Lab 3. Nearest Neighbors classification and K-Nearest Neighbors

Nearest Neighbors (NN) Classification

Nearest Neighbor (NN) classification is a type of instance-based or memory-based learning. Unlike k-Nearest Neighbors (kNN), where you consider a fixed number of neighbors (k) for classification, NN classification specifically considers only the single nearest neighbor.

Here's how NN classification works:

1. **Distance Metric:** Choose a distance metric to measure the similarity or distance between instances in your dataset. Common metrics include Euclidean distance, Manhattan distance, or cosine similarity.
2. **Find Nearest Neighbor:** Given a new, unlabeled instance, identify the single instance in the training set that is closest to it based on the chosen distance metric.
3. **Assign Label:** Assign the label of the nearest neighbor to the new instance. In other words, the new instance is classified with the same class as its nearest neighbor.

The idea is that instances with similar features or characteristics should belong to the same class. By looking at the closest neighbor in the training set, you make a prediction for the class of the new instance.

NN classification is very simple and easy to understand. However, it can be sensitive to noise and outliers, as it relies heavily on the single nearest neighbor. It's particularly useful in situations where the decision boundaries are complex and nonlinear.

One drawback of NN classification is that it may not generalize well to the overall structure of the data, especially if the dataset is sparse or has irregular patterns. Despite its simplicity, NN classification can be quite effective in certain scenarios.

What is K-Nearest Neighbors Algorithm?

K-Nearest Neighbors is one of the most basic yet essential classification algorithms in Machine Learning. It belongs to the supervised learning domain and finds intense application in pattern recognition, data mining, and intrusion detection. It basically is a generalization of the Nearest Neighbor (NN) classification. K-Nearest Neighbors (kNN), considers a fixed number of neighbors (k) for classification while NN classification specifically considers only the single nearest neighbor.

It is widely disposable in real-life scenarios since it is non-parametric, meaning, it does not make any underlying assumptions about the distribution of data (as opposed to other algorithms such as GMM, which assume a Gaussian distribution of the given data). We are given some prior data (also called training data), which classifies coordinates into groups identified by an attribute.

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.

28x28 MNIST labeled image

```
%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')
```

```
## 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))
```

Visualizing the images of MNIST dataset

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

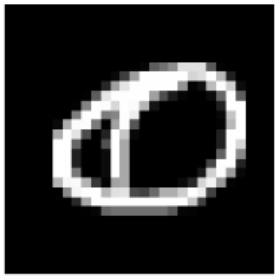
```
## Define a function that displays a digit given its vector representation
def show_digit(x):
    plt.axis('off')
    plt.imshow(x.reshape((28,28)), cmap=plt.cm.gray)
    plt.show()
    return

## 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")

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

```
Label 9
```



```
Label 0
```

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***:

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

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

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

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

Computing NN

```
## 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]
```

```
## 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")
```

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.

```
## 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)
```

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. 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. A **BallTree** is a space-partitioning data-structure that allows for finding nearest neighbors in logarithmic time. **k-d tree** is also a space-partitioning data structure for organizing points in a k-dimensional space. The key difference is that each node in a k-d tree partitions space across one plane per level of depth, leading to a binary tree.

```
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))
```

Exercises:

1. On Desktop create a new folder Lab3. Inside this folder copy the dataset (directory) MNIST and then, create a new Notebook file and introduce and run all the above cod cells.

2. 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?

3. Implement a function for KNN and compare its performance for several points (e.g. 0, 39, 100). Display the indices and the labels of the selected k neighborhoods for k=3,5,7,11.