James Zhao
A15939512

1 1

10 x 10 image, 1 channel = 10 x 10 x 1 = 100 dimensional vector

2 2

$$\sqrt{(1-3)^2+(2-2)^2+(3-1)^2} = \sqrt{4+0+4} = 2\sqrt{2}$$

3 3

3.1 a

Correctly Classified 25% of the time, **Error Rate** = 75%.

3.2 b

The most frequent label (label A) should be always returned. This results in the smallest error possible, of 50% (error = 1 - frequency).

4 4

4.1 a

The point (0.5, 0.5) lies in the region enclosed by Class 2. Thus, this point has a label of 2.

4.2 b

$$\|(0.5, 0.5) - (1.5, 0.5)\| = \|(-1, 0)\| = 1$$
$$\|(0.5, 1.5) - (1.5, 0.5)\| = \|(-1, 1)\| = \sqrt{2}$$

Label will be the label of the nearest neighbor (0.5, 0.5), or Class 2

4.3 c

$$\|(0.5, 0.5) - (2, 2)\| = \|(1.5, 1.5)\| = \sqrt{1.5^2 + 1.5^2} = \sqrt{4.5}$$
$$\|(0.5, 1.5) - (2, 2)\| = \|(1.5, .5)\| = \sqrt{2.5}$$

Label will be the label of the nearest neighbor (0.5, 1.5), or Class 1

4.4 d

Class 3 will never be predicted because there does not exist any points in the training set that are within the boundary enclosed by Class 3

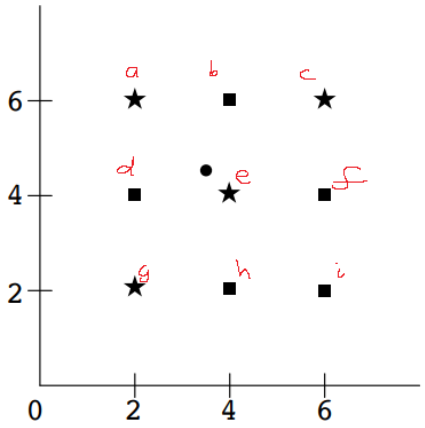
4.5 e

With a training set of just two points, the decision boundary is simply a perpendicular line passing through the midpoint of the two training points. That is, a horizontal line passing through (0.5, 1), or the line y=1. This means that all points in the half-square above y=1 are classified as class 1, and the half-square below y=1 are classified as class 2. All Class 1 and Class 2 test labels are correctly classified, but all Class 3 test points are incorrectly classified (consisting of 50% of the points). Thus, the error rate would be 50%.

5 5

Based on the image, we find the distances to the query point from each point:

$\sqrt{1.5^2 + 1.5^2} = \sqrt{4.5}$	$\sqrt{1.5^2 + 0.5^2} = \sqrt{2.5}$	$\sqrt{1.5^2 + 2.5^2} = \sqrt{8.5}$
$\sqrt{0.5^2 + 1.5^2} = \sqrt{2.5}$	$\sqrt{0.5^2 + 0.5^2} = \sqrt{0.5}$	$\sqrt{0.5^2 + 2.5^2} = \sqrt{6.5}$
$\sqrt{2.5^2 + 1.5^2} = \sqrt{8.5}$	$\sqrt{2.5^2 + 0.5^2} = \sqrt{6.5}$	$\sqrt{2.5^2 + 2.5^2} = \sqrt{12.5}$



Order of closest points: [{e}, {d, b}, {a}, {h, f}, {c, g}, {i}]

5.1 a

1-NN: {(e, star)}. The majority label is **star**.

5.2 b

3-NN: {(e, star), (d, square), (b, square)}. The majority label is **square**.

5.3 c

5-NN: {(e, star), (d, square), (b, square), (a, star), (h/f, square)}. The majority label is **square**. Even though the 5th nearest neighbor has two options, the classification outcome is the same since all possible candidate points have the same label (square).

6 6

Each fold is 10,000/4 = 2,500, but the training sets are (k-1) folds, so each training set is 2500*3=7500 data values.

7 7

Point	**Label**	**Predicted Label**	Correct
Point 1	+	{+}	Yes
Point 2	+	{+}	Yes
Point 3	-	{+}	No
Point 4	+	{-}	No

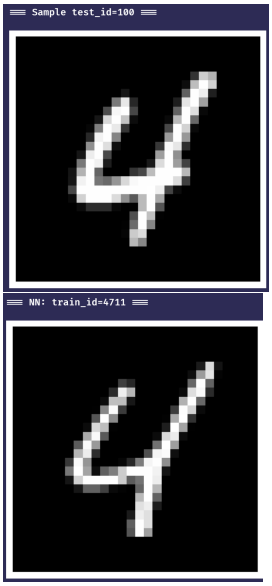
LOOCV error for 1-NN: 50%.

Point	**Label**	**Predicted Label**	Correct
Point 1	+	$\{+,-,+\} \rightarrow +$	Yes
Point 2	+	$\{+,-,+\} \rightarrow +$	Yes
Point 3	-	$\{+,+,+\} \rightarrow +$	No
Point 4	+	$\{-,+,+\} \rightarrow +$	Yes

LOOCV error for 3-NN: 25%

8 8

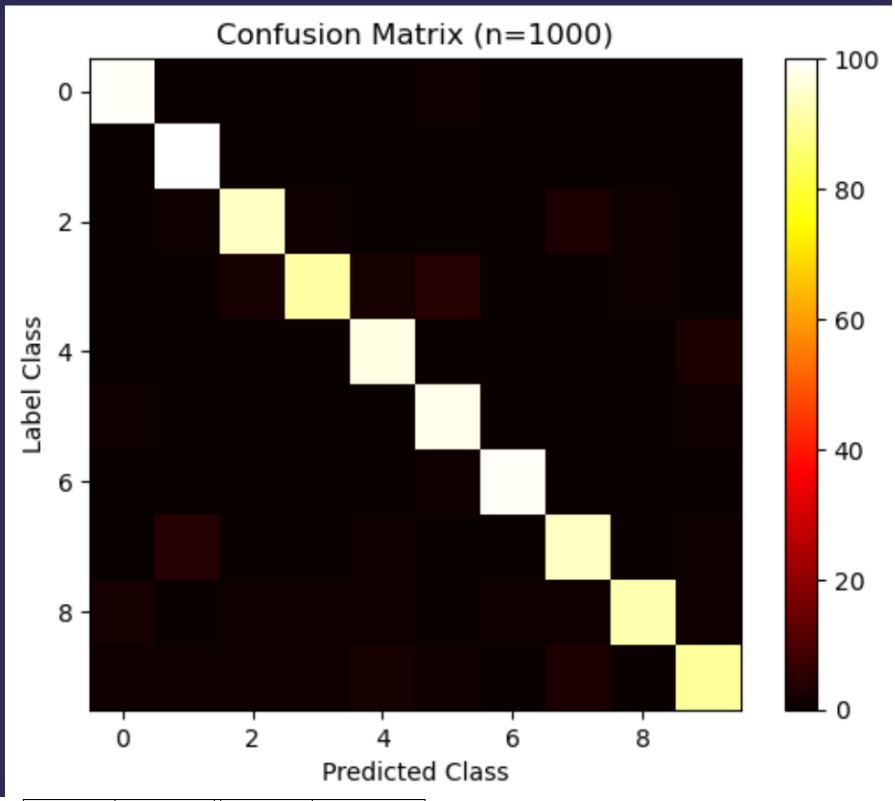
8.1 a



Because both the images have the same label, it is correctly classified.

8.2 b

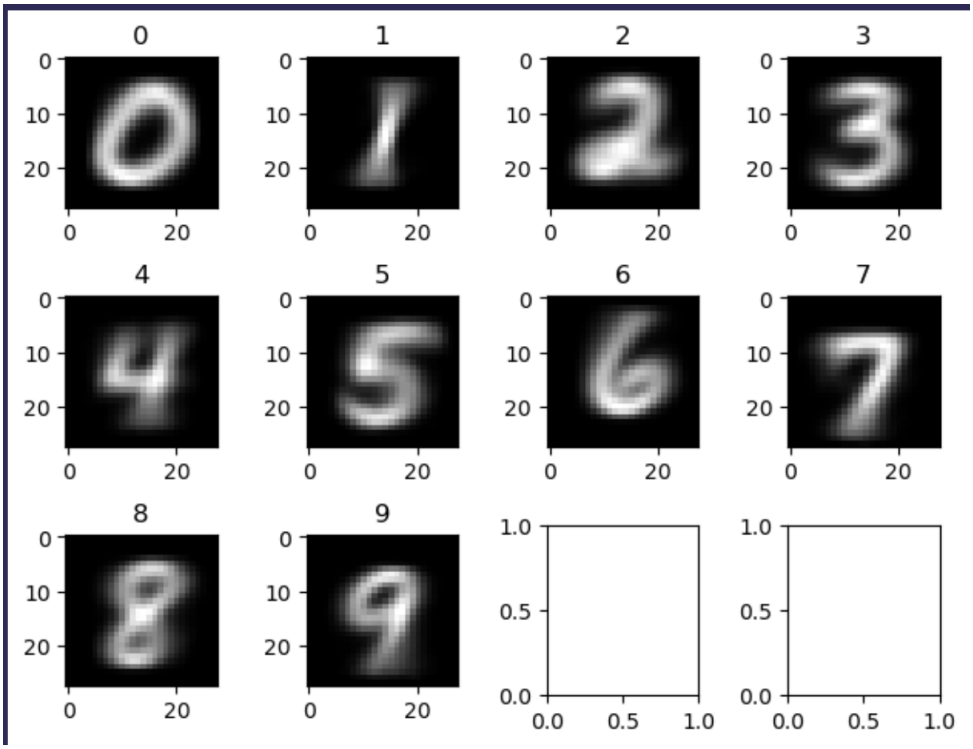
```
[[ 99.   0.   0.   0.   0.   1.   0.   0.   0.   0.]
 [  0. 100.   0.   0.   0.   0.   0.   0.   0.   0.]
 [  0.   1.  94.   1.   0.   0.   0.   3.   1.   0.]
 [  0.   0.   2.  91.   2.   4.   0.   0.   1.   0.]
 [  0.   0.   0.   0.  97.   0.   0.   0.   0.   3.]
 [  1.   0.   0.   0.   0.  98.   0.   0.   0.   1.]
 [  0.   0.   0.   0.   0.   1.  99.   0.   0.   0.]
 [  0.   4.   0.   0.   1.   0.   0.  94.   0.   1.]
 [  2.   0.   1.   1.   1.   0.   1.   1.  92.   1.]
 [  1.   1.   1.   1.   2.   1.   0.   3.   0.  90.]]

<matplotlib.colorbar.Colorbar at 0x2daa16f33a0>
```



Digit	Error	Digit	Error
0	1/100	1	0/100
2	6/100	3	9/100
4	3/100	5	2/100
6	1/100	7	6/100
8	8/100	9	10/100

Digit 9 was the most mis-classified, digit 1 was the least mis-classified.

8.3 c



9 9

9.1 a

L1 Norm Error: 0.2167
L2 Norm Error: 0.2333

9.2 b

L1 Norm:

	NO	DH	SL
NO	14	0	2
DH	9	9	0
SL	1	1	24

L2 Norm:

	NO	DH	SL
NO	12	1	3
DH	9	9	0
SL	1	0	25

Nearest neighbor for handwritten digit recognition

In this notebook we will build a classifier that takes an image of a handwritten digit and outputs a label 0-9. We will look at a particularly simple strategy for this problem known as the **nearest neighbor classifier.**

To run this notebook you should have the following Python packages installed:

- `numpy`
- `matplotlib`
- `sklearn`

1. The MNIST dataset

`MNIST` is a classic dataset in machine learning, consisting of 28x28 gray-scale images handwritten digits. The original training set contains 60,000 examples and the test set contains 10,000 examples. In this notebook we will be working with a subset of this data: a training set of 7,500 examples and a test set of 1,000 examples.

In [18]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import time

## Load the training set
train_data = np.load('../data/MNIST/train_data.npy')
train_labels = np.load('../data/MNIST/train_labels.npy')

## Load the testing set
test_data = np.load('../data/MNIST/test_data.npy')
test_labels = np.load('../data/MNIST/test_labels.npy')
```

In [19]:
```
## Print out their dimensions
print("Training dataset dimensions: ", np.shape(train_data))
print("Number of training labels: ", len(train_labels))
print("Testing dataset dimensions: ", np.shape(test_data))
print("Number of testing labels: ", len(test_labels))
```

```
Training dataset dimensions:  (7500, 784)
Number of training labels:  7500
Testing dataset dimensions:  (1000, 784)
Number of testing labels:  1000
```

In [20]:
```
## Compute the number of examples of each digit
train_digits, train_counts = np.unique(train_labels, return_counts=True)
print("Training set distribution:")
print(dict(zip(train_digits, train_counts)))
```

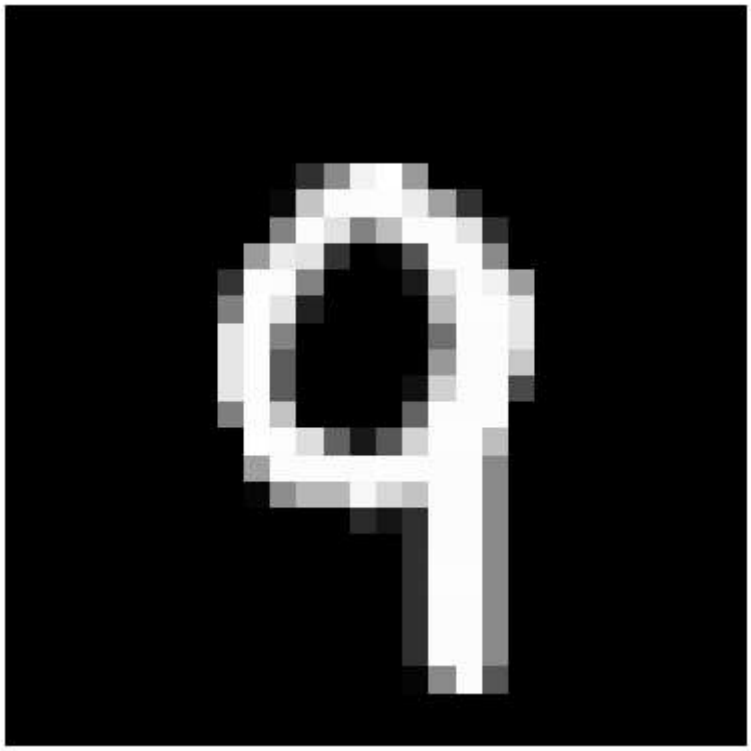
```
test_digits, test_counts = np.unique(test_labels, return_counts=True)
print("Test set distribution:")
print(dict(zip(test_digits, test_counts)))
```

```
Training set distribution:
{0: 750, 1: 750, 2: 750, 3: 750, 4: 750, 5: 750, 6: 750, 7: 750, 8: 750, 9: 750}
Test set distribution:
{0: 100, 1: 100, 2: 100, 3: 100, 4: 100, 5: 100, 6: 100, 7: 100, 8: 100, 9: 100}
```

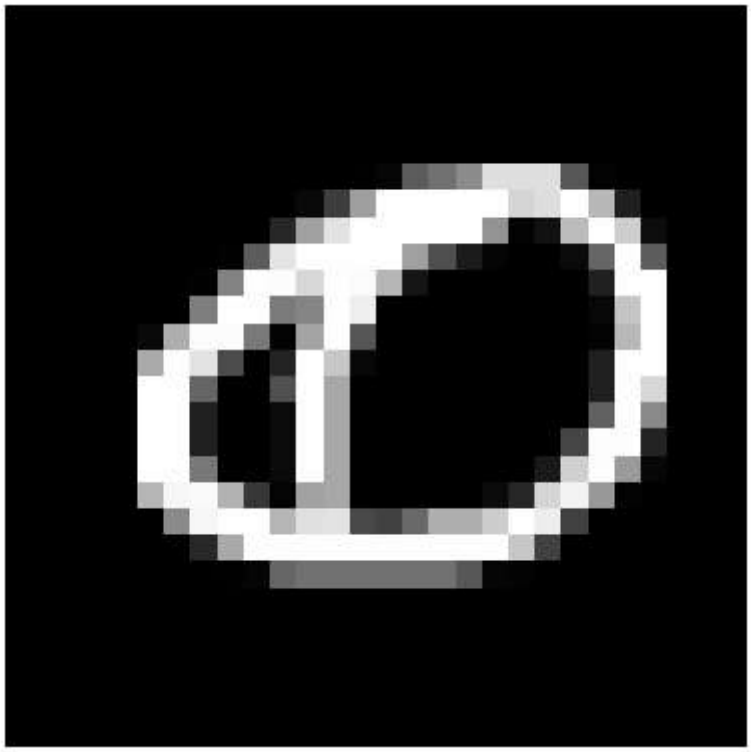
2. Visualizing the data

Each data point is stored as 784-dimensional vector. To visualize a data point, we first reshape it to a 28x28 image.

In [21]:
```
## Define a function that displays a digit given its vector representation
def show_digit(x):
    plt.axis('off')
    plt.imshow(x.reshape((28,28)), cmap=plt.cm.gray)
    plt.show()
    return

## Define a function that takes an index into a particular data set ("train" or "test"
def vis_image(index, dataset="train"):
    if(dataset=="train"):
        show_digit(train_data[index,])
        label = train_labels[index]
    else:
        show_digit(test_data[index,])
        label = test_labels[index]
    print("Label " + str(label))
    return

## View the first data point in the training set
vis_image(0, "train")

## Now view the first data point in the test set
vis_image(0, "test")
```



```
Label 9
```



```
Label 0
```

3. Squared Euclidean distance

To compute nearest neighbors in our data set, we need to first be able to compute distances between data points. A natural distance function is *Euclidean distance*: for two vectors $x, y \in \mathbb{R}^d$, their Euclidean distance is defined as

$$||x - y|| = \sqrt{\sum_{i=1}^{d} (x_i - y_i)^2}.$$

Often we omit the square root, and simply compute *squared Euclidean distance*:

$$||x - y||^2 = \sum_{i=1}^{d} (x_i - y_i)^2.$$

For the purposes of nearest neighbor computations, the two are equivalent: for three vectors $x, y, z \in \mathbb{R}^d$, we have $||x - y|| \leq ||x - z||$ if and only if $||x - y||^2 \leq ||x - z||^2$.

Now we just need to be able to compute squared Euclidean distance. The following function does so.

In [22]:
```
## Computes squared Euclidean distance between two vectors.
def squared_dist(x,y):
    return np.sum(np.square(x-y))

## Compute distance between a seven and a one in our training set.
print("Distance from 7 to 1: ", squared_dist(train_data[4,],train_data[5,]))

## Compute distance between a seven and a two in our training set.
print("Distance from 7 to 2: ", squared_dist(train_data[4,],train_data[1,]))

## Compute distance between two seven's in our training set.
print("Distance from 7 to 7: ", squared_dist(train_data[4,],train_data[7,]))
```

```
Distance from 7 to 1:  5357193.0
Distance from 7 to 2:  12451684.0
Distance from 7 to 7:  5223403.0
```

4. Computing nearest neighbors

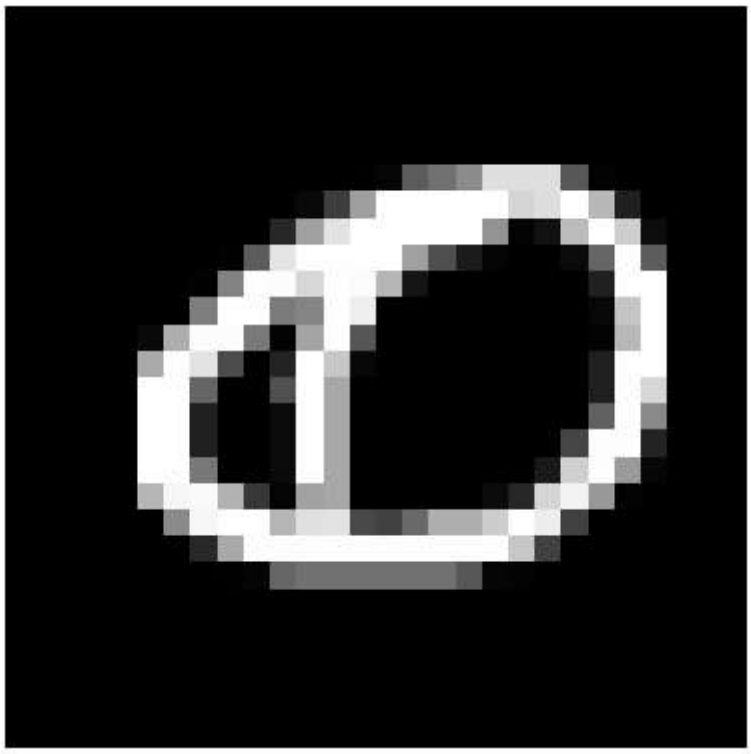
Now that we have a distance function defined, we can now turn to nearest neighbor classification.

In [23]:
```
## Takes a vector x and returns the index of its nearest neighbor in train_data
def find_NN(x):
    # Compute distances from x to every row in train_data
    distances = [squared_dist(x,train_data[i,]) for i in range(len(train_labels))]
    # Get the index of the smallest distance
    return np.argmin(distances)

## Takes a vector x and returns the class of its nearest neighbor in train_data
def NN_classifier(x):
    # Get the index of the the nearest neighbor
    index = find_NN(x)
    # Return its class
    return train_labels[index]
```

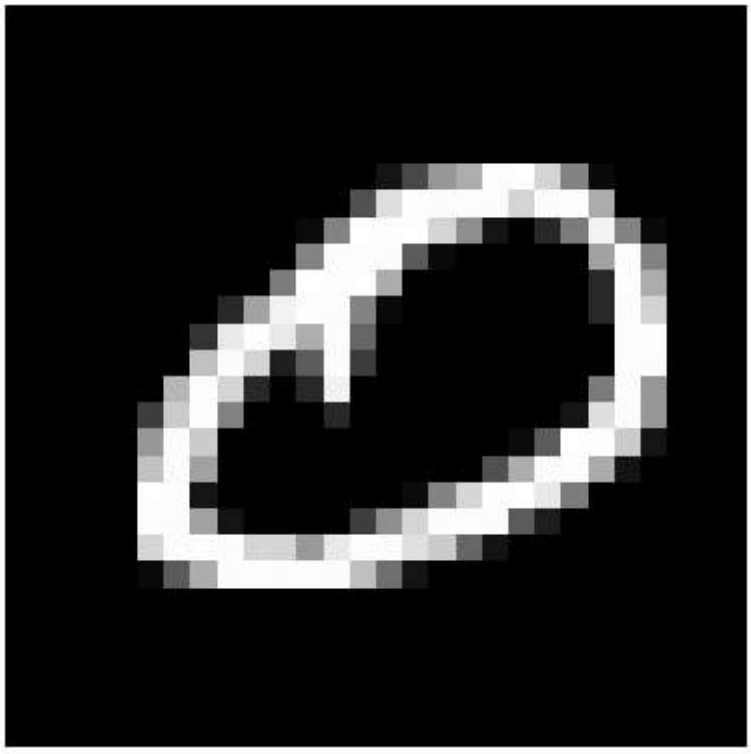
In [24]:
```
## A success case:
print("A success case:")
print("NN classification: ", NN_classifier(test_data[0,]))
```

```
print("True label: ", test_labels[0])
print("The test image:")
vis_image(0, "test")
print("The corresponding nearest neighbor image:")
vis_image(find_NN(test_data[0,]), "train")
```

```
A success case:
NN classification:  0
True label:  0
The test image:
```



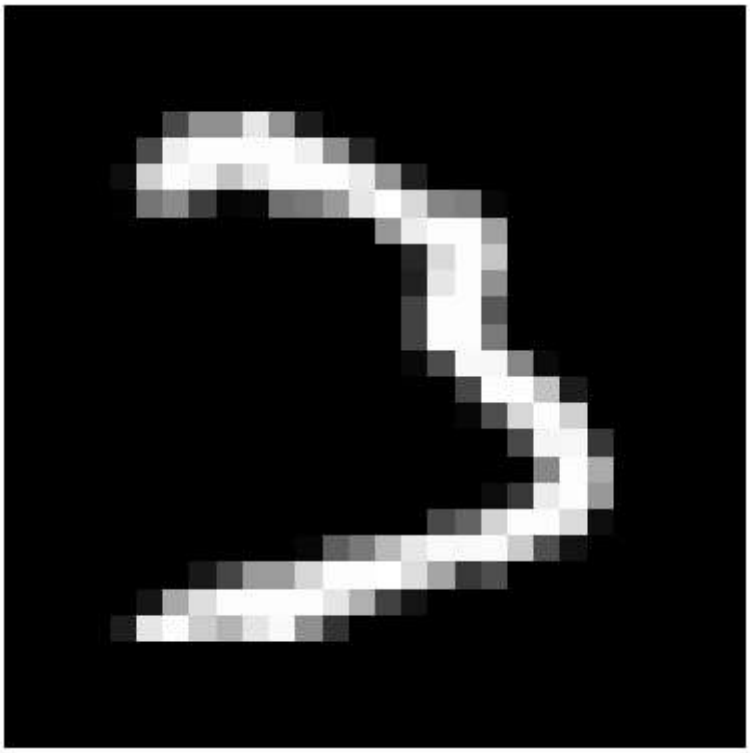
```
Label 0
The corresponding nearest neighbor image:
```



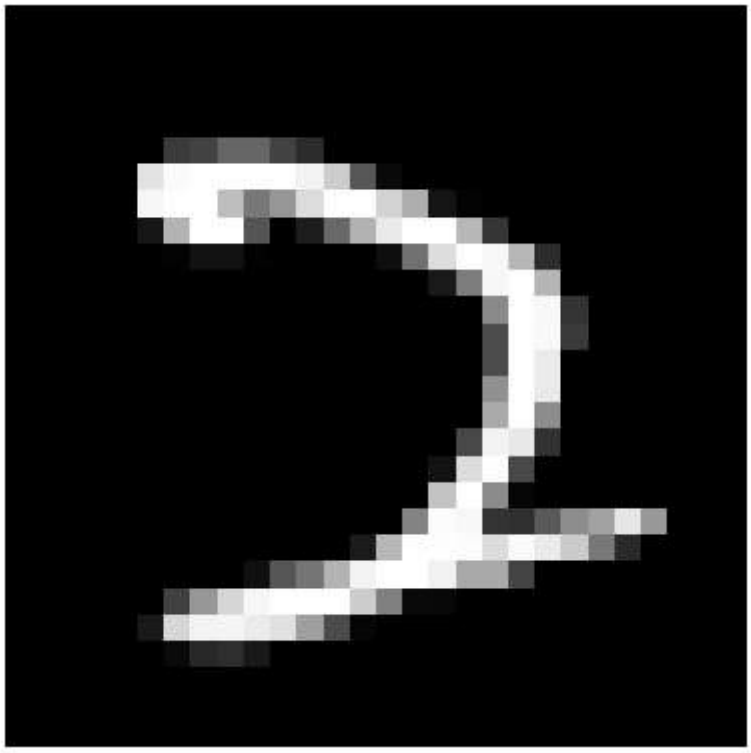
```
Label 0
```

In [25]:
```
## A failure case:
print("A failure case:")
print("NN classification: ", NN_classifier(test_data[39,]))
print("True label: ", test_labels[39])
print("The test image:")
vis_image(39, "test")
print("The corresponding nearest neighbor image:")
vis_image(find_NN(test_data[39,]), "train")
```

```
A failure case:
NN classification:  2
True label:  3
The test image:
```



```
Label 3
The corresponding nearest neighbor image:
```



```
Label 2
```

5. For you to try

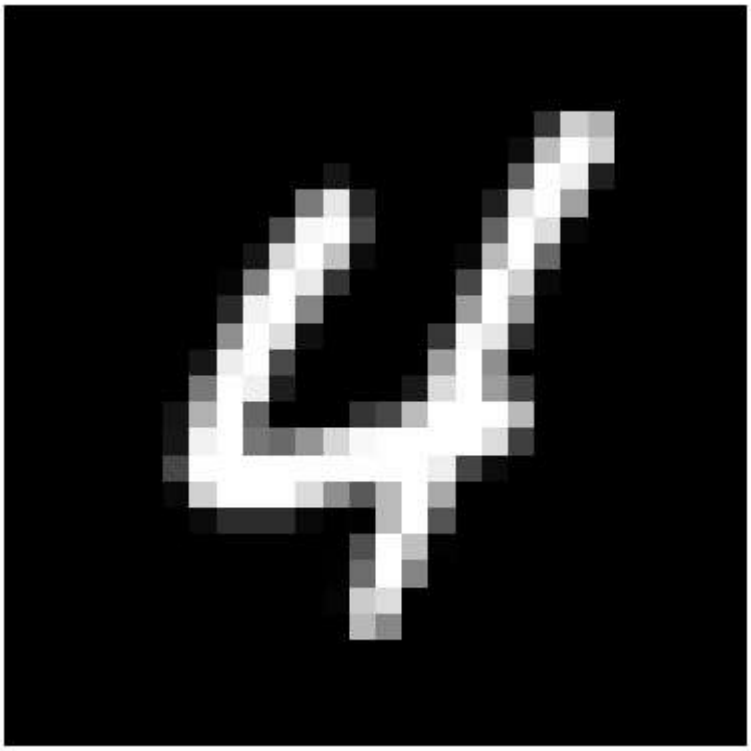
The above two examples show the results of the NN classifier on test points number 0 and 39.

Now try test point number 100.

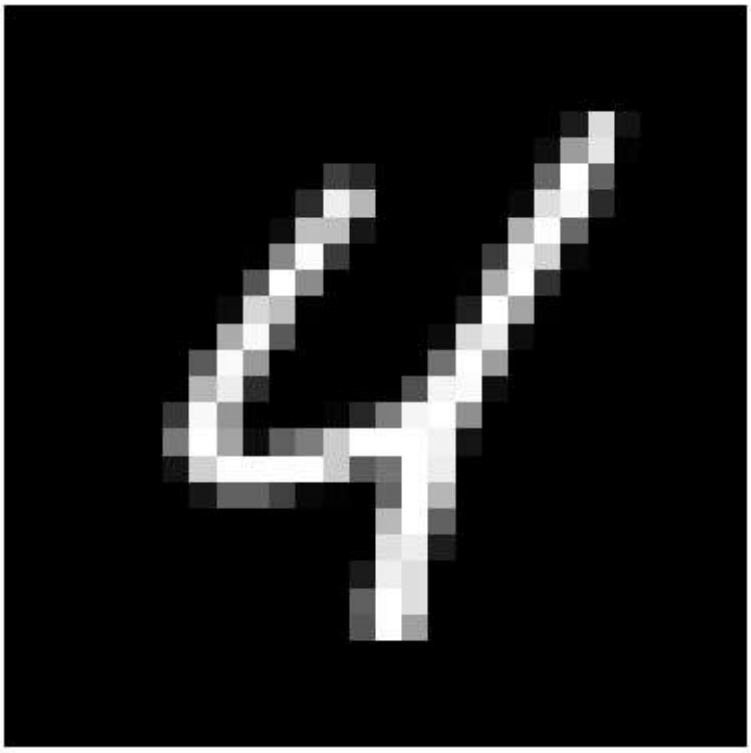
- What is the index of its nearest neighbor in the training set? *Record the answer: you will enter it as part of this week's assignment.*
- Display both the test point and its nearest neighbor.
- What label is predicted? Is this the correct label?

In [26]:
```
print("=== Sample test_id=100 ===")
vis_image(100, "test")
print(f"=== NN: train_id={find_NN(test_data[100])} ===")
vis_image(find_NN(test_data[100]), "train")
```

```
=== Sample test_id=100 ===
```



```
Label 4
=== NN: train_id=4711 ===
```



```
Label 4
```

6. Processing the full test set

Now let's apply our nearest neighbor classifier over the full data set.

Note that to classify each test point, our code takes a full pass over each of the 7500 training examples. Thus we should not expect testing to be very fast. The following code takes about 100-150 seconds on 2.6 GHz Intel Core i5.

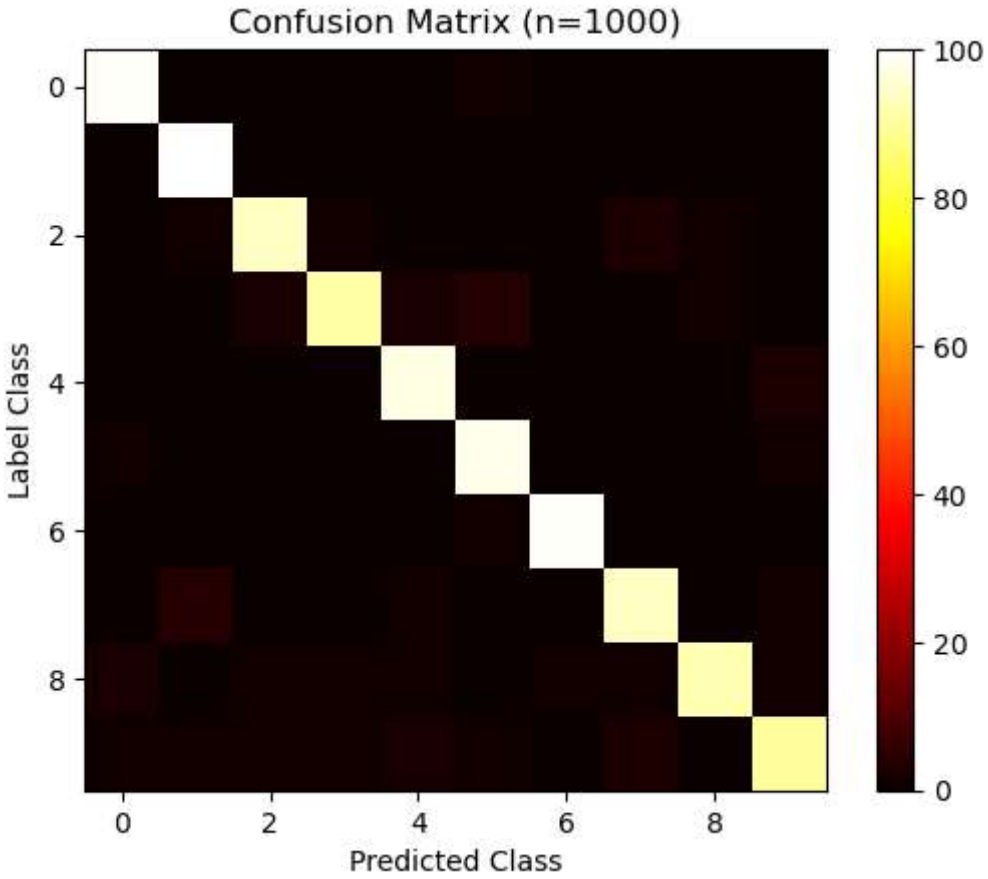
In [27]:
```
## Predict on each test data point (and time it!)
t_before = time.time()
test_predictions = [NN_classifier(test_data[i,]) for i in range(len(test_labels))]
t_after = time.time()

## Compute the error
err_positions = np.not_equal(test_predictions, test_labels)
error = float(np.sum(err_positions))/len(test_labels)

print("Error of nearest neighbor classifier: ", error)
print("Classification time (seconds): ", t_after - t_before)
```

```
Error of nearest neighbor classifier:  0.046
Classification time (seconds):  44.084826707839966
```

In [28]:
```
len(test_predictions)
mat = np.zeros((10, 10))
for lab, pred in zip(test_labels, test_predictions):
  mat[lab, pred] += 1
print(mat)
plt.imshow(mat, cmap="hot")
plt.xlabel("Predicted Class")
plt.ylabel("Label Class")
plt.title(f"Confusion Matrix (n={len(test_predictions)})")
plt.colorbar()
```

```
[[ 99.   0.   0.   0.   0.   1.   0.   0.   0.   0.]
 [  0. 100.   0.   0.   0.   0.   0.   0.   0.   0.]
 [  0.   1.  94.   1.   0.   0.   0.   3.   1.   0.]
 [  0.   0.   2.  91.   2.   4.   0.   0.   1.   0.]
 [  0.   0.   0.   0.  97.   0.   0.   0.   0.   3.]
 [  1.   0.   0.   0.   0.  98.   0.   0.   0.   1.]
 [  0.   0.   0.   0.   0.   1.  99.   0.   0.   0.]
 [  0.   4.   0.   0.   1.   0.   0.  94.   0.   1.]
 [  2.   0.   1.   1.   1.   0.   1.   1.  92.   1.]
 [  1.   1.   1.   1.   2.   1.   0.   3.   0.  90.]]
```

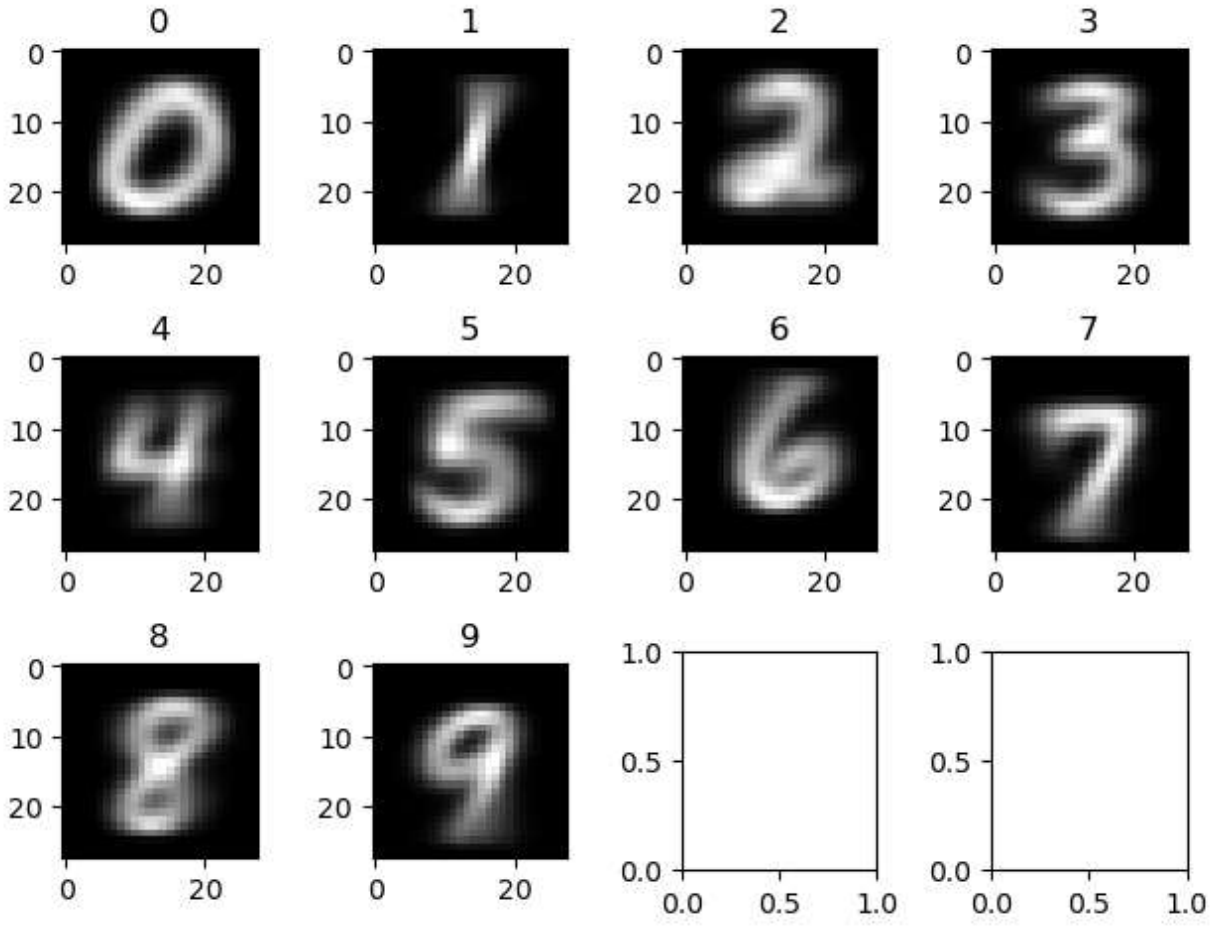
Out[28]:
```
<matplotlib.colorbar.Colorbar at 0x2daa16f33a0>
```



In [29]:
```
miscls_arr = np.zeros((10,))
ct_arr = np.zeros((10,))

for lab, pred in zip(test_labels, test_predictions):
  ct_arr[lab] += 1
  if lab != pred:
    miscls_arr[lab] += 1
print(miscls_arr)
print(ct_arr)
```

```
[ 1.  0.  6.  9.  3.  2.  1.  6.  8. 10.]
[100. 100. 100. 100. 100. 100. 100. 100. 100. 100.]
```

In [34]:
```
fig, axs = plt.subplots(nrows=3, ncols=4)
fig.tight_layout()
for i in range(10):
  ct, img = 0, np.zeros(784)
  for d, l in zip(train_data, train_labels):
    if l == i:
      img += d
      ct += 1
  mean_img = img / ct
  axs[i//4,i%4].imshow(mean_img.reshape((28, 28)), cmap="gray")
  axs[i//4,i%4].set_title(str(i))
  # show_digit(mean_img.reshape((28, 28)))
```



7. Faster nearest neighbor methods

Performing nearest neighbor classification in the way we have presented requires a full pass through the training set in order to classify a single point. If there are N training points in $\mathbb{R}^d$, this takes $O(Nd)$ time.

Fortunately, there are faster methods to perform nearest neighbor look up if we are willing to spend some time preprocessing the training set. `scikit-learn` has fast implementations of two useful nearest neighbor data structures: the *ball tree* and the *k-d tree*.

In [31]:
```
from sklearn.neighbors import BallTree

## Build nearest neighbor structure on training data
t_before = time.time()
ball_tree = BallTree(train_data)
t_after = time.time()

## Compute training time
t_training = t_after - t_before
print("Time to build data structure (seconds): ", t_training)

## Get nearest neighbor predictions on testing data
t_before = time.time()
test_neighbors = np.squeeze(ball_tree.query(test_data, k=1, return_distance=False))
ball_tree_predictions = train_labels[test_neighbors]
t_after = time.time()
```

```
## Compute testing time
t_testing = t_after - t_before
print("Time to classify test set (seconds): ", t_testing)

## Verify that the predictions are the same
print("Ball tree produces same predictions as above? ", np.array_equal(test_predictior
```

```
Time to build data structure (seconds):  0.6670012474060059
Time to classify test set (seconds):  5.663107872009277
Ball tree produces same predictions as above?  True
```

In [32]:

```
from sklearn.neighbors import KDTree

## Build nearest neighbor structure on training data
t_before = time.time()
kd_tree = KDTree(train_data)
t_after = time.time()

## Compute training time
t_training = t_after - t_before
print("Time to build data structure (seconds): ", t_training)

## Get nearest neighbor predictions on testing data
t_before = time.time()
test_neighbors = np.squeeze(kd_tree.query(test_data, k=1, return_distance=False))
kd_tree_predictions = train_labels[test_neighbors]
t_after = time.time()

## Compute testing time
t_testing = t_after - t_before
print("Time to classify test set (seconds): ", t_testing)

## Verify that the predictions are the same
print("KD tree produces same predictions as above? ", np.array_equal(test_predictions,
```

```
Time to build data structure (seconds):  1.1469993591308594
Time to classify test set (seconds):  6.542593240737915
KD tree produces same predictions as above?  True
```

In [1]:
```
import numpy as np
labels = [b'NO', b'DH', b'SL']
data = np.loadtxt("../data/spine-data.txt", converters={6: lambda s : labels.index(s)}
```

In [2]:
```
tr = data[:250, :-1]
te = data[250:, :-1]
tr_y = data[:250, -1]
te_y = data[250:, -1]

print(tr.shape, tr_y.shape)
```

```
(250, 6) (250,)
```

In [3]:
```
# copied from example code
def squared_dist(x,y):
  v = np.sum(np.square(x - y))
  return v

def l1_norm(x, y):
  return np.sum(np.abs(x - y))

def find_NN(x, dist_fn):
  # Compute distances from x to every row in train_data
  distances = [dist_fn(x,tr[i,]) for i in range(len(tr_y))]
  # Get the index of the smallest distance
  return np.argmin(distances)
def NN_classifier(x, dist_fn):
  # Get the index of the the nearest neighbor
  index = find_NN(x, dist_fn)
  # Return its class
  return tr_y[index]
```

In [4]:
```
def experiment(dist_fn):
  test_predictions = [NN_classifier(te[i,], dist_fn) for i in range(len(te_y))]
  err_positions = np.not_equal(test_predictions, te_y)
  error = float(np.sum(err_positions))/len(te_y)

  mat = np.zeros((3, 3))
  for lab, pred in zip(te_y, test_predictions):
    x, y = int(lab), int(pred)
    mat[x, y] += 1

  return error, mat
```

In [5]:
```
print("=== L1 Norm ===")
l1_e, l1_mat = experiment(l1_norm)
print(f"Error: {l1_e:0.4f}")
print(l1_mat.astype(int))
print("=== L2 Norm ===")
l2_e, l2_mat = experiment(squared_dist)
print(f"Error: {l2_e:0.4f}")
print(l2_mat.astype(int))
```

```
=== L1 Norm ===
Error: 0.2167
[[14  0  2]
 [ 9  9  0]
 [ 1  1 24]]
=== L2 Norm ===
Error: 0.2333
[[12  1  3]
 [ 9  9  0]
 [ 1  0 25]]
```