3 Nearest Neighbor Search

Lab Objective: *The nearest neighbor problem is an optimization problem that arises in applications such as computer vision, internet marketing, and data compression. The problem can be solved efficiently with a k-d tree, a generalization of the binary search tree. In this lab we implement a k-d tree, use it to solve the nearest neighbor problem, then use that solution as the basis of an elementary machine learning algorithm.*

The Nearest Neighbor Problem

Let $X \subset \mathbb{R}^k$ be a collection of data, called the *training set*, and let $\mathbf{z} \in \mathbb{R}^k$, called the *target*. The *nearest neighbor search problem* is determining the point $\mathbf{x}^* \in X$ that is "closest" to $\mathbf{z}$.

For example, 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. The set X could be addresses or latitude and longitude data for each post office in the city; $\mathbf{z}$ would be the data that represents your new home. The task is to find the closest post office in $\mathbf{x} \in X$ to your home $\mathbf{z}$.

Metrics and Distance

Solving the nearest neighbor problem requires a definition for distance between $\mathbf{z}$ and elements of X. In $\mathbb{R}^k$, distance is typically defined by the *Euclidean metric.*

$$d(\mathbf{x}, \mathbf{z}) = \|\mathbf{x} - \mathbf{z}\| = \sqrt{\sum_{i=1}^{k} (x_i - z_i)^2} \tag{3.1}$$

Here $\|\cdot\|$ is the standard *Euclidean norm*, which computes vector length. In other words, $d(\mathbf{x}, \mathbf{z})$ is the length of the straight line from $\mathbf{x}$ to $\mathbf{z}$. With this notation, the nearest neighbor search problem can be written as follows.

$$\mathbf{x}^* = \underset{\mathbf{x} \in X}{\operatorname{argmin}}\, d(\mathbf{x}, \mathbf{z}) \qquad\qquad d^* = \min_{\mathbf{x} \in X} d(\mathbf{x}, \mathbf{z}) \tag{3.2}$$

NumPy and SciPy implement the Euclidean norm (and other norms) in `linalg.norm()`. This function accepts vectors or matrices. Use the `axis` argument to compute the norm along the rows or columns of a matrix: `axis=0` computes the norm of each column, and `axis=1` computes the norm of each row (see the NumPy Visual Guide).

```
>>> import numpy as np
>>> from scipy import linalg as la

>>> x0 = np.array([1, 2, 3])
>>> x1 = np.array([6, 5, 4])

# Calculate the length of the vectors x0 and x1 using the Euclidean norm.
>>> la.norm(x0)
3.7416573867739413
>>> la.norm(x1)
8.7749643873921226

# Calculate the distance between x0 and x1 using the Euclidean metric.
>>> la.norm(x0 - x1)
5.9160797830996161

>>> A = np.array([[1, 2, 3],        # or A = np.vstack((x0,x1)).
...               [6, 5, 4]])
>>> la.norm(A, axis=0)              # Calculate the norm of each column of A.
array([ 6.08276253,  5.38516481,  5.        ])
>>> la.norm(A, axis=1)              # Calculate the norm of each row of A.
array([ 3.74165739,  8.77496439])   # This is ||x0|| and ||x1||.
```

Exhaustive Search

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, measuring the distance travelled in each trip. That is, we solve (3.2) by computing $\|\mathbf{x} - \mathbf{z}\|$ for every point $\mathbf{x} \in X$. This strategy is called a *brute force* or *exhaustive search*.

Problem 1. Write a function that accepts a $m \times k$ NumPy array X (the training set) and a 1-dimensional NumPy array $\mathbf{z}$ with k entries (the target). Each of the m rows of X represents a point in $\mathbb{R}^k$ that is an element of the training set.

Solve (3.2) with an exhaustive search. Return the nearest neighbor $\mathbf{x}^*$ and its Euclidean distance d^* from the target $\mathbf{z}$.
(Hint: use array broadcasting and the `axis` argument to avoid using a loop.)

The complexity of an exhaustive search for $X \subset \mathbb{R}^k$ with m points is $O(km)$, since (3.1) is $O(k)$ and there are m norms to compute. This method works, but it is only feasible for relatively small training sets. Solving the problem with greater efficiency requires the use of a specialized data structure.

K-D Trees

A *k-d tree* is a generalized binary search tree where each node in the tree contains k-dimensional data. Just as a BST makes searching easy in $\mathbb{R}$, a k-d tree provides a way to efficiently search $\mathbb{R}^k$.

A BST creates a partition of $\mathbb{R}$: if a node contains the value x, all of the nodes in its left subtree contain values that are less than x, and the nodes of its right subtree have values that are greater than x. Similarly, a k-d tree partitions $\mathbb{R}^k$. Each node is assigned a *pivot* value $i \in \{0, 1, \ldots, k-1\}$ corresponding to the depth of the node: the root has $i = 0$, its children have $i = 1$, their children have $i = 2$, and so on. If a node has $i = k-1$, its children have $i = 0$, their children have $i = 1$, and so on. The tree is constructed such that for a node containing $\mathbf{x} = [x_0, x_1, \ldots, x_{k-1}]^\mathsf{T} \in \mathbb{R}^k$, if a node in the left subtree contains $\mathbf{y}$, then $y_i < x_i$. Conversely, if a node in the right subtree contains $\mathbf{z}$, then $x_i \leq z_i$. See Figure 3.1 for an example where $k = 3$.

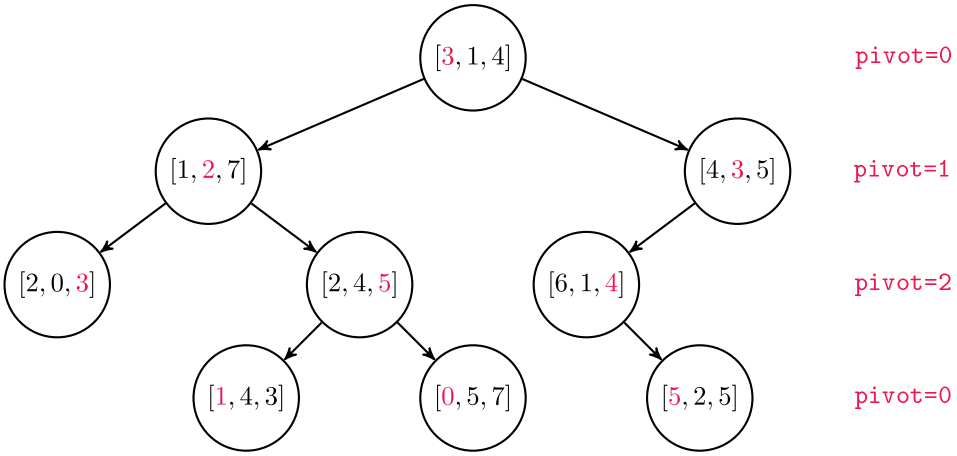


Figure 3.1: A k-d tree with $k = 3$. The root $[3, 1, 4]$ has an `pivot` of 0, so $[1, 2, 7]$ is to the left of the root because $1 < 3$, and $[4, 3, 5]$ is to the right since $3 \leq 4$. Similarly, the node $[2, 4, 5]$ has an `pivot` of 2, so $[1, 4, 3]$ is to its left since $4 < 5$ and $[0, 5, 7]$ is to its right because $5 \leq 7$. The nodes that are furthest from the root have an `pivot` of 0 because their parents have an `pivot` of $2 = k - 1$.

Problem 2. Write a `KDTNode` class whose constructor accepts a single parameter $\mathbf{x} \in \mathbb{R}^k$. If $\mathbf{x}$ is not a NumPy array (of type `np.ndarray`), raise a `TypeError`. Save $\mathbf{x}$ as an attribute called `value`, and initialize attributes `left`, `right`, and `pivot` as `None`. The `pivot` will be assigned when the node is inserted into the tree, and `left` and `right` will refer to child nodes.

Constructing the Tree

Locating Nodes

The `find()` methods for k-d trees and binary search trees are very similar. Both recursively compare the values of a target and nodes in the tree, but in a k-d tree, these values must be compared according to their `pivot` attribute. Every comparison in the recursive `_step()` function, implemented below, compares the data of `target` and `current` based on the `pivot` attribute of `current`. See Figure 3.2.

```python
class KDT:
    """A k-dimensional tree for solving the nearest neighbor problem.

    Attributes:
        root (KDTNode): the root node of the tree. Like all other nodes in
```

```
            the tree, the root has a NumPy array of shape (k,) as its value.
        k (int): the dimension of the data in the tree.
    """
    def __init__(self):
        """Initialize the root and k attributes."""
        self.root = None
        self.k = None

    def find(self, data):
        """Return the node containing the data. If there is no such node in
        the tree, or if the tree is empty, raise a ValueError.
        """
        def _step(current):
            """Recursively step through the tree until finding the node
            containing the data. If there is no such node, raise a ValueError.
            """
            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.pivot] < current.value[current.pivot]:
                return _step(current.left)          # Recursively search left.
            else:
                return _step(current.right)         # Recursively search right.

        # Start the recursive search at the root of the tree.
        return _step(self.root)
```

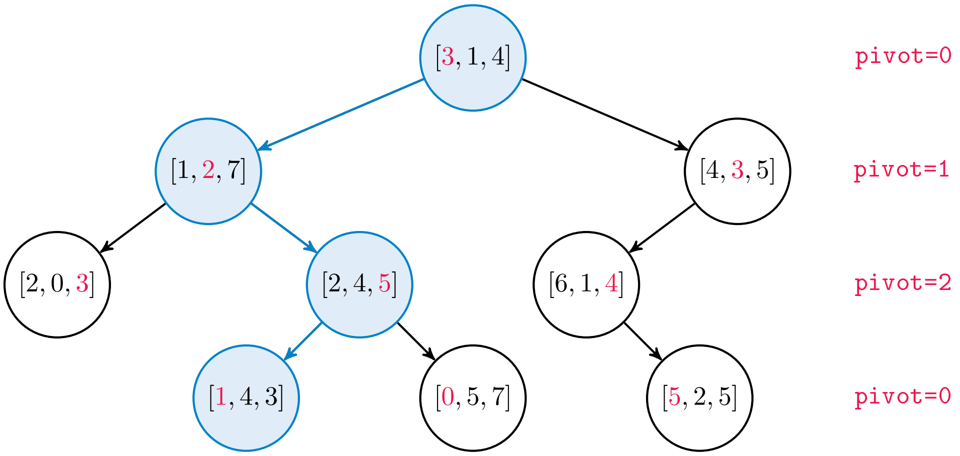
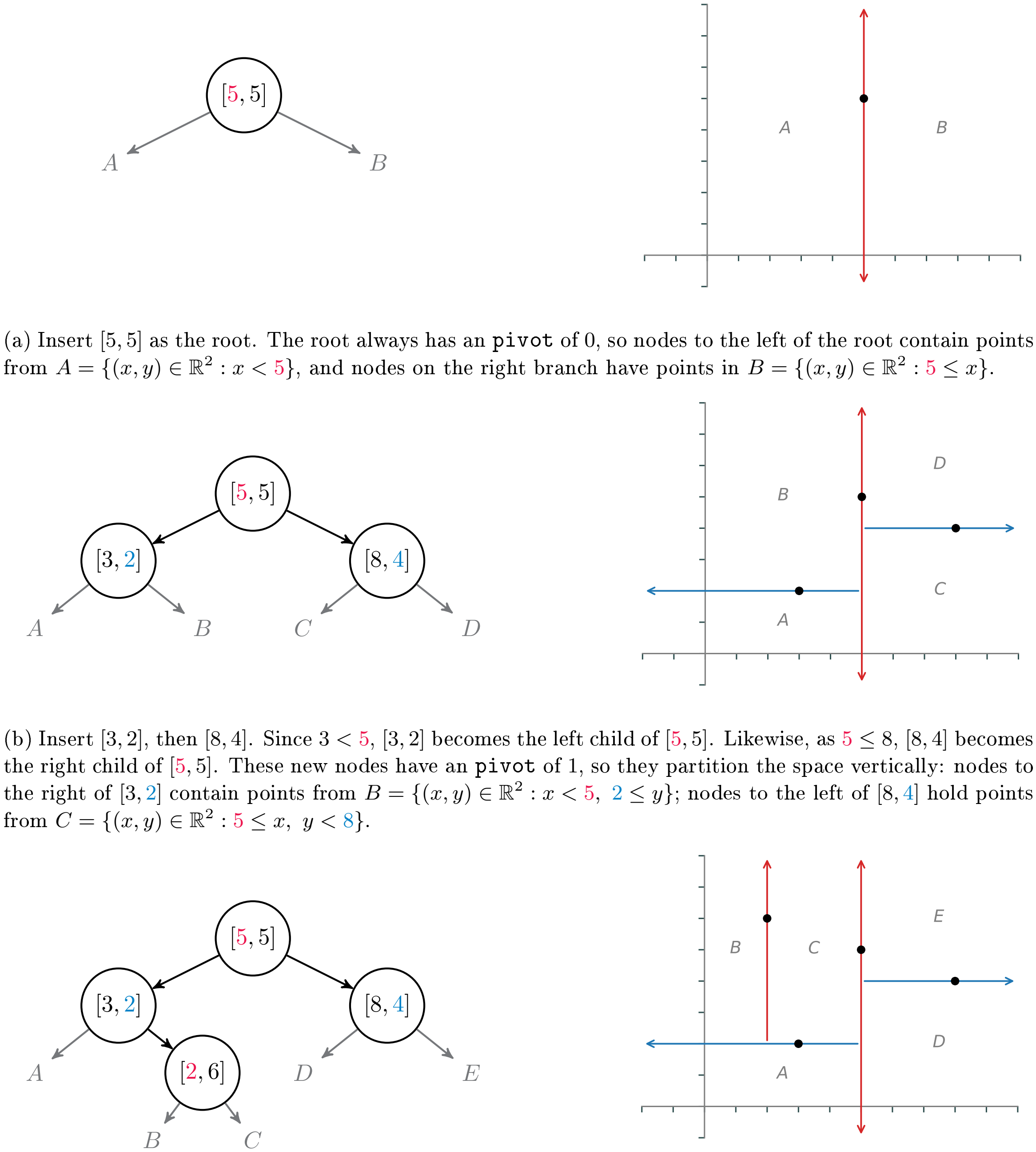


Figure 3.2: To locate the node containing $[1, 4, 3]$, start by comparing $[1, 4, 3]$ to the root $[3, 1, 4]$. The root has an `pivot` of 0, so compare the first component of the data to the first component of the root: since $1 < 3$, step left. Next, $[1, 4, 3]$ must be to the right of $[1, 2, 7]$ because $2 \leq 4$. Similarly, $[1, 4, 3]$ must be to the left of $[2, 4, 5]$ as $3 < 5$.



(a) Insert $[5,5]$ as the root. The root always has an `pivot` of 0, so nodes to the left of the root contain points from $A = \{(x,y) \in \mathbb{R}^2 : x < 5\}$, and nodes on the right branch have points in $B = \{(x,y) \in \mathbb{R}^2 : 5 \leq x\}$.

(b) Insert $[3,2]$, then $[8,4]$. Since $3 < 5$, $[3,2]$ becomes the left child of $[5,5]$. Likewise, as $5 \leq 8$, $[8,4]$ becomes the right child of $[5,5]$. These new nodes have an `pivot` of 1, so they partition the space vertically: nodes to the right of $[3,2]$ contain points from $B = \{(x,y) \in \mathbb{R}^2 : x < 5,\ 2 \leq y\}$; nodes to the left of $[8,4]$ hold points from $C = \{(x,y) \in \mathbb{R}^2 : 5 \leq x,\ y < 8\}$.

(c) Insert $[2,6]$. The `pivot` cycles back to 0 since $k = 2$, so nodes to the left of $[2,6]$ have points that lie in $B = \{(x,y) \in \mathbb{R}^2 : x < 2,\ 2 \leq y\}$ and nodes to the right store points in $C = \{(x,y) \in \mathbb{R}^2 : 2 \leq x < 5,\ 2 \leq y\}$.

Figure 3.3: As a k-d tree is constructed (left), it creates a partition of $\mathbb{R}^k$ (right) by defining separating hyperplanes that pass through the points. The more points, the finer the partition.

Inserting Nodes

To add a new node to a k-d tree, determine which existing node should be the parent of the new node by recursively stepping down the tree as in the `find()` method. Next, assign the new node as the `left` or `right` child of the parent, and set its `pivot` based on its parent's `pivot`: if the parent's `pivot` is i, the new node's `pivot` should be $i + 1$, or 0 if $i = k - 1$.

Consider again the k-d tree in Figure 3.2. To insert $[2, 3, 4]$, search the tree for $[2, 3, 4]$ until hitting an empty slot. In this case, the search steps from the root down to $[1, 4, 3]$, which has an `pivot` of 0. Then since $1 \leq 2$, the new node should be to the right of $[1, 4, 3]$. However, $[1, 4, 3]$ has no right child, so it becomes the parent of $[2, 3, 4]$. The `pivot` of the new node should therefore be 1. See Figure 3.3 for another example.

Problem 3. Write an `insert()` method for the `KDT` class that accepts a point $\mathbf{x} \in \mathbb{R}^k$.

1. If the tree is empty, create a new `KDTNode` containing $\mathbf{x}$ and set its `pivot` to 0. Assign the `root` attribute to the new node and set the `k` attribute as the length of $\mathbf{x}$. Thereafter, raise a `ValueError` if data to be inserted is not in $\mathbb{R}^k$.

2. If the tree is nonempty, create a new `KDTNode` containing $\mathbf{x}$ and find the existing node that should become its parent. Determine whether the new node will be the parent's `left` or `right` child, then link the parent to the new node accordingly. Set the `pivot` of the new node based on its parent's `pivot`.
(Hint: write a recursive function like `_step()` to find and link the parent.)

3. Do not allow duplicates in the tree: if there is already a node in the tree containing $\mathbf{x}$, raise a `ValueError`.

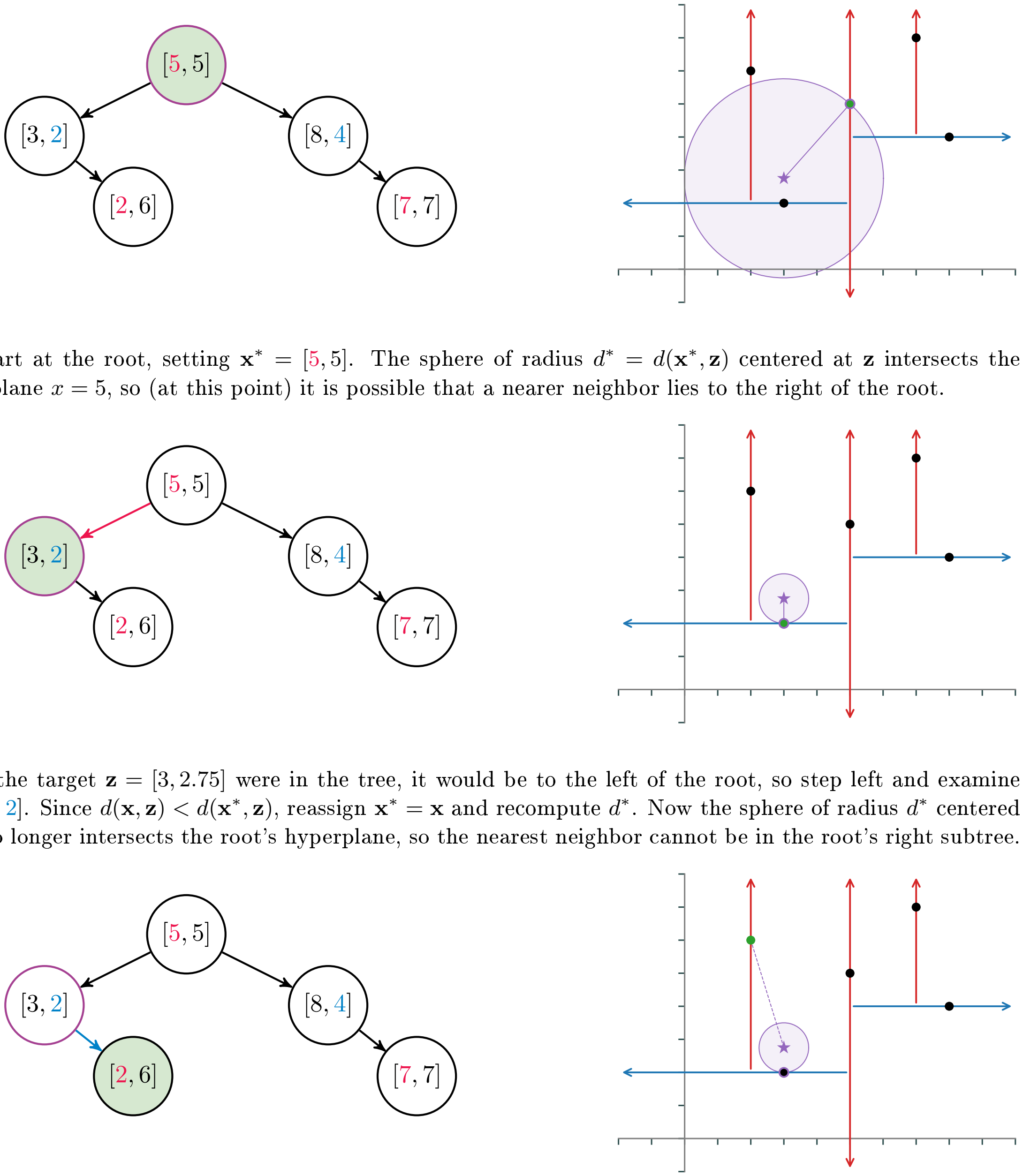
To test your method, use the `__str__()` method provided in the Additional Materials section. Try constructing the trees in Figures 3.1 and 3.3. Also check that the provided `find()` method works as expected.

Nearest Neighbor Search with K-D Trees

Given a target $\mathbf{z} \in \mathbb{R}^k$ and a k-d tree containing a set $X \subset \mathbb{R}^k$ of m points, the nearest neighbor problem can be solved by traversing the tree in a manner that is similar to the `find()` or `insert()` methods from the previous section. The advantage of this strategy over an exhaustive search is that not every $\mathbf{x} \in X$ has to be compared to $\mathbf{z}$ via (3.1); the tree structure makes it possible to rule out some elements of X without actually computing their distances to $\mathbf{z}$. The complexity is $O(k \log(m))$, a significant improvement over the $O(km)$ complexity of an exhaustive search.

To begin, set $\mathbf{x}^*$ as the value of the root and compute $d^* = d(\mathbf{x}^*, \mathbf{z})$. Starting at the root, step down through the tree as if searching for the target $\mathbf{z}$. At each step, determine if the value $\mathbf{x}$ of the current node is closer to $\mathbf{z}$ than $\mathbf{x}^*$. If it is, assign $\mathbf{x}^* = \mathbf{x}$ and recompute $d^* = d(\mathbf{x}^*, \mathbf{z})$. Continue this process until reaching a leaf node.

Next, backtrack along the search path and determine if the non-explored branch needs to be searched. To do this, check that the sphere of radius d^* centered at $\mathbf{z}$ does not intersect with the separating hyperplane defined by the current node. That is, if the separating hyperplane is further than d^* from $\mathbf{z}$, then no points on the other side of the hyperplane can possibly be the nearest neighbor. See Figure 3.4 for an example and Algorithm 3.1 for the details of the procedure.



(a) Start at the root, setting $\mathbf{x}^* = [5, 5]$. The sphere of radius $d^* = d(\mathbf{x}^*, \mathbf{z})$ centered at $\mathbf{z}$ intersects the hyperplane $x = 5$, so (at this point) it is possible that a nearer neighbor lies to the right of the root.

(b) If the target $\mathbf{z} = [3, 2.75]$ were in the tree, it would be to the left of the root, so step left and examine $\mathbf{x} = [3, 2]$. Since $d(\mathbf{x}, \mathbf{z}) < d(\mathbf{x}^*, \mathbf{z})$, reassign $\mathbf{x}^* = \mathbf{x}$ and recompute d^*. Now the sphere of radius d^* centered at $\mathbf{z}$ no longer intersects the root's hyperplane, so the nearest neighbor cannot be in the root's right subtree.

(c) Continuing the search, step right to check the point $\mathbf{x} = [2, 6]$. In this case $d(\mathbf{x}, \mathbf{z}) > d(\mathbf{x}^*, \mathbf{z})$, meaning $\mathbf{x}$ is **not** nearer to $\mathbf{z}$ than $\mathbf{x}^*$. Since $[2, 6]$ is a leaf node, retrace the search steps up the tree to check the non-searched branches. However, the sphere around $\mathbf{z}$ does not intersect any splitting hyperplanes defined by the tree, so $\mathbf{x}^*$ is guaranteed to be the nearest neighbor.

Figure 3.4: Nearest neighbor search of a k-d tree with $k = 2$. The target is $\mathbf{z} = [3, 2.75]$ and the nearest neighbor is $\mathbf{x}^* = [3, 2]$ with minimal distance $d^* = 0.75$. The tree structure allows the algorithm to eliminate $[8, 4]$ and $[7, 7]$ from consideration without computing their distance from $\mathbf{z}$.

Algorithm 3.1 k-d tree nearest neighbor search

```
procedure NEAREST NEIGHBOR SEARCH($\mathbf{z}$, root)
    procedure KDSEARCH(current, nearest, $d^*$)
        if current is None then                              ▷ Base case: dead end.
            return nearest, $d^*$
        $\mathbf{x} \leftarrow$ current.value
        $i \leftarrow$ current.pivot
        if $d(\mathbf{x}, \mathbf{z}) < d^*$ then          ▷ Check if current is closer to $\mathbf{z}$ than nearest.
            nearest $\leftarrow$ current
            $d^* \leftarrow d(\mathbf{x}, \mathbf{z})$
        if $z_i < x_i$ then                                  ▷ Search to the left.
            nearest, $d^* \leftarrow$ KDSearch(current.left, nearest, $d^*$)
            if $z_i + d^* \geq x_i$ then                     ▷ Search to the right if needed.
                nearest, $d^* \leftarrow$ KDSearch(current.right, nearest, $d^*$)
        else                                                 ▷ Search to the right.
            nearest, $d^* \leftarrow$ KDSearch(current.right, nearest, $d^*$)
            if $z_i - d^* \leq x_i$ then                     ▷ Search to the left if needed.
                nearest, $d^* \leftarrow$ KDSearch(current.left, nearest, $d^*$)
        return nearest, $d^*$
    node, $d^* \leftarrow$ KDSearch(root, root, $d$(root.value, $\mathbf{z}$))
    return node.value, $d^*$
```

Problem 4. Write a method for the KDT class that accepts a target point $\mathbf{z} \in \mathbb{R}^k$. Use Algorithm 3.1 to solve (3.2). Return the nearest neighbor $\mathbf{x}^*$ (the actual NumPy array, not the KDTNode) and its distance d^* from $\mathbf{z}$.

Compare your method to the exhaustive search in Problem 1 and to SciPy's built-in KDTree class. This structure is essentially a heavily optimized version of the KDT class. 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.

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

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

# Query the tree for the nearest neighbor and its distance from 'target'.
>>> min_distance, index = tree.query(target)
>>> print(min_distance)
0.24929868807
>>> tree.data[index]                    # Get the actual nearest neighbor.
array([ 0.26927057,  0.03160271,  0.46830759,  0.26766863,  0.63073275])
```

Achtung!

There are a few caveats to using a k-d tree for the nearest neighbor search problem.

- Constructing the tree takes time. For small enough data sets, an exhaustive search may be faster than the combined time of constructing and searching a tree. On the other hand, once the tree is constructed, it can be used for multiple nearest-neighbor queries.
- In the worst case—when the tree is completely unbalanced—the search complexity is $O(km)$ instead of $O(k\log(m))$. Fortunately, there are algorithms for constructing the tree intelligently so that it is mostly balanced, and a random insertion order usually results in a somewhat balanced tree.

K-Nearest Neighbors

The nearest neighbor algorithm provides one way to solve a common machine learning problem. In *supervised learning*, a *training set* $X \subset D$ has a corresponding set of *labels* Y that specifies a category for each element of X. For instance, X could contain financial data on m individuals, and Y could be a set of m booleans indicating which individuals have filed for bankruptcy. Supervised learning algorithms use the training data to construct a function $f : D \to Y$ that maps points to their corresponding label. In other words, the algorithm "learns" enough about the relationship between X and Y to intelligently label arbitrary elements of D. In the bankruptcy example, a person could then use their own financial data to learn whether or not they look more like someone who files for bankruptcy or someone who does not.

A *k-nearest neighbors* classifier uses a simple strategy to label an arbitrary $\mathbf{z} \in D$: find the k elements of X that are nearest to $\mathbf{z}$ (usually in terms of the Euclidean metric) and choose the most common label from those k elements as the label of $\mathbf{z}$. That is, the points in the k labeled points that are most like $\mathbf{z}$ are allowed to "vote" on how $\mathbf{z}$ should be labeled. See Figure 3.5.

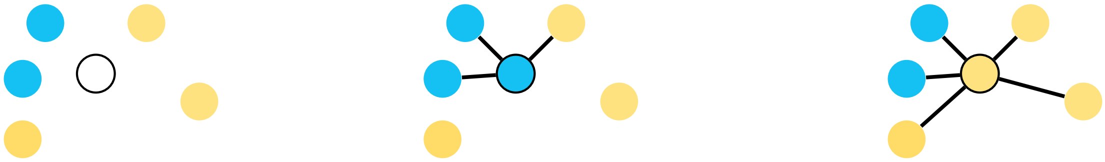


Figure 3.5: To classify the center node, determine its k-nearest neighbors and pick the most common label of the neighbors. If $k = 3$, the k nearest points are two blues and a yellow, so the center node is labeled blue. For $k = 5$, the k nearest points consists of two blues and three yellows, so the center node is labeled yellow.

Achtung!

The k in k-d tree refers to the **dimension** of the data housed in the tree, but the k in k-nearest neighbors refers to the **number of neighbors** to use in the voting scheme. Unfortunately, both names are standard.

Problem 5. Write a KNeighborsClassifier class with the following methods.

1. The constructor should accept an integer n_neighbors, the number of neighbors to include in the vote (the k in k-nearest neighbors). Save this value as an attribute.

2. fit(): accept an $m \times k$ NumPy array X (the training set) and a 1-dimensional NumPy array $\mathbf{y}$ with m entries (the training labels). As in Problems 1 and 4, each of the m rows of X represents a point in $\mathbb{R}^k$. Here y_i is the label corresponding to row i of X.

   Load a SciPy KDTree with the data in X. Save the tree and the labels as attributes.

3. predict(): accept a 1-dimensional NumPy array $\mathbf{z}$ with k entries. Query the KDTree for the n_neighbors elements of X that are nearest to $\mathbf{z}$ and return the most common label of those neighbors. If there is a tie for the most common label (such as if $k = 2$ in Figure 3.5), choose the alphanumerically smallest label.
   (Hint: use scipy.stats.mode(). The default behavior splits ties correctly.)

   To get several nearest neighbors from the tree, specify k in KDTree.query().

```
>>> data = np.random.random((100,5))    # 100 5-dimensional points.
>>> target = np.random.random(5)
>>> tree = KDTree(data)

# Query the tree for the 3 nearest neighbors.
>>> distances, indices = tree.query(target, k=3)
>>> print(indices)
[26 30 32]
```

NOTE

The format of the KNeighborsClassifier in Problem 5 conforms to the style of *scikit-learn* (sklearn), a large machine learning library in Python. In fact, scikit-learn has a class called sklearn.neighbors.KNeighborsClassifier that is a more robust version of the class from Problem 5. See http://scikit-learn.org/stable/modules/neighbors.html for more tools from scikit-learn for solving the nearest neighbor problem in the context of machine learning.

Handwriting Recognition

Computer vision is a challenging area of artificial intelligence that focuses on autonomously interpreting images. Perhaps the simplest computer vision problem is that of translating images into text. Roughly speaking, computers store grayscale images as $M \times N$ arrays of pixel brightness values: 0 corresponds to black, and 255 to white. Flattening out such an array yields a vector in $\mathbb{R}^{MN}$. Given some images of characters with labels (assigned by humans), a k-nearest neighbor classifier can intelligently decide what character the image represents.

Problem 6. The file mnist_subset.npz contains part of the MNIST dataset,[a] a collection of 28 × 28 images of handwritten digits and their labels. The data is split into four parts.

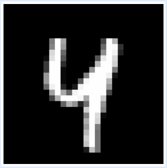
- X_train: A 3000 × 728 matrix, the training set. Each of the 3000 rows is a flattened 28 × 28 image to be used in training the classifier.
- y_train: A 1-dimensional NumPy array with 3000 entries. The entries are integers from 0 to 9, the labels corresponding to the images in X_train.
- X_test: A 500 × 728 matrix of 500 images to classify.
- y_test: A 1-dimensional NumPy array with 500 entries. These are the labels corresponding to X_test, the "right answers" that the classifier will try to guess.

The following code uses np.load() to extract the data.

```
>>> data = np.load("mnist_subset.npz")
>>> X_train = data["X_train"].astype(np.float)          # Training data
>>> y_train = data["y_train"]                           # Training labels
>>> X_test = data["X_test"].astype(np.float)            # Test data
>>> y_test = data["y_test"]                             # Test labels
```

To visualize one of the images, reshape it as a 28 × 28 array and use plt.imshow().

```
>>> from matplotlib import pyplot as plt
>>> plt.imshow(X_test[0].reshape((28,28)), cmap="gray")
>>> plt.show()
```



Write a function than accepts an integer n_neighbors. Load a classifier from Problem 5 with the data X_train and the corresponding labels y_train. Use the classifier to predict the labels of each image in X_test. Return the classification accuracy, the percentage of predictions that match y_test. The accuracy should be at least 90% using 4 nearest neighbors.

[a] See http://yann.lecun.com/exdb/mnist/.

Note

The k-nearest neighbors algorithm is **not** the best machine learning algorithm for this problem, but it is a good starting point because of its simplicity. In fact, k-nearest neighbors is often used as a baseline to compare against more complicated machine learning techniques.

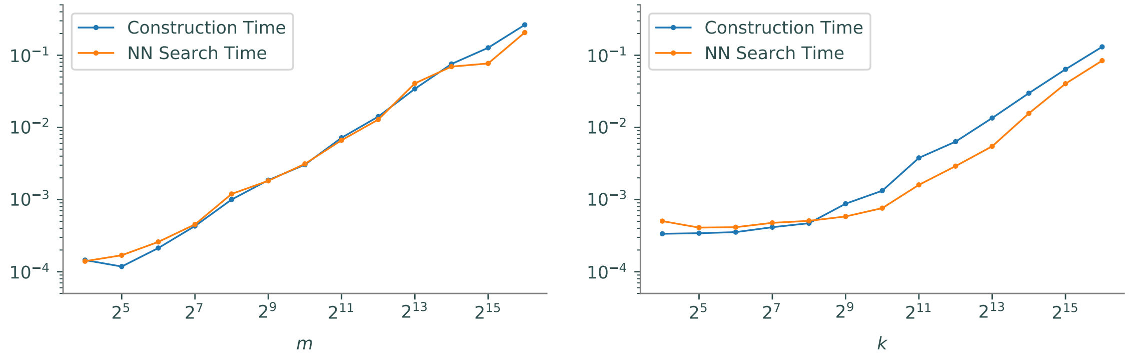
Additional Material

Ball Trees

The nearest neighbor problem can also be solved efficiently with a *ball tree*, another space-partitioning data structure. Instead of separating $\mathbb{R}^k$ by hyperplanes, a ball tree uses nested hyperspheres to split up the space. Since the partitioning scheme is different, a nearest neighbor search through a ball tree is more efficient than the k-d tree search for some data sets. See `https://en.wikipedia.org/wiki/Ball_tree` for more details.

The Curse of Dimensionality

The *curse of dimensionality* refers to a phenomena that occurs when dealing with high-dimensional data: the computational cost of an algorithm increases much more rapidly as the dimension increases than it does when the number of points increases. This problem occurs in many other areas involving multi-dimensional data, but it is quite apparent in a nearest neighbor search.



(a) Fixing k and increasing m leads to consistent growth in execution time.

(b) For fixed m, the times takes a sharp upturn around $k = 2^9$ relative to previous growth rates.

Figure 3.6: Construction and nearest neighbor search times for a k-d tree with a $m \times k$ training set.

See `https://en.wikipedia.org/wiki/Curse_of_dimensionality` for more examples. One way to avoid the curse of dimensionality is via *dimension reduction*, a process usually based on the singular value decomposition (SVD) that projects data into a lower-dimensional space.

Tiebreaker Strategies

As mentioned in Problem 5, the majority voting scheme in the k-nearest neighbor algorithm can often result in a tie. Breaking the tie intelligently is a science unto itself, but here are a few common strategies.

1. For binary classification (meaning there are only two labels), choose an odd k to avoid a tie in the first place.
2. Redo the search with $k - 1$ neighbors, repeating as needed until $k = 1$.
3. Choose the label that appears more frequently in the test set.
4. Choose randomly among the labels that are tied for most common.

Additional Code

The following code creates a string representation for the KDT class. Use this to test Problem 3.

```
class KDT:
    # ...
    def __str__(self):
        """String representation: a hierarchical list of nodes and their axes.

        Example:                           'KDT(k=2)
                    [5,5]                   [5 5]   pivot = 0
                    /   \                   [3 2]   pivot = 1
                [3,2]   [8,4]               [8 4]   pivot = 1
                    \       \               [2 6]   pivot = 0
                    [2,6]   [7,5]           [7 5]   pivot = 0'
        """
        if self.root is None:
            return "Empty KDT"
        nodes, strs = [self.root], []
        while nodes:
            current = nodes.pop(0)
            strs.append("{}\tpivot = {}".format(current.value, current.pivot))
            for child in [current.left, current.right]:
                if child:
                    nodes.append(child)
        return "KDT(k={})\n".format(self.k) + "\n".join(strs)
```