NEAREST NEIGHBOUR FOR HANDWRITTEN DIGIT RECOGNITION

Objective

The goal of this project is to develop a classifier for recognizing handwritten digits using the nearest neighbor classification method. Specifically, the project aims to:

1. **Load** and **preprocess** the MNIST dataset.
2. **Implement** a simple nearest neighbor classifier to predict digit labels based on pixel values.
3. **Evaluate** the performance of the classifier on a test set.
4. **Compare** the performance and efficiency of the nearest neighbor method with optimized implementations using Ball Tree and KD Tree.

Summary

The MNIST dataset consists of 28x28 grayscale images of handwritten digits. For this project, a subset of the dataset is used, with 7500 training examples and 1000 test examples. The classification task involves recognizing digits from the test set using the nearest neighbor classifier.

1. **Data Description**:
    - **Training Set**: 7500 images with corresponding labels.
    - **Test Set**: 1000 images with corresponding labels.
    - **Data Dimensions**: Each image is represented as a 784-dimensional vector (28x28 pixels flattened).
2. **Data Visualization**:
    - Visualized sample images from both training and test sets to understand the data.
3. **Distance Metric**:
    - Implemented squared Euclidean distance as a measure for comparing the similarity between images.
4. **Nearest Neighbor Classification**:
    - Developed functions to compute nearest neighbors and classify digits based on the nearest training example.
    - Evaluated success and failure cases to demonstrate the classifier's performance.
5. **Processing the Full Test Set**:
    - Applied the nearest neighbor classifier to the entire test set.
    - Measured classification error and processing time to assess efficiency.

6. **Optimized Nearest Neighbor Methods**:
    - Implemented and compared faster nearest neighbor algorithms using Ball Tree and KD Tree from scikit-learn.
    - Compared the performance (in terms of time and accuracy) of these optimized methods with the basic nearest neighbor approach.

Results

1. **Data Preparation**:
    - Loaded and preprocessed data successfully.
    - Visualized sample images to confirm data integrity.
2. **Distance Computation**:
    - Verified the correctness of the distance metric by computing squared Euclidean distances between various digit images.
3. **Nearest Neighbor Classification**:
    - Achieved classification results with the basic nearest neighbor method.
    - Sampled predictions showed that the classifier could successfully recognize some digits but failed in others.
4. **Error and Timing**:
    - Computed classification error on the test set.
    - Recorded the time taken for classification using the basic nearest neighbor method.
5. **Optimized Methods**:
    - **Ball Tree**: Reduced training and testing times significantly compared to the basic method.
    - **KD Tree**: Also reduced times and matched predictions with the basic method.

Conclusion

The nearest neighbor classifier effectively recognizes handwritten digits but is computationally intensive. The optimized implementations using Ball Tree and KD Tree provide significant improvements in efficiency, making them suitable for large datasets. The results demonstrate that while the basic nearest neighbor method is straightforward and works, optimized nearest neighbor methods offer practical solutions for real-world applications where performance and speed are crucial.

Code :

'''

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 we 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 7500 examples and a test set of

1,000 examples.

About the dataset

Four files are available on this package:

 training set images

 training set labels

 test set images

 test set labels

These files are not in any standard image format. You have to write your own (very simple) program

to read them. You can find more details about the dataset here: http://yann.lecun.com/exdb/mnist/

'''

```
import numpy as np

import matplotlib.pyplot as plt

import time

#Load the training set:

train_data = np.load("D:\\Aditya's Notes\\Aditya's Data Science Notes\\Projects and Other Datasets\\ML PROJECTS\\data\\train_data.npy")
```

```
train_labels = np.load("D:\\Aditya's Notes\\Aditya's Data Science Notes\\Projects and Other Datasets\\ML PROJECTS\\data\\train_labels.npy")

#Load the testing set:

test_data = np.load("D:\\Aditya's Notes\\Aditya's Data Science Notes\\Projects and Other Datasets\\ML PROJECTS\\data\\test_data.npy")

test_labels = np.load("D:\\Aditya's Notes\\Aditya's Data Science Notes\\Projects and Other Datasets\\ML PROJECTS\\data\\test_labels.npy")


#Print out data 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))


#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)))


#2. Visualizing the data

#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.

#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") 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 first data point in the test set:
vis_image(0, "test")

'''
3. Squared Euclidean distance
 To compute nearest neighbors in our data set, we need to compute distances between data points.
A natural distance function is Euclidean distance: for two vectors , , their Euclidean distance is
defined as

 Often we omit the square root, and simply compute squared Euclidean distance:
 We will compute squared Euclidean distance. The following function does so:
'''

#Function that 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.
vis_image(4, "train")
```

```
vis_image(5, "train")
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.
vis_image(4, "train")
vis_image(1, "train")
print("Distance from 7 to 2: ", squared_dist(train_data[4,],train_data[1,]))
#Compute distance between two seven's in our training set.
vis_image(4, "train")
vis_image(7, "train")
print("Distance from 7 to 7: ", squared_dist(train_data[4,],train_data[7,]))

'''
4. Computing nearest neighbors
Now that we have a distance function defined, we can now turn to nearest neighbor classification.
Function that takes a vector x and returns the index of its nearest neighbor in train_data.
'''

#4. Computing nearest neighbors
#Now that we have a distance function defined,
# we can now turn to nearest neighbor classification.
#Function that 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)
#Function that takes a vector x and returns the class of its nearest neighbor in train_data.
def NN_classifier(x):
    # Get the index of the nearest neighbor
```

```
    index = find_NN(x)
    # Return its class
    return train_labels[index]
#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 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")

'''
5. Processing the full test set
 Let us apply our nearest neighbor classifier over the full data set.
 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.
 Let us predict on each test data point (and time it!)
'''

#5. Processing the full test set
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)

'''
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 ℝd , this takes
0(Nd) time.
 Fortunately, there are faster methods to perform nearest neighbor. "scikit-learn" has fast
implementations of two useful nearest neighbor data structures: the ball tree and the k-d tree.
'''

#6. Faster nearest neighbor methods
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))


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))
```