Algoritmos Baseados em Instâncias

Paulo Trigo Silva

Aspectos do suporte (base) nas instâncias

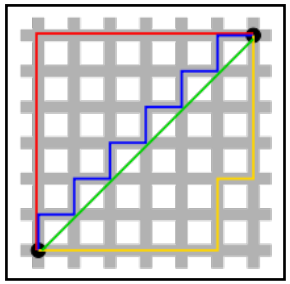
- Os exemplos de treino são "memorizados" tal como estão
  - não se constrói um modelo à-priori do conjunto de treino
  - ... "lasy" learning", i.e., só classifica quando surge uma nova instância!
- Recorre-se à noção de "função de distância"
  - usada para determinar a distância entre duas quaisquer instâncias
- Para classificar uma nova instância (e.g., y)
  - escolhe-se a classe do(s) exemplo(s) de treino mais próximo(s) (de y)
- ... algumas possíveis "função de distância"
  - Euclidiana (só domínios numéricos)
  - Manhattan (só domínios numéricos)
  - Hamming (domínios numéricos e nominais)
  - Quadrada (mesmo que Euclidiana mas sem cálculo da raiz quadrada)
  - ...

Paulo Trigo Silva

... as funções de distância

Euclidiana:
verde

Manhattan:
vermelha, amarela azul



- Sejam duas instâncias A e B com atributos
  - respectivamente $a_1, a_2, ..., a_n$ e $b_1, b_2, ..., b_n$
- Distância **Euclidiana**: $(\Sigma_{i=1..n} (a_i - b_i)^2)^{½}$
  - i.e,, distância "em linha recta entre dois pontos"
  - para comparar distâncias não é calcular a raiz quadrada
  - ... variações incluem potências maiores que 2; aumentar potência aumenta influência das maiores diferenças (à custa das menores)
- Distância de **Manhattan**: $\Sigma_{i=1..n} | a_i - b_i |$
  - i.e., distância "em blocos entre dois pontos"
- Distância de **Hamming**: $\Sigma_{i=1..n} h(a_i, b_i)$, onde $h(x, y) \begin{cases} 0, \text{ se } x = y \\ 1, \text{ se } x \neq y \end{cases}$
  - i.e., número de posições em que A e B diferem entre si

Implemente, e.g., em Python, estas três funções de distância

Paulo Trigo Silva

As funções de distância (uma implementação em Python)

```
def distance_euclidean( exampleA, exampleB ):
   import math
   assert len( exampleA ) == len( exampleB )
   return math.sqrt( sum( ( itemA - itemB )**2 \
      for ( itemA, itemB ) in zip( exampleA, exampleB ) ) )

def distance_manhattan( exampleA, exampleB ):
   assert len( exampleA ) == len( exampleB )
   return sum( abs( itemA - itemB ) \
      for ( itemA, itemB ) in zip( exampleA, exampleB ) )

def distance_hamming( exampleA, exampleB ):
   assert len( exampleA ) == len( exampleB )
   return sum( itemA != itemB \
      for ( itemA, itemB ) in zip( exampleA, exampleB ) )
```

`zip( arg1, arg2, ...)` returns a list of tuples, where the i-th tuple contains the i-th element from each argument

Exemplo:

```
zip( [11, "aaa", 33, 5], [33, "xx", 33, 5] )
> [(11, 33), ('aaa', 'xx'), (33, 33), (5, 5)]
```

```
distance_hamming( \
   [11, "aaa", 33, 5], \
   [33, "xx", 33, 5] )
> 2
```

Paulo Trigo Silva

Normalizar os valores

- Os valores dos domínios dos vários atributos têm diferentes escalas
  - pelo que ao usar uma função de distância é usual normalizar
  - ... para que o cálculo de distância não seja influenciado pela escala
- ... é usual normalizar os valores no intervalo [0 .. 1] calculando
  - calculando $v_i = (a_i – \min a_i) / (\max a_i – \min a_i)$
  - onde a_i é o valor real do atributo i e min e max são, respectivamente, o valor mínimo e máximo do atributo i no conjunto de treino
- ... a formulação assume que os atributos têm domínios numéricos
  - primeiro normalizam-se os valores de cada atributo dos exemplos
  - e depois pode calcular-se a distância entre quaisquer dois exemplos
- No caso de atributos com domínio nominal não existe normalização
  - considera-se a distância de Hamming (distância 1 se valores diferentes)
  - ... pelo que só se usam os valores 1 e 0 (logo não é preciso normalizar)

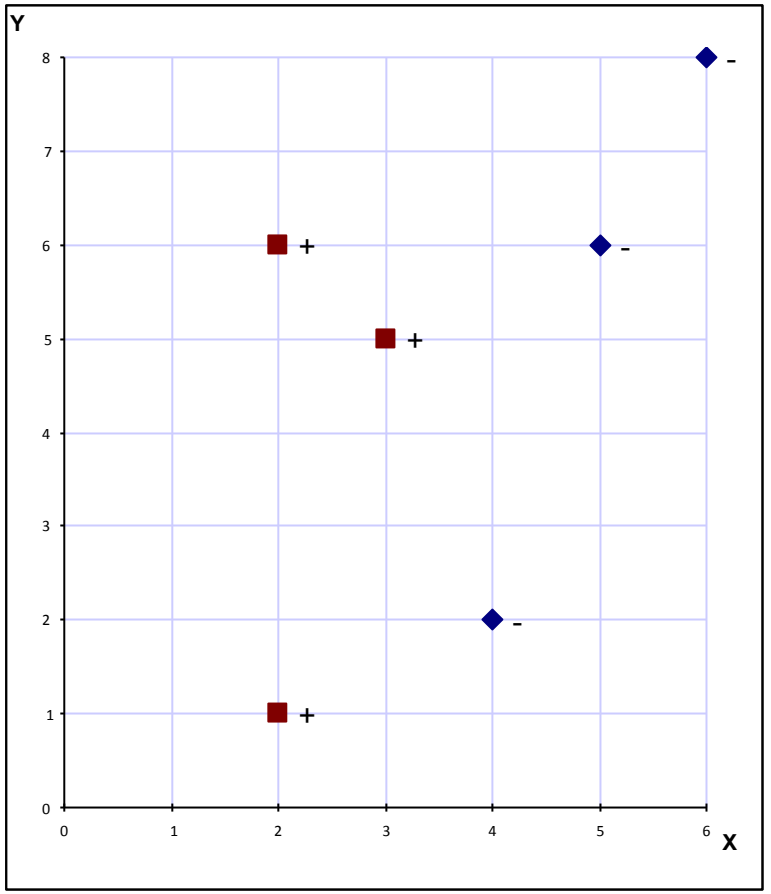
Lidar com a distância envolvendo valores omissos

- A abordagem segue a ideia de se "assumir a diferença máxima"
  - entre um valor omisso e qualquer outro valor
  - i.e., a ausência de informação origina distância máxima entre atributos
- Ao comparar 2 valores de atributos nominais temos,
  - se pelo menos um é omisso então a distância é 1 (normalizado)
  - ... notar que é também 1 quando ambos são omissos
  - só quando ambos são iguais e não omissos a distância será 0 (zero)
- Ao comparar 2 valores, a_i e b_i, de atributos numéricos temos,
  - se ambos forem omissos então a distância é 1
  - se só 1 é omisso, e.g., a_i é omisso, então a distância é **max**(b_i, 1 – b_i)
  - ... ou seja, se só 1 é omisso a distância é tão grande quanto possível

Um exemplo (abstracto)

Conjunto de treino

Instância	X	Y	Conceito
1	6	8	-
2	2	6	+
3	5	6	-
4	3	5	+
5	4	2	-
6	2	1	+



No conjunto de treino qual a instância mais próxima de a_1=4, a_2=4?

Considere a distância de Manhattan.

... o exemplo abstracto (detalhe dos cálculos)

Atenção: normalizar também, com os mínimos e máximos do conjunto de treino, o exemplo a classificar (se não normalizar estará sempre mais perto dos exemplos na "fronteira" do conjunto de treino)

a_1	a_2
4	4

$a_{inorm}=(a_i-min)/(max-min)$

0,50	0,43

$(i-min)/(max-min)$

Instância	X	Y	Conceito	X_{norm}	Y_{norm}	$\lvert a_{1norm}-X_{morm}\rvert$	$\lvert a_{2norm}-X_{morm}\rvert$	Σ	ordem	Conceito
1	6	8	-	1,00	1,00	0,50	0,57	1,07	6º	-
2	2	6	+	0,00	0,71	0,50	0,29	0,79	4º	+
3	5	6	-	0,75	0,71	0,25	0,29	0,54	3º	-
4	3	5	+	0,25	0,57	0,25	0,14	0,39	2º	+
5	4	2	-	0,50	0,14	0,00	0,29	0,29	1º	-
6	2	1	+	0,00	0,00	0,50	0,43	0,93	5º	+

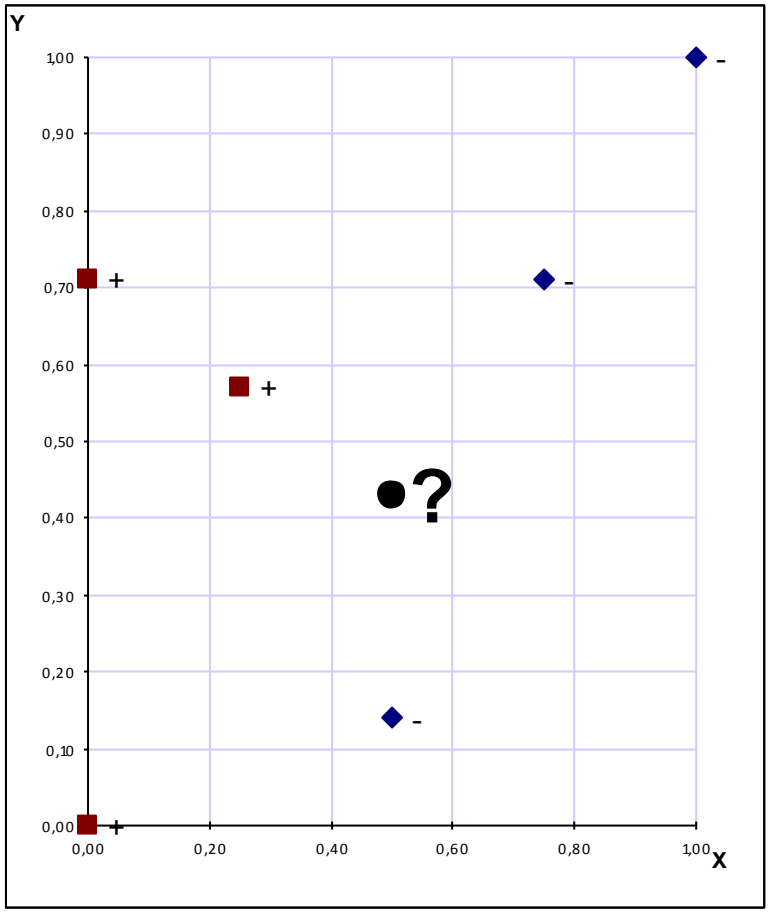
max	6	8
min	2	1

No conjunto de treino a **instância 5** é a **mais próxima de a_1=4, a_2=4**

Então, a que classe pertencerá o exemplo a_1=4, a_2=4?

Classificação do exemplo "olhando para os vizinhos"

Então, a que classe pertencerá o exemplo a_1=4, a_2=4?



ordem	Conceito
6º	-
4º	+
3º	-
2º	+
1º	-
5º	+

Classificar pela "regra da maioria" usando os "K vizinhos mais próximos"

com K=1 é " –"
com K=2 são 1" –" e 1"+" logo aleatório
com K=3 são 2" –" e 1"+" logo "–"
com K=4 são 2" –" e 2"+" logo aleatório
com K=5 são 2" –" e 3"+" logo "+"
...

Paulo Trigo Silva

Como procurar vizinhos (próximos) de modo eficiente?

- Para procurar os vizinhos mais próximos de uma qualquer instância
  - podemos calcular a distância a cada um dos exemplos
  - ... e escolher a menor dessas distâncias
- Calcular distância a todos os exemplos tem complexidade temporal
  - linear no número de exemplos
  - ... i.e., tempo aumenta proporcionalmente com o número de exemplos
- Para **acelerar a procura** pode
  - organizar-se o conjunto de treino numa estrutura do tipo árvore
- ... a **KD-Treee** é uma árvore binária adequada para representar
  - pontos num espaço de dimensão K (daí o KD – "*K Dimensional" space*)
  - ... i.e., K é o número de atributos de cada ponto

Paulo Trigo Silva

KD-Tree – características da estrutura em árvore

- Uma KD-Tree (árvore KD) é uma árvore binária
  - i.e., cada nó tem no máximo dois descendentes
  - ... sub-árvore da *esquerda* e sub-árvore da *direita*
- Cada nó representa um exemplo e uma dimensão (ou eixo, direcção)
  - *esquerda*: exemplos com valores *menor ou igual* nessa dimensão
  - *direita*: exemplos com valores *maiores* nessa dimensão
  - ... organização idêntica à da árvore de pesquisa binária
- Para construir a árvore é preciso escolher, para cada nó,
  - o exemplo que esse nó irá representar, e
  - a dimensão pela qual se separam as sub-árvores (*esquerda* e *direita*)

... estratégias – escolher dimensão e exemplo em cada nó

- Para escolher a dimensão (ou eixo) – uma estratégia "pesada"
  - calcular a variância dos dados em cada eixo (dimensão) individualmente
  - e escolher o eixo (dimensão) com maior variância
- ... escolher dimensão (ou eixo) – uma estratégia "mais leve"
  - "rodar" nos atributos de acordo com a profundidade (*depth*) do nó
  - i.e., "resto da divisão inteira da profundidade pelo número de atributos"
- Para seleccionar o valor (e exemplo) considerar a mediana
  - ordenar os exemplo pela dimensão escolhida
  - escolher o ponto (mediana) que separa a metade inferior da superior
  - ... pode originar partições do tipo "rectângulo fino e comprido"
- ... ou calcular a média e seleccionar o exemplo mais próximo
  - implica mais cálculo (é "mais pesado") do que na mediana
  - ... mas pode originar partições mais "quadradas" do que as da mediana

Exemplo – construção da KD-Tree

Construir uma KD-Tree para a lista de exemplos:

`[ (6, 8), (2, 6), (5, 6), (3, 5), (4, 2), (2, 1) ]`

Para construir a KD-Tree utilizar as seguintes estratégias:

– "rodar" a dimensão de acordo com a profundidade da árvore

– seleccionar o exemplo que corresponde à mediana

Paulo Trigo Silva

... exemplo – a árvore KD

Construir uma KD-Tree para a lista de exemplos:

[(6, 8), (2, 6), (5, 6), (3, 5), (4, 2), (2, 1)]

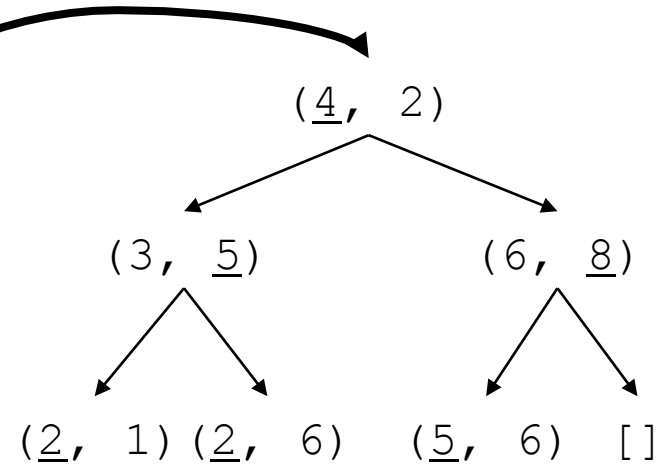
d = depth mod número-atributos = 0 mod 2 = 0

[(2, 6), (2, 1), (3, 5), (4, 2), (5, 6), (6, 8)]

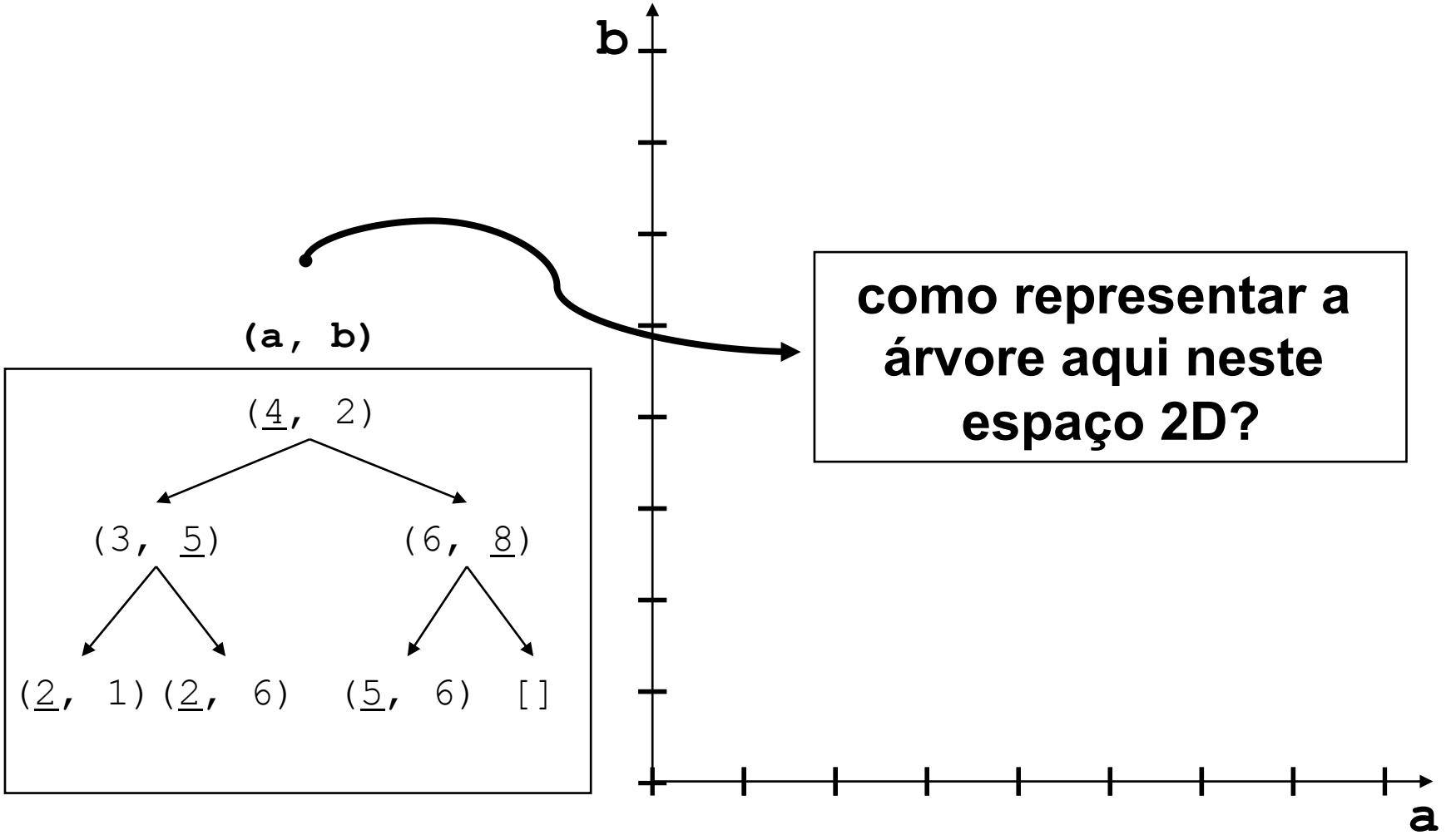
ordenar de acordo com dimensão, d, e escolher a mediana

```
(4, 2) d=0
  /(3, 5) d=1
    /(2, 1) d=0
    \(2, 6) d=0
  \(6, 8) d=1
    /(5, 6) d=0
    \[]
```

outra representação gráfica

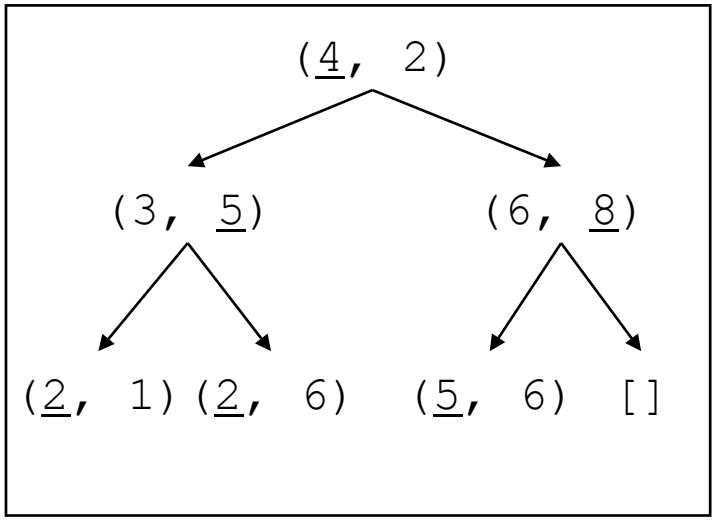


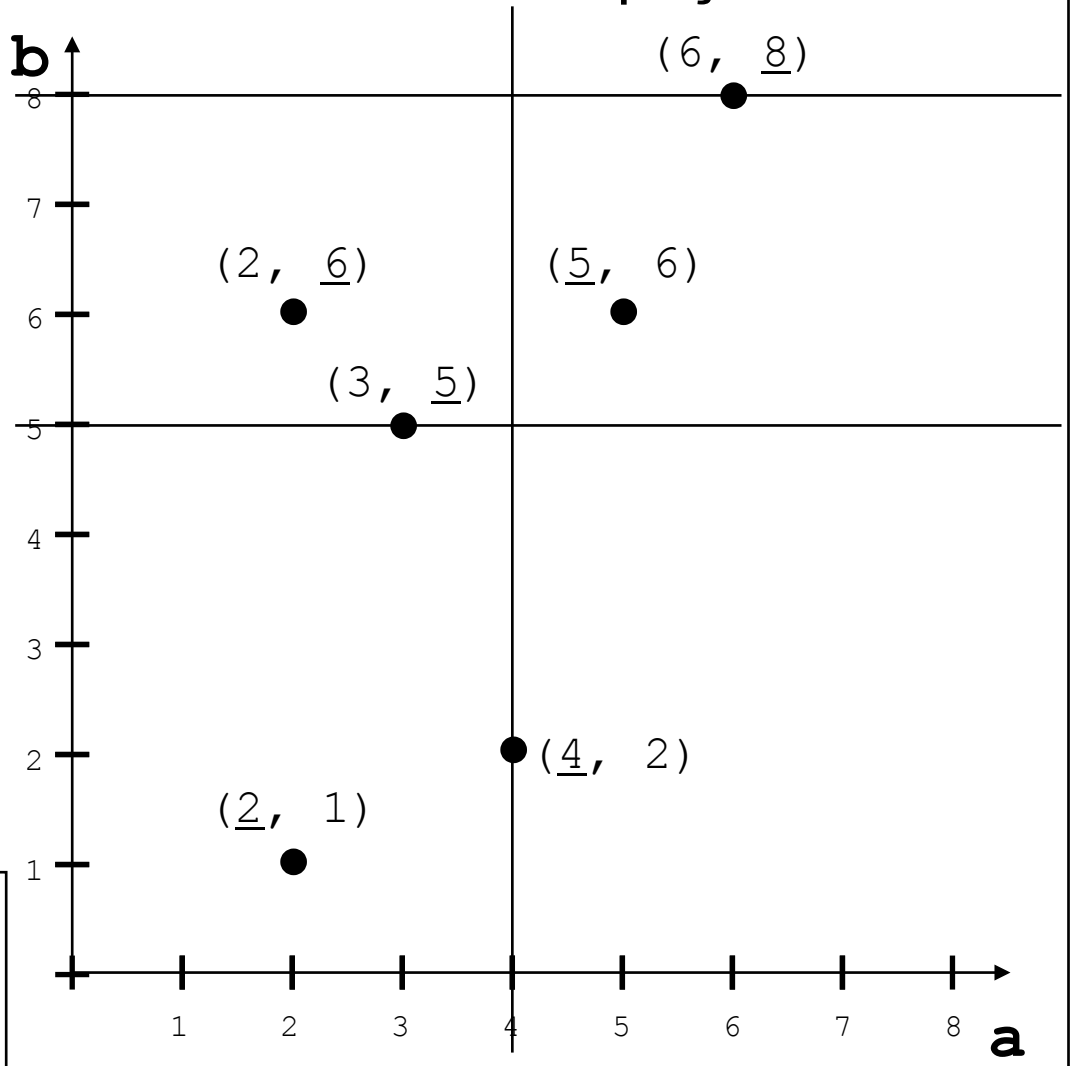
Exemplo – representar a árvore KD no espaço 2D



Paulo Trigo Silva

... exemplo – representar a árvore KD no espaço 2D





Notar que os planos não delimitam regiões de decisão;

nas decisões consideram-se as distâncias mais próximas.

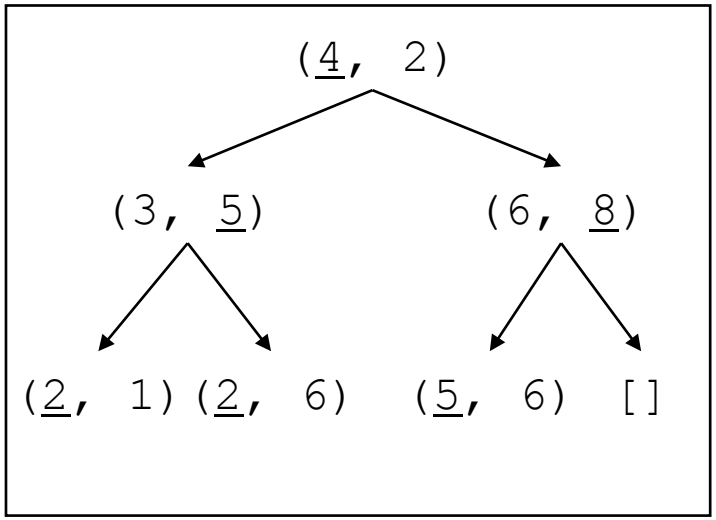
Paulo Trigo Silva

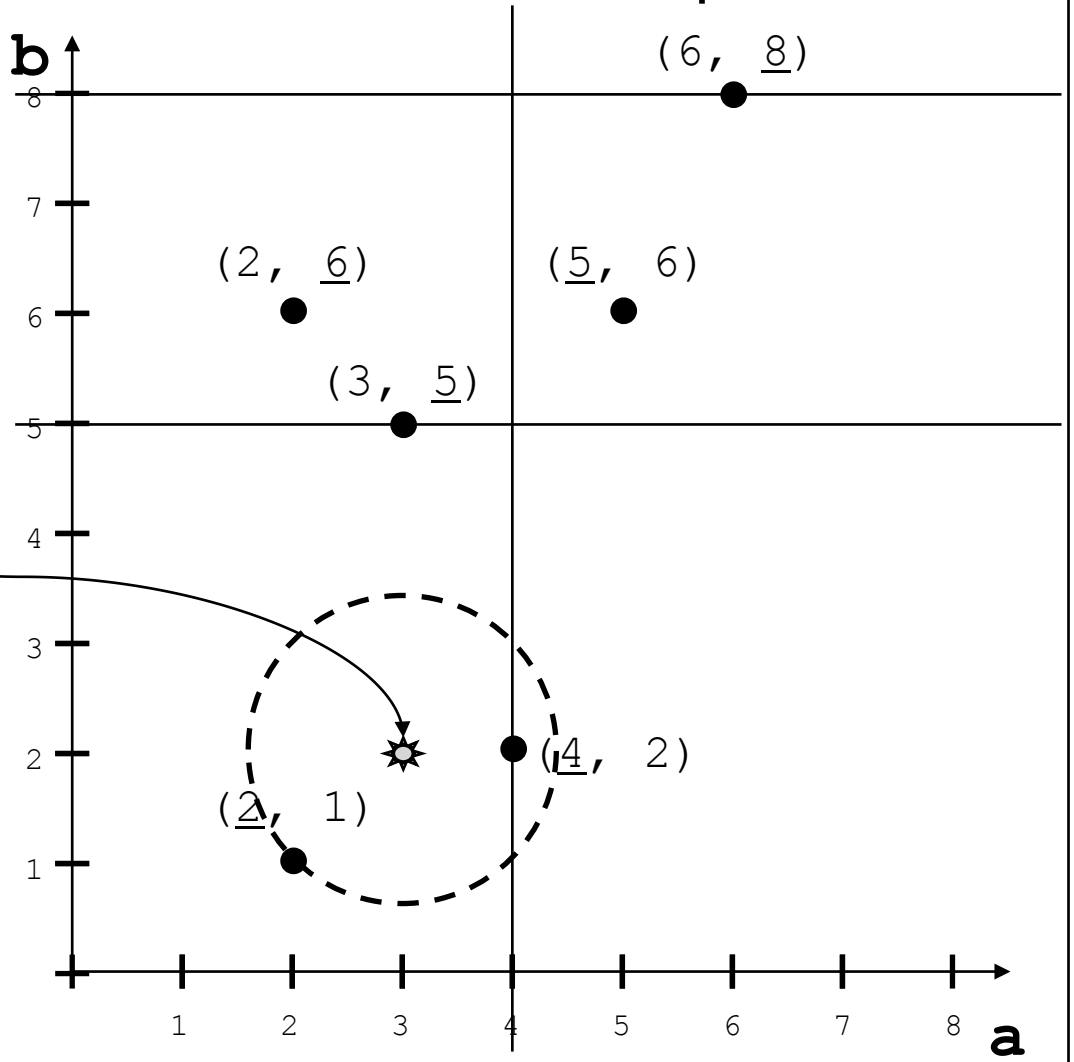
Usar a KD-Tree para encontrar o vizinho mais próximo

Qual o vizinho mais próximo de:

(3, 2)

?





E como usar a KD-Tree para procurar vizinhos próximos?

- Dado um novo exemplo percorrer a KD-Tree até uma folha
  - a folha não é necessariamente o vizinho mais próximo
  - mas é uma boa primeira aproximação
  - ... um melhor vizinho tem que estar a menor distância do que essa folha
  - ... no exemplo anterior é "estar no interior do círculo tracejado"
- Para ver se existe um melhor vizinho
  - verificar se o nó irmão tem um melhor vizinho
  - retroceder ao nó pai e verificar se é um melhor vizinho
  - verificar se o irmão do nó pai é melhor vizinho
  - se num novo caminho um nó é melhor vizinho ver as suas sub-árvores
  - ... neste percurso ir sempre guardando o melhor nó até ao momento

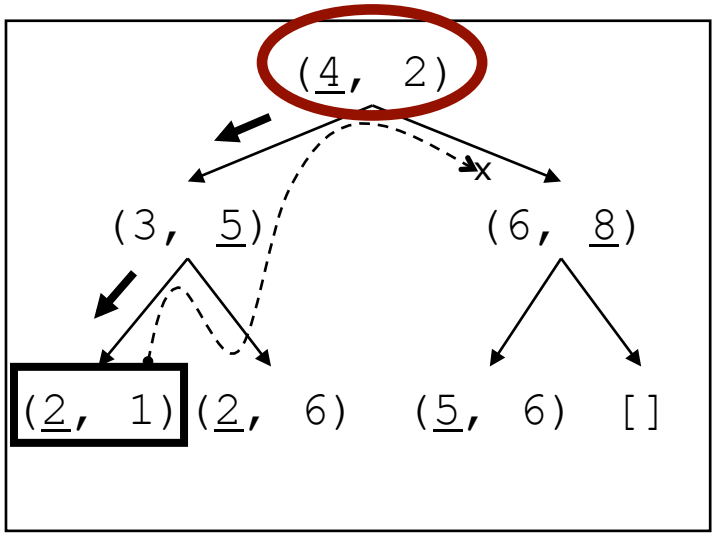
Paulo Trigo Silva

... procurar vizinhos

Qual o vizinho mais próximo de:

(3, 2)

?



O vizinho mais próximo é:

(4, 2)

(que é avô de (2, 1))

Primeira aproximação: (2, 1)

distância = $(3-2)^2 + (2-1)^2 =$ **2**

Há outro(s) mais próximo?

• análise do **pai**, (3, 5), e **descendentes**:

"valor na 2ª dimensão tem que ser superior a 5 logo distância a (3, 2) necessariamente superior a $(5-2)^2 = 9 >$ **2**; logo nem o pai nem nenhum dos seus outros descendentes pode ser mais próximo"

• análise do **avô**, (4, 2), e **descendentes**:

"valor na 1ª dimensão tem que ser superior a 4 logo distância a (3, 2) necessariamente superior a $(4-3)^2 = 1 <$ **2**; logo pode haver mais próximo"

• o **avô** tem *distância* = $(4-3)^2 + (2-2)^2 =$ **1**

"qualquer descendente (ramo à direita) do avô tem valor na 1ª dimensão superior a 4 logo distância a (3, 2) superior a $(4-3)^2 = 1$; logo não há mais próximo"

Paulo Trigo Silva

Síntese – classificação com o algoritmo kNN

- Assume-se um conjunto de treino com instâncias da forma < x, f(x) >
  - onde x são valores de atributos e f(x) é o valor da classe de x
  - ... se a função f é discreta, então f: $R^n \rightarrow V = \{v_1, ..., v_n\}$
  - ... se a função f é contínua, então f: $R^n \rightarrow R$
- Dada a instância, x_c, para classificar, pesquisar, no conjunto treino,
  - as K instâncias, x_1, x_2, ..., x_k, mais próximas de x_c
  - ... aqui pode usar-se KD-Tree para melhorar desempenho na pesquisa
- Para funções f discretas, após encontrar as K instâncias x_1, x_2, ..., x_k,
  - devolver $f(x_c) = \arg\max_{v \in V} \#\{x_i : f(x_i) = v\}$ (ou seja, "*regra de maioria*")
  - i.e., $f(x_c) = \arg\max_{v \in V} \Sigma_{i=1..K} \delta(f(x_i), v)$, com $\delta(a,b)=1$ se a=b, 0 c.c.
- Para funções f contínuas, após encontrar as K instâncias x_1, x_2, ..., x_k,
  - devolver $f(x_c) = (\Sigma_{i=1..K} f(x_i)) / K$
  - ou seja, classifica como a média dos K exemplos mais próximos de x_c

Paulo Trigo Silva

... refinamento – kNN com ponderação pelas distâncias

- Refinamento para pesar a contribuição de cada um dos K vizinhos
  - com base na sua distância ao ponto a classificar
- Considera-se, para cada vizinho, x_i, o peso, w_i, como sendo
  - uma função monótona decrescente com o aumento da distância
  - usando, por exemplo, a inversa da função de distância adoptada
  - i.e., $\mathbf{w_i} = 1 / \mathit{distancia}(x_c, x_i)$
- Para funções f discretas, após encontrar as K instâncias $x_1, x_2, ..., x_k$,
  - devolver $f(x_c) = \arg\max_{v \in V} \Sigma_{i=1..K} \mathbf{w_i} \times \delta(f(x_i), v)$
- Para funções f contínuas, após encontrar as K instâncias $x_1, x_2, ..., x_k$,
  - devolver $f(x_c) = (\Sigma_{i=1..K} \mathbf{w_i} \times f(x_i)) / (\Sigma_{i=1..K} \mathbf{w_i})$; i.e., média ponderada
- ... se vizinho coincidente com x_c ($\mathit{distancia}(x_c,x_i)=0$) fazer w_i máximo

Paulo Trigo Silva

Com que estrutura(s) implementar o kNN com KD-Tree?

- Uma estrutura que represente a noção de "nó de árvore"
  - com capacidade para armazenar dados
  - que tem uma lista de nós descendentes
  - e que sabe indicar se é, ou não, uma folha
- Um nó de uma KD-Tree especializa o anterior
  - pois adiciona a dimensão usada para separar o espaço
  - e apenas admite dois descendentes (KD-Tree é árvore binária)

Construir, e.g., em Python, aquelas duas estruturas.

Paulo Trigo Silva

... ”nó de árvore” – em Python

```
class TreeNode():
   def __init__( self, data, children ):
      self.__data = data
      # list with root node (TreeNode) of descending trees
      self.__children = children

   def is_leaf( self ):
      for item in self.children:
         if item <> None: return False
      return True

   @property
   def data( self ): return self.__data

   @data.setter
   def data( self, value ): self.__data = value

   @property
   def children( self ): return self.__children
```

Paulo Trigo Silva

... “nó de KD-Tree”

```
class KDTreeNode( TreeNode ):
   def __init__( self, point, dimension, left, right ):
      TreeNode.__init__( self, point, [ left, right ] )
      self.__dimension = dimension

   @property
   def dimension( self ): return self.__dimension

   @property
   def point( self ): return self.data

   @property
   def left( self ): return self.children[ 0 ]

   @property
   def right( self ): return self.children[ 1 ]
```

Paulo Trigo Silva

O que é necessário para construir a KD-Tree?

- Uma estratégia para escolher, em cada passo, uma dimensão
  - a mais simples é "rodar" de acordo com a profundidade do nó
  - a mais "pesada" é calcular a dimensão de maior variância
- Uma estratégia para escolher o ponto de separação do conjunto
  - a mais simples é escolher a mediana
  - a mais "pesada" é escolher o ponto mais próximo da média
- Um algoritmo que, de modo recorrente, construa a árvore KD
  - aplicando as estratégias em cada passo
  - e terminando quando o conjunto em análise é vazio

Construir, e.g., em Python, aquelas estratégias e sua aplicação na construção da KD-tree.

Admita que `point_list` contém uma lista de pontos e.g., [(6, 8), (2, 6), (5, 6)]

Paulo Trigo Silva

... estratégias para escolher dimensão e ponto separação

```
def choose_dimension_fromDepth( point_list, depth ):
   # choose dimension based on depth
   # (so that dimension cycles through all valid values)
   # assumes all points have the same dimension
   return depth % len( point_list[ 0 ] )
```

```
def choose_index_median( point_list ):
   # choose the index that corresponds to the median
   return len( point_list ) / 2
```

... e uma classe onde se vai registar essa informação e onde se vai implementar o método de construção da KD-Tree.

```
class KDTree():
   def __init__( self, data,
                 choose_dimension=choose_dimension_fromDepth,
                 choose_index=choose_index_median ):
      self.__choose_dimension = choose_dimension
      self.__choose_index = choose_index
      self.__root = self.__build( data, depth=0 )
```

... construção (método `build`) da KD-Tree

```
def __build( self, point_list, depth ):
   if not point_list: return None

   # choose the dimension to split the tree
   dimensionSplit = self.choose_dimension( point_list, depth )

   # "sort" modifies the list in-place (and returns None)
   # "key" is a function with a single argument;
   #the return is used for sorting
   point_list.sort( key = lambda point: point[ dimensionSplit ] )

   # choose the (data) index that splits the data in two subtrees
   indexSplit = self.choose_index( point_list )

   # create node and
   # recursively construct subtrees
   node = KDTreeNode(
      point = point_list[ indexSplit ],
      dimension = dimensionSplit,
      left = self.__build( point_list[ 0:indexSplit ], depth + 1 ),
      right = self.__build( point_list[ indexSplit+1: ], depth + 1 ) )
   return node
```

Como pesquisar na KD-Tree K vizinhos mais próximos?

- Manter uma estrutura que "memorize" os melhores vizinhos
  - essa estrutura vai sendo actualizada enquanto se analisa a árvore
- Um algoritmo que, de modo recorrente, pesquise a árvore KD
  - dado o exemplo a classificar,
  - primeiro avança até uma folha (aproximação do vizinho mais próximo)
  - depois retrocede a avalia a necessidade de analisar outros ramos

Construir, e.g., em Python, aquela estrutura e sua aplicação na pesquisa da KD-tree.

Paulo Trigo Silva

... estrutura que memoriza “melhores vizinhos”

```
class NearestNeighbours():
   def __init__( self, point_query, k ):
      self.__point_query = point_query
      self.__k = k
      self.__current_best = []

   def add_ifBetter( self, point, distance_function ):
      distance = distance_function( point, self.__point_query )
      # run through current_best, try to find appropriate place
      for i, e in enumerate( self.__current_best ):
         # all neighbours found, this one is farther, so return
         if i == self.__k: return
         if e[ 1 ] > distance:
            self.__current_best.insert( i, ( point, distance ) )
            return
      # otherwise, append point and distance to the end
      self.__current_best.append( ( point, distance ) )

   @property
   def largest_distance( self ):
      if self.__k >= len( self.__current_best ):
         return self.__current_best[ -1 ][ 1 ]
      return self.__current_best[ self.__k - 1 ][ 1 ]
   @property
   def best_k_neighbours( self ):
      return [ element[ 0 ] \
         for element in self.__current_best[ :self.__k ] ]
```

... a classe que implementa a pesquisa kNN

Esta KNN assume que existe uma KD-tree para pesquisar

```
class KNN():
   def __init__( self, kdTree, distance_function=distance_euclidean ):
      self.__kdTree = kdTree
      self.__distance_function = distance_function
      self.statistics = {}

   """
   point_query = the point to search for k neighbours
   k = the number of neighbours to search for
   """
   def query( self, point_query, k=1 ):
      if self.__kdTree.root == None: result = []
      neighbours = NearestNeighbours( point_query, k )
      self.__search( self.__kdTree.root, point_query, k, \
                     best_neighbours=neighbours )
      result = neighbours.best_k_neighbours
      return result
```

Paulo Trigo Silva

... de modo recorrente pesquisar a KD-Tree

```
def __search( self, node, point_query, k, best_neighbours ):
    if node == None: return
    if node.is_leaf():
        best_neighbours.add_ifBetter( node.point, self.distance_function )
        return
    dimensionSplit = node.dimension
    # decide which subtree is the "near" and which is the "far"
    if point_query[ dimensionSplit ] < node.point[ dimensionSplit ]:
        near_subtree = node.left; far_subtree = node.right
    else:
        near_subtree = node.right; far_subtree = node.left
    # until leaf found, recursively search through "near" subtree
    self.__search( near_subtree, point_query, k, best_neighbours )

    # while unwinding the recursion, check if better than current best
    best_neighbours.add_ifBetter( node.point, self.distance_function )
    # check for any points on the other side that can be better
    point_component = [ node.point[ dimensionSplit ] ]
    point_query_component = [ point_query[ dimensionSplit ] ]
    component_distance = self.distance_function( point_component, \
                                                 point_query_component )
    # the case where we also traverse the "far" subtree subtree
    if component_distance < best_neighbours.largest_distance:
        # until leaf found, recursively search through "far" subtree
        self.__search( far_subtree, point_query, k, best_neighbours )
    return
```

Exemplos de execução do algoritmo

```
d = [ (6, 8), (2, 6), (5, 6), (3, 5), (4, 2), (2, 1) ]
point = (3, 2)

tree = KDTree( d )
tree.pprint()

knn = KNN( tree, distance_euclidean )
nearest = knn.query( point, k=2 )
print nearest
```

```
(4, 2) d=0
  /(3, 5) d=1
    /(2, 1) d=0
    \(2, 6) d=0
  \(6, 8) d=1
    /(5, 6) d=0
    \[]
```

point = (3, 2)

[(4, 2), (2, 1)]

Exercício

Considerar o código em `b04_knn_kdtree.py` e alterar de modo a obter o seguinte resultado

```
d = [ (3, "d"), (4, "a"), (6, "g"), (2, "b"), \
      (8.5, "e"), (3, "h"), (7, "e"), (5, "h") ]

point = (3, "b")

tree = KDTree( d )
tree.pprint()

knn = KNN( tree, distance_euclidean )
nearest = knn.query( point, k=2 )
print nearest
```

```
(5, 'h') d=0
  /(3, 'd') d=1
    /(4, 'a') d=0
      /(2, 'b') d=1
      \[]
    \(3, 'h') d=0
  \(8.5, 'e') d=1
    /(7, 'e') d=0
    \(6, 'g') d=0
```

```
kNN para instância: (3, 'b') com k = 2
[(2, 'b'), (3, 'd')]
```