Trabalho Prático 1 - Algoritmos 2

Leandro Diniz silva

Matrícula: 2019105718

Departamento de Ciência da computação - Universidade Federal de Minas Gerais(UFMG)

leandrodinizsilva9@gmail.com

1. Introdução

Este trabalho prático tem como objetivo usar os algoritmos estudados em aula para produzir um programa capaz de atender as especificações do mesmo, com isso melhorando o entendimento por parte do aluno sobre implementação de algoritmos que interagem sobre árvores KD.

Para solucionar esse problema foi implementado a classe ***Node***, que representa uma folha da árvore KD, ***KDtree*** que contém as funções para gerar uma árvore KD, ***xNN*** que interage sobre a árvore buscando os "N" vizinhos de um dado ponto e a classe ***analisaResultados*** que exibe os resultados para o usuário final.

Essa documentação mostra uma visão geral do sistema, seus principais componentes usados para desenvolver o sistema além de sua implementação.

2. Estruturas de Dados

A classe *Node*, foi a primeira estrutura de dados implementada ela é um Node com três atributos, esses são:

Esquerda, salva o node a "esquerda" do node atual
Direita, salva o node a "direita" do node atual
Data, salva o valor do node atual

Para implementar a *KDtree* fora utilizado uma sequencia de nodes a esquerda e a direita de um node raiz, gerando assim uma árvore KD de acordo com a função "*constroiKDtree",* que segue o pseudo-código apresentado no slide da aula 6.

Ao se buscar pelos N vizinhos mais próximos de um certo ponto foi necessário a utilização de diversas estruturas de dados em conjunto, além da árvore KD foi usado também uma lista python que simula um *maxheap*, pois o python implementa em seu código base somente uma heapq, que é uma *minheap*, essa heap precisava ser uma *max* pois ela precisa retirar seu maior valor sempre que um atributo menor entrar nesse *heap.*

Como contêiner foi utilizado de um dataframe pandas, que facilita a leitura dos arquivos *.dat* diretamente para seu formato de leitura dos dados

3. Implementação

O principal algoritmo desenvolvido neste trabalho foi o *encontraNN*, que pode ser explorado na classe *xNN*, o mesmo utiliza de uma árvore KD já gerada anteriormente pelo algoritmo.

Com essa árvore já gerada na função *mainNN* a raiz da mesma é passada para o *encontraNN*, em conjunto com duas listas de tamanho N, "*proximos"* e "*proximosIndex"*, a lista *proximos* salva um heap setado inicialmente com todos seus valores como infinito positivo e a *proximosIndex* inicia com todos seus valores em nulo. A lista *proximos* salva a distância dos valores encontrados atualmente como mais próximos do ponto e a *proximosIndex* salva o ID que esses valores se encontram no dataframe.

A função *encontraNN* tem como passo base verificar se o node passado para ela no momento é vazio e se sim retorna a função sem nenhum valor, caso esse valor não seja nulo o seu próximo "*else if"* verifica se o node atual é uma folha, analisando se o node atual a sua esquerda ou direita tem seus valores nulos e se verdadeiro retorna o mesmo como folha, se ele for uma folha calcula sua distância do ponto atual e verifica se a mesma é menor que pelo menos um valor encontrado dentro do *heapmax proximos*, caso essas duas condições sejam verdadeiras, o mesmo atualiza o *heap* e retorna a função, a última verificação feita pelos "*IFs"* principais do *encontraNN* é caso o node atual não seja uma folha, executar a função *encontraNN* tanto na esquerda do node atual quando na sua direita, garantindo assim a recursividade do sistema por toda a árvore.

Na classe *mainNN* existe a função de controle que é quem executa todas as outras funções de sua classe na ordem correta, a primeira que ela executa é a explicada acima *encontraNN*, mas antes de executar a mesma ela garante a proporção de treino e teste setenta-trinta pelo método panda."*sample*" que de acordo com o atributo "*frac"* passado recupera uma porcentagem aleatória da base dados, garantindo assim também sua aleatoriedade.

A parte de treino é igual a setenta por cento da base de dados e a parte de teste é igual a trinta por cento, a parte de treino é a que tem sua árvore KD gerada, a parte de teste é usada na função *encontraNN* onde cada tupla, tirando a coluna que tem nome "Class", é usada como um ponto a ter seus vizinhos encontrados, onde para cada ponto foi buscado somente os dez pontos mais próximos.

Com os dados agora na proporção correta e com os resultados mostrando seus vizinhos mais próximos é possível determinar qual a classe o ponto de teste o algoritmo consegue encontrar, para isso é executado uma função simples que conta quantas vezes uma classe aparece nos vizinhos e depois seleciona a que mais apareceu e essa vai ser a nova classe do ponto de teste.

Essa nova classe é adicionado em uma nova coluna no dataframe chamada de “NovaClass” e com o auxílio da biblioteca sklearn é gerada uma *“Confusion Matrix”* para encontrar os dados necessários para exibir a acurácia, precisão e revocação das novas classes encontradas pelo algoritmo de kNN.

4. Bases usadas

Foi usado um total de dez bases de dados encontradas no KEEL referenciado na descrição do trabalho, algumas dessas bases tiveram seus tamanhos reduzidos utilizando a função do pandas dataframe.samples(), para que a quantidade de dados usadas fosse em média de quinhentas tuplas. As bases usadas e as informações de Precisão, Revocação e Acurácia do algoritmo kNN de treino foram as seguintes:

Appendicitis
Instâncias: 106
Colunas: 8

```
df1 = lerDatParaPanda('data/appendicitis.dat')
df1 = df1.astype(np.float64)
xNNPrincipal = xNN()
result = xNNPrincipal.mainNN(df1, 10)
```

```
Precisão:  81.25 %
Revocação:  81.25 %
Acurácia:  81.25 %
```

Banana
Instâncias: 5300
Instâncias usadas (aleatoriamente): 1060
Colunas: 2

```
df2= lerDatParaPanda('data/banana.dat')
df2 = df2.sample(frac = 0.2)
df2= df2.astype(np.float64)
result = xNNPrincipal.mainNN(df2, 10)
```

```
Precisão:  52.2 %
Revocação:  52.2 %
Acurácia:  52.2 %
```

Bupa
Instâncias: 345
Colunas: 6

```
df3 = lerDatParaPanda('data/bupa.dat')
df3 = df3.astype(np.float64)
result = xNNPrincipal.mainNN(df3, 10)
```

```
Precisão:  62.5 %
Revocação:  62.5 %
Acurácia:  62.5 %
```

Contraceptive
Instâncias: 1473
Instâncias usadas (aleatoriamente): 294
Colunas: 9

```
df4 = lerDatParaPanda('data/contraceptive.dat')
df4 = df4.sample(frac = 0.2)
df4 = df4.astype(np.float64)
result = xNNPrincipal.mainNN(df4, 10)
```

```
Precisão:  35.23 %
Revocação:  35.23 %
Acurácia:  56.82 %
```

Glass
Instâncias: 214
Colunas: 9

```
df5 = lerDatParaPanda('data/glass.dat')
df5 = df5.astype(np.float64)
result = xNNPrincipal.mainNN(df5, 10)
```

```
Precisão:  18.75 %
Revocação:  18.75 %
Acurácia:  72.92 %
```

Page Blocks
Instâncias: 5472
Instâncias usadas (aleatoriamente): 1094
Colunas: 10

```
df6 = lerDatParaPanda('data/page-blocks.dat')
df6 = df6.sample(frac = 0.2)
df6 = df6.astype(np.float64)
result = xNNPrincipal.mainNN(df6, 10)
```

```
Precisão:  90.24 %
Revocação:  90.24 %
Acurácia:  96.1 %
```

Phoneme
Instâncias: 5404
Instâncias usadas (aleatoriamente): 1080
Colunas: 5

```
df7 = lerDatParaPanda('data/phoneme.dat')
df7 = df7.sample(frac = 0.2) #Diminuindo o tamanho dessa base de dados
df7 = df7.astype(np.float64)
result = xNNPrincipal.mainNN(df7, 10)
```

```
Precisão:  70.06 %
Revocação:  70.06 %
Acurácia:  70.06 %
```

Shuttle
Instâncias: 58000
Instâncias usadas (aleatoriamente): 580
Colunas: 9

```
df8 = lerDatParaPanda('data/shuttle.dat')
df8 = df8.sample(frac = 0.01) #Diminuindo o tamanho
df8 = df8.astype(np.float64)
result = xNNPrincipal.mainNN(df8, 10)
```

```
Precisão:  81.03 %
Revocação:  81.03 %
Acurácia:  92.41 %
```

Tae
Instâncias: 151
Colunas: 5

```
df9 = lerDatParaPanda('data/tae.dat')
df9 = df9.astype(np.float64)
result = xNNPrincipal.mainNN(df9, 10)
```

```
Precisão:  31.11 %
Revocação:  31.11 %
Acurácia:  54.07 %
```

Titanic
Instâncias: 2201
Instâncias usadas (aleatoriamente): 440,2
Colunas: 3

```
df10 = lerDatParaPanda('data/titanic.dat')
df10 = df10.sample(frac = 0.2)
df10 = df10.astype(np.float64)
result = xNNPrincipal.mainNN(df10, 10)
```

```
Precisão:  67.42 %
Revocação:  67.42 %
Acurácia:  67.42 %
```