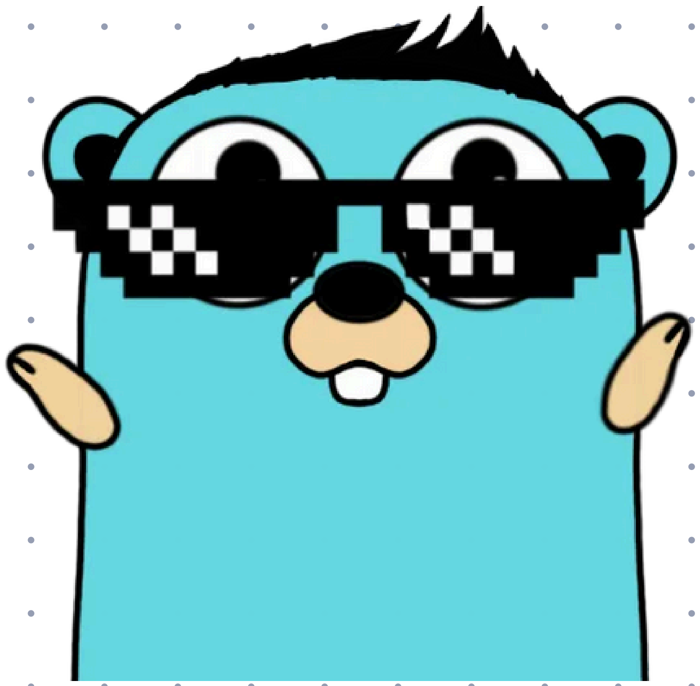
Análise de algoritmos

K-VIZINHOS MAIS PRÓXIMOS

Baseado no livro Entendendo Algoritmos, Um guia ilustrado para programadores e outros curiosos. Aditya Y. Bhargava, Novatec, 2017.



Análise de algoritmos

K-VIZINHOS MAIS PRÓXIMOS

Pense numa situação hipotética: você necessita ir ao mercado e ao chegar, se depara com uma gôndula cheias de frutas parecidas. Para identificarmos quais são as frutas, é possível escolhermos uma fruta dentre muitas para classificar, posteriormente selecionando uma amostra considerável para comparação, chamados vizinhos próximos.

O algoritmo de k-vizinhos (KNN) mais próximos ajuda a determinar os vizinhos do objeto-base e agrupamento por similaridade, considerando características boas ou ruins de forma imparcial, idealmente. O tempo de execução do KNN no Big O é de **O(n *d)**, sendo:

- **n** sendo o número de pontos de dados no conjunto de treinamento e;
- **d** correspondente à dimensionalidade dos dados (número de atributos).

A tarefa mais importante consiste em determinar a distância entre a fruta e todos as outras frutas do conjunto de dados. Existem diversas

K-VIZINHOS MAIS PRÓXIMOS

abordagens para chegarmos no valor desejado, como:

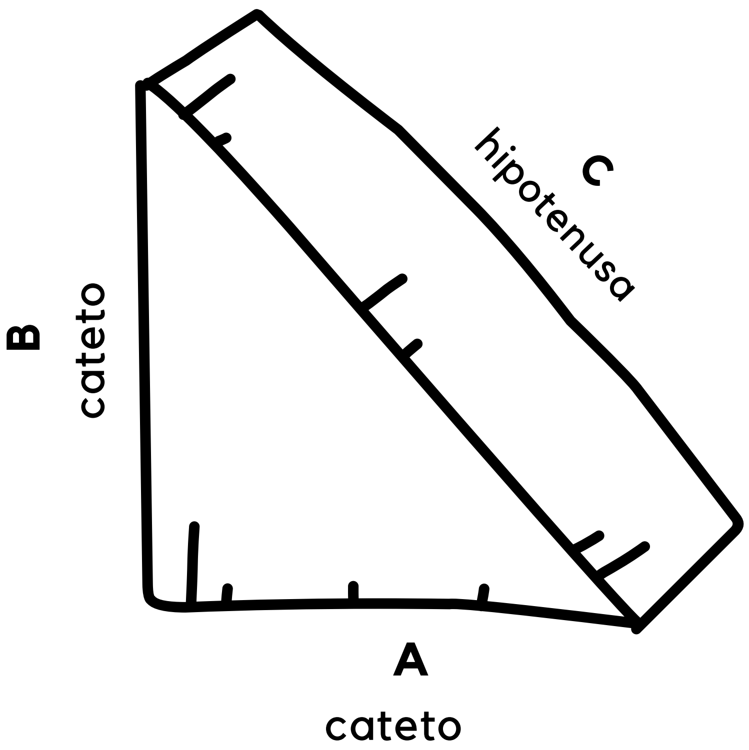
- Distância Euclidiana: distância linear entre pontos, considerando o caminho mais curto;
- Distância de Manhattan: cálculo da distância de lados de um retângulo, considerando o caminho mais longo;
- Distância de *Minkowski*: generalização da distância euclidiana e de *Manhattan;*
- Distância de *Hamming*: aplicada aos dados binários, contando o número de posições que os *bits* se diferem;
- Distância do Coseno: mede a similaridade entre dois vetores, útil para dados textuais.

O teorema de Pitágoras é uma maneira simples de determinar a distância entre objetos no gráfico, considerando um conjunto de dados, seguindo o príncipio da distância euclidiana. De maneira sucinta, o teorema de

Análise de algoritmos

K-VIZINHOS MAIS PRÓXIMOS

$$a^2 + b^2 = c^2$$



Pitágoras estabelece relação entre lados de um triângulo retângulo por meio da afirmação de que o quadrado da medida da hipotenusa é igual à somente dos quadrados das medidas dos catetos.

Para descobrirmos as coordenadas dos objetos no gráfico, é necessário traçarmos uma reta no eixo *X* e *Y*, determinando uma escala de valores. Interligando os pontos no gráfico, é possível determinar as coordenadas, adaptando na fórmula do Teorema de Pitágoras.

É importante mencionar que cada ponto corresponde a uma fruta, sendo seu grau de similaridade determinado pela distância.

/evelynthrossell

Análise de algoritmos

K-VIZINHOS MAIS PRÓXIMOS

Foi mencionado que uma forma de separar os objetos é por meio da classificação, mas temos também a possibilidade de trabalhar com a regressão, que nada mais é do que a média de valores dos objetos para advinhar uma resposta.

APLICAÇÕES

Amplamente empregado por fazer uso de boas características para determinar a similaridade entre objetos, o algoritmo do k-vizinhos mais próximos encontra-se nos seguintes setores:

CIÊNCIA DA COMPUTAÇÃO

- Recomendação de produtos: sugestão de produtos direcionados baseado em histórico de compras e adquiridos por usuários de perfis semelhantes;
- Sistemas de filtragem: filtrar conteúdo de acordo com preferências do usuário;

Análise de algoritmos

K-VIZINHOS MAIS PRÓXIMOS

- Detecção de anomalias: identificar padrões atípicos em dados em fraudes em transações financeiras ou falhas em sistemas.

ENGENHARIA

- Controle de qualidade: classificar proditos defeituosos ou não baseados em características;
- Predição de falhas: prever falha de equipamentos industriais baseados no histórico de manuteção;
- Reconhecimento de padrões: identificar padrões em dados de sensores para otimizar processos.

BIOINFORMÁTICA

- Classificação de genes: classificação de acordo com categorias funcionais;
- Predição de estruturas de proteínas: estrutura tridimensional de uma proteína baseado em sequência de aminoácidos;

Análise de algoritmos

K-VIZINHOS MAIS PRÓXIMOS

- Expressão gênica: identificar genes em diferentes condições.

FINANÇAS

- Previsão de preços de ações: baseado em dados anteriores;
- Detecção de fraudes: transações financeiras fraudulentas;
- Análise de crédito: avaliar risco de crédito de um cliente.

MARKETING

- Segmentação de clientes: campanhas direcionadas à um grupo com interesses ou características em comum;
- Comportamento do consumidor: preferências e hábitos de consumo.

GEOCIÊNCIAS

- Classificação de solos: propriedades físicas e químicas;
- Previsão de eventos naturais: dados geofísicos para previsão.

Análise de algoritmos

K-VIZINHOS MAIS PRÓXIMOS

PROCESSAMENTO DE IMAGENS E VÍDEOS

- Reconhecimento de padrões: identificar objetos em imagens ou vídeos;
- Segmentação de imagens: dividir imagem em regiões com características semelhantes.

Análise de algoritmos

K-VIZINHOS MAIS PRÓXIMOS

A partir de agora, analisaremos um problema de K-Vizinhos Mais Próximos chamado o mínimo de espaço total desperdiçado com k operações de redimensionamento.

Você está atualmente projetando um array dinâmico. Você recebe um *array* de inteiros **indexado por 0**, *nums*, onde *nums[i]* é o número de elementos que estarão no *array* no tempo *i*. Além disso, você recebe um inteiro *k*, o número **máximo** de vezes que você pode **redimensionar** o *array* (para qualquer tamanho).

O tamanho do *array* no tempo *t*, *sizet*, deve ser pelo menos *nums[t]* porque precisa haver espaço suficiente no *array* para armazenar todos os elementos. O **espaço desperdiçado** no tempo *t* é definido como *sizet - nums[t]*, e o espaço **total** desperdiçado é a **soma** do espaço desperdiçado em todos os tempos *t* onde *0 <= t > nums.length*.

Análise de algoritmos

K-VIZINHOS MAIS PRÓXIMOS

Retorne o mínimo de espaço total desperdiçado se você puder redimensionar o *array* no máximo *k* vezes.

Observação: o *array* pode ter **qualquer tamanho** no início e **não** conta para o número de operações de redimensionamento.

Exemplo 1:
Entrada: *nums* = [10,20], *k* = 0
Saída: 10
Explicação: *size* = [20,20]
Podemos definir o tamanho inicial como 20.
O espaço total desperdiçado é (20 - 10) + (20 - 20) = 10.

Exemplo 2:
Entrada: *nums* = [10,20,30], *k* = 1
Saída: 10

Análise de algoritmos

K-VIZINHOS MAIS PRÓXIMOS

Explicação: *size* = [20, 20, 30]
Podemos definir o tamanho inicial como 20 e redimensionar para 30 no tempo de 2.
O espaço total desperdiçado é (20 - 10) + (20 - 20) + (30 - 30) = 10.

Exemplo 3:
Entrada: *nums* = [10,20,15,30,20], *k* = 2
Saída: 15
Explicação: *size* = [10,20,20,30,30]
Podemos definir o tamanho inicial como 10, redimensionar para 20 no tempo 1 e redimensionar para 30 no tempo 3.
O espaço total desperdiçado é (10 - 10) + (20 - 20) + (20 - 15) + (30 - 30) + (30 - 20) = 15.

Restrições:

- 1 <= *nums.length* <= 200

Análise de algoritmos

K-VIZINHOS MAIS PRÓXIMOS

- 1 <= *nums[i]* <= 10^6
- 0 <= k <= nums.length - 1

K-VIZINHOS MAIS PRÓXIMOS

```go
package main

import (
	"fmt"
	"math"
)

func minimumValue(a, b int) int {
	if a < b { return a }
	return b
}
```

/evelynthrossell

K-VIZINHOS MAIS PRÓXIMOS

```go
func maximumValue(a, b int) int {
      if a > b { return a }
      return b
}

const (
     MaximumArraySize = 201
     MaximumNumberOfResizes = 201
)

var t [MaximumArraySize]
[MaximumNumberOfResizes]int
```

/evelynthrossell

K-VIZINHOS MAIS PRÓXIMOS

```
func calculateMinimumWastedSpace(currentIndex, k
int, nums []int) int {
      n := len(nums)
      if currentIndex == n { return 0 }
      if k < 0 { return math.MaxInt32 }
      if t[currentIndex][k] != -1 {
          return t[currentIndex][k]
      }

      maximumRequiredSize := nums[currentIndex]
      totalElements := 0
      minimumWastedSpace := math.MaxInt32
```

/evelynthrossell

K-VIZINHOS MAIS PRÓXIMOS

```
    for endIndex := currentIndex; endIndex < n;
endIndex++ {
        maximumRequiredSize =
maximumValue(maximumRequiredSize, nums[endIndex])
        totalElements += nums[endIndex]
        minimumWastedSpace =
minimumValue(minimumWastedSpace,
maximumRequiredSize * (endIndex - currentIndex +1) -
totalElements +
calculateMinimumWastedSpace(endIndex +1, k -1,
nums))
```

/evelynthrossell

K-VIZINHOS MAIS PRÓXIMOS

```
        }
        t[currentIndex][k] = minimumWastedSpace
        return minimumWastedSpace
}

func minimumSpaceWastedResizing(nums []int, k int) int
{
        for currentIndex := 0; currentIndex <
MaximumArraySize; currentIndex++ {
                for endIndex := 0; endIndex <
MaximumNumberOfResizes; endIndex++ {
                        t[currentIndex][endIndex] = -1
```

K-VIZINHOS MAIS PRÓXIMOS

```
        }
    }
    return calculateMinimumWastedSpace(0, k, nums)
}

func main() {
    nums := []int{10, 20}
    k := 0
    result := minimumSpaceWastedResizing(nums, k)
    fmt.Println("Minimum total space wasted to the first
example:", result)
```

K-VIZINHOS MAIS PRÓXIMOS

```
	nums = []int{10, 20, 30}
	k = 1
	result = minimumSpaceWastedResizing(nums, k)
	fmt.Println("Minimum total space wasted to the
second example:", result)

	nums = []int{10, 20, 15, 30, 20}
	k = 1
	result = minimumSpaceWastedResizing(nums, k)
	fmt.Println("Minimum total space wasted to the
third example:", result)
}
```

SAÍDA DO PROGRAMA

CENÁRIO OTIMISTA

Minimum total space wasted to the first example: 10
Minimum total space wasted to the second example: 10
Minimum total space wasted to the third example: 15

Análise de algoritmos

CONSIDERAÇÕES

Para o exemplo apresentado, seguiu-se as boas práticas de desenvolvimento com *clean code* e SOLID, na tentativa de simular um cenário otimista.
Essa iniciativa vai de encontro com a ideia de trazer conteúdos relevantes altamente abordados em processos seletivos e desmitificar a ideia de algoritmos e estrutura de dados. Espero que seja de bom proveito e bons estudos.