Trabalho Prático 1
Aplicações de Algoritmos Geométricos

Otávio de Meira Lima - 2019054900
otavioml@ufmg.br

Universidade Federal de Minas Gerais
Belo Horizonte - Brasil

Disciplina: Algoritmos 2

1. Introdução

O problema proposto foi implementar um algoritmo geométrico para classificação baseado nos K vizinhos mais próximos. Tal problema pode ser solucionado por meio de um Aprendizado de Máquina Supervisionado. Nessa ocasião, o algoritmo recebe pontos de treinamento para ajustar o modelo que é utilizado para classificar pontos de teste.

2. Implementação e Estrutura de Dados

O algoritmo foi implementado na linguagem Python e está disponível em um repositório no github: https://github.com/otavioml/TP-Algorithms-2

2.1. Leitura das Bases de Dados

A leitura dos dados disponíveis no site Keel(disponível nas referências) foi feita por meio da biblioteca numpy. O arquivo read_files.py armazena todas as implementações referentes à leitura dos dados.

Os arquivos tiveram que ser alterados, retirando a documentação sobre as variáveis e classes no início do arquivo já que esse era um fator que atrapalhava a leitura.

Todos os valores dos pontos do arquivo são lidos como String, mas a função identifica floats e altera seus tipos, permitindo que a base de dados seja usada na construção da árvore kd, que será explicada posteriormente.

```python
def getDataPoints(filename):
    data = np.loadtxt(filename, delimiter=',', dtype=str)

    listofpoints = []

    for d in data:
        listofpoints.append(tuple(d))

    ans = []
    for tupl in listofpoints:
        temp = []
        for x in tupl:
            try:
                temp.append(float(x))
            except Exception:
                temp.append(x)
        ans.append(tuple(temp))

    return ans
```

Imagem 1:função para leitura de dados.

A função retorna uma lista, onde cada elemento é uma tupla que contém todos os valores dos eixos do ponto e a classe. Todos os pontos de todas as bases utilizadas são identificados por uma classe que sempre está localizada na última dimensão do ponto. Ou seja, se um ponto com 5 dimensões é lido de uma base de dados, o valor disponível na 5ª dimensão é a sua classificação e as 4 dimensões restantes são utilizadas para a construção e busca no algoritmo.

Exemplo de um ponto: (3.7, 4.5, 2.8, ‘positive’)

O valor da primeira dimensão do ponto é 3.7, a segunda é 4.5, a terceira é 2.8 e sua classificação é ‘positive’.

A função GetTrainingAndTestsPoints divide os dados, onde 70% são pontos de treinamento e 30% são pontos de teste. A função random.shuffle(data) foi utilizada para manter a aleatoriedade da seleção dos pontos a cada execução do algoritmo.

```python
def getTrainingAndTestsPoints(data):

    seventyPercent = int((70/100)*len(data))
    random.shuffle(data)

    trainingPoints = data[:seventyPercent]
    testPoints = data[seventyPercent + 1:]

    return trainingPoints, testPoints
```

Imagem 2: função para selecionar pontos de treinamento e teste.

No mesmo arquivo, é implementado uma função getUniqueClasses que captura todas as classificações de pontos da base de dados, retornando uma lista que será usada posteriormente para avaliar a precisão do algoritmo. Sabemos previamente que a classificação dos pontos está presente na sua última dimensão, logo basta percorrer todos os pontos e selecionar as classificações.

```
def getUniqueClasses(data):        You, a day ag
    classes = []
    dimension = getDimensions(data)

    for p in data:
        if classes.count(p[dimension-1]) == 0:
            classes.append(p[dimension-1])

    return classes
```

Imagem 3: função para capturar todas as classificações dos pontos da base de dados.

2.2. Árvore Kd

Árvore Kd é uma árvore binária de busca, onde cada nó possui até 2 filhos, usada para o particionamento de um espaço k-dimensional.

Cada nó interno representa o valor de um ponto na i-ésima dimensão e cada nó folha armazena uma tupla com todos os valores das coordenadas de um certo ponto.

A cada profundidade da árvore, o espaço é dividido em uma certa dimensão **D** determinada sequencialmente. Analisando todos os pontos apenas na dimensão **D**, é selecionado o ponto mediano **P** que divide a dimensão em 2, onde metade dos pontos estão à sua esquerda e a outra metade à sua direita, e seu valor é adicionado ao nó interno. Pontos com valor em tal dimensão menor que o ponto **P** estão localizados na subárvore da esquerda. Caso contrário, o ponto fica na subárvore da direita. Naturalmente, o ponto selecionado como mediana ficará localizado na subárvore da esquerda.

O algoritmo para a construção da árvore é executado novamente de forma recursiva, onde cada passo recebe metade do espaço, resultado da divisão feita anteriormente.

Quando o algoritmo recebe apenas um espaço com apenas 1 ponto, ele é armazenado no nó folha com todas as suas coordenadas, ao contrário dos nós internos que armazenam o valor de apenas uma coordenada.

```python
def kdtree(point_list, depth=0):
    try:
        k = len(point_list[0]) - 1
    except IndexError as e:
        return None

    if len(point_list) == 1:
        return Node(value=point_list[0],left=None,right=None)

    axis = depth % k

    point_list.sort(key=lambda x: x[axis])

    l = len(point_list)
    if l%2 == 0:
        median = int((l/2)-1)
    else:
        median = l // 2

    return Node(
        value=point_list[median][axis],
        left=kdtree(point_list[:median+1], depth + 1),
        right=kdtree(point_list[median + 1:], depth + 1)
    )
```

Imagem 4: função para construir a árvore kd.

A função implementada acima, presente no arquivo kd_tree.py, segue os mesmos passos da explicação anterior.

Essa estrutura de dados foi escolhida para armazenar os pontos, pois, apesar de usar mais espaço, é mais eficiente para a busca de pontos próximos. Sabendo o valor das coordenadas de um ponto **P**, podemos percorrer a árvore ignorando áreas do espaço em que há certeza que não há pontos próximos.

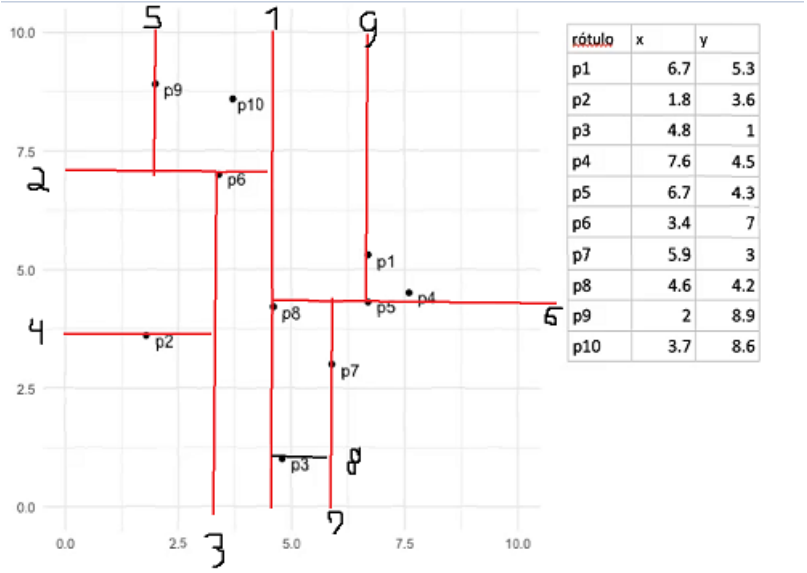
```python
class Node():
    def __init__(self, value, left, right):
        self.value = value
        self.left = left
        self.right = right
...
class Kdtree():

    def buildKdtree(point_list):
        return kdtree(point_list)
```

Imagem 5: classes Node e Kdtree.

No mesmo arquivo está a declaração das classes Node e Kdtree. A classe Node segue a implementação padrão como qualquer árvore binária. Já a classe Kdtree possui apenas uma função que constrói a árvore por completo e retorna o Nó raiz.

Como exemplificação, podemos tomar 10 pontos em um plano bidimensional e construir a árvore para facilitar a visualização.



rótulo	x	y
p1	6.7	5.3
p2	1.8	3.6
p3	4.8	1
p4	7.6	4.5
p5	6.7	4.3
p6	3.4	7
p7	5.9	3
p8	4.6	4.2
p9	2	8.9
p10	3.7	8.6

Imagem 6: plano com os pontos.

Cada divisão está numerada. É possível perceber que a cada iteração, uma das duas dimensões é dividida.

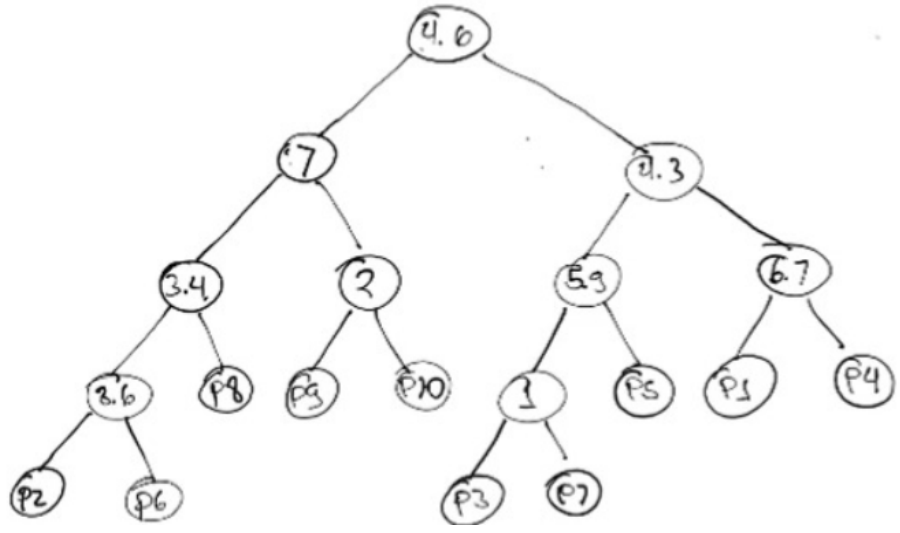


Imagem 7: exemplo de árvore kd.

Os pontos 9 e 1 foram separados em espaços diferentes na primeira divisão, logo estão em subárvores diferentes desde o primeiro nó.

2.3. K vizinhos mais próximos

A implementação do algoritmo para classificação por meio da seleção dos K vizinhos mais próximos foi feita em um arquivo separado, denominado xnn.py, para facilitar a compreensão e leitura do código.

A classe recebe uma lista de pontos de treinamento e constrói a árvore kd por meio da função implementada na própria classe Kdtree.

Após a construção do modelo, se recebe uma lista de pontos de teste e, para cada um deles, é calculado os k vizinhos mais próximos por distância euclidiana, que são armazenados na fila de prioridades que é um atributo da classe. K é um número arbitrário selecionado pelo usuário.

Sabendo-se os k vizinhos mais próximos, são capturados todas as suas classes e aquela mais frequente será usada para classificação do ponto. Sabendo-se sua classificação real e sua classificação por meio do modelo, é possível fornecer estatísticas de teste como precisão, revocação e acurácia.

Cada função da classe está explicada em detalhes abaixo:

- Xnn.euclidianDistance (cálculo para distância euclidiana):
  Utiliza função de álgebra linear do numpy (numpy.linalg.norm(a, b)).

- Xnn.k_nearest (cálculo dos k vizinhos mais próximos):

Essa função se baseia em outra função recursiva auxiliar k_nearestAux que percorre a árvore Kd. Recebe como argumentos a dimensão dos pontos, a quantidade k de vizinhos mais próximos, o ponto cujo vizinhos serão calculados e o nó atual da árvore Kd.

Caso o nó atual seja um nó interno, é necessário saber qual subárvore tem mais potencial para ter vizinhos mais próximos. Isso é calculado comparando o valor presente no nó com o valor do ponto na i-ésima dimensão. Caso o ponto esteja à direita do espaço dividido pelo nó (ou seja, seu valor é maior do que o presente no nó), a subárvore mais próxima é a da direita. O mesmo raciocínio se aplica para a subárvore da esquerda. A função é chamada recursivamente para a subárvore mais próxima e só é chamada para a subárvore oposta se a fila de prioridades ainda não estiver preenchida ou se o ponto de menor prioridade na fila pode ser substituído por algum ponto da subárvore.

Após a fila de prioridades estar cheia e não haver nenhuma subárvore com pontos potencialmente mais próximos, a fila de prioridades é retornada com todos os pontos próximos ordenados por distância euclidiana.

A fila de prioridades é uma lista de tuplas, onde o primeiro elemento é a distância entre os pontos e o segundo é o ponto em si. Dessa forma, é possível organizar os pontos de forma eficiente.

```python
def k_nearestAux(dimensions, k, point, current_node, priority_queue=[], depth=0):

    axis = depth % (dimensions-1)
    depth += 1

    if current_node.left == None and current_node.right == None:
        distance = euclideanDistance(point[:dimensions-1], current_node.value[:dimensions-1])
        if len(priority_queue) < k:
            heapq.heappush(priority_queue, (-distance,current_node.value))
            priority_queue = sorted(priority_queue)

        elif -distance < -priority_queue[0][0]:
            heapq.heappushpop(priority_queue, (-distance, current_node.value))
            priority_queue = sorted(priority_queue)

        return priority_queue

    else:

        if point[axis] > current_node.value:
            nearBranch = current_node.right
            opositeBranch = current_node.left
        else:
            nearBranch = current_node.left
            opositeBranch = current_node.right

        priority_queue = k_nearestAux(dimensions, k, point, nearBranch, priority_queue, depth)

        if len(priority_queue) < k or priority_queue[0][1][axis] <= abs(point[axis] - current_node.value):

            priority_queue = k_nearestAux(dimensions, k, point, opositeBranch, priority_queue, depth)

        return priority_queue
```

Imagem 8: função recursiva para capturar os x vizinhos mais próximos.

A fila de prioridades é construída de forma que o ponto com menor prioridade esteja na primeira posição da fila. Por isso é adicionado o valor negativo da distância euclidiana na fila para manter a ordem invertida.

- Xnn.getClassificationFromPQ (método para classificação do ponto):

Com os K vizinhos mais próximos armazenados em xnn.priority_queue, podemos pegar suas classificações. A classificação mais frequente dentre todos os pontos próximos será aquela em que o modelo classifica o ponto de teste.

```python
    def getClassificationFromPQ(self, dimensions):
        temp = []
        for p in self.priority_queue:
            temp.append(p[1][dimensions-1])

        return mode(temp)
```

Imagem 9: função para classificação do ponto de teste.

- Xnn.getStatisticsFromTestPoints (método para avaliar o modelo):

Esse método recebe uma lista de pontos de teste e suas classificações e, para cada um dos pontos, o modelo o classifica e verifica sua corretude.

A primeira classe recebida pela lista de classificação é utilizada como sendo a classe positiva e as classes diferentes dela são as negativas. Classes positivas classificadas corretamente são consideradas verdadeiras positivas. Caso contrário, são falsos negativos. Classes negativas classificadas corretamente são consideradas verdadeiros negativos. Caso contrário, são falsos positivos.

Sabendo a quantidade de verdadeiros positivos e negativos e falsos positivos e negativos, podemos calcular as estatísticas de teste como precisão, revocação e acurácia.

```
63      def getStatisticsFromTestPoints(self, k, test_point_list, classifications):
64
65          tp = fp = tn = fn = 0
66          dimensions = getDimensions(test_point_list)
67          i = 0
68          for point in test_point_list:
69              self.k_nearest(dimensions, k, point, self.kdtree)
70              classification = self.getClassificationFromPQ(dimensions)
71              if point[dimensions-1] == classifications[0]:
72                  if classification == point[dimensions-1]:
73                      tp += 1
74                  else:
75                      fn += 1
76              else:
77                  if classification == point[dimensions-1]:
78                      tn += 1
79                  else:
80                      fp += 1
81          try:
82              precision = tp/(tp + fp) * 100
83              revocation = tp/(tp+fn) * 100
84              accuracy = (tp+tn)/(tp+tn+fp+fn) * 100
85          except ZeroDivisionError:
86              precision = revocation = 0
87              accuracy = (tp+tn)/(tp+tn+fp+fn) * 100
88
89          print("Quantidades de vizinhos próximos calculados: ", k)
90          print("Precisão: ", round(precision, 2), "%")
91          print("Revocação: ", round(revocation, 2), "%")
92          print("Acurácia: ", round(accuracy, 2), "%")
```

Imagem 10: função para avaliação estatística do modelo.

3. Análises

3.1. Bases de Dados

Foram utilizadas 11 bases de dados com dimensões e quantidade de classificações diferentes, todas presentes na pasta “data” no formato .dat e originadas da ferramenta Keel.

Foram utilizados 5 vizinhos mais próximos e os resultados para todas as bases de dados com uma breve análise estão disponíveis a seguir:

Database: appendicitis
Precisão: 77.42 %

Revocação: 100.0 %
Acurácia: 77.42 %
Análise: Database com 7 dimensões, 2 tipos de classes e 106 instâncias. Precisão e acurácia alta. Com poucas instâncias e apenas duas classes, os pontos estão bem divididos no espaço, facilitando a classificação.

Database: haberman
Precisão: 72.53 %
Revocação: 100.0 %
Acurácia: 72.53 %
Análise: com 3 dimensões e 2 tipos de classes com 306 instâncias, a análise é semelhante à referente ao database anterior.

Database: pima
Precisão: 63.49 %
Revocação: 81.08 %
Acurácia: 57.83 %
Análise: com 8 dimensões e 2 tipos de classes com 768 instâncias, os pontos agora já não estão mais tão bem distribuídos, o que afeta todas as estatísticas.

Database: led7digit
Precisão: 1.59 %
Revocação: 16.67 %
Acurácia: 10.07 %
Análise: possui 7 dimensões, 10 tipos de classes com apenas 500 instâncias. Com um número baixo de instâncias e um número alto de classificações e dimensões, os pontos não estão distribuídos de forma heterogênea, o que afeta a classificação dos pontos de teste.

Database: monk-2
Precisão: 55.56 %
Revocação: 15.87 %
Acurácia: 52.71 %
Análise: possui 6 dimensões e 2 tipos de classes com 432 instâncias. Com número baixo de instâncias e avaliando as estatísticas, pode-se perceber que os pontos estão distribuídos de forma homogênea, o que dificulta a classificação.

Database: heart
Precisão: 58.7 %
Revocação: 61.36 %
Acurácia: 55.0 %
Análise: com 13 dimensões e duas classificações com apenas 270 instâncias, pode-se perceber que os pontos estão muito mais divididos por classificações no espaço devido à maior revocação comparado com a base de dados anterior.

Database: wdbc
Precisão: 58.24 %
Revocação: 100.0 %
Acurácia: 58.24 %
Análise: com 30 dimensões e 2 tipos de classes com 569 instâncias, pode-se perceber que os dados estão divididos no espaço de acordo por classificação de forma heterogênea. Com esse número alto de dimensões, é possível descartar algumas já que provavelmente nem todas são relevantes para a classificação.

Database: phoneme
Precisão: 70.88 %
Revocação: 90.43 %
Acurácia: 67.49 %
Análise: possui 5 dimensões com 2 tipos de classes e 5404 instâncias. Com essa quantidade alta de instâncias, pode-se perceber que as estatísticas ficam mais precisas, já que a tendência é de que os dados fiquem mais separados de acordo com classe com o aumento do número de pontos.

Database: iris
Precisão: 9.52 %
Revocação: 16.67 %
Acurácia: 34.09 %
Análise: possui 4 dimensões e 3 tipos de classes com 150 instâncias. Quantidade baixa de instâncias afeta todas as estatísticas.

Database: ecoli
Precisão: 33.0 %
Revocação: 100.0 %
Acurácia: 33.0 %
Análise: possui 7 dimensões e 6 tipos de classes com 336 instâncias. A alta quantidade de classificações e a baixa quantidade de instâncias levam as estatísticas para baixo.

Database: banana
Precisão: 56.26 %
Revocação: 100.0 %
Acurácia: 56.29 %
Análise: possui 2 dimensões e duas classificações com 5300 instâncias. A alta quantidade de dados aumenta o valor das estatísticas, mas a distribuição homogênea dos pontos pode não favorecer a classificação.

3.2. Complexidade

- Construção da Árvore Kd

A cada iteração da construção da árvore, a lista de pontos é ordenada em um certo eixo do espaço com custo nlogn. Logo, a construção da árvore possui custo de tempo da ordem de O(n*log n*log n), onde n é a quantidade de pontos presentes.

- K vizinhos mais próximos

Esse algoritmo depende não apenas das variáveis, mas também depende da distribuição dos pontos no espaço. Pontos podem estar próximos no espaço, mas podem estar muito distantes na árvore Kd dependendo das divisões do espaço que serão feitas nos nós internos da árvore.

- Percorrer lista de pontos

Funções auxiliares que percorrem listas ou até mesmo toda a base de dados, separando-as em pontos de treinamento e testes, possuem complexidade de tempo linear.

Concluindo, o algoritmo para determinar os K vizinhos mais próximos e retornar estatísticas de precisão, acurácia e revocação possui custo dominado pela construção da árvore Kd e pela captura dos vizinhos.

4. Conclusão

Concluindo o projeto, acredito que a utilização de algoritmos geométricos e estruturas de dados eficientes como a Árvore Kd foram essenciais para o entendimento da resolução do problema de classificação por meio do Aprendizado de Máquina.

5. Referências

https://pt.wikipedia.org/wiki/Precis%C3%A3o_e_revoca%C3%A7%C3%A3o
https://medium.com/computando-arte/o-que-%C3%A9-precis%C3%A3o-e-revoca%C3%A7%C3%A3o-b0b991b67cde
https://sci2s.ugr.es/keel/category.php?cat=clas