CS 189 Introduction to Machine Learning
Spring 2021

HW3

Hieu Nguyen

SID: 26369732

Group: I worked with a Tree on this one, but I did all the work.

1 Academic Integrity Statement

"I certify that all solutions are entirely in my own words and that I have not looked at another student's solutions. I have given credit to all external sources I consulted."

Hieu Nguyen

– Hieu Nguyen

2 Gaussian Classification

1. Finding the Bayes optimal decision boundary for point(s) at which the posterior probabilities are equal

$$
\begin{aligned}
P(C_1|x) = P(C_2|x) &\iff f(x|C_1) = f(x|C_2) \\
&\iff \frac{(x-\mu_1)^2}{2\sigma^2} = \frac{(x-\mu_2)^2}{2\sigma^2} \\
&\iff (x-\mu_1)^2 = (x-\mu_2)^2 \\
&\iff x-\mu_1 = \mu_2 - x \\
&\iff x = \frac{\mu_1+\mu_2}{2}
\end{aligned}
$$

We have the Bayes optimal decision boundary: $\boxed{x = \frac{\mu_1+\mu_2}{2}}$

And the corresponding Bayes decision rule is that for any data point $x \in \mathbb{R}$, if $x < \frac{\mu_1+\mu_2}{2}$, then x is classified as class 1; and if $x > \frac{\mu_1+\mu_2}{2}$, then x is classified as class 2.

2. Supposed the decision boundary for the classifier is $x = b$. The Bayes error is given by

$$P_e = P((C_1 \text{ misclassified as } C_2) \cup (C_2 \text{ misclassified as } C_1))$$

which is equivalent to

$$P_e = P((\text{ misclassified as } C_1|C_2)P(C_2) + P((\text{ misclassified as } C_2|C_1)P(C_1)$$

For the first part, $P((\text{ misclassified as } C_1|C_2)P(C_2)$,

$$P((\text{ misclassified as } C_1|C_2)P(C_2) = \frac{1}{2}\int_{-\infty}^{b} \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-\mu_2)^2/(2\sigma)^2}\, dx$$

Similarly, we have,

$$P((\text{ misclassified as } C_2|C_1)P(C_1) = \frac{1}{2}\int_{b}^{\infty} \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-\mu_1)^2/(2\sigma)^2}\, dx$$

Therefore, we have

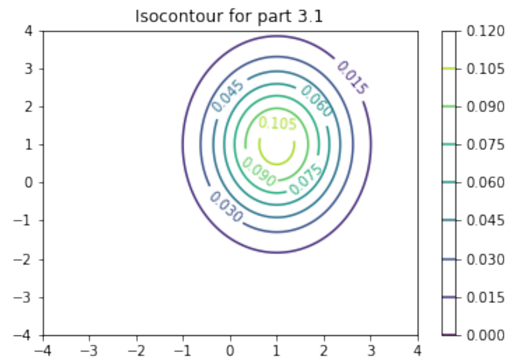
$$
\begin{gathered}
P_e = P((C_1 \text{ misclassified as } C_2) \cup (C_2 \text{ misclassified as } C_1)) = \\
\frac{1}{2\sqrt{2\pi}\sigma}\left(\int_{-\infty}^{b} e^{-(x-\mu_2)^2/(2\sigma)^2}\, dx + \int_{b}^{\infty} e^{-(x-\mu_1)^2/(2\sigma)^2}\, dx\right)
\end{gathered}
$$

3. We want to calculate the optimal decision boundary b^* that minimize $P_e(b)$, given that $P_e(b)$ is convex for $\mu_1 < b < \mu_2$

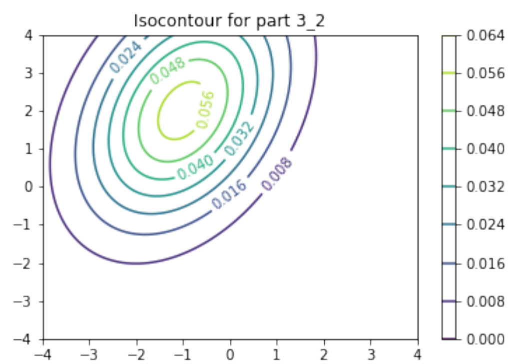
3 Isocontours of Normal Distributions

For the code, please see the appendix.

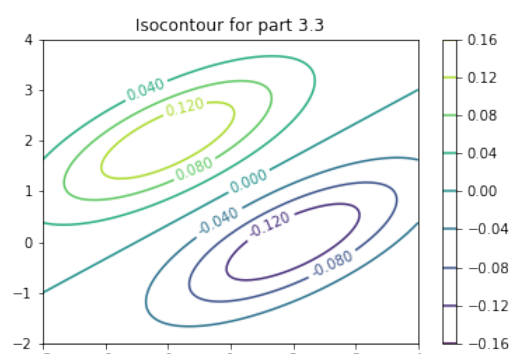
1. Picture



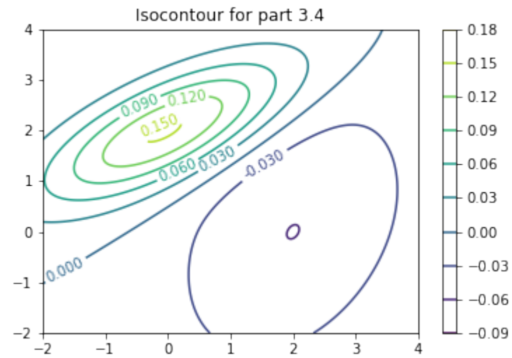
2. Picture



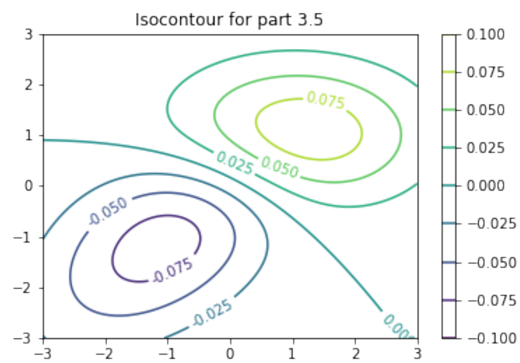
3. Picture



4. Picture



5. Picture



4 Eigenvectors of the Gaussian Covariance Matrix

(a) (Please see the appendix for the code) The mean (in $\mathbb{R}^2$) of the sample is $[3.31998 \;\; 5.423675]$.

(b) (Please see the appendix for the code) The 2×2 covariance matrix of the sample Σ is

$$\begin{bmatrix} 76.0392 & 45.73699 \\ 45.73699 & 40.59195 \end{bmatrix}$$

(c) (Please see the appendix for the code)

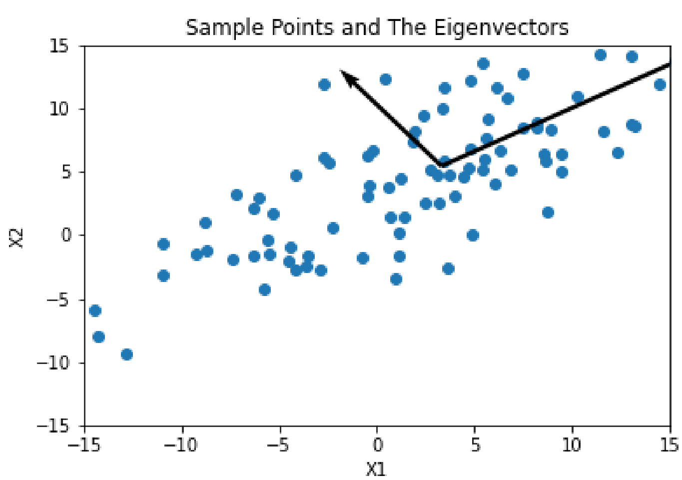
For the eigenvalue of $\lambda_1 = 107.366$, the eigenvector is:

$$\begin{bmatrix} 0.8225 \\ 0.5651 \end{bmatrix}$$

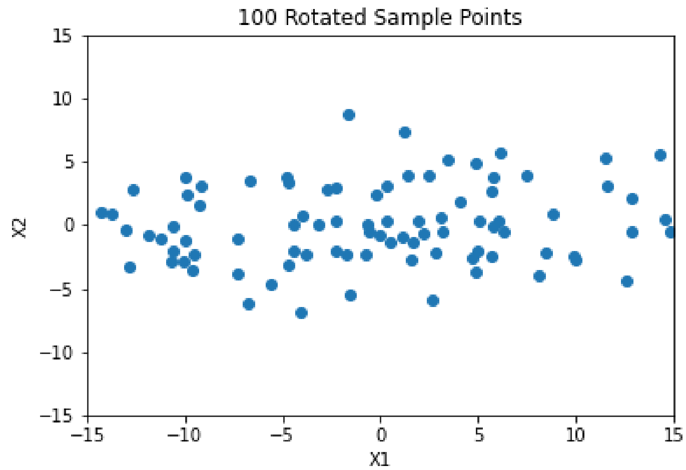
For the eigenvalue of $\lambda_2 = 9.2646$, the eigenvector is:

$$\begin{bmatrix} -0.5651 \\ 0.825 \end{bmatrix}$$

(d) Picture



(e) Picture



5 Classification and Risk

1. Show that the following policy obtains the minimum risk when $\lambda_r \leq \lambda_s$

   (a) Choose class i if $P(Y = i|x) \geq P(Y = j|x)$ for all j and $P(Y = i|x) \geq 1 - \lambda_r/\lambda_s$.

   Let $r : \mathbb{R}^d \to \{1, \ldots, c+1\}$ be a decision rule. And the loss function is defined as follow:

$$L(r(x) = i, y = j) = \begin{cases} 0 & \text{if} \quad i = j \quad i, j \in \{1, \ldots, c\}, \\ \lambda_r & \text{if} \quad i = c+1, \\ \lambda_s & \text{otherwise}, \end{cases}$$

   Then we want to show that, for an arbitrary decision function, f, regardless of the combination of true label, $r(x)$, and $f(x)$, we have $R(r(x)|x) \leq R(f(x)|x)$.

$$R(r(x) = i|x) = \sum_{j=1}^{c} L(r(x) = i, y = j)P(Y = j|x) = \lambda_s \sum_{j \neq i} P(Y = j|x) = \lambda_s(1 - P(Y = i|x))$$

   - For $f(x) \neq c+1$, we have that $R(f(x) \neq c+1|x) = \lambda_s(1 - P(Y \neq c+1|x)$, then we can conclude that

$$\to R(r(x) = i|x) \leq R(f(x) \neq c+1|x)$$

   - For $f(x) = c+1$, we have that $R(f(x) = c+1|x) = \lambda_r$, then we can also conclude that

$$\to R(r(x) = i|x) \leq R(f(x) = c+1|x)$$

     because $P(Y = i|x) \geq 1 - \lambda_r/\lambda_s \to \lambda_r \geq \lambda_s(1 - P(Y = i|x)$ which is resulted in the expression above. because $P(Y = i|x) \geq P(Y \neq c+1|x)$

(b) Choose doubt otherwise.

If $r(x) = c+1$, similar to reasoning above, we have $R(r(x) = c+1|x) = \lambda_r$. Consider an arbitrary $f(x) \neq c+1$, we have

$$R(f(x) \neq c+1|x) = \lambda_s(1 - P(Y \neq c+1|x)$$

And since we choose doubt otherwise, we fail to satisfy the conditions of the previous case, then $P(Y \neq c+1|x) \leq 1 - \lambda_r/\lambda_s$. This lead to the conclusion that

$$R(r(x) = c+1|x) < R(f(x) \neq c+1|x)$$

2.
- What happens if $\lambda_r = 0$? We have $P(Y = i|x) = 1$ or $P(r(x) = i|x) = 1$. So we can choose class i to classify x if we know for sure (probability of 1), otherwise, we can choose doubt.
- What happens if $\lambda_r > \lambda_s$? Then we have $1 - \lambda_r/\lambda_s < 0$, this means we should choose class i (labeled $1, \ldots, c$) to classify x to attain the highest chance for optimal classification.
- Intuitively, when $\lambda_r = 0$, it is better to choose doubt since there are no risks for doing so (with $\lambda_r = 0$); and when $\lambda_r > \lambda_s$, it is consistent with the intuition that it's better to classify x by choosing the class i that provides the best chance for a correct classification, rather than choosing doubt with higher risk.

6 Maximum Likelihood Estimation and Bias

(a) The likelihood function is

$$\mathbf{L}(\mu,\sigma;x_1,x_2,\ldots,x_n)=\prod_{i=1}^{n}\frac{1}{\sqrt{2\pi}\sqrt{\sigma^2/i}}e^{\frac{-1}{2}\frac{(x_i-\mu)^2}{\sigma_i^2}}$$

Then the log-likelihood function is

$$l(\mu,\sigma,\mathbf{X})=-\frac{n}{2}\ln(2\pi)+\frac{1}{2}\sum_{i=1}^{n}\ln i-n\sum_{i=1}^{n}\ln\sigma-\frac{1}{2\sigma^2}\sum_{i=1}^{n}i(x_i-\mu)^2$$

Looking at the partial derivative of the parameters to find the estimator,

$$\begin{cases}\frac{\partial l}{\partial\mu}=\frac{1}{\sigma^2}\sum_{i=1}^{n}i(x_i-\mu)=0\rightarrow\sum_{i=1}^{n}i(x_i-\mu)=0\\[2ex]\frac{\partial l}{\partial\sigma}=-\frac{n}{\sigma}+\frac{1}{\sigma^3}\sum_{i=1}^{n}i(x_i-\mu)^2=0\rightarrow\sigma^2=\frac{1}{n}\sum_{i=1}^{n}i(x_i-\mu)^2\end{cases}$$

$$\begin{cases}\sum_{i=1}^{n}ix_i=\mu\sum_{i=1}^{n}i\rightarrow\sum_{i=1}^{n}ix_i=\mu\frac{n(n+1)}{2}\rightarrow\mu=\frac{2}{n(n+1)}\sum_{i=1}^{n}ix_i\\[2ex]\sigma^2=\frac{1}{n}\sum_{i=1}^{n}i(x_i-\mu)^2\end{cases}$$

Then we have the MLE estimators for μ and σ,

$$\begin{cases}\hat{\mu}=\frac{2}{n(n+1)}\sum_{i=1}^{n}iX_i\\[2ex]\hat{\sigma}^2=\frac{1}{n}\sum_{i=1}^{n}i(X_i-\mu)^2\end{cases}$$

(b) • Statement 1: "The MLE sample estimator $\hat{\mu}$ is unbiased.

$$\mathbf{E}[\hat{\mu}] - \mu = \mathbf{E}\left[\frac{2}{n(n+1)}\sum_{i=1}^{n} iX_i\right] - \mu = \frac{2}{n(n+1)}\sum_{i=1}^{n} i\mathbf{E}[X_i] - \mu$$
$$= \frac{2\mu}{n(n+1)}\sum_{i=1}^{n} i - \mu = \mu - \mu = 0$$

Therefore, the statement is true and that the MLE sample estimator $\hat{\mu}$ is unbiased. Note that we use the linearity of expectation above to bring the expectation inside the summation.

• Statement 2: "The MLE sample estimator $\hat{\sigma}^2$ is unbiased.

$$\mathbf{E}[\hat{\sigma}^2] - \sigma^2 = \mathbf{E}\left[\frac{1}{n}\sum_{i=1}^{n} i(X_i - \mu)^2\right] - \sigma^2 = \frac{1}{n}\sum_{i=1}^{n} i\mathbf{E}\left[(X_i - \mu)^2\right] - \sigma^2$$

Using the fact that $\mathbf{E}[X^2] = \mathbf{Var}[X] + (\mathbf{E}[X])^2$, we have

$$\frac{1}{n}\sum_{i=1}^{n} i\mathbf{E}\left[(X_i - \mu)^2\right] - \sigma^2 = \frac{1}{n}\sum_{i=1}^{n} i\left[\mathbf{Var}[(X_i - \mu)] + (\mathbf{E}[X_i - \mu])^2\right] - \sigma^2$$

$$= \frac{1}{n}\sum_{i=1}^{n} i\left[\mathbf{Var}[X_i] + 0\right] - \sigma^2 = \frac{1}{n}\sum_{i=1}^{n} i\sigma_i^2 - \sigma^2 = \left(\frac{1}{n}\sum_{i=1}^{n} i \cdot \sigma^2/i\right) - \sigma^2 = 0$$

Therefore, the statement is true and that the MLE sample estimator $\hat{\sigma}^2$ is unbiased.

7 Covariance Matrices and Decompositions

(a) $\hat{\Sigma}$ is not invertible if and only if there exists a hyperplane in $d-1$ dimensions such that all the points can lie on that plane. In the context of linear algebra, this is similar to $\hat{\Sigma}$ not having full rank. $\hat{\Sigma}$ is not invertible if and only if all the sample points lie on a common hyperplane in the space that doesn't span all of $\mathbb{R}^{d\times d}$

(b) We can make a new matrix by adding a bit of noise to the diagonal so we can make all the eigenvalues of the new covariance matrix positive in order for it to be invertible (i.e: $\hat{\Sigma} = \hat{\Sigma} + c\mathbf{I}$). To avoid biasing the covariance matrix, we can adjust the constant c to not be too big or too small (certain order of magnitude) than the minimum non-zero eigenvalue of the original covariance matrix. This will avoid blowing up the eigenvalues.

(c) When $\mu = 0$, we have,

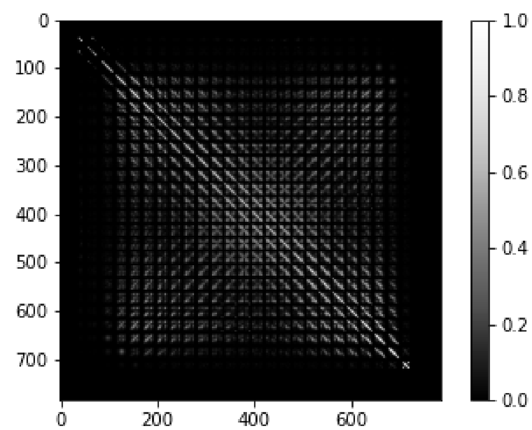
$$f(x) = \left(\frac{1}{(\sqrt{2\pi})^d |\Sigma|}\right) e^{-\frac{x^\mathsf{T}\Sigma^{-1}x}{2}}$$

So if we want to maximize $f(x)$, we want to maximize $x^\mathsf{T}\Sigma^{-1}x$. This can be done by finding the largest eigenvalue of Σ^{-1}, and having x equal to the corresponding eigenvector and normalizing it to have the length one. Similarly, we can minimize $f(x)$ by letting x equal to the normalized eigenvector with the smallest eigenvalue of Σ^{-1}

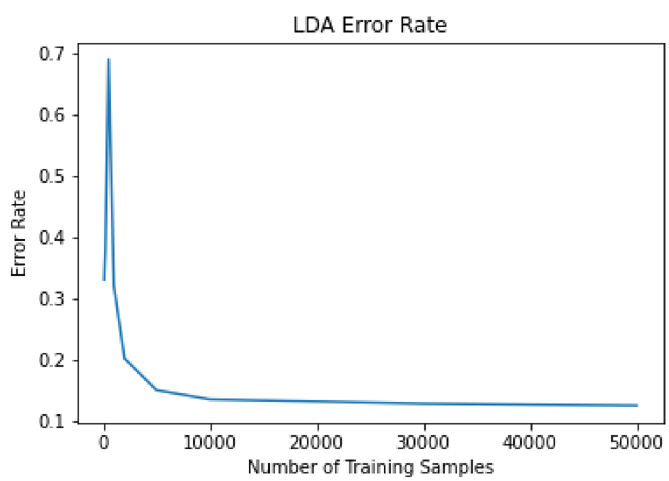
8 Gaussian Classifiers for Digits and Spam

1. Please see the apendix for the code

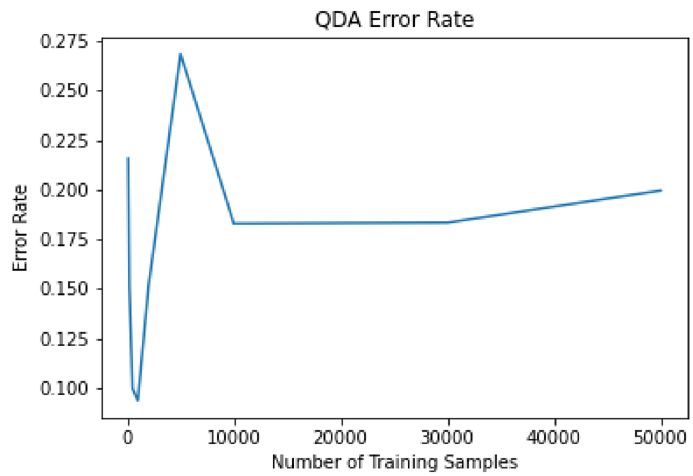
2. Below is the visualization for the covariance matrix of digit 0. We can see the the diagonal terms have a higher covariance in comparison to the off-diagonal terms. This indicates that the the covariance between each sample points and itself, as well as each sample point and adjacent points, is higher than the covariance between a sample point and a point farther away.



3. (a) Picture

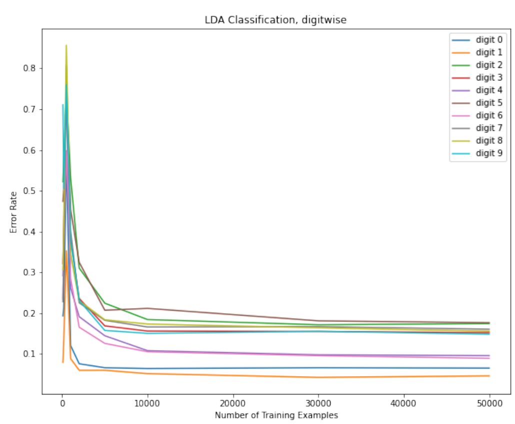


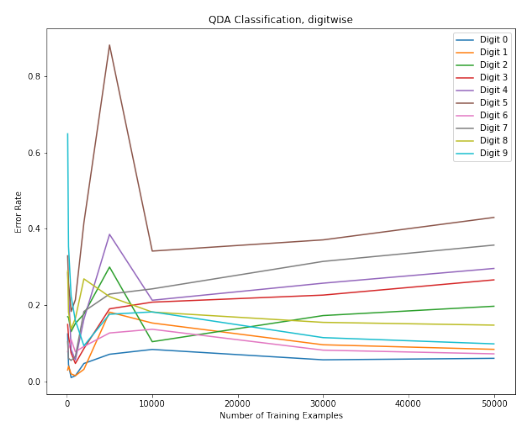
(b) Picture



(c) LDA performed better than QDA because QDA is prone to overfitting, especially when there are a larger number of free variables

(d) From the plot, we can see that the it is the best for LDA to classify digit 1 and digit 0 is the best for QDA based on the validation error.





4. **Kaggle Name:** Hieu Tang Nguyen BA
   Kaggle Link: `https://www.kaggle.com/hieutangnguyenba`
   MNIST Accuracy: 0.95800

5. **Kaggle Name:** Hieu Tang Nguyen BA
   Kaggle Link: `https://www.kaggle.com/hieutangnguyenba`
   Spam Accuracy: 0.80640

9 Code Appendix

9.1 Constants, packages, etc.

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

# ... etc. This is an environment where you can enter code.
# You could also just include screenshots using
# \includegraphics[options]{image name}
```

9.2 Isocontours of Normal Distributions

```
import numpy as np
from scipy import stats
import matplotlib.pyplot as plt

# 3.1

x = np.linspace(-4, 4, 1000)
y = np.linspace(-4, 4, 1000)

X,Y = np.meshgrid(x, y)

pos = np.array([Y, X]).T

z = scipy.stats.multivariate_normal([1, 1], [[1, 0], [0, 2]])

Z = z.pdf(pos)

plt.figure()
contour = plt.contour(X, Y, Z)
plt.clabel(contour, inline=1, fontsize=10)
plt.title('Isocontour for part 3.1')
plt.colorbar()
plt.savefig("3_1.png")
plt.show()

# 3.2

x = np.linspace(-4, 4, 1000)
y = np.linspace(-4, 4, 1000)

X,Y = np.meshgrid(x, y)

pos = np.array([Y, X]).T

z = scipy.stats.multivariate_normal([-1, 2], [[2, 1], [1, 4]])

Z = z.pdf(pos)

plt.figure()
contour = plt.contour(X, Y, Z)
plt.clabel(contour, inline=1, fontsize=10)
plt.title('Isocontour for part 3_2')
plt.colorbar()
plt.savefig("3_2.png")
plt.show()

# 3.3

x = np.linspace(-2, 4, 1000)
y = np.linspace(-2, 4, 1000)

```

```
X,Y = np.meshgrid(x, y)

pos = np.array([Y, X]).T

z1 = scipy.stats.multivariate_normal([0, 2], [[2, 1], [1, 1]])
z2 = scipy.stats.multivariate_normal([2, 0], [[2, 1], [1, 1]])


Z = z1.pdf(pos) - z2.pdf(pos)


plt.figure()
contour = plt.contour(X, Y, Z)
plt.clabel(contour, inline=1, fontsize=10)
plt.title('Isocontour for part 3.3')
plt.colorbar()
plt.savefig("3_3.png")
plt.show()

# 3.4

x = np.linspace(-2, 4, 1000)
y = np.linspace(-2, 4, 1000)

X,Y = np.meshgrid(x, y)

pos = np.array([Y, X]).T

z1 = scipy.stats.multivariate_normal([0, 2], [[2, 1], [1, 1]])
z2 = scipy.stats.multivariate_normal([2, 0], [[2, 1], [1, 4]])
Z = z1.pdf(pos) - z2.pdf(pos)

plt.figure()
contour = plt.contour(X, Y, Z)
plt.clabel(contour, inline=1, fontsize=10)
plt.title('Isocontour for part 3.4')
plt.colorbar()
plt.savefig("3_4.png")
plt.show()

# 3.5

x = np.linspace(-3, 3, 1000)
y = np.linspace(-3, 3, 1000)

X,Y = np.meshgrid(x, y)
pos = np.array([Y, X]).T

z1 = scipy.stats.multivariate_normal([1, 1], [[2, 0], [0, 1]])
z2 = scipy.stats.multivariate_normal([-1, -1], [[2, 1], [1, 2]])

Z = z1.pdf(pos) - z2.pdf(pos)

plt.figure()
contour = plt.contour(X, Y, Z)
plt.clabel(contour, inline=1, fontsize=10)
plt.title('Isocontour for part 3.5')
plt.colorbar()
plt.savefig("3_5.png")
plt.show()
```

9.3 Eigenvectors of the Gaussian Covariance Matrix

```

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

```python
np.random.seed(26)

size = 100
mu_x = 3
var_x = 9
mu_y = 4
var_y = 4

x_sample = np.random.normal(mu_x, var_x, size)
y_sample = np.random.normal(mu_y, var_y, size)

sample_points = np.array([np.array((x, 0.5 * x + y)) for (x, y) in zip(x_sample, y_sample)])

# (a): compute the mean of the sample

sample_mean = np.mean(sample_points, axis=0)

print('Mean of the Sample:')

print(sample_mean)

# (b): compute the 2x2 covariance matrix of the sample

sample_covariance = np.cov(sample_points.T)

print('2x2 Covariance Matrix of the Sample')

print(sample_covariance)

# (c) compute the eigenvectors and eigenvalues of the covariance matrix

eigenvalues, eigenvectors = np.linalg.eig(sample_covariance)

print('Eigenvalues of the Covariance Matrix:')

print(eigenvalues)

print("-------------------")

print('Eigenvectors of the Covariance Matrix:')

print(eigenvectors)

# (d): plot 100 data points and eigenvectors

plt.figure()

plt.title("Sample Points and The Eigenvectors")
plt.xlabel("X1")
plt.ylabel("X2")
plt.xlim(-15, 15)
plt.ylim(-15, 15)


plt.scatter(sample_points[:, 0], sample_points[:, 1])

x_vector = [sample_mean[0], sample_mean[0]]
y_vector = [sample_mean[1], sample_mean[1]]
v_1_vector = [eigenvectors[0][0] * eigenvalues[0], eigenvectors[0][1] * eigenvalues[1]]
v_2_vector = [eigenvectors[1][0] * eigenvalues[0], eigenvectors[1][1] * eigenvalues[1]]

plt.quiver(x_vector, y_vector, v_1_vector, v_2_vector, angles="xy", scale_units="xy", scale=1)


plt.savefig("4d.png")
plt.show()
```

```python
# (e): plot rotated points

rotated_points = np.dot(eigen_vectors.T, (sample_points - sample_mean).T).T

plt.figure()

plt.title("100 Rotated Sample Points")
plt.xlabel("X1")
plt.ylabel("X2")
plt.xlim(-15, 15)
plt.ylim(-15, 15)

plt.scatter(rotated_points[:, 0], rotated_points[:, 1])


plt.savefig("4e.png")
plt.show()
```

9.4 Gaussian Classifiers for Digits and Spam

```python


import numpy as np
import scipy.cluster
import scipy.ndimage
import matplotlib
import matplotlib.pyplot as plt
import os
from scipy import io
import pandas as pd
import pprint
from scipy.stats import multivariate_normal


# Usage results_to_csv(clf.predict(X_test))
def results_to_csv(y_test):
    y_test = y_test.astype(int)
    df = pd.DataFrame({'Category': y_test})
    df.index += 1  # Ensures that the index starts at 1.
    df.to_csv('submission.csv', index_label='Id')

for data_name in ["mnist", "spam"]:
    data = io.loadmat("data/%s_data.mat" % data_name)
    print("\nloaded %s data!" % data_name)
    fields = "test_data", "training_data", "training_labels"
    for field in fields:
        print(field, data[field].shape)

# Load all data for MNIST
mnist = io.loadmat("data/mnist_data.mat")
mnist_X, mnist_y = mnist['training_data'].astype(float), mnist['training_labels']
mnist_test_X = mnist['test_data'].astype(float)

# Load all data for SPAM
spam = io.loadmat("data/spam_data.mat")
spam_X, spam_y = spam['training_data'].astype(float), spam['training_labels']
spam_test_X = spam['test_data'].astype(float)


#### 8.1

# Contrast normalization to the training set.
cn_mnist_X = []

for i in range(len(mnist_X)):

```

```
    val = mnist_X[i]/(np.linalg.norm(mnist_X[i])+1e-15)
    cn_mnist_X.append(val)

cn_mnist_X = np.array(cn_mnist_X)
cn_mnist_X.shape

# Contrast normalization to the testing set.
cn_mnist_test_X = []
for i in range(len(mnist_test_X)):
    val = mnist_test_X[i]/(np.linalg.norm(mnist_test_X[i])+1e-15)
    cn_mnist_test_X.append(val)

cn_mnist_test_X = np.array(cn_mnist_test_X)
cn_mnist_test_X.shape

mnist_fitted = {}
for i in np.unique(mnist_y):
    indices = (mnist_y == i).flatten()
    data = cn_mnist_X[indices]
    mean = np.mean(data, axis=0)
    cov = np.cov(data, rowvar = False)
    mnist_fitted[i] = (mean, cov)


### 8.2

indices = (mnist_y == np.unique(mnist_y)[0]).flatten()
data = cn_mnist_X[indices]
ncov = np.corrcoef(data, rowvar=False)
ncov[np.isnan(ncov)] = 0
ncov = np.abs(ncov)
plt.imshow(ncov, cmap=matplotlib.cm.Greys_r)
plt.colorbar()
#plt.savefig("8_2.png")
plt.show()


### 8.3

def split_train_val_data(data, labels, val_size):
    num_items = len(data)
    assert num_items == len(labels)
    assert val_size >= 0
    if val_size < 1.0:
        val_size = int(num_items * val_size)
    train_size = num_items - val_size
    idx = np.random.permutation(num_items)
    data_train = data[idx][:train_size]
    label_train = labels[idx][:train_size]
    data_val = data[idx][train_size:]
    label_val = labels[idx][train_size:]
    return data_train, data_val, label_train, label_val

class LDA:
    def __init__(self):
        self.mu = None
        self.sigma = None
        self.Pi = None

    def fit(self, X, y):

        N, labels = len(y), np.unique(y)
        d = X.shape[1]
        C = len(labels)
        mu = np.zeros((C, d))
        Pi = np.zeros(C)
        sigma = np.zeros((d, d))
        for i, label in enumerate(labels):
```

```
            X_i = []
            for index, result in enumerate(y):
                if result == label:
                    X_i.append(X[index])
            X_i = np.array(X_i)
            #print(len(X_i))
            Pi[i] = len(X_i) / N
            mu[i, :] = np.mean(X_i, axis=0)
            sigma += np.cov(X_i.T)*X_i.shape[0]
        sigma /= N
        self.mu = mu
        self.sigma = sigma
        self.Pi = Pi

    def predict(self, X):

        d = len(self.sigma)
        self.sigma += 1e-15*np.array(np.eye(d))

        L1 = self.mu.dot(np.linalg.solve(self.sigma, X.T)).T

        L2 = 1/2*np.diag(self.mu.dot\
                         (np.linalg.solve(self.sigma, self.mu.T)))

        L = L1 - L2 - np.log(self.Pi)

        pred = np.argmax(L.T, axis=0)


        return pred

    def accuracy(self, X, y):
        N = len(y)
        correct = 0
        pred = self.predict(X)
        for i in range(N):
            if pred[i] == y[i]:
                correct += 1
        return correct/N, pred

mnist_train_X, mnist_val_X, mnist_train_y, mnist_val_y = split_train_val_data(cn_mnist_X, mnist_y
    ↪ , 10000)


training_size = np.array([100, 200, 500, 1000, 2000, 5000, 10000, 30000, 50000])
lda_accuracy = np.zeros(len(training_size))

for i, N in enumerate(training_size):
    lda = LDA()
    lda.fit(mnist_train_X[:N,:], mnist_train_y[:N])
    lda_accuracy[i], _ = lda.accuracy(mnist_val_X, mnist_val_y)

lda_accuracy

plt.figure()
plt.plot(training_size, (1-lda_accuracy))
plt.title('LDA Error Rate')
plt.xlabel('Number of Training Samples')
plt.ylabel('Error Rate')
#plt.savefig("8_3_a.png")
plt.show()

class QDA:
    def __init__(self, a):
        self.mu = None
        self.sigma = None
        self.Pi = None
        self.a = a
```

```python

    def fit(self, X, y):
        N, labels = len(y), np.unique(y)
        #print(labels)
        d = X.shape[1]
        C = len(labels)
        sigma = [np.zeros([d, d]) for i in range(N)]
        mu = np.zeros([C, d])
        Pi = np.zeros([C])
        for i, label in enumerate(labels):
            # Get each classes' statistics
            X_i = []
            for index, result in enumerate(y):
                if result == label:
                    X_i.append(X[index])
            X_i = np.array(X_i)
            Pi[i] = len(X_i) / N
            mu[i, :] = np.mean(X_i, axis=0)
            sigma[i] = np.cov(X_i.T) + self.a*np.eye(d)
        self.mu = mu
        self.sigma = sigma
        self.Pi = Pi


    def predict(self, X):
        C = len(self.Pi)
        L = np.zeros((C, len(X)))
        mu = self.mu
        sigma = self.sigma
        Pi = self.Pi
        for i in range(C):
            #print('start')
            L[i,:] = multivariate_normal.logpdf(X, mean=mu[i], cov=sigma[i])
            L[i,:] += np.log(Pi[i])
        pred = np.argmax(L, axis=0)
        return pred

    def accuracy(self, X, y):
        N = len(y)
        correct = 0
        pred = self.predict(X)
        for i in range(N):
            if pred[i] == y[i]:
                correct += 1
        #print('accuracy', correct/N)
        return correct/N, pred

training_size = [100, 200, 500, 1000, 2000, 5000, 10000, 30000, 50000]

qda_accuracy = np.zeros(len(training_size))


for i, N in enumerate(training_size):
    qda = QDA(1e-8)
    qda.fit(mnist_train_X[:N,:], mnist_train_y[:N])
    qda_accuracy[i], _ = qda.accuracy(mnist_val_X, mnist_val_y)

qda_accuracy

plt.figure()
plt.plot(training_size, (1-qda_accuracy))
plt.title('QDA Error Rate')
plt.xlabel('Number of Training Samples')
plt.ylabel('Error Rate')
#plt.savefig("8_3_b.png")
plt.show()


```

```python
mnist_lda = LDA()
mnist_lda.fit(mnist_train_X, mnist_train_y)
train_accuracy, train_prev = mnist_lda.accuracy(mnist_train_X, mnist_train_y)
val_accuracy, val_prev = mnist_lda.accuracy(mnist_val_X, mnist_val_y)
print(train_accuracy, val_accuracy)

q_mnist_qda = QDA(1e-8)
q_mnist_qda.fit(mnist_train_X, mnist_train_y)
train_accuracy, train_prev = q_mnist_qda.accuracy(mnist_train_X, mnist_train_y)
val_accuracy, val_prev = q_mnist_qda.accuracy(mnist_val_X, mnist_val_y)
print(train_accuracy, val_accuracy)


# Training for LDA.
training_size = np.array([100, 200, 500, 1000, 2000, 5000, 10000, 30000, 50000])
val_acc_list = np.zeros(len(training_size))
val_pred_list = []

for i, N in enumerate(training_size):
    clf = LDA()
    clf.fit(mnist_train_X[:N], mnist_train_y[:N])
    val_acc_list[i], val_pred = clf.accuracy(mnist_val_X, mnist_val_y)
    val_pred_list.append(val_pred)

print(val_acc_list)
print(len(val_pred_list),len(val_pred_list[0]))

# Training for QDA
training_size = np.array([100, 200, 500, 1000, 2000, 5000, 10000, 30000, 50000])
q_val_acc_list = np.zeros(len(training_size))
q_val_pred_list = []

for i, N in enumerate(training_size):
    clf = QDA(1e-8)
    clf.fit(mnist_train_X[:N], mnist_train_y[:N])
    q_val_acc_list[i], val_pred = clf.accuracy(mnist_val_X, mnist_val_y)
    q_val_pred_list.append(val_pred)

print(q_val_acc_list)
print(len(q_val_pred_list),len(q_val_pred_list[0]))

def error_evaluate(prediction, labels, N=10):
    digit_errors = np.zeros((N))
    prediction = prediction.reshape(len(prediction),1)
    for i in range(N):
        same = (labels == i)
        total_digit = sum(same)
        #print(total_digit)
        correct_digit = sum(prediction[same] == labels[same])
        error_rate = 1 - (correct_digit/total_digit)
        digit_errors[i] = error_rate
    return digit_errors


lda_0 = error_evaluate(val_pred_list[0], np.array(mnist_val_y)).reshape(10,1)
lda_1 = error_evaluate(val_pred_list[1], np.array(mnist_val_y)).reshape(10,1)
lda_2 = error_evaluate(val_pred_list[2], np.array(mnist_val_y)).reshape(10,1)
lda_3 = error_evaluate(val_pred_list[3], np.array(mnist_val_y)).reshape(10,1)
lda_4 = error_evaluate(val_pred_list[4], np.array(mnist_val_y)).reshape(10,1)
lda_5 = error_evaluate(val_pred_list[5], np.array(mnist_val_y)).reshape(10,1)
lda_6 = error_evaluate(val_pred_list[6], np.array(mnist_val_y)).reshape(10,1)
lda_7 = error_evaluate(val_pred_list[7], np.array(mnist_val_y)).reshape(10,1)
lda_8 = error_evaluate(val_pred_list[8], np.array(mnist_val_y)).reshape(10,1)

qda_0 = error_evaluate(q_val_pred_list[0], np.array(mnist_val_y)).reshape(10,1)
qda_1 = error_evaluate(q_val_pred_list[1], np.array(mnist_val_y)).reshape(10,1)
qda_2 = error_evaluate(q_val_pred_list[2], np.array(mnist_val_y)).reshape(10,1)
qda_3 = error_evaluate(q_val_pred_list[3], np.array(mnist_val_y)).reshape(10,1)
```

```
qda_4 = error_evaluate(q_val_pred_list[4], np.array(mnist_val_y)).reshape(10,1)
qda_5 = error_evaluate(q_val_pred_list[5], np.array(mnist_val_y)).reshape(10,1)
qda_6 = error_evaluate(q_val_pred_list[6], np.array(mnist_val_y)).reshape(10,1)
qda_7 = error_evaluate(q_val_pred_list[7], np.array(mnist_val_y)).reshape(10,1)
qda_8 = error_evaluate(q_val_pred_list[8], np.array(mnist_val_y)).reshape(10,1)

to_plot = np.concatenate((lda_0,lda_1,lda_2,lda_3,lda_4,
                          lda_5,lda_6,lda_7,lda_8), axis=1)
plt.figure(figsize=(10,8))
for i in range(10):
    plt.plot(training_size, to_plot[i], label='digit '+ str(i))
    plt.legend()
    plt.title('LDA Classification, digitwise')
    plt.xlabel('Number of Training Examples')
    plt.ylabel('Error Rate')

#plt.savefig("8_3_d_lda.png")

to_plot = np.concatenate((qda_0,qda_1,qda_2,qda_3,qda_4,
                          qda_5,qda_6,qda_7,qda_8), axis=1)

plt.figure(figsize=(10,8))
for i in range(10):

    plt.plot(training_size, to_plot[i], label='Digit '+ str(i))
    plt.legend()
    plt.title('QDA Classification, digitwise')
    plt.xlabel('Number of Training Examples')
    plt.ylabel('Error Rate')

plt.savefig("8_3_d_qda.png")

#### 8.4

#Calculate the HOG feature

from skimage.feature import hog
def hog_dataset(dataset):
    list_hog_fd = []
    for sample in dataset:
        #print(sample.shape)
        fd = hog(sample.reshape((28, 28)), orientations=8,
                 pixels_per_cell=(4, 4), cells_per_block=(1, 1),
                 visualize=False)
        list_hog_fd.append(fd)
    return np.array(list_hog_fd, 'float64')

hog_train_X = hog_dataset(cn_mnist_X)
#hog_val_X = hog_dataset(mnist_val_X)
hog_test_X = hog_dataset(cn_mnist_test_X)

std_train_X= np.array([np.std(cn_mnist_X, axis=1)]).T
std_test_X = np.array([np.std(cn_mnist_test_X, axis=1)]).T


final_training = np.concatenate((cn_mnist_X, hog_train_X, std_train_X), axis=1)

final_test = np.concatenate((cn_mnist_test_X, hog_test_X, std_test_X), axis=1)


from sklearn.model_selection import GridSearchCV
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis as QDA

qda = QDA()
param_grid_1 = {"reg_param" :[0.1, 0.5, 1, 2, 5, 10, 15]}

gs = GridSearchCV(estimator=qda, param_grid=param_grid_1, scoring='accuracy', cv=3, n_jobs=-1)

```

```python
gs = gs.fit(final_training, mnist_y)
print(gs.best_score_)
print(gs.best_params_)


param_grid_2 = {"reg_param" :[0.01, 0.05, 0.1]}

gs = GridSearchCV(estimator=qda, param_grid=param_grid_2, scoring='accuracy', cv=3, n_jobs=-1)

gs = gs.fit(final_training, mnist_y)
print(gs.best_score_)
print(gs.best_params_)

param_grid_3 = {"reg_param" :[0.001, 0.01]}

gs = GridSearchCV(estimator=qda, param_grid=param_grid_3, scoring='accuracy', cv=3, n_jobs=-1)

gs = gs.fit(final_training, mnist_y)
print(gs.best_score_)
print(gs.best_params_)


#### 8.5

spam = io.loadmat("data/spam_data.mat")
spam_X, spam_y = spam['training_data'].astype(float), spam['training_labels']
spam_test_x = spam['test_data'].astype(float)


spam_train_x, spam_val_x, spam_train_y, spam_val_y = split_train_val_data(spam_X, spam_y, 500)



# Computes prior, mean, and covariance
def spam_train_lda(train_x, train_y):
    n,d = train_x.shape
    priors = np.zeros((2,1))
    covariance = np.zeros((d,d)).astype(np.float32)
    mean_samples = np.zeros((2, d))
    train_y = train_y.reshape(-1,)

    for spam in [0,1]:
        ## Compute priors
        prob = (train_y == spam).astype(np.int32).sum() / n
        priors[int(spam)] = prob

        data = train_x[train_y == spam, :]
        mean_samples[int(spam),:] = data.mean(axis=0)
        covariance += np.cov(data.T)

    covariance /= 2
    return priors, mean_samples, covariance

# Train
lda_priors, lda_means, lda_covariance = spam_train_lda(spam_train_x, spam_train_y)

N_train = spam_train_x.shape[0]
out_train = np.zeros((N_train, 2))

N_val = spam_val_x.shape[0]
out_val = np.zeros((N_val, 2))

for class_y in [0,1]:
    prior = lda_priors[class_y]
    mean = lda_means[class_y, :].reshape(-1, 1)

    w = np.linalg.solve(lda_covariance.T, mean)
    alpha = -0.5 * w.T.dot(mean) + np.log(prior)
```

```
    out_train[:, class_y] = (spam_train_x.dot(w) + alpha).reshape(-1,)
    out_val[:, class_y] = (spam_val_x.dot(w) + alpha).reshape(-1,)

train_pred = np.argmax(out_train, axis=1).reshape(-1, 1)
val_pred = np.argmax(out_val, axis=1).reshape(-1, 1)

def computeDiagonals(X, X_T):
    N,D = X.shape
    out_diag = np.zeros((N,1))

    for n in range(N):
        i = X[n, :].reshape(1, -1)
        i_t = X_T[:, n].reshape(-1, 1)
        out_diag[n] = i.dot(i_t)

    return out_diag

def spam_train_qda(train_x, train_y):
    n,d = train_x.shape
    priors = np.zeros((2,1))
    covariances = np.zeros((2,d,d)).astype(np.float32)
    mean_samples = np.zeros((2, d))
    train_y = train_y.reshape(-1,)

    for spam in [0,1]:
        ## Compute priors
        prob = (train_y == spam).astype(np.int32).sum() / n
        priors[int(spam)] = prob

        data = train_x[train_y == spam, :]

        mean_samples[int(spam),:] = data.mean(axis=0)
        covariances[int(spam), :, :] = np.cov(data.T)

    return priors, mean_samples, covariances

# Train
qda_priors, qda_means, qda_covariances = spam_train_qda(spam_train_x, spam_train_y)

N_train = spam_train_x.shape[0]
out_train = np.zeros((N_train, 2))

N_val = spam_val_x.shape[0]
out_val = np.zeros((N_val, 2))

for spam in [0,1]:
    prior = qda_priors[spam]
    mean = qda_means[spam, :]
    covariance = qda_covariances[spam, :, :]

    u, sig, v = np.linalg.svd(covariance)
    covariance = covariance + (sig.min()*(10**-1)) * np.eye(*covariance.shape)

    alpha = -0.5 * np.linalg.det(covariance) + prior

    mean_centered_train = spam_train_x - mean
    weight_train = np.linalg.solve(covariance, mean_centered_train.T)
    prediction_train = -0.5*computeDiagonals(mean_centered_train, weight_train) + alpha
    out_train[:, spam] = prediction_train.reshape(N_train,)

    mean_centered_val = spam_val_x - mean
    weight_val = np.linalg.solve(covariance, mean_centered_val.T)
    prediction_val = -0.5*computeDiagonals(mean_centered_val, weight_val) + alpha
    out_val[:, spam] = prediction_val.reshape(N_val,)


train_pred = np.argmax(out_train, axis=1).reshape(-1, 1)
```

```
val_pred = np.argmax(out_val, axis=1).reshape(-1, 1)


from save_csv import results_to_csv
N_test = spam_test_x.shape[0]
out_test = np.zeros((N_test, 2))

for class_y in [0,1]:
    prior = lda_priors[class_y]
    mean = lda_means[class_y, :].reshape(-1, 1)

    w = np.linalg.solve(lda_covariance.T, mean)
    alpha = -0.5 * w.T.dot(mean) + np.log(prior)

    out_test[:, class_y] = (spam_test_x.dot(w) + alpha).reshape(-1,)

test_pred = np.argmax(out_test, axis=1).reshape(-1,)
results_to_csv(test_pred)
```