KD - Tree

Giovani de Almeida Valdrighi - FGV EMAp

Objetivo

- Um fazendeiro deseja encontrar uma fazenda para efetuar compra.
- As fazendas são definidas por 5 características:
  - Produção mensal de morangos (de 500kg a 900kg)
  - Área de propriedade (de 40 a 100 hectares)
  - Valor de venda (de 1 a 5 milhões)
  - Longitude
  - Latitude

Objetivo

- Para encontrar as melhores fazendas, o fazendeiro apresentará três valores:
  - A descrição de uma fazenda ideal com as 3 primeiras características.
  - Sua localização.
  - Um número X número de opções que ele deseja receber.
- A resposta deve ser formada pelas X fazendas que estão mais próximas da descrição ideal e com um destaque para as 2 fazendas mais próximas da sua localidade.

Por que KD-Tree? Por que duas?

- As árvores KD são adaptadas para lidar com dados com qualquer dimensionalidade e permitem a busca de uma vizinhança de um ponto de forma eficiente.
- A utilização de duas KD-Tree facilita a operação pois iremos buscar focando em dois conjuntos de coordenadas diferentes.

O que é uma fazenda “próxima”?

- Iremos inicialmente buscar fazendas mais próxima considerando os três primeiros valores: produção de morangos, tamanho e preço.
- Todos possuem ordem de escala diferente, por esse motivo iremos normalizar os dados para o intervalo [0, 1].
- A distância entre duas fazendas será a distância euclidiana.

Implementação

Detalhes

- Implementação feita em Python.
- Estruturas:
  - KD Tree
  - Node
  - Bounded Priority Queue

Nó

- Cada nó possui um *list* que representa um ponto de dimensão dim, armazena seu valor *k* de consulta e referência para os filhos da esquerda e direita.

```python
class Node:
    """Node of KD-Tree, the value k is the dimension of lookup of the node."""
    def __init__(self, point, k = 0):
        self.point = np.array(point)
        self.k = k
        self.left = None
        self.right = None
```

KD-Tree

- A KD-Tree possui um inteiro que representa sua dimensão e um nó raiz.
- As funções que a árvore possui são de inserção, inserção balanceada, pesquisa pelos k pontos mais próximos e plot.

KD-Tree inserção

```
30    def insert(self, point, curNode = -1):
31        """
32        Recursive insertion function, starts at the root,
33        walks the structure verifying if what side to follows.
34        Assumes all points are different.
35        """
36        #It the tree is empty, create the root node
37        if self.root == None:
38            self.root = Node(point, k = 0)
39            return
40
41        #If this is the first call of intersetion, start at the root
42        if curNode == -1:
43            curNode = self.root
```

```
45        k = curNode.k
46        #New point is on left subtree
47        if point[k] < curNode.point[k]:
48            #if child exists, call recursive
49            if curNode.left != None:
50                self.insert(point, curNode.left)
51            #if child don't exists, create child
52            else:
53                curNode.left = Node(point, k = (k + 1)%self.dim)
54        #New point is on right subtree
55        else:
56            #if child exists, call recursive
57            if curNode.right != None:
58                self.insert(point, curNode.right)
59            #if child don't exists, create child
60            else:
61                curNode.right = Node(point, k = (k + 1)%self.dim)
```

KD-Tree
inserção balanceada

```python
    def insert_many(self, points, curNode = -1, k = None):
        """
        Recursive function to add many points to the tree in a balanced manner.

        Inputs:
            points - numpy array of size [m, self.dim + q]
            curNode - Node to check if will have subtrees
            k - int dimension to sort
        """
        m = len(points)
        if curNode == -1:
            k = 0
            points = points[points[:, 0].argsort()] #Sort
            self.root = Node(points[m//2, :], k) #Add the middle to the root
            curNode = self.root

        #Separate in left, right, and sort
        k = (k + 1)%self.dim
        left_points = points[:m//2, :]
        left_points = left_points[left_points[:, k].argsort()]
        right_points = points[m//2+1:]
        right_points = right_points[right_points[:, k].argsort()]

        #Check if there will be a subtree, add the subtree
        if left_points.shape[0] > 0:
            curNode.left = Node(left_points[len(left_points)//2, :], k)
        if right_points.shape[0] > 0:
            curNode.right = Node(right_points[len(right_points)//2, :], k)

        #Check if it is necessary to call recursive
        if left_points.shape[0] > 1:
            self.insert_many(left_points, curNode.left, k)
        if right_points.shape[0] > 1:
            self.insert_many(right_points, curNode.right, k)
```

KD-Tree pesquisa k vizinhos

```
102     def k_neighbors(self, point, k):
103         """
104         Search for the [k] closest neighbors of [point] in the tree.
105         Start the bounded priority queue and call the rescursive search.
106 
107         Inputs:
108             point - numpy array of dim [self.dim + q]
109             k - int, number of neighbors
110         """
111         candidates = BoundedQueue(k)
112         curNode = self.root
113         self.recursive_search(point, curNode, candidates)
114         result = candidates.items
115         #Sort candidates based on distance
116         result = [(-dist, node.point) for dist, node in result]
117         result.sort(key = lambda x : x[0])
118         neighbors = [x[1] for x in result]
119         dists = [x[0] for x in result]
120         return neighbors, dists
```

KD-Tree
pesquisa k vizinhos

```python
    def recursive_search(self, point, curNode, candidates):
        """
        Recursive function to search for k neighbors.

        Inputs:
            point - numpy array of dim [self.dim + q]
            curNode - Node of current search
            candidates - bounded priority queue
        """
        if curNode == None:
            return

        dist = np.sqrt(np.square(point[:self.dim] - curNode.point[:self.dim]).sum())
        candidates.add(curNode, -dist)

        #Decide the side to search
        k = curNode.k
        search_left = False
        if point[k] < curNode.point[k]:
            search_left = True
            self.recursive_search(point, curNode.left, candidates)
        else:
            self.recursive_search(point, curNode.right, candidates)

        #Follow other side if necessary
        if (~(candidates.is_full()) |
            (abs(point[k] - curNode.point[k]) < -candidates.max_priority())):
            if search_left:
                self.recursive_search(point, curNode.right, candidates)
            else:
                self.recursive_search(point, curNode.left, candidates)

```

Bounded Priority Queue

- Será utilizada para a pesquisa dos k vizinhos mais próximos.
- Irá manter os nós percorridos e as distâncias do ponto pesquisado.
- No entanto, só irá manter no máximo os k vizinhos mais próximos, removendo elementos adicionais quando necessário.
- Implementada utilizando do pacote de Python heapq.

Bounded Priority Queue

```python
class BoundedQueue:
    """
    Bounded priority queue with max length equals to [self.max_size].
    The elements are inside the list self.items.
    self.items[0] contain the element with lowest dist.
    """
    def __init__(self, max_size):
        self.items = []
        self.size = 0
        self.max_size = max_size

    def add(self, node, dist):
        """Add a new element of the list (if there is size)."""
        heapq.heappush(self.items, [dist, node])
        self.size += 1

        if self.size > self.max_size:
            self.items.pop(0)
            self.size -=1

    def is_full(self):
        """Check if queue is full."""
        return self.size == self.max_size

    def max_priority(self):
        """Return the priority of the element with max priority."""
        return self.items[0][0]
```

Problema

Construção

- Para representar o problema, realizamos o seguinte processo:
  - Criamos n fazendas aleatórias. Latitude e longitude são coordenadas aleatórias de capitais do Brasil.
  - Normalizamos os valores, salvamos intervalos e criamos a árvore com os valores: (produção de morangos, tamanho, preço).
  - Para realizar uma busca, normalizamos a fazenda pesquisa com os mesmos valores anteriores.

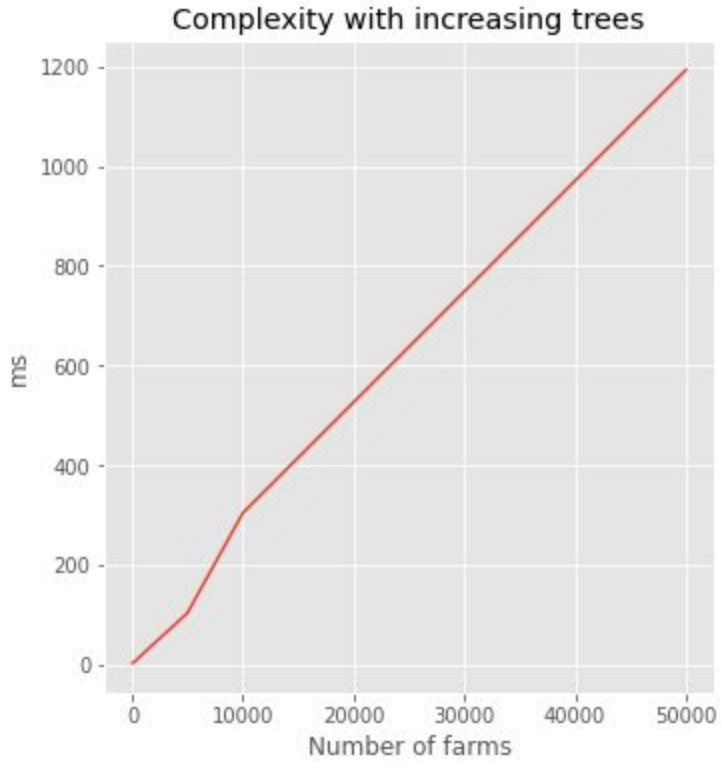
Construção

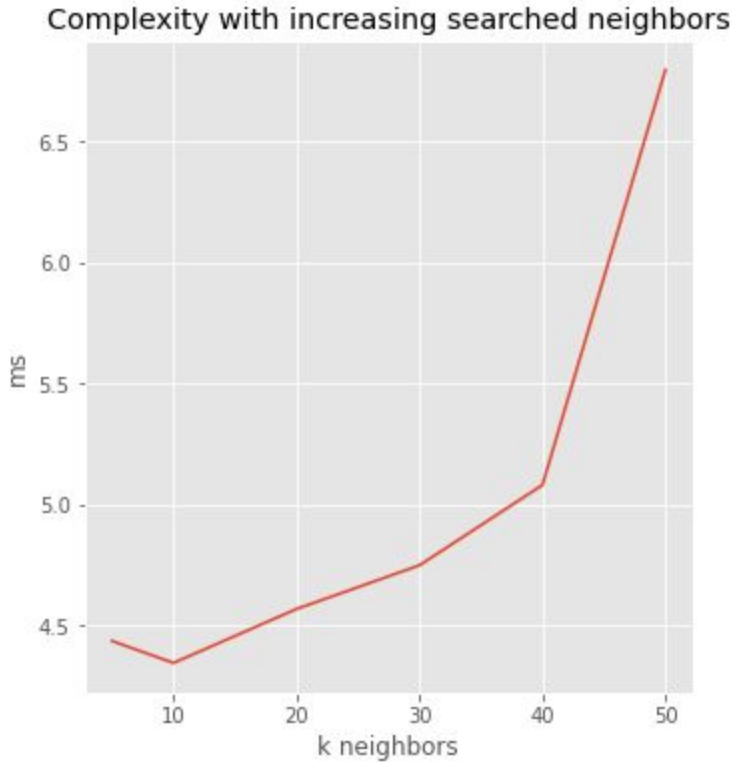
- Aos buscar k vizinhos, retornamos ordenados pela distância.
- Construímos uma segunda árvore com os k vizinhos, com os valores: (longitude, latitude).
- Buscamos baseado nas valores (longitude, latitude) da fazenda objetivo para encontrar as duas fazendas mais próximas.

Exemplo

```
1 farms = create_random_farms(50)
2 coordinates_my_city = [-47.74361, -23.165]
3 farm = np.array([600, 65, 3, coordinates_my_city[0], coordinates_my_city[1]])
4 complete_search(farms, farm, k)
```

```
My objective farm: [600.       65.        3.      -47.74361 -23.165  ]
First search:
Dist: 0.11, [634.62246197  67.87144379   3.11195168 -34.86305556  -7.115     ]
Dist: 0.183, [577.83163056  71.02830122   3.54559282 -46.63611111 -23.5475    ]
Dist: 0.2, [565.24806804  73.30699957   2.58703933 -48.50444444  -1.45583333]
Dist: 0.227, [619.28815922  74.59839016   3.57926313 -38.51083333 -12.97111111]
Dist: 0.262, [574.63548963  71.98783945   2.12436416 -40.33777778 -20.31944444]
Second search:
Dist: 638.946, [634.62246197  67.87144379   3.11195168 -34.86305556  -7.115     ]
Dist: 643.254, [619.28815922  74.59839016   3.57926313 -38.51083333 -12.97111111]
```


Complexity with increasing trees
1200
1000
800
600
400
200
0
ms
0
10000
20000
30000
40000
50000
Number of farms


Complexity with increasing searched neighbors
6.5
6.0
5.5
5.0
4.5
ms
10
20
30
40
50
k neighbors