Árvore k-d

Lucas Emanuel Resck Domingues

Estruturas de Dados e Algoritmos
Professor: Jean Roberto Ponciano

2 de setembro de 2021

Enunciado

A Secretaria de Saúde do seu município acaba de te contratar para ajudá-los em um problema. Você precisa oferecer um meio de repassar, para um fiscal dessa secretaria que circula pela cidade, as informações dos pacientes que possuem características parecidas com as que ele está procurando e que estão sendo atendidos no hospital mais próximo a ele. Com essa informação, ele poderá ir ao hospital e, se for preciso, tomar decisões importantes em relação à saúde pública da cidade.

A implementação da sua ideia depende de algumas informações:

1. Você precisa saber qual é o hospital de interesse (qual é o hospital mais próximo de onde o fiscal está?)
2. As características importantes para o fiscal são o peso do paciente (40 kg a 130 kg), idade (15 a 90 anos) e nível de saturação (80% a 100%). Ele informará os valores para esses parâmetros e você deve identificar os x pacientes que melhor se encaixam neles.
3. O fiscal informará a quantidade de pacientes que interessa a ele ($1 \leq x \leq 6$) e você os exibirá em ordem do mais semelhante ao menos semelhante. Por conveniência, você pode assumir que, qualquer que seja o hospital, há sempre 7 pacientes ou mais em atendimento.

Mão na massa:

1. Você deve resolver este problema utilizando duas árvores k-d.
2. Você não pode utilizar implementações prontas de bibliotecas como a Scikit-learn.
3. Você pode implementar em qualquer linguagem de programação.
4. Crie um dataset com, no mínimo, 10 hospitais. Embora estejamos assumindo um contexto local (município), para facilitar você pode escolher localizações como se os hospitais estivessem em cidades, estados ou países diferentes.
5. Crie um dataset com ao menos 7 pacientes para cada hospital. Você pode usar valores aleatórios dentro dos intervalos mencionados anteriormente, mas é importante salvar o arquivo para compararmos o resultado obtido com os valores do dataset.
6. Atente-se à medida de distância que você usará.

1 Implementação

Toda a implementação da *k-d tree* e da *bounded priority queue* foram realizadas em *Python*, podendo ser conferidas em https://github.com/lucasresck/data-structures-algorithms/blob/main/trees/scripts/k_d_tree.py.

1.1 *k-d tree*

A *k-d tree*, implementada em *Python*, é construída utilizando listas aninhadas. Cada lista guarda três elementos: a subárvore da esquerda (uma lista), o nó (um dicionário) e a subárvore da direita (outra lista). Por exemplo, adicionando os nós $(51, 75)$ e $(25, 40)$, nessa ordem, a árvore fica

```
[[[], {'node': [25, 40], 'metadata': None}, []], {'node': [51, 75], 'metadata':
None}, []]
```

Ou seja, $(51, 75)$ é a raiz e $(25, 40)$ é a raiz da subárvore esquerda (em azul).

A implementação permite a inserção de metadados, que ficam armazenados no dicionário do nó sob a chave *metadata*, algo que é importante para nosso cenário de uso, como veremos na Seção 2.

O cálculo da distância entre dois pontos é calculado da forma tradicional, utilizando a distância euclidiana. A função para adicionar novos nós é análoga à teoria, adicionando os nós de forma recursiva na subárvore correspondente, até que se atinja uma folha e efetivamente se adicione o nó na árvore. O k-NN também é análogo à teoria, buscando os k vizinhos mais próximos nas duas subárvores, porém descartando aquelas subárvore onde não é possível ter um vizinho mais próximo. Quando $k > 1$, é necessário guardar os melhores candidatos a vizinhos mais próximos em uma *bounded priority queue*, detalhada na Subseção 1.2.

1.2 Bounded priority queue

Por padrão, o *Python* já possui uma implementação de *priority queue* (PriorityQueue, do pacote queue), que podemos utilizar para simplificar a implementação da nossa *bounded priority queue*. Na nossa implementação, os elementos são armazenados da forma (prioridade, elemento).

Nossa classe BoundedPriorityQueue herda da classe PriorityQueue e a extende de tal forma que, quando o número máximo de elementos é atingido, novos elementos só são adicionados quando têm maior prioridade. Isso é feito mantendo salvo o valor da prioridade mínima. Veja um *snippet* do código para adicionar um novo elemento:

```
if not self.full():
    self.put((-priority, node))
    if priority > self.max_priority:
        self.max_priority = priority
else:
    if priority < self.max_priority:
        self.get()
        self.put((-priority, node))
        self.max_priority = priority
```

Outra modificação é que, na nossa aplicação, um nó vizinho com alta prioridade é aquele que está próximo do nó que está sendo buscado, ou seja, menores distâncias têm maior prioridade. Como usamos a distância como prioridade, multiplicamos a prioridade por -1 e trabalhamos com prioridade máxima ao invés de mínima.

2 Cenário de uso

O cenário de uso da nossa implementação consiste em:

- dada a localização do fiscal, indicar o hospital mais próximo;
- dado o hospital, indicar os x pacientes mais similares ao informado.

Para isso, precisamos criar um *dataset* com essas informações.

Toda a preparação do cenário de uso, incluindo a geração do *dataset*, podem ser visualizadas no *Jupyter Notebook* em https://nbviewer.jupyter.org/github/lucasresck/data-structures-

algorithms/blob/main/notebooks/k_d_tree.ipynb. O *dataset*, por sua vez, pode ser consultado em https://github.com/lucasresck/data-structures-algorithms/blob/main/data/hospitals.json.

2.1 *Dataset*

Nosso *dataset* contém 10 hospitais, com 7 pacientes cada. Por facilidade, cada hospital possui uma localização aleatória dentro de um quadrado unitário, e cada paciente possui características aleatórias dentro dos limites informados no enunciado. Veja, por exemplo, o início do arquivo JSON gerado:

```
{
    "hospitais": [
        {
            "localização": [
                0.62,
                0.61
            ],
            "pacientes": [
                {
                    "peso": 91,
                    "idade": 29,
                    "saturação": 90
                },
                {
                    "peso": 111,
                    "idade": 75,
                    "saturação": 100
                }, ...
```

Cada hospital é um objeto que contém sua localização e uma lista de seus pacientes; o *dataset*, portanto, é uma lista desses hospitais.

2.2 Hospital mais próximo

Suponhamos que o fiscal esteja na posição $(0.5, 0.5)$ do quadrado unitário. A Figura 1 mostra a localização dos hospitais e do fiscal.

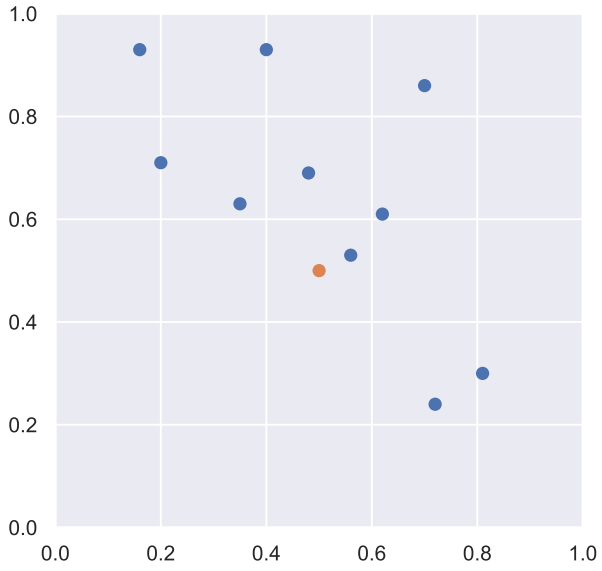

Figura 1: Localização dos hospitais (em azul) e do fiscal (em laranja).

Depois de montarmos a árvore k-d com dimensão $k = 2$ utilizando a localização dos hospitais como nó, realizamos uma busca k-NN com k igual a 1 para encontrarmos o hospital mais próximo. A saída do algoritmo é

```
[{'node': [0.56, 0.53],
  'metadata': {'localização': [0.56, 0.53],
   'pacientes': [{'peso': 98, 'idade': 50, 'saturação': 98},
    {'peso': 129, 'idade': 81, 'saturação': 98},
    {'peso': 59, 'idade': 85, 'saturação': 99},
    {'peso': 72, 'idade': 54, 'saturação': 86},
    {'peso': 121, 'idade': 15, 'saturação': 90},
    {'peso': 96, 'idade': 64, 'saturação': 89},
    {'peso': 46, 'idade': 30, 'saturação': 95}]}}]
```

Encontramos o hospital localizado no ponto $(0.56, 0.53)$, realmente o ponto mais próximo na Figura 1. Além disso, pelo fato de que nossa implementação permite o armazenamento de metadados, temos também todas as informações desse hospital, incluindo a lista de pacientes e suas características.

2.3 Pacientes mais parecidos

Suponhamos que as características dos pacientes do hospital mais próximo (Subseção 2.2) são adicionadas em uma nova k-d tree, de dimensão $k = 3$, sem nenhum tratamento. Quando fôssemos buscar os vizinhos mais próximos utilizando a distância euclidiana, teríamos, estaríamos priorizando algumas características em relação em outras, neste caso aquelas características com maior valor absoluto. Por exemplo, uma pessoa com saturação 80% é completamente diferente de uma com saturação 100%, porém, em valor absoluto, isso equivale à diferença de peso entre uma pessoa de 80 kg e 100 kg, que são parecidas neste contexto. Para evitar priorizar características, antes de inserirmos os pacientes na árvore normalizamos as *features* para valores entre 0 e 1, utilizando a seguinte função:

```
def normalize(weight, age, saturation):
    return (weight-40)/90, (age-15)/75, (saturation-80)/20
```

Tendo a árvore k-d montada utilizando as características normalizadas, podemos consultar os pacientes mais similares. Por exemplo, suponha que o fiscal esteja buscando os três pacientes mais similares àquele com as seguintes características: peso 46 kg, idade 30 e saturação 96%. Utilizando uma busca k-NN, com $x = 3$, temos a seguinte saída:

```
[{'node': [0.06666666666666667, 0.2, 0.75],
  'metadata': {'peso': 46, 'idade': 30, 'saturação': 95}},
 {'node': [0.6444444444444445, 0.4666666666666667, 0.9],
  'metadata': {'peso': 98, 'idade': 50, 'saturação': 98}},
 {'node': [0.35555555555555557, 0.52, 0.3],
  'metadata': {'peso': 72, 'idade': 54, 'saturação': 86}}]
```

Isto é, o paciente mais parecido tem as mesmas características, com exceção da saturação, que variou em 1% (46 kg, 30 anos e 95%). Nessa saída do algoritmo, podemos observar que as chaves nos nós foram armazenadas no formato normalizado.