STA414 - Modeling MNIST dataset

Zikun Chen

March 2019

1. Fitting a Naïve Bayes Model

a) MLE Estimates for θ

If we have N images in the dataset, to derive the MLE estimator for the pixel means $\boldsymbol{\theta} = [\theta_{cd}]_{N\times 784}$, we are assuming datapoints (MNIST digits) x_n's are sampled independent to each other. Denote the dataset to be $\boldsymbol{X} = [x_1, \cdots, x_N]^T$. Their class label set is denoted as $\boldsymbol{C} = [c_1, \cdots, c_N]^T$.

Therefore, the distribution of images given parameters $\{\boldsymbol{\theta}, \pi\}$ is:

$$p(\boldsymbol{X}|\boldsymbol{\theta}, \pi) = \prod_{n=1}^{N} p(c_n|\pi) \prod_{d=1}^{784} p(x_{nd}|c_n, \theta_c d) = \prod_{n=1}^{N} p(c_n|\pi) \prod_{d=1}^{784} \theta_{cd}^{x_{nd}} (1-\theta_{cd})^{(1-x_{nd})}$$

$$ln[p(\boldsymbol{X}|\boldsymbol{\theta}, \pi)] = \sum_{n=1}^{N} \{ln[p(c_n|\pi)] + \sum_{d=1}^{784} (x_{nd} ln[\theta_{cd}] + (1-x_{nd}) ln[1-\theta_{cd}])\}$$

Let $R_c = \{n : p(c_n|\pi) = \pi_c, n \in \{1, 2, \cdots, N\}\}$
i.e. R_c is the collection of indices of digits from class c.

For a particular class c, and a particular digit d:

$$\begin{aligned} \frac{\partial ln[p(\boldsymbol{X}|\boldsymbol{\theta}, \pi)]}{\partial \theta_{cd}} &= \sum_{n\in R_c} [\frac{x_{nd}}{\theta_{cd}} - \frac{1-x_{nd}}{1-\theta_{cd}}] \\ &= \sum_{n\in R_c} \frac{x_{nd} - \theta_{cd}}{\theta_{cd}(1-\theta_{cd})} \end{aligned}$$

Setting $\frac{\partial ln[p(\boldsymbol{X}|\theta, \pi)]}{\partial \theta_{cd}} = 0$ gives:

$$\sum_{n\in R_c} x_{nd} - |R_c|\theta_{cd} = 0$$

$$\Rightarrow \hat{\theta}_{cd}^{ML} = \frac{\sum_{n\in R_c} x_{nd}}{|R_c|}$$

$\hat{\theta}_{cd}$ is the ratio between the count of the d'th white pixels (where $x_{nd} = 1$) in class c to the number of digit c within the whole image dataset $\boldsymbol{X}$.

b) MAP Estimates for $\boldsymbol{\theta}$

Beta(2,2) prior on θ_{cd}:

$$p(\theta_{cd}) = \frac{\Gamma(4)}{\Gamma(2)\Gamma(2)}\theta_{cd}(1-\theta_{cd}) = 6\theta_{cd}(1-\theta_{cd}) \propto \theta_{cd}(1-\theta_{cd})$$

$$p(\boldsymbol{\theta}) \propto \prod_{n=1}^{N}\prod_{c=0}^{9}\theta_{cd}(1-\theta_{cd})$$

Therefore,

$$p(\boldsymbol{X},\boldsymbol{\theta}|\pi) = p(\boldsymbol{X}|\boldsymbol{\theta},\pi)p(\boldsymbol{\theta}) \propto [\prod_{n=1}^{N}\prod_{c=0}^{9}\theta_{cd}(1-\theta_{cd})][\prod_{n=1}^{N}p(c_n|\pi)\prod_{d=1}^{784}\theta_{cd}^{x_{nd}}(1-\theta_{cd})^{(1-x_{nd})}]$$

$$\begin{aligned} ln[p(\boldsymbol{X}|\boldsymbol{\theta},\pi)p(\boldsymbol{\theta})] = &\sum_{n=1}^{N}\sum_{c=0}^{9}\{ln[\theta_{cd}] + ln[1-\theta_{cd}]\} \\ &+ \sum_{n=1}^{N}\{ln[p(c_n|\pi)] + \sum_{d=1}^{784}(x_{nd}ln[\theta_{cd}] + (1-x_{nd})ln[1-\theta_{cd}])\} \end{aligned}$$

Similar to part a):

$$\begin{aligned} \frac{\partial ln[p(D|\theta,\pi)]}{\partial\theta_{cd}} &= \frac{1}{\theta_{cd}} - \frac{1}{1-\theta_{cd}} + \sum_{n\in R_c}(\frac{1+x_{nd}}{\theta_{cd}} + \frac{2-x_{nd}}{1-\theta_{cd}}) \\ &= \frac{1-2\theta_{cd}}{\theta_{cd}(1-\theta_{cd})} + \sum_{n\in R_c}\frac{1+x_{nd}+\theta_{nd}-2\theta_{cd}x_{nd}}{\theta_{cd}(1-\theta_{cd})} \\ &= 0 \end{aligned}$$

$$\Rightarrow 1 - 2\theta_{cd} + \sum_{n\in R_c}x_{nd} - |R_c|\theta_{cd} = 0$$

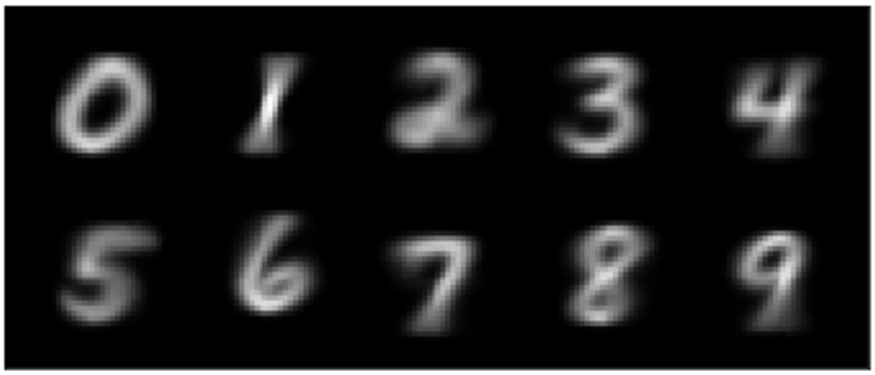
This gives the MAP estimator:

$$\hat{\theta}_{cd}^{MAP} = \frac{1+\sum_{n\in R_c}x_{nd}}{2+|R_c|}$$

Again, $\sum_{n\in R_c}x_{nd}$ is the count of the d'th white pixels (where $x_{nd}=1$) in class c, and $|R_c|$ is the count of digit c within the whole image dataset $\boldsymbol{X}$. In addition, the MAP estimator resulting from the Beta prior adds pseudo-counts 1 and 2 to these two counts respectively.

c) Plot $\boldsymbol{\theta}$

Using the MAP formulation in part b) and our training set $\boldsymbol{X}$ (first 10,000 images), we arrived at the following $\boldsymbol{\theta}$, visualized as 10 seperated images:



d) Posterior for a single image x

$$p(c|\boldsymbol{x},\boldsymbol{\theta},\pi) = \frac{p(\boldsymbol{x},c|\theta,\pi)}{\sum_{c=0}^{9} p(\boldsymbol{x},c|\boldsymbol{\theta},\pi)} = \frac{\pi_c \prod_{d=1}^{784} \theta_{cd}^{x_d}(1-\theta_{cd})^{(1-x_d)}}{\sum_{c=0}^{9} \pi_c \prod_{d=1}^{784} \theta_{cd}^{x_d}(1-\theta_{cd})^{(1-x_d)}}$$

$$ln[p(c|\boldsymbol{x},\boldsymbol{\theta},\pi)] = ln\pi_c + \sum_{d=1}^{784}[x_d ln\theta_{cd} + (1-x_d)ln(1-\theta_{cd})] - ln\sum_{c=0}^{9}\pi_c \prod_{d=1}^{784}\theta_{cd}^{x_d}(1-\theta_{cd})^{(1-x_d)}$$

e) Average Log-Likelihood and Accuracy

```
Naive Bayes Model:
Average Log-Likelihood on Training Set: -3.3371833755635434
Average Log-Likelihood on Test Set: -3.170983371734733
Training Accuracy: 0.8398
Testing Accuracy: 0.8414
```

2. Generating from Naïve Bayes Model

a) Conditional Independence

This is true. Conditional independence assumption of Naïve Bayes Model assumes that for any two pixels i and j, $x_i|c \perp\!\!\!\perp x_j|c$

$$p(x_i, x_j|c,\boldsymbol{\theta},\pi) = p(x_i|c,\boldsymbol{\theta},\pi)p(x_j|c,\boldsymbol{\theta},\pi)$$

b) Marginal Independence

This is false. Conditional independence assumption of does not imply that the marginal distributions of x_i and x_j are independent.
Since:

$$p(x_i|\boldsymbol{\theta}, \pi) = \sum_{c=0}^{9} p(c|\pi)p(x_i|c, \boldsymbol{\theta}, \pi)$$

$$p(x_j|\boldsymbol{\theta}, \pi) = \sum_{c=0}^{9} p(c|\pi)p(x_j|c, \boldsymbol{\theta}, \pi)$$

By conditional independence:

$$p(x_i, x_j|\boldsymbol{\theta}, \pi) = \sum_{c=0}^{9} p(c|\pi)p(x_i, x_j|c, \boldsymbol{\theta}, \pi) = \sum_{c=0}^{9} p(c|\pi)p(x_i|c, \boldsymbol{\theta}, \pi)p(x_j|c, \boldsymbol{\theta}, \pi)$$

Therefore, x_i and x_j are not marginally independent:

$$p(x_i, x_j|\boldsymbol{\theta}, \pi) \neq p(x_i|\boldsymbol{\theta}, \pi)p(x_j|\boldsymbol{\theta}, \pi)$$

c) Sample 10 images from marginal distribution

Marginal distribution:

$$p(\boldsymbol{x}|\boldsymbol{\theta}, \pi) = \sum_{c=0}^{9} p(c|\pi)p(\boldsymbol{x}, c|\boldsymbol{\theta}, \pi)$$

To sample from this, we first sample c (choose a class) using categorical distribution π and then given c, and fitted $\boldsymbol{\theta}$ from part b) as parameters, we sample from binomial distribution for each pixel d. (See function `sample_image` in Appendix 1))

d) Missing Data Distribution

For a single pixel i in the bottom half of the image:

$$p(\boldsymbol{x}_{i\in bottom}|x_{top},\boldsymbol{\theta},\pi) = \frac{p(\boldsymbol{x}_{i\in bottom},\boldsymbol{x}_{top}|\boldsymbol{\theta},\pi)}{p(\boldsymbol{x}_{top}|\boldsymbol{\theta},\pi)}$$

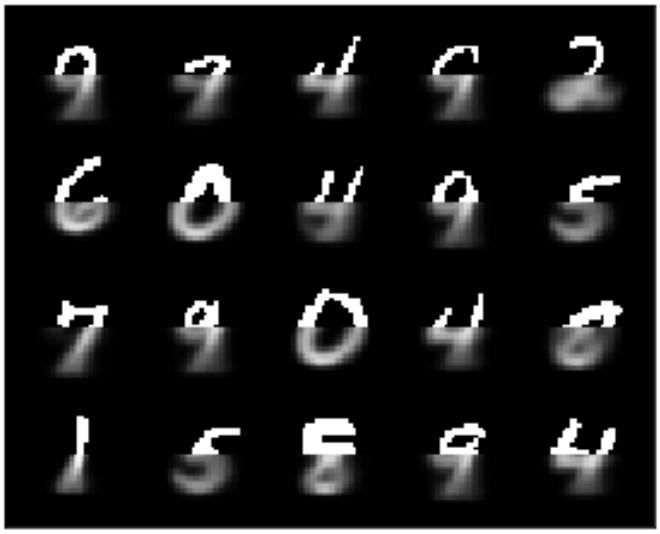
$$= \frac{\sum_{c=0}^{9} p(c|\pi)p(\boldsymbol{x}_{i\in bottom},\boldsymbol{x}_{top}|c,\boldsymbol{\theta},\pi)}{\sum_{c=0}^{9} p(c|\pi)p(\boldsymbol{x}_{top}|c,\boldsymbol{\theta},\pi)}$$

$$= \frac{\sum_{c=0}^{9} p(c|\pi)p(\boldsymbol{x}_{i\in bottom}|c,\boldsymbol{\theta},\pi)p(\boldsymbol{x}_{top}|c,\boldsymbol{\theta},\pi)}{\sum_{c=0}^{9} p(c|\pi)p(\boldsymbol{x}_{top}|c,\boldsymbol{\theta},\pi)}$$

$$= \frac{\sum_{c=0}^{9} \pi_c \theta_{ci}^{x_i}(1-\theta_{ci})^{1-x_i}\prod_{d=1}^{392}\theta_{cd}^{x_d}(1-\theta_{cd})^{1-x_d}}{\sum_{c=0}^{9}\pi_c\prod_{d=1}^{392}\theta_{cd}^{x_d}(1-\theta_{cd})^{1-x_d}}$$

e) Generative 20 Half Images

To generate half images, we first compute each component of the sum in the denominator of the formula in part d) and store these components in an vector `class_sum`. The denominator is the same for any pixel in the bottom half. We than compute the numerator and construct the probability for each pixel in the bottom half. (See function `sample_half` in Appendix 1))



3. Logistic Regression:

a) Number of Parameters

There are $10 \times 784 = 7840$ parameters in the logistic regression model.

b) Gradient of Perdictive Log-Likelihood

For a particular class k, the gradient of the log-likelihood function is:

$$\nabla_{w_k} lnp(c|\mathbf{x}, \mathbf{w}) = \nabla_{w_k}[\mathbf{w}_c^T\mathbf{x} - \log \sum_{c'=0}^{9} exp(\mathbf{w}_{c'}^T\mathbf{x})] = \begin{cases} \mathbf{x} - \frac{\mathbf{x}exp(\mathbf{w}_k^T\mathbf{x})}{\sum_{c'=0}^{9} exp(\mathbf{w}_{c'}^T x)} & \text{if } k = c \\ -\frac{\mathbf{x}exp(\mathbf{w}_k^T\mathbf{x})}{\sum_{c'=0}^{9} exp(\mathbf{w}_{c'}^T\mathbf{x})} & \text{otherwise} \end{cases}$$

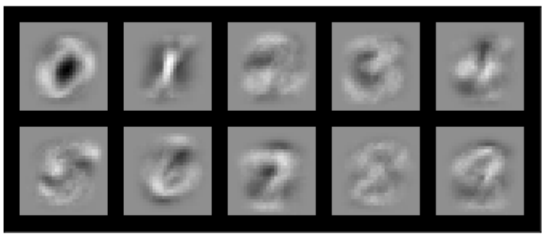
For the whole dataset:

$$\begin{aligned} \nabla_{w_k} \sum_{n=1}^{N} lnp(c|\mathbf{x}_n, \mathbf{w}) &= \nabla_{w_k} \sum_{n=1}^{N} [\mathbf{w}_c^T\mathbf{x}_n - \log \sum_{c'=0}^{9} exp(\mathbf{w}_{c'}^T\mathbf{x}_n)] \\ &= \begin{cases} \sum_{n=1}^{N} \mathbf{x}_n\{1 - \frac{exp(\mathbf{w}_k^T\mathbf{x})}{\sum_{c'=0}^{9} exp(\mathbf{w}_{c'}^T\mathbf{x}_n)}\} & \text{if } k = c \\ -\sum_{n=1}^{N} \frac{\mathbf{x}_n exp(\mathbf{w}_k^T\mathbf{x}_n)}{\sum_{c'=0}^{9} exp(\mathbf{w}_{c'}^T\mathbf{x}_n)} & \text{otherwise} \end{cases} \end{aligned}$$

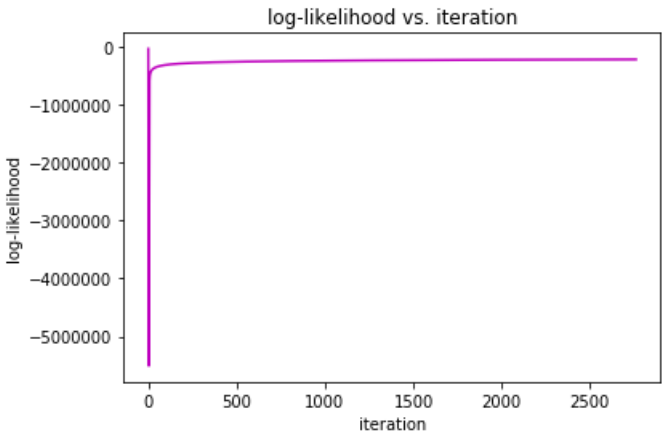
c) Gradient-Based Optimizer

Here, I have implemented an gradient-based vanilla gradient ascent optimizer to maximize the log likelihood until the likelihood converges. For simplicity, The learning rate is initially 0.05 and decreases proportionally to number of iterations. (See Appendix 2), where `grad_LL` computes the gradient over the whole dataset, `log_likelihood` computes the log-likelihood over the whole dataset. The optimizer is in the main function, see section 2 b) Gradient Based Optimizer)

Visualization of Trained Weights:



Log-Likelihood Curve:



d) Average Log-Likelihood and Accuracy

```
Logistic Regression Model:
Average Log-Likelihood on Training Set: -20.741011784141325
Average Log-Likelihood on Test Set: -23.847041204038447
Training Accuracy: 0.9098
Testing Accuracy: 0.8921
```

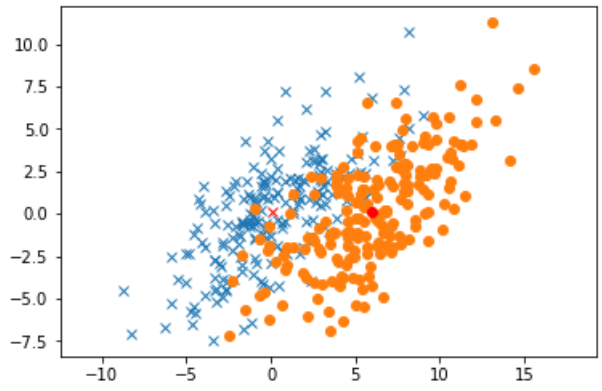
Logistic Regression acheived better performance than Naïve Bayes on the training set and test set.

4. Unsupervised Learning

a) Generate Data

Visualization of true cluster assignments generated from two bivariate normal distributions (Random SEED set to 2019 for reproducibility):

$$\mu_1 = \begin{bmatrix} 0.1 \\ 0.1 \end{bmatrix}, \mu_2 = \begin{bmatrix} 6.0 \\ 0.1 \end{bmatrix}, \Sigma_1 = \Sigma_2 = \begin{bmatrix} 10 & 7 \\ 7 & 10 \end{bmatrix}$$



b) K Nearest Neighbour

(See code in Appendix 3) KNN.py)

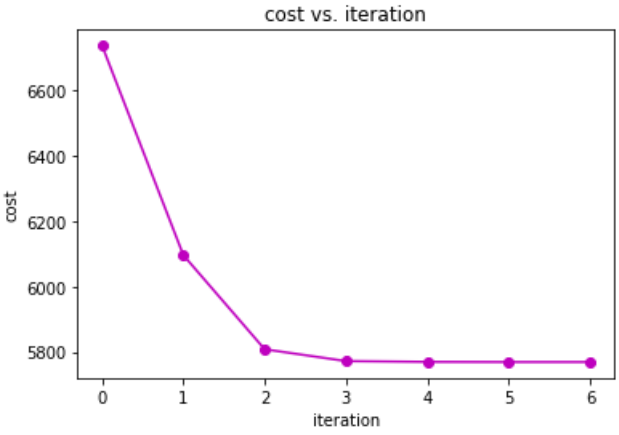
Initialize cluster means:

$$\hat{\mu}_1 = \begin{bmatrix} 0.0 \\ 0.0 \end{bmatrix}, \hat{\mu}_2 = \begin{bmatrix} 1.0 \\ 1.0 \end{bmatrix}$$

Results:

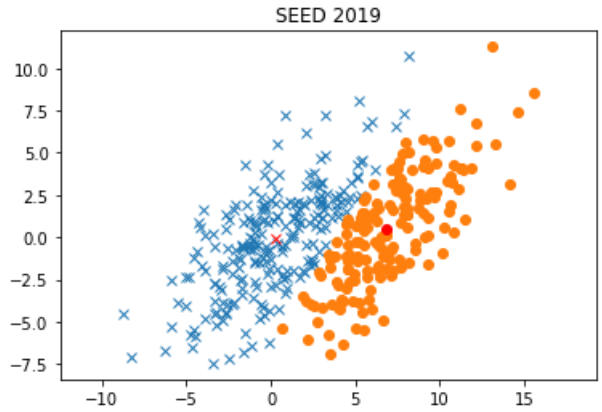
```
In 9 steps, KNN converges to the minimized cost: 5768.860682
The miss-classification error is: 0.26
```

Cost Curve:



(1 Iteration means 1 E-step and 1 M-Step)

KNN Assignments:



c) Expectation Maximization

(See code in Appendix 4) EM.py)

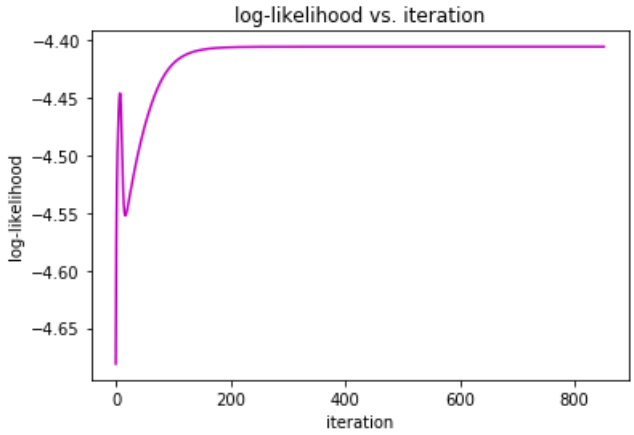
Initialization:

$$\hat{\mu}_1 = \begin{bmatrix} 0.0 \\ 0.0 \end{bmatrix}, \hat{\mu}_2 = \begin{bmatrix} 1.0 \\ 1.0 \end{bmatrix}, \hat{\Sigma}_1 = \hat{\Sigma}_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \hat{\pi}_1 = \hat{\pi}_2 = 0.5$$

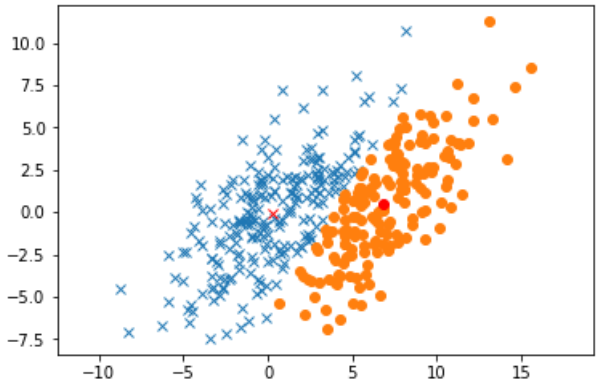
Results:

```
In 852 steps, EM converges to the maximized log-likelihood: -4.405573
The miss-classification error is: 0.14
```

Log-Likelihood Curve:



EM Assignments:



d) Experiment with Data Realizations

While EM correctly identifies clusters from other data generations (e.g. SEED = 2018), when the random SEED is set to 2017, the algorithm fails to identify and seperate the two clusters. The two means are close together as we can see in Figure 2, with only a few data points classified as the second cluster. Therefore, the algorithm performance does depend on different realizations of data.

Results:

```
SEED = 2018
In 759 steps, EM converges to the maximized log-likelihood: -5.498959
The miss-classification error is: 0.08

SEED = 2017
In 4011 steps, EM converges to the maximized log-likelihood: -6.332573
The miss-classification error is: 0.49
```

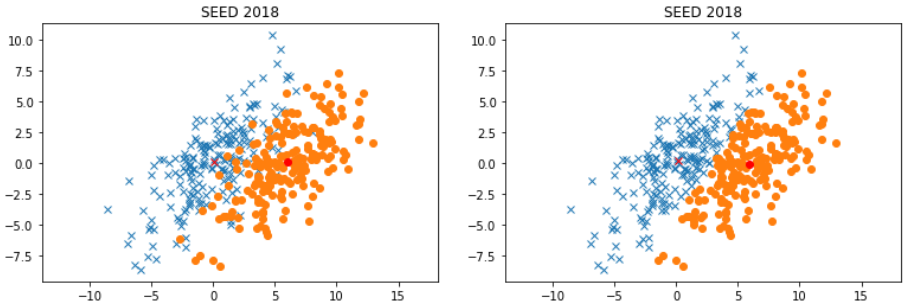


Figure 1: True Assignments vs. EM Result

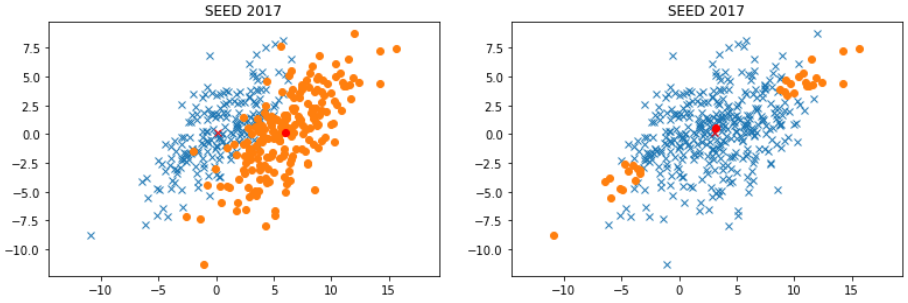


Figure 2: True Assignments vs. EM Result

Appendix

1) naive_bayes.py

```
import loadMNIST as load
import matplotlib.pyplot as plt
import numpy as np


def binarize(train_x):
    train_x[train_x >= 0.5] = 1
    train_x[train_x < 0.5] = 0
    return train_x

def fit(train_x, train_c):
    D = train_x.shape[1]
    K = train_c.shape[1]
    # Number of images that belongs to each class
    class_counts = np.sum(train_c, axis = 0)
    thetas = np.empty([K, D])
    for c in range(K):
        indices_c = np.where(np.argmax(train_c, axis =1) == c)[0]
        images_c = train_x[indices_c]
        pixel_sums_c = np.sum(images_c, axis = 0) # pixel sums of each class
        thetas[c] = (1 + pixel_sums_c) / (2 + class_counts[c])
    return thetas

def image_LL(image, label, thetas, pis):
    D = len(image)
    marginal = 0
    for c in range(10):
        product = 1
        for d in range(D):
            product *= thetas[c,d]**image[d] * (1-thetas[c,d])**(1-image[d])
        if label == c:
            joint = pis[c] * product
        marginal += pis[c] * product
    return joint/marginal

def avg_LL(images, labels, thetas, pis):
    sum_LL = 0
    N_data = images.shape[0]
    for n in range(N_data):
        image = images[n]
        label = np.argmax(labels[n])
        sum_LL += np.log(image_LL(image, label, thetas, pis))
```

```
    return (1/N_data) * sum_LL

def perdict(image, thetas, pis):
    D = len(image)
    K = len(pis)
    joint_dists = [0]*K
    for c in range(K):
        product = 1
        for d in range(D):
            product *= thetas[c,d]**image[d] * (1-thetas[c,d])**(1-image[d])
        joint_dists[c] = pis[c] * product
    return np.argmax(joint_dists)

def accuracy(images, labels, thetas, pis):
    correct_count = 0
    N = images.shape[0]
    labels = np.argmax(labels, axis =1)
    for n in range(N):
        if perdict(images[n], thetas, pis) == labels[n]:
            correct_count += 1
    return correct_count/N_data

def sample_image(thetas, pis):
    D = thetas.shape[1]
    K = len(pis)
    c = np.random.choice(K, 1, p=pis)
    theta_c = thetas[c].flatten()
    image = np.zeros(D)
    for d in range(D):
        image[d] = np.random.binomial(1, p=theta_c[d])
    return image

def sample_half(image, thetas, pis):
    D = thetas.shape[1]
    K = len(pis)
    class_sum = np.zeros(K)
    cut = int(D/2)
    top = range(cut)
    bottom = range(cut, D)
    # denominator
    for c in range(K):
        Theta_X = thetas[c, top]**image[top] \
            * (1 - thetas[c, top])**(1 - image[top])
        class_sum[c] = np.prod(Theta_X)
    denom = np.sum(class_sum)
    # numerator
```

```
    numer = np.zeros(cut)
    for d in bottom:
        for c in range(K):
            numer[d - cut] += class_sum[c] * thetas[c,d]
    image[bottom] = numer/denom
    return image


if __name__ == "__main__":

# ================================================================================
#     Load and Store Dataset
# ================================================================================

    training_cutoff = 10000
    debug_cutoff = 100
    image_size = 784

    N_data, train_images, train_labels, \
        test_images, test_labels = load.load_mnist()

    data = {'train_x': binarize(train_images[:training_cutoff]),
            'train_c': train_labels[:training_cutoff],
            'test_x': test_images, 'test_c': test_labels}

    debug_data = {'train_x': binarize(train_images[:debug_cutoff]),
                  'train_c': train_labels[:debug_cutoff],
                  'test_x': test_images, 'test_c': test_labels}

    train_x = data['train_x']
    train_c = data['train_c']

    test_x = data['test_x']
    test_c = data['test_c']

# ================================================================================
#     1 c) Thetas
# ================================================================================

    thetas = fit(train_x, train_c)

    fig = plt.figure(1)
    fig.clf()
    ax = fig.add_subplot(111)
    load.plot_images(thetas, ax)
    plt.show()
```

```
# ======================================================================
#     1 e) Average Likelihood and Accuracy
# ======================================================================

    pis = [0.1]*10
    avg_LL_train = avg_LL(train_x, train_c, thetas, pis)
    avg_LL_test = avg_LL(test_x, test_c, thetas, pis)
    accuracy_train = accuracy(train_x, train_c, thetas, pis)
    accuracy_test = accuracy(test_x, test_c, thetas, pis)

    print('Naive Bayes Model:')
    print('Average Log-Likelihood on Training Set:', avg_LL_train)
    print('Average Log-Likelihood on Test Set:', avg_LL_test)
    print('Training Accuracy:', accuracy_train)
    print('Testing Accuracy:', accuracy_test)

# ======================================================================
#     2 c) Sample 10 binary images
# ======================================================================

    print('Sample 10 Images from Marginal Distribution p(x)')

    ten_samples = [sample_image(thetas, pis) for _ in range(10)]
    ten_samples = np.array(np.stack(ten_samples, axis=0))

    fig = plt.figure(1)
    fig.clf()
    ax = fig.add_subplot(111)
    load.plot_images(ten_samples, ax)
    plt.show()

# ======================================================================
#     2 e) Generate 20 half images
# ======================================================================

    print('Sample 20 half images from Training Set with p(bottom | top)')

    images = train_x[np.random.choice(image_size,20),:]

    twenty_half_samples = [sample_half(image, thetas, pis) for image in images]
    twenty_half_samples = np.array(np.stack(twenty_half_samples, axis=0))

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

```
    load.plot_images(twenty_half_samples, ax)
    plt.show()
```

2) logistic_regression.py

```
import loadMNIST as load
import matplotlib.pyplot as plt
import numpy as np
#from autograd import grad
#from autograd import scipy
from scipy import special

def binarize(train_x):
    train_x[train_x >= 0.5] = 1
    train_x[train_x < 0.5] = 0
    return train_x

def sum_exp(x):
   e_x = np.exp(x - np.max(x))
   return e_x / e_x.sum()

def grad_LL(W, train_x, train_c):

    N = train_x.shape[0]
    D = train_x.shape[1]
    K = train_c.shape[1]

    gradient = np.zeros((D,K))
    labels = np.argmax(train_c, axis=1)
    WX = np.matmul(train_x, W)

    for n in range(N):
        wx_n = WX[n]
        exp_wx_n = np.exp(wx_n - np.max(wx_n))
        perdict_n = exp_wx_n / np.sum(exp_wx_n)
        x = train_x[n]
        for c in range(K):
            if labels[n] == c:
                gradient[:,c] -= x * (perdict_n[c] - 1)
            else:
                gradient[:,c] -= x * perdict_n[c]
    return gradient

def log_likelihood(W, train_x, train_c):
   N = train_x.shape[0]
   labels = np.argmax(train_c, axis=1)
```

```
    wcTx = 0
    LSE = 0
    WX = np.matmul(train_x, W)
    for n in range(N):
        wcTx += np.matmul(train_x[n], W[:, labels[n]])
        # log of sum (over c) of exp(wcTx)
        LSE += special.logsumexp(WX[n])
    return (wcTx - LSE)

def accuracy(W, train_x, train_c):
    N = train_x.shape[0]
    labels = np.argmax(train_c, axis=1)
    WX = np.matmul(train_x, W)
    predictions = np.zeros(N)
    for n in range(N):
        wx_n = WX[n]
        exp_wx_n = np.exp(wx_n - np.max(wx_n))
        perdict_n = exp_wx_n / np.sum(exp_wx_n)
        predictions[n] = np.argmax(perdict_n)
    correct_count = len(np.where(predictions == labels)[0])
    return correct_count/N


if __name__ == "__main__":

# ================================================================================
#     Load and Store Dataset
# ================================================================================

    training_cutoff = 10000
    debug_cutoff = 100
    image_size = 784

    N_data, train_images, train_labels, test_images, test_labels = load.load_mni

    data = {'train_x': binarize(train_images[:training_cutoff]),
            'train_c': train_labels[:training_cutoff],
            'test_x': test_images, 'test_c': test_labels}

    debug_data = {'train_x': binarize(train_images[:debug_cutoff]),
                  'train_c': train_labels[:debug_cutoff],
                  'test_x': test_images, 'test_c': test_labels}

    train_x = data['train_x']
    train_c = data['train_c']
```

```
    test_x = data['test_x']
    test_c = data['test_c']

# ======================================================================
#     2 b) Gradient Based Optimizer
# ======================================================================

    N = train_x.shape[0]
    K = train_c.shape[1]
    D = train_x.shape[1]

    initial_W = np.zeros((D, K))
    rate = 0.01
    steps = 10000

    W = initial_W
    n_step = 0
    old_LL = 0
    LL = log_likelihood(W, train_x, train_c)
    LL_history = [LL]

    while (abs(LL - old_LL) != 0):
        old_LL = LL
        gradient = grad_LL(W, train_x, train_c)
        rate = 0.05/(n_step + 1)
        W += rate * gradient
        LL = log_likelihood(W, train_x, train_c)
        LL_history.append(LL)
        n_step += 1
        print('epoch: %d | log-likelihood: %f'  % (n_step, LL))

    plt.plot(np.arange(len(LL_history[:])), LL_history[:], 'm-')


# log-likelihood curve

    plt.title('log-likelihood vs. iteration')
    plt.ylabel('log-likelihood')
    plt.xlabel('iteration')
    plt.plot(np.arange(n_step + 1), LL_history, 'm-')
    plt.show()

    # In 3488 steps, Logistic Regression converges to the maximized log-likeliho
    print('In %d steps, Logistic Regression converges to the maximized log-likel

    fig = plt.figure(1)
```

```
    fig.clf()
    ax = fig.add_subplot(111)
    load.plot_images(W.T, ax)
    plt.show()

    avg_LL_train = log_likelihood(W, train_x, train_c)/N
    avg_LL_test = log_likelihood(W, test_x, test_c)/N

    accuracy_train = accuracy(W, train_x, train_c)
    accuracy_test = accuracy(W, test_x, test_c)

# Average Likelihood and Accuracy

    print('Logistic Regression Model:')
    print('Average Log-Likelihood on Training Set:', avg_LL_train)
    print('Average Log-Likelihood on Test Set:', avg_LL_test)
    print('Training Accuracy:', accuracy_train)
    print('Testing Accuracy:', accuracy_test)
```

3) KNN.py

```
import matplotlib.pyplot as plt
import numpy as np

def binarize(train_x):
    train_x[train_x >= 0.5] = 1
    train_x[train_x < 0.5] = 0
    return train_x

def cost(X, R, MU):
    J = 0
    for k in range(K):
        for n in range(N):
            J += R[n,k] * np.linalg.norm(X[n,] - MU[k,])**2
    return J

def km_e_estep(X, MU):
    N = X.shape[0]
    K = MU.shape[0]
    R = np.zeros((N, K)) # N x K
    for k in range(K):
        for n in range(N):
            l = [np.linalg.norm(X[n,] - MU[k,])**2 for k in range(K)]
            k_min = np.argmin(l)
            R[n,k] = 1 if k == k_min else 0
    return R
```

```
def km_m_estep(X, R):
    # X: 400 x 2
    # R: 400 x 2
    K = R.shape[1]
    MU = np.zeros((K, D)) # K x D
    K = R.shape[1]
    for k in range(K):
        numer = np.matmul(X.T, R[:, k]) # 784 x 1
        denom = np.sum(R[:, k])
        MU[k] = numer/denom
    return MU

def accuracy(X, R):

    N_data = X.shape[0]
    cutoff = int(N_data/2)

    true_id1 = np.arange(cutoff)
    true_id2 = np.arange(cutoff, N_data)

    R_id1 = np.where(R[:,0] == 1)[0]
    R_id2 = np.where(R[:,0] == 0)[0]

    correct_count_1 = len(list(set(true_id1).intersection(R_id1)))
    correct_count_2 = len(list(set(true_id2).intersection(R_id2)))

    return (correct_count_1 + correct_count_2) / N_data



if __name__ == "__main__":

# ========================================================================
#     4 a) Data Generation
# ========================================================================

    m1 = [0.1, 0.1]
    m2 = [6.0, 0.1]
    cov = [[10, 7], [7, 10]]

    np.random.seed(2019)

    x1, y1 = np.random.multivariate_normal(m1, cov, 200).T
    x2, y2 = np.random.multivariate_normal(m2, cov, 200).T
    plt.plot(x1, y1, 'x')
```

```
    plt.plot(m1[0], m1[1], 'x', color='red')

    plt.plot(x2, y2, 'o')
    plt.plot(m2[0], m2[1], 'o', color='red')
    plt.axis('equal')

    plt.show()

# ================================================================================
#     4 b) KNN
# ================================================================================

### Training

    K = 2
    D = 2
    N = 400

    m1_init = [0.0, 0.0]
    m2_init = [1.0, 1.0]

    X = np.vstack((np.stack((x1, y1)).T, np.stack((x2, y2)).T))
    MU = np.array([m1_init, m2_init])
    R = km_e_estep(X, MU)

    n_step = 0
    J_old = 0
    J = cost(X, R, MU)
    cost_history = [J]

    while (J - J_old != 0):
        # E Step
        R = km_e_estep(X, MU)
        # M Step
        MU = km_m_estep(X, R)
        # cost
        J_old = J
        J = cost(X, R, MU)
        print('epoch: %d | cost: %f'  % (n_step, J))
        cost_history.append(J)
        n_step += 1

### Cost Curve

    plt.title('cost vs. iteration')
    plt.ylabel('cost')
```

```
    plt.xlabel('iteration')
    plt.plot(np.arange(n_step + 1), cost_history, 'mo-')
    plt.show()

    print('In %d steps, KNN converges to the minimized cost: %f'  %(n_step, J))

### Plot Result

    x = X[:,0]
    y = X[:,1]

    id1 = np.where(R[:,0] == 1)[0]
    id2 = np.where(R[:,0] == 0)[0]

    x1 = x[id1]
    y1 = y[id1]
    plt.plot(x1, y1, 'x')
    plt.plot(MU[0,0], MU[0,1], 'x', color='red')

    x2 = x[id2]
    y2 = y[id2]
    plt.plot(x2, y2, 'o')
    plt.plot(MU[1,0], MU[1,1], 'o', color='red')

    plt.axis('equal')
    plt.show()

### Miss-Classification Error

    missed_rate = 1 - accuracy(X, R)
    print('The miss-classification error is: %.2f'  % missed_rate)
```

4) EM.py

```
import matplotlib.pyplot as plt
import numpy as np

def binarize(train_x):
    train_x[train_x >= 0.5] = 1
    train_x[train_x < 0.5] = 0
    return train_x

def cost(X, R, MU):
    J = 0
    for k in range(K):
        for n in range(N):
```

```
            J += R[n,k] * np.linalg.norm(X[n,] - MU[k,])**2
    return J

def km_e_estep(X, MU):
    N = X.shape[0]
    K = MU.shape[0]
    R = np.zeros((N, K)) # N x K
    for k in range(K):
        for n in range(N):
            l = [np.linalg.norm(X[n,] - MU[k,])**2 for k in range(K)]
            k_min = np.argmin(l)
            R[n,k] = 1 if k == k_min else 0
    return R

def km_m_estep(X, R):
    # X: 400 x 2
    # R: 400 x 2
    K = R.shape[1]
    MU = np.zeros((K, D)) # K x D
    K = R.shape[1]
    for k in range(K):
        numer = np.matmul(X.T, R[:, k]) # 784 x 1
        denom = np.sum(R[:, k])
        MU[k] = numer/denom
    return MU

def accuracy(X, R):

    N_data = X.shape[0]
    cutoff = int(N_data/2)

    true_id1 = np.arange(cutoff)
    true_id2 = np.arange(cutoff, N_data)

    R_id1 = np.where(R[:,0] == 1)[0]
    R_id2 = np.where(R[:,0] == 0)[0]

    correct_count_1 = len(list(set(true_id1).intersection(R_id1)))
    correct_count_2 = len(list(set(true_id2).intersection(R_id2)))

    return (correct_count_1 + correct_count_2) / N_data

if __name__ == "__main__":

# ========================================================================
#     4 a) Data Generation
```

```
# ============================================================================

    m1 = [0.1, 0.1]
    m2 = [6.0, 0.1]
    cov = [[10, 7], [7, 10]]

    np.random.seed(2019)

    x1, y1 = np.random.multivariate_normal(m1, cov, 200).T
    x2, y2 = np.random.multivariate_normal(m2, cov, 200).T
    plt.plot(x1, y1, 'x')
    plt.plot(m1[0], m1[1], 'x', color='red')

    plt.plot(x2, y2, 'o')
    plt.plot(m2[0], m2[1], 'o', color='red')
    plt.axis('equal')

    plt.show()

# ============================================================================
#     4 b) KNN
# ============================================================================

### Training

    K = 2
    D = 2
    N = 400

    m1_init = [0.0, 0.0]
    m2_init = [1.0, 1.0]

    X = np.vstack((np.stack((x1, y1)).T, np.stack((x2, y2)).T))
    MU = np.array([m1_init, m2_init])
    R = km_e_estep(X, MU)

    n_step = 0
    J_old = 0
    J = cost(X, R, MU)
    cost_history = [J]

    while (J - J_old != 0):
        # E Step
        R = km_e_estep(X, MU)
        # M Step
        MU = km_m_estep(X, R)
```

```
        # cost
        J_old = J
        J = cost(X, R, MU)
        print('epoch: %d | cost: %f'  % (n_step, J))
        cost_history.append(J)
        n_step += 1

### Cost Curve

    plt.title('cost vs. iteration')
    plt.ylabel('cost')
    plt.xlabel('iteration')
    plt.plot(np.arange(n_step + 1), cost_history, 'mo-')
    plt.show()

    print('In %d steps, KNN converges to the minimized cost: %f'  %(n_step, J))

### Plot Result

    x = X[:,0]
    y = X[:,1]

    id1 = np.where(R[:,0] == 1)[0]
    id2 = np.where(R[:,0] == 0)[0]

    x1 = x[id1]
    y1 = y[id1]
    plt.plot(x1, y1, 'x')
    plt.plot(MU[0,0], MU[0,1], 'x', color='red')

    x2 = x[id2]
    y2 = y[id2]
    plt.plot(x2, y2, 'o')
    plt.plot(MU[1,0], MU[1,1], 'o', color='red')

    plt.axis('equal')
    plt.show()

### Miss-Classification Error

    missed_rate = 1 - accuracy(X, R)
    print('The miss-classification error is: %.2f'  % missed_rate)
```