# Trabalho Prático 1

## Algoritmos 2

Luiz Henrique da Silva Gonçalves

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

Matricula:2018054559

Luiz10hdsg@ufmg,br

## 1. Objetivo

O trabalho prático propõe o desenvolvimento de uma ferramenta de análise de dados comumente usada em técnicas e práticas de machine learning. A proposta do sistema a ser implementado se trata em um classificar de pontos no plano , usando para tal a estrutura de dados de árvore Kd , possibilitando assim, diminuir a complexidade das pesquisas efetuadas durante o processo em razão de suas particularidades. Dessa forma, o sistema deve guardar os pontos das bases de treino nessa estrutura e buscar os pontos mais próximos para cada ponto requisitado no conjunto de teste. Ao final , o classificador deve retornar algumas métricas de interesse.

## 2. Linguagem de Programação

A implementação do sistema foi feita na linguagem de programação Python .

## 3. Código

Na implementação do sistema foram feitas três classes **:**

**class KDNode:**
Contem a função de inicialização dos no da arvore.Cada node da arvore possui os seguintes atributos:
value: Valor armazenado no node , so armazena um ponto se for a folha da arvore
label:Armazena o label do node.Esse valor se refere a partições da árvore kd
left:Subarvore da esqueda
right:Subarvore da direita
depth:Profundidade da arvore

```
class KDNode(object):
    def __init__(self,value,label,left,right,depth):
        self.value = value
        self.label =label
        self.left = left
        self.right = right
        self.depth = depth
```

**class arvoreKD:**

Possui a implementação da estrutura de dados árvore KD com as funções de inicialização do node root e a função de construção da arvore( build_KDtree)

O função de construção da árvore , funciona da seguinte forma:

Como parametros da função recebemos os pontos a serem armazenados e a profundidade da árvore. A arvore desenvolvida aqui armazena os pontos somente nos nodes folhas.A estrutura começa verificando os pontos a serem armazenados, em seguida percorremos a árvore verificando em qual eixo do plano é necessário dividir o plano, se a profundidade for par, dividimos o palno verticalmente , caso ao contrário horizontalmente.Dessa forma, pegamos os pontos e ordenamos conforme o eixo em questão e atribuímos a variável p onde é feita o corte, dividindo o plano. Em seguida , após dividir o plano em p , chamamos recursivamente a função de construção para cada subárvore da esquerda( lado menor que p) e para subarvore da direita(lado maior que p).Construímos um novo node e guardando p como elemento de corte.Repetimos esse processo até restar apenas um ponto a ser armazenado, nessa etapa guardamos  o ponto em questão no node corrente.

```
class arvoreKD(object):
    def __init__(self):
          self.root = KDNode(None,None,None,None,None)
          self.root = None

    def build_KDTree(self,points,depth):
      if len(points)==0:
        return None
      if len(points)==1:
        node=KDNode(points,0,None,None,depth)
        return node
      else:

        if depth % 2 ==0:
          k= len(points)
          axis = depth % k
          points.sort(key=itemgetter(0))
          median_idx = len(points)// 2
          p=points[median_idx][0]

        else:
          k= len(points)
          axis = depth % k
          points.sort(key=itemgetter(1))
          median_idx = len(points)// 2
          p=points[median_idx][1]

        vleft = self.build_KDTree(points[:median_idx],depth+1)
        vright = self.build_KDTree(points[median_idx:],depth+1)

        node=KDNode(0,p,vleft,vright,depth)
        return node
```

## 4. Etapas

**class x_NN:**
É nessa classe que fazemos a classificação dos pontos.Temos 4 funções nessa classe:

Responsável por calcular a distância entre dois pontos.

```
class x_NN(object):

  def dist(self,point1,point2):

        p1x=point1[0][0]
        p2x=point2[0]
        p1y=point1[0][1]
        p2y=point2[1]
        res=np.sqrt(((p2x-p1x)**2)+((p2y-p1y)**2))

        return np.sqrt(((p2x-p1x)**2)+((p2y-p1y)**2))
```

Retornar os vizinhos mais próximos ao ponto de pesquisa.

```
def find_nearest_neighbor(self, query, node, best_node, best_distance,axis,best_distances,best_neighbors):

  good_side =None
  bad_side =None

  if node==None:
    return best_neighbors

  if node.label==0: #quer dizer que achei o ponto na folha da arvore ccom label 0
    d=self.dist(node.value,query)      # calculo a distancia entre os pontos
    best_neighbors.append((node.value, d))
    if  d < best_distance:
      best_node = node
      best_distance = d

  else:

    #decidindo os lados da pesquina na arvore

     if node.depth %2 == 0: #verifico em qual eixo comparar com o label no no da arvore
        axis=0
        if query[0]<node.label:#node.label > query[0]:

          good_side =node.left
          bad_side =node.right
        else:

          good_side = node.right
          bad_side = node.left

     else:
        axis=1
        if query[1]<node.label: #node.label > query[1]:

          good_side =node.left
```

Essa função é responsável em retornar os vizinhos mais próximos do node query em pesquisa.A função recebe como parâmetro o ponto corrente do conjunto de teste o node corrente da arvore KD, o best_node, a menor distância , o eixo de pesquisa axis, uma lista com as melhores distâncias junto com uma outra lista com os vizinhos mais próximos. A função de pesquisa dos vizinhos se dar da seguinte forma; primeiro verificamos o label do node corrente , se o label é zero quer dizer que chegamos em um node folha e podemos calcular a distância dessa folha ao ponto de pesquisa do conjunto de teste, caso contrário não chegamos a uma folha e temos que continuar a pesquisa seguindo os cortes no plano apontado pelo label do node corrente.Verificamos a paridade da profundidade do node se for par atribuímos ao eixo axis 0, referente ao eixo vertical, caso contrário 1 para o eixo vertical.Apos verificar o corte correto no plano atribuimos dois lados de pesquisa o good_side e bad_side , recorrentes a região correta de pesquisa no plano. Ao final chamamos a função recursivamente para a região de pesquisa que estamos. Também temos que verificar se vale a pena pesquisar o lado ruim do plano, isso porque pode haver a possibilidade de existir um ponto com distância menor ao raio estipulado pelo eixo horizontal e vertical , para isso verificamos se menor distância atual de um ponto corrente é menor que a distância até um dos eixos e se for o caso fazemos a pesquisa chamando a função recursivamente para o bad_side.Ao fim retornamos os x pontos mais próximos ao ponto de pesquisa de teste.

Responsável por retornar as métricas de avaliação e chamar a função de pesquisa para cada ponto do conjunto de teste.

```
def get_Neighbors(self,trainingSet, testSet, x): #Como não foi expecificado no trabalho o quantidade de vizinhos mais prox , vamos considerar x=3
  best_neighbors=[]
  neighbors=[]
  best_distances=[]
  best_distancel=[]
  predictions=[]
  lista_aux=[]
  tree=arvoreKD()
  node=tree.build_KDTree(trainingSet,0)
  for i in range(len(testSet)):
    best_neighbors = self.find_nearest_neighbor(testSet[i],node,None,np.infty,None,best_distances,best_distancel)
    best_neighbors.sort(key=itemgetter(1))
    for i in range(x): #seleciona os x vizinhos mais prox
      neighbors.append(best_neighbors[i][0])
```

Recebe como parâmetro o conjunto de treino e de teste. Constrói a arvore KD com o conjunto de treinamento e para cada ponto do conjunto de teste chama a função de pesquisa para esse ponto e tbm a função que retorna as métricas de avaliação

Retorna as métricas de validação do método

```
def get_metrics(self, testSet, predictions):
    metrics = [0.0, 0.0, 0.0, 0.0]
    metrica=[]
    tp = 0
    fp = 0
    tn = 0
    fn = 0
    last_column = len(testSet[0]) - 1
    for i in range(len(testSet)):
        if testSet[i][last_column] == predictions[i][last_column] == 1:
          tp += 1
        elif testSet[i][last_column] == 0 and predictions[i][last_column] == 1:
          fp += 1
        elif testSet[i][last_column] == 1 and predictions[i][last_column] == 0:
          fn += 1
        elif testSet[i][last_column] == 0 and predictions[i][last_column] == 0:
          tn += 1

    metrics[0] = (tp + tn) / (tp + tn + fp + fn)  # accuracy

    if tp+fp != 0:
      metrics[1] = tp / (tp + fp)  # precision
    if tp+fn != 0:
      metrics[2] = tp / (tp + fn)  # recall

    return metrics
```

Funcionamento:
O sistema de classificação proposto se baseia na chamada de classificação de pontos no plano. Dessa forma , o sistema primeiro recebe como parâmetro a base de dados de treino e de teste na chamada da função get_Neighbors da classe x_NN. Dentro da função construimos a arvore KD com o conjunto de dados de treinamento, chamando a função

build_KDTree. Após construída a árvore, passamos o node root para a função de pesquisa. Pegamos o conjunto de dados de teste , e a cada ponto desse conjunto buscamos os pontos mais próximos ao mesmo.A função find_nearest_neighbor retorna os x vizinhos mais próximos ao ponto em análise, que no caso para simplificar pegamos os 3 vizinhos mais próximos. Ao final chamamos a função das métricas de avaliação.

## 5. Experimentos e Testes

Dados de teste

```
def main():

    #Aqui fazemos os testes .Fazemos 10 testes com bases aleatorias divindo em conjunto de treino e teste
    for i in range(10):
      x1, y1=get_data2()
      x_treino, x_teste, y_treino,y_teste  = train_test_split(x1, y1, test_size = 0.25,random_state = 1)
      fit=x_NN()
      fit.get_Neighbors(x_treino,y_teste,3)



if __name__ == "__main__":
    main()
```

Geração dos Dados

```
import random
def get_data2():
  trainSet = []
  testSet = []
  for i in range(0,1000):
    p1 = random.randint(1,100)
    p2 = random.randint(1,100)
    trainSet.append((p1,p2))
  for i in range(0,1000):
    p1 = random.randint(1,100)
    p2 = random.randint(1,100)
    testSet.append((p1,p2))
  return trainSet, testSet
```

OBS: O site informado na especificação do tp para obtenção das bases estava fora do ar.

Experimentos: Teste com geração de base com 1000 entradas.

```
Ponto A
(34, 53)
Os x vizinhos mais proximos
[[(93, 65)], [(10, 37)], [(93, 68)]]
Metricas da classe do ponto A
Ponto A
(67, 72)
Os x vizinhos mais proximos
[[(93, 65)], [(10, 37)], [(93, 68)]]
Metricas da classe do ponto A
Ponto A
(4, 60)
Os x vizinhos mais proximos
[[(4, 60)], [(93, 65)], [(10, 37)]]
Metricas da classe do ponto A
Ponto A
(59, 71)
Os x vizinhos mais proximos
[[(4, 60)], [(93, 65)], [(10, 37)]]
Metricas da classe do ponto A
Ponto A
(33, 45)
Os x vizinhos mais proximos
[[(4, 60)], [(93, 65)], [(10, 37)]]
Metricas da classe do ponto A
Ponto A
(80, 21)
Os x vizinhos mais proximos
[[(4, 60)], [(93, 65)], [(10, 37)]]
Metricas da classe do ponto A
Ponto A
(59, 41)
Os x vizinhos mais proximos
[[(4, 60)], [(93, 65)], [(10, 37)]]
Metricas da classe do ponto A
Ponto A
(61, 27)
```