STA414: Statistical Methods for Machine Learning II

Homework 2

Tianyu (Shin) Ren

1002379188

University of Toronto

March 2019

In this assignment, I'll fit both generative and discriminative models to the MNIST dataset of handwritten numbers. Each datapoint in the MNIST [`http://yann.lecun.com/exdb/mnist/`] dataset is a 28x28 black-and-white image of a number in $\{0 \dots 9\}$, and a label indicating which number.

MNIST is the 'fruit fly' of machine learning - a simple standard problem useful for comparing the properties of different algorithms.

For this assignment, I'll binarize the dataset, converting the grey pixel values to either black or white (0 or 1) with > 0.5 being the cutoff. When comparing models, I'll need a training and test set. Use the first 60000 samples for training, and another 10000 for testing.

1. Fitting a Naïve Bayes Model

(a) **Derive MLE for the class-conditional pixel means, $\hat{\theta}_{MLE}$**

$$L(\theta) = P(X_i, c|\theta, \pi) = \pi_c \prod_{d=1}^{784} \theta_{cd}^{x_d^i}(1-\theta_{cd})^{(1-x_d^i)}$$

$$\implies l(\theta) = \log(L(\theta)) = \sum_{i=1}^{N}(\log \pi_c^i + \sum_{d=1}^{784}(x_d^i \log\theta_{cd} + (1-x_d^i)\log(1-\theta_{cd})))$$

$$\operatorname*{argmax}_\theta l(\theta) \quad \text{s.t.} \quad \sum_{c}^{10} \pi_c = 1$$

$$\implies \frac{\partial l}{\partial \theta_{cd}} = \sum_i^N \mathbb{1}(C^i = c)(\frac{x_d^i}{\theta_{cd}} - \frac{1-x_d^i}{1-\theta_{cd}}) = 0$$

$$\implies \sum_i^N \mathbb{1}(C^i = c)\theta_{cd} = \sum_i^N \mathbb{1}(C^i = c)x_d^i$$

$$\implies \hat{\theta}_{cd}^{MLE} = \frac{\sum_i^N \mathbb{1}(C^i = c)x_d^i}{\sum_i^N \mathbb{1}(C^i = c)} \qquad \text{where c} \in \{0, ..., 9\}, d \in \{1, ..., 784\}$$

(b) **Derive MAP for the class-conditional pixel means, $\hat{\theta}_{MAP}$**

$$P(\boldsymbol{\theta}|\boldsymbol{x}, c, \pi) \propto P(\boldsymbol{\theta})P(\boldsymbol{x}, c|\boldsymbol{\theta}, \boldsymbol{\pi}) \quad \text{where} \quad \theta_{cd} \sim \text{Beta(2,2)}$$

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

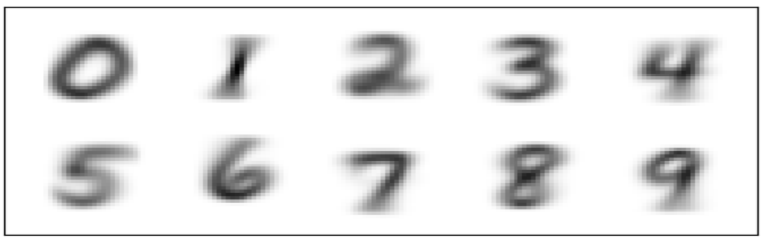
$$= \log(\theta_{cd}(1-\theta_{cd})) + \log \pi_c^i + \sum_{d=1}^{784}(x_d^i \log\theta_{cd} + (1-x_d^i)\log(1-\theta_{cd})) + \text{constant}$$

$$\implies \frac{\partial l}{\partial \theta_{cd}} = (\frac{1}{\theta_{cd}} - \frac{1}{1-\theta_{cd}}) + \sum_{i=1}^{N} \mathbb{1}(C^i = c)(\frac{x_d^i}{\theta_{cd}} - \frac{1-x_d^i}{1-\theta_{cd}}) = 0$$

$$\implies (1-\theta_{cd}) - \theta_{cd} + \sum_{i=1}^{N} \mathbb{1}(C^i = c)(x_d^i(1-\theta_{cd}) - (1-x_d^i)\theta_{cd}) = 0$$

$$\implies \qquad \hat{\theta}_{cd}^{MAP} = \frac{\sum_{i=1}^{N} \mathbb{1}(C^i = c)x_d^i + 1}{\sum_{i=1}^{N} \mathbb{1}(C^i = c) + 2}$$

(c) **Fit $\hat{\theta}_{MAP}$ to the training set. Plot $\hat{\theta}_{MAP}$ as 10 separate greyscale images, one for each class.**



(d) **Derive the log-likelihood $\log p(c|\boldsymbol{x}, \boldsymbol{\theta}, \pi)$ for a single training image**

$$\begin{aligned}
\log P(c|\boldsymbol{x}, \boldsymbol{\theta}, \pi) &= \log P(\boldsymbol{x}|c, \boldsymbol{\theta}) + \log P(c|\pi) - \log P(\boldsymbol{x}) \\
&= \log \prod_{d=1}^{784} P(x_d|c, \theta_{cd}) + \log P(c|\pi) - \log \sum_{c=0}^{9} P(\boldsymbol{x}|C = c)P(C = c) \\
&= \sum_{d=1}^{784} (x_d \log \theta_{cd} + (1-x_d)\log(1-\theta_{cd})) + \log \pi_c - \log \sum_{c=0}^{9} \pi_c \prod_{d=1}^{784} \theta_{cd}^{x_d}(1-\theta_{cd})^{(1-x_d)}
\end{aligned}$$

(e) Given parameters fit to the training set, and $\pi_c = \frac{1}{10}$:

the average log-likelihood per datapoint on the training set: -3.15

the average log-likelihood per datapoint on the test set: -2.98

the accuracy on the training set: 0.834

the accuracy on the test set: 0.845

2. Generating from a Naïve Bayes Model

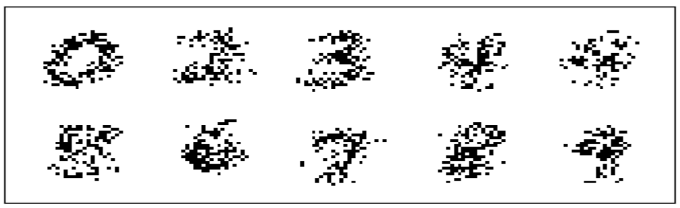
(a) **True**: Given this model's assumptions, any two pixels x_i and x_j where $i \neq j$ are independent given c.

This is the assumption of Naïve Bayes model.

(b) **False**: Given this model's assumptions, any two pixels x_i and x_j where $i \neq j$ are independent when marginalizing over c.

x_i, x_j are not independent since $p(x_i, x_j) = \sum_c p(x_i, x_j|c) = \sum_c p(x_i|c)p(x_j|c)$ and $p(x_i)p(x_j) = \sum_c p(x_i|c) \sum_c p(x_j|c)$. So, $p(x_i, x_j) \neq p(x_i)p(x_j$

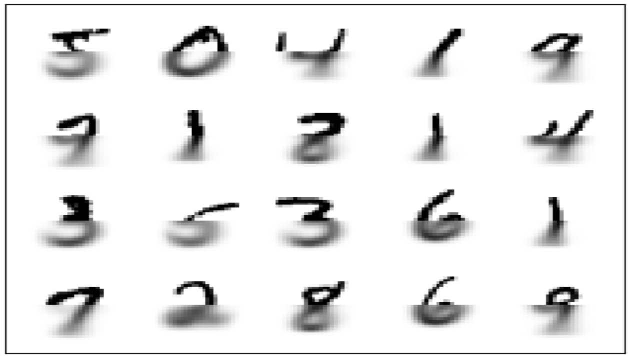
(c) **Using the parameters fit in question 1, randomly sample and plot 10 binary images from the marginal distribution $p(\boldsymbol{x}|\boldsymbol{\theta}, \boldsymbol{\pi})$.**



(d) **Derive $p(\boldsymbol{x}_{i \in bottom}|\boldsymbol{x}_{top}, \boldsymbol{\theta}, \boldsymbol{\pi})$**

$$
\begin{aligned}
p(\boldsymbol{x}_{i \in bottom}|\boldsymbol{x}_{top}) &= \sum_c p(x_i|x_t, c)p(c|x_t) \\
&= \sum_c p(x_i|c)\frac{p(x_t|c)p(c)}{p(x_t)} \\
&= \sum_c p(x_i|c)\frac{\prod_{d=1}^{392} p(x_t^d|c)p(c)}{\sum_{c'} \prod_{d=1}^{392} p(x_t^d|c')p(c')} \\
&= \frac{\sum_{c=0}^{9} \theta_{ci}^{x_i}(1-\theta_{ci})^{(1-x_i)} \prod_{d=1}^{392} \theta_{cd}^{x_t^d}(1-\theta_{cd})^{(1-x_t^d)}\pi_c}{\sum_{c'=0}^{9} \prod_{d=1}^{392} \theta_{c'd}^{x_t^d}(1-\theta_{c'd})^{(1-x_t^d)}\pi_{c'}}
\end{aligned}
$$

(e) **For 20 images from the training set, plot the top half the image concatenated with the marginal distribution over each pixel in the bottom half.**



3. Logistic Regression

Our model will be multiclass logistic regression:

$$p(c|\boldsymbol{x},\boldsymbol{w}) = \frac{\exp(\boldsymbol{w_c^T x})}{\sum_{c'=0}^{9}\exp(\boldsymbol{w_{c'}^T x})}$$

(a) This model has **7840** parameters. Since each data point $\boldsymbol{x_i}$ is a vector with dimension 781 * 1, and there are 10 classes in total. So, it's $784 \times 10 = 7840$

(b) **Derive the gradient of the predictive log-likelihood w.r.t. $\boldsymbol{w}$:** $\nabla_{\boldsymbol{w}} \log p(c|\boldsymbol{x},\boldsymbol{w})$

$$\log P(C|\boldsymbol{x},\boldsymbol{w}) = \boldsymbol{w_c^T x} - \log\sum_{c'=0}^{9}\exp(\boldsymbol{w_{c'}^T x})$$

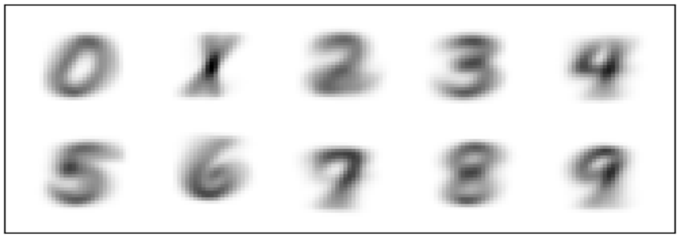
To compute $\nabla_{\boldsymbol{w_j}} \log P(C|\boldsymbol{x},\boldsymbol{w})$,

$$\text{if } j = c \implies \nabla_{\boldsymbol{w_c}} \log P(C|\boldsymbol{x},\boldsymbol{w}) = \boldsymbol{x} - \frac{\exp(\boldsymbol{w_c^T x})}{\sum_{c'=0}^{9}\exp(\boldsymbol{w_{c'}^T x})}\boldsymbol{x}$$

$$\text{if } j \neq c \implies \nabla_{\boldsymbol{w_j}} \log P(C|\boldsymbol{x},\boldsymbol{w}) = -\frac{\exp(\boldsymbol{w_j^T x})}{\sum_{c'=0}^{9}\exp(\boldsymbol{w_{c'}^T x})}\boldsymbol{x}$$

$$\begin{aligned}\text{Also, } \nabla_{\boldsymbol{w_c}} \log\prod_{i=1}^{N} P(C^i|\boldsymbol{x_i},\boldsymbol{w}) &= \sum_{i=1}^{N}\nabla_{\boldsymbol{w_c}} \log P(C^i|\boldsymbol{x_i},\boldsymbol{w}) \\ &= \sum_{i=1}^{N}\mathbb{1}(C^i = c)(\boldsymbol{x_i} - \frac{\exp(\boldsymbol{w_c^T x_i})}{\sum_{c'=0}^{9}\exp(\boldsymbol{w_{c'}^T x_i})}\boldsymbol{x_i})\end{aligned}$$

(c) **The plot of weights, W, one image per class:**



(d) Given parameters fit to the training set:

the average log-likelihood per datapoint on the training set: -0.61

the average log-likelihood per datapoint on the test set: -0.58

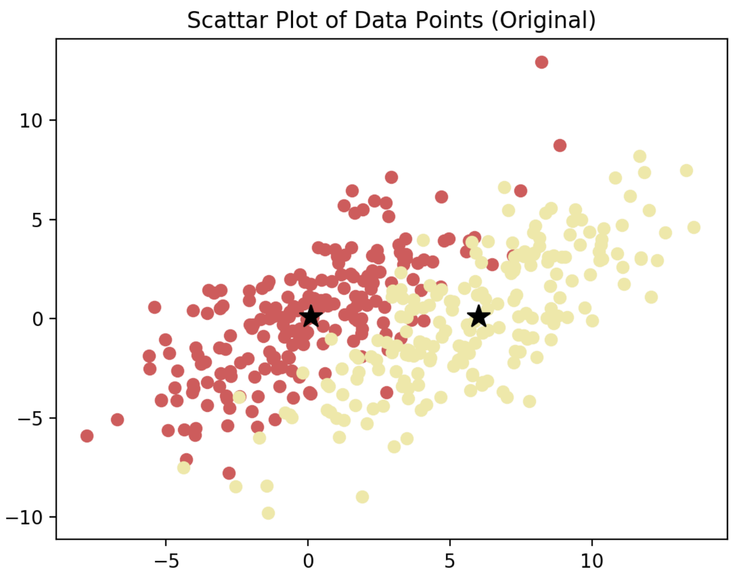
the accuracy on the training set: 0.85

the accuracy on the test set: 0.86

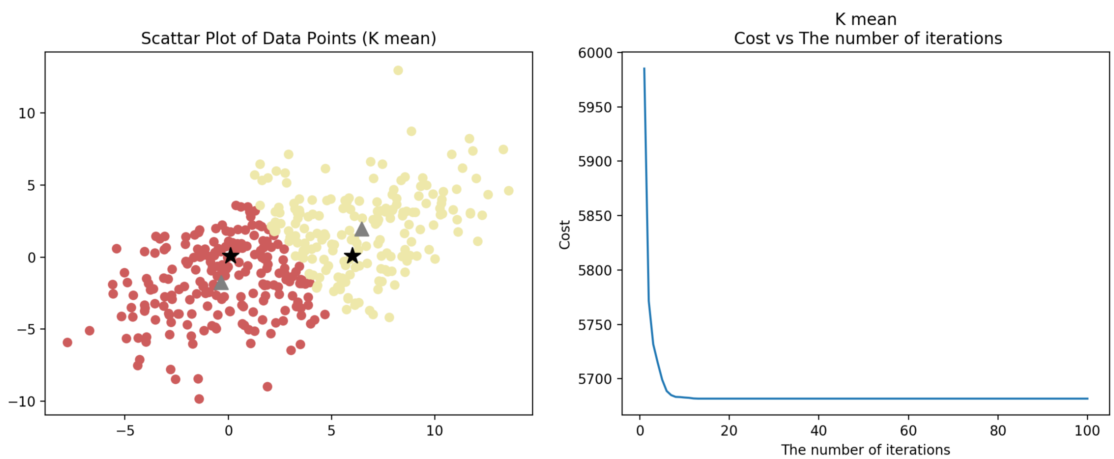
The logistic regression has a better performance than Naïve Bayes both in training and test sets.

4. Unsupervised Learning

(a) First, we will generate some data for this problem. Set the number of points $N = 400$, their dimension $D = 2$, and the number of clusters $K = 2$, and generate data from the distribution $p(x|\mathcal{C}_k) = \mathcal{N}(\mu_k, \Sigma_k)$. Sample 200 data points for $\mathcal{C}_1$ and 200 for $\mathcal{C}_2$

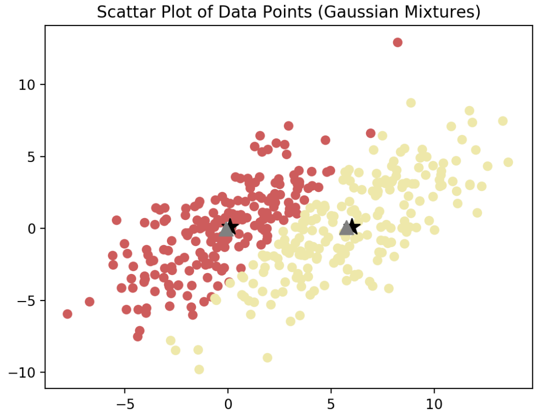


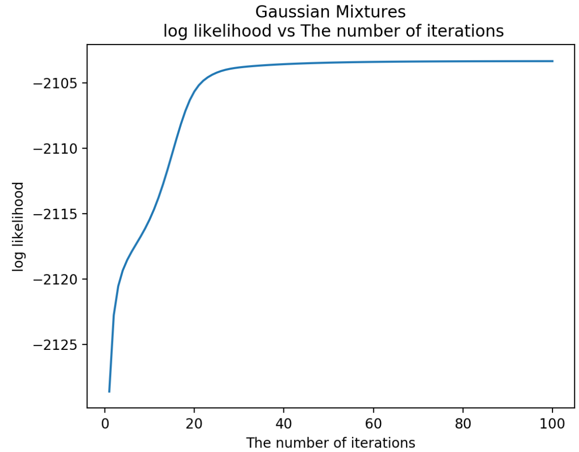
(b) **Now, we assume that the true class labels are not known and implement the k-means algorithm.**



The misclassification error for k mean: 0.255

(c) **Next, implement the EM algorithm for Gaussian mixtures.**





The misclassification error for Gaussian mixtures: 0.08

(d) **Comments:**
As we can see the plots from (b) and (c), resulting centers are very different from true means in the K mean algorithm while the centers are almost the same in the EM algorithm. Thus, we had a poor clustering performance in K mean and a good performance in EM. Regarding to the number of iterations, we can see that K mean converges in only a few steps, within 5 iterations while EM takes more than 30 iterations. If I change the covariace matrix to:

$$\Sigma_1 = \Sigma_2 = \begin{bmatrix} 1 & 7 \\ 7 & 1 \end{bmatrix}$$

then both K mean and EM have good performances and K mean converges faster than EM. So by hint, in most cases, k-means should fail to identify the correct cluster labels due to the covariance structure. There may be realizations that EM also fails to find the clusters correctly but in general it should work better than k-means.

Appendix

Python Code

loadMNIST.py

```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

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, test_labels


```

```
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
    test_images  = partial_flatten(test_images)  / 255.0
    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)
```

model.py

```
from loadMNIST import *
from scipy.special import logsumexp
import time
np.random.seed(1)


COLORS = ["indianred", "palegoldenrod", "black", "gray"]


def get_images_by_label(images, labels, query_label):
    """
    Helper function to return all images in the provided array which match the query
    label.
```

```python
    """
    assert images.shape[0] == labels.shape[0]
    matching_indices = labels == query_label
    return images[matching_indices]


class NaiveBayes:
  """
  Q1, Naive Bayes model.
  """
  def __init__(self, train_images, train_labels):
    self.train_images = train_images
    self.train_labels = train_labels

  def map_naive_bayes(self, plot=False):
    """
    return a matrix 10 * 784 where each row represents a class.
    """
    theta = np.zeros((10, 784))
    for c in range(10):
        images = get_images_by_label(self.train_images, self.train_labels, c)
        theta[c] = np.divide(np.sum(images, axis=0) + 1., images.shape[0] + 2.)
    if plot:
      save_images(theta, "theta_map.png")
    return theta

  def log_likelihood(self, X, y, theta):
    """
    return a matrix N * 10 where each row represents log likelihood of a data point,
    and each column represents log lilihood of a class.
    """
    ll = np.zeros((X.shape[0], 10))
    log_p_x = logsumexp(np.log(0.1) + np.dot(X, np.log(theta.T)) + np.dot((1. - X), np.log(1. - theta.T)), axis=1)
    for c in range(10):
      ll[:, c] = np.dot(X, np.log(theta[c])) + np.dot((1. - X), np.log(1. - theta[c])) + np.log(0.1) - log_p_x
    return ll

  def avg_log_likelihood(self, X, y, theta):
    ll = 0
    for c in range(10):
      X_c = get_images_by_label(X, y, c)
      log_p_x = logsumexp(np.log(0.1) + np.dot(X_c, np.log(theta.T)) + np.dot((1. - X_c), np.log(1. - theta.T)), axis=1)
      ll += np.sum(np.dot(X_c, np.log(theta[c])) + np.dot((1. - X_c), np.log(1. - theta[c])) + np.log(0.1) - log_p_x)
    return ll / X.shape[0]

  def predict(self, X, y, theta, train=False, test=False):
    ll = self.log_likelihood(X, y, theta)
    pred = np.argmax(ll, axis=1)
    avg_ll = self.avg_log_likelihood(X, y, theta)
    accuracy = np.mean(pred == y)
    name = "test" if test else "train"
    print("average log-likelihood of naive bayes model on the {} set: ".format(name) + str(avg_ll))
    print("accuracy of naive bayes model on the {} set: ".format(name) + str(
```

```
    accuracy))


class GenerativeNaiveBayes:
  """
  Q2, Generating from a Naive Bayes Model
  """
  def __init__(self, theta):
    self.theta = theta

  def sample_plot(self):
    """
    randomly sample and plot 10 binary images from the marginal distribution, p(x|
    theta, pi)
    """
    c = np.random.multinomial(10, [0.1]*10)
    images = np.zeros((10, 784))
    count = 0
    for i in range(10):
      for j in range((c[i])):
        images[count] = np.random.binomial(1, self.theta[i]).reshape((1, 784))
        count += 1
    save_images(images, "samples.png")

  def predict_half(self, X_top):
    """
    plot the top half the image concatenated with the marginal distribution over
    each pixel in the bottom half.
    """
    X_bot = np.zeros((X_top.shape[0], X_top.shape[1]))
    theta_top, theta_bot = self.theta[:, :392].T, self.theta[:, 392:].T
    for i in range(392):
      constant = np.dot(X_top, np.log(theta_top)) + np.dot(1 - X_top, np.log(1 -
    theta_top))
      X_bot[:, i] = logsumexp(np.add(constant, np.log(theta_bot[i])), axis=1) -
    logsumexp(constant, axis=1)
    save_images(np.concatenate((X_top, np.exp(X_bot)), axis=1), "predict_half.png")


class LogisticRegression:
  """
  Q3, Fitting a simple predictive model using gradient descent.
  Our model will be multiclass logistic regression.
  """
  def __init__(self, train_images, train_labels):
    self.train_images = train_images
    self.train_labels = train_labels
    self.W = np.zeros((10, 784))

  def softmax(self, X, W):
    """
    return a N * 10 vector where each row is a data point
    and each column is the probability of that class.
    """
    return (np.exp(np.dot(X, W.T)).T / np.exp(logsumexp(np.dot(X, W.T), axis=1))).T

  def grad_pred_ll(self, X, W, c):
    """
```

```
    This function calculate the gradient of the predictive log-likelihood.
    return a 1 * 784 vector
    """
    constant = np.exp(logsumexp(np.dot(X, W.T), axis=1))
    return np.sum(X - (X.T * np.divide(np.exp(np.dot(X, W[c])), constant)).T, axis=0)

  def gradient_ascent(self, lr=0.00001, iters=100):
    for _ in range(iters):
      prob = self.softmax(self.train_images, self.W)
      pred = np.argmax(prob, axis=1)
      accuracy = np.mean(pred == self.train_labels)
      print("training accuracy: {}, iterations: {}/{}".format(round(accuracy, 2), _, iters))
      for c in range(10):
        X_c = get_images_by_label(self.train_images, self.train_labels, c)
        self.W[c] = self.W[c] + lr * self.grad_pred_ll(X_c, self.W, c)

  def log_likelihood(self, X, y, W):
    ll = 0
    for c in range(10):
      X_c = get_images_by_label(X, y, c)
      ll += np.sum(np.dot(X_c, W[c]) - logsumexp(np.dot(X_c, W.T), axis=1))
    return ll / X.shape[0]

  def predict(self, X, y, train=False, test=False):
    if train:
      self.gradient_ascent()
      save_images(self.W, "weights.png")
    avg_ll = self.log_likelihood(X, y, self.W)
    pred = np.argmax(self.softmax(X, self.W), axis=1)
    accuracy = np.mean(pred == y)
    name = "test" if test else "train"
    print("average log-likelihood of softmax model on the {} set: ".format(name) + str(avg_ll))
    print("accuracy of softmax model on the {} set: ".format(name) + str(accuracy))


class EM:
  """
  Q4, EM algorithm for K means and Gaussian mixtures.
  """
  def __init__(self, initials, c1, c2):
    self.initials = initials
    self.data = np.concatenate((c1, c2), axis=0)
    self.N, self.D = self.data.shape      # Data is a N * D matrix, and here N=400, D=2

    # Initial values for K mean and GMM
    self.miu_hat = np.concatenate((self.initials['MIU1_HAT'], self.initials['MIU2_HAT']), axis=0).reshape((2,2))
    self.clusters = np.concatenate((np.zeros(int(self.N/2)), np.ones(int(self.N/2))), axis=0)
    self.costs_iter = [[], []]

  def plot_clusters(self, km=False, gmm=False):
    """
    a scatter plot of the data points showing the true cluster assignment of each
```

```
    point.
    Also plot a scatter plot of K mean or gaussian mixtures.
    """
    f2 = plt.figure()
    ax2 = f2.add_subplot(111)
    for i in range(self.D):
      plt.scatter(self.data[self.clusters == i][:, 0], self.data[self.clusters == i
    ][:, 1], c=COLORS[i])
      plt.scatter(self.initials['MIU'+str(i+1)][0], self.initials['MIU'+str(i+1)
    ][1], marker='*', c=COLORS[2], s=150)
    plt.title("Scattar Plot of Data Points (Original)")
    if km or gmm:
      name = "K mean" if km else "Gaussian Mixtures"
      plt.scatter(self.miu_hat[:,0], self.miu_hat[:,1], marker='^', c=COLORS[3], s
    =100)
      plt.title("Scattar Plot of Data Points ({})".format(name))

  def misclassification_error(self):
    return (np.sum(self.clusters[:int(self.N/2)] == 1) + np.sum(self.clusters[int(
    self.N/2):] == 0)) / self.N


class KMean(EM):
  def __init__(self, initials, c1, c2):
    super().__init__(initials, c1, c2)

  def cost(self):
    cost = 0
    for i in range(self.D):
      cost += np.sum(np.linalg.norm(self.data[self.clusters == i] - self.miu_hat[i],
     axis=1) ** 2)
    return cost

  def km_e_step(self):
    distances = np.zeros((self.N, 2))
    for i in range(self.D):
      distances[:,i] = np.linalg.norm(self.data - self.miu_hat[i], axis=1)
    self.clusters = np.argmin(distances, axis=1)

  def km_m_step(self):
    for i in range(self.D):
      self.miu_hat[i] = np.mean(self.data[self.clusters == i], axis=0)

  def train(self, max_iter=100):
    i = 1
    while i <= max_iter:
      self.km_e_step()
      self.km_m_step()
      self.costs_iter[0].append(self.cost())
      self.costs_iter[1].append(i)
      i += 1
    f3 = plt.figure()
    ax3 = f3.add_subplot(111)
    plt.plot(self.costs_iter[1], self.costs_iter[0])
    plt.title("K mean\n Cost vs The number of iterations")
    plt.xlabel("The number of iterations")
    plt.ylabel("Cost")
    self.plot_clusters(km=True)
```

```
    print("misclassification error for k mean: " + str(self.misclassification_error()))


class GaussianMixtures(EM):
  def __init__(self, initials, c1, c2):
    super().__init__(initials, c1, c2)
    # Initial values for Gaussian mixtures
    self.simga_hat = [np.eye(self.D)] * 2
    self.pi_hat = [0.5, 0.5]          # Mixing proportions
    self.R = np.zeros((self.N, 2))        # This is the posterior/responsibilities
    self.N_k = []               # The number of data in class K

  def normal_density(self, X, miu, sigma):
    """
    This is a vectorized normal_densitym, where X is N*D, miu is 1*D, sigma is D*D
    Output is N*1, where each element is a pdf value.
    """
    constant = 1 / np.sqrt((2 * np.pi) ** self.D * np.linalg.det(sigma))
    return constant * np.diag(np.exp(-0.5 * np.dot(np.dot((X - miu), np.linalg.inv(sigma)), (X - miu).T)))

  def log_likelihood(self):
    normal_sum = np.zeros(self.N)
    for i in range(self.D):
      normal_sum += self.pi_hat[i] * self.normal_density(self.data, self.miu_hat[i], self.simga_hat[i])
    return np.sum(np.log(normal_sum))

  def em_e_step(self):
    for i in range(self.D):
      self.R[:, i] = self.pi_hat[i] * self.normal_density(self.data, self.miu_hat[i].reshape((1,2)), self.simga_hat[i])
    # Normalize R
    self.R = (self.R.T / np.sum(self.R, axis=1)).T
    # assign datapoints to each gaussian
    self.N_k = np.sum(self.R, axis = 0)

  def em_m_step(self):
    for i in range(self.D):
      self.miu_hat[i] = 1. / self.N_k[i] * np.sum(self.R[:, i] * self.data.T, axis=1).T
      diff = self.data - self.miu_hat[i]
      self.simga_hat[i] = 1. / self.N_k[i] * np.dot(np.multiply(diff.T,  self.R[:, i]), diff)
      self.pi_hat[i] = self.N_k[i] / self.N

  def train(self, max_iter=100):
    i = 1
    while i <= max_iter:
      self.em_e_step()
      self.em_m_step()
      self.costs_iter[0].append(self.log_likelihood())
      self.costs_iter[1].append(i)
      i += 1
    f4 = plt.figure()
    ax4 = f4.add_subplot(111)
    plt.plot(self.costs_iter[1], self.costs_iter[0])
```

```
    plt.title("Gaussian Mixtures\n log likelihood vs The number of iterations")
    plt.xlabel("The number of iterations")
    plt.ylabel("log likelihood")
    self.clusters = np.argmax(self.R, axis=1)
    self.plot_clusters(gmm=True)
    print("misclassification error for gmm: " + str(self.misclassification_error()))


if __name__ == '__main__':
  start = time.time()
  print("loading data...")
  N_data, train_images, train_labels, test_images, test_labels = load_mnist()
  train_labels = np.argmax(train_labels, axis=1)
  test_labels = np.argmax(test_labels, axis=1)

  print("trainning a Naive Bayes model...")
  nb_model = NaiveBayes(train_images, train_labels)
  theta_map = nb_model.map_naive_bayes(plot=True)
  nb_model.predict(train_images, train_labels, theta_map, train=True)
  nb_model.predict(test_images, test_labels, theta_map, test=True)

  print("training a generative Naive Bayes model...")
  gnb = GenerativeNaiveBayes(theta_map)
  gnb.sample_plot()
  gnb.predict_half(train_images[:20,:392])

  print("training a softmax model...")
  lr_model = LogisticRegression(train_images, train_labels)
  lr_model.predict(train_images, train_labels, train=True)
  lr_model.predict(test_images, test_labels, test=True)

  print("training K mean and GMM algorithms...")
  initials = {'Nk': 200,
        'MIU1': np.array([0.1, 0.1]),
        'MIU2': np.array([6., 0.1]),
        'COV': np.array([[10., 7.], [7., 10.]]),
        'MIU1_HAT': np.array([0., 0.]),
        'MIU2_HAT': np.array([1., 1.])
        }
  # Sampling data from a multivariate guassian distribution
  c1 = np.random.multivariate_normal(initials['MIU1'], initials['COV'], initials['Nk'])
  c2 = np.random.multivariate_normal(initials['MIU2'], initials['COV'], initials['Nk'])
  kmean = KMean(initials, c1, c2)
  kmean.plot_clusters()
  kmean.train()
  gmm = GaussianMixtures(initials, c1, c2)
  gmm.train()
  end = time.time()
  print("running time: {}s".format(round(end - start, 2)))
  plt.show()
```