DSC 190 - Homework 03

Due: Thursday, February 4

Write your solutions to the following problems by either typing them up or handwriting them on another piece of paper. Unless otherwise noted by the problem's instructions, show your work or provide some justification for your answer. Homeworks are due via Gradescope on Thursday at 11:59 p.m.

Problem 1.

Suppose you initialize an LSH data for storing points in $\mathbb{R}^2$ by choosing $\ell = 1$, $w = 1$, and $k = 2$ random unit vectors (shown below):

$$\vec{u}^{(1)} = \left(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right)^T$$
$$\vec{u}^{(2)} = \left(\frac{\sqrt{2}}{\sqrt{3}}, -\frac{1}{\sqrt{3}}\right)^T$$

You insert the following points into the LSH data structure one-by-one:

x	y
-1.5	1.5
2.5	-1.5
-0.5	0.5
1.5	-0.5
-1.5	-1.5
0.5	0.5

Upon querying the point (-2, -2), what other point(s) will be in the same cell? Do your calculations by hand and show your work (though you can use code to check your answer, if you wish).

Solution: The only other point in the same cell will be (-1.5, -1.5). To see that these two points are in the same cell, compute the cell id by projecting each point onto the two unit vectors. For instance, the projection of $(-1.5, -1.5)$ onto $\vec{u}^{(1)}$ is

$$(-1.5, 1.5) \cdot (1/\sqrt{2}, 1/\sqrt{2}) = -2.12$$

The floor of this is -3. Likewise, the projection of this point onto the second unit vector is -0.35, the floor of which is -1. Therefore the cell id is (-3, -1). The same cell id will be found for the query point.

Programming Problem 1.

In a file named `knn_query.py`, implement a function named `knn_query(node, p, k)` which accepts three arguments:

- `node`: A `KDInternalNode` object representing the root of a k-d tree. (See lecture for the implementation of `KDInternalNode`.)
- `p`: A numpy array representing a query point.
- `k`: An integer representing the number of nearest neighbors to find.

Your function should return two things:

- A numpy array of distances to the kth nearest neighbors, in sorted order from smallest to largest.

- A $k \times d$ numpy array of the k nearest neighbors in order of distance to the query point. Each row of this array should represent a point. In the case of a tie (two points at the same distance), break the tie arbitrarily.

In the corner case that there are fewer than k points in the tree, simply return all of the points.

Internally, use a heap to store the k closest points. Your heap should never get larger than $k + 1$ elements.

Example:

```
>>> data = np.array([
        [1, 2, 3],
        [4, 2, 1],
        [1, 1, 1],
        [7, 5, 5],
        [3, 2, 0]
    ])
>>> p = np.array([1, 1, 0])
>>> root = build_kd_tree(data) # implemented in lecture
>>> knn_query(root, p, k=3)
(array([1.        , 2.23606798, 3.16227766]),
 array([[1, 1, 1],
        [3, 2, 0],
        [1, 2, 3]]))
```

Solution: First we implement a helper class to keep track of the k smallest things inserted into a container. This class wraps `MaxHeap` from a previous week:

```
class KSmallest:

    def __init__(self, k):
        self.k = k
        self.heap = MaxHeap()

    def insert(self, key):
        if len(self.heap.keys) < self.k or key < self.heap.max():
            self.heap.insert(key)

        if len(self.heap.keys) > self.k:
            self.heap.pop_max()

    def as_list(self):
        return list(self.heap.keys)

    def top(self):
        return self.heap.max()
```

Next, we modify `nn_query` to perform a k-NN query. In the original implementation, we only keep track of the nearest neighbor to the query point. Now we keep track of the closest k neighbors by passing around a heap (wrapped inside of an instance of `KSmallest`. When we find a leaf node, we perform a brute-force search to find the k closest neighbors and insert them one-by-one into the instance of `KSmallest`. We only explore a branch if the distance to the boundary is less than the largest distance among the k smallest distances found so far.

```
def nn_query_helper(node, p, k, k_closest):
```

```
    if isinstance(node, np.ndarray):
        candidates = brute_force_nn_search(node, p, k=k)
        for point, distance in zip(*candidates):
            k_closest.insert((distance, point))
    else:
        # print('\t' * tabs, f'is dimension {node.dimension} >= {node.threshold}?')
        # find the most likely branch
        if p[node.dimension] >= node.threshold:
            most_likely_branch, other_branch = node.right, node.left
        else:
            most_likely_branch, other_branch = node.left, node.right

        # compute distance to boundary
        distance_to_boundary = abs(p[node.dimension] - node.threshold)

        # find nn within most likely branch
        nn_query_helper(most_likely_branch, p, k=k, k_closest=k_closest)

        # check the other branch, but only if necessary
        if distance_to_boundary < k_closest.top()[0]:
            nn_query_helper(other_branch, p, k=k, k_closest=k_closest)


def nn_query(node, p, k=1):
    k_closest = KSmallest(k)
    nn_query_helper(node, p, k, k_closest)
    distances, points = list(zip(*k_closest.as_list()))
    distances = np.array(distances)
    points = np.vstack(points)
    sort_ix = np.argsort(distances)
    return distances[sort_ix], points[sort_ix]
```