LDA/QDA Classification of MNIST

In this problem, we will apply LDA and QDA to classify a real dataset: MNIST, which is a set of 28x28 grayscale images of handwritten digits.

Let's first load the dataset and look at some images. Notice that we truncate the dataset so that our computation will run faster.

In [3]:
```
! pip install mnist
import mnist
import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interact, interactive, fixed, interact_manual, IntSlider
from collections import defaultdict
```

```
Defaulting to user installation because normal site-packages is not writeable
Collecting mnist
  Downloading mnist-0.2.2-py2.py3-none-any.whl (3.5 kB)
Requirement already satisfied: numpy in /home/mjayasur/.local/lib/python3.8/site
-packages (from mnist) (1.19.3)
Installing collected packages: mnist
Successfully installed mnist-0.2.2
```

In [4]:
```
training_data = mnist.train_images()[:10000]
training_labels = mnist.train_labels()[:10000]
test_data = mnist.test_images()[:1000]
test_labels = mnist.test_labels()[:1000]
```

Look at some random images from our training set.

In [5]:
```
interact(lambda i: plt.imshow(training_data[i], cmap='gray'), i=IntSlider(min=0,
```

Out[5]:
```
<function __main__.<lambda>(i)>
```

Let's first try to implement QDA. Recall that discriminative methods have two steps: first, we want to construct a model based on our training data - then, we want to use this model to make predictions.

For simplicity, assume that all the digits are equally probable. Looking at our training data, this seems to be a valid assumption:

In [6]:
```
cnts = defaultdict(int)
for label in training_labels:
    cnts[label] += 1
print(cnts)
```

```
defaultdict(<class 'int'>, {5: 863, 0: 1001, 4: 980, 1: 1127, 9: 978, 2: 991, 3:
1032, 6: 1014, 7: 1070, 8: 944})
```

For QDA, we assume our true distribution is of the form

$$\Pr(\boldsymbol{X}|Y=k) = N(\mu_k, \Sigma_k)$$

for each class k - in our case, $k \in \{0, 1, \ldots, 9\}$. To find estimates $\hat{\mu}_k$ and $\hat{\Sigma}_k$, we use the MLE estimates.

We have seen that they are what you would intuitively expect:

$$\hat{\mu}_k = \frac{1}{n_k} \sum_{i|y_i=k} x_i,$$

where n_k is the number of points in class k, and

$$\hat{\Sigma}_k = \frac{1}{n_k} \sum_{i|y_i=k} (x_i - \hat{\mu}_k)(x_i - \hat{\mu}_k)^T.$$

Part (a)

Fill in the below functions to compute these estimates.

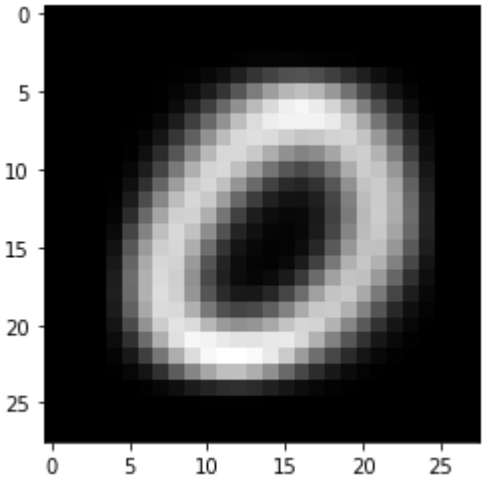
In [7]:
```
def compute_qda_params_single_class(data):
    """
    Computes the mean and MLE covariance for the input data points from a single
    training_data is an n * d matrix, where each row is a separate data point.
    Returns a tuple (mu, sigma) with the desired output.
    """
    n, d = data.shape
    ### start qda_single_class ###
    mu = np.mean(data, 0)

    s = np.outer(data[0] - mu, data[0] - mu)

    for i in range(1, n):
        s += np.outer(data[i] - mu, data[i] - mu)

    sigma = 1/n * s
    ### end qda_single_class ###

    assert sigma.shape == (d, d), "Sigma is not the right shape"
    return mu, sigma


def compute_qda_params(data, labels):
    """
    Computes the mean and MLE covariance for each class individually, given
    labeled input data. Returns a list [(mu_0, sigma_0), (mu_1, sigma_1), ...]
    with one entry for each class
    """
    num_classes = 10
    assert len(set(labels)) == num_classes

    # "flatten" each sample point from a 2D array into a single row
    data = data.reshape((data.shape[0], -1))
    n, d = data.shape
    params = []
    ### start qda_multi_class ###
    classes = [[] for i in range(num_classes)]
    for datum, label in zip(data, labels):
        classes[label].append(datum)
    for _class in classes:
        params.append(compute_qda_params_single_class(np.array(_class)))
    ### end qda_multi_class ###
```

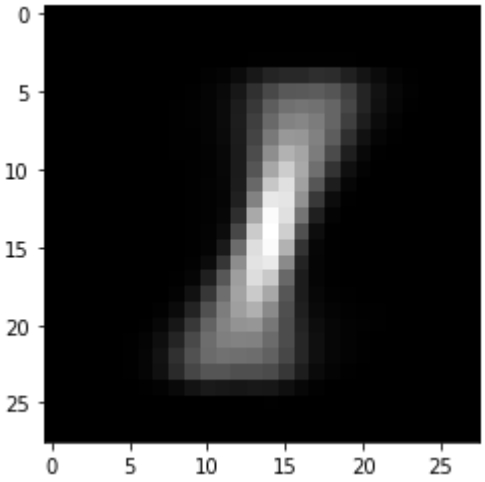
```
    assert len(params) == num_classes
    return params
```

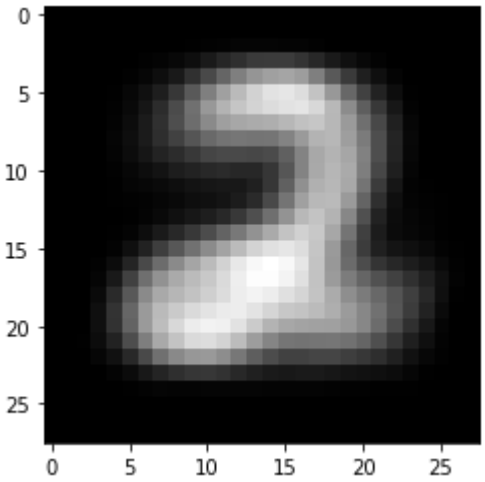
In [8]:
```
qda_params = compute_qda_params(training_data, training_labels)
```

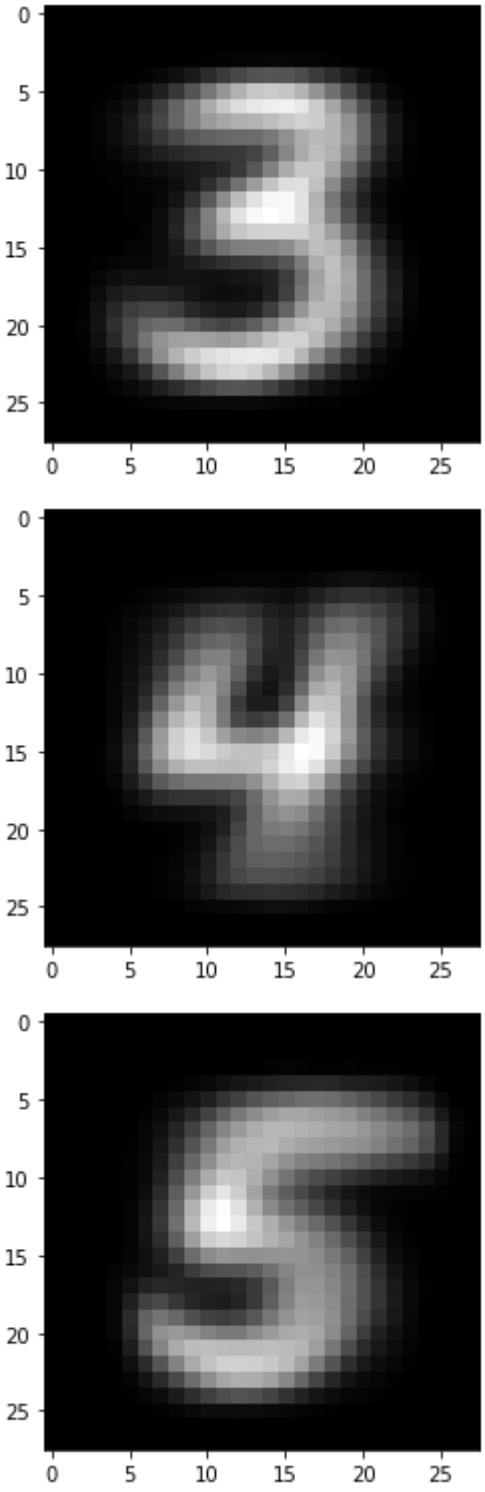
Let's try to visualize these parameters. It is reasonable to expect that plotting the means of each class will produce a sort of "representative" image for each digit.

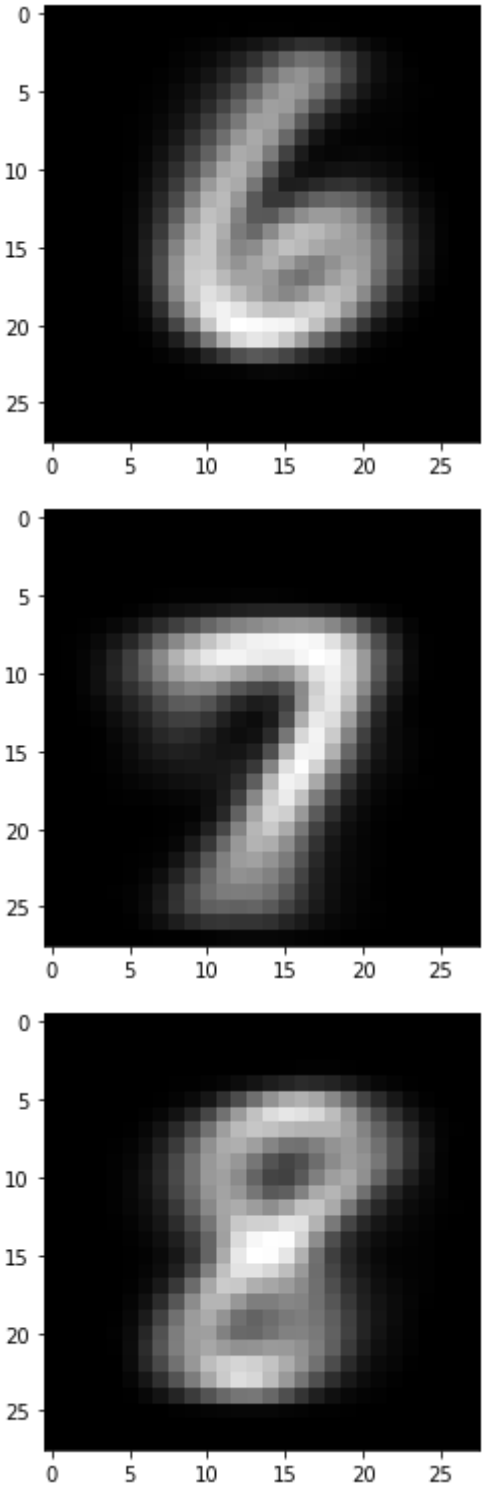
In [9]:
```
for mu, sigma in qda_params:
    plt.imshow(mu.reshape(28, 28), cmap='gray')
    plt.show()
```

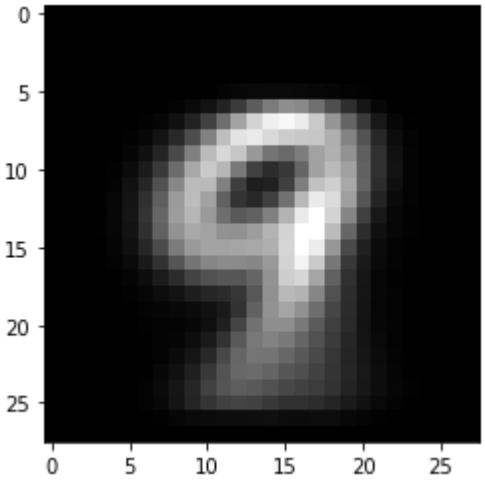









0
5
10
15
20
25
0
5
10
15
20
25
0
5
10
15
20
25
0
5
10
15
20
25
0
5
10
15
20
25
0
5
10
15
20
25



You should obtain images that look like digits. If not, you've probably made a mistake. Next, we'll try to classify some of our test data and see how well our approach works.

Part (b)

Implement the below function, that takes in the `params` you computed in the previous part, as well as a test point, and tries to determine its label. Recall that

$$\hat{y} = \arg\max_k \left(-\frac{1}{2}(x - \hat{\mu}_k)^T \hat{\Sigma}_k^{-1} (x - \hat{\mu}_k) - \frac{1}{2} \ln\left(|\hat{\Sigma}_k| \right) \right)$$

Your covariance matrices may be singular. This will be a problem, since you need to invert them and compute the log of their determinant, both of which are undefined operations on singular matrices. There are a number of ways to get around this - here, we will simply implement a "hack", by "fuzzing" the matrix in a manner that preserves its singular values, but lifts its zero singular values up to a small nonzero value. Specifically, let $\text{fuzz}(\Sigma) = \Sigma + \varepsilon I$, where ε is some small hyperparameter.

We have implemented this `fuzz` function for you with $\varepsilon = 10^{-6}$, as well as a helper function `compute_accuracy` to help you evaluate your results.

In [10]:
```
def compute_accuracy(predicted_labels, test_labels):
    predicted_labels = predicted_labels.reshape(-1)
    test_labels = test_labels.reshape(-1)
    n = len(predicted_labels)
    assert n == len(test_labels)
    return np.sum(predicted_labels == test_labels) / n

eps = 1e-6

def fuzz(matrix):
    try:
        np.linalg.inv(matrix)
    except np.linalg.LinAlgError:
        return matrix + np.eye(len(matrix)) * eps
    else:
        return matrix

def classify(params, test_points):
```

```
    """
    `params` is as generated by `compute_qda_params`
    `test_points` is an array of test points, with one row for each point.

    You should try to vectorize as much of your solution as possible, so it does
    not take too long to run.
    """
    # reshape test_points so each test point is in a single row
    test_points = test_points.reshape(test_points.shape[0], -1)
    n, d = test_points.shape
    labels = []
    ### start classify ###

    dets = [np.linalg.slogdet(fuzz(param[1])) for param in params]
    dets = [np.log(det[0] * abs(det[1])) for det in dets]
    invs = [np.linalg.inv(fuzz(param[1])) for param in params]
    for x in test_points:
        max_p = float('-inf')
        max_label = 0
        for k in range(10):
            mu = params[k][0]
            sigma =  params[k][1]
            p = -.5 * ((x - mu).T@invs[k]@(x - mu)  + dets[k])
            if p >= max_p:
                max_p = p
                max_label = k

        labels.append(max_label)

    labels = np.array(labels)
    ### end classify ###
    labels = labels.reshape(-1)
    assert len(labels) == n, "{} != {}".format(len(labels), n)
    return labels
```

In [ ]:

Let's see how well your classifier does on MNIST!

In [11]:
```
print("Train Accuracy:", compute_accuracy(classify(qda_params, training_data), t
print("Test Accuracy:", compute_accuracy(classify(qda_params, test_data), test_l
```

```
Train Accuracy: 0.9797
Test Accuracy: 0.789
```

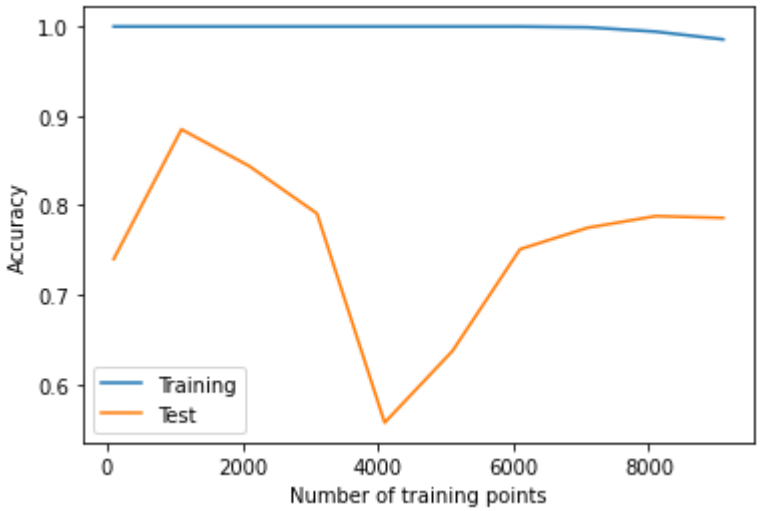
Part (c)

Another interesting thing to plot is the train and test accuracy as a function of the number of data points. This will tell us whether we are overfitting to the training set. Fill in the code blocks to generate the plot.

In [12]:
```
def accuracy_vs_n(compute_params):
    candidate_ns = range(100, 10000, 1000)
    training_accuracies = []
    test_accuracies = []
    for n in candidate_ns:
        n = int(n)
        print("Evaluating with {} samples".format(n))
        ### start compute_qda_accuracies ###
```

```
        params = compute_params(training_data[:n], training_labels[:n])
        train_acc = compute_accuracy(classify(params, training_data[:n]), traini
        test_acc = compute_accuracy(classify(params, test_data[:n]), test_labels
        training_accuracies.append(train_acc)
        test_accuracies.append(test_acc)
        ### end compute_qda_accuracies ###
    plt.plot(candidate_ns, training_accuracies, label="Training")
    plt.plot(candidate_ns, test_accuracies, label="Test")
    plt.ylabel("Accuracy")
    plt.xlabel("Number of training points")
    plt.legend()
    plt.show()

accuracy_vs_n(compute_qda_params)
```

```
Evaluating with 100 samples
Evaluating with 1100 samples
Evaluating with 2100 samples
Evaluating with 3100 samples
Evaluating with 4100 samples
Evaluating with 5100 samples
Evaluating with 6100 samples
Evaluating with 7100 samples
Evaluating with 8100 samples
Evaluating with 9100 samples
```



Comment on these results. You should notice that the test accuracy fluctuates quite dramatically. One reason for this is because our estimates of the covariances for each label are very noisy, since we are only looking at one tenth of the data when calculating each covariance matrix.

start qda-comment

Test accuracy is completely erratic, and the train accuracy actually decreases very slightly as more and more training examples are shown. The train accuracy being at basically 100% makes sense because modeling the individual distributions' covariance matrices will mean that the model will fit the noise well and work well on the training data, but not the test data.

end qda-comment

Part (d)

LDA solves this problem, by making the assumption that the covariance matrices for each class are the same $\Sigma = \Sigma_k$ for all k. We compute our estimate $\hat{\Sigma}$ by simply averaging the covariance matrices for each class. In principle, this estimate should be less noisy since it is constructed using much more data, so our test accuracy should not vary by as much.

Let's find out if this is the case. Implement the below function to compute the `params` to be passed into `classify` for LDA. You should reuse the `compute_qda_params` function that you previously implemented.

In [13]:
```
def compute_lda_params(data, labels):
    """
    Computes the mean for each class individually and the MLE covariance across
    given labeled input data. Returns a list [(mu_0, sigma), (mu_1, sigma), ...]
    with one entry for each class
    """
    n, d = data.shape[0], data.shape[1] * data.shape[2]
    ### start lda_multi_class ###
    params = compute_qda_params(data, labels)
    sigma = np.mean([param[1] for param in params], 0)
    params = [(param[0], sigma) for param in params]
    ### end lda_multi_class ###
    assert all((sigma == params[0][1]).all() for mu, sigma in params), "Covariar
    return params
```

We can now evaluate the LDA classifier using the `classify` function from earlier.

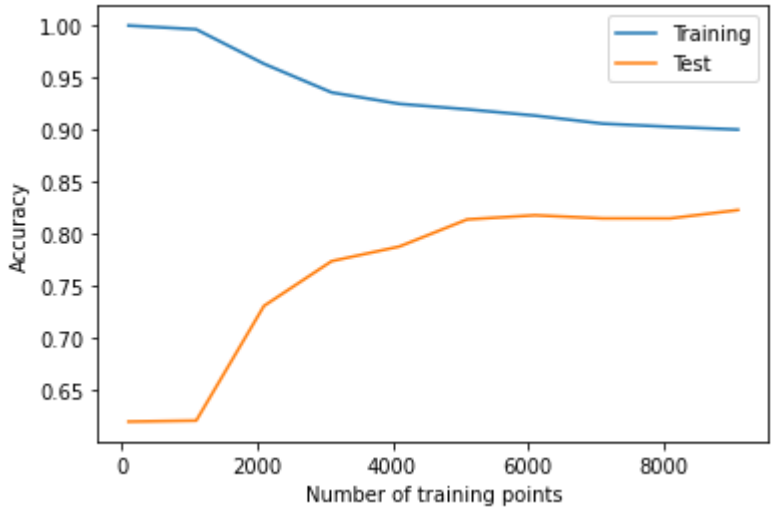
In [14]:
```
lda_params = compute_lda_params(training_data, training_labels)
print("Train Accuracy:", compute_accuracy(classify(lda_params, training_data), t
print("Test Accuracy:", compute_accuracy(classify(lda_params, test_data), test_l
```

```
Train Accuracy: 0.8984
Test Accuracy: 0.826
```

And finally, we can run the LDA classifier with a variable number of input samples:

In [15]:
```
accuracy_vs_n(compute_lda_params)
```

```
Evaluating with 100 samples
Evaluating with 1100 samples
Evaluating with 2100 samples
Evaluating with 3100 samples
Evaluating with 4100 samples
Evaluating with 5100 samples
Evaluating with 6100 samples
Evaluating with 7100 samples
Evaluating with 8100 samples
Evaluating with 9100 samples
```



Comment on your observations in both the LDA and QDA cases. Did you achieve the expected reduction in variation with LDA? What about in absolute terms - is it a better method than QDA? Why, or why not?

start lda-comment

There is definitely a reduction in noise, and in this example it is a better method than QDA, as it has higher training accuracy and also is predictable in its behavior with increasing training data. The training accuracy decreases more in this, which makes sense because it is taking the mean of the covariance matrices so the model is not adjusted to the noise of a particular class

end lda-comment

Part (e)

One thing we can investigate further is the per-digit accuracy of our two classifiers. It is reasonable to suppose that they might be much better at classifying some digits compared to others. Run the below function to plot their per-digit training and test accuracies. It will create two plots: one with the training accuracy for each digit as a function of the number of samples, and another with the test accuracy.

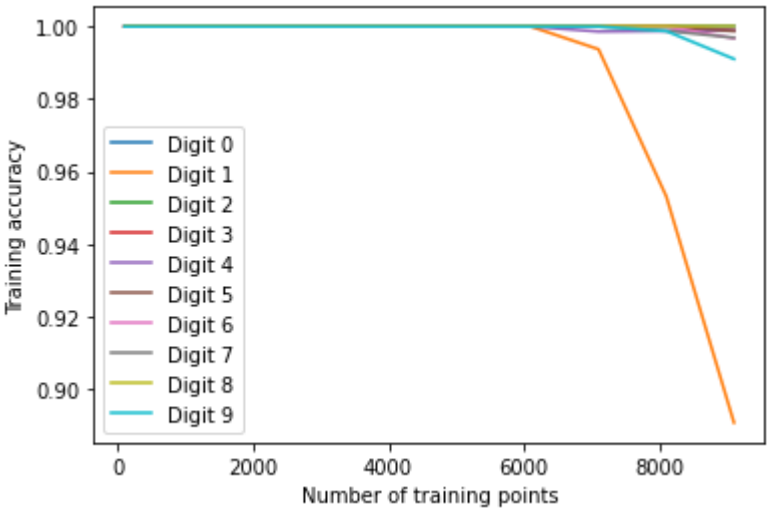
This may take a while to run!

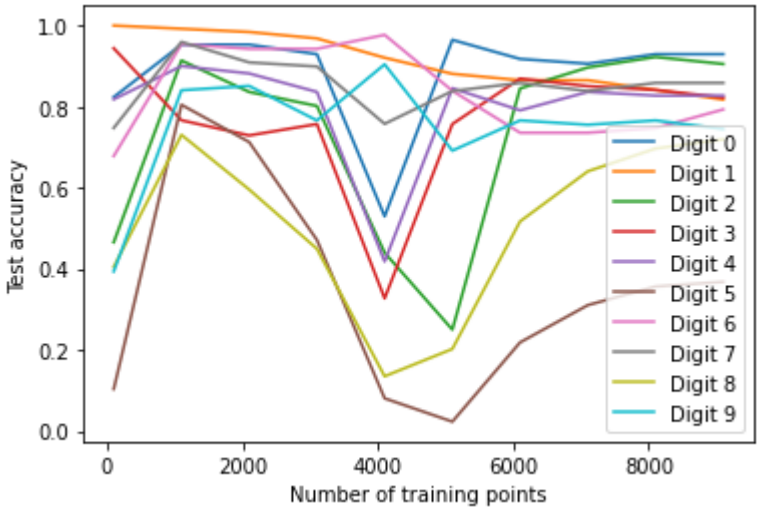
In [21]:
```
def digit_accuracy_vs_n(compute_params):
    candidate_ns = range(100, 10000, 1000)
    training_accuracies_by_digit = [[] for _ in range(10)]
    test_accuracies_by_digit = [[] for _ in range(10)]
    for n in candidate_ns:
        n = int(n)
        print("Evaluating with {} samples".format(n))
        params = compute_params(training_data[:n], training_labels[:n])
        predicted_training_labels = classify(params, training_data[:n])
        predicted_test_labels = classify(params, test_data)
        for k in range(10):
            training_indices_with_k = training_labels[:n] == k
            test_indices_with_k = test_labels == k
            training_accuracies_by_digit[k].append(compute_accuracy(
```

```
                predicted_training_labels[training_indices_with_k], training_lab
            ))
            test_accuracies_by_digit[k].append(compute_accuracy(
                predicted_test_labels[test_indices_with_k], test_labels[test_inc
            ))

    for k, training_accuracies in enumerate(training_accuracies_by_digit):
        plt.plot(candidate_ns, training_accuracies, label="Digit {}".format(k))
    plt.xlabel("Number of training points")
    plt.ylabel("Training accuracy")
    plt.legend()
    plt.show()

    for k, test_accuracies in enumerate(test_accuracies_by_digit):
        plt.plot(candidate_ns, test_accuracies, label="Digit {}".format(k))
    plt.xlabel("Number of training points")
    plt.ylabel("Test accuracy")
    plt.legend()
    plt.show()
```

In [17]:
```
# For QDA
digit_accuracy_vs_n(compute_qda_params)
```

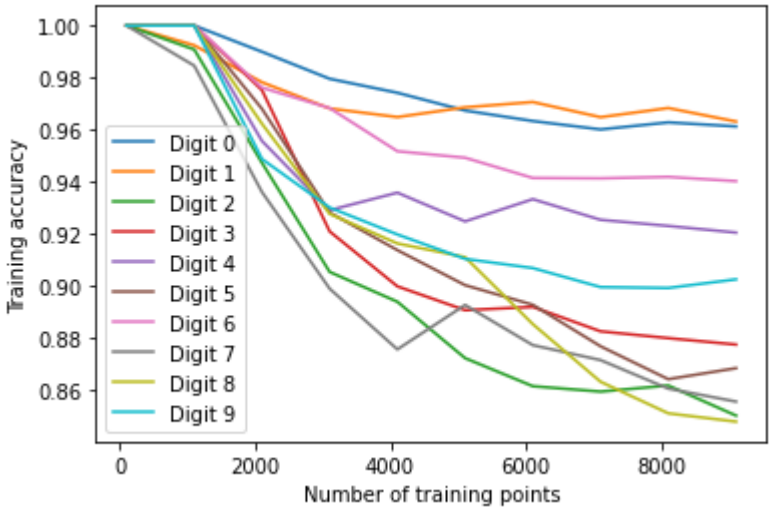
```
Evaluating with 100 samples
Evaluating with 1100 samples
Evaluating with 2100 samples
Evaluating with 3100 samples
Evaluating with 4100 samples
Evaluating with 5100 samples
Evaluating with 6100 samples
Evaluating with 7100 samples
Evaluating with 8100 samples
Evaluating with 9100 samples
```

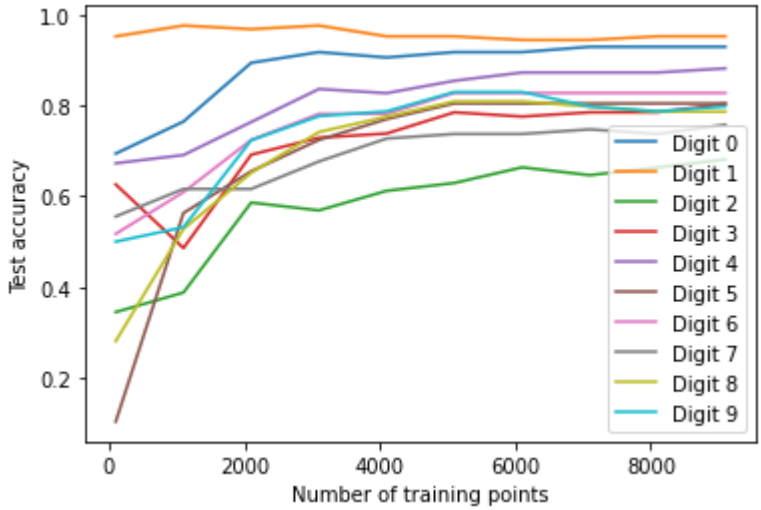




In [18]:
```
# For LDA
digit_accuracy_vs_n(compute_lda_params)
```

```
Evaluating with 100 samples
Evaluating with 1100 samples
Evaluating with 2100 samples
Evaluating with 3100 samples
Evaluating with 4100 samples
Evaluating with 5100 samples
Evaluating with 6100 samples
Evaluating with 7100 samples
Evaluating with 8100 samples
Evaluating with 9100 samples
```





Comment on your results. Which digits were easy to classify? Which were harder? Why might this be the case? Did it differ between LDA and QDA?

start digit-comment

Digits 0 and 1 were the easiest to classify, which makes sense as they're pretty simple and most 0's and 1's are pretty close to the mean of the training data. Digit 2 and 5 were harder to classify, I'd imagine because they look very similar to eachother and there is more variation in how 2's are written (maybe a fancy 2 haha).

end digit-comment

Part (f)

Finally, we will return to our discussion of singular covariance matrices. Recall that the `fuzz` function perturbed our covariance matrix to lift the zero eigenvalues up to some small `eps`, a hyperparameter that we arbitrarily set to 10^{-6}. Now, you should experiment with different choices for `eps`, to attempt to choose the optimal hyperparameter.

For simplicity, we will only consider tuning `eps` to optimize the performance of LDA, since the accuracy of QDA with this few data points is too noisy to obtain reliable results.

Implement the `tune_eps` and `compute_validation_score` functions. Consider what would be an appropriate range of hyperparameters, as well as how the test values will be spaced over that interval.

In [19]:
```
def tune_eps(validation_score):
    """
    validation_score() is a zero-argument function that returns
    the validation accuracy of some model. `eps` is passed into it
    by modifying it as a global variable.

    You should try at least 10 different values of `eps`.
    """
    global eps
```

```
    best_eps = None
    best_validation = 0

    eps_vals = [10**(-i) for i in range(10)]
    scores = []
    ### start tune_eps ###
    for val in eps_vals:
        eps = val
        score = validation_score()
        if score > best_validation:
            best_validation = score
            best_eps = val
        scores.append(score)
    ### end tune_eps ###
    print ("Validation Scores: ", scores)
    eps = best_eps
    return best_eps

def compute_validation_score(compute_params):
    def validation_score():
        # update these lines
        n_validation = int(0.2 * training_data.shape[0])
        n_train = training_data.shape[0] - n_validation
        hyper_tuning_training_data = training_data[:n_train]
        hyper_tuning_training_labels = training_labels[:n_train]
        hyper_tuning_validation_data = training_data[n_train:]
        hyper_tuning_validation_labels = training_labels[n_train:]

        ### start split_validation ###

        ### end split_validation ###

        return compute_accuracy(classify(
            compute_params(hyper_tuning_training_data, hyper_tuning_training_lab
            hyper_tuning_validation_data,
        ), hyper_tuning_validation_labels)

    return validation_score
```

In [20]:
```
print("Best epsilon for LDA:", tune_eps(compute_validation_score(compute_lda_par
```

```
Validation Scores:  [0.8525, 0.85, 0.8505, 0.85, 0.85, 0.85, 0.85, 0.85, 0.85,
0.85]
Best epsilon for LDA: 1
```

In [22]:
```
print("Best epsilon for QDA:", tune_eps(compute_validation_score(compute_qda_par
```

```
Validation Scores:  [0.8375, 0.8315, 0.8215, 0.8175, 0.8115, 0.8085, 0.809, 0.80
9, 0.809, 0.8085]
Best epsilon for QDA: 1
```

Did your hyperparameter tuning work? Why or why not? Comment on your observations.

start eps-comment

Not really, because all of the validation scores were essentially the same so there is little significant reason to believe the chosen hyperparameter would even be better. A larger hyperparameter might make more sense though to avoid having to have big values in our matrix inverse.

end eps-comment

In [ ]: