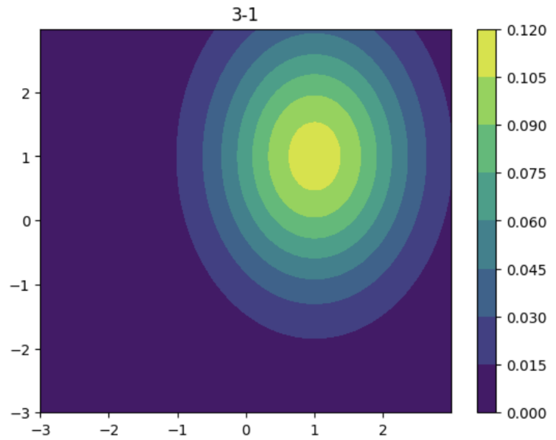
Hiva Mohammadzadeh
3036919598

Question 1: Honor Code

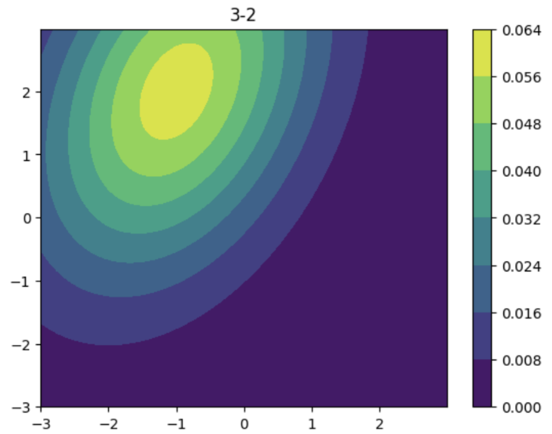
> *"I certify that all solutions are entirely in my own words and that I have not looked at another student's solutions. I have given credit to all external sources I consulted."*

Question 3: Isocontours of Normal Distributions

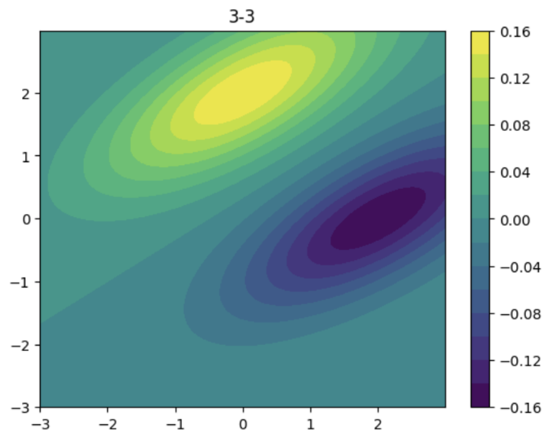
Part 1:



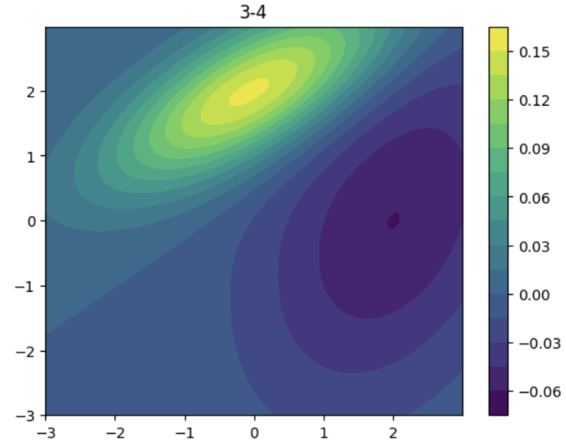
Part 2:



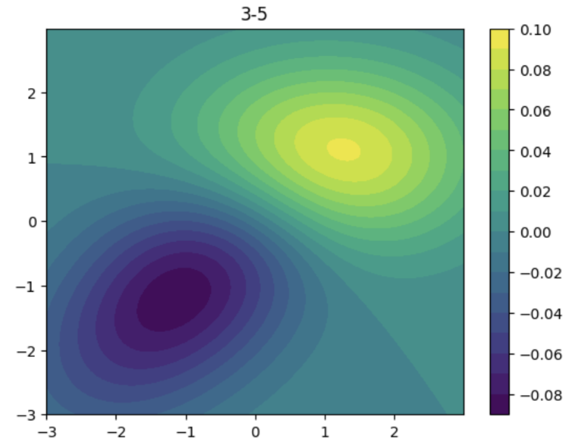
Part 3:



Part 4:


3-4
0.15
0.12
0.09
0.06
0.03
0.00
−0.03
−0.06
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2

Part 5:


3-5
0.10
0.08
0.06
0.04
0.02
0.00
−0.02
−0.04
−0.06
−0.08
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2

Question 4: Eigenvectors of the Gaussian Covariance Matrix

Part 1:

```
Mean: [3.18174856 5.66142143]
```

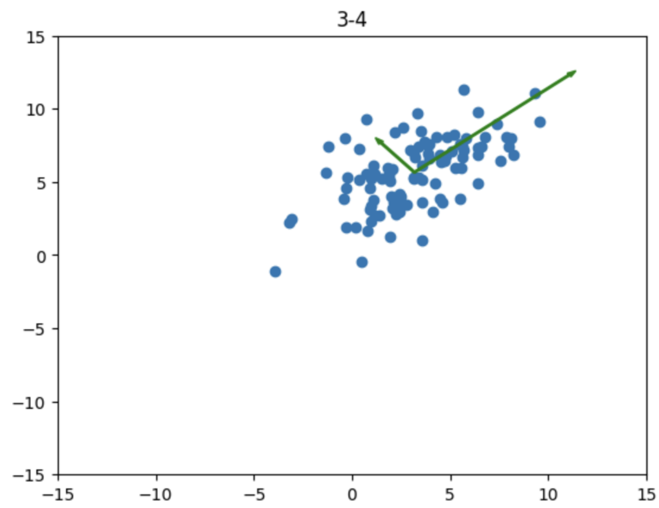
Part 2:

```
Covariance matrix:
 [[7.12274112 3.78326186]
 [3.78326186 5.82706493]]
```

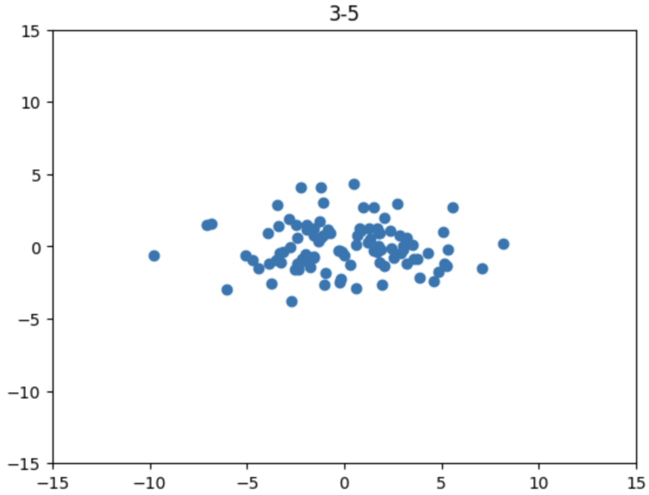
Part 3:

```
Eigenvectors:
 [[ 0.76445448 -0.64467771]
 [ 0.64467771  0.76445448]]
Eigenvalues:
 [10.31323137  2.63657468]
```

Part 4:



Part 5:

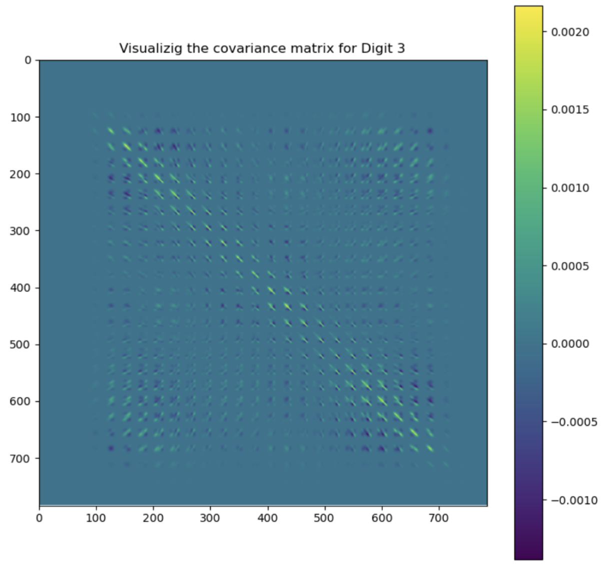

3-5
15
10
5
0
−5
−10
−15
−15
−10
−5
0
5
10
15

Question 8: Gaussian Classifiers for Digits and Spam

Part 1:

```
Fitting Gaussian Distribution:
Mean: dict_keys([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
Length of mean of digit 5: 784
Covariance: dict_keys([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
Length of covariance of digit 7: 784
```

Part 2:

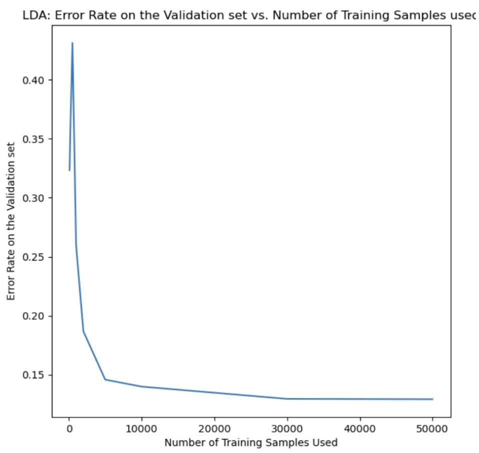


How do the diagonal terms compare to off-diagonal terms?

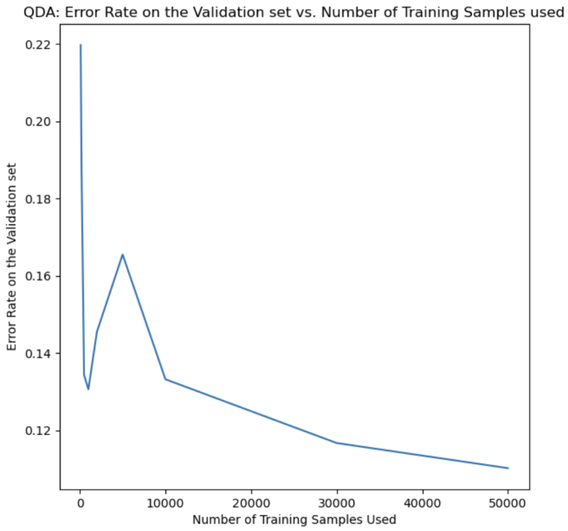
The covariances on the diagonal are brighter than the covariances that are not on the diagonal which means that they have a higher covariance value on the diagonal. The terms on the diagonal should be a higher value since the values on the diagonal are where the covariance is 1 (since it's the covariance of the same sample with itself). Everywhere else is less than or equal to 1 because it's the covariance between 2 different samples.

Part 3:

a) LDA:



b) QDA:



c) Which was better?

LDA performed better because generally LDA does better on larger datasets. Also, looking at their error rate graphs we can see that the validation accuracy for QDA decreases as our number of training points increases but LDA's validation increases as we increase the number of training points. The error graph also shows that LDA has a very smooth decreasing error rate but QDA does not and has some bumps before it starts to also decrease. ALso, QDA on larger datasets can cause overfitting.

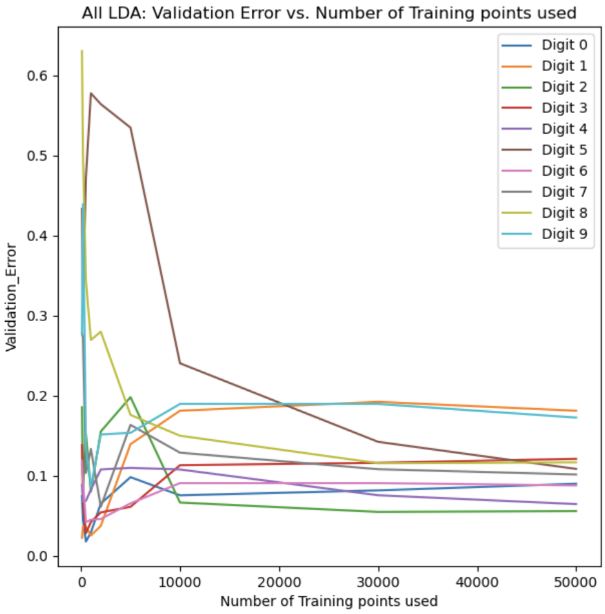
LDA:

```
LDA:
Training with 100 points
Validation accuracy: 0.6767

Training with 200 points
Validation accuracy: 0.6451

Training with 500 points
Validation accuracy: 0.5688

Training with 1000 points
Validation accuracy: 0.7404

Training with 2000 points
Validation accuracy: 0.8132

Training with 5000 points
Validation accuracy: 0.8541

Training with 10000 points
Validation accuracy: 0.86

Training with 30000 points
Validation accuracy: 0.8704

Training with 50000 points
Validation accuracy: 0.8707
```

QDA:

```
QDA:
Training with 100 points
Validation acc: 0.7803

Training with 200 points
Validation acc: 0.812

Training with 500 points
Validation acc: 0.8657

Training with 1000 points
Validation acc: 0.8694

Training with 2000 points
Validation acc: 0.8545

Training with 5000 points
Validation acc: 0.8345

Training with 10000 points
Validation acc: 0.8668

Training with 30000 points
Validation acc: 0.8833

Training with 50000 points
Validation acc: 0.8898
```

d) Plot all 10 curves:



Which digit is easiest to classify?

Digit 0 and 1 are the easiest digits to classify. I found these to be the easiest to classify because they have the most unique features as other digits. Looking at their LDA Loss graph as well, they have the smallest minimum value on the all digits LDA graph.

Part 4: Kaggle Submission

Kaggle username: **Hiva Mohammadzadeh**
Kaggle Scores:

a) MNIST: 0.87933 = 88%

Part 5: LDA and QDA on SPAM

Kaggle username: **Hiva Mohammadzadeh**
Kaggle Scores:

a) SPAM: 0.784 = 78%

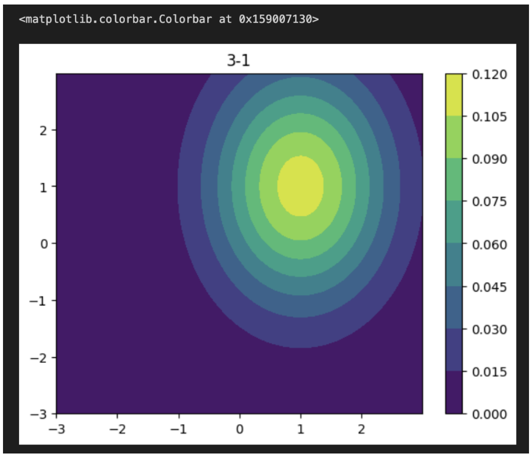
CODE APPENDIX:

Question 3: Isocontours of Normal Distributions

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

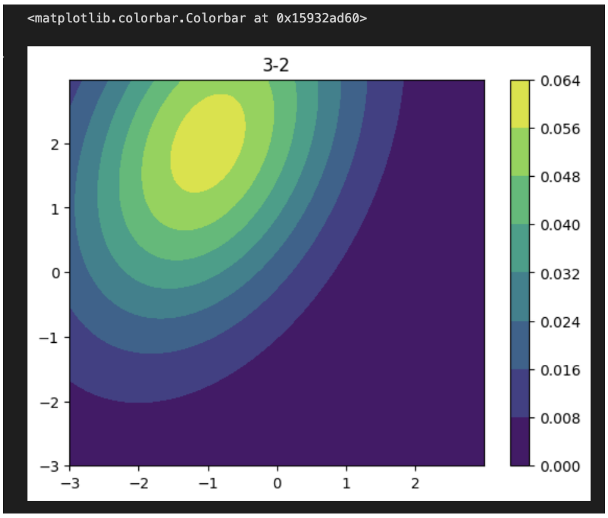
Part 1:

```
np.random.seed(0)
f = multivariate_normal([1,1], [[1,0],[0,2]])
x_domain, y_domain = np.mgrid[-3:3:.01, -3:3:.01]
axises = np.dstack((x_domain, y_domain))
plt.figure(0)
plt.title("3-1")
plt.contourf(x_domain, y_domain, f.pdf(axises))
plt.colorbar()
```



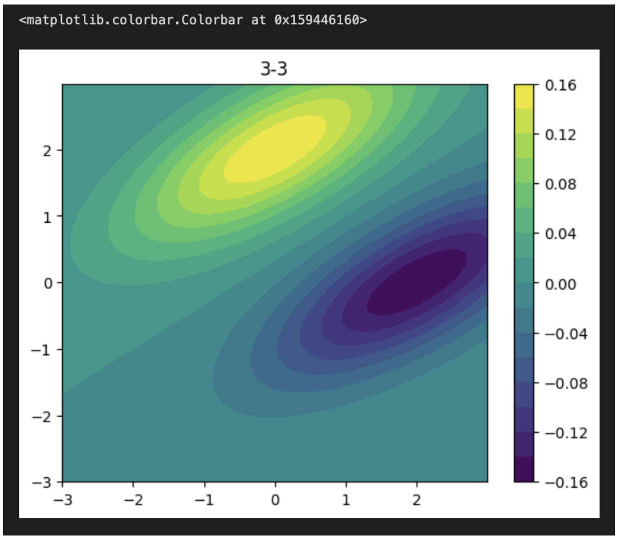
Part 2:

```
f = multivariate_normal([-1,2], [[2,1],[1,4]])
x_domain, y_domain = np.mgrid[-3:3:.01, -3:3:.01]
axises = np.dstack((x_domain, y_domain))
plt.figure(1)
plt.title("3-2")
plt.contourf(x_domain, y_domain, f.pdf(axises))
plt.colorbar()
```



Part 3:

```
f1 = multivariate_normal([0,2], [[2,1],[1,1]])
f2 = multivariate_normal([2,0], [[2,1],[1,1]])
x_domain, y_domain = np.mgrid[-3:3:.01, -3:3:.01]
axises = np.dstack((x_domain, y_domain))
plt.figure(2)
plt.title("3-3")
plt.contourf(x_domain, y_domain, f1.pdf(axises) - f2.pdf(axises), 20)
plt.colorbar()
```

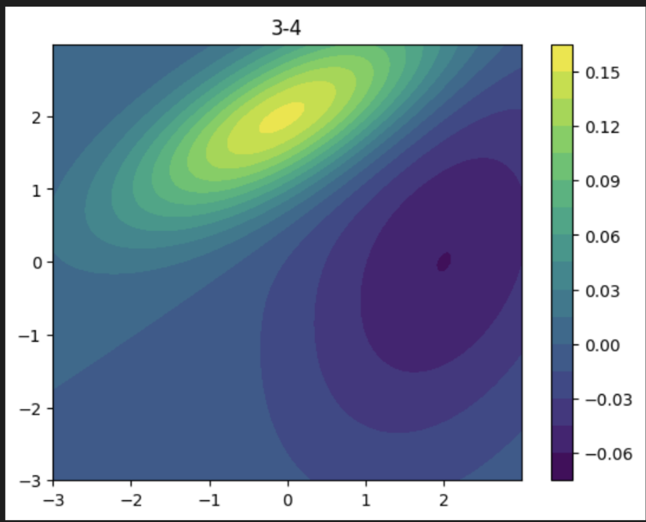


Part 4:

```
f1 = multivariate_normal([0,2], [[2,1],[1,1]])
f2 = multivariate_normal([2,0], [[2,1],[1,4]])
x_domain, y_domain = np.mgrid[-3:3:.01, -3:3:.01]
axises = np.dstack((x_domain, y_domain))
plt.figure(3)
```

```
plt.title("3-4")
plt.contourf(x_domain, y_domain, f1.pdf(axises) - f2.pdf(axises), 20)
plt.colorbar()
```

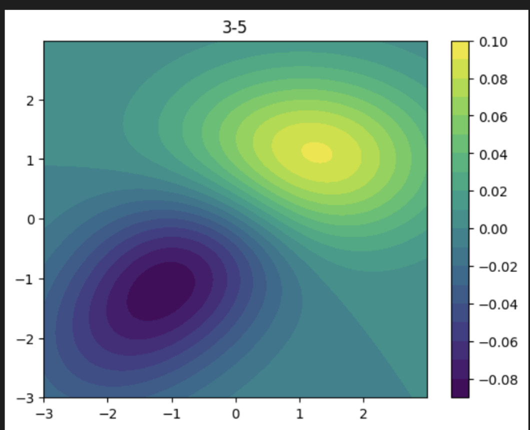
```
<matplotlib.colorbar.Colorbar at 0x159551550>
```



Part 5:

```
f1 = multivariate_normal([1,1], [[2,0],[0,1]])
f2 = multivariate_normal([-1,-1], [[2,1],[1,2]])
x_domain, y_domain = np.mgrid[-3:3:.01, -3:3:.01]
axises = np.dstack((x_domain, y_domain))
plt.figure(4)
plt.title("3-5")
plt.contourf(x_domain, y_domain, f1.pdf(axises) - f2.pdf(axises), 20)
plt.colorbar()
```

```
<matplotlib.colorbar.Colorbar at 0x15965fa00>
```



Question 4: Eigenvectors of the Gaussian Covariance Matrix

```
x1 = multivariate_normal.rvs(mean=3.0, cov=9.0, size=100, random_state=1)

x2 = 0.5 * x1 + multivariate_normal.rvs(mean=4.0, cov=4.0, size=100,
random_state=4)

position_vectors = np.vstack((x1, x2))
```

Part 1:

```
## Part 1:

mean = np.mean(position_vectors,axis=0)

print("Mean:",mean)
```

```
Mean: [3.18174856 5.66142143]
```

Part 2:

```
## Part 2:

covariance = np.cov(position_vectors)

print("Covariance matrix:\n",covariance)
```

```
Covariance matrix:
 [[7.12274112 3.78326186]
 [3.78326186 5.82706493]]
```

Part 3:

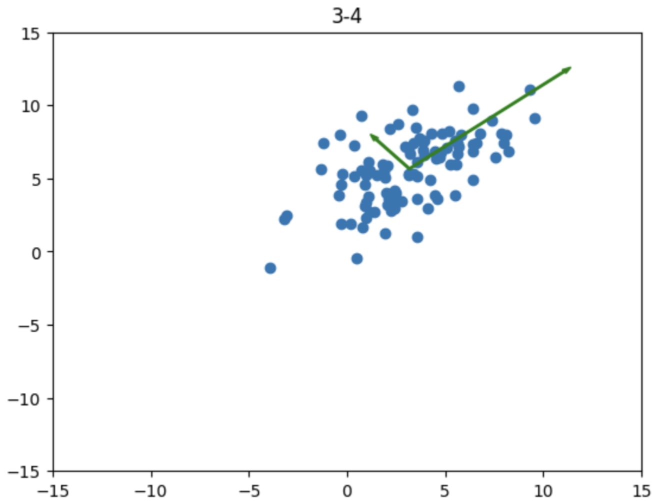
```
## Part 3:

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

print("Eigenvectors:\n",eigenvectors)

print("Eigenvalues:\n",eigenvalues)
```

```
Eigenvectors:
 [[ 0.76445448 -0.64467771]
 [ 0.64467771  0.76445448]]
Eigenvalues:
 [10.31323137  2.63657468]
```

Part 4:

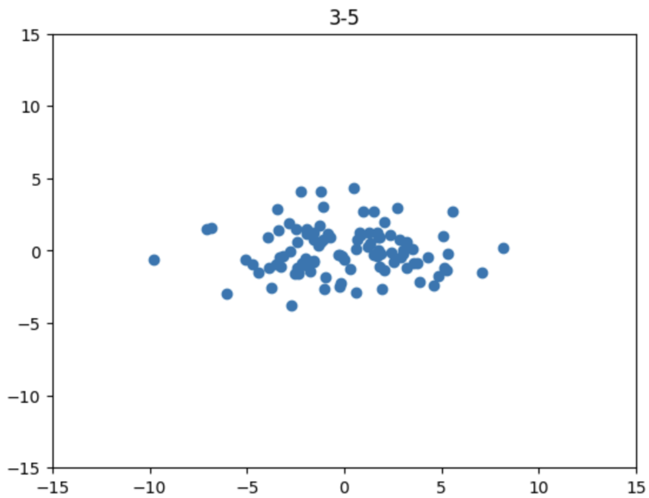
```
## Part 4:
plt.axis((-15,15,-15,15))
plt.title("3-4")
plt.scatter(position_vectors[0],position_vectors[1])
plt.arrow(*mean, *(eigenvalues[1]*eigenvectors[:,1]), color='green', width=0.1)
```

```
plt.arrow(*mean, *(eigenvalues[0]*eigenvectors[:,0]), color='green', width=0.1)
```



Part 5:

```
## Part 5:

rotated_points = eigenvectors.T @ (position_vectors.T - mean).T

plt.axis((-15,15,-15,15))

plt.title("3-5")

plt.scatter(rotated_points[0],rotated_points[1]);
```



Question 8: Gaussian Classifiers for Digits and Spam

```
import sys
if sys.version_info[0] < 3:
   raise Exception("Python 3 not detected.")
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
from scipy import io
from save_csv import results_to_csv

if __name__ == "__main__":
   for data_name in ["mnist", "spam"]:
       data = np.load(f"../data/{data_name}-data-hw3.npz")
       print("\nloaded %s data!" % data_name)
       fields = "test_data", "training_data", "training_labels"
       for field in fields:
           print(field, data[field].shape)
```

```
loaded mnist data!
test_data (10000, 1, 28, 28)
training_data (60000, 1, 28, 28)
training_labels (60000,)

loaded spam data!
test_data (1000, 32)
training_data (4172, 32)
training_labels (4172,)
```

```
###### QUESTION 8: Gaussian Classifiers for Digits and SPAM
```

Part 1:

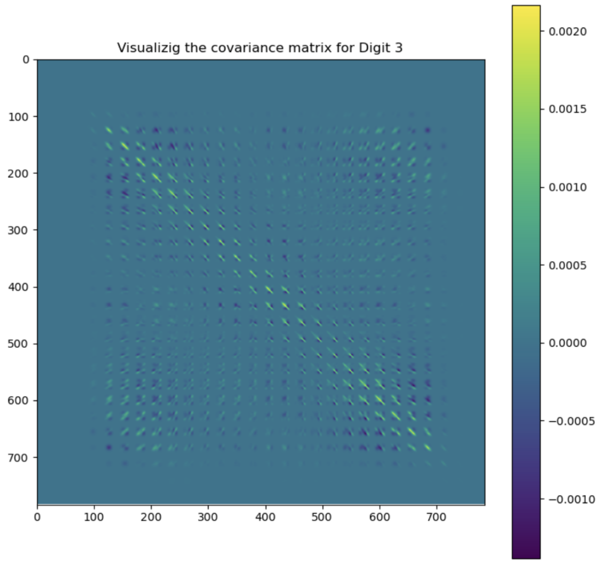
```
### Question 1 and 2
   # Load the MNIST training data
   print("\nMNIST:")
   data = np.load(f"../data/mnist-data-hw3.npz")
   mnist_training_data = data["training_data"]
   mnist_training_labels = data["training_labels"]
   # Reshape them to match
   mnist_training_data = np.reshape(mnist_training_data, (60000, 784))
   mnist_training_labels = np.reshape(mnist_training_labels,(60000, 1))
   # Check the shapes
   print(f"\nData size: {mnist_training_data.shape}")
   print(f"Labels size: {mnist_training_labels.shape}")

   # Contrast Normalizing the Mnist image data before using their values by
dividing it by its norm
   mnist_training_data = mnist_training_data / np.linalg.norm(mnist_training_data,
axis=1)[:, None]

   # Fitting a Gaussian distribution to each digit using MLE
   samples_for_mean = {}
   samples_for_covariance = {}

   # Computing a mean and a covariance matrix for each digit class
   for digit in list(np.unique(mnist_training_labels)):

       digits_data = mnist_training_data[mnist_training_labels.reshape(-1,) ==
digit, :]
       # print(digits_data.shape)
       mean = digits_data.mean(axis=0)
       covariance = np.cov(digits_data.T)
       # Set the mean and covariance of each digit.
       samples_for_mean[digit] = mean
       samples_for_covariance[digit] = covariance
```

```
   # Print and check the shapes
   print(f"\nFitting Gaussian Distribution: \nMean: {samples_for_mean.keys()}
\nLength of mean of digit 5: {len(samples_for_mean[5])}")
   print(f"Covariance: {samples_for_covariance.keys()} \nLength of covariance of
digit 7: {len(samples_for_covariance[7])}")
```

```
MNIST:

Data size: (60000, 784)
Labels size: (60000, 1)

Fitting Gaussian Distribution:
Mean: dict_keys([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
Length of mean of digit 5: 784
Covariance: dict_keys([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
Length of covariance of digit 7: 784
```

Part 2:

```
### Question 2: Visualising the covariance matrix for a particular class (I chose
the digit 3)

   figure = plt.figure(0)
   # print(samples_for_covariance.keys())
   plt.title("Visualizig the covariance matrix for Digit 3")
   plt.imshow(samples_for_covariance[3])
   plt.colorbar()
   # plt.show()
   plt.savefig('MNIST_visualization_of_covariance.png')
```



The covariances on the diagonal are brighter than the covariances that are not on the diagonal which means that they have a higher covariance value on the diagonal. The terms on the diagonal should be a higher value since the values on the diagonal are where the covariance is 1 (since it's the covariance of the same sample with itself). Everywhere else is less than or equal to 1 because it's the covariance between 2 different samples.

Part 3:

```
### Question 3: Classification of Digits
```

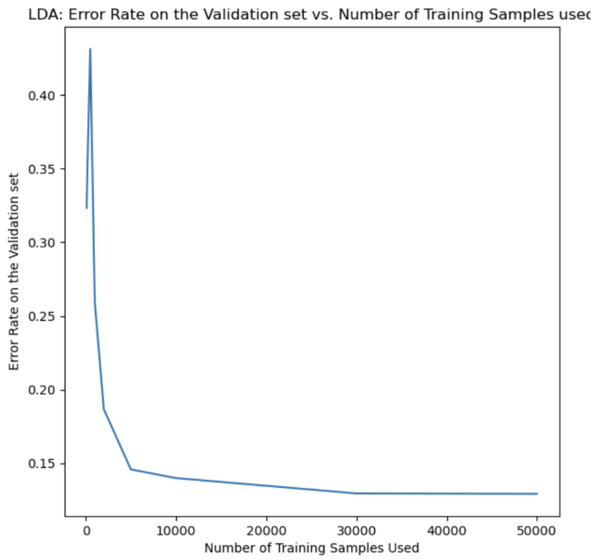
```python
    ## Partition the data just like HW1:
    # Concatenating the labels with data in order to shuffle better
    concatenated_data = np.concatenate((mnist_training_data, mnist_training_labels),
axis=1)
    np.random.shuffle(concatenated_data)

    # Split using the amount we want in validation set
    amount_set_aside = 10000
    mnist_validation, mnist_training = concatenated_data[0:amount_set_aside, :],
concatenated_data[amount_set_aside:, :]

    mnist_validation_data = mnist_validation[:, :-1]
    mnist_validation_labels = mnist_validation[:, -1:]

    mnist_training_data = mnist_training[:, :-1]
    mnist_training_labels = mnist_training[:, -1:]

    #Print the datasets' shapes
    print(f"\nPartitioned: \nTraining data: {mnist_training_data.shape} \nTraining
labels: {mnist_training_labels.shape}")
    print(f"Validation data: {mnist_validation_data.shape} \nValidation labels:
{mnist_validation_labels.shape}")


    ## Normalizing the data
    training_normalized = np.sqrt((mnist_training_data ** 2).sum(axis = 1))
    mnist_training_data = mnist_training_data / training_normalized.reshape(-1, 1)

    validation_normalized = np.sqrt((mnist_validation_data ** 2).sum(axis = 1))
    mnist_validation_data = mnist_validation_data /
validation_normalized.reshape(-1, 1)

    mnist_testing_data = np.reshape(data["test_data"], (10000, 784))
    testing_normalized = np.sqrt((mnist_testing_data ** 2).sum(axis = 1))
    mnist_testing_data = mnist_testing_data / testing_normalized.reshape(-1, 1)
    # checking shapes
    print(f"\nAfter Normalization: \nTraining data: {mnist_training_data.shape}")
    print(f"Validation data: {mnist_validation_data.shape}")
    print(f"Testing data: {mnist_testing_data.shape}")
```

```
Partitioned:
Training data: (50000, 784)
Training labels: (50000, 1)
Validation data: (10000, 784)
Validation labels: (10000, 1)

After Normalization:
Training data: (50000, 784)
Validation data: (10000, 784)
Testing data: (10000, 784)
```

A)

```python
### Question 8-3: Part a) Doing LDA for MNIST:

    training_points = [100, 200, 500, 1000, 2000, 5000, 10000, 30000, 50000]
    validation_accuracy_dict = {"validation": []}
    predictions = []

    ## Function to train LDA for the MNIST
    def Mnist_lda(train_data, train_labels):
    #     print(train_labels.shape)
    #     print(train_labels.shape)
```

```
        a,b = train_data.shape
        priors = np.zeros((10,1))
        covariance = np.zeros((b,b)).astype(np.float32)
        samples_for_mean = np.zeros((10, b))
        train_labels = train_labels.reshape(-1,)

        ## Calculates prior, mean, and covariance for each digit
        for digit in range(0, 10):

            priors[int(digit)] = (train_labels == digit).astype(np.int32).sum() / a

            data = train_data[train_labels == digit, :]

            samples_for_mean[int(digit), :] = data.mean(axis=0)
            covariance += np.cov(data.T)

        covariance /= 10
        return priors, samples_for_mean, covariance

    # Training the LDA Model for MNIST for each training point
    print("\nLDA:")
    for training_point in training_points:
        print(f"Training with {training_point} points")

        training_data = mnist_training_data[:training_point, :]
        training_labels = mnist_training_labels[:training_point, :]

        priors, means, covariance = Mnist_lda(training_data, training_labels)
        # Fixing and preventing the singular covariance matrices. I just add a very
small value (1e-6) to the diagonal values of the covariance to make the eigenvalues
positive.
        covariance = covariance + (1e-6) * np.eye(*covariance.shape)

        validation_num = mnist_validation_data.shape[0]
        validation_out = np.zeros((validation_num, 10))

        # Calculate mean, prior and covariance for each digit
        for digit in range(0, 10):
            prior = priors[digit]
            mean = means[digit, :].reshape(-1, 1)

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

            validation_out[:, digit] = (mnist_validation_data.dot(weight) +
alpha).reshape(-1,)
        # Calculate validation predictions for all digits
        validation_prediction = np.argmax(validation_out, axis=1).reshape(-1, 1)

        predictions.append(validation_prediction)
        validation_accuracy_dict["validation"].append((mnist_validation_labels ==
validation_prediction).mean())

        print(f"Validation accuracy:
{validation_accuracy_dict['validation'][-1]}\n")
        print()


    figure = plt.figure(1)
    plt.plot(training_points, [1 - x for x in
validation_accuracy_dict["validation"]])
    plt.title("LDA: Error Rate on the Validation set vs. Number of Training Samples
used")
    plt.xlabel("Number of Training Samples Used")
    plt.ylabel("Error Rate on the Validation set")
    # plt.show()
    plt.savefig('LDA.png')
```

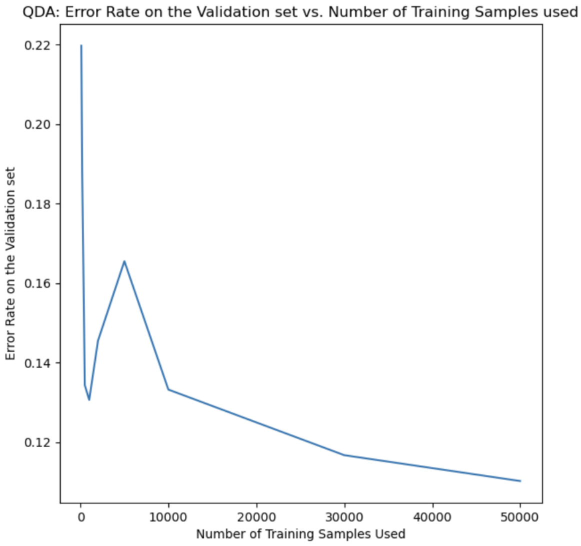



```
LDA:
Training with 100 points
Validation accuracy: 0.6334

Training with 200 points
Validation accuracy: 0.6434

Training with 500 points
Validation accuracy: 0.5534

Training with 1000 points
Validation accuracy: 0.7283

Training with 2000 points
Validation accuracy: 0.8116

Training with 5000 points
Validation accuracy: 0.846

Training with 10000 points
Validation accuracy: 0.8585

Training with 30000 points
Validation accuracy: 0.8648

Training with 50000 points
Validation accuracy: 0.8679
```

B)

```
### Question 8-3: Part b) Doing QDA for MNIST:
   training_points = [100, 200, 500, 1000, 2000, 5000, 10000, 30000, 50000]
   validation_accuracy_dict = {"validation": []}
   predictions = []

   ## Function to train QDA for the MNIST
   def mnist_qda(train_data, train_labels):
       n,d = train_data.shape
       priors = np.zeros((10,1))
       covariances = np.zeros((10,d,d)).astype(np.float32)
       samples_for_mean = np.zeros((10, d))

       digits = list(np.unique(train_labels))
       # Calculate prior, mean, and covariance for each digit
       for digit in digits:
           priors[int(digit)] = ((train_labels == digit).astype(np.int32).sum() /
n)

           data = train_data[train_labels == digit, :]

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

       return priors, samples_for_mean, covariances

   # Training QDA on MNIST
   print("\nQDA: ")
   for training_point in training_points:
       print(f"Training with {training_point} points")

       train_data = mnist_training_data[:training_point, :]
       train_labels = mnist_training_labels[:training_point, :].reshape(-1,)

       priors, means, covariances = mnist_qda(train_data, train_labels)
```

```
       validation_num = mnist_validation_data.shape[0]
       validation_out = np.zeros((validation_num, 10))

       train_num = mnist_training_data.shape[0]
       train_out = np.zeros((train_num, 10))

       validation_num = mnist_validation_data.shape[0]
       validation_out = np.zeros((validation_num, 10))
       # calculate mean, prior and covariance of a single digit and finding the
validation prediction
       for digit in range(0, 10):
           prior = priors[digit]
           mean = means[digit, :]
           covariance = covariances[digit, :, :]

       # Fixing and preventing the singular covariance matrices. I just add a very
small value (1e-6) to the diagonal values of the covariance to make the eigenvalues
positive.
           covariance = covariance + (1e-6) * np.eye(*covariance.shape)
           alpha = -0.5 * np.linalg.det(covariance) + prior

           mean_centered_validation = mnist_validation_data - mean
           validation_weight = np.linalg.solve(covariance,
mean_centered_validation.T)

           a, b = mean_centered_validation.shape
           out_diagonal = np.array([mean_centered_validation[n, :].reshape(1,
-1).dot(validation_weight[:, n].reshape(-1, 1)) for n in range(a)])

           validation_prediction = -0.5 * out_diagonal + alpha
           validation_out[:, digit] =
validation_prediction.reshape(validation_num,)

       validation_prediction_final = np.argmax(validation_out, axis=1).reshape(-1,
1)

       predictions.append(validation_prediction_final)
       validation_accuracy_dict["validation"].append((mnist_validation_labels ==
validation_prediction_final).mean())

       print(f"Validation acc: {validation_accuracy_dict['validation'][-1]}\n")


   figure = plt.figure(2)
   plt.plot(training_points, [1 - x for x in
validation_accuracy_dict["validation"]])
   plt.title("QDA: Error Rate on the Validation set vs. Number of Training Samples
used")
   plt.xlabel("Number of Training Samples Used")
   plt.ylabel("Error Rate on the Validation set")
   # plt.show()
   plt.savefig('QDA.png')
```




```
QDA:
Training with 100 points
Validation acc: 0.7529

Training with 200 points
Validation acc: 0.8232

Training with 500 points
Validation acc: 0.8831

Training with 1000 points
Validation acc: 0.9071

Training with 2000 points
Validation acc: 0.88

Training with 5000 points
Validation acc: 0.8262

Training with 10000 points
Validation acc: 0.8641

Training with 30000 points
Validation acc: 0.8805

Training with 50000 points
Validation acc: 0.8815
```

C) Which was better?

LDA performed better because generally LDA does better on larger datasets. Also, looking at their error rate graphs we can see that the validation accuracy for QDA decreases as our number of training points increases but LDA's validation increases as we increase the number of training points. The error graph also shows that LDA has a very smooth decreasing error rate but QDA does not and has some bumps before it starts to also decrease. ALso, QDA on larger datasets can cause overfitting.

LDA:

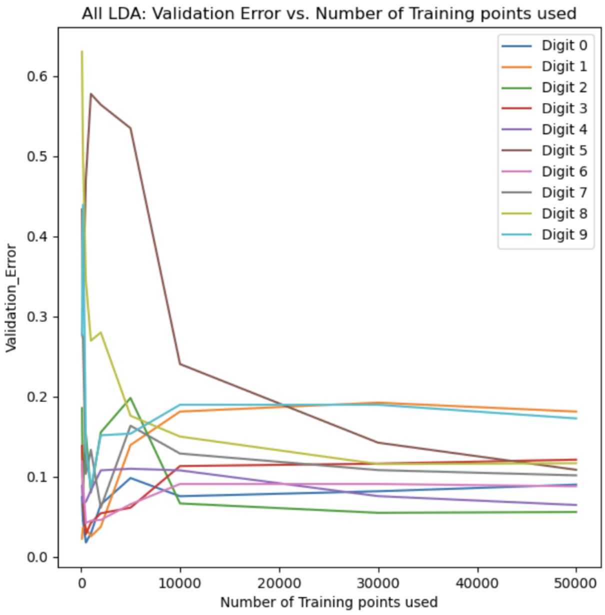
```
LDA:
Training with 100 points
Validation accuracy: 0.6767

Training with 200 points
Validation accuracy: 0.6451

Training with 500 points
Validation accuracy: 0.5688

Training with 1000 points
Validation accuracy: 0.7404

Training with 2000 points
Validation accuracy: 0.8132

Training with 5000 points
Validation accuracy: 0.8541

Training with 10000 points
Validation accuracy: 0.86

Training with 30000 points
Validation accuracy: 0.8704

Training with 50000 points
Validation accuracy: 0.8707
```

QDA:

```
QDA:
Training with 100 points
Validation acc: 0.7803

Training with 200 points
Validation acc: 0.812

Training with 500 points
Validation acc: 0.8657

Training with 1000 points
Validation acc: 0.8694

Training with 2000 points
Validation acc: 0.8545

Training with 5000 points
Validation acc: 0.8345

Training with 10000 points
Validation acc: 0.8668

Training with 30000 points
Validation acc: 0.8833

Training with 50000 points
Validation acc: 0.8898
```

D)

```
    #### QUESTION 8-3: Part d) plotting all 10 curves for all 10 digits for LDA
    figure = plt.figure(3)

    digits_error = {}
    # generate error rates for all predictions
    for prediction in predictions:
        for digit in range(0, 10):
            digit_predictions = prediction[mnist_validation_labels == digit]
            labels = mnist_validation_labels[mnist_validation_labels == digit]
            digits_error[digit] = digits_error.get(digit, []) + [(digit_predictions
== labels).mean()]

    # plot the error for each digit
    for digit in digits_error.keys():
        plt.plot(training_points, [1 - digit for digit in digits_error[digit]])

    plt.legend([f"Digit {digit}" for digit in list(range(0, 10))], loc='upper
right')
    plt.title("All LDA: Validation Error vs. Number of Training points used")
    plt.xlabel("Number of Training points used")
    plt.ylabel("Validation_Error")
    # plt.show()
    plt.savefig('LDA_all_10.png')
```



Which digit is easiest to classify?

Digit 0 and 1 are the easiest digits to classify. I found these to be the easiest to classify because they have the most unique features as other digits. Looking at their LDA Loss graph as well, they have the smallest minimum value on the all digits LDA graph.

Part 4: Kaggle Submission

Kaggle username: **Hiva Mohammadzadeh**

Kaggle Scores:

MNIST: 0.87933 = 88%

```
## QUESTION 8-4: Kaggle Submission on the better performed model on MNIST.

   # Use the calculated priors and means and covariance from training LDA to
generate predictions for test data
   testing_num = mnist_testing_data.shape[0]
   # print(mnist_testing_data.shape)
   test_out = np.zeros((testing_num, 10))

   # Run on test data and make predictions for each digit
   for digit in range(0, 10):
       prior = priors[digit]
       mean = means[digit, :].reshape(-1, 1)

       weight = np.linalg.solve(covariance.T, mean)

       alpha = -0.5 * weight.T.dot(mean) + np.log(prior)

       test_out[:, digit] = (mnist_testing_data.dot(weight) + alpha).reshape(-1,)

   test_predictions = np.argmax(test_out, axis=1).reshape(-1,)
   # Save the result to a csv file
   results_to_csv(test_predictions)
   print("Successfully ran on test data for MNIST and saved predictions to the csv
file\n"
```

```
Successfully ran on test data for MNIST and saved predictions to the csv file
```

Part 5: LDA and QDA on SPAM

```
#### QUESTION 8-5: SPAM Classification of Mails using LDA and QDA

   # Load the SPAM training data
   print("\nSPAM:\n")
   data = np.load(f"../data/spam-data-hw3.npz")
   spam_training_data = data["training_data"]
   spam_training_labels = data["training_labels"]
   #reshape them to match
   spam_training_labels = np.reshape(spam_training_labels,(4172, 1))
   # Check the shapes
   print(f"data size: {spam_training_data.shape}")
   print(f"labels size: {spam_training_labels.shape}")

   # Fitting a Gaussian distribution to each digit using MLE
   samples_for_mean = {}
   samples_for_covariance = {}

   # Computing a mean and a covariance matrix for each digit class
   for mail in list(np.unique(spam_training_labels)):

       data_for_mail = spam_training_data[spam_training_labels.reshape(-1,) ==
mail, :]

       mean = data_for_mail.mean(axis=0)
       covariance = np.cov(data_for_mail.T)

       samples_for_mean[mail] = mean
       samples_for_covariance[mail] = covariance

   # Print and check the shapes
   print(f"\nFitting Gaussian Distribution: \nMean: {samples_for_mean.keys()}
\nLength of mean of spam email: {len(samples_for_mean[1])}")
   print(f"Covariance: {samples_for_covariance.keys()} \nLength of covariance of
non spam email: {len(samples_for_covariance[0])}")


   # Partitioning dataset using bag of words just like HW1

   # Concatenating the labels with data in order to shuffle better
   concatenated_data = np.concatenate((spam_training_data, spam_training_labels),
axis = 1)

   #Shuffling the data
   np.random.shuffle(concatenated_data)
   spam_training_data = concatenated_data[:,:-1]
   spam_training_labels = np.reshape(concatenated_data[:,-1], (4172, 1))
   # print(training_labels.shape)

   #Split using the amount we want in validation set
   amount_set_aside = 500
   spam_validation_data, spam_validation_labels =
spam_training_data[:amount_set_aside,:], spam_training_labels[:amount_set_aside,:]
   spam_training_data, spam_training_labels =
spam_training_data[amount_set_aside:,:], spam_training_labels[amount_set_aside:,:]


   #Print the datasets' shapes
   print(f"\nPartitioned: \nTraining data: {spam_training_data.shape} \nTraining
labels: {spam_training_labels.shape}")
   print(f"Validation data: {spam_validation_data.shape} \nValidation labels:
{spam_validation_labels.shape}")

   spam_testing_data = data["test_data"]
   print(f"Testing data: {spam_testing_data.shape}")
```

```
### Question 8-5: LDA on Spam

   ## Function to train LDA for the SPAM
   def spam_lda(training_data, training_labels):
       a, b = training_data.shape
       priors = np.zeros((2,1))
       covariance = np.zeros((b,b)).astype(np.float32)
       samples_for_mean = np.zeros((2, b))
       training_labels = training_labels.reshape(-1,)

       ## Calculates prior, mean, and covariance for each mail
       for mail in [0,1]:

           priors[int(mail)] = (training_labels == mail).astype(np.int32).sum() / a

           data = training_data[training_labels == mail, :]
           samples_for_mean[int(mail), :] = data.mean(axis = 0)
           covariance += np.cov(data.T)

       covariance /= 2
       return priors, samples_for_mean, covariance

   # Training the LDA Model for MNIST
   priors, means, covariance = spam_lda(spam_training_data, spam_training_labels)

   train_num = spam_training_data.shape[0]
   train_out = np.zeros((train_num, 2))

   validation_num = spam_validation_data.shape[0]
   validation_out = np.zeros((validation_num, 2))

   # Calculate mean, prior and covariance for each digit
   for mail in [0,1]:
       prior = priors[mail]
       mean = means[mail, :].reshape(-1, 1)

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

       validation_out[:, mail] = (spam_validation_data.dot(weight) +
alpha).reshape(-1,)

   # Calculate validation predictions for all mails
   validation_predictions = np.argmax(validation_out, axis=1).reshape(-1, 1)

   print("\nLDA:")
   print(f"Validation accuracy: {(spam_validation_labels ==
validation_predictions).mean()}\n")

### QUESTION 8-5: Doing QDA on Spam

  ## Function to train QDA for the SPAM
   def spam_qda(training_data, training_labels):
       a, b = training_data.shape
       priors = np.zeros((2,1))
       covariances = np.zeros((2,b,b)).astype(np.float32)
       samples_for_mean = np.zeros((2, b))
       training_labels = training_labels.reshape(-1,)

       ## Computes prior, mean, and covariance for each mail
       for mail in [0,1]:

           priors[int(mail)] = ((training_labels == mail).astype(np.int32).sum() /
a)

           data = training_data[training_labels == mail, :]

           samples_for_mean[int(mail),:] = data.mean(axis=0)
```

```
            covariances[int(mail), :, :] = np.cov(data.T)

        return priors, samples_for_mean, covariances

    # Train QDA on SPAM
    priors, means, covariances = spam_qda(spam_training_data, spam_training_labels)
    validation_num = spam_validation_data.shape[0]
    validation_out = np.zeros((validation_num, 2))

    # calculate the qda validation accuracies for mails
    for mail in [0,1]:
        prior = priors[mail]
        mean = means[mail, :]
        covariance = covariances[mail, :, :]
        # Fixing and preventing the singular covariance matrices. I just add a very
small value (1e-6) to the diagonal values of the covariance to make the eigenvalues
positive.
        covariance = covariance + (1e-6) * np.eye(*covariance.shape)

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

        mean_centered_validation= spam_validation_data - mean
        validation_weight = np.linalg.solve(covariance, mean_centered_validation.T)

        a, b = mean_centered_validation.shape
        out_diagonal = np.array([mean_centered_validation[n, :].reshape(1,
-1).dot(validation_weight[:, n].reshape(-1, 1)) for n in range(a)])

        validation_prediction = -0.5 * out_diagonal + alpha
        validation_out[:, mail] = validation_prediction.reshape(validation_num,)


    validation_prediction_final = np.argmax(validation_out, axis=1).reshape(-1, 1)
    print("QDA:")
    print(f"Validation accuracy: {(spam_validation_labels ==
validation_prediction_final).mean()}\n")



 ## QUESTION 8-4: Kaggle Submission on the better performed model on SPAM.

    # Use the calculated priors and means and covariance from training LDA to
generate predictions for test data
    testing_num = spam_testing_data.shape[0]
    # print(mnist_testing_data.shape)
    out_test = np.zeros((testing_num, 2))

    # Run on test data and make predictions for each mail
    for mail in [0,1]:
        prior = priors[mail]
        mean = means[mail, :].reshape(-1, 1)

        weight = np.linalg.solve(covariance.T, mean)

        alpha = -0.5 * weight.T.dot(mean) + np.log(prior)

        out_test[:, mail] = (spam_testing_data.dot(weight) + alpha).reshape(-1,)

    test_predictions = np.argmax(out_test, axis=1).reshape(-1,)
    # Save the result to a csv file
    results_to_csv(test_predictions)
    print("Successfully ran on test data for SPAM and saved predictions to the csv
file\n")
```

Kaggle username: **Hiva Mohammadzadeh**

Kaggle Scores:

b) SPAM: 0.784 = 78%

```
SPAM:

data size: (4172, 32)
labels size: (4172, 1)

Fitting Gaussian Distribution:
Mean: dict_keys([0, 1])
Length of mean of spam email: 32
Covariance: dict_keys([0, 1])
Length of covariance of non spam email: 32

Partitioned:
Training data: (3672, 32)
Training labels: (3672, 1)
Validation data: (500, 32)
Validation labels: (500, 1)
Testing data: (1000, 32)

LDA:
Validation accuracy: 0.76

Successfully ran on test data for SPAM and saved predictions to the csv file

QDA:
Validation accuracy: 0.788
```