3 Nearest Neighbor Search

Lab Objective: *The nearest neighbor problem is an optimization problem that arises in applications such as computer vision, pattern recognition, internet marketing, and data compression. Solving the problem efficiently requires the use of a k-d tree, a variation of the binary search tree. In this lab we implement a k-d tree, use it to solve the nearest neighbor problem, then apply SciPy's k-d tree object to a handwriting recognition algorithm.*

The Nearest Neighbor Problem

Suppose you move into a new city with several post offices. Since your time is valuable, you wish to know which post office is closest to your home. This is called the nearest neighbor search problem, and it has many applications.

In general, suppose that X is a collection of data, called a *training set*. Let y be any point (often called the *target* point) in the same space as the data in X. The nearest neighbor search problem determines the point in X that is closest to y. For example, in the post office problem, the set X could be addresses or latitude and longitude data for each post office in the city. Then y would be the data that represents your new home, and the task is to find the closest post office in X to y.

Problem 1. Roughly speaking, a function that measures the distance between two points in a set is called a *metric*.[a] The *Euclidean metric* measures the distance between two points in $\mathbb{R}^n$ with the familiar distance formula:

$$d(\mathbf{x},\mathbf{y}) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} = \|\mathbf{x}-\mathbf{y}\|_2$$

Write a function that accepts two 1-dimensional NumPy arrays and returns the Euclidean distance between them. Raise a `ValueError` if the arrays don't have the same number of entries. (Hint: NumPy already has some functions to help do this quickly.)

[a] Metrics and metric spaces are examined in detail in Chapter 5 of Volume I.

Consider again the post office example. One way to find out which post office is closest is to drive from home to each post office, measure the mileage, and then choose the post office that is

the closest. This is called an *exhaustive search*. More precisely, measure the distance of y to each point in X, and choose the point in X with the smallest distance from y. However, this method is inefficient, and only feasible for relatively small training sets.

Problem 2. Write a function that solves the nearest neighbor search problem by exhaustively checking all of the distances between a given point and each point in a data set. The function should take in a set of data points (as an $m \times k$ NumPy array, where each row represents one of the m points in the data set) and a single target point (as a 1-dimensional NumPy array with k entries). Return the point in the training set that is closest to the target point and its distance from the target.

The complexity of this algorithm is $O(mk)$, where k is the number of dimensions and m is the number of data points.

K-D Trees

A *k-d tree* is a special kind of binary search tree for high dimensional data (i.e., more dimensions than one). While a binary search tree excludes regions of the number line from a search until the search point is found, a k-d tree works on regions of $\mathbb{R}^k$. In other words, a regular binary search tree partitions $\mathbb{R}$, but a k-d tree partitions $\mathbb{R}^k$. So long as the data in the tree meets certain dimensionality requirements, similar efficiency gains may be made.

Recall that to search for a value in a binary search tree, start at the root, and if the value is less than the root, proceed down the left branch of the tree. If it is larger, proceed down the right branch. By doing this, a subset of values (and therefore the subtree containing those values) is excluded from the search. By eliminating this subset from consideration, there are far fewer points to search and the efficiency of the search is greatly increased.

Like a binary search tree, a k-d tree starts with a root node with a depth, or level, of 0. At the i^{th} level, the nodes to the left of a parent have a strictly lower value in the i^{th} dimension. Nodes to the right have a greater or equal value in the i^{th} dimension. At the next level, do the same for the next dimension. For example, consider data in $\mathbb{R}^3$. The root node partitions the data according to the first dimension. The children of the root partition according to the second dimension, and the grandchildren along the third. See Figures 3.1 and 3.2 for examples in $\mathbb{R}^2$ and $\mathbb{R}^3$.

As with any other data structure, the first task is to construct a node class to store data. A `KDTNode` is similar to a `BSTNode`, except it has another attribute called `axis`. The `axis` attribute indicates the dimension of $\mathbb{R}^k$ to compare points.

Problem 3. Import the `BSTNode` class from the previous lab using the following code:

```
import sys
sys.path.insert(1, "../Trees")
from trees import BSTNode
```

Write a `KDTNode` class that inherits from `BSTNode`. Modify the constructor so that a `KDTNode` can only hold a NumPy array (of type `np.ndarray`). If any other data type is given, raise a `TypeError`. Create an `axis` attribute (set it to `None` or 0 for now).

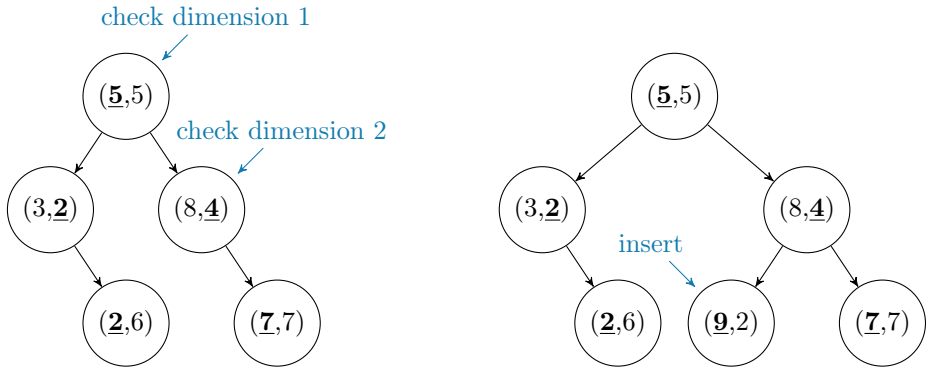

Figure 3.1: These trees illustrate an insertion of (9, 2) into a k-d tree in $\mathbb{R}^2$ loaded with the points (5, 5), (8, 4), (3, 2), (7, 7), and (2, 6), in that order. To insert the point (9, 2), find the node that will be the new node's parent. Start at the root. Since the x-coordinate of (9, 2) is greater than the x-coordinate of (5, 5), move into the right subtree of the root node, thus excluding all points (x, y) with $x < 5$. Next, compare (9, 2) to the root's right child, (8, 4). Since the y-coordinate of (9, 2) is less than the y-coordinate of (8, 4), move to the left of (8,4), thus excluding all points (x, y) with $y > 4$. Since (8,4) does not have a left child, insert (9, 2) as the left child of (8,4).

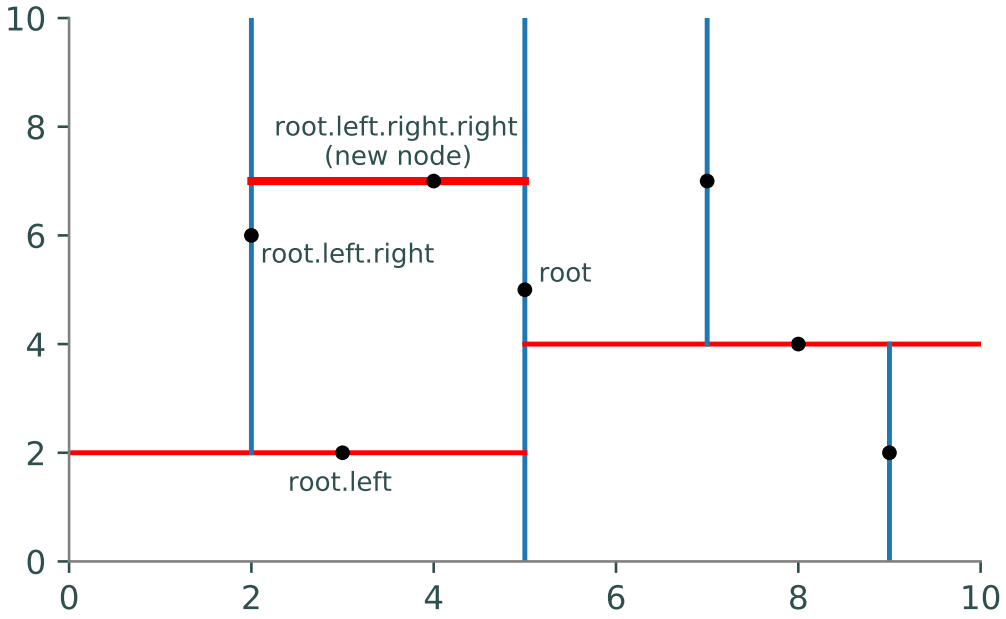

Figure 3.2: This figure is another illustration of the k-d tree from Figure 3.1 with the point (9,2) already inserted. To insert the point (4, 7), start at the root. Since the x-coordinate of (4, 7) is less than the x-coordinate of (5, 5), move into the region to the left of the middle blue line, to the root's left child, (3, 2). The y-coordinate of (4, 7) is greater than the y-coordinate of (3, 2), so move above the red line on the left, to the right child (2, 6). Now return to comparing the x-coordinates, and since $4 > 2$ and (2, 6) has no right child, install (4, 7) as the right child of (2, 6).

The major difference between a k-d tree and a binary search tree is how the data is compared at each depth level. Though the nearest neighbor problem does not need to use a `find()` method, the k-d tree version of `find()` is provided as an instructive example.

In the `find()` method, every comparison in the recursive `_step()` function compares the data of `target` and `current` based on the `axis` attribute of `current`. This way, if each existing node in the tree has the correct `axis`, the correct comparisons are made when descending through the tree.

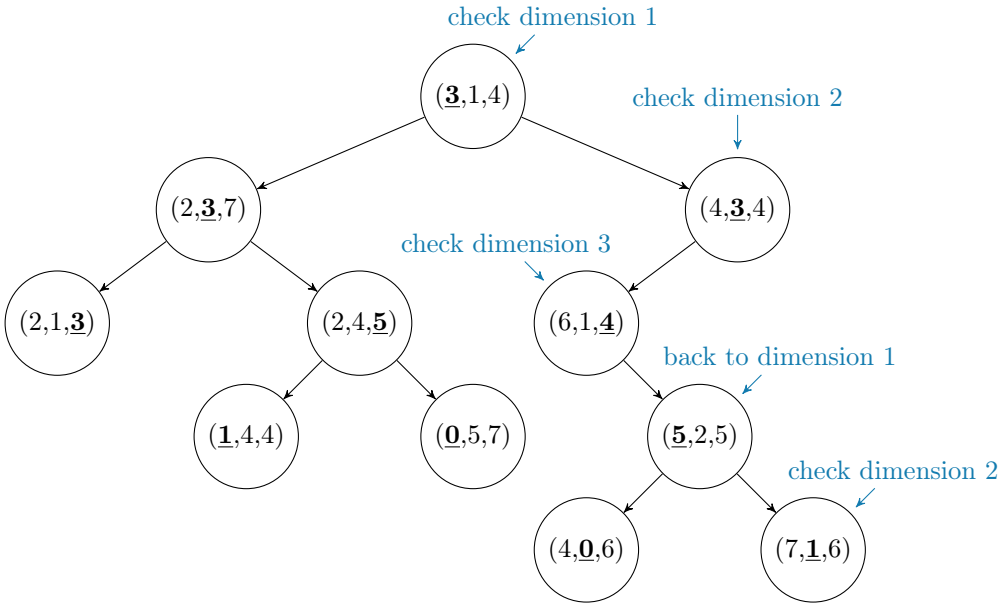

Figure 3.3: To find the point (7, 1, 6), start at the root. Since the x-coordinate of (7, 1, 6) is greater than the x-coordinate of (3, 1, 4), move to the right subtree of the root node, thus excluding all points (x, y, z) with $x < 7$. Next, compare (7, 1, 6) to the root's right child, (4, 3, 4). Since the y-coordinate of (7, 1, 6) is less than the y-coordinate of (4, 3, 4), move to the left subtree of (4, 3, 4), thus excluding all points (x, y, z) with $y > 1$. Continue in this manner until (7, 1, 6) is found.

```
import numpy as np
# Import the BST class from the previous lab.
import sys
sys.path.insert(1, "../Trees")
from trees import BST

class KDT(BST):
    """A k-dimensional binary search tree object.
    Used to solve the nearest neighbor problem efficiently.

    Attributes:
        root (KDTNode): The root node of the tree. Like all nodes in the tree,
            the root houses data as a NumPy array.
        k (int): The dimension of the tree (the 'k' of the k-d tree).
    """
```

```
    def find(self, data):
        """Return the node containing 'data'. If there is no such node in the
            tree, or if the tree is empty, raise a ValueError.
        """

        # Define a recursive function to traverse the tree.
        def _step(current):
            """Recursively step through the tree until the node containing
            'data' is found. If there is no such node, raise a Value Error.
            """
            if current is None:                     # Base case 1: dead end.
                raise ValueError(str(data) + " is not in the tree")
            elif np.allclose(data, current.value):
                return current                      # Base case 2: data found!
            elif data[current.axis] < current.value[current.axis]:
                return _step(current.left)          # Recursively search left.
            else:
                return _step(current.right)         # Recursively search right.

        # Start the recursion on the root of the tree.
        return _step(self.root)
```

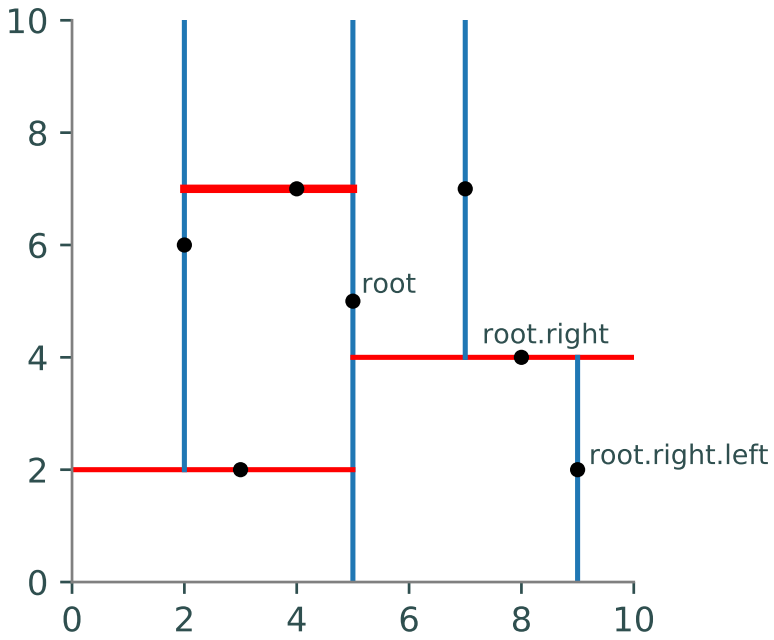

Figure 3.4: The above graph is another illustration of the k-d tree from Figure 3.1. To find the point (9, 2), start at the root. Since the x-coordinate of (9, 2) is greater than the x-coordinate of (5, 5), move into the region to the right of the middle blue line, thus excluding all points (x, y) with $x < 5$. Next, compare (9, 2) to the root's right child, (8, 4). Since the y-coordinate of (9, 2) is less than the y-coordinate of (8, 4), move below the red line on the right, thus excluding all points (x, y) with $y > 4$. The point (9, 2) is now found, since it is the left child of (8, 4).

Problem 4. Finish implementing the `KDT` class.

1. Override the `insert()` method. To insert a new node, find the node that should be the parent of the new node by recursively descending through the tree as in the `find()` method (see Figure 3.2 for a geometric example). Do not allow duplicate nodes in the tree. Note that the `k` attribute will be initialized when a k-d tree object is instantiated.

   The `axis` attribute of the new node will be one more than that axis of the parent node. If the last dimension of the data has been reached, start `axis` over at 0.

2. To prevent a user from altering the tree, disable the `remove()` method. Raise a `NotImplementedError` if the method is called, and allow it to receive any number of arguments. (Disabling the `remove()` method ensures that the k-d tree remains the same after it is created. The same k-d tree is used multiple times with different target points to solve the nearest neighbor search problem.)

Using a k-d tree to solve the nearest neighbor search problem requires some care. At first glance, it appears that a procedure similar to `find()` or `insert()` will immediately yield the result. However, this is not always the case (see Figure 3.5).

To correctly find the nearest neighbor, keep track of the target point, the current search node,

current best point, and current minimum distance. Start at the root node. Then the current search node and current best point will be root, and the current minimum distance will be the Euclidean distance from `root` to `target`. Then proceed recursively as in the `find()` method. As closer points (nearer neighbors) are found, update the appropriate variables accordingly.

Once the bottom of the tree is reached, a "close" neighbor has been found. However, this is not guaranteed to be the closest point. One way to ensure that this is the closest point is to draw a hypersphere with a radius of the current minimum distance around the candidate nearest neighbor. If this hypersphere does not intersect any of the hyperplanes that split the k-d tree, then this is indeed the closest point.

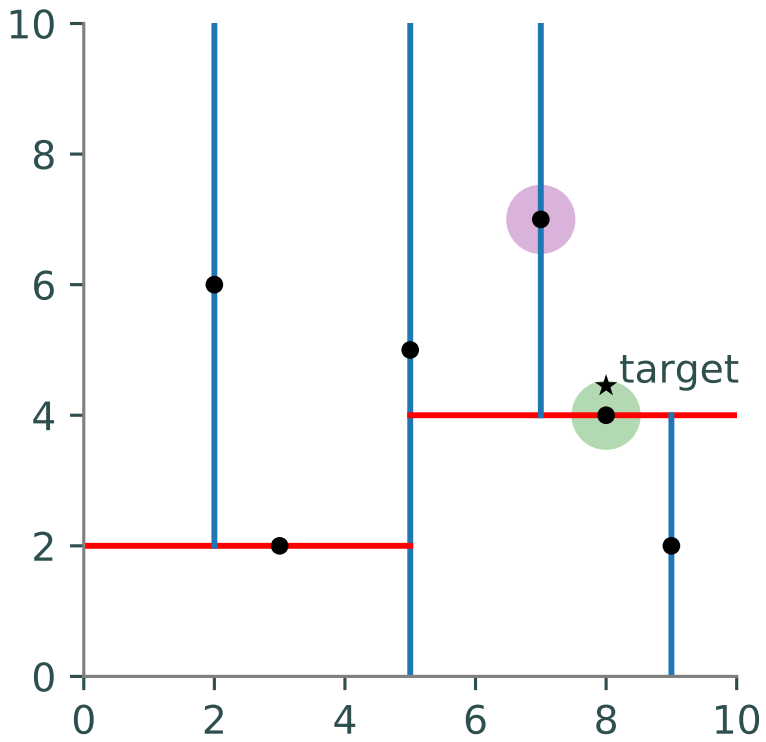

Figure 3.5: To find the point in the k-d tree of Figure 3.2 that is closest to (8, 4.5), first record the distance from the root to the target as the current minimum distance (about 3.04). Then travel down the tree to the right. The right child, (8, 4), is only 0.5 units away from the target (the green circle), so update the minimum distance. Since (8, 4) is not a leaf in the tree, the algorithm could continue down to the left child, (7, 7). However, this leaf node is much further from the target (the purple circle). To ensure that algorithm terminates correctly, check to see if the hypersphere of radius 0.5 around the current node (the green circle) intersects with any other hyperplanes. Since it does not, stop descending down the tree and conclude (correctly) that (8, 4) is the nearest neighbor.

While the correct hypersphere cannot be easily drawn, there is an equivalent procedure that has a straightforward implementation in Python. Before deciding to descend in one direction, add the minimum distance to the i^{th} entry of the target point's data, where i is the `axis` of the candidate nearest neighbor. If this sum is greater that the i^{th} entry of the current search node, then the hypersphere would necessarily intersect one of the hyperplanes drawn by the tree (why?).

The algorithm is summarized below.

Algorithm 3.1 k-d tree nearest neighbor search

```
Given a set of data and a target, build a k-d tree out of the data set.
procedure SEARCH(current, neighbor, dist)
    if current is None then                                          ▷ Base case.
        return neighbor, dist
    index ← current.axis
    if metric(current, target) < dist then
        neighbor ← current                                  ▷ Update the best estimate.
        dist ← metric(current, target)
    if target[index] < current.value[index] then                     ▷ Recurse left.
        neighbor, dist ← Search(current.left, neighbor, dist)
        if target[index] + dist ≥ current.value[index] then
            neighbor, dist ← Search(current.right, neighbor, dist)
    else                                                             ▷ Recurse right.
        neighbor, dist ← Search(current.right, neighbor, dist)
        if target[index] - dist ≤ current.value[index] then
            neighbor, dist ← Search(current.left, neighbor, dist)
    return neighbor, dist
Start SEARCH() at the root of the tree.
```

Problem 5. Use Algorithm 3.1 to write a function that solves the nearest neighbor search problem by searching through your `KDT` object. The function should take in a NumPy array of data and a NumPy array representing the target point. Return the nearest neighbor in the data set and the distance from the nearest neighbor to the target point, as in Problem 2 (be sure to return a NumPy array for the neighbor).

To test your function, use SciPy's built-in `KDTree` object. This structure behaves like the `KDT` class, but its operations are heavily optimized. To solve the nearest neighbor problem, initialize the tree with data, then "query" the tree with the target point. The `query()` method returns a tuple of the minimum distance and the index of the nearest neighbor in the data.

```
>>> from scipy.spatial import KDTree

# Initialize the tree with data (in this example, use random data).
>>> data = np.random.random((100,5))
>>> target = np.random.random(5)
>>> tree = KDTree(data)

# Query the tree and print the minimum distance.
>>> min_distance, index = tree.query(target)
>>> print(min_distance)
0.309671532426

# Print the nearest neighbor by indexing into the tree's data.
```

```
>>> print(tree.data[index])
[ 0.68001084  0.02021068  0.70421171  0.57488834  0.50492779]
```

k-Nearest Neighbors

Previously in the lab, a k-d tree was used to find the nearest neighbor of a target point. A more general problem is to find the k nearest neighbors to a point for some k (using some metric to measure "distance" between data points). The k-nearest neighbors algorithm is a machine learning model. In machine learning, a set of data points has a corresponding set of *labels*, or classifications, that specifies the category of a specific data point in the training set. A machine learning algorithm takes unlabelled data and learns how to classify it. For example, suppose a data set contains the incomes and debt levels of n individuals. Along with this data, there is a set of n data points that state whether an individual has filed for bankruptcy; these points are the labels. The goal of a machine learning model would be to correctly predict whether a new individual would go bankrupt.

Classification

In classification, the k nearest neighbors of a new point are found, and each neighbor "votes" to decide what label to give the new point. The "vote" of each neighbor is its label, or output class. The output class with the highest number of votes determines the label of the new point. See Figure 3.6 for an illustration of the algorithm.

Consider the bankruptcy example. If the 10 nearest neighbors to a new individual are found and 8 of them went bankrupt, then the algorithm would predict that the individual would also go bankrupt. On the other hand, if 7 of the nearest neighbors had not filed for bankruptcy, the algorithm would predict that the individual was at low risk for bankruptcy.

Handwriting Recognition

The problem of recognizing handwritten letters and numbers with a computer has many applications. A computer image may be thought of as a vector in $\mathbb{R}^n$, where n is the number of pixels in the image and the entries represent how bright each pixel is. If two people write the same number, the vectors representing a scanned image of those numbers are expected to be close in the Euclidean metric. This insight means that given a training set of scanned images along with correct labels, the label of a new scanned image can be confidently inferred.

Problem 6. Write a class that performs the k-nearest neighbors algorithm. The constructor should accept a NumPy array of data (the training set) and a NumPy array of corresponding labels for that data. Use SciPy's `KDTree` class to build a k-d tree from the training set in the constructor.

Write a method for this class that accepts a NumPy array of new data points and an integer `k`. Use the k-nearest neighbors algorithm to predict the label of the new data points. Return a NumPy array of the predicted labels. In the case of ties between labels, choose the label with the smaller identifier (hint: try `scipy.stats.mode()`).

To test your classifier, use the data file `PostalData.npz`. This is a collection of labeled handwritten digits that the United States Postal Service has made available to the public. This

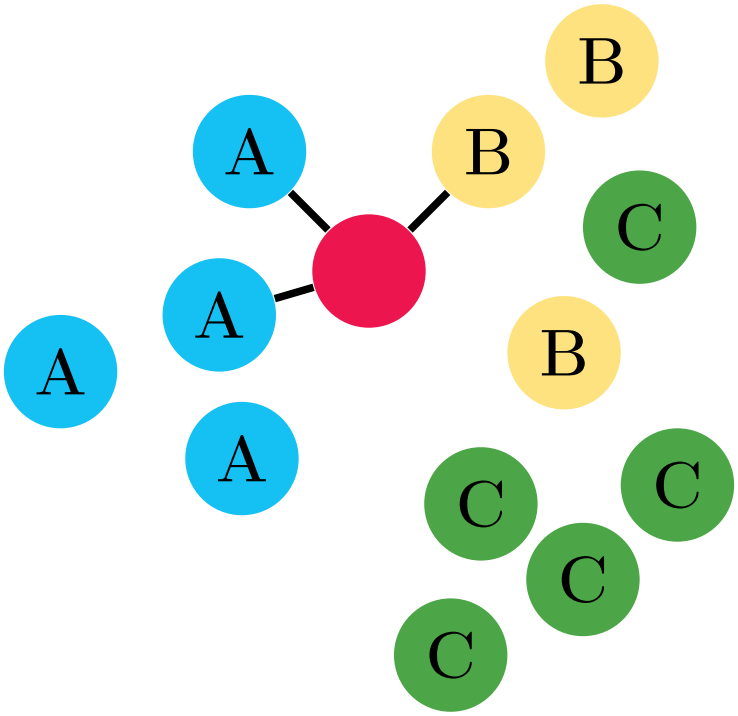

Figure 3.6: In this example, the red node is the new point that needs to be classified. Its three nearest neighbors are two type A nodes and one type B node. The red node is classified as type A since that is the most common label of its three nearest neighbors.

data set may be loaded by using the following command:

```
points, testpoints, labels, testlabels = np.load('PostalData.npz').items()
```

The first entry of each array is a name, so `points[1]` and `labels[1]` are the actual points and labels to use. Each point is an image that is represented by a flattened 28×28 matrix of pixels (so each image is a NumPy array of length 784). The corresponding label indicates the number written and is represented by an integer.

Use `labels[1]` and `points[1]` to initialize the classifier. Use `testpoints[1]` for predicting output classes. Choose a random data point from `testpoints[1]`. Plot this point with the following code:

```
import matplotlib.pyplot as plt
plt.imshow(img.reshape((28,28)), cmap="gray")
plt.show()
```

where `img` is the random data point (a NumPy array of length 784). Your plot should look similar to Figure 3.7.

Use your classifier to predict the label of this data point. Does the output match the number you plotted? Compare the output of your classifier to the corresponding label in

`testlabels[1]`. They should be the same.

A similar classification process is used by the United States Postal Service to automatically determine the zip code written or printed on a letter.

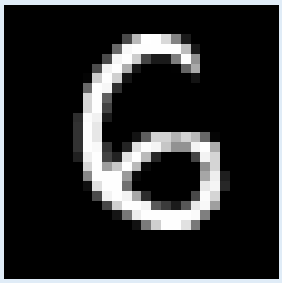

Figure 3.7: The number 6 taken from the data set.

Additional Material

sklearn

The `sklearn` package contains powerful tools for solving the nearest neighbor problem. To start nearest neighbors classification, import the `neighbors` module from `sklearn`. This module has a class for setting up a k-nearest neighbors classifier.

```
# Import the neighbors module
>>> from sklearn import neighbors

# Create an instance of a k-nearest neighbors classifier.
# 'n_neighbors' determines how many neighbors to find.
# 'weights' may be 'uniform' or 'distance.'  The 'distance' option
#     gives nearer neighbors a more weighted vote.
# 'p=2' instructs the class to use the Euclidean metric.
>>> nbrs = neighbors.KNeighborsClassifier(n_neighbors=8, weights='distance', p↩
    =2)
```

The `nbrs` object has two useful methods for classification. The first, `fit`, takes arrays of data (the training set) and labels and puts them into a k-d tree. This is used to find the k-nearest neighbors, much like the `KDT` class implemented previously in the lab.

```
# 'points' is some NumPy array of data
# 'labels' is a NumPy array of labels describing the data in points.
>>> nbrs.fit(points, labels)
```

The second method, `predict`, does a k-nearest neighbor search on the k-d tree and uses the result to attach a label to unlabeled points.

```
# 'testpoints' is an array of unlabeled points.
# Perform the search and calculate the accuracy of the classification.
>>> prediction = nbrs.predict(testpoints)
>>> nbrs.score(testpoints, testlabels)
```

More information about this module can be found at `http://scikit-learn.org/stable/modules/neighbors.html`.