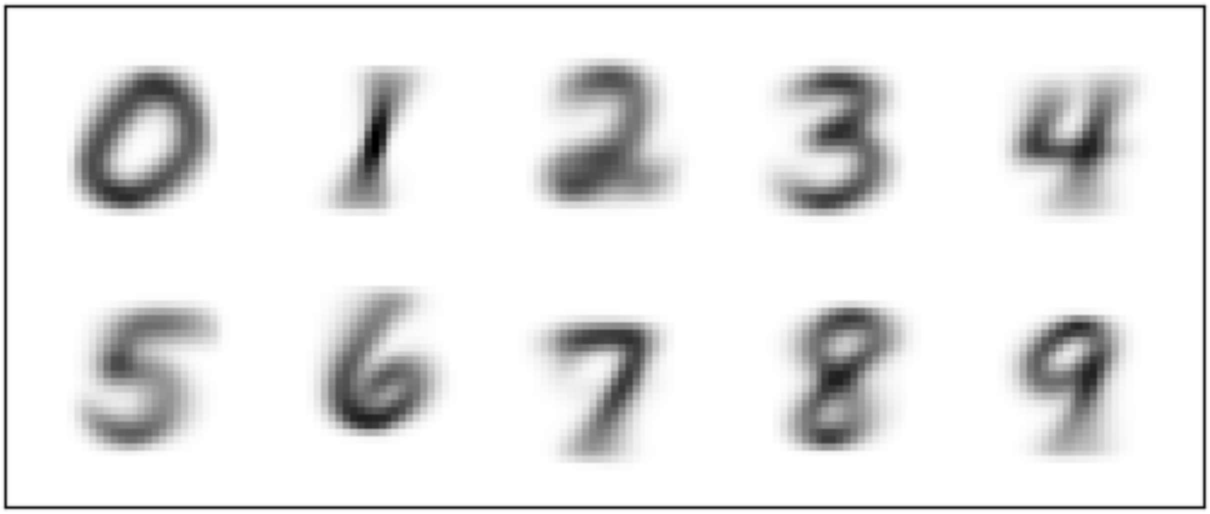
f) Average log-likelihood for MAP is -3.3570631378602855

and

accuracy on training set : 0.8352166666666667

accuracy on test set: 0.816

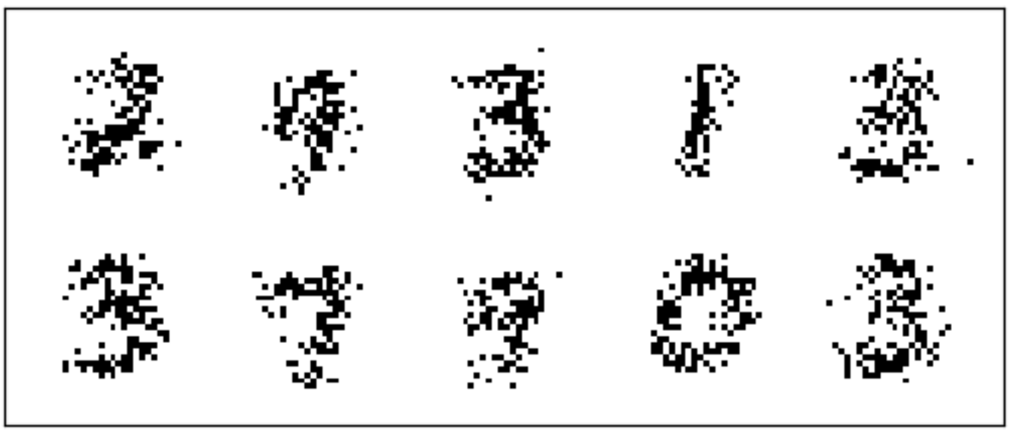
g)



2. a) True — by the nature of Naïve Bayes.

b) False — X_i, X_j is dependent, $p(X_i, X_j) \neq p(X_i)\,p(X_j)$

c)



Q1,2 code:

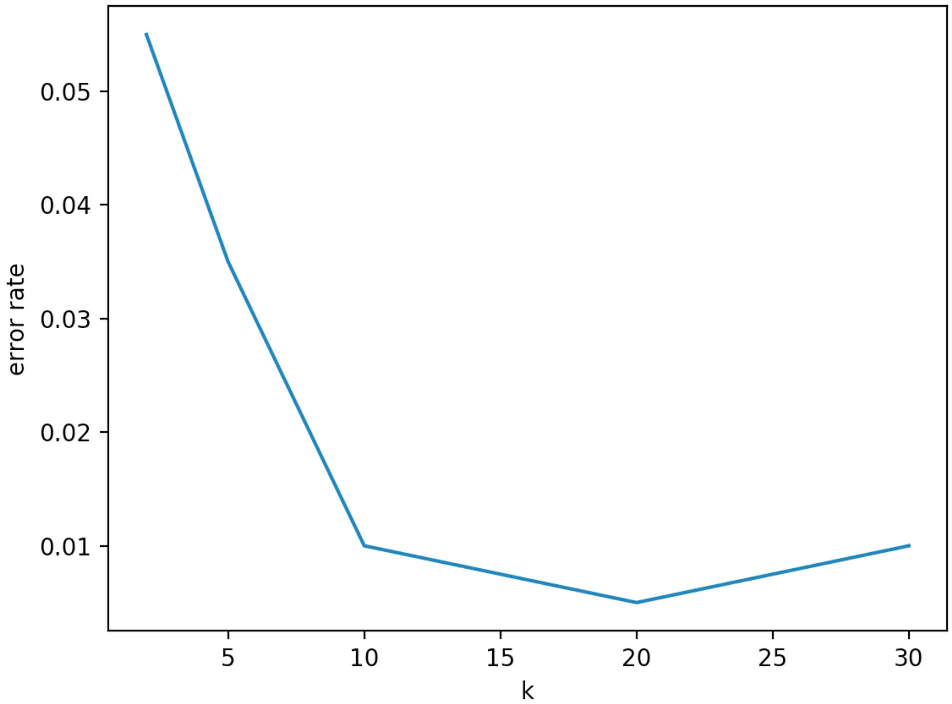
```
from __future__ import absolute_import
from __future__ import print_function
from future.standard_library import install_aliases
install_aliases()
import numpy as np
import os
import gzip
import struct
import array
import matplotlib.pyplot as plt
import matplotlib.image
from urllib.request import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


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)


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
    theta_mle = np.divide(np.matmul(train_images.T, train_labels), np.sum(train_labels, axis=0))

    pi_mle = 1 / train_images.shape[0] * np.sum(train_labels, axis=0)

    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
    theta_map = np.divide(np.matmul(train_images.T, train_labels) + 2, np.sum(train_labels,
axis=0) + 4)

    pi_map = 1 / train_images.shape[0] * np.sum(train_labels, axis=0)
    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
    a = -np.log(np.matmul(np.tile(pi.T, (10, 1)),
                       np.exp(np.matmul(images, np.log(theta)) + np.matmul(1 - images, np.log(1-
theta))).T))
    b = np.matmul(images, np.log(theta)) + np.matmul(1 - images, np.log(1-theta)) + np.log(pi)
    log_like = b+a.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

    p = predict(log_like)
    accuracy = np.mean(p == np.argmax(labels, axis=1))
    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
```

```
    c = np.random.choice(10, 10, p=pi)
    #c = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    sampled_images = []
    for i in range(num_images):
        sampled_images.append([np.random.binomial(1, theta.T[c[i]][j]) for j in range(784)])
    sampled_images = np.asarray(sampled_images)
    return sampled_images


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
    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()
```

3. a)



b) I will choose the model with 20 eigenvectors. Since according to classification error rate for different k values, it is easy to observe that error rate for model with 20 eigenvectors is at least a local minimum, therefore, I suppose this model will have a good, or even best performance among all models that we experiments.

c) The performance of the final classifier over test data:

$k = 20$, error rate: 0.010000000000000009 q3 code: q3.py

```
from q3_materials.utils import *
import numpy as np
import matplotlib.pyplot as plt

def l2_distance(a, b):
    """Computes the Euclidean distance matrix between a and b.
    """

    if a.shape[0] != b.shape[0]:
        raise ValueError("A and B should be of same dimensionality")

    aa = np.sum(a**2, axis=0)
    bb = np.sum(b**2, axis=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.
    """

    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

if __name__ == "__main__":
    inputs_train_T, inputs_valid_T, inputs_test_T, target_train_T, target_valid_T, target_test_T = \
        load_data("/Users/wuqiang/Desktop/311/a3/q3_materials/digits.npz")

    k_lst = [2, 5, 10, 20, 30]
    input_train_recon = []
    error = []

    m = np.mean(inputs_train_T, axis=0)  # compute mean
    input_train_centered = inputs_train_T - np.tile(m, (inputs_train_T.shape[0], 1))  # subtract
mean
    input_train_cov = np.cov(input_train_centered.T)  # compute covariance matrix
    w, v = np.linalg.eig(input_train_cov)

    error_lst = []
    for k in k_lst:
        vm = np.matmul(inputs_valid_T, v[:, :k])
        tm = np.matmul(inputs_train_T, v[:, :k])
        labels = run_knn(1, tm, target_train_T, vm)
        correct = sum([1 for i in range(target_valid_T .shape[0])if labels[i] == target_valid_T[i]])
        error_lst.append(1 - correct / target_valid_T.shape[0])

    plt.plot(k_lst, error_lst)
    plt.xlabel('k')
    plt.ylabel("error rate")
    plt.show()

    f = 20
    test_m = np.matmul(inputs_test_T, v[:, :f])
    tm = np.matmul(inputs_train_T, v[:, :f])
    labels = run_knn(1, tm, target_train_T, test_m)
    correct = sum([1 for i in range(target_test_T.shape[0])if labels[i] == target_test_T[i]])
    print(1 - correct / target_test_T.shape[0])
```

q3 helper: utils.py

```
import numpy as np

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
```