Lab 6

Data Structures III: K-D Trees

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

The Nearest Neighbor Search 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 longtitude 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 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, we measure the distance of y to each point in X, and choose the point with the smallest distance. This method is ineffecient however, 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 1). 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 $\mathbb{R}^k$. So long as the data in the tree meets certain dimensionality requirements, similar efficiency gains may be made.

Recall that to search for a point in a binary search tree, we start at the root, and if the point we are searching for is less than the root we proceed down the left branch of the tree. If it is larger, we proceed down the right branch. By doing this, we exclude a region of the number line (and therefore the subtree in the opposite direction) from our search. By eliminating this region from consideration, we have far fewer points to search and the efficiency of our 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 lower value in the i^{th} dimension. Nodes to the right have a greater value in the i^{th} dimension. At the next level, we 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 Figure 6.1 for an example in $\mathbb{R}^2$.

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 tells us which dimension of $\mathbb{R}^k$ to split on.

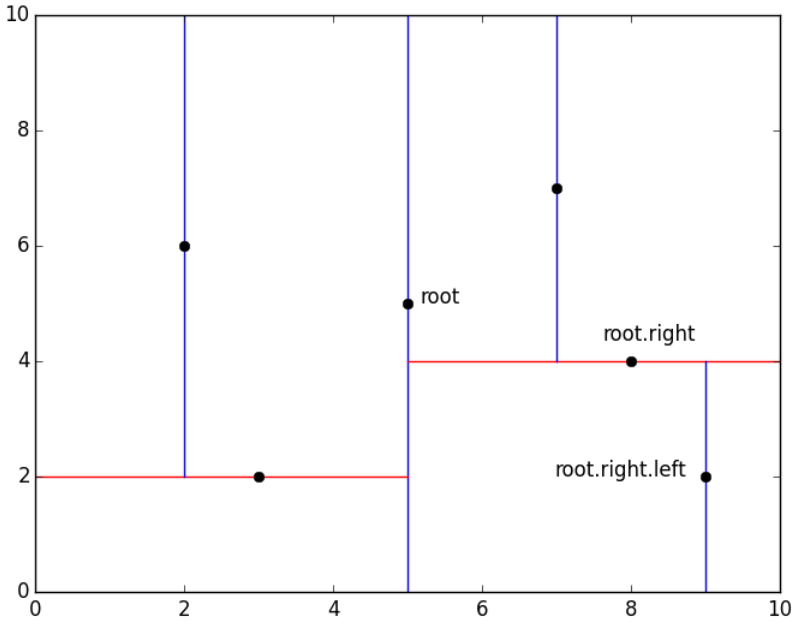

Figure 6.1: A regular binary search tree partitions $\mathbb{R}$, but a k-d tree partitions $\mathbb{R}^k$. The above graph illustrates the partition for a k-d tree loaded with the points (5, 5), (8, 4), (3, 2), (7, 7), (2, 6), and (9, 2), in that order. To find the point (9, 2), we start at the root. Since the x-coordinate of (9, 2) is greater than the x-coordinate of (5, 5), we move into the region to the right of the middle blue line, thus excluding all points (x, y) with $x < 5$. Next we 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), we move below the red line on the right, thus excluding all points (x, y) with $y > 4$. We have now found (9, 2), since it is the left child of (8, 4).

Problem 3. Copy the `BSTNode` class from the previous lab.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`. Also create an `axis` attribute (set it to `None` or 0 for now).

The major difference between a k-d tree and a binary search tree is how the data is compared at each depth level. Though we don't need to use a `find()` method in solving the nearest neighbor problem, we provide the k-d tree version of `find()` 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 as we descend through the tree.

```
# Copy or import the BST class from the previous lab.

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 other
            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)
```

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 6.2 for a geometric example). Note that the `k` attribute of the will have to be set at some point.

   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 solve the nearest neighbor search problem, we need only create the k-d tree once. Then we can use it multiple times with different target points. To prevent the 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.

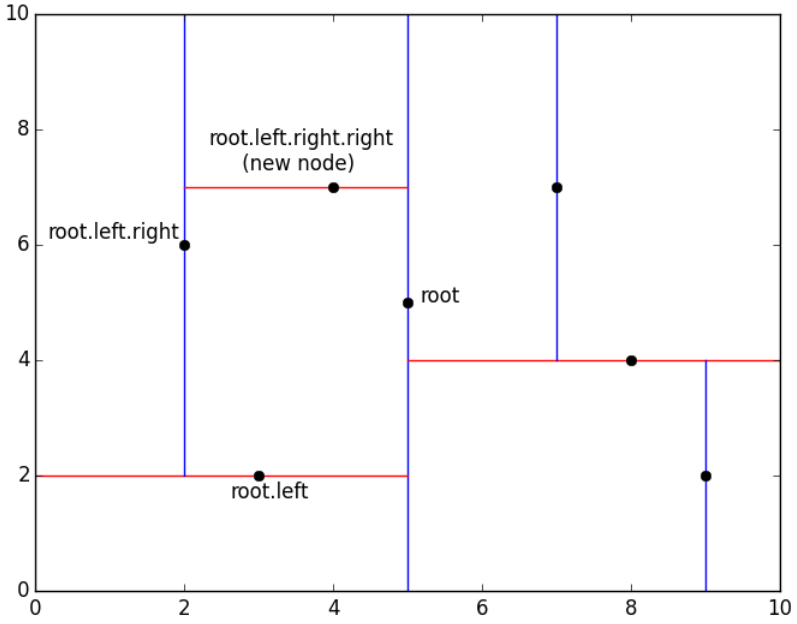

Figure 6.2: To insert the point (4, 7) into the k-d tree of figure 6.1, we find the node that will be the new node's parent. Start at the root, (5, 5). Since the x-coordinate of (4, 7) is less than the x-coordinate of (5, 5), we move into the region to the left of the middle blue line, to the root's left child, (3, 2). The y-coordinate of (4, 7) is greater than the y-coordinate of (3, 2), so we move above the red line on the left, to the right child (2, 6). Now we return to comparing the x-coordinates, and since $4 > 2$ and (2, 6) has no right child, we install (4, 7) as the right child of (2, 6).

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 6.3).

To correctly find the nearest neighbor we will 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`. We then proceed recursively as in the `find()` method. As we find better points (nearer neighbors), we update the appropriate variables accordingly.

Once we have reached the bottom of the tree, we will have a good guess for the nearest neighbor. However, we are not guaranteed to have arrived at the correct point. One way to ensure that we have arrived at the correct 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 we know that have found a best point.

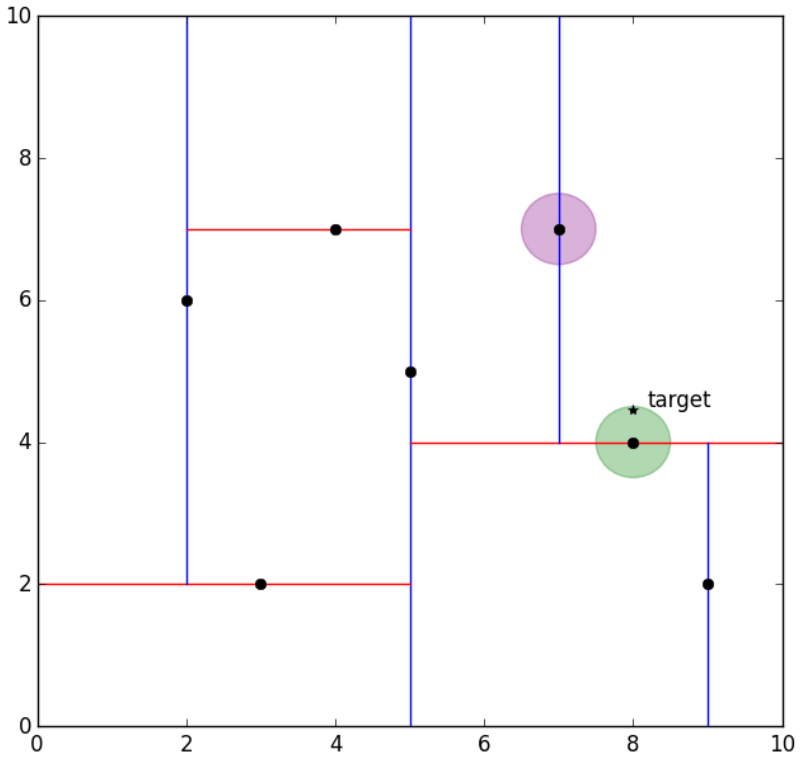

Figure 6.3: Suppose we want to find the point in the k-d tree of figure 6.2 that is closest to (8, 4.5). First we record the distance from the root to the target as the current minimum distance (about 3.04), then travel down the tree to the right. The right child, (8, 4), is only .5 units away from the target (the green circle), so we update the minimum distance. Since (8, 4) is not a leaf in the tree, we 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 we terminate the algorithm correctly we check to see if the hypersphere of radius .5 around the current node (the green circle) intersects with any other hyperplanes. Since it does not, we stop descending down the tree and conclude (correctly) that (8, 4) is the nearest neighbor.

While we can not easily draw the correct hypersphere, there is an equivalent procedure that has a straightforward implementation in Python. Before we finally decide to descend in one direction, we 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?).

We summarize the algorithm below.

Algorithm 6.1 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 6.1 to write a function that solves the nearest neighbor search problem by searching through a k-d tree (your `KDT` object). The function should take in a data set and a single 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, not a KDTNode 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 minimun distance and the index of the nearest neighbor in the data.

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

# Initialize the tree with data (in this example we 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]
```

Handwriting Recognition

Classification

Suppose that we are given a training set of data as well as a set of *labels* that describe each datum in the training set. For example, suppose that we had a training set containing the incomes and debt levels of N individuals. Along with this data, we have a set N labels that state whether the individual has filed for bankruptcy. The classification problem is to try and assign the correct label to an unlabelled data point.

k-Nearest Neighbors

In our previous work, we used a k-d tree to find the nearest neighbor of a target point. A more general problem is to find the k nearest neighbors to a point (using some metric to measure "distance" between data points). In classification, we find the k nearest neighbors, we let each neighbor "vote" to decide what label to give the new point. For example, consider the bankrupty case in the previous section. If we find the 10 nearest neighbors to a new individual, and 8 of them went bankrupt, then we would predict that the individual will also go bankrupt. On the other hand, if 7 of the nearest neighbors had not filed for bankruptcy, we would predict that the individual was at low risk for bankruptcy.

The Handwriting Recognition Problem

The problem of recognizing handwritten letters and numbers with a computer has many applications. A computer image may be thought of 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, we would expect the vectors representing a scanned image of those number to be close in the euclidean metric. This insight means that given a training set of scanned images along with correct labels, we may confidently infer the label of a new scanned image.

sklearn

The `sklearn` module contains powerful tools for solving the nearest neighbor problem. To start nearest neighbors classification, we 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 give votes to.
# 'weights' may be 'uniform' or 'distance.'  The 'distance' option
#     gives nearer neighbors more weight.
# '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`, will take arrays of data (the training set) and labels and put them into a k-d tree. This can then be used to find k-nearest neighbors, much like the `KDT` class that we implemented previously.

```
# '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`, will do a k-nearest neighbor search on the k-d tree and use the result to attach a label to unlabelled points.

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

Problem 6. The United States Postal Service has made a collection of labeled hand written digits available to the public, provided in `PostalData.npz`. We will use this data for k-nearest neighbor classification. This data set may be loaded by using the following command:

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

This contains a training set and a test set. 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. The corresponding label indicates which number was written.

Classify the testpoints with `n_neighbors` as 1, 4 or 10, and with `weights` as `'uniform'` or `'distance'`. For each trial print a report indicating how your classifier performs in terms of percentage of correct classifications. Which combination gives the most correct classifications? (Hint: define an inner function that takes in `n_neighbors` and `weights` as arguments calls the neighbors functions appropriately)

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

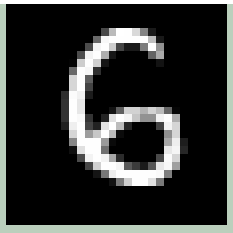

Figure 6.4: An example of the number 6 taken from the data set