Kd Tree

In [ ]:

```
import os
import sys
import copy
import numpy as np
from scipy import stats
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score


import warnings
warnings.filterwarnings("ignore")
%load_ext autoreload
%autoreload 2
sys.path.append("./../../src")

from load_data import gen_data
```

KD-Tree's Theory and Code

Function list

- build_tree: Build a KDTree, store every data in its branchs and leaves.
  - `_split_feat` : Assume we already have chosen a feature, find the **median value**, and split the data into left (<median) and right(>median). Then save the feature and split value.

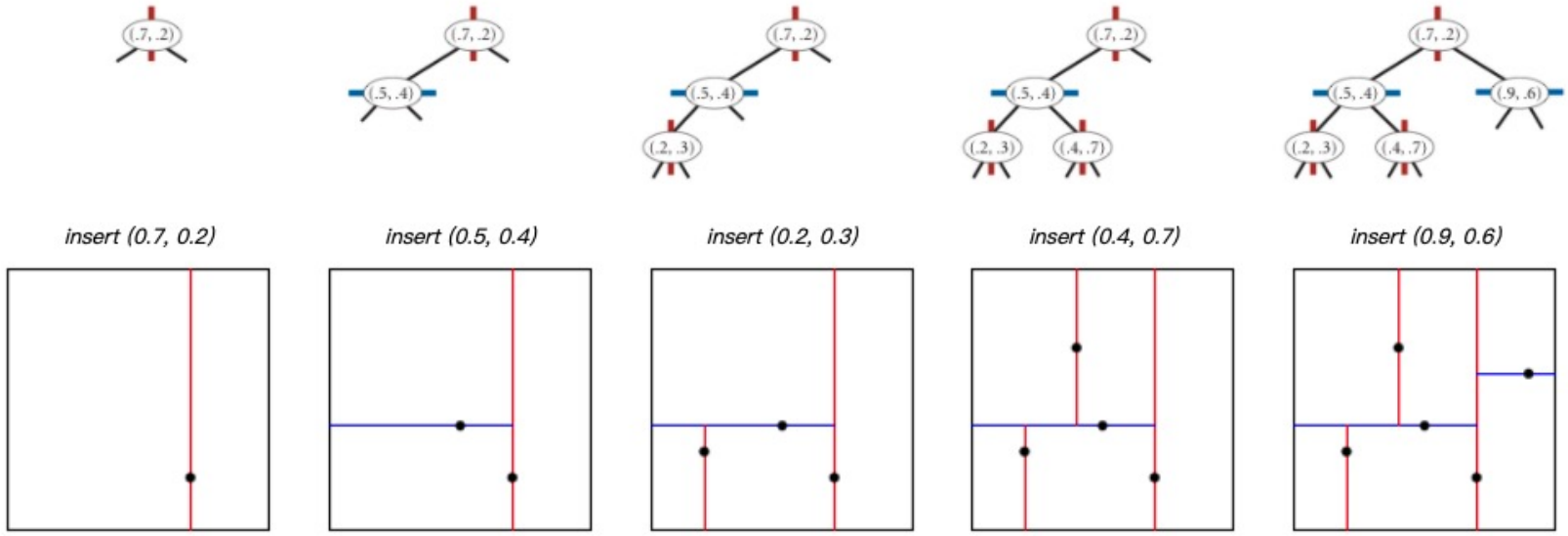
    ```
    _split_feat:

    1. Find the feat_id with biggest variance
    2. Find the median value of X[feat_id], assume it occurs in the nth sample in data
    3. Save feat_id in node.feature, save the Nth sample in node.split
    4. Return samples (0 to N-1) as X_left, samples (N+1 to end) as X_right `
    ```
  - `build_tree` : BFS function. Use breadth first search to buile every node. run the loop until the data can not be spilt.

    ```
```

build_tree:

1. Build a Stack, initiate with a null KDTree and data X. stack = [(nd, X)]
2. Take out the first item in stack, use _split_feat(X), save the feature and sample in nd.feature and nd.split.
3. Add (nd.left, X_left), (nd.right, X_right) in the stack
4. Run the loop until there is no item in the stack. In this situation, every node in the KDTree only have one data point, and every data in the sample are stored in the KDTree.

`



- search_knearest: Use that KDTree, search the top k nearest data.
  - `_search` : Given a new data Xi and a current node, search Xi from the KDTree until Xi is at an leafnode.
    ```
    _search:
      1. Given a data Xi and a node in the KDTree.
      2. Check the current feat_id and split_value, if Xi[feat_id] < split_value, go to the left node. Otherwise, go to the right node.
      3. Iterate step 2, till the current node is the leaf node. `
    ```
  - `_backtrace` : Given a new data Xi, a current node and a target node, backtrace Xi from the KDTree begining with the current node to the target node.
    ```
    ```

_backtrace:

1. Given a data Xi, current node in the KDTree, and the target node in the KDTree.
2. Starting from the current node A, calculate the distance between Xi and node A. If it is smallest, save it in a list.
3. Considering current node's father node B, calculate the distance between Xi and hyper plane PB.
    - Explaination! This one is extremely important. if a node's father node's hyper plane is farther away from Xi, this means all nodes in brother node are even farther. So we dont need to consider those nodes, and go to the father node.
    - if this distance is smaller than k smallest nodes in the list, stop backtrace and return brother node
    - if this distance is larger than k smallest nodes in the list, go to the father node without checking brother node.
4. Run the loop until the node arrive target node, or we need to check the one brother node first. `

- `search_knearest` : BFS function. Use breadth first search to find k nearest neighbour of X.

```
```

search_knearest:

1. Initiate the stack, stack = [(kdtree.root, kdtree.root, 'f')]. Every item in stack has 3 elements. First one is current node, the second one is target node, the third one is a sign to determine whether we use backtrace or not.
2. Take the item from stack.
    - if sign is "f", we need to use _search function to go to the leaf node from current node and use _backtrace.
        - if we need to check one brother tree in _traceback, add (brother_node, brother_node, "f"), (father_node, kdtree.root, "b") in the stack.
    - if sign is "b", then we just do _traceback. It is same as below.
3. Run the loop until there is nothing in the stack. During the process, the knearest neighbour is keep changing. `

check the illustrator on youtube: https://www.youtube.com/watch?v=2SbVSxWGtpI

KD-Tree's Code

In [ ]:

```
class Node():

    def __init__(self):
        self.father = None
        self.left = None
        self.right = None
        self.feature = None
        self.split = None

    def _str_(self):
        feature, split = self.feature, self.split
        print(f"Feature:{feature}, Split_value:{split}")
```

```
    @property
    def brother(self):
        if self.father is None:
            ret = None
        else:
            if self.father.left is self:
                ret = self.father.right
            else:
                ret = self.father.left
        return ret


class KDTree():

    def __init__(self, k=3):
        self.root = Node
        self.k = k

    @staticmethod
    def distance(p1, p2):
        if p1 is None or p2 is None:
            return 0
        return ((p2 - p1) ** 2).sum() ** 0.5

    @staticmethod
    def _split_feat(X):

        feat_idx = np.argmax(X.var(axis=0))
        # print(len(X), feat_idx)

        if len(X) % 2 == 0:
            mid = int(len(X) / 2) - 1
        else:
            mid = int(len(X) / 2 - 0.5)

        X_sort = X[X[:,feat_idx].argsort()]
        split = X_sort[:][mid]
        X_left = X_sort[:mid]
        X_right = X_sort[mid+1:]

        return feat_idx, split, X_left, X_right

    def _search(self, X_i, node):

        """
        Search Xi from the KDTree until Xi is at an leafnode.
```

```
        Arguments:
            Xi[list] -- 1d list with int or float.

        Returns:
            node -- Leafnode.
        """

        while node.left or node.right:

            # print("feat_idx:{}, Xi value:{}, Node value:{}".format(node.feature, X_i[node.feature], node.split))

            if node.left is None:
                node = node.right
                direct = 'right'
            elif node.right is None:
                node = node.left
                direct = 'left'
            else:
                if X_i[node.feature] < node.split[node.feature]:
                    node = node.left
                    direct = 'left'
                else:
                    node = node.right
                    direct = 'right'

            # print(f"searching path turn {direct}")

        # print("feat_idx:{}, Xi value:{}, Node value:{}".format(node.feature, X_i[node.feature], node.split))
        return node

    def _get_dist_eu(self, X_i, node):

        """
        Calculate euclidean distance between Xi and node.

        Arguments:
            Xi[list] -- 1d list with int or float.
            nd[node]

        Returns:
            float -- Euclidean distance.
        """


        return self.distance(X_i, node.split)
```

```
    def _get_dist_hyper(self, X_i, node):

        """
        Calculate euclidean distance between Xi and hyper plane.

        Arguments:
            Xi[list] -- 1d list with int or float.
            nd[node]

        Returns:
            float -- Euclidean distance.
        """

        feat_idx = node.feature

        return abs(X_i[feat_idx] - node.split[feat_idx])

    def _backtrace(self, X_i, node, root):

        """
        Backtrace node from the KDTree begining with the leafnode.

        Arguments:
            Xi]list] -- 1d list with int or float.

        Returns:
            node -- The nearest node to Xi.
        """

        dist_max = self.kbest[-1][1]
        # print(X_i, node.split, root.split)

        while node is not root:

            dist = self._get_dist_eu(X_i, node)
            # print(self.kbest, dist, dist_max)

            if dist < dist_max:
                self.kbest[-1][1] = dist
                self.kbest[-1][0] = node
                self.kbest = sorted(self.kbest, key=lambda x:x[1])
                dist_max = self.kbest[-1][1]

            node_father = node.father
```

```
            if self._get_dist_hyper(X_i, node_father) <= dist_max:
                return [(node.brother, node.brother, "f"), (node_father, root, "b")]
            else:
                node = node_father

        dist = self._get_dist_eu(X_i, node)
        if dist < dist_max:
            self.kbest[-1][1] = dist
            self.kbest[-1][0] = node
            self.kbest = sorted(self.kbest, key=lambda x:x[1])
        return []

    def show(self):

        queue = [self.root]

        while queue:
            nd = queue.pop(0)
            feature, split = nd.feature, nd.split
            print(f"Feature:{feature}, Split_value:{split}")
            if nd.left is not None:
                queue.append(nd.left)
            if nd.right is not None:
                queue.append(nd.right)

    def stat_tree(self):

        queue = [(self.root, 0)]
        max_depth = -1
        leaf_num = 0

        while queue:

            node, depth = queue.pop(0)

            if depth > max_depth:
                max_depth = depth

            if node.left is not None:
                queue.append((node.left, depth+1))

            if node.right is not None:
                queue.append((node.right, depth+1))

            if node.left is None and node.right is None:
                leaf_num += 1
```

```
        print(f"Max_depth of KD-tree: {max_depth}, Leaf number of KD-tree: {leaf_num}")


    def build_tree(self, X):

        nd = self.root
        queue = [(nd, X)]

        while queue:

            nd, X = queue.pop(0)
            nd.feature, nd.split, X_left, X_right = self._split_feat(X)
            # print(nd.feature, nd.split, len(X_left), len(X_right))

            if len(X_left) != 0:
                nd.left = Node()
                nd.left.father = nd
                queue.append((nd.left, X_left))

            if len(X_right) != 0:
                nd.right = Node()
                nd.right.father = nd
                queue.append((nd.right, X_right))

    def search_knearest(self, X_i):

        self.kbest = [[None, np.inf] for _ in range(self.k)]

        node = self.root
        queue = [(node, node, "f")]

        while queue:

            current, root, sign = queue.pop(0)

            if current is None:
                continue

            if sign == "f":
                current = self._search(X_i, current)
                queue += self._backtrace(X_i, current, root)
            elif sign == "b":
                queue += self._backtrace(X_i, current, root)

            # print(self.kbest)
```

```
## Use exhuast search to compare
def exhuast_search(X_i, X_train):

    def distance(p1, p2):
        if p1 is None or p2 is None:
            return 0
        return ((p2 - p1) ** 2).sum() ** 0.5

    ds = []
    for i in X_train:
        d = distance(i, X_i)
        ds.append([X_i, i, d])

    ds = sorted(ds, key=lambda x:x[2])

    return ds
```

Test the result of KD-Tree

Build KD-Tree

In [ ]:
```
X_train = gen_data(0, 5, 1000, 5, seed=123)
```

In [ ]:
```
kd_tree = KDTree()

kd_tree.build_tree(X_train)

kd_tree.stat_tree()
```

```
Max_depth of KD-tree: 9, Leaf number of KD-tree: 489
```

Test the code's accuracy

In [ ]:
```
import time

print("Result of KD search ...")

X_i = X_train[500]

time1 = time.time()
kd_tree.search_knearest(X_i)
```

```
time2 = time.time()
print("Time consumption:{}".format(time2 - time1))
display([(X_i, x[0].split, x[1]) for x in kd_tree.kbest])

print("\r\nResult of violate search ...")
time1 = time.time()
sds = exhuast_search(X_i, X_train)[:3]
time2 = time.time()
print("Time consumption:{}".format(time2 - time1))
display(sds)
```

```
Result of KD search ...
Time consumption:0.0003330707550048828
[(array([0.38702781, 3.91090853, 0.294159  , 3.8063372 , 4.2228834 ]),
  array([0.38702781, 3.91090853, 0.294159  , 3.8063372 , 4.2228834 ]),
  0.0),
 (array([0.38702781, 3.91090853, 0.294159  , 3.8063372 , 4.2228834 ]),
  array([0.48944018, 4.39072174, 0.8783012 , 3.73667697, 4.97712649]),
  1.0750192075329),
 (array([0.38702781, 3.91090853, 0.294159  , 3.8063372 , 4.2228834 ]),
  array([1.17334826, 3.53623598, 0.32819185, 3.9362685 , 4.89778163]),
  1.1100483643138432)]
Result of violate search ...
Time consumption:0.007420778274536133
[[array([0.38702781, 3.91090853, 0.294159  , 3.8063372 , 4.2228834 ]),
  array([0.38702781, 3.91090853, 0.294159  , 3.8063372 , 4.2228834 ]),
  0.0],
 [array([0.38702781, 3.91090853, 0.294159  , 3.8063372 , 4.2228834 ]),
  array([0.48944018, 4.39072174, 0.8783012 , 3.73667697, 4.97712649]),
  1.0750192075329],
 [array([0.38702781, 3.91090853, 0.294159  , 3.8063372 , 4.2228834 ]),
  array([1.17334826, 3.53623598, 0.32819185, 3.9362685 , 4.89778163]),
  1.1100483643138432]]
```

Test 25 samples

In [ ]:

```
X_is = gen_data(0, 5, 25, 5, seed=560)

fail_sum, succ_sum = 0, 0

for idx, X_i in enumerate(X_is):

    rst1 = exhuast_search(X_i, X_train)[:3]
    kd_tree.search_knearest(X_i)
    rst2 = [[X_i, x[0].split, x[1]] for x in kd_tree.kbest]
```

```
    if sum([str(xx) == str(yy) for xx, yy in zip(rst1, rst2)]) != 3:
        fail_sum += 1
    else:
        succ_sum += 1

print("Test Result: All samples num is {}, Correct/Failure:{}/{}".format(fail_sum + succ_sum, succ_sum, fail_sum))
```

```
Test Result: All samples num is 25, Correct/Failure:25/0
```

Test time complexity

In [ ]:

```
%%timeit

X_is = gen_data(0, 5, 1, 5)

kd_tree.search_knearest(X_i)
rst2 = kd_tree.kbest
```

```
1.39 ms ± 101 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [ ]:

```
%%timeit

X_is = gen_data(0, 5, 1, 5)

rst1 = exhuast_search(X_i, X_train)[:3]
```

```
5.6 ms ± 61.7 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```