1. Class-Conditional Gaussians

1.1 According to Bayes' rule, we have:

$$p(y=k|\mathbf{x},\boldsymbol{\mu},\boldsymbol{\sigma}) = \frac{p(\mathbf{x}|y=k,\boldsymbol{\mu},\boldsymbol{\sigma})p(y=k)}{p(\mathbf{x}|\boldsymbol{\mu},\boldsymbol{\sigma})}$$

Prior probability:

$$p(y=k) = \alpha_k$$

Likelihood:

$$p(\mathbf{x}|y=k,\boldsymbol{\mu},\boldsymbol{\sigma}) = (\prod_{i=1}^{D} 2\pi\sigma_i^2)^{-1/2}\exp\{-\sum\nolimits_{i=1}^{D} \frac{1}{2\sigma_i^2}(x_i-\mu_{ki})^2\}$$

Normalized constant:

$$p(\mathbf{x}|\boldsymbol{\mu},\boldsymbol{\sigma}) = \sum_{y=1}^{\mathrm{K}} p(\mathbf{x}|y=k,\boldsymbol{\mu},\boldsymbol{\sigma})p(y=k)$$

So the posterior probability:

$$p(y=k|\mathbf{x},\boldsymbol{\mu},\boldsymbol{\sigma}) = \frac{(\prod_{i=1}^{D} 2\pi\sigma_i^2)^{-1/2}\exp\{-\sum_{i=1}^{D}\frac{1}{2\sigma_i^2}(x_i-\mu_{ki})^2\}\alpha_k}{\sum_{k=1}^{k}(\prod_{i=1}^{D} 2\pi\sigma_i^2)^{-1/2}\exp\{-\sum_{i=1}^{D}\frac{1}{2\sigma_i^2}(x_i-\mu_{ki})^2\}\alpha_k}$$

$$= \frac{\exp\{-\sum_{i=1}^{D}\frac{1}{2\sigma_i^2}(x_i-\mu_{ki})^2\}\alpha_k}{\sum_{k=1}^{k}\exp\{-\sum_{i=1}^{D}\frac{1}{2\sigma_i^2}(x_i-\mu_{ki})^2\}\alpha_k}$$

1.2 Likelihood $L(\theta) = p(\mathbf{x},\mathbf{y}|\theta) = p(\mathbf{x}|\mathbf{y},\theta)p(\mathbf{y}|\theta)$

$$\ell(\theta;D) = -logL(\theta) = -logp(\mathbf{x}|\mathbf{y},\theta) - logp(\mathbf{y}|\theta)$$

$$= \sum\nolimits_{i=1}^{d} \frac{1}{2}\log(2\pi\sigma_i^2) + \frac{1}{2\sigma_i^2}(x_i-\mu_{ki})^2 - log\alpha_{y^{(i)}}$$

1.3 The partial derivatives of the likelihood

$$\frac{\partial\, log\, L}{\partial\, \mu_{ki}} = -\sum_{i=0}^{N} 1(y^{(i)}=k)\frac{x^{(i)}-\mu_{ki}}{\sigma_i^2}$$

I use reciprocal of the variances instead of the variances. Because, they will get the same answer

$$\frac{\partial \log L}{\partial {\sigma_i}^{-2}} = -\sum_{i=0}^{N} \mathbf{1}\big(y^{(i)} = k\big)[-\frac{\sigma_i^2}{2} + \frac{1}{2}(x_i - \mu_{ki})^2]$$

1.4

$$\frac{\partial \log L}{\partial \mu_k} = 0 \qquad \mu_k = \frac{\sum_{i=1}^{N} \mathbf{1}(y^{(i)} = k) x^{(i)}}{\sum_{i=1}^{N} \mathbf{1}(y^{(i)} = k)}$$

$$\frac{\partial \log L}{\partial ({\sigma_i}^2)^{-1}} = 0 \qquad {\sigma_i}^2 = \frac{\sum_{i=1}^{N} \mathbf{1}\big(y^{(i)} = k\big)\,(x_i - \mu_{ki})^2}{\sum_{i=1}^{N} \mathbf{1}(y^{(i)} = k)}$$

1.5 Same as the situation in Bernoulli:

$$\frac{\partial L(\theta)}{\partial \alpha_k} + \lambda \frac{\partial L(\theta)}{\partial \alpha_k} = 0 \quad \Rightarrow \lambda = -\sum_{i=1}^{N} \mathbf{1}\big(y^{(i)} = k\big)\frac{1}{\alpha_k}$$

$$\alpha_k = -\frac{\sum_{i=1}^{N} \mathbf{1}(y^{(i)} = k)}{\lambda}$$

Considering the constraint: $\sum_k \alpha_k = 1 \;\Rightarrow\; \lambda = -N$

$$\alpha_k = \frac{\sum_{i=1}^{N} \mathbf{1}(y^{(i)} = k)}{N}$$

2 Handwritten Digit Classification:

2.0

In this part, I use load_all_data_from_zip() fiction to load the data from a2digits.zip. Then, I used np.mean() faction to calculate each mean value of 10 digits and used plt.imshow() faction to output the pictures.

Code:

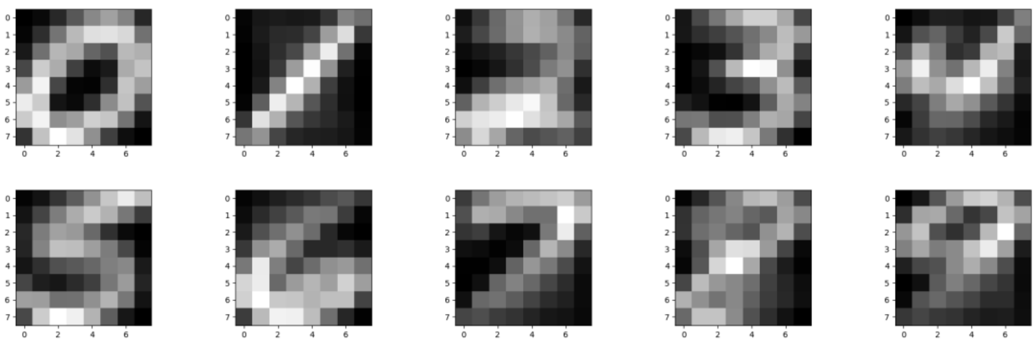
```
def plot_means(train_data):

    one_digit = 700
    mean = np.zeros((10, 64))
    for i in range(10):
        mean[i] = np.mean(train_data[i*one_digit:(i+1)*one_digit], axis=0)
        # Compute mean of class

    for i in range(0, 10):
        plt.subplot(2, 5, i+1)
        plt.imshow(mean[i].reshape((8, 8)), cmap='gray')

    # Plot all means on same axis
    plt.tight_layout()
    plt.show()
```

Result:



2.1 K-NN Classifier

2.1.1:

My method to solve the knn algorithm:

```
def query_knn(self, test_point, k):
    '''
    Query a single test point using the k-NN algorithm

    You should return the digit label provided by the algorithm
    '''
    l2 = self.l2_distance(test_point)
    sorted_l2 = np.argsort(l2)
    classCount = {}
    for i in range(k):
        voteLabel = self.train_labels[sorted_l2[i]]
        classCount[voteLabel] = classCount.get(voteLabel, 0) + 1
    maxCount = 0
    for key, value in classCount.items():
        if value > maxCount:
            maxCount = value
            maxIndex = key
    return maxIndex
```

The method to calculate the accuracy:

```
classification_accuracy(knn, k, eval_data, eval_labels):
'''
Evaluate the classification accuracy of knn on the given 'eval_data'
using the labels
'''
res = np.zeros(eval_data.shape[0])
for i in range(eval_data.shape[0]):
    res[i] = knn.query_knn(eval_data[i], k) == eval_labels[i]
return res.mean()
```

Result:

```
Accuracy for train data with k=1:  1.0
Accuracy for train data with k=15:  0.963714285714
Accuracy for test data with k=1:  0.96875
Accuracy for test data with k=15:  0.961
```

(a) For K=1 the train and test classification accuracy are 1.0 and 0.96975 respectively.

(b) For K=15 the train and test classification accuracy are 0.9637 and 0.961 respectively.

2.1.2:

Ties that need to be broken to make a decision can be following:

(1) If we set K=1, there are two points have the same Euclidean distance.

(2) If all the points have the same Euclidean distance.

Solution:

(1) Choose a different k

(2) Choose a point randomly

2.1.3

By using cross-validation to find the best K:

```python
def cross_validation(knn, k_range=np.arange(1, 16)):
    optimal_k = 0
    optimal_accuracy = 0
    for k in k_range:
        # Loop over folds
        # Evaluate k-NN
        # ...
        accuracies = k_fold(knn.train_data, knn.train_labels, k)
        print('accuracy ', accuracies, ' of ', k)

        if accuracies.mean() >= optimal_accuracy:
            optimal_accuracy = accuracies.mean()
            optimal_k = k

    return optimal_k, optimal_accuracy
```

```python
def k_fold(data, label, k):
    i = 0
    Kf = KFold(10, shuffle=True)
    accuracies = np.zeros(10)
    for index_train, index_valid in Kf.split(data, label):
        train_x, valid_x = data[index_train], data[index_valid]
        train_y, valid_y = label[index_train], label[index_valid]
        knn = KNearestNeighbor(train_x, train_y)
        accuracies[i] = classification_accuracy(knn, k, valid_x, valid_y)
        i = i + 1
    return accuracies.mean()
```

Result: sometimes I get k=4 and sometimes I get k=3

```
Optimal K for KNN and the corresponding mean k_fold loss:  4 & 0.967
Accuracy for train data with optimal k:  0.986428571429
Accuracy for test data with optimal k:  0.97275
```

K=4 the train classification is 0.967, the average accuracy across folds is 0.986 and the test accuracy across folds is 0.973

2.2 Conditional Gaussian Classifier Training

2.2.1

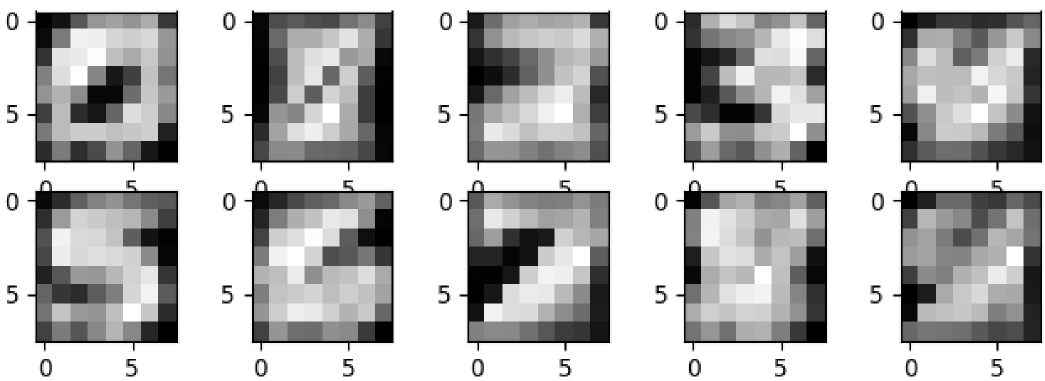
I calculate the covariance by faction:

```python
def generative_likelihood(digits, means, covariances):
    '''
    Compute the generative log-likelihood:
        log p(x|y,mu,Sigma)

    Should return an n x 10 numpy array
    '''
    logZ = np.zeros(10)
    for i in range(10):
        # print(cov[i].shape)
        logZ[i] = np.log(np.sqrt(((2 * np.pi) ** 64) * np.linalg.det(covariances[i])))

    likelihood = np.zeros((digits.shape[0], 10))
    for i in range(10):
        x_minus_mean=digits-means[i]
        for n,data in enumerate(x_minus_mean):
            likelihood[n][i] = -(logZ[i]+0.5*np.dot(np.dot(data.T,np.linalg.inv(covariances[i])),data))
    return likelihood
```

The picture of 8x8 image of the log of the diagonal elements of each covariance matrix.



2.2.2

I used the parameters you fit on the training set and Bayes rule, compute the average conditional log-likelihood as shown in the table.

```python
def avg_conditional_likelihood(digits, labels, means, covariances, stem):
    '''
    Compute the average conditional likelihood over the true class labels

        AVG( log p(y_i|x_i, mu, Sigma) )

    i.e. the average log likelihood that the model assigns to the correct class label
    '''
    sum = 0
    cond_likelihood = conditional_likelihood(digits, means, covariances)
    sum = 0
    for n in range(digits.shape[0]):
        sum += cond_likelihood[n][int(labels[n])]
    AVG = sum/digits.shape[0]
    print('Average conditional likelihood for ' + stem + 'data in correct class is: ', AVG)
```

Result:

Average conditional likelihood for train data in correct class is: -0.1246

Average conditional likelihood for test data in correct class is: -0.1967

```
Train_data:
Average conditional likelihood for train_data in correct class is:  -0.124624436669
Test_data:
Average conditional likelihood for test_data in correct class is:  -0.196673203255
```

2.2.3

I used np.argmax(cond_likelihood, axis=1) to calculate the most likely posterior class for each training and test data.

```python
def classify_data(digits, means, covariances):
    '''
    Classify new points by taking the most likely posterior class
    '''
    cond_likelihood = conditional_likelihood(digits, means, covariances)
    # Compute and return the most likely class
    return np.argmax(cond_likelihood, axis=1)
```

This faction is used to calculate the accuracy:

```python
def accuracy(labels, digits, means, covariance):
    '''
    acc = np.zeros(10)
    for i in range(10):
        acc[i] = labels[i] == classify_data(digits[i], means[i], covariance[i])
    return acc.mean()
    '''
    return np.equal(labels,classify_data(digits,means,covariance)).mean()
```

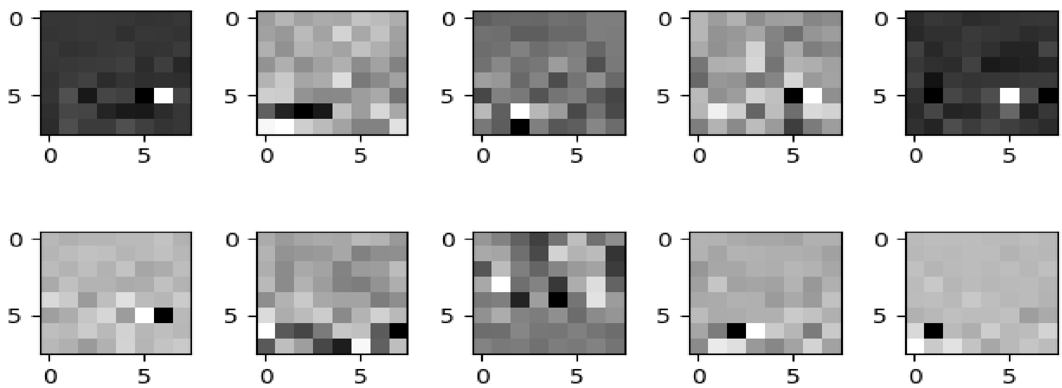
Result:

```
The accuracy of train data is:  0.981428571429
The accuracy of test data is:  0.97275
```

The accuracy on train set is 0.981 and the accuracy on test set is 0.97

2.2.4

```
def leading_eig(cov):
ted 2 blank lines, found 1
    1d_eig=np.zeros((10,64))
    for label in range(10):
        w,v = np.linalg.eig(cov[label])
        1d_eig[label] = v[np.argmax(w)]
    for i in range(10):
        plt.subplot(3, 5, i + 1)
        plt.imshow(1d_eig[i].reshape((8, 8)), cmap='gray')
    plt.tight_layout()
    plt.show()
```



2.3 Naïve Bayes Classifier Training

2.3.1

```python
def binarize_data(pixel_values):
    '''
    Binarize the data by thresholding around 0.5
    '''
    return np.where(pixel_values > 0.5, 1.0, 0.0)
```

In this section, numpy.where() faction is used to binarize the data by threshold 0.5.

2.3.2

I use the faction that $\eta_{kj,map} = \frac{N_{kj}+\alpha-1}{Nk+\alpha+\beta-2}$ Nk is the number of each digit, namely, 700, Nkj = the number of ones at each pixel of a certain digit. $\alpha = 2, \beta = 2$. In this section, we can see that $\eta_{kj} = \frac{N_k+1}{N+2}$, which means that adding two sample into the training data, one is all pixels on and one is all pixels off. So, I used $\eta_{kj} = \frac{N_k+1}{N+2}$ to calculate the eta.

```python
def compute_parameters(train_data):
    '''
    Compute the eta MAP estimate/MLE with augmented data

    You should return a numpy array of shape (10, 64)
    where the ith row corresponds to the ith digit class.
    '''
    eta = (train_data.sum(axis=1) + 1) / (train_data.shape[1] + 2)
    return eta
```

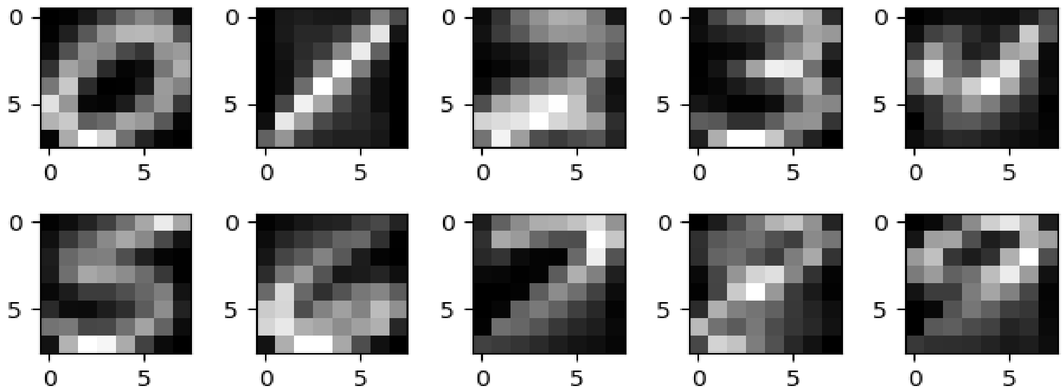
2.3.3

Pictures of η_k vectors

According to the value of η ,we build the images according to the η

```python
def plot_images(X):
    '''
    Plot each of the images corresponding to each class side by side in grayscale
    '''
    for i in range(10):
        plt.subplot(2, 5, i + 1)
        plt.imshow(X[i].reshape((8, 8)), cmap='gray')

    # Plot all means on same axis
    plt.tight_layout()
    plt.show()
```



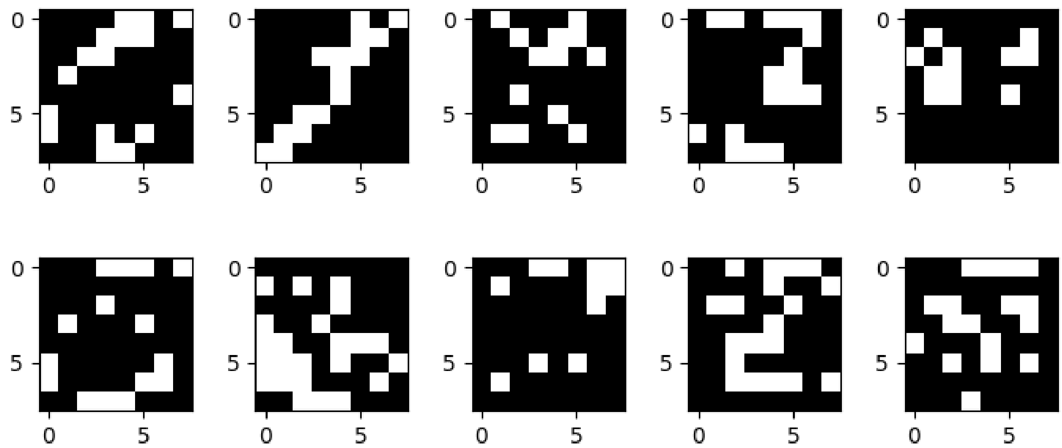
2.3.4

Picture of new data points

```
def generate_new_data(eta):
    '''
    Sample a new data point from your generative distribution p(x|y,theta) for
    each value of y in the range 0...10

    Plot these values
    '''

    plot_images(np.array([[np.random.binomial(1, eta[i, j]) for j in range(64)] for i in range(10)]))
```



2.3.5

Result:

```python
def conditional_likelihood(bin_digits, eta):
    '''
    Compute the conditional likelihood:

        log p(y|x, eta)

    This should be a numpy array of shape (n, 10)
    Where n is the number of datapoints and 10 corresponds to each digit class
    '''
    gene_likeihood = generative_likelihood(bin_digits, eta)
    cond = np.sum(np.exp(gene_likeihood - np.log(10)), axis=1).reshape((bin_digits.shape[0],))
    cond_standard = np.full((10, bin_digits.shape[0]), cond).T
    con_likelihood = (gene_likeihood - np.log(10)) - np.log(cond_standard)
    return con_likelihood
```

```python
def avg_conditional_likelihood(data, labels, eta):
    '''
    Compute the average conditional likelihood over the true class labels

        AVG( log p(y_i|x_i, eta) )

    i.e. the average log likelihood that the model assigns to the correct class label
    '''

    # Compute as described above and return
    cond_hd = conditional_likelihood(data, eta)
    sum_hd = 0
    for n in range(data.shape[0]):
        sum_hd += cond_hd[n][int(labels[n])]
    return sum_hd / data.shape[0]
```

The average conditional log-likelihood of training set is -0.944 and the average conditional log-likelihood of testing set is -0.987.

2.3.6

Result:

```python
def classify_data(data, eta):
    '''
    Classify new points by taking the most likely posterior class
    '''
    cond_likelihood = np.argmax(conditional_likelihood(data, eta), axis=1)
    # Compute and return the most likely class
    return cond_likelihood
```

```python
def accuracy(labels, data, eta):
    return np.equal(labels, classify_data(data, eta)).mean()
```

```
The accuracy for train data is:  0.774142857143
The accuracy for test data is:  0.76425
```

The accuracy of the train set is 0.774 and the accuracy of test set id 0.764

2.4 Performance

The conditional Gaussian performs the best and the Bernoulli naïve Bayes performs the worst.

In terms of the accuracy, the test accuracy of K-NN, conditional gaussian and naïve Bayes are 0.973, 0.973 and 0.764 respectively.
As for the running time, naïve Bayes has the shortest running time, following by the conditional gaussian and K-NN is the slowest one.

It matches with my exception, because the dependence of data let naïve Bayes performance bad. When using binarization, many data lost. And K-NN also play a good performance because the digits are not very complex, and the number of data is not very big, but it running time is much more longer than the other two. conditional Gaussian performs the best due to each data is assume to the multi-variable normal distribution.