In [2]:
```
# This file is in scripts/load.py
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
import random
```

Problem Specification

In this problem, we will build classifiers based on Gaussian discriminant analysis. However, I will be forcing myself to NOT use any Machine Learning libraries in Python. That means no precious TensorFlow, no PyTorch, and NO SciKit Learn! After I classify my two datasets (Digits and Spam Emails), I will then submit the results to Kaggle, which happens to be where I obtained these datasets.

Part 1

First, we need to load in all of our data.

In [14]:
```
data = [0, 0]
for i, data_name in enumerate(["mnist-data-hw3.npz", "spam-data-hw3.npz"]):
    data[i] = np.load(f"data/{data_name}")
    print("\nloaded %s data!" % data_name)
    fields = "test_data", "training_data", "training_labels"
    for field in fields:
        print(field, data[i][field].shape)
        #Downloads/Classes SP23/CS_189/Homework/hw3-2023/data/mnist-data-hw3
mnist_data = data[0]
spam_data = data[1]
```

```
loaded mnist-data-hw3.npz data!
test_data (10000, 1, 28, 28)
training_data (60000, 1, 28, 28)
training_labels (60000,)

loaded spam-data-hw3.npz data!
test_data (1000, 32)
training_data (4172, 32)
training_labels (4172,)
```

In [15]:
```
mnist_train = mnist_data["training_data"]
mnist_train_labels = mnist_data["training_labels"]
mnist_test = mnist_data["test_data"]

spam_train = spam_data["training_data"]
spam_train_labels = spam_data["training_labels"]
spam_test = spam_data["test_data"]
```

Part 2

Taking pixel values as features (no new features yet), we fit a Gaussian distribution to each digit class using maximum likelihood estimation. This involves computing a mean and a covariance matrix for each digit class.

In [50]:
```
for i in range(len(mnist_train)):
    mnist_train[i] = mnist_train[i] / np.linalg.norm(mnist_train[i])

for i in range(len(mnist_test)):
    mnist_test[i] = mnist_test[i] / np.linalg.norm(mnist_test[i])
```

In [51]:
```
mnist_train = np.squeeze(mnist_train)
mnist_test = np.squeeze(mnist_test)
mnist_train.shape
```

Out[51]:
```
(60000, 28, 28)
```

In [52]:
```
all_means = []
all_cov = []

for i in range(2):
    all_digit = mnist_train[mnist_train_labels == i]
    mean1 = np.mean(all_digit, axis=0)
    mean2 = np.concatenate(mean1)
    all_means.append(mean1)

    cov_array = []
    for num in all_digit:
        num = np.concatenate(num)
        cov = np.outer((num-mean2), (num-mean2))
        cov_array.append(cov)

    all_cov.append(np.mean(cov_array, axis=0))
```

In [53]:
```
for i in range(2, 4):
    all_digit = mnist_train[mnist_train_labels == i]
    mean1 = np.mean(all_digit, axis=0)
    mean2 = np.concatenate(mean1)
    all_means.append(mean1)

    cov_array = []
    for num in all_digit:
        num = np.concatenate(num)
        cov = np.outer((num-mean2), (num-mean2))
        cov_array.append(cov)

    all_cov.append(np.mean(cov_array, axis=0))
```

In [54]:
```
for i in range(4, 6):
    all_digit = mnist_train[mnist_train_labels == i]
    mean1 = np.mean(all_digit, axis=0)
    mean2 = np.concatenate(mean1)
    all_means.append(mean1)

    cov_array = []
    for num in all_digit:
        num = np.concatenate(num)
        cov = np.outer((num-mean2), (num-mean2))
        cov_array.append(cov)

    all_cov.append(np.mean(cov_array, axis=0))
```

In [55]:
```
for i in range(6, 8):
    all_digit = mnist_train[mnist_train_labels == i]
    mean1 = np.mean(all_digit, axis=0)
    mean2 = np.concatenate(mean1)
    all_means.append(mean1)

    cov_array = []
    for num in all_digit:
        num = np.concatenate(num)
        cov = np.outer((num-mean2), (num-mean2))
        cov_array.append(cov)

    all_cov.append(np.mean(cov_array, axis=0))
```

In [56]:
```
for i in range(8, 10):
    all_digit = mnist_train[mnist_train_labels == i]
    mean1 = np.mean(all_digit, axis=0)
    mean2 = np.concatenate(mean1)
    all_means.append(mean1)

    cov_array = []
    for num in all_digit:
        num = np.concatenate(num)
        cov = np.outer((num-mean2), (num-mean2))
        cov_array.append(cov)

    all_cov.append(np.mean(cov_array, axis=0))
```

In [57]:
```
np.array(all_means).shape, np.array(all_cov).shape
```

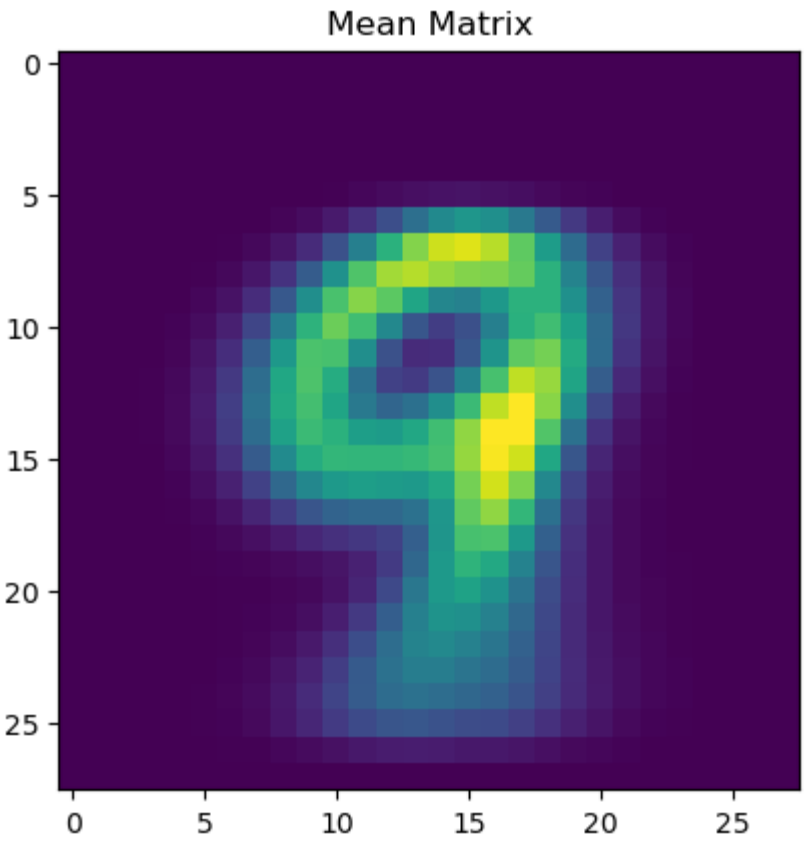
Out[57]:
```
((10, 28, 28), (10, 784, 784))
```

Part 3

Visualizing the mean and covariance matrix of 9 (because its cool).

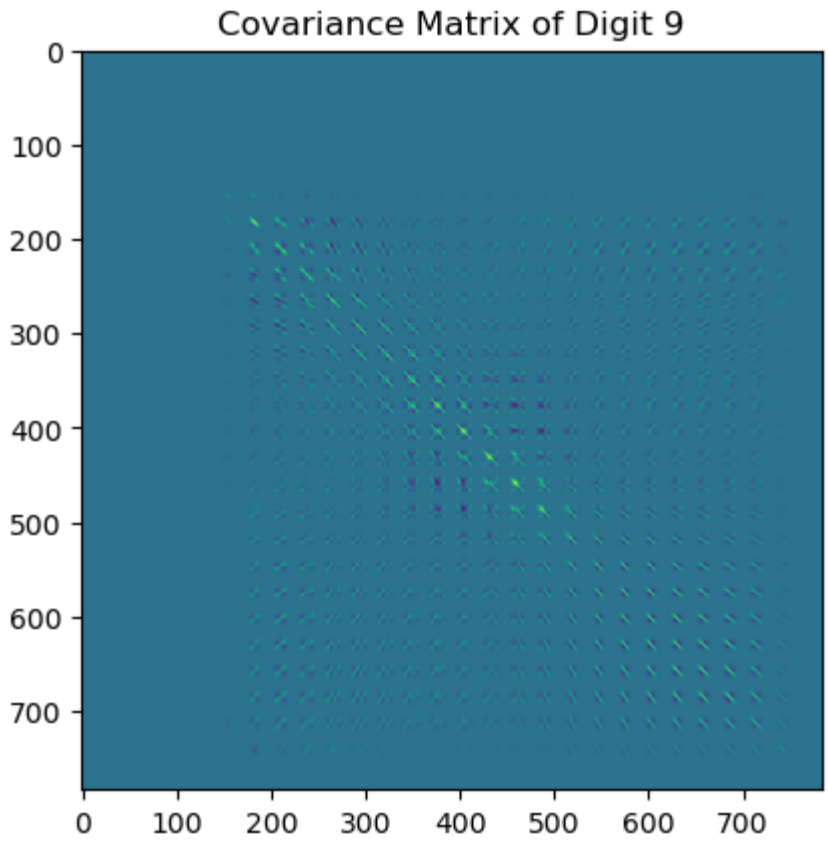
In [23]:
```
plt.imshow(all_means[9])
plt.title("Mean Matrix")
```

Out[23]:
```
Text(0.5, 1.0, 'Mean Matrix')
```



In [24]:
```
plt.imshow(all_cov[9])
plt.title("Covariance Matrix of Digit 9")
```

Out[24]:
```
Text(0.5, 1.0, 'Covariance Matrix of Digit 9')
```



In [22]:
```
import scipy
import sklearn
```

Part 4

Here, I am going to define some functions to help us implement LDA and QDA. These two functions are going to take in training data, training labels, and the number of data points we will train on. They are going to return:

LDA Function - ONE covariance matrix, TEN mean vectors, our training labels (To calculate error rate)

QDA Function - TEN covariance matrices, TEN mean vectors, our training labels (To calculate error rate)LDA Function - ONE covariance matrix, TEN mean vectors, our training labels (To calculate error rate)LDA Function - ONE covariance matrix, TEN mean vectors, our training labels (To calculate error rate)

In [18]:

```
# This function takes in:
#     Training data,
#     Training labels,
#     Num of training points

# This function returns:
#     The covariance matrix,
#     The mean vector for 10 classes,
#     The labels of our training points

def create_parameters_lda(train, labels, train_points):
    all_means = []
    all_cov = []
    random_subset = random.sample(list(np.arange(len(train))), train_points)
    random_train, random_labels = np.array([train[i] for i in random_subset]

    for i in range(10):
        all_digit = random_train[random_labels == i]
        mean1 = np.mean(all_digit, axis=0)
        mean2 = np.concatenate(mean1)
        all_means.append(mean1)

    for num in random_train:
        num = np.concatenate(num)
        cov = np.outer((num-mean2), (num-mean2))
        all_cov.append(cov)


    for i in range(len(all_means)):
        all_means[i] = np.concatenate(all_means[i])

    return np.add(np.mean(all_cov, axis=0), .1 * np.identity(784)), np.array
```

In [19]:
```
# This function takes in:
#     Training data,
#     Training labels,
#     Num of training points

# This function returns:
#     The covariance matrix for 10 classes,
#     The mean vector for 10 classes,
#     The labels of our training points

def create_parameters_qda(train, labels, train_points):
    all_means = []
    all_cov = []
    random_subset = random.sample(list(np.arange(len(train))), train_points)
    random_train, random_labels = np.array([train[i] for i in random_subset]

    for i in range(10):
        all_digit = random_train[random_labels == i]
        mean1 = np.mean(all_digit, axis=0)
        mean2 = np.concatenate(mean1)
        all_means.append(mean1)

        cov_array = []
        for num in all_digit:
            num = np.concatenate(num)
            cov = np.outer((num-mean2), (num-mean2))
            cov_array.append(cov)

        all_cov.append(np.add(np.mean(cov_array, axis=0), .1 * np.identity(7
        np.add(np.mean(all_cov, axis=0), .1 * np.identity(784))

    for i in range(len(all_means)):
        all_means[i] = np.concatenate(all_means[i])

    return np.array(all_cov), np.array(all_means), random_labels
```

Part 5

Here, we are going to create the function which generates predictions based on our mean and covariance matrices (or matrix for LDA). We calculate the posterior probabilities of whether or not a data point is in classes 0-9 based on our mean and covariance matrices. Then we look for the highest probability, and select that label as our prediction.

In [64]:
```
# Predicts classes of each point based on:
#     Mean vectors,
#     Covariance matrix/Covariance matrices (if qda),
#     "lda" or "qda"

def predict(mean, covariance, points, mode):
    results = []
    points = np.array([np.concatenate(points[i]) for i in range(len(points))
    if str.lower(mode) == "lda":
        for i in range(10):
            log_p = scipy.stats.multivariate_normal.logpdf(points, mean=mear
            results.append(log_p)
        predictions = np.argmax(results, axis=0)
        return predictions
    elif str.lower(mode) == "qda":
        for i in range(10):
            log_p = scipy.stats.multivariate_normal.logpdf(points, mean=mear
            results.append(log_p)
        predictions = np.argmax(results, axis=0)
        return predictions
```

Part 6

Our process is going to be:

1. Create a validation set.
2. Classify each image in the validation set using our trained model.
3. Compute the error.
4. Do this for [100, 200, 500, 1000, 2000, 5000] sample points.
5. Plot error rate vs # of sample points.

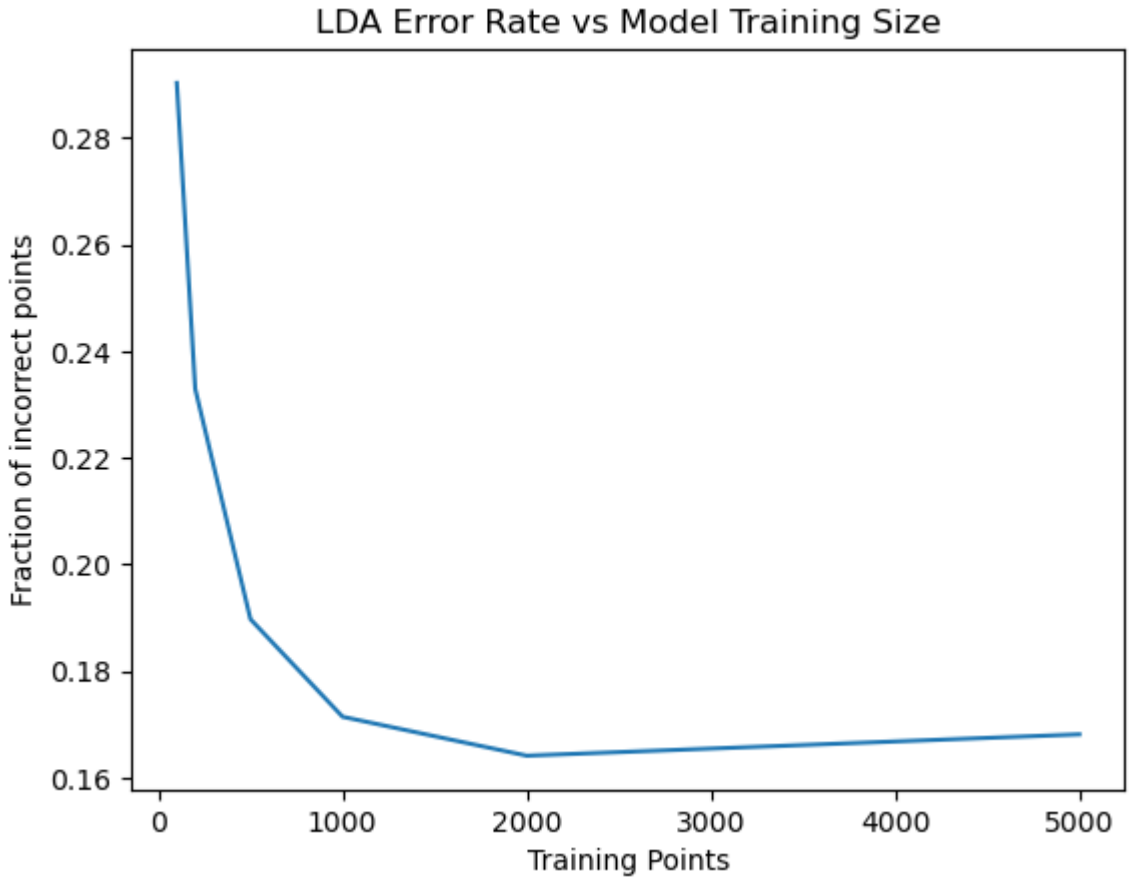
In [23]:
```
# Create validation set of size 10000.
X_train, y_train, X_labels, y_labels = sklearn.model_selection.train_test_sp
```

In [24]:
```
# Create models using following sample points
# First we train the model (Get our parameters)
# Then we predict our 10000 validation values using the model
# After that, we calculate the error and append it to a list for plotting.
error_rate = []
for sample_pts in [100, 200, 500, 1000, 2000, 5000]:
    cov, mean, lmaoDontCare = create_parameters_lda(X_train, X_labels, sampl
    predictions = predict(mean, cov, y_train, "lda")
    error = 1 - sklearn.metrics.accuracy_score(predictions, y_labels)
    error_rate.append(error)
    print("Error for", str(sample_pts), ":", str(error))
```

```
Error for 100 : 0.2903
Error for 200 : 0.2329
Error for 500 : 0.18969999999999998
Error for 1000 : 0.1714
Error for 2000 : 0.16410000000000002
Error for 5000 : 0.16810000000000003
```

In [27]:
```
# Plotting our LDA error
plt.plot([100, 200, 500, 1000, 2000, 5000], error_rate)
plt.xlabel("Training Points")
plt.ylabel("Fraction of incorrect points")
plt.title("LDA Error Rate vs Model Training Size")
```

Out[27]:
```
Text(0.5, 1.0, 'LDA Error Rate vs Model Training Size')
```

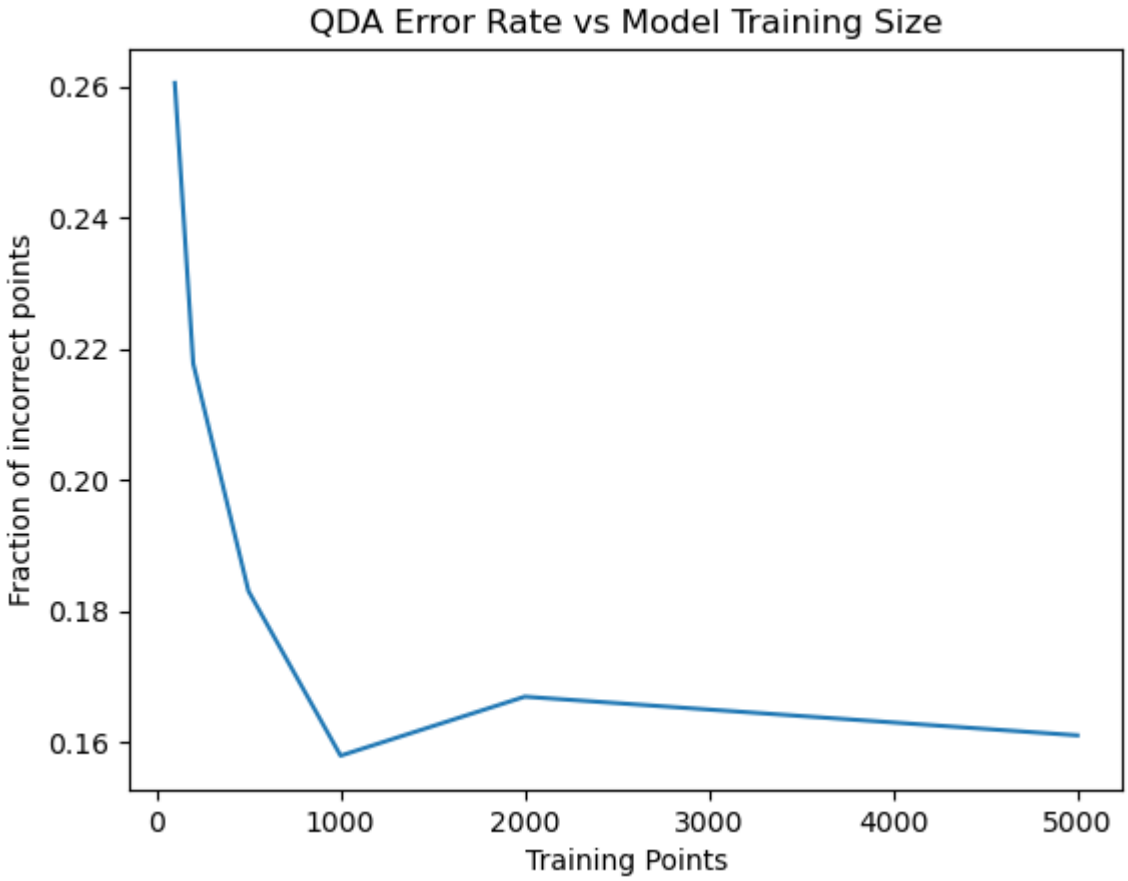


In [25]:
```
# Same process as before, except for QDA.
# Smaller sample sizes
error_rate_qda = []
for sample_pts in [100, 200, 500, 1000, 2000, 5000]:
    cov, mean, lmaoDontCare = create_parameters_qda(X_train, X_labels, sampl
    predictions = predict(mean, cov, y_train, "qda")
    error = 1 - sklearn.metrics.accuracy_score(predictions, y_labels)
    error_rate_qda.append(error)
    print("Error for", str(sample_pts), ":", str(error))
```

```
Error for 100 : 0.26049999999999995
Error for 200 : 0.2177
Error for 500 : 0.18300000000000005
Error for 1000 : 0.15790000000000004
Error for 2000 : 0.16690000000000005
Error for 5000 : 0.16100000000000003
```

In [29]:
```
# Plotting our QDA error.
plt.plot([100, 200, 500, 1000, 2000, 5000], error_rate_qda)
plt.xlabel("Training Points")
plt.ylabel("Fraction of incorrect points")
plt.title("QDA Error Rate vs Model Training Size")
```

Out[29]:
```
Text(0.5, 1.0, 'QDA Error Rate vs Model Training Size')
```



Part 7

It appears that our QDA model performed a little better than our LDA model. If I had to guess, this is likely because the covariance matrices of each digit differ greatly. In other words, digits like 1 and 8 are very different. It doesn't make much sense to assume that the covariance matrix of 1 and 8 are the same, or even similar to each other. With QDA, we assume that our classes all have different covariance matrices, so this likely is why we have a better result with it. With a large number of data points, I expect this result to only get better.

For the reasons above, I am going to use a QDA model for my final predictions to Kaggle. This will (hopefully) give me the best test accuracy.

Part 8

Let us examine the errors of EACH digit, for EACH sample size we tested. Let us also create a plot of these errors, and examine which one is the lowest, and which one is the highest. This may help us later when we create another model to generate predictions for Kaggle.

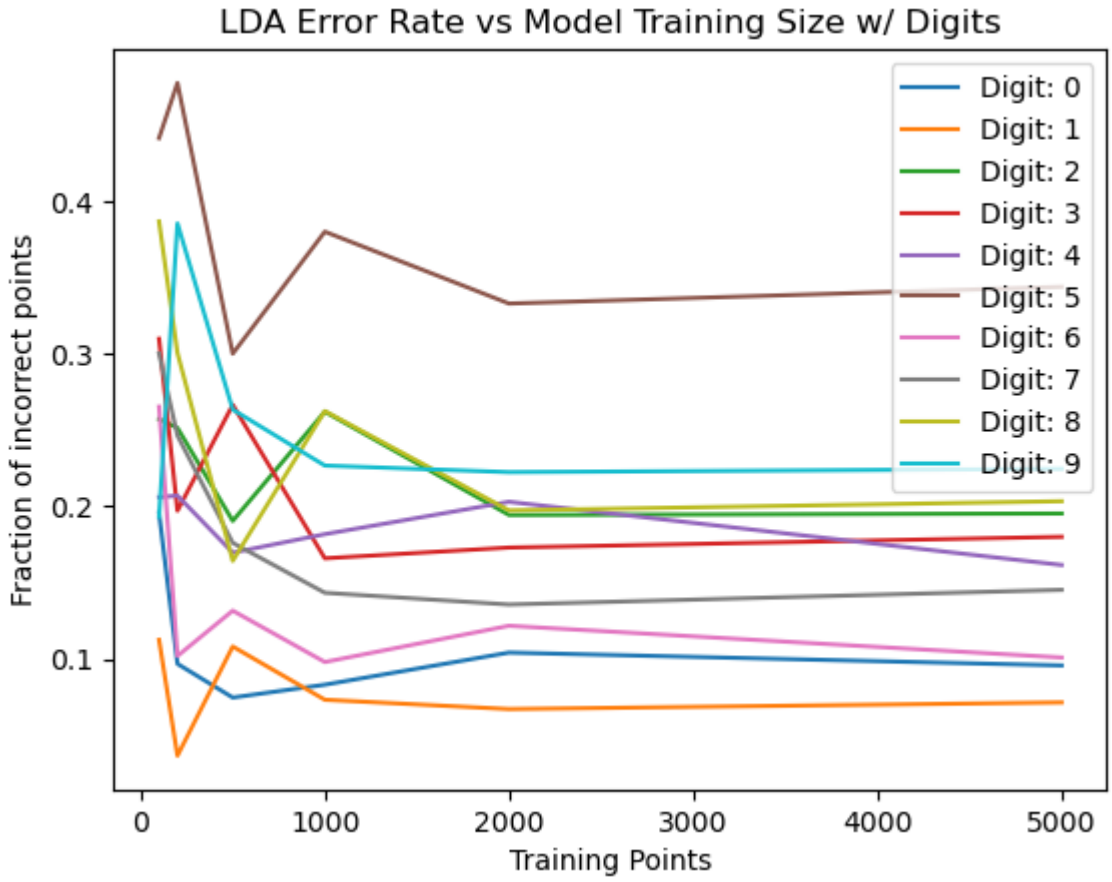
In [32]:
```
error_rate2 = []
for sample_pts in [100, 200, 500, 1000, 2000, 5000]:
    cov, mean, lmaoDontCare = create_parameters_lda(X_train, X_labels, sampl
    predictions = predict(mean, cov, y_train, "lda")
    err = []
    for i in range(10):
        error = 1 - sklearn.metrics.accuracy_score(predictions[y_labels == i
        err.append(error)
        print("Size: ", str(sample_pts), ". Error for digit", str(i), ": ",
    error_rate2.append(np.array(err))
    print('')
```

```
Size:  100 . Error for digit 0 :  0.1957894736842105
Size:  100 . Error for digit 1 :  0.11256544502617805
Size:  100 . Error for digit 2 :  0.25725338491295935
Size:  100 . Error for digit 3 :  0.3098591549295775
Size:  100 . Error for digit 4 :  0.20606060606060606
Size:  100 . Error for digit 5 :  0.4414019715224534
Size:  100 . Error for digit 6 :  0.26534653465346536
Size:  100 . Error for digit 7 :  0.30028873917228105
Size:  100 . Error for digit 8 :  0.38677354709418843
Size:  100 . Error for digit 9 :  0.193304535637149

Size:  200 . Error for digit 0 :  0.09684210526315784
Size:  200 . Error for digit 1 :  0.03664921465968585
Size:  200 . Error for digit 2 :  0.2514506769825918
Size:  200 . Error for digit 3 :  0.19718309859154926
Size:  200 . Error for digit 4 :  0.20707070707070707
Size:  200 . Error for digit 5 :  0.47754654983570644
Size:  200 . Error for digit 6 :  0.10198019801980196
Size:  200 . Error for digit 7 :  0.246390760346487
Size:  200 . Error for digit 8 :  0.3006012024048096
Size:  200 . Error for digit 9 :  0.3855291576673866

Size:  500 . Error for digit 0 :  0.07473684210526321
Size:  500 . Error for digit 1 :  0.10820244328097728
Size:  500 . Error for digit 2 :  0.19052224371373305
Size:  500 . Error for digit 3 :  0.2665995975855131
Size:  500 . Error for digit 4 :  0.16969696969696968
Size:  500 . Error for digit 5 :  0.3001095290251917
Size:  500 . Error for digit 6 :  0.1316831683168317
Size:  500 . Error for digit 7 :  0.1761308950914341
Size:  500 . Error for digit 8 :  0.1643286573146293
Size:  500 . Error for digit 9 :  0.2634989200863931

Size:  1000 . Error for digit 0 :  0.0831578947368421
Size:  1000 . Error for digit 1 :  0.0732984293193717
Size:  1000 . Error for digit 2 :  0.2620889748549323
Size:  1000 . Error for digit 3 :  0.16599597585513082
Size:  1000 . Error for digit 4 :  0.18181818181818177
Size:  1000 . Error for digit 5 :  0.38006571741511497
Size:  1000 . Error for digit 6 :  0.098019801980198
Size:  1000 . Error for digit 7 :  0.1434071222329163
Size:  1000 . Error for digit 8 :  0.26252505010020044
Size:  1000 . Error for digit 9 :  0.2267818574514039

Size:  2000 . Error for digit 0 :  0.10421052631578942
Size:  2000 . Error for digit 1 :  0.06719022687609078
Size:  2000 . Error for digit 2 :  0.19439071566731136
Size:  2000 . Error for digit 3 :  0.17303822937625757
Size:  2000 . Error for digit 4 :  0.203030303030303
Size:  2000 . Error for digit 5 :  0.33296823658269437
Size:  2000 . Error for digit 6 :  0.12178217821782178
Size:  2000 . Error for digit 7 :  0.1357074109720885
Size:  2000 . Error for digit 8 :  0.19739478957915835
Size:  2000 . Error for digit 9 :  0.22246220302375808

Size:  5000 . Error for digit 0 :  0.09578947368421054
Size:  5000 . Error for digit 1 :  0.07155322862129143
Size:  5000 . Error for digit 2 :  0.19535783365570603
Size:  5000 . Error for digit 3 :  0.18008048289738432
```

```
Size:  5000 . Error for digit 4 :  0.16161616161616166
Size:  5000 . Error for digit 5 :  0.34392113910186195
Size:  5000 . Error for digit 6 :  0.10099009900990097
Size:  5000 . Error for digit 7 :  0.14533205004812322
Size:  5000 . Error for digit 8 :  0.20340681362725455
Size:  5000 . Error for digit 9 :  0.22462203023758098
```

In [47]:
```
for i in range(10):
    plt.plot([100, 200, 500, 1000, 2000, 5000], np.array(error_rate2).T[i],
plt.legend(loc="upper right")
plt.xlabel("Training Points")
plt.ylabel("Fraction of incorrect points")
plt.title("LDA Error Rate vs Model Training Size w/ Digits")
```

Out[47]:
```
Text(0.5, 1.0, 'LDA Error Rate vs Model Training Size w/ Digits')
```



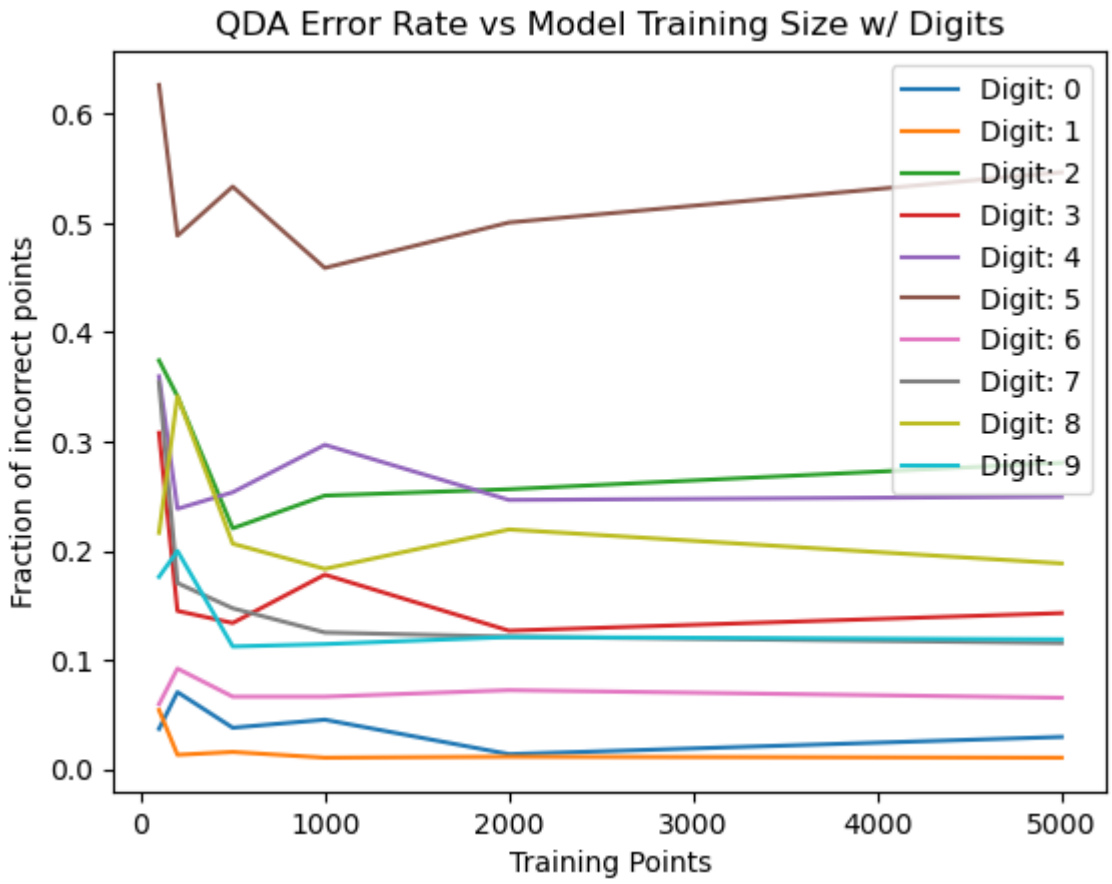
In [48]:
```
error_rate3 = []
for sample_pts in [100, 200, 500, 1000, 2000, 5000]:
    cov, mean, lmaoDontCare = create_parameters_qda(X_train, X_labels, sampl
    predictions = predict(mean, cov, y_train, "qda")
    err2 = []
    for i in range(10):
        error = 1 - sklearn.metrics.accuracy_score(predictions[y_labels == i
        err2.append(error)
        print("Size: ", str(sample_pts), ". Error for digit", str(i), ": ",
    error_rate3.append(np.array(err2))
    print('')
```

```
Size:  100 . Error for digit 0 :  0.0368421052631579
Size:  100 . Error for digit 1 :  0.054101221640488695
Size:  100 . Error for digit 2 :  0.37427466150870403
Size:  100 . Error for digit 3 :  0.30784708249496984
Size:  100 . Error for digit 4 :  0.3595959595959596
Size:  100 . Error for digit 5 :  0.6265060240963856
Size:  100 . Error for digit 6 :  0.05940594059405946
Size:  100 . Error for digit 7 :  0.3541867179980751
Size:  100 . Error for digit 8 :  0.21643286573146292
Size:  100 . Error for digit 9 :  0.17602591792656586

Size:  200 . Error for digit 0 :  0.07052631578947366
Size:  200 . Error for digit 1 :  0.01308900523560208
Size:  200 . Error for digit 2 :  0.3413926499032882
Size:  200 . Error for digit 3 :  0.14486921529175045
Size:  200 . Error for digit 4 :  0.23838383838383836
Size:  200 . Error for digit 5 :  0.488499452354874
Size:  200 . Error for digit 6 :  0.09207920792079205
Size:  200 . Error for digit 7 :  0.17035611164581332
Size:  200 . Error for digit 8 :  0.3416833667334669
Size:  200 . Error for digit 9 :  0.19978401727861772

Size:  500 . Error for digit 0 :  0.037894736842105314
Size:  500 . Error for digit 1 :  0.015706806282722474
Size:  500 . Error for digit 2 :  0.22050290135396522
Size:  500 . Error for digit 3 :  0.1338028169014085
Size:  500 . Error for digit 4 :  0.2535353535353535
Size:  500 . Error for digit 5 :  0.5334063526834611
Size:  500 . Error for digit 6 :  0.06633663366336628
Size:  500 . Error for digit 7 :  0.14725697786333014
Size:  500 . Error for digit 8 :  0.2064128256513026
Size:  500 . Error for digit 9 :  0.11231101511879049

Size:  1000 . Error for digit 0 :  0.045263157894736894
Size:  1000 . Error for digit 1 :  0.010471204188481686
Size:  1000 . Error for digit 2 :  0.2504835589941973
Size:  1000 . Error for digit 3 :  0.17806841046277666
Size:  1000 . Error for digit 4 :  0.296969696969697
Size:  1000 . Error for digit 5 :  0.45892661555312153
Size:  1000 . Error for digit 6 :  0.06633663366336628
Size:  1000 . Error for digit 7 :  0.12512030798845042
Size:  1000 . Error for digit 8 :  0.18336673346693388
Size:  1000 . Error for digit 9 :  0.1144708423326134

Size:  2000 . Error for digit 0 :  0.013684210526315743
Size:  2000 . Error for digit 1 :  0.011343804537521818
Size:  2000 . Error for digit 2 :  0.2562862669245648
Size:  2000 . Error for digit 3 :  0.12676056338028174
Size:  2000 . Error for digit 4 :  0.2464646464646465
Size:  2000 . Error for digit 5 :  0.5005476451259584
Size:  2000 . Error for digit 6 :  0.07227722772277223
Size:  2000 . Error for digit 7 :  0.12127045235803657
Size:  2000 . Error for digit 8 :  0.21943887775551107
Size:  2000 . Error for digit 9 :  0.12095032397408212

Size:  5000 . Error for digit 0 :  0.02947368421052632
Size:  5000 . Error for digit 1 :  0.010471204188481686
Size:  5000 . Error for digit 2 :  0.28046421663442944
Size:  5000 . Error for digit 3 :  0.1428571428571429
```

```
Size:  5000 . Error for digit 4 :  0.24949494949494955
Size:  5000 . Error for digit 5 :  0.5465498357064622
Size:  5000 . Error for digit 6 :  0.0653465346534653
Size:  5000 . Error for digit 7 :  0.1154956689124158
Size:  5000 . Error for digit 8 :  0.188376753507014
Size:  5000 . Error for digit 9 :  0.11879049676025921
```

In [49]:
```
for i in range(10):
    plt.plot([100, 200, 500, 1000, 2000, 5000], np.array(error_rate3).T[i],
plt.legend(loc="upper right")
plt.xlabel("Training Points")
plt.ylabel("Fraction of incorrect points")
plt.title("QDA Error Rate vs Model Training Size w/ Digits")
```

Out[49]:
```
Text(0.5, 1.0, 'QDA Error Rate vs Model Training Size w/ Digits')
```



Part 9

IT IS FINALLY TIME TO CREATE OUR REAL PREDICTIONS!! Because we already generated covariance matrices for each digit using ALL data points in **Part 2**, let us use those matrices and mean vectors to generate our predictions for the "digits" dataset.

In [72]:
```
# Preprocessing step
all_means_reshaped = [np.concatenate(all_means[i]) for i in range(len(all_me
all_cov_reshaped = []
for i in range(10):
    all_cov_reshaped.append(np.add(np.mean(all_cov[i], axis=0), .1 * np.ider
```

In [73]:
```
np.array(all_means_reshaped).shape
```

Out[73]:
```
(10, 784)
```

In [75]:
```
#all_means, all_cov
predictions = predict(all_means_reshaped, all_cov_reshaped, mnist_test, "qda
```

In [78]:
```
import pandas as pd
```

In [79]:
```
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')
```

In [80]:
```
results_to_csv(predictions)
```

Part 10

Now let us create an entirely new model to classify spam emails! While QDA is not exactly the best fit for this task, I am curious to see how it will do for a binary classification rather than a multi-class one. We will use the same QDA process as before, using indicators as our input vectors. By "indicators", I mean that we are going to check if certain words are in email i, and if they are, we will input a 1 at v_i. Otherwise, $v_i = 0$. While this may result in sparse vectors, and hence a lower test accuracy, I am going to do it anyways because I want to see what happens. Sorry!

In [83]:
```
# spam_train, spam_train_labels, spam_test

all_spam_means = []
all_spam_cov = []

for i in range(len(np.unique(spam_train_labels))):
    all_spam_classes = spam_train[spam_train_labels == i]
    mean_spam = np.mean(all_spam_classes, axis=0)
    all_spam_means.append(mean_spam)

    cov_array_spam = []
    for num in all_spam_classes:
        cov = np.outer((num-mean_spam), (num-mean_spam))
        cov_array_spam.append(cov)

    all_spam_cov.append(np.mean(cov_array_spam, axis=0))
```

In [87]:
```
np.array(all_spam_cov).shape
```

Out[87]:
```
(2, 32, 32)
```

In [89]:
```
def predict_final(mean, covariance, points):
    results = []
    for i in range(2):
        log_p = scipy.stats.multivariate_normal.logpdf(points, mean=mean[i],
        results.append(log_p)
    predictions = np.argmax(results, axis=0)
    return predictions
```

In [90]:
```
spam_predictions = predict_final(all_spam_means, all_spam_cov, spam_test)
```

In [96]:
```
print(spam_predictions[:100])
```

```
[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 0 0 0 0 0 0 0 1 0 0 0
 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0]
```

In [97]:
```
results_to_csv(spam_predictions)
```

Part 11

Kaggle Results:

Spam Data Set - 75.8% Test Accuracy

Digits Data Set - 81.6% Test Accuracy

Actually, not that bad! I was expecting a lot worse if I'm being honest. However, I'm just glad this works at all. If I had to do this project again, I would first start off with some EDA to examine useful features and quirks of these datasets. Then I would implement the model using a Object-Oriented approach, similar to how PyTorch works. Using pure functions made this project harder than it should've been, but I did just implement these statistical models using only math, numpy, and other lightweight libraries. Also, I'm never training a ML model on my laptop. Ever. Again.