Worksheet 1 - Nearest Neighbor

1) $d = 100$

2) length of vector $= \sqrt{d}$

3) distance $[(1,2,3), (3,2,1)] = \sqrt{2^2 + 0^2 + 2^2}$

$= 2\sqrt{2}$

4) $x, x' \in R^d$

Euclidean distance $\|x - x'\| = \sqrt{4d}$

$= 2\sqrt{d}$

5)

Label	Frequency
A	50%
B	20%
C	20%
D	10%

(a) Since we are picking a label (A, B, C, D) at random.

Error rate $= \frac{3}{4} = 0.75$

(b) Label to return $\rightarrow$ A

Error rate $\rightarrow$ 0.5 since those would be cases where label $\neq$ A

6) $X = [0,2]^2$

Labels $Y = \{1,2,3\}$

a) Class 2

b) Class 2 since

Class 2 → (0.5, 0.5)

Pt. (1.5, 0.5)

$\sqrt{1^2 + 0} = 1$

Class 1 - (0.5, 1.5)

Pt. (1.5, 0.5)

$\sqrt{1^2 + 1^2}$

$= \sqrt{2}$

Since class 1 with our point has less euclidean distance, it would be the label.

c) dist. to class 2 $= \sqrt{(0.5-2)^2 + (0.5-2)^2}$

$= \sqrt{1.5^2 + 1.5^2}$

$= 2.12$

dist to class 1 : $\sqrt{(0.5-2)^2 + (1.5-2)^2}$

$= 1.58$

Class 1 label would be picked since distance is closest.

d) Class 3

e) Since test points are uniformly distributed we would pick class 3 with 50% prob.

∴ error rate = 50%.

7) x. pt = (3.5, 4.5)

x to (2,2) = $\sqrt{(3.5-2)^2+(4.5-2)^2}$ = 2.915

x to (4,2) = $\sqrt{(3.5-4)^2+(4.5-2)^2}$ = 2.549

x to (6,2) = $\sqrt{(3.5-6)^2+(4.5-2)^2}$ = 3.535

x to (2,4) = $\sqrt{(3.5-2)^2+(4.5-4)^2}$ = 1.581 — 2

x to (4,4) = $\sqrt{(3.5-4)^2+(4.5-4)^2}$ = 0.707, — 1

x to (6,4) = $\sqrt{(3.5-6)^2+(4.5-4)^2}$ = 2.549 — 5

x to (2,6) = $\sqrt{(3.5-2)^2+(4.5-6)^2}$ = 2.121 — 4

x to (4,6) = $\sqrt{(3.5-4)^2+(4.5-6)^2}$ = 1.581 — 3

x to (6,6) = $\sqrt{(3.5-6)^2+(4.5-6)^2}$ = 2.915

(a) 1-NN Classification

Since l_2 is closest with (4,4) point, i.e Star *

(b) 3-NN Classification

Label would be square ■ since two pt. are square among 3 points

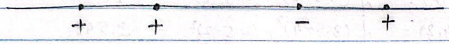
(c) 5-NN Classification

Label would be square ■ since 3 pts more points are squares in 5-NN.

8) Since we are doing 4-fold cross-validation for data set of size 10,000

that means we have 2500 would be in each of the traing sets

9)



LOOCV error for 1-NN

- If we leave out one data point for the classification we would be wrong twice when we leave out (−) & (+) on the extreme right.
  So the error would be $\frac{2}{4} = 50\%$

LOOCV error for 3-NN

- We would misclassify only once when we leave out (−).
  Hence error $= \frac{1}{4} = 25\%$

Worksheet 2

1) $x = (-1, 1, -1, 1)$ $\quad x' = (1, 1, 1, 1)$

(a) $L_2 = \sqrt{2^2 + 0^2 + 2^2 + 0^2}$

$= \sqrt{8}$

(b) $L_1 = 2 + 0 + 2 + 0$

$= 4$

(c) $L_\infty = 2$

2) $x = (1, 2, 3, 4)$

(a) $\|x\|_1 = 1 + 2 + 3 + 4$

$= 10$

(b) $\|x\|_2 = \sqrt{1^2 + 2^2 + 3^2 + 4^2}$

$= \sqrt{1 + 4 + 9 + 16}$

$= \sqrt{30}$

(c) $\|x\|_\infty = 4$

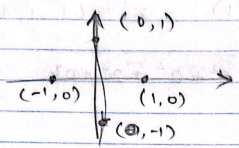
3)

(a) l_2 = ball

(b) l_1 = diamond

(c) l_∞ = box

4) $\mathbb{R}^2$ with $\|x\|_1 = \|x\|_2 = 1$



$(0,1)\ (-1,0)\ (1,0)\ (0,-1)$

5) (a) $X = \mathbb{R}$

$d(x,y) = x - y$

- $d(x,y)$ can be negative.
- $d(x,y)$ is not symmetric.

$d(x,z) \leq d(x,y) + d(y,z)$

But. $x - z = x - y + y - z$

$\therefore$ satisfy triangle of inequality

$\Rightarrow$ But is does not satisfy all metric properties

(b) Hamming distance

$d(x,y) = \#$ of position on which x & y differ

$\Rightarrow$ It satisfies all of the metric properties

c)

$\Rightarrow$ It does not satisfy triangle of inequality

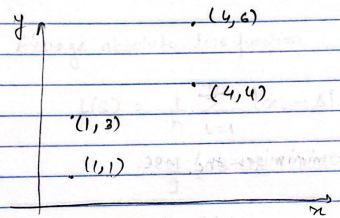
considering example

Worksheet 3

1 (a) Classification
(b) Regression
(c) Regression
(d) classification

Worksheet 4

1.



(a) Without knowing x we would predict y = mean = 3.5

$$MSE = \frac{1}{4}(2.5^2 + 0.5^2 + 0.5^2 + 2.5^2)$$

$$= 3.25$$

(b) Linear function $y = x$

$$MSE = \frac{1}{4}((1-1)^2 + (3-1)^2 + (4-4)^2 + (6-4)^2)$$

$$= 2.$$

c)

$$\bar{x} = 2.5$$

$$\bar{y} = 3.5$$

To minimize the MSE for the points

$$a = \frac{\sum_{i=1}^{n} (y^{(i)} - \bar{y})(x^{(i)} - \bar{x})}{\sum_{i=1}^{n} (x^{(i)} - \bar{x})^2}$$

$$= 1$$

$$b = \bar{y} - a\bar{x}$$
$$= 3.5 - 2.5$$
$$= 1$$

∴ line $\underline{y = x + 1}$ minimizes the MSE

3.

$$L(s) = \frac{1}{n} \sum_{i=1}^{n} (x_i - s)^2$$

(a) $$\frac{dL(s)}{ds} = \frac{1}{n} \sum_{i=1}^{n} 2(x_i - s).(-1)$$

$$= -\frac{2}{n} \sum_{i=1}^{n} (x_i - s)$$

(b) $$\frac{dL}{ds} = 0$$

$$-\frac{2}{n} \sum_{i=1}^{n} (x_i - s) = 0$$

$$s = \frac{1}{n} \sum_{i=1}^{n} x_i$$

4.) $L(s) = \frac{1}{n} \sum_{i=1}^{n} |x_i - s|$

a) 1, 2, 3, 4, 5, 6, 7, 8, 90

Mean = 14

b) Average absolute loss function

$$L(s) = \frac{1}{n} \sum_{i=1}^{n} |x_i - s|$$

$$= \frac{1}{9}(13+12+11+10+9+8+7+6+76)$$

$$= 16.88$$

c) Average absolute loss if $s = 5$

$$L(s) = \frac{1}{9}(4+3+2+1+0+1+2+3+85)$$

$$= 11.22$$

d) Median of the example = 5

6.

a) $\frac{1}{n} y \cdot 1$

b) $\sum_{n=1}^{d} x_n^{(i)} x_n^{(j)}$

$= XX^T$

c) $\frac{1}{n} 1 x$

d) $\frac{1}{n} X^T X$

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 [1]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import time

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

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

In [2]:
```
## 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 [3]:
```
## 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)))
```

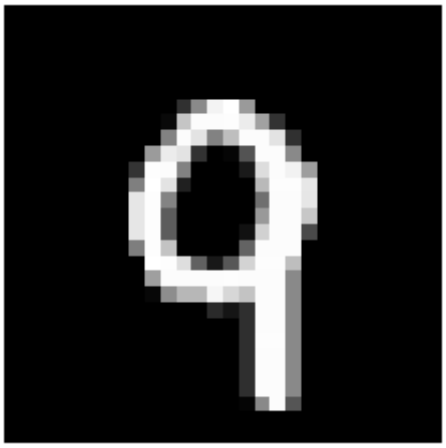
```
Training set distribution:
{0: 750, 1: 750, 2: 750, 3: 750, 4: 750, 5: 750, 6: 750, 7: 750, 8: 750, 9: 750}
Test set distribution:
{0: 100, 1: 100, 2: 100, 3: 100, 4: 100, 5: 100, 6: 100, 7: 100, 8: 100, 9: 100}
```

2. Visualizing the data

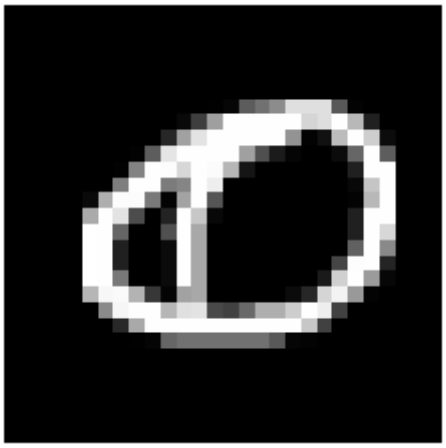
Each data point is stored as 784-dimensional vector. To visualize a data point, we first reshape it to a 28x28 image.

In [4]:

```
## 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
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 [5]:
```
## 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
```

In [19]:
```
vis_image(1, "train")
```

```
Label 2
```

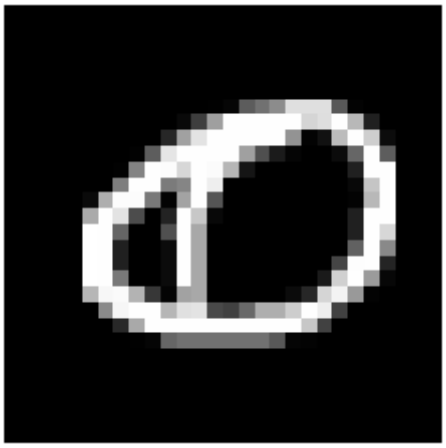
4. Computing nearest neighbors

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

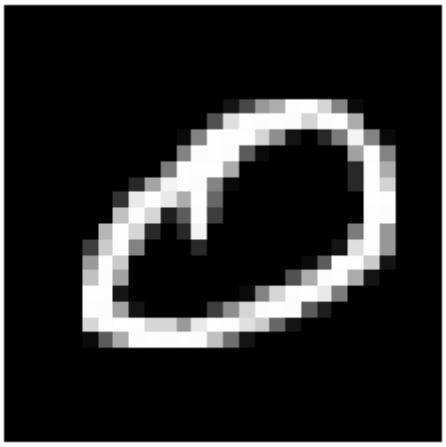
In [13]:
```
## Takes a vector x and returns the index of its nearest neighbor in train_data
def find_NN(x):
    # Compute distances from x to every row in train_data
    distances = [squared_dist(x,train_data[i,]) for i in range(len(train_labels))]
    # Get the index of the smallest distance
    return 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 [14]:
```
## 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:
```

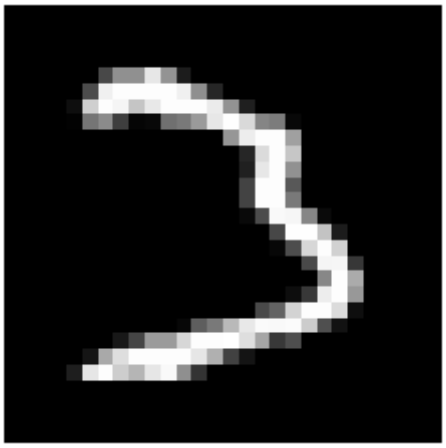


```
Label  0
```

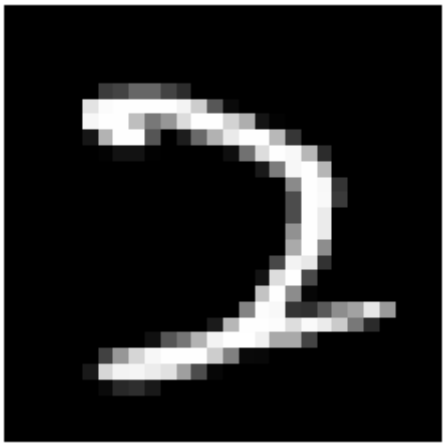
In [15]:

```
## 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. 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 [20]:

```
## 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):  71.36217617988586
```

6. 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 $O(N d)$ 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 [21]:
```
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=Fals
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_pred
```

```
Time to build data structure (seconds):  0.6666340827941895
Time to classify test set (seconds):  6.306871175765991
Ball tree produces same predictions as above?  True
```

In [22]:
```
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_predic
```

```
Time to build data structure (seconds):  0.7134437561035156
Time to classify test set (seconds):  7.845537185668945
KD tree produces same predictions as above?  True
```

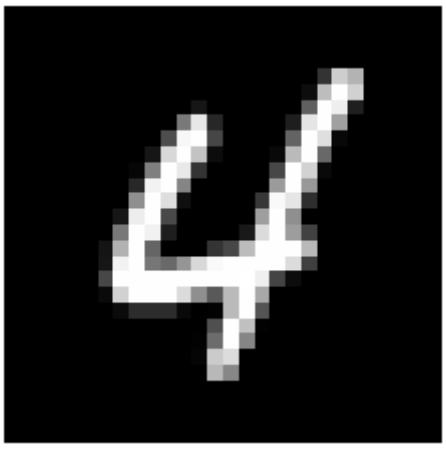
LAB1

1. a) For test point 100, print its image as well as the image of its nearest neighbor in the training set. Put these images in your writeup. Is this test point classified correctly?

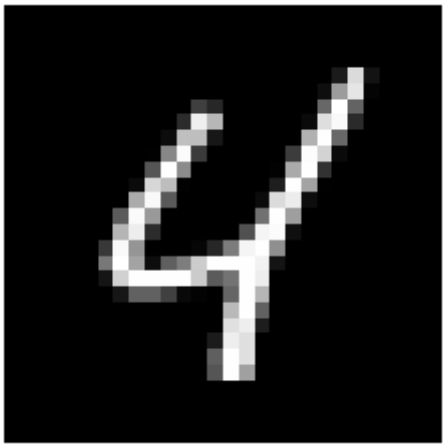
In [24]:

```
print("The test image 100:")
vis_image(100, "test")
print("Nearest Neighbour to test point 100:")
vis_image(find_NN(test_data[100,]), "train")
```

```
The test image 100:
```



```
Label 4
Nearest Neighbour to test point 100:
```



```
Label 4
```

The classified label 4 seems to be correct.

1. b)

In [77]:

```
import numpy as np
```

```
#initializing 10x10 matrix with just zeroes.
dimensions = (10, 10)
confusion_matrix = np.zeros(dimensions)

#Looping through original test labels and ball_tree_predictions to create confus
for x, y in zip(test_labels, ball_tree_predictions):
    confusion_matrix[x][y] = confusion_matrix[x][y] + 1
    #print(str(x)+" "+str(y))

#printing the 10x10 confusion matrix
confusion_matrix
```

Out[77]:
```
array([[ 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.]])
```

In [49]:
```
non_diagonal_elements = np.ones(10, dtype=bool)
non_diagonal_elements[0] = False
non_diagonal_elements
```

Out[49]:
```
array([False,  True,  True,  True,  True,  True,  True,  True,  True,
        True])
```

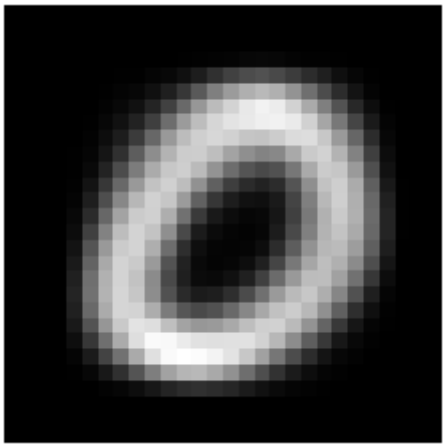
In [58]:
```
#creating disctionary tp store the digit , misclassified count of digits
digit_missclassified = {}
for i in range(10):
    #creating a array with all true values
    non_diagonal_elements = np.ones(10, dtype=bool)
    #need to exclude the diagonal element , hence making the flag as false to ex
    non_diagonal_elements[i] = False
    #calculating the sum of row excluding the diagonal element
    digit_missclassified[i] = np.sum(confusion_matrix[i],where = non_diagonal_el

print("Digit misclassified most often :",max(digit_missclassified, key=digit_mis
print("Digit misclassified least often :", min(digit_missclassified, key=digit_m
```

```
Digit misclassified most often : 9
Digit misclassified least often : 1
```

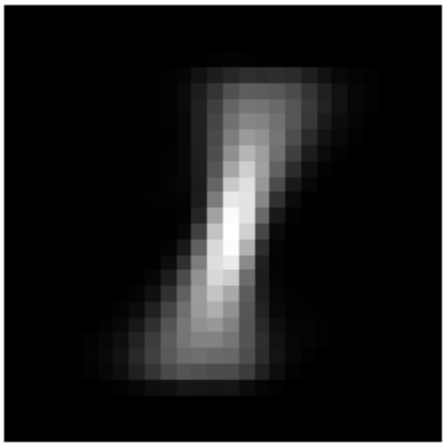
1 . c) For each digit 0 ≤ i ≤ 9: look at all training instances of image i, and compute their mean. This average is a 784-dimensional vector. Use the show digit routine to print out these 10 average-digits.

In [71]:
```
for i in range(10):
    indexes = np.where(train_labels == i)
    i_matrix = train_data[indexes]
    print("Average Digit Image for : ",i)
    show_digit(i_matrix.mean(0))
```

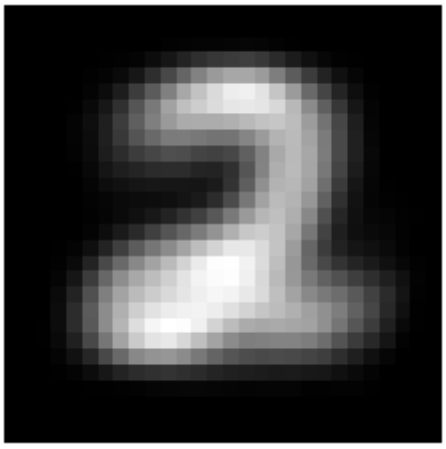
```
Average Digit Image for :  0
```



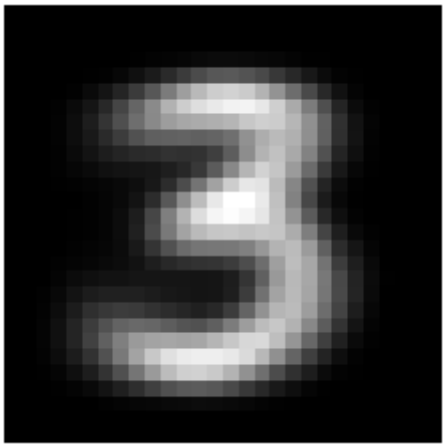
```
Average Digit Image for :  1
```



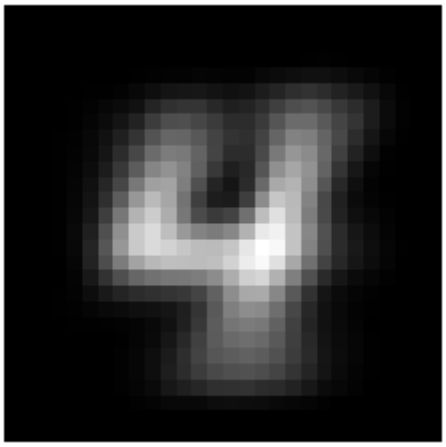
```
Average Digit Image for :  2
```



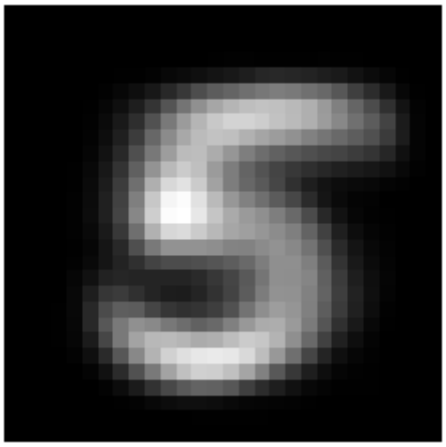
```
Average Digit Image for :  3
```



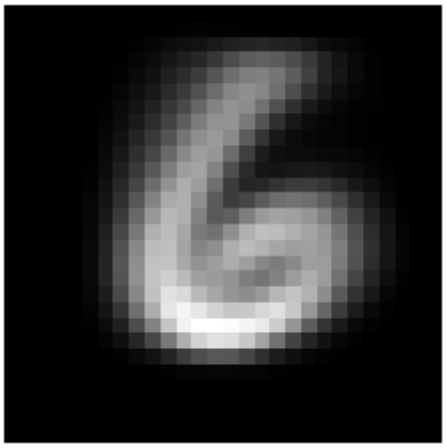
```
Average Digit Image for :  4
```



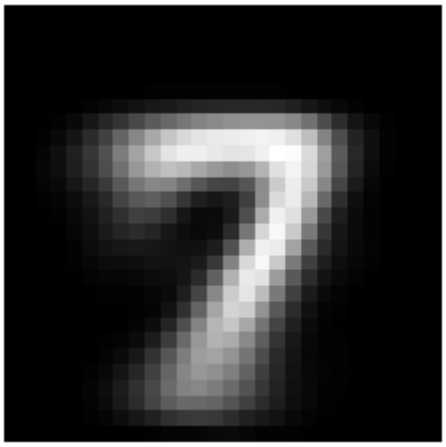
```
Average Digit Image for :  5
```



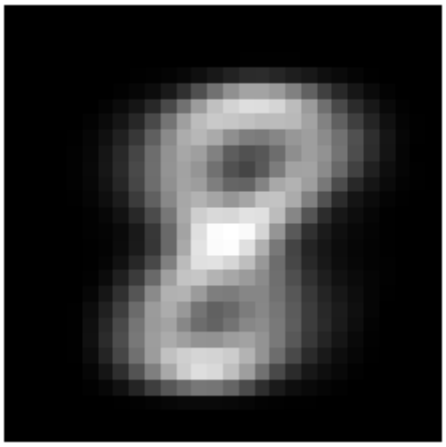
```
Average Digit Image for :  6
```



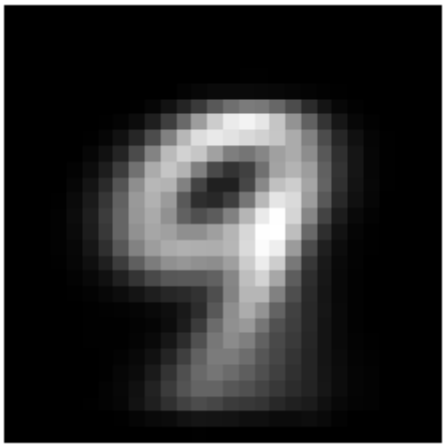
```
Average Digit Image for :  7
```



```
Average Digit Image for :  8
```



```
Average Digit Image for :  9
```



In [ ]:

In [1]:
```
import numpy as np
# Load data set and code labels as 0 = 'NO', 1 = 'DH', 2 = 'SL'
labels = [b'NO', b'DH', b'SL']
data = np.loadtxt('spine-data.txt', converters={6: lambda s: labels.index(s)})
```

In [2]:
```
X, y = data[:,:6], data[:, 6]
```

In [3]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size = 250, test
```

In [5]:
```
## Computes squared Euclidean distance between two vectors.
def l2_dist(x,y):
    return np.sum(np.square(x-y))
```

In [6]:
```
def l1_dist(x,y):
    return np.sum(np.abs(x-y))
```

In [7]:
```
## Takes a vector x and returns the index of its nearest neighbor in X_train
def find_NN(x, dist="l2"):
    if(dist=="l2"):
        # Compute distances from x to every row in train_data
        distances = [l2_dist(x,X_train[i,]) for i in range(len(y_train))]
    else:
        distances = [l1_dist(x,X_train[i,]) for i in range(len(y_train))]
    return np.argmin(distances)

## Takes a vector x and returns the class of its nearest neighbor in X_train
def NN_classifier(x, distance="l2"):
    # Get the index of the the nearest neighbor
    if(distance=="l2"):
        index = find_NN(x, "l2")
    else:
        index = find_NN(x, "l1")
    # Return its class
    return y_train[index]
```

2 a) What error rates do you get on the test set for each of the two distance functions?

In [8]:
```
## Predict on each test data point with l2 distance
test_predictions_l2 = [NN_classifier(X_test[i,],"l2") for i in range(len(y_test)

## Predict on each test data point with l1 distance
test_predictions_l1 = [NN_classifier(X_test[i,],"l1") for i in range(len(y_test)
```

In [9]:
```
## Compute the error with l2 distance
err_positions = np.not_equal(test_predictions_l2, y_test)
error = float(np.sum(err_positions))/len(y_test)

print("Error of nearest neighbor classifier with l2 distance: ", error)
```

```
Error of nearest neighbor classifier with l2 distance:  0.23333333333333334
```

In [10]:
```
## Compute the error with l1 distance
err_positions = np.not_equal(test_predictions_l1, y_test)
error = float(np.sum(err_positions))/len(y_test)

print("Error of nearest neighbor classifier with l1 distance: ", error)
```

```
Error of nearest neighbor classifier with l1 distance:  0.21666666666666667
```

2 b) For each of the two distance functions, give the confusion matrix of the NN classifier.

In [11]:
```
import numpy as np

#initializing 10x10 matrix with just zeroes.
dimensions = (3, 3)
confusion_matrix_l2 = np.zeros(dimensions)
#Looping through original test labels and test predictions to create confusion m
for a, b in zip(y_test, test_predictions_l2):
    confusion_matrix_l2[int(a)][int(b)] = confusion_matrix_l2[int(a)][int(b)] +
    #print(str(a)+" "+str(b))

#printing the 10x10 confusion matrix
print("Confusion matrix with l2 distance:")
confusion_matrix_l2
```

Out[11]:
```
Confusion matrix with l2 distance:
array([[12.,  1.,  3.],
       [ 9.,  9.,  0.],
       [ 1.,  0., 25.]])
```

For l2 distance :

- label 0 i.e NO (Normal) was misclassified 4 times
- label 1 i.e DH (herniated disk) was misclassified 9 times
- label 2 i.e SL (spondilolysthesis) was misclassified 1 time

In [12]:
```
import numpy as np

#initializing 10x10 matrix with just zeroes.
dimensions = (3, 3)
confusion_matrix_l1 = np.zeros(dimensions)
#Looping through original test labels and test predictions to create confusion m
for a, b in zip(y_test, test_predictions_l1):
    confusion_matrix_l1[int(a)][int(b)] = confusion_matrix_l1[int(a)][int(b)] +
    #print(str(a)+" "+str(b))

#printing the 10x10 confusion matrix
print("Confusion matrix with l1 distance:")
confusion_matrix_l1
```

Out[12]:
```
Confusion matrix with l1 distance:
array([[14.,  0.,  2.],
       [ 9.,  9.,  0.],
       [ 1.,  1., 24.]])
```

For l1 distance :

- label 0 i.e NO (Normal) was misclassified 2 times
- label 1 i.e DH (herniated disk) was misclassified 9 times
- label 2 i.e SL (spondilolysthesis) was misclassified 2 times

In [ ]:

In [1]:
```
import numpy as np
data = np.loadtxt('wine.data', delimiter=',')
```

In [2]:
```
features_data, labels = data[:,1:], data[:,0]
```

3. a) Use leave-one-out cross-validation (LOOCV) to estimate the accuracy of the classifier and also to estimate the 3 ×3 confusion matrix.

In [3]:
```
#Computes squared Euclidean distance between two vectors
def squared_dist(x,y):
    return np.sum(np.square(x-y))
```

In [4]:
```
#predicts the label for a data row using leave-one-out cross-validation
def find_label_for_record_at_index(i,features_data,labels):
    training_data = np.delete(features_data, i, 0)
    train_lebels = np.delete(labels, i, 0)
    x = features_data[i]
    distances = [squared_dist(x,training_data[j,]) for j in range(len(train_lebe
    index = np.argmin(distances)
    return int(train_lebels[index])
```

In [5]:
```
predictions = [find_label_for_record_at_index(i,features_data,labels) for i in r
acc_predictions = np.equal(predictions,labels)
accuracy = float(np.sum(acc_predictions))/len(labels)
print("Accuracy of LOOCV :" ,accuracy)
```

```
Accuracy of LOOCV : 0.7696629213483146
```

In [6]:
```
#Confusion Matrix

#initializing the 3x3 matrix
dimensions = (3, 3)
confusion_matrix = np.zeros(dimensions)
#print(confusion_matrix)
for a,b in zip(labels,predictions):
    #I am suubtracting 1 because our labels are 1,2,3 and indexes are 0,1,2
    confusion_matrix[int(a)-1][b-1] = confusion_matrix[int(a)-1][b-1] + 1
    #print(str(int(a))+" "+str(b))

print("Confusion matrix for LOOCV :")
confusion_matrix
```

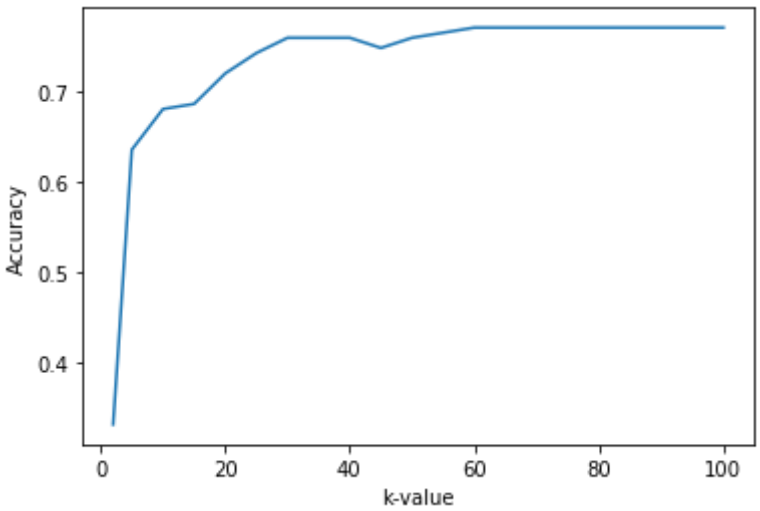
```
Confusion matrix for LOOCV :
```

Out[6]:
```
array([[52.,  3.,  4.],
       [ 5., 54., 12.],
       [ 3., 14., 31.]])
```

3 b) Estimate the accuracy of the 1-NN classifier using k-fold cross-validation using 20 different choices of k that are fairly well spread out across the range 2 to 100. Plot these estimates: put k on the horizontal axis and accuracy estimate on the vertical axis.

In [7]:

```
accuracy_dict = {}

#picking 20 k values for folds
folds = [2,5,10,15,20,25,30,35,40,45,50,60,65,70,75,80,85,90,95,100]

# Using for all the folds
for k in folds:
    k_fold_train_data = np.array_split(features_data, k)
    k_fold_labels = np.array_split(labels, k)

    predicted_actual = []

    #print(len(k_fold_train_data))
    for fold in range(len(k_fold_train_data)):

        #print("fold ",fold)
        hold_out_data = k_fold_train_data[fold]
        hold_out_labels = k_fold_labels[fold]

        #creating copy of list to keep original list intact
        temp_train_data = k_fold_train_data.copy()
        temp_labels = k_fold_labels.copy()

        #removing the holdout set
        del temp_train_data[fold]
        del temp_labels[fold]

        #flattening all the groups to create single training data exlcuing the f
        training_data = np.concatenate( temp_train_data, axis=0 )
        train_labels = np.concatenate( temp_labels, axis=0 )
        #print("size of hold our set ",len(k_fold_train_data[fold]))
        for i in range(len(hold_out_data)):
            x = hold_out_data[i]
            distances = [squared_dist(x,training_data[j,]) for j in range(len(tr
            index = np.argmin(distances)

            predicted_actual.append([int(train_labels[index]), int(hold_out_labe

    #print("k = ",k)
    predicted_actual_arr = np.asarray(predicted_actual)
    #print(predicted_actual_arr[:,0])
    predictions = predicted_actual_arr[:,0]
    actual_labels = predicted_actual_arr[:,1]
    correct_predictions = np.equal(predictions,actual_labels)
    #print(err_predictions)
    accuracy = float(np.sum(correct_predictions))/len(predicted_actual)
    #print("Accuracy ",accuracy)
    accuracy_dict[k] = accuracy


import matplotlib.pylab as plt

lists = sorted(accuracy_dict.items()) # sorted by key, return a list of tuples

x, y = zip(*lists) # unpack a list of pairs into two tuples
plt.xlabel("k-value")
plt.ylabel("Accuracy")
plt.plot(x, y)
plt.show()
```



3 c) Use leave-one-out cross-validation (LOOCV) to estimate the accuracy of the classifier and also to estimate the 3 ×3 confusion matrix.

In [8]:
```
#normalizing all the columns to [0,1]
features_data_normed = features_data / features_data.max(axis=0)
```

In [9]:
```
#Now calculating the error with normalized feature data
predictions = [find_label_for_record_at_index(i,features_data_normed,labels) for
acc_predictions = np.equal(predictions,labels)
acc = float(np.sum(acc_predictions))/len(labels)
print("Accuracy of LOOCV with normalized data:" ,acc)
```

```
Accuracy of LOOCV with normalized data: 0.9606741573033708
```

Clearly the normalization has increased the accuracy to 96%

In [ ]:
```

```