THE NEAREST NEIGHBOUR PROBLEM

An Algorithmic Approach

by Nayan Saxena & Alexander Shum

Introduction

The nearest neighbour search problem (NNS) is a statistical classification problem that arises in various fields of applied computing especially Machine Learning. The primary focus of this problem is to search for the proximal or most similar point to a given point, and several different variations of this problem like the k-Nearest Neighbour Problem (kNN) have played a significant role in fields like computer vision and speech recognition.

Nearest Neighbour Search is used for Image Completion

Through this paper we propose algorithms and corresponding data structures to successfully perform nearest neighbour search to reasonably higher dimensions (≤ 20)– since for dimensionality more than 20 no data structure is known that is useful in practice (Kibriya and Frank, 2007)[1]. We also use statistical approaches and methods like amortised cost to examine in great detail the efficiency and correctness of various proposed implementations that provide a solution to this problem.

Problem Statement

Let $\mathcal{S} \subseteq \mathbb{R}^k$ be a finite set of points where each point $\mathbf{x} \in \mathcal{S}$ is in k dimensional space. Given another point $\mathbf{y} \in \mathbb{R}^k$ we intend to find the closest point in $\mathcal{S}$ to $\mathbf{y}$.

Instead of the commonly used Euclidean Norm to define "closeness" between two points, we broaden scope of the problem by using L_p norm – a natural generalization of the p-norms for finite-dimensional vector spaces for $p > 1 \in \mathbb{R}$ given by,

$$d(\mathbf{x}, \mathbf{y}, p) = \left(\sum_{i=1}^{k} |(x_i - y_i)|^p \right)^{\frac{1}{p}}$$

Proposed Abstract Data Type

Let $\mathcal{S} \subseteq \mathbb{R}^k$ be a finite set of points where each point $\mathbf{x} \in \mathcal{S}$ is in k dimensional space and let $p > 1 \in \mathbb{R}$. We propose an ADT denoted by $\mathcal{D}$ that stores points from set $\mathcal{S}$ which supports the following operations,

i) `INSERT(D,x)` : Insert k-dimensional point `x` into `D`

ii) `REMOVE(D,x)` : Remove k-dimensional point `x` from `D`

iii) `NEAREST_NEIGHBOUR(D,x,p)` : Find the point `y` in `D` that is nearest to a given input `x` using the p norm. Sometimes, the algorithm is allowed to return an approximate guess whose distance from `x` is at most some error factor $\epsilon > 1$ times the distance from `x` to its true nearest points.

iv) `K_NEAREST_NEIGHBOURS(D,x,k,p)` Find the first k points in `D` that are nearest to a given input `x` using the p-norm. Sometimes, the algorithm is allowed to return approximate guesses whose distance from `x` is at most some error factor $\epsilon > 1$ times the distance from `x` to its true nearest points.

1 Locality-Sensitive Hashing

This algorithm is based on methods outlined in a seminal paper by Indyk where LSH was proposed as a more efficient solution to NNS especially in higher dimensions. (Indyk & Motwani, 1997)[2].

LSH is an algorithmic technique that hashes similar input items into the same "buckets" with high probability. Since similar items end up in the same buckets, this technique can be used for data clustering and nearest neighbor search. It differs from conventional hashing techniques in that hash collisions are maximized, not minimized.

Algorithm pseudocode along with a basic outline of a Python implementation is provided below.

Algorithm Pseudocode

Let $\mathcal{S} \subseteq \mathbb{R}^d$ be a finite set of points where each point $\mathbf{x} \in \mathcal{S}$ is in d dimensional space such that $|\mathcal{S}| = n$ and $p > 1 \in \mathbb{R}$ be the norm.

We use $\mathcal{H}$ to denote a family of hash functions from which m hash functions are chosen such that $g_i = (h_{1i}, \ldots, h_{mi})$. A total of q such functions $g_1 \ldots g_q$ with their corresponding hash buckets are used to to store these n points. In our implementation we use the method of random projection to obtain these q hashes.

Algorithm 1 Algorithm for Insertion

```
procedure INSERT(LSH,x)
new_hash ← Hash signature array for x using random projection, h(x) = sgn(x · v) for a random plane v
indices ← [0,...,len(new_hash)-1]
    for table ∈ hash_tables do
        for i, h in tuples of (indices,new_hash) do
            if h in table then table.append(i)
            else  table[h] = [i]
```

Algorithm 2 Algorithm for Deletion

```
procedure DELETE(LSH,x)
hash ← Hash signature array for x using random projection, h(x) = sgn(x · v) for a random plane v
indices ← [0,...,len(hash)-1]
    for table ∈ hash_tables do
        for i, h in tuples of (indices,hash) do
            if h in hash_tables and length(h) > 1 then table.remove(i)
            else   Delete h from table
```

Algorithm 3 Algorithm for Nearest Neighbour Search

```
procedure NEAREST_NEIGHBOUR(LSH,x,p)
hash ← Hash signature array for x using random projection
Set value of ϵ ≥ 1                                   ▷ Fix Search Approximation Error
indices ← [0,...,len(hash)-1]
    for table ∈ hash_tables do
        for i, h in tuples of (indices[j],hash[j]), j ∈ ℕ do
            if h in hash_tables then
                for point y in points from bucket at h do if  d(x, y, p) ≤ ϵ return y
```

Algorithm 4 Algorithm for $k-$Nearest Neighbour Search

procedure K-NEAREST_NEIGHBOUR(LSH,**x**,K,P)
hash ← Hash signature array for **x** using random projection
candidates ← []
Set value of $\epsilon \geq 1$ ▷ Fix Search Approximation Error
indices ← [0, . . . ,len(hash)-1]
    for table ∈ hash_tables **do**
        for i, h in tuples of (indices[j],hash[j]), $j \in \mathbb{N}$ **do**
            if h in hash_tables and
                for point y in points from bucket at h **do**
                    if $d(x, y, p) \leq \epsilon$ **then then** candidates.append(y)

    Sort candidates by d(x,candidate,p)
return candidates[:k]

Algorithm Implementation in Python

```
import numpy as np
import operator

def norm(y, x, p):
    """Function to calculate the p-norm of 2 different points,
    if p > 20, will just take the infinity norm instead,
    this assumes y, x have the same dimension, and p > 1"""
    d = 0
    if p < 20:
        for i in range(len(y)):
            d += abs(x[i] - y[i]) ** p
        d = d ** (1 / p)
        return d
    else:
        for j in range(len(x)):
            if d < abs(x[j] - y[j]):
                d = abs(x[j] - y[j])
        return d


class HashTable:
    def __init__(self, hash_len, dim):
        self.hash_table = {}
        self.hash_len = hash_len
        self.rand_proj = np.random.randn(dim, hash_len)

    def build_table(self, data):
        data_hashes = self.hash_fun(data)
        return self.fill_table(self.hash_table, data_hashes)

    def add_new_entry(self, new_entry, n_rows):
        new_hash = self.hash_fun(new_entry)
        return self.fill_table(self.hash_table, new_hash, n_rows)

    def remove_entry(self, entry, n_rows):
        hashes = self.hash_fun(entry)
        indices = list(range(n_rows, n_rows + len(hashes)))
```

```
        for i, h in zip(indices, hashes):
            if h in self.hash_table and len(h)>1:
                self.hash_table[h].remove(i)
            else:
                del self.hash_table[h]

    def hash_fun(self, inp_vector):
        binaries = np.dot(inp_vector, self.rand_proj) > 0
        return [int("".join((binary * 1).astype(str)), 2)**2 for binary in binaries]

    def fill_table(self,dct, hashes, n_rows=0):
        indices = list(range(n_rows, n_rows + len(hashes)))
        for i, h in zip(indices, hashes):
            if h in dct:
                dct[h].append(i)
            else:
                dct[h] = [i]

class LSH:
    def __init__(self, data, hash_len, num_tables=1):
        self.n_rows, self.dims = data.shape
        self.hash_len = hash_len
        self.tables = []
        for i in range(num_tables):
            t = HashTable(hash_len, self.dims)
            self.tables.append(t)
            self.tables[i].build_table(data)

    def insert(self, x):
        for t in self.tables:
            t.add_new_entry(x, self.n_rows)
        self.n_rows += x.shape[0]

    def delete(self, x):
        for t in self.tables:
            t.remove_entry(x, self.n_rows)
        self.n_rows -= x.shape[0]


    def nearest_neighbours(self, x, p):
        """ find the closest point to x according to the supplied metric """

        candidates = set()

        for point, table in self.tables:
            matches = table[self.hash_fun(point)]
            candidates.update(matches)

        # rerank candidates
        candidates = [(ix, norm(x, self.points[ix], p)) for ix in candidates]
        candidates.sort(key=operator.itemgetter(1))

        return candidates[0]
```

```
    def k_nearest_neighbours(self, x, p, k):
        """ find the k closest indexed points to x according to the supplied metric """

        candidates = set()

        for point, table in self.tables:
            matches = table[self.hash_fun(point)]
            candidates.update(matches)

        # rerank candidates
        candidates = [(ix, norm(x, self.points[ix], p)) for ix in candidates]
        candidates.sort(key=operator.itemgetter(1))

        return candidates[:k]
```

2 k-d tree

This algorithm is based on Bentley's original paper when he invented the K-d tree data structure. (Bentley, 1975)[3] Techniques for similarity search in k-d trees were further developed and are based on a seminal paper by Friedman & Bentley in which techniques were developed to perform NNS (Freidman, 1977)[4].

The node object stored by the k-d tree assumed to store the its depth in the tree, the median value use to partition the data, the number of leave nodes at or this node, the point stored at it, the number of leaf nodes pointers towards the left, right and parent nodes.

Root node will have parent to be $NULL$, and leaf nodes have left, right and the median value to be $NULL$.

Algorithm pseudocode along with a basic outline of a Python implementation is provided below.

Algorithm Pseudocode

Algorithm 5 Construction of K-d tree

```
1: procedure BUILD_K-D_TREE(data)
2:     dimension ← size(data[0])
3:     if size(data) == 1 then
4:         Build root node, and let it pointed by this.root, stores the only data point, with depth = 0
5:     else
6:         Calculate the median of the data at dimension depth  (mod dimension)
7:         Or, data[median][depth (mod dimension)]
8:         Calculate mean point
9:         Build the node, store the depth, and mean point
10:        Recursively build the left subtree with data[:median]
11:        Recursively build the right subtree with data[median:]
```

The mean point is needed, as it is essential for the nearest neighbour and k nearest neighbours queries.

Algorithm 6 Insertion of a new data point in $K-d\ tree$

procedure INSERT(kd_tree, x)
    Traverse down the tree to a leaf node that stores a point most similar to x
    Update all nodes' mean point and number of leaf nodes below it in between that is reached
    $axis \leftarrow leaf\_node.depth \pmod{tree.kd\_dimension}$
    if $x[axis] < leaf\_node.point[axis]$ **then**
        Build left node for $leaf\_node$ and store x
        Build right node for $leaf\_node$ and store $leaf\_node.point$
    else
        Build left node for $leaf\_node$ and store $leaf\_node.point$
        Build right node for $leaf\_node$ and store x
        $leaf\_node.mid = leaf\_node.right.point[axis]$
        $leaf\_node.num\_of\_leafs \leftarrow 2$ t
        $leaf\_node.point \leftarrow \text{mean}(leaf\_node.left.point, leaf\_node.right.point)$

Unless $n >> m$, where n is the number of leaf nodes or data points, and m is the number of new data points, it is not recommended to insert new data points. In such situation performing insertions will break the balance of the tree and reduce the performance of the data structure. Instead it is better to extract all the data points and build a new tree with the new data points and the extracted data points.

Algorithm 7 Deletion of a data point in K-d tree

procedure DELETE(kd_tree, x)
    Traverse through the to find the leaf node that stores x
    if cannot find the node that stores x **then**
        end algorithm
    Traverse through again to same leaf node again
    update all nodes' mean point and number of leaf nodes below that is reached
    $prev\_node \leftarrow leaf\_node.parent$
    Delete $leaf\_node$
    Repair at $prev\_node$

Repairing the tree is for the purpose of the tree maintains its necessary properties and ideally stays balance. Repairing at a node refers to extracting all the points stores at leaf nodes rooted at the original node, delete every node below the original node and itself and finally rebuild the subtree using the extracted points.

Algorithm for finding the k nearest neighbours will be most similar to finding the nearest neighbour with minor variations– both of which are shown in the following pages.

Algorithm 8 Algorithm for finding the a point's nearest neighbour in the K-d tree

procedure NEAREST_NEIGHBOUR(kd_tree, x, p)
    $curr\_min \leftarrow \infty$
    $closest\_node \leftarrow NULL$
    Initialize a priority queue, pq that stores a tuple of (distance, Node) that priorities minimal distances
    Push the tuple of distance between x, mean point of kd_tree using the user specific p norm into pq
    $pq.push((d(kd\_tree.root.point, x, p), kd\_tree.root))$
    while pq not empty **do**
        $(dist, node) \leftarrow pq.pop()$
        if $dist < curr\_min$ **then**
            if $node$ is not a leaf **then**
                $dist\_l \leftarrow d(node.left.point, x, p)$
                $dist\_r \leftarrow d(node.right.point, x, p)$
                $pq.push((dist\_l, node.left))$
                $pq.push((dist\_r, node.right))$
            else
                $curr\_min \leftarrow dist$
                $closest\_node = node$
        else
            if $closest\_node$ is not $NULL$ **then**
                return $closest\_node.point$
    return $closest\_node.point$

Algorithm 9 Algorithm for finding the a point's k nearest neighbours in the K-d tree

procedure K_NEAREST_NEIGHBOUR(kd_tree, k, x, p)
    $curr\_min \leftarrow \infty$
    $closest\_node \leftarrow NULL$
    Initialize pq
    Initialize empty list nn_lst
    $pq.push((d(kd\_tree.root.point, x, p), kd\_tree.root))$
    while pq not empty **do**
        $(dist, node) \leftarrow pq.pop()$
        if $dist < curr\_min$ **then**
            if $node$ is not a leaf **then**
                $dist\_l \leftarrow d(node.left.point, x, p)$
                $dist\_r \leftarrow d(node.right.point, x, p)$
                $pq.push((dist\_l, node.left))$
                $pq.push((dist\_r, node.right))$
            else
                if $size(nn\_lst) == 0$ **then**
                    $nn\_lst.append((dist, node))$
                    $curr\_min = nn\_lst[0][0]$
                else if $size(nn\_lst) == k$ **then**
                    max_index = The index value of the tuple stores the largest distance
                    $nn\_lst.pop(max\_index)$
                    $nn\_lst.insert(max\_index, (dist, node))$
                    Update max_index the same way as before
                    $curr\_min = nn\_lst[max\_index][0]$
                else
                    $nn\_lst.append((dist, node))$
                    max_index = The index value of the tuple stores the largest distance
                    $curr\_min = nn\_lst[max\_inde][0]$
        else
            if $size(nn\_lst) == k$ **then**
                break
    return All the points stores in the individual nodes within nn_lst

Algorithm Implementation in Python

Will first write helper functions that will be used in the implementation

```
import heapq

def sort_by_index(n_lst, index):
    """Sort a list of iterables based on one of the inner iterables' index, this assumes
    every inner list has the same length. This is done through mergesort."""
    m = len(n_lst)//2
    l = n_lst[:m]
    r = n_lst[m:]
    sort_by_index(l, index)
    sort_by_index(r, index)
    i = 0
    j = 0
    k = 0
    while i < len(l) and j < len(r):
        if l[i][index] < r[j][index]:
            n_lst[k] = l[i]
            i += 1
        else:
            n_lst[k] = j[r]
            j += 1
        k += 1
        while i < len(l):
            n_lst[k] = l[i]
            i += 1
            k += 1
        while j < len(r):
            n_lst[k] = r[j]
            j += 1
            k += 1

def get_max_by_index(lst, index):
    """Return the index of a maximum element in a list of iterables, based on one of the iterables's in
    i = 0
    curr_max = 0
    for j in range(len(data)):
        if data[i][index] < data[j][index]:
            i = j
    return i

def center_point(data):
    """Given a list of data points in k-dimension, generate
    the center of those point in the k-dimension"""
    cp = []
    for i in range(len(data)):
        coor = 0
        for j in range(len(data[i])):
            coor += data[i][j]
        coor /= len(data[i])
        cp.append(coor)
    return cp
```

```
def compare_points(y, x):
    """Compare 2 points, return true if they are equivalent, as in
    all the value of all coordinates are the same."""
    if len(y) != len(x):
        return False
    for j in range(len(y)):
        if y[j] != x[j]:
            return False
    return True

def norm(y, x, p):
    """Function to calculate the p-norm of 2 different points,
    if p > 20, will just take the infinity norm instead,
    this assumes y, x have the same dimension, and p > 1"""
    d = 0
    if p < 20:
        for i in range(len(y)):
            d += abs(x[i]-y[i])**p
        d = d**(1 / p)
        return d
    else:
        for j in range(len(x)):
            if d < abs(x[j] - y[j]):
                d = abs(x[j] - y[j])
        return d



class Node:
    def __init__(self, depth, mid = None, point = None, parent = None):
        """Node class for the k-d tree
        depth: int, how deep the node is in the tree
        mid: int, the median used to partition the data
        num: int, number of leave nodes at or below this node
        point: list[k], a data point in the k-dimension
        left: pointer to left node
        right: pointer to right node
        parent: pointer to parent node"""

        self.depth = depth
        self.mid = mid
        self.num = num
        self.point = point
        self.left = None
        self.right = None
        self.parent = parent

    def is_leaf(self):
        """Helper method to help determine whether is a node is a leaf node"""
        if self.mid == None and self.left == None and self.right == None:
            return True
        else:
            return False
```

```
    def delete(self):
        """Helper method to delete a node"""
        self.depth = None
        self.mid = None
        self.num = None
        self.point = None
        self.left = None
        self.right = None
        self.parent = None
        del self

class Kd_Tree:
    def __init__(self, data):
        """ The class for the Kd_Tree itself, assume there is at least 1 data point
        dim: int, the dimension the data lives in, or k
        root: Node, pointer to root node
        """

        self.dim = len(data[0])
        if len(data) == 1:
        ##if the number of data point is just 1, then only the root node is needed
            self.root = Node(depth = 0, num = 1, point = data)
        else:
            depth = 0
            ##otherwise, first find the median of the data at the depth mod k dimension
            median = len(self.data) // 2
            ##then find the average point for the data that will be use to present this
            ##node/space partition
            cp = center_point(data)
            ##sort the data based on that same dimension
            sort_by_index(data, depth % self.dim)
            ##Build the node and recursively build the subtrees using the helper method
            self.root = Node(depth = 0, mid = data[median][depth % self.dim],
                    num = len(data), point = cp)
            self.root.left = self.build_subtree(self.root, self.root.depth + 1, data[:median])
            self.root.right = self.build_subtree(self.root, self.root.depth + 1, data[median:])

    def build_subtree(self, build_at, depth, data):
        """Helper method to help build the subtrees, the procedure is the same
        as the Kd_tree constructor"""

        if len(data) == 1:
            return Node(depth = depth, num = 1, point = data, parent = build_at)
        else:
            median = len(data) // 2
            cp = center_point(data)
            sort_by_index(data, depth % self.dim)
            temp_node = Node(depth = depth, mid = data[median][depth % self.dim],
                        num = len(data) , point = cp, parent = build_at)
            temp_node.left = self.build_subtree(temp_node, depth + 1, data[:median])
            temp_node.right = self.build_subtree(temp_node, depth + 1, data[median:])
            return temp_node

    def nearest_neighbour(self, x, p):
```

```
        """Assumes the x has the the same dimensionality as the data
        points in the tree, and p > 1, this method returns the given
        input data point x's nearest neighbour with in the kd-tree,
        using the metric of the user's specified p-norm"""

        curr_min = float("inf")
        pq = []
        ##this is a min priority queue using the library heapq, that stores
        ##tuples with 2 elements, with the first element being the distance
        ##between the point and arbitrary node, this will be priorities in
        ##the priority queue, and the second being the node itself

        closest_node = None
        temp_dist = norm(self.root.point, x, p)
        heapq.heappush(pq, (temp_dist, self.root))

        while (pq != []):
            (dist, node) = heapq.heappop(pq)
            ##pop the tuple of distance and node that from priority queue,
            ##that prioritize minimal distance
            if dist < curr_min:
            ##if the distance is smaller than the current minimum
                if node.is_leaf() == False:
                ##check whether the associate node is a leaf node, if it isn't
                    if node.left != None:
                        temp_dist = norm(node.left.point, x, p)
                        heapq.heappush(pq, (temp_dist, node.left))
                        ##add the norm between the query point and the left node
                        ##to the priority queue, if the left node is not None
                    if node.right != None:
                        temp_dist = norm(node.right.point, x, p)
                        heapq.heappush(pq, (temp_dist, node.right))
                        ##perform the same procedure to the right node, if it is not None
                else:
                    ##If the associate node is a leaf node
                    ##check whether the minimum distance just obtained from the
                    ##priority queue is smaller than the current minimum
                    curr_min = dist
                    closest_node = node
                    ##store that minimal distance and its associate node
            else:
                ##If the distance that just obtained from the min priority queue
                ##is >= curr_min, then all the norms that is within the priority queue
                ##should be larger.
                if closest_node != None:
                    ##Afterwards check whether closest_node is None,
                    ##this indicates whether a leaf node has been visited.
                    ##If it isn't, than the point store in that closest_node is
                    ##the nearest neighbour
                    return closest_node.point
        return closest_node.point

    def k_nearest_neighbours(self, k, x, p):
        """Assumes x has the same dimensionality as the data points as the data
```

```
points in the tree, p > 1, and k is an integer > 0, this method
returns the given input point x's k number of nearest neighbours
within the kd-tree using the metric of the user's specified p-norm.
If k > number of points in the tree, than just return all the points in the tree"""

##The approach will the same as nearest neighbour with some minor variations.
if k == 1:
    [return self.nearest_neighbour(x, p)]
curr_min = float("inf")
pq = []
nn_lst = []
tempest_dist = norm(self.root.point, x, p)
heapq.heappush(pq, (temp_dist, self.root))
while pq != []:
    (dist, node) = heapq.heqpop(pq)
    if dist < curr_min:
        if node.is_leaf() == False:
            if node.left != None:
                temp_dist = norm(node.left.point,x, p)
                heaq.heappush(pq, (temp_dist, node.left))
            if node.right != None:
                temp_dist = norm(node.right.point, x, p)
                heapq.heappush(pq, (temp_dist, node.right))

        else:
            if len(nn_lst) == 0:
            ##if nn_lst is empty, added the tuple of the minimal norm
            ##and its corresponding leaf node to nn_lst regardless
            ##and keep track of the maximum distance in the list
                nn_lst.append((dist, node))
                curr_min = nn_lst[0][0]

            elif len(nn_lst) == k:
            ##if the list is full, and the new distance is less than the
            ##maximum in the list, then get rid of the tuple holding the maximum
            ##distance and node, and insert the one popped from the pq,
            ## sort the list according to the distances in the tuples and keep of
            ##track the maximum distance
                max_index = get_max_by_index(nn_lst, 0)
                nn_lst.pop(max_index)
                nn_lst.insert(max_index, (dist, node))
                max_index = get_max_by_index(nn_lst, 0)
                curr_min = nn_lst[max_index][0]
            else:
               ##nn_lst it not empty, but also not full, add the tuple to the list,
               ##sort it, and keep track of maximum
                nn_lst.append((dist, node))
                max_index = get_max_by_index(nn_lst, 0)
                curr_min = nn_lst[max_index][0]
        else:
            if len(nn_lst) == k:
            ##if the list is full, and distance in the tuple just popped
            ##from the pq is bigger than the maximum distance in the list,
            ##than all distances in the pq is bigger than the maximum distance
```

```
                ##in the list, the while loop can be terminated, and the k nearest
                ##neighbours has been found
                    break
        return_lst = []
        for pair in nn_lst:
            return_lst.append(pair[1].point)
        return return_lst



    def insert(self, x):
    """Insert the point x in the k dimension into an already built kd_tree
    """

        curr_node = self.root
        while curr_node.is_leaf() != True:
            ##First update the mean point of curr_node
            for i in range(len(curr_node.point)):
                curr_node.point[i] = (curr_node.point[i] * curr_node.num + x[i])
                curr_node.point[i] /= (curr_node.num + 1)
            ##updating num attribute of curr_node
            curr_node.num += 1
            ##getting the axis that will be used for comparison
            axis = curr_node.depth % self.dim
            if x[axis] < curr_node.mid:
                ##if the value of x[axis] is less than the median that
                ##is use to partition the data, set curr_node to its left node
                curr_node = curr_node.left
            else:
                ##otherwise set curr_node to its right node
                curr_node = curr_node.right
        ##After a leaf node is reach, getting the axis that will be used for comparsion
        axis = curr_node.depth % self.dim
        if x[axis] < curr_node.point[axis]:
            curr_node.left = Node(curr_node.depth + 1, num = 1, point = x, parent = build_at)
            curr_node.right = Node(curr_node.depth + 1, num = 1, point = curr_node.point,
                        parent = build_at)
        else:
            curr_node.right = Node(curr_node.depth + 1, num = 1, point = x, parent = build_at)
            curr_node.left = Node(curr_node.depth + 1, num = 1, point = curr_node.point,
                        parent = build_at)
        ##Make new nodes according to  how x[axis] compares curr_node.mid
        for j in range(len(curr_node.point)):
            curr_node.point[j] += x[j]
            curr_node.point[j] /= 2
        curr_node.num = 2
        curr_node.mid = curr_node.right.point[axis]
        ##Update the old leaf node

    def search(self, x):
        """Search and return the Kd_tree to a leaf node that stores a point that is
        the most similar to x. This assumes x's dimension is the same as the dimension
        of the points in the Kd_tree.
        Note: This is not the nearest neighbour"""
```

```
    curr_node = self.root
    while curr_node.is_leaf() != True:
        axis = curr_node.depth % self.dim
        if x[axis] < curr_node.mid:
            curr_node = curr_node.left
        else:
            curr_node = curr_node.right
    return curr_node


def delete(self, x):
    """Deletes the point x in the already built kd_tree, this assumes
    x also has the same dimension as the points in the the Kd_tree

    If x is not a point in the Kd_tree, -1 will be returned"""

    leaf_node = self.search(x)
    if compare_points(leaf_node.point, x) == False:
        ##x is not found in the tree, simply return -1
        return -1

    if self.root is leaf_node:
        self.root.delete()
        return None
    ##Edge case if the root node is the also the leaf node

    curr_node = self.root
    while curr_node.is_leaf() != True:
        ##First update the mean point of curr_node
        for i in range(len(curr_node.point)):
            curr_node.point[i] = curr_node.point * curr_node.num - x[i]
            curr_node.point[i] /= curr_node.num - 1
        ##Update the num attribute of curr_node
        curr_node.num -= 1
        axis = curr_node.depth % self.dim
        if x[axis] < curr_node.mid
            curr_node = curr_node.left
        else:
            curr_node = curr_node.right

    ##Deleting the node itself and repairing the tree so it maintains the properties
    ##of a kd_tree
    if curr_node.left is curr_node:
        prev_node = curr_node.parent
        curr_node.delete()
        self.repair(prev_node)

    elif prev_node.right is curr_node:
        prev_node = curr_node.parent
        curr_node.delete()
        self.repair(prev_node)
```

```
def delete_at(self, _node):
    """Delete every node that is either at _node or below _node"""

    ##Will use postorder traversal to delete the subtree

    if _node == None:
        return
    delete_at(self, _node.left)
    delete_at(self, _node.right)
    _node.delete()

def get_points(self, _node, lst = []):
    """Gets every data points at leaf nodes that is either at _node or below _node"""

    if _node != None:
        if _node.is_leaf() == True:
            lst += [_node.point]
        else:
            self.get_points(_node.left, lst)
            self.get_points(_node.right, lst)
    return lst

def repair(self, _node):
    """Wrapper function to repair the tree once a deletion is done"""

    par_node = _node.parent
    point_lst = self.get_points(_node)
    temp_node = self.build_subtree(par_node, par_node.depth + 1, point_lst)
    if par_node.left is _node:
        par_node.left = temp_node
    elif par_node.right is _node:
        par_node.right = temp_node
    self.delete_at(_node)
```

Analysis

Summary Table of BC/WC of all the functions

For the table below, n is the number of inital data points, d is the dimension the data lives in, k is the number of neighbours for `k_nearest_neightbours`.

Table of temporal complexity			
Function/Method	Best Case $\Omega(\cdot)$	Worst Case $\mathcal{O}(\cdot)$	Comment
LSH			
`norm`	d	d	
Constructor of `LSH`	qn	qn	
`insert`	qd	qd	
`delete`	qd	qd	
`nearest_neighbour`	Sub-Linear Time	Linear Time	Specific parameter values provide us with best expected sub-linear running time performance guarantees
`k_nearest_neighbour`	Sub-Linear Time	Linear Time	Specific parameter values provide us with best expected sub-linear running time performance guarantees
K-d Tree			
`sort_by_index`	$n\log(n)$	$n\log(n)$	It is just merge sort
`get_max_by_index`	n	n	
`center_point`	dn	dn	
`compare_points`	1	d	BC : returns early, 2 points lives in different dimensions. WC : 2 points are the same
`norm`	d	d	
Constructor of `Kd_tree`	n^2d	n^2d	This assumes $n >> d$
`insert`	d	dn	assuming the scenario of an extremely unbalanced tree
`delete`	$d\log(n)$	n^2d	BC : Tree is perfectly balanced, repairing takes constant time. WC : Tree is very unbalanced, repairing almost requires rebuilding tree from ground up
`search`	1	n	assuming the scenario of an extremely unbalanced tree
`nearest_neighbour`	$d\log(n)$	$dn + n\log(n)$	BC: The search quickly identified the first leave node reached to be the nearest neighbour. WC: Have to go through all the nodes
`k_nearest_neighbour`	$dk\log(n)$	$dn + n\log(n)$	This assumes $n >> k$

Spatial Complexity

Locality Sensitive Hashing

Let $\mathcal{H}$ denote a family of hash functions from which m hash functions are chosen such that $g_i = (h_{1i}, \ldots, h_{mi})$. Suppose the total number of points are n that belong to a d dimensional space, we now look at space complexity of using a total of q such functions $g_1 \ldots g_q$ with their corresponding hash buckets to store these points.

During the pre-processing stage when n points are stored in each of the q hash tables. Each hash table will have n non-zero entries therefore taking up $\mathcal{O}(n)$ space in memory where it is assumed that the space used for storing each point is constant.

Finally, since we have q such hash tables, the total spatial complexity becomes $\mathcal{O}(qn)$

K-d tree

For any arbitrary k-d tree constructed with n number of points, for the sake of simplicity assume n is a power of 2, and lives in $\mathbb{R}^d$ has a spatial complexity of

$$\mathcal{O}(dn)$$

This is because it requires:

$$\sum_{i=0}^{\log_2(n)} 2^i = 2^{\log_2(n)+1} - 1 = 2n - 1 = \text{Number of nodes}$$

Each nodes requires to store a d dimensional data point (and a few pointers, which is a constant), therefore it requires a memory space of $\mathcal{O}(dn)$, (this is true even if the tree is unbalanced, as based on our implementation, every inner node must have 2 child nodes).

Randomised Analysis of Nearest Neighbour Search Using Locality Sensitive Hashing

Let $\mathcal{S} \subseteq \mathbb{R}^d$ be a finite set of points where each point $\mathbf{x} \in \mathcal{S}$ is in d dimensional space such that $|\mathcal{S}| = n$ and $p > 1 \in \mathbb{R}$ be the norm.

We use $\mathcal{H}$ to denote a family of hash functions from which m hash functions are chosen such that $g_i = (h_{1i}, \ldots, h_{mi})$. A total of q such functions $g_1 \ldots g_q$ with their corresponding hash buckets are used to to store these n points.

The main goal of Locally Sensitive Hashing is to hash close points in Euclidean space in similar buckets and points far away to different buckets. For functions $g_1 \ldots g_q$, points $x, y \in \mathcal{S}$ and error approximation threshold ϵ mathematically this idea can be expressed as follows,

$$\begin{cases} \text{If } d(x,y,p) \geq \epsilon, & \text{then} \quad \Pr[g_i(x) = g_i(y)] \leq p_1 \\ \text{If } d(x,y,p) < \epsilon, & \text{then} \quad \Pr[g_i(x) = g_i(y)] \geq p_2 \\ p_1 \ll p_2 \end{cases}$$

For each of the q functions of the form $g_1 \ldots g_q$ where $g_i = (h_{1i}, \ldots, h_{mi})$, the numbers q and m are chosen to ensure constant probability of finding a nearest neighbour approximately if a nearest neighbour exists as well as to reduce the number of collisions if it does not.

Let $\delta = \frac{\log \frac{1}{p_2}}{\log \frac{1}{p_1}}$ and suppose we have $m = \frac{\log n}{\log \frac{1}{p_1}}$ and $q = 2n^\delta$. We will now show that these specific parameter values provide us with best expected sub-linear running time performance guarantees and also maximise the probability of collisions.

The probability of finding a nearest neighbour for query point y when one exists is given by,

$$\Pr[g_i(x) = g_i(y)] \geq p_2 \geq p_2^{-\frac{\log n}{\log p_1}} = p_2^{\frac{\log n}{\log \frac{1}{p_1}}} = n^{-\frac{\log \frac{1}{p_2}}{\log \frac{1}{p_1}}} = n^{-\delta} \qquad \text{(Using Log Properties)}$$

$$\implies \Pr[g_i(x) \neq g_i(y)] \leq 1 - n^{-\delta} = 1 - \frac{2}{q} \leq 1 - \frac{1}{q} \leq \left(1 - \frac{1}{q}\right)^q \leq \frac{1}{e} \qquad (\because q = 2n^\delta, \text{ Bernoulli's Inequality})$$

Using the above we can then compute the probability of at least one collision across the q hash tables for query point y when one exists given by,

$$\Pr[\text{x and y collide at least once in } q \text{ tables}] \geq 1 - (\Pr[g_i(x) \neq g_i(y)])^q = 1 - (1 - n^{-\delta})^q \geq 1 - \frac{1}{e} \geq 0.6$$

Therefore, our choice of ideal parameters will ensure that two points are hashed with probability of at least 0.6 in the same buckets and ensures constant probability of finding a nearest neighbour approximately if a nearest neighbour exists– this ensures correctness and proves that a hash function family can be found maintaining these properties.

We set arbitrary thresholds $p_1 = 1 - \frac{1+\epsilon}{d}$ and $p_2 = 1 - \frac{\epsilon}{d}$ and assume without loss of generality $1 < \epsilon < \frac{d}{\log n}$, we will now use the previous results to find complexity of finding nearest neighbour search.

For these values of p_1 and p_2 we have,

$$\delta = \frac{\log \frac{1}{p_2}}{\log \frac{1}{p_1}} = \frac{\log \frac{1}{1-\frac{\epsilon}{d}}}{\log \frac{1}{1-\frac{1+\epsilon}{d}}} = \frac{\log(1 - \frac{\epsilon}{d})}{\log(1 - \frac{1+\epsilon}{d})} = \frac{\frac{d}{\epsilon}\log(1 - \frac{\epsilon}{d})}{\frac{d}{\epsilon}\log(1 - \frac{1+\epsilon}{d})} = \frac{\log(1 - \frac{\epsilon}{d})^{\frac{d}{\epsilon}}}{\log(1 - \frac{1+\epsilon}{d})^{\frac{d}{\epsilon}}}$$

We now bound the numerator from below and denominator from above by using the following inequalities (Motwani & Raghavan[5])

$$(1-(1+\epsilon)/d)^{d/\epsilon} < e^{-1+\epsilon} \tag{1}$$

$$\left(1-\frac{\epsilon}{d}\right) > e^{-1}\left(1-\frac{1}{d/\epsilon}\right) \tag{2}$$

Using (1) and (2) and the initial assumptions that $\frac{d}{\log n} > \epsilon > 1$ we now have,

$$\delta < \frac{\log\left(e^{-1}\left(1-\frac{1}{d/\epsilon}\right)\right)}{\log\left(e^{-1+\epsilon}\right)} = \frac{\log\left(1-\frac{1}{d/\epsilon}\right)-1}{-(\epsilon+1)} = \frac{1}{1+\epsilon} - \frac{\log\left(1-\frac{1}{d/\epsilon}\right)}{1+\epsilon} < \frac{1}{1+\epsilon} - \log(1-1/\log n)$$

Finally since the expected number of hash tables to be searched are $q = 2n^{\delta}$ with each pass through the hash function taking $\mathcal{O}(d)$ expected time (points are d dimensional), therefore the total expected steps are $2n^{\delta}d$ and we obtain the following final result using previously proven results,

$$\mathbb{E}(\mathcal{T}_n) = 2dn^{\delta} \leq 2dn^{(\frac{1}{1+\epsilon}-\log(1-1/\log n))} = 2dn^{\frac{1}{1+\epsilon}}\left(1-\frac{1}{\log n}\right)^{-\log n} \in \mathcal{O}(dn^{\frac{1}{1+\epsilon}})$$

Therefore we have shown that the proposed algorithm performs nearest neighbour search in $\mathcal{O}(dn^{\frac{1}{1+\epsilon}})$ expected running time which is sub-linear.

■

Analysis of Insertion using Locality Sensitive Hashing

Let $\mathcal{S} \subseteq \mathbb{R}^d$ be a finite set of points where each point $\mathbf{x} \in \mathcal{S}$ is in d dimensional space such that $|\mathcal{S}| = n$ and $p > 1 \in \mathbb{R}$ be the norm.

We use $\mathcal{H}$ to denote a family of hash functions from which m hash functions are chosen such that $g_i = (h_{1i}, \ldots, h_{mi})$. A total of q such functions $g_1 \ldots g_q$ with their corresponding hash buckets are used to to store these n points.

Suppose $\mathbf{x} \in \mathbb{R}^d$ is a new point to be inserted using our algorithm. Then the point is hashed q times through the hash functions $g_1 \ldots g_q$ with each pass through the hash function taking $\mathcal{O}(d)$ expected time (points are d dimensional). Assuming that the time taken to hash and store the point is constant the procedure therefore takes at most qd steps and has worst case running time $\mathcal{O}(qd)$.

Note that choosing the right number of hash tables (q) can result in the time complexity being sub-linear as shown previously.

Worst Case analysis of K-d tree's Nearest Neighbour

In this analysis, we will assume the algorithm never terminates early, so the last else block within the while loop is never reached, and the while loop only terminates if the priority queue is empty, which requires almost all nodes to be visited. (Imagine a scenario where nearest neighbour query is done/ with the query point being roughly in the center of a cluster of points/ with all those points being roughly equal distance to the query point/ the dimensionality is too high (Marimont and Shapiro, 1979)[6]). The priority queue is to keep track of nodes that have been visited, and it ordered by the distance between the query point and the points stores in the nodes.

Since we are assuming the nearest neighbour search requires to visit and process all nodes, and there are $2n-1$ nodes in the k-d tree as mentioned in the spatial complexity analysis we obtain the following,

- $2n-1$ distance calculations will be required– each having time complexity $\mathcal{O}(d)$ per operation.
- $2n-1$ number of nodes will be inserted and popped from the priority queue (using the heap implementation of a priority queue) both operations requiring $\mathcal{O}(\log(n))$ steps per operation.

Using the above results we finally obtain the total worst case running time of this procedure as follows,

$$\mathcal{O}(2dn - d + 2n\log(n) - 2\log(n)) \in \mathcal{O}(dn + n\log(n))$$

It should be noted that even though the worst case running time of this procedure is poorer as compared to brute force loop approaches, situations requiring the algorithm to visit all nodes likely don't come often.

Amortized analysis of K-d tree's Insertion Algorithm

For this analysis, we will assume the K-d tree with n nodes to be inserted with m new data points is extremely unbalanced, and that $n >> m$. In addition, we will assume the new m points inserted into the tree will make the tree even further unbalanced, (Such as at the first insertion a new leaf node will be created as the leaf node that is causing the worst unbalancing issue, and the new nodes afterwards as the child node of the previous inserted node and so on). The aggregate method will be employed.

Since the first new point will be inserted as a child node of the node that is causing the worst unbalancing issue, traversing from the root node to that node will be expected to traverse between:

$$\log(n) \leq \text{Number of nodes traversed} \leq n$$

Within the while loop for the traversal, updating the point store at each node takes a runtime of d. Assuming building the new node takes constants time, coupled with the fact a total of n nodes are being updated, including the leaf node reached. This gives each individual worst case insertion to be

$$\mathcal{O}(dn)$$

Then according to the procedure of insert m points described above gives a $WCSC$ of:

$$WCSC = \sum_{i=0}^{m-1} d(n+i) = dnm + \sum_{i=0}^{m-1} i = dnm + \frac{m(m-1)}{2}$$

Thus gives an amoritized runtime of:

$$\frac{WCSC}{m} = \frac{dnm}{m} + \frac{m(m-1)}{2m} = dn + \frac{(m-1)}{2} \overset{n>>m}{\in} \mathcal{O}(dn)$$

References

[1]Kibriya A.M., Frank E. (2007) An Empirical Comparison of Exact Nearest Neighbour Algorithms. In: Kok J.N., Koronacki J., Lopez de Mantaras R., Matwin S., Mladenič D., Skowron A. (eds) Knowledge Discovery in Databases: PKDD 2007. PKDD 2007. Lecture Notes in Computer Science, vol 4702. Springer, Berlin, Heidelberg

[2]Indyk, Piotr; Motwani, Rajeev; Raghavan, Prabhakar; Vempala, Santosh (1997). "Locality-preserving hashing in multidimensional spaces". Proceedings of the twenty-ninth annual ACM symposium on Theory of computing. pp. 618–625.

[3]Bentley J.L., (1975). Multidimensional binary search trees used for associative searching. Communications of the ACM, 18(9):509–517, 1975.

47

[5]R. Motwani and P. Raghavan. Randomized Algorithms. Cambridge University Press, 1995

[6]Marimont R.B., Shapiro M.B. (1979). Nearest Neighbour Searches and the Curse of Dimensionality g. Communications of the ACM,MA J Appl Math. 24 (1): 59–70.