Handwritten Digit Recognition

Soumik Chaudhuri
E-mail: contact@soumik.in

Soumik.in

Abstract

Recognize handwritten digits from a dataset of 7500 training points and 1000 test points.

Methods Used: Nearest Neighbor (NN), BallTree, KDTree

Conclusion: BallTree offers the best performance and should be used for large data sets.

Dataset

We have taken a part of the original MNIST Database which contains 60,000 training points.

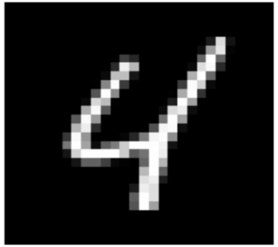
Our dataset contains **7500 training points and 1000 test points**.

Results of the NN Classifier on the Test Point 100

```
In [12]: ## Show the results of the NN classifier on the test point 100:
         print("NN classification: ", NN_classifier(test_data[100,]))
         print("True label: ", test_labels[100])
         print("The test image:")
         vis_image(100, "test")
         print()
         print("The corresponding nearest neighbor image:")
         vis_image(find_NN(test_data[100,]), "train")
```

```
NN classification:  4
True label:  4
The test image:
```

```
Label 4

The corresponding nearest neighbor image:
```



```
Label 4
```

Findings

NN Classification time for the test set: 72.2703025341034 seconds

BallTree Classification time for the test set: 7.922435760498047 seconds

KDTree Classification time for the test set: 9.811786651611328 seconds

Conclusion

Nearest Neighbor is the simplest method but becomes less efficient as the number of points increases.

BallTree is the fastest and should be used when we have a large dataset.

Handwritten Digit Recognition

June 11, 2020

Handwritten Digit Recognition

Dataset: 7500 training points, 1000 test points.

Methods: Nearest Neighbor (NN), BallTree, KDTree

Classification Time: NN - 72.27 seconds. BallTree - 7.92 seconds. KDTree - 9.81 seconds. (The execution time can vary depending on the CPU of the system.)

Conclusion: NN is the simplest method but becomes less efficient as the number of points increases. BallTree is the fastest and should be used when we have a large dataset.

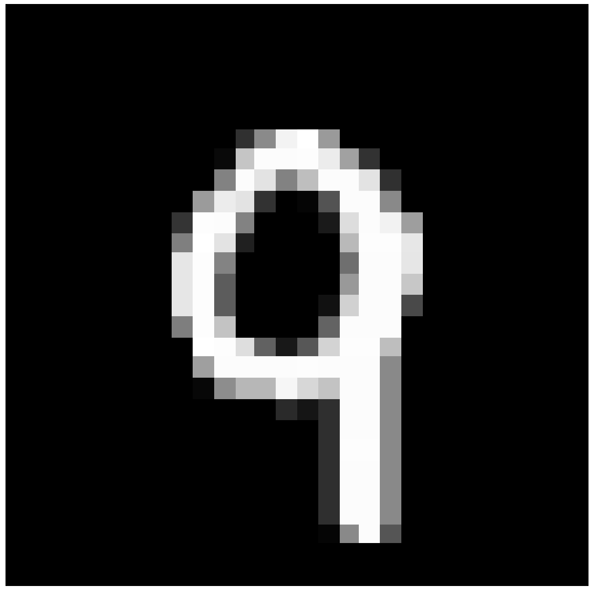
[1]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import time

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

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

[2]:
```
## Print the 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))
print()
## 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)))
print()
test_digits, test_counts = np.unique(test_labels, return_counts=True)
print("Test set distribution:")
print(dict(zip(test_digits, test_counts)))
```

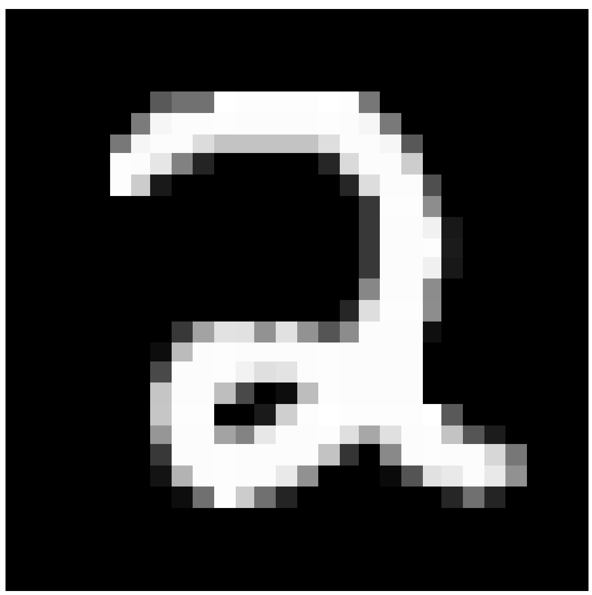
```
Training dataset dimensions:  (7500, 784)
Number of training labels:  7500
Testing dataset dimensions:  (1000, 784)
Number of testing labels:  1000

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}
```

[3]:

```
## 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) # Reshape the image to␣
↪28x28 pixels
    plt.show()
    return

## Define a function that takes an index into a particular data set ("train" or␣
↪"test") and displays that image.
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")

## View the second data point in the test set
vis_image(1, "test")
```



Label 9



Label 2

[4]:
```
## 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,]))
print()
## 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,]))
print()
## 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
```

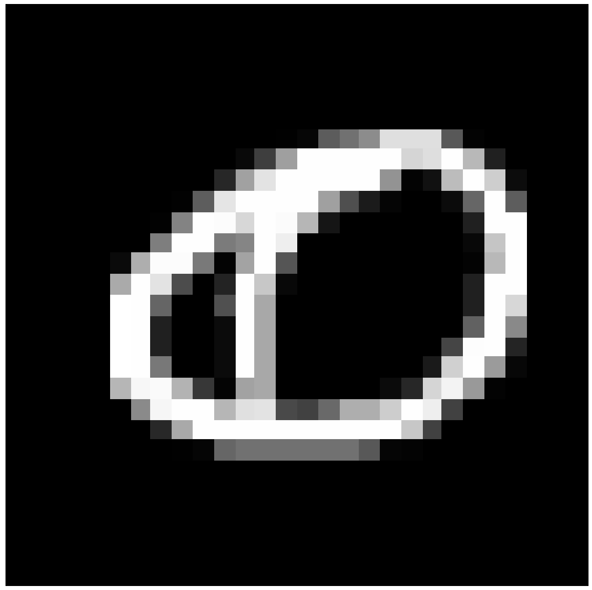
[5]:

```
## 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]
```

[6]:

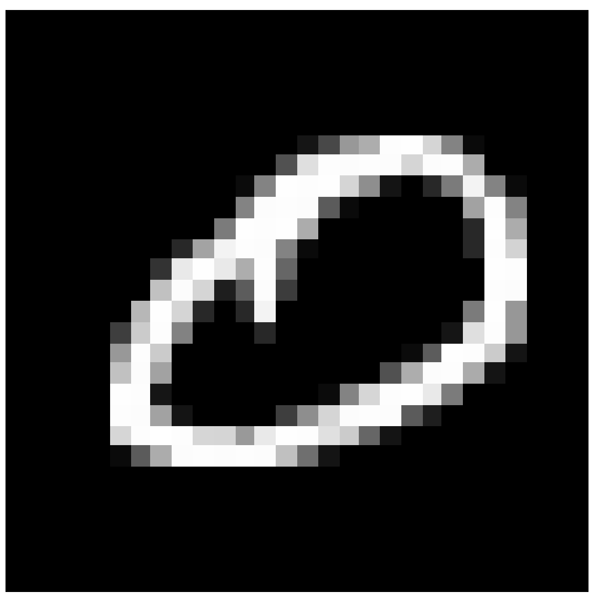
```
## Show the results of the NN classifier on the test point 0:
print("NN classification: ", NN_classifier(test_data[0,]))
print("True label: ", test_labels[0])
print("The test image:")
vis_image(0, "test")
print()
print("The corresponding nearest neighbor image:")
vis_image(find_NN(test_data[0,]), "train")
```

```
NN classification:  0
True label:  0
The test image:
```



Label 0

The corresponding nearest neighbor image:

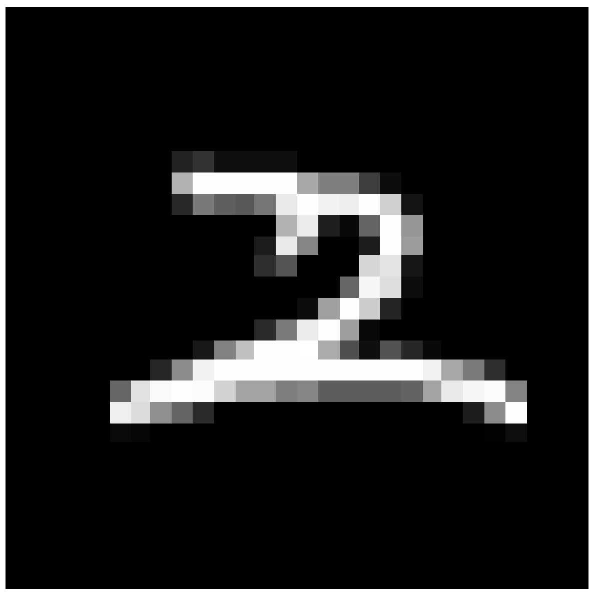


Label 0

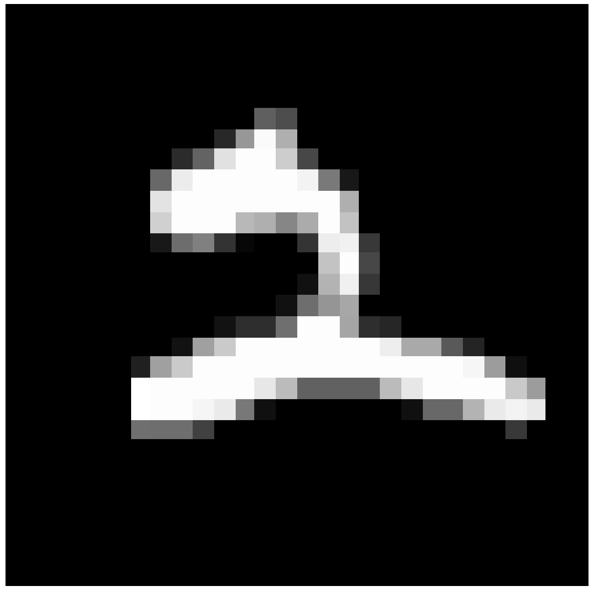
[7]:

```python
## Show the results of the NN classifier on the test point 40:
print("NN classification: ", NN_classifier(test_data[40,]))
print("True label: ", test_labels[40])
print("The test image:")
vis_image(40, "test")
print()
print("The corresponding nearest neighbor image:")
vis_image(find_NN(test_data[40,]), "train")
```

```
NN classification:  2
True label:  2
The test image:
```



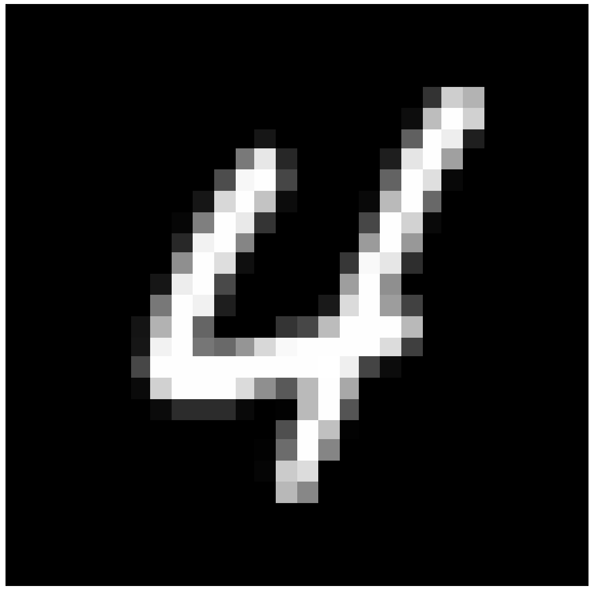
```
Label 2

The corresponding nearest neighbor image:
```



```
Label 2
```

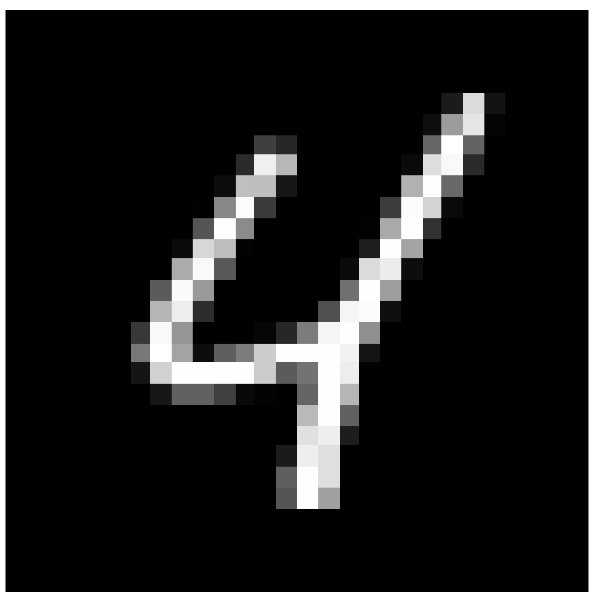
[8]:
```
## Show the results of the NN classifier on the test point 100:
print("NN classification: ", NN_classifier(test_data[100,]))
print("True label: ", test_labels[100])
print("The test image:")
vis_image(100, "test")
print()
print("The corresponding nearest neighbor image:")
vis_image(find_NN(test_data[100,]), "train")
```

```
NN classification:  4
True label:  4
The test image:
```



Label 4

The corresponding nearest neighbor image:



Label 4

[9]:
```python
## 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):  72.2703025341034
```

[10]:
```python
## Predict on each test data point using BallTree and time it

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_predictions, ball_tree_predictions))
```

```
Time to build data structure (seconds):  1.1969671249389648
Time to classify test set (seconds):  7.922435760498047
Ball tree produces same predictions as above?  True
```

[11]:

```python
## Predict on each test data point using KDTree and time it

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, kd_tree_predictions))
```

```
Time to build data structure (seconds):  1.8171391487121582
Time to classify test set (seconds):  9.811786651611328
KD tree produces same predictions as above?  True
```