Aprendizado de máquina funcional

Pedro Faustini

UFABC

Paradigmas de Programação Q2.2019

- Estes slides foram preparados para o curso de Paradigmas de Programação na UFABC.
- Conteúdo baseado no material preparado pelo Professor Fabrício Olivetti de França da UFABC.

Roteiro

- Introdução
- Pré-processamento de dados
- Classificação
- Agrupamento
- Regressão

Introdução

- Extrair conhecimentos de bases de dados.
- Interdisciplinar: aprendizado de máquina, estatística, banco de dados...
- Independe de paradigma.
- Hoje veremos exemplos com linguagem funcional.

Classificação

- Supervisionado, ideia é atribuir classes a objetos desconhecidos a partir daquilo que se conhece de objetos conhecidos e rotulados.
- Exemplos de algoritmos:
  - K-NN
  - Naïve Bayes
  - Support Vector Machines

Agrupamento

- Não supervisionado, ideia é separar objetos de uma base em grupos, a partir dos atributos dos objetos.
- Exemplos de algoritmos:
  - K-Means (número de grupos definido por um parâmetro).
  - DBSCAN (número de grupos descoberto pelo algoritmo).

Regressão

- Semelhante a classificação, porém em vez de atribuir classes a objetos, atribui-se números.
- Por exemplo: precificar um imóvel a partir do preço de outros imóveis.

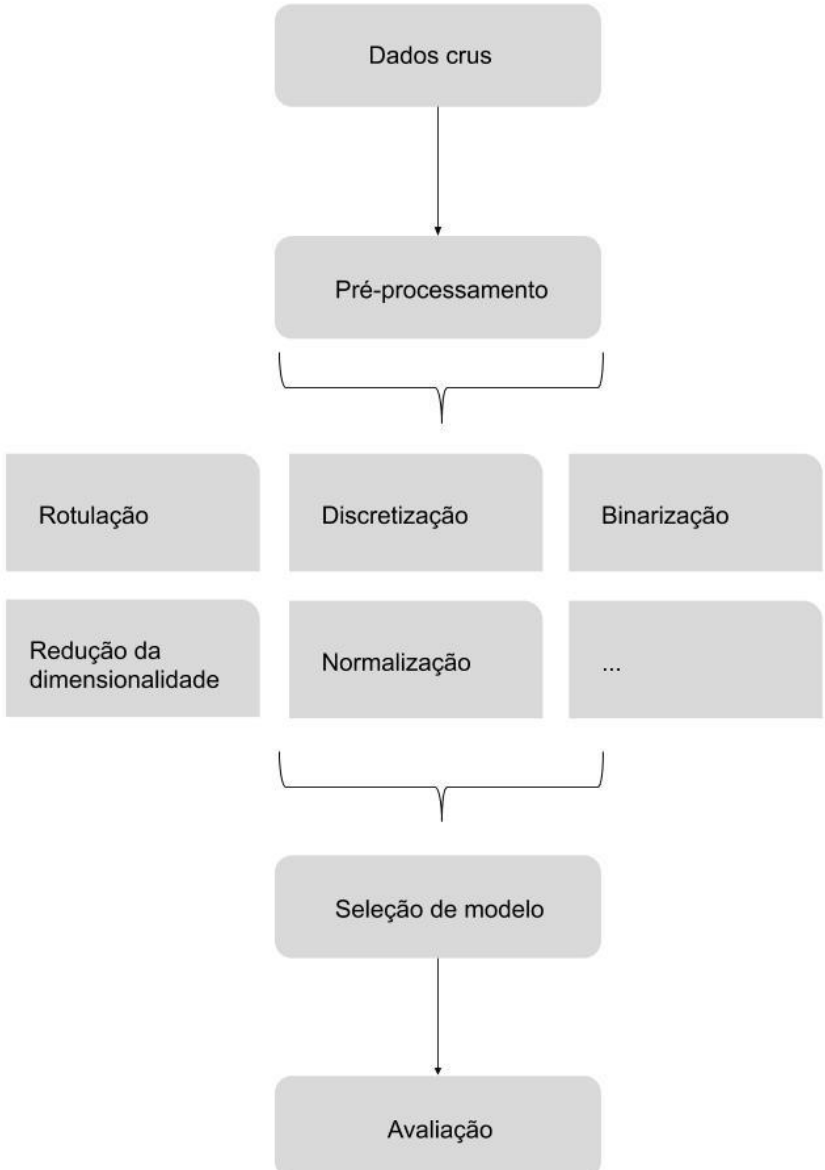
E o paradigma funcional (Haskell) nisso tudo?

- "Bibliotecas de aprendizado de máquina precisam ser velozes" [1]
- Linguagens mais populares para aprendizado de máquina são interpretadas (R e Python).
- Pontos específicos podem ser feitos em outras linguagens (e.g. LibSVM), mas ainda deixa a desejar.

[1]: HLearn: A Machine Learning Library for Haskell

E o paradigma funcional (Haskell) nisso tudo?

- Tarefas como cálculo de estatística, procedimentos de seleção, filtragem, junção de atributos podem não ser baratas.
- É comum em tarefas de mineração de dados buscar paralelismo para otimizar a computação.
- Imutabilidade do paradigma funcional evita o problema de race condition, que é o conflito de acesso.


Dados crus
Pré-processamento
Rotulação
Discretização
Binarização
Redução da dimensionalidade
Normalização
...
Seleção de modelo
Avaliação

Pré-processamento

- Conceitos básicos:
  - Base de dados é composta por objetos.
    - Funcionário, filme, máquina, etc.
  - Objetos são representados por atributos.
    - Categóricos
    - Numéricos

Pré-processamento

- Atributos categóricos:
  - Nominal: cargo de um funcionário, cor de um objeto.
  - Binário: 0, 1
  - Ordinal: {péssimo, ruim, médio, bom, ótimo}

- Atributos numéricos:
  - Intervalar: Dados que podem ser somados e subtraídos: temperatura em Celsius ou Fahrenheit.
  - Razão: Dados que também podem ser multiplicados e divididos, possuem um zero definido: temperatura em Kelvin, altura.

Atributos numéricos

- Suponha que esteja fazendo 40 C.
- Esta temperatura é o dobro de 20 C?

Atributos numéricos

- Suponha que esteja fazendo 40 C.
- Esta temperatura é o dobro de 20 C?
- Não. Temperatura em Celsius é um valor intervalar.
  - Não tem 0 definido.

Atributos numéricos

- Suponha que esteja fazendo 40 C.
- Esta temperatura é o dobro de 20 C?
- Não. Temperatura em Celsius é um valor intervalar.
  - Não tem 0 definido.
- 40 C = 104 F.
- Porém, 20 C = 60 F.

Atributos numéricos

- Uma pessoa com 2,3m de altura tem o dobro da altura de alguém com 1,15m?

Atributos numéricos

- Uma pessoa com 2,3m de altura tem o dobro da altura de alguém com 1,15m?
- Sim, pois altura é um valor racional, possui um 0 definido (ninguém tem altura negativa).

Pré-processamento

- Precisamos de uma representação numérica
  - Atributos numéricos: trivial.
  - Atributos ordinais: ranking.
  - Atributos nominais: vetor binário (OHE)

Pré-processamento

Profissão	Conceito	Salário
Engenheiro	A	10000
Professor	B	11000
Engenheiro	A	9000
Gerente	D	12000
Estudante	C	400

Pré-processamento

```haskell
type Point = ([Double], Double)

data Profissao = Engenheiro | Professor | Gerente | Estudante deriving (Eq, Show, Read, Enum, Bounded)

data Conceito = F | D | C | B | A deriving (Show, Read, Enum)

type Nota = Double

type Objeto = (Profissao, Conceito, Nota)

type Objeto' = [Double]
```

Pré-processamento

```
parseFile :: String -> [Objeto]
parseFile file = map parseLine (lines file)
    where
        parseLine l = toObj (words l)
        toObj [w1, w2, w3] = (read w1 :: Profissao, read w2 :: Conceito, read w3 :: Nota)
```

Pré-processamento

Profissão	Conceito	Salário
Engenheiro	A	10000
Professor	B	11000
Engenheiro	A	9000
Gerente	D	12000
Estudante	C	400

Pré-processamento: One-Hot Enconding

```haskell
binariza :: Profissao -> [Double]
binariza p = map bool2double [p == p' | p' <- profissoes]
    where
        profissoes = [minBound..] :: [Profissao]
        bool2double True = 1.0
        bool2double _ = 0.0
```

Profissão	Vetor OHE
Engenheiro	[1, 0, 0, 0]
Professor	[0, 1, 0, 0]
Gerente	[0, 0, 1, 0]
Estudante	[0, 0, 0, 1]

Pré-processamento: ranking

```
rank :: Conceito -> Double
rank co = (fromEnum' co) / (fromEnum' A)
    where
        fromEnum' = fromIntegral . fromEnum
```

Conceito	Valor
F	0
D	0,25
C	0,5
B	0,75
A	1

Pré-processamento

```
transformData :: [Objeto] -> [Objeto']
transformData data' = map parseObj data'
    where
        parseObj (prof, conc, nota) = (binariza prof) ++ [rank conc, nota]
```

Pré-processamento

Profissão	Conceito	Salário
Engenheiro	A	10000
Professor	B	11000
Engenheiro	A	9000
Gerente	D	12000
Estudante	C	400

Engenheiro	Professor	Gerente	Estudante	Conceito	Salário
1	0	0	0	1	10000
0	1	0	0	0.75	11000
1	0	0	0	1	9000
0	0	1	0	0.25	12000
0	0	0	1	0.5	400

Pré-processamento

```
transformDataPar :: ChunksOf [Objeto] -> ChunksOf [Objeto']
transformDataPar chunks = (map transformData chunks
                              `using` parList rdeepseq)
```

Pré-processamento: normalização

- Temos os dados em formato numérico.
- Tudo pronto para alimentá-los a algoritmos de classificação / agrupamento / regressão?
  - Depende, mas geralmente não.
  - Normalização dos dados.

Pré-processamento: normalização

- Exemplo: objetos têm dois atributos: idade e renda.
- Idade possui um intervalo de 0 a 122*
- Renda possui um intervalo de 0 a... ??
- Atributo renda vai dominar a distância entre os objetos, fazendo com que a idade seja irrelevante em muitos algoritmos (como KNN).

* Jeanne Calment (1875-1997)

Pré-processamento: normalização

- Três pessoas:

Pessoa	Idade	Renda
Amanda	22	1800
André	18	2500
Adriel	98	2000

Pré-processamento: normalização

- Três pessoas:

Pessoa	Idade	Renda
Amanda	22	1800
André	18	2500
Adriel	98	2000

	Amanda	André	Adriel
Amanda	0	700	213
André		0	506
Adriel			0

Pré-processamento: normalização

- Três pessoas:

Pessoa	Idade	Renda
Amanda	22	1800
André	18	2500
Adriel	98	2000

	Amanda	André	Adriel
Amanda	0	700	213
André		0	506
Adriel			0

Intuitivamente, parece correto afirmar que Amanda está tão mais mais distante André do que de Adriel?

Pré-processamento: normalização

```haskell
padroniza :: [[Double]] -> [[Double]]

padroniza x = mapColunas padroniza' x

padroniza' :: [Double] -> [Double]

padroniza' x = devMedia ./ sigma
  where
    media xs = (sum xs) / n
    devMedia = x .- (media x)
    sigma    = sqrt $ media $ devMedia .** 2
    n        = length' x
```

Pré-processamento: normalização

- Muito importante: normalizar os dados de **teste** usando a média e desvio encontrados nos dados de **treino**.
- Evitar a todo custo: normalizar o dataset inteiro uma vez, e depois separá-lo em pastas de treino e teste.
  - Dados de teste estarão "poluídos" com informações do conjunto de treino - data leakage.

Pré-processamento: normalização

```haskell
getMediaSigma :: [[Double]] -> [[Double]]
getMediaSigma trainData = mapColunas getMediaSigma' trainData

getMediaSigma' :: [Double] -> [Double]
getMediaSigma' x = [media, sigma]
    where
        media = (sum x) / n
        mediav xs = (sum xs) / n
        devMedia = x .- (mediav x)
        sigma = sqrt $ mediav $ devMedia .** 2
        n = length' x

padronizaTestData :: [[Double]] -> [[Double]] -> [[Double]]
padronizaTestData _ [] = []
padronizaTestData media_sigma testData = [padronizaTestData' media sigma (head testData')] ++ (padronizaTestData (tail media_sigma) (tail testData))
    where
        testData' = transpose testData
        media = head $ head media_sigma
        sigma = head $ tail $ head media_sigma

padronizaTestData' :: Double -> Double -> [Double] -> [Double]
padronizaTestData' media sigma x = devMedia ./ sigma
    where
        devMedia = x .- media
```

Pré-processamento: redução da dimensionalidade

- Quando nossos objetos possuem muitos atributos, podemos ter alguns problemas.
  - Tempo de execução (treino / teste).
  - Atributos redundantes.

Pré-processamento: redução da dimensionalidade

- Quando nossos objetos possuem muitos atributos, podemos ter alguns problemas.
  - Tempo de execução (treino / teste).
  - Atributos redundantes.
- Imagine um KNN calculando distância em uma base com **muitos** objetos, e cada objeto com **milhares** de atributos.

Pré-processamento: redução da dimensionalidade

- Quando nossos objetos possuem muitos atributos, podemos ter alguns problemas.
  - Tempo de execução (treino / teste).
  - Atributos redundantes.
- Imagine um KNN calculando distância em uma base com **muitos** objetos, e cada objeto com **milhares** de atributos.
- E se a métrica de distância for a distância do cosseno, que calcula raízes quadradas ainda por cima?

Pré-processamento: redução da dimensionalidade

- Quando nossos objetos possuem muitos atributos, podemos ter alguns problemas.
  - Tempo de execução (treino / teste).
  - Atributos redundantes.
- Imagine um KNN calculando distância em uma base com **muitos** objetos, e cada objeto com **milhares** de atributos.
- E se a métrica de distância for a distância do cosseno, que calcula raízes quadradas ainda por cima?
  - Vai demorar.

Pré-processamento: redução da dimensionalidade

- Existem várias técnicas de redução de dimensionalidade.
  - PCA (Principal Component Analysis)
  - LSA (Latent Semantic Analysis)
  - Johnson–Lindenstrauss lemma
  - DCDistance (Document-Class Distance)
- Ideia deles é: reduzir (preferencialmente bastante) o número de atributos.

DCDistance

DCDistance

- Autor: Charles Ferreira (UFABC).
- Algoritmo faz o número de atributos ser igual ao número de classes.
  - Geralmente, temos duas classes – ou pouco mais que isso – em nossos problemas.

DCDistance

- Soma-se os vetores de classes iguais.
- Os vetores resultantes são chamados vetores representativos de classe.
- Calcula-se a distância de cada vetor representativo de classe a cada vetor numérico original.
- Cada distância gera um novo atributo.

DCDistance

doc	Atributos	Rótulo
D1	1 0 0 1 0 0 0 1 0	C1
D2	1 1 1 0 0 0 0 1 1	C1
D3	1 1 0 1 0 0 0 1 1	C1
D4	0 0 0 0 1 0 1 1 1	C2
D5	0 0 1 0 1 1 1 1 1	C2

vetores representativos de classe

c1:	3 2 1 2 0 0 0 3 2
c2:	0 0 1 0 2 1 2 2 2

geração de atributos

doc	Atributos	Rótulo
D1	0.17, 0.73	C1
D2	0.12, 0.47	C1
D3	0.04, 0.58	C1
D4	0.55, 0.06	C2
D5	0.56, 0.04	C2

Adaptado de Ferreira, França e Medeiros (2018)

DCDistance

```
sumArrays :: [Double]->[Double]->[Double]
sumArrays a [] = a
sumArrays [] a = a
sumArrays (x:xs) (y:ys) = (x+y) : (sumArrays xs ys)

dcdistance :: [Point] -> [Point]
dcdistance points = [distances x | x <- points]
    where
        points' = groupBy ((==) `on` snd) $ sortBy (comparing snd) points :: [[Point]]
        vrc' = map (map (\x -> fst x)) points'
        vrc = map (foldr (\x -> sumArrays x) []) vrc'
        distances x = (d, label) :: Point
            where
                label = snd x
                f = fst x
                d = map (euclidean f ) vrc
```

DCDistance

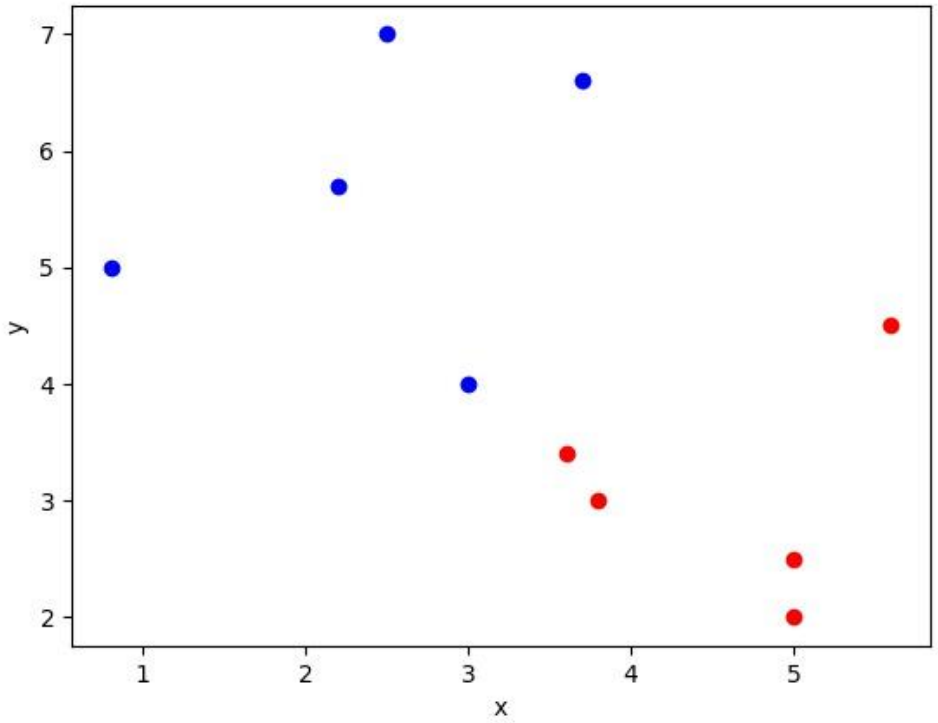
- Em experimentos, reduziu-se a dimensionalidade em mais de 99%, sem perdas (inclusive com ganhos) em relação a aplicação de algoritmos com a representação original dos dados.

Classificação

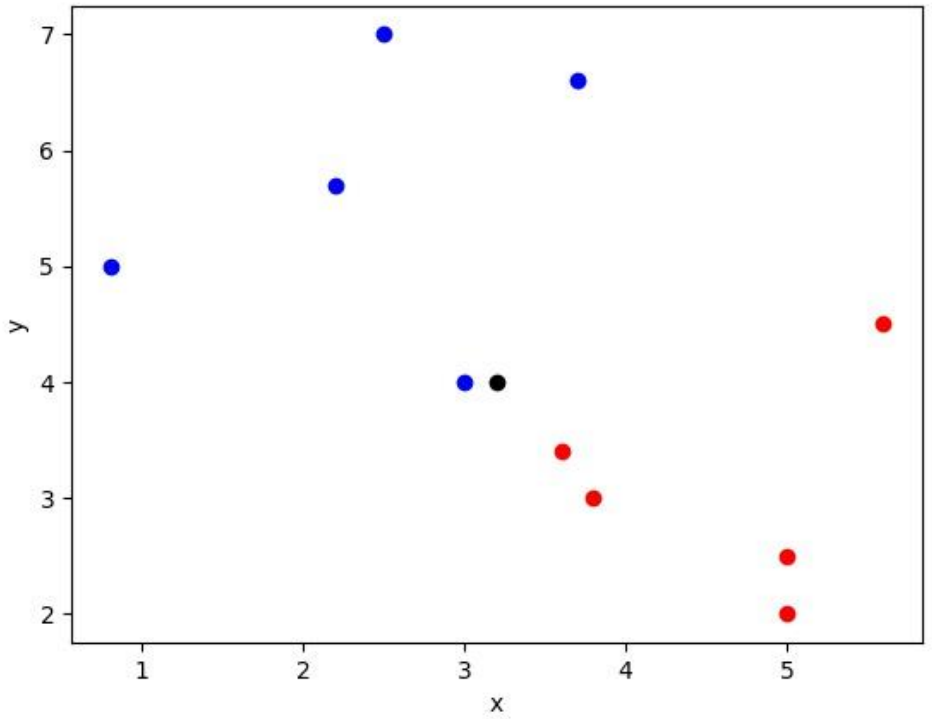
- Tendo os dados devidamente pré-processados, resta proceder com a classificação dos objetos não rotulados.

```
knn1 :: [Point] -> [Point] -> [Double]
knn1 train test = map (nearest) test
     where
     nearest t =     snd
                     $ head
                     $ sortByKey
                     $ map (distance t) train
     distance (t1,y1) (t2,y2) = (euclidean t1 t2, y2)
```

Classificação

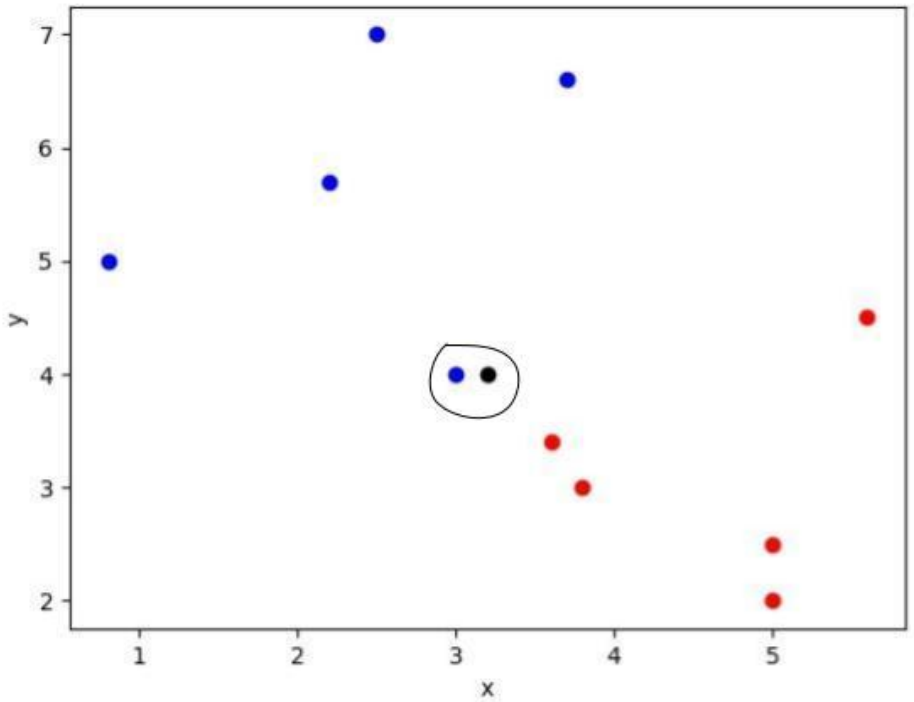


Classificação



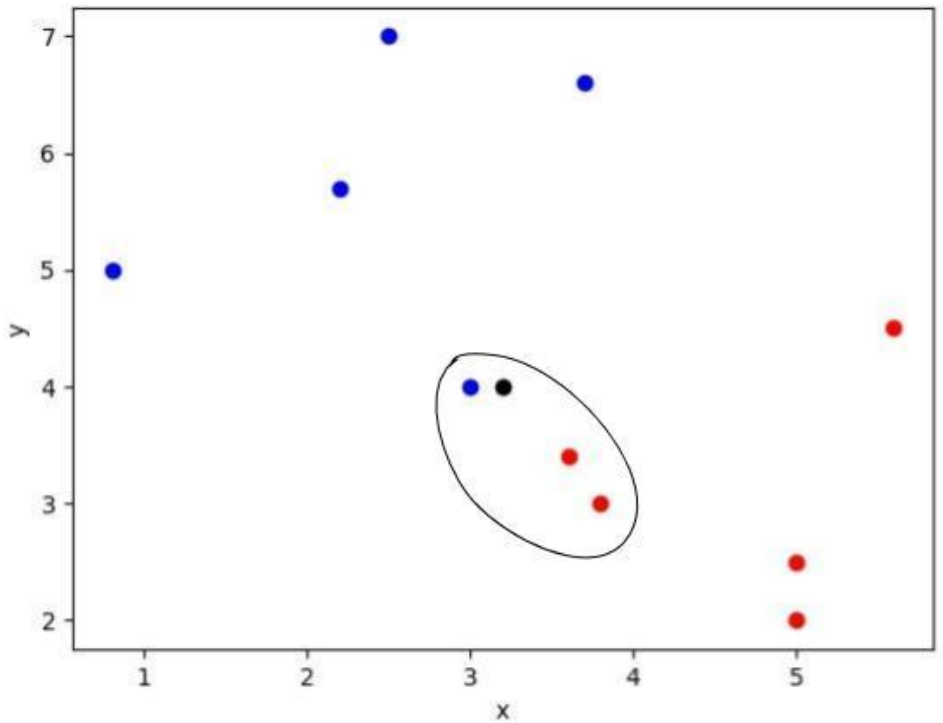
K = ?

Classificação



K = 1

Classificação



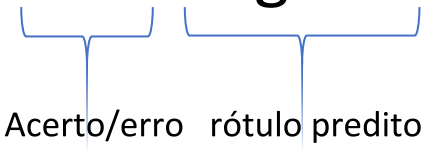
K = 3

Classificação: métricas de avaliação

- Temos um vetor de classes preditas e um vetor de classes de gabarito.
- O que fazemos com eles?
  - Ver o quanto acertamos e erramos!

Métricas de avaliação

- TP: True Positive
- FP: False Positive
- FN: False Negative
- TN: True Negative



	Rótulo do gabarito: A	Rótulo do gabarito: B
Rótulo predito: A	TP	FP
Rótulo predito: B	FN	TN

Métricas de avaliação

- Acurácia
- Precisão
- Revocação
- F-Score

Métricas de avaliação

- Acurácia: quanto acertei.
- Precisão: daqueles objetos que eu disse ser da classe X, quantos realmente eram da classe X?
- Revocação: de todos os objetos existentes na classe X, quantos eu identifiquei?
- F-Score: tradeoff entre precisão e revocação.

Métricas de avaliação

- Acurácia: $\frac{TP + TN}{TP + TN + FP + FN}$
- Precisão: $\frac{TP}{TP + FP}$
- Revocação: $\frac{TP}{TP + FN}$
- F-Score: $2\frac{p * r}{p + r}$

Métricas de avaliação: acurácia

- Acurácia: $\frac{TP + TN}{TP + TN + FP + FN}$

```haskell
acuraccy :: [Double] -> [Double] -> Double
acuraccy ytrue ypred = corrects / n_objects
    where
        n_objects = fromIntegral (length ypred) :: Double
        corrects = fromIntegral (length $ filter (\(t, p) -> t == p) (zip ytrue ypred)) :: Double
```

Métricas de avaliação: precisão

- Precisão: $\frac{TP}{TP + FP}$

```
precision :: Double -> [Double] -> [Double] -> Double
precision targetClass ytrue ypred = tp / (tp + fp)
    where
        correctP = fromIntegral (length $ filter (\(t, p) -> p == targetClass && t == targetClass) (zip ytrue ypred))  :: Double
        objectsP = fromIntegral (length $ filter (\t -> t == targetClass) ytrue)  :: Double
        tp = correctP / objectsP
        correctN = fromIntegral (length $ filter (\(t, p) -> p == targetClass && t /= targetClass) (zip ytrue ypred)) :: Double
        objectsN = fromIntegral (length $ filter (\t -> t /= targetClass) ytrue)  :: Double
        fp = correctN / objectsN
```

Métricas de avaliação: revocação

- Revocação: $\frac{TP}{TP + FN}$

```
recall :: Double -> [Double] -> [Double] -> Double
recall targetClass ytrue ypred = tp / (tp + fn)
    where
        correctP = fromIntegral (length $ filter (\(t, p) -> p == targetClass && t == targetClass) (zip ytrue ypred))  :: Double
        objectsP = fromIntegral (length $ filter (\t -> t == targetClass) ytrue)  :: Double
        tp = correctP / objectsP
        classifyN = fromIntegral (length $ filter (\(t, p) -> p /= targetClass && t == targetClass) (zip ytrue ypred)) :: Double
        fn = classifyN / objectsP
```

Métricas de avaliação

- Suponha que você tenha um algoritmo que classifica o estado de uma usina nuclear como "seguro" ou "vai explodir".

Métricas de avaliação

- Suponha que você tenha um algoritmo que classifica o estado de uma usina nuclear como "seguro" ou "vai explodir".
- Em 300 vezes, seu algoritmo acertou 298 casos. Possui ~99% de acurácia.

Métricas de avaliação

- Suponha que você tenha um algoritmo que classifica o estado de uma usina nuclear como "seguro" ou "vai explodir".
- Em 300 vezes, seu algoritmo acertou 298 casos. Possui ~99% de acurácia. As duas vezes que ele errou ele disse "seguro".

Métricas de avaliação

- Suponha que você tenha um algoritmo que classifica o estado de uma usina nuclear como "seguro" ou "vai explodir".
- Em 300 vezes, seu algoritmo acertou 298 casos. Possui ~99% de acurácia. As duas vezes que ele errou ele disse "seguro".
- O que é crítico neste caso é, quando a usina estiver para explodir, ele classificar o estado da usina como da classe "vai explodir".

Métricas de avaliação

- Suponha que você tenha um algoritmo que classifica o estado de uma usina nuclear como "seguro" ou "vai explodir".
- Em 300 vezes, seu algoritmo acertou 298 casos. Possui ~99% de acurácia. As duas vezes que ele errou ele disse "seguro".
- O que é crítico neste caso é, quando a usina estiver para explodir, ele classificar o estado da usina como da classe "vai explodir".
  - Revocação da classe "vai explodir".

Métricas de avaliação

- O juiz tem que proferir uma sentença (e.g. pena de morte) quanto a um suspeito. Há duas classes: inocente ou culpado.
- Revocação de 100%: sempre que um culpado é julgado, é condenado.

Métricas de avaliação

- O juiz tem que proferir uma sentença (e.g. pena de morte) quanto a um suspeito. Há duas classes: inocente ou culpado.
- Revocação de 100%: sempre que um culpado é julgado, é condenado.
- E os inocentes?
- Caso de precisão: quando modelo classifica alguém como culpado, difilmente erra.

Agrupamento

- Aprendizado não supervisionado.
- Não temos um conjunto rotulado para inferir o rótulo de objetos desconhecidos.
- Ideia é procurar por grupos na base de dados.
- Grupos podem ser entendidos como quão similares ou não determinados objetos são.

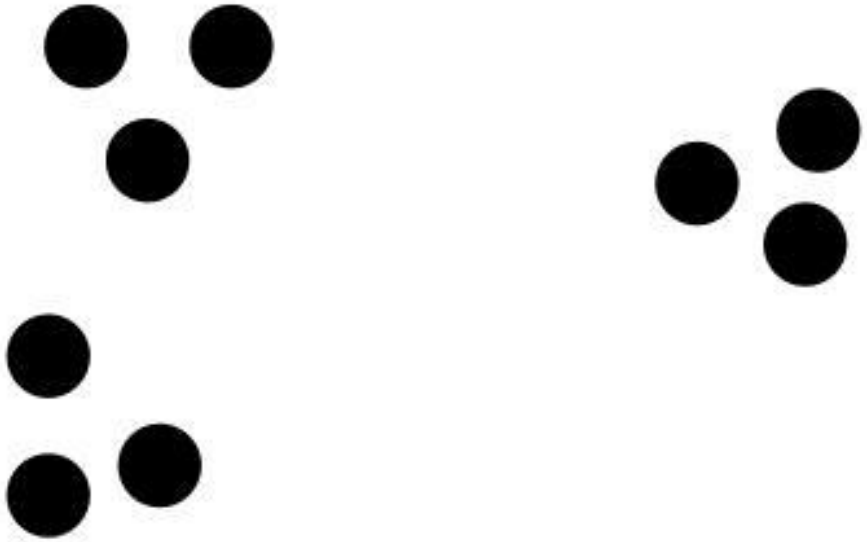
Agrupamento

- Similaridade não significa igualdade!

Keogh, E. A Gentle Introduction to Machine Learning and Data Mining for the Database Community, SBBD 2003, Manaus.

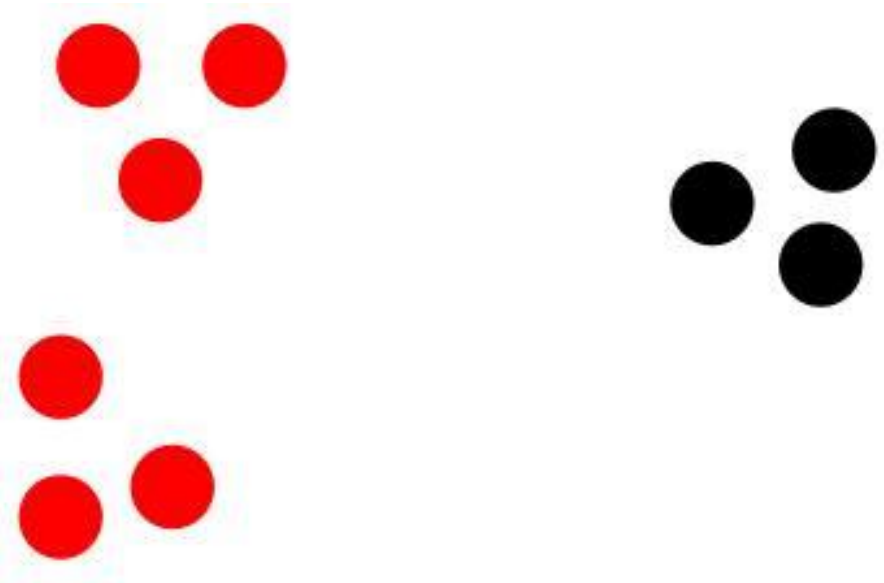
Agrupamento

- Mesmos dados podem levar a interpretações diferentes.



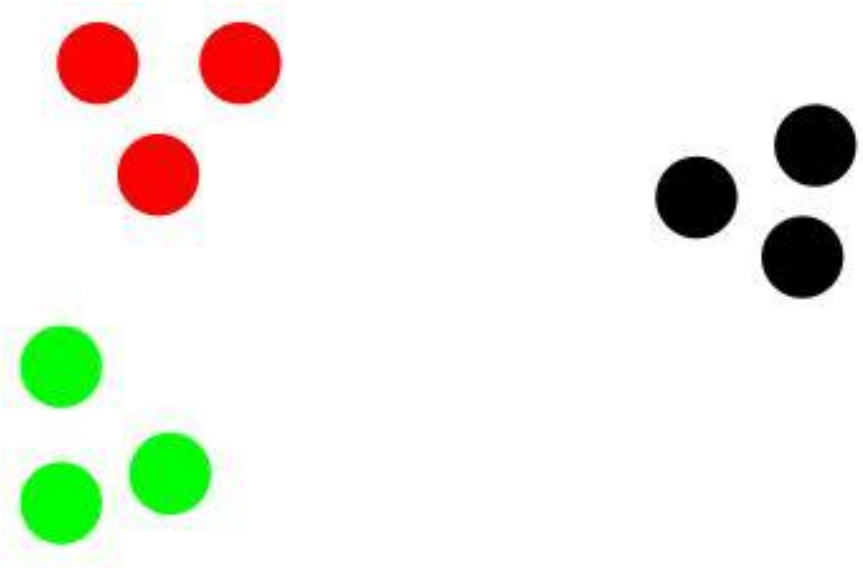
Agrupamento

- Mesmos dados podem levar a interpretações diferentes.



Agrupamento

- Mesmos dados podem levar a interpretações diferentes.



Agrupamento

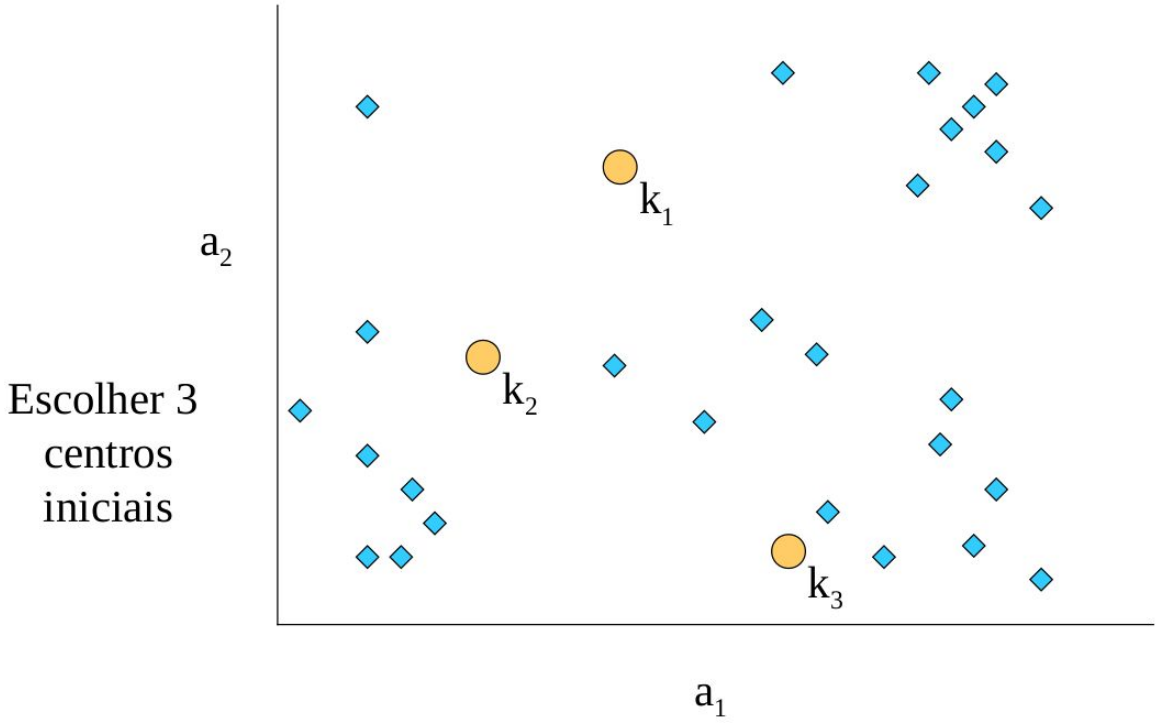
- Kmeans:
  - Agrupa os dados em K grupos, com base nos atributos dos objetos.
  - K é um parâmetro do algoritmo.

Agrupamento

- Kmeans:
  - Seleciona-se centroides iniciais.
  - Associa-se cada objeto ao centroide mais próximo, gerando K partições.
  - Para cada partição, calcular um novo centroide como a média dos pontos da partição.
  - Repetir processo até critério de parada.
    - Até centroides não mudarem.
    - Repetir N vezes.
  - Ponto crítico: qual valor de K escolher?

Agrupamento

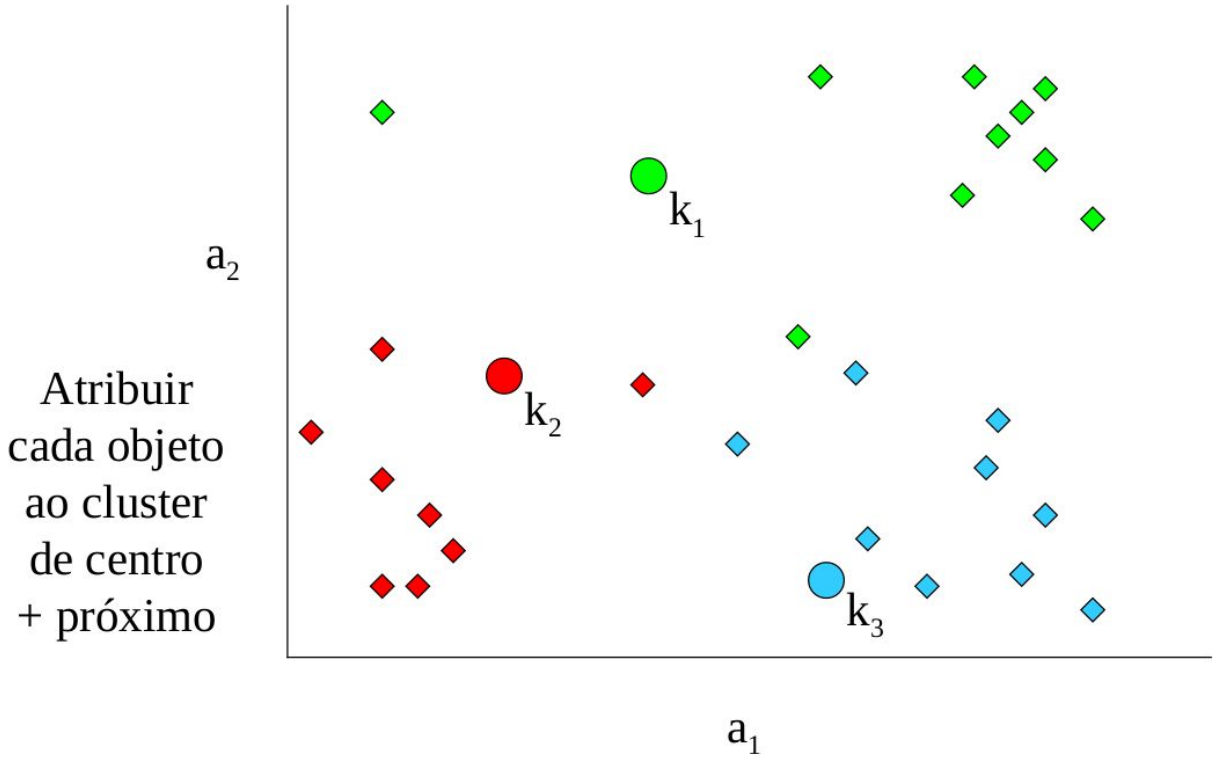
k-Means - passo 1:



Baseado no curso de Gregory Piatetsky-Shapiro, disponível em http://www.kdnuggets.com

Agrupamento

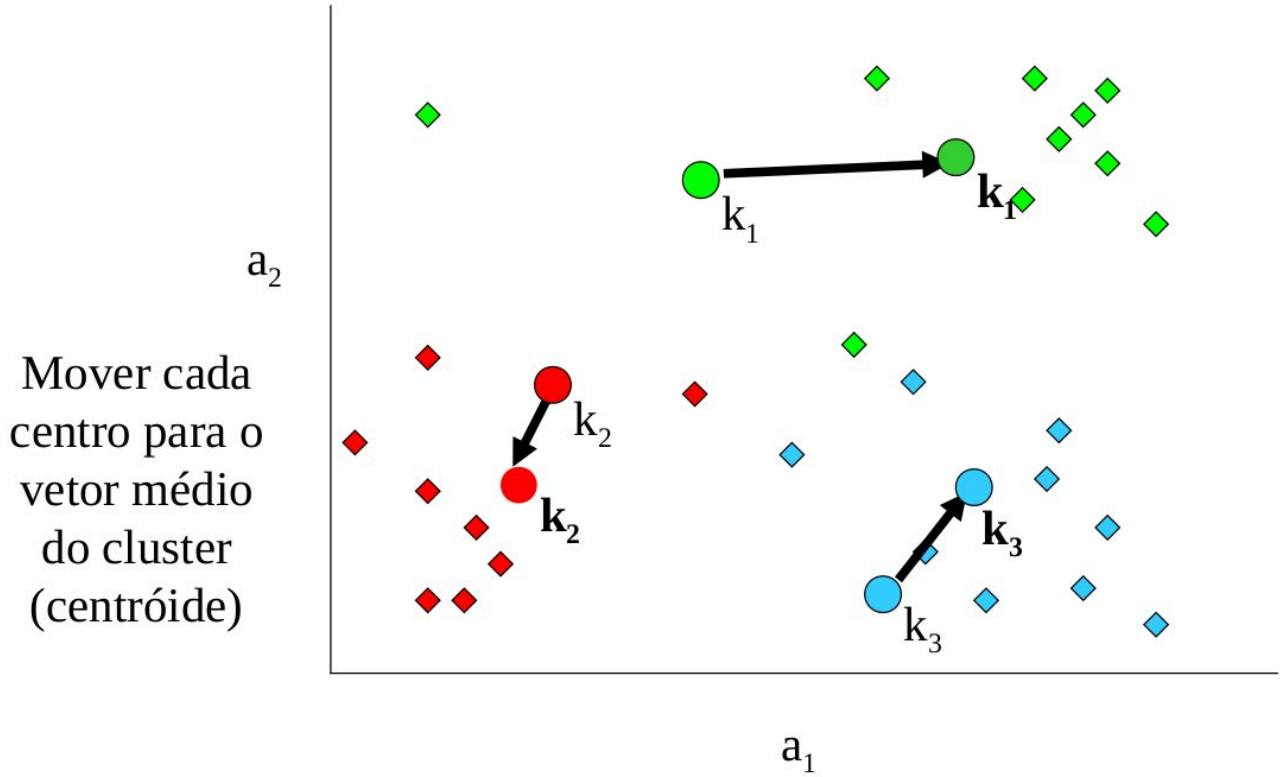
k-Means - passo 2:



Baseado no curso de Gregory Piatetsky-Shapiro, disponível em http://www.kdnuggets.com

Agrupamento

k-Means - passo 3:



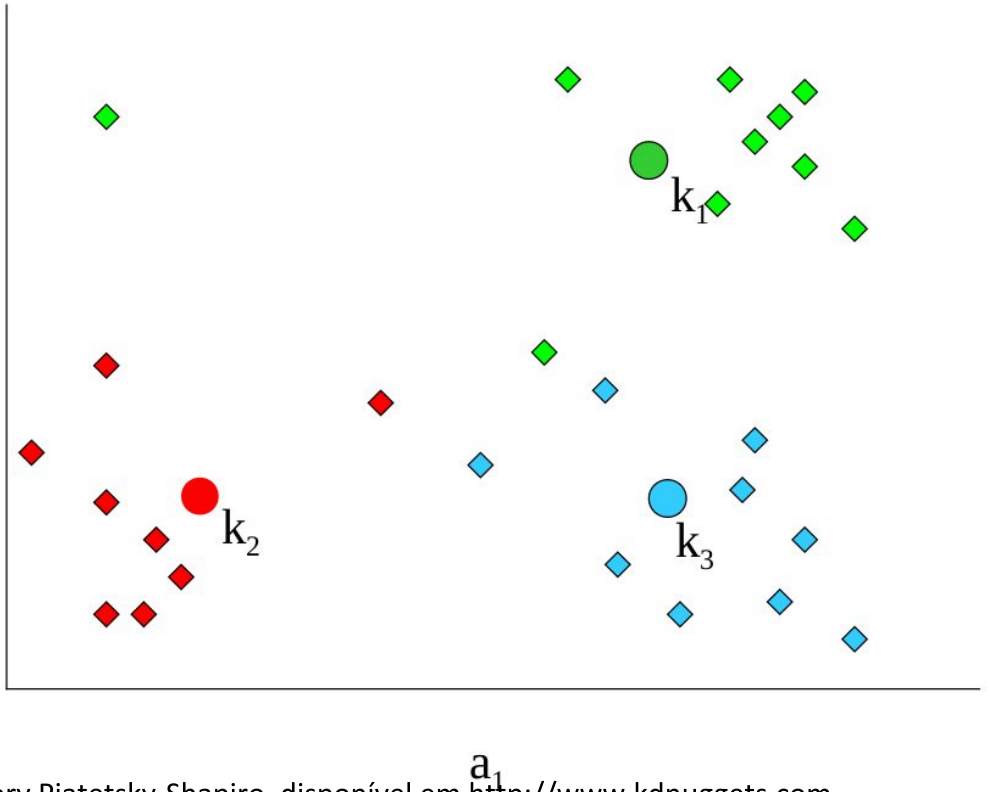
Baseado no curso de Gregory Piatetsky-Shapiro, disponível em http://www.kdnuggets.com

Agrupamento

k-Means:

Re-atribuir objetos aos clusters de centróides mais próximos

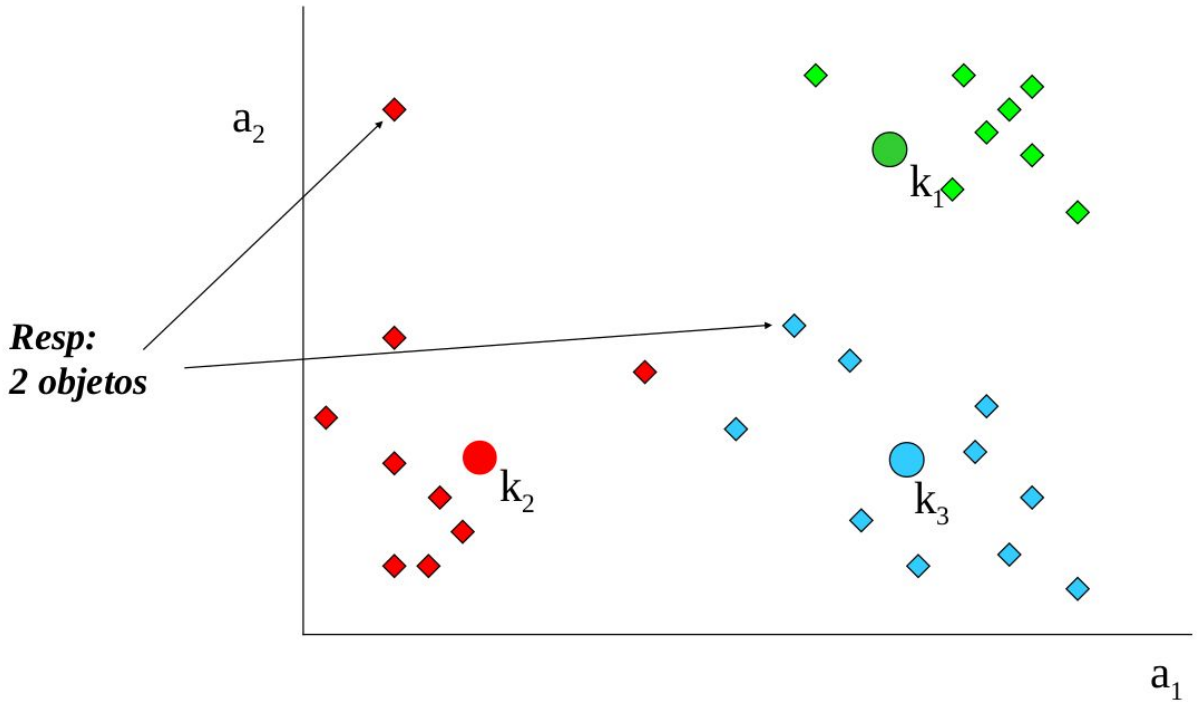
Quais objetos mudarão de cluster?



Baseado no curso de Gregory Piatetsky-Shapiro, disponível em http://www.kdnuggets.com

Agrupamento

k-Means:

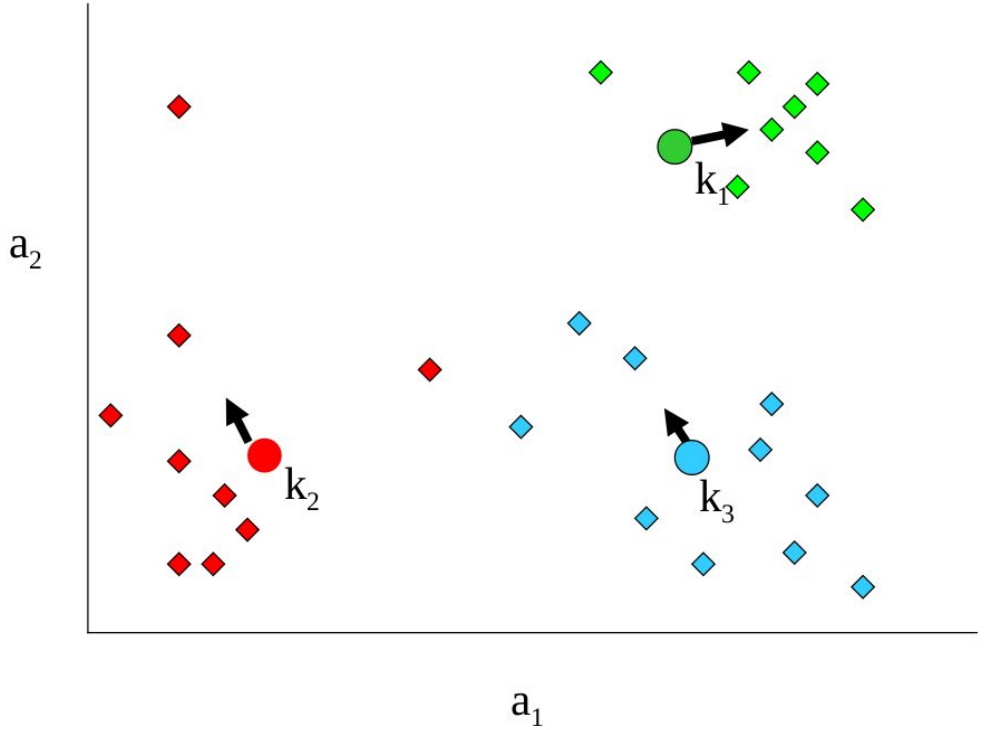


Baseado no curso de Gregory Piatetsky-Shapiro, disponível em http://www.kdnuggets.com

Agrupamento

k-Means:

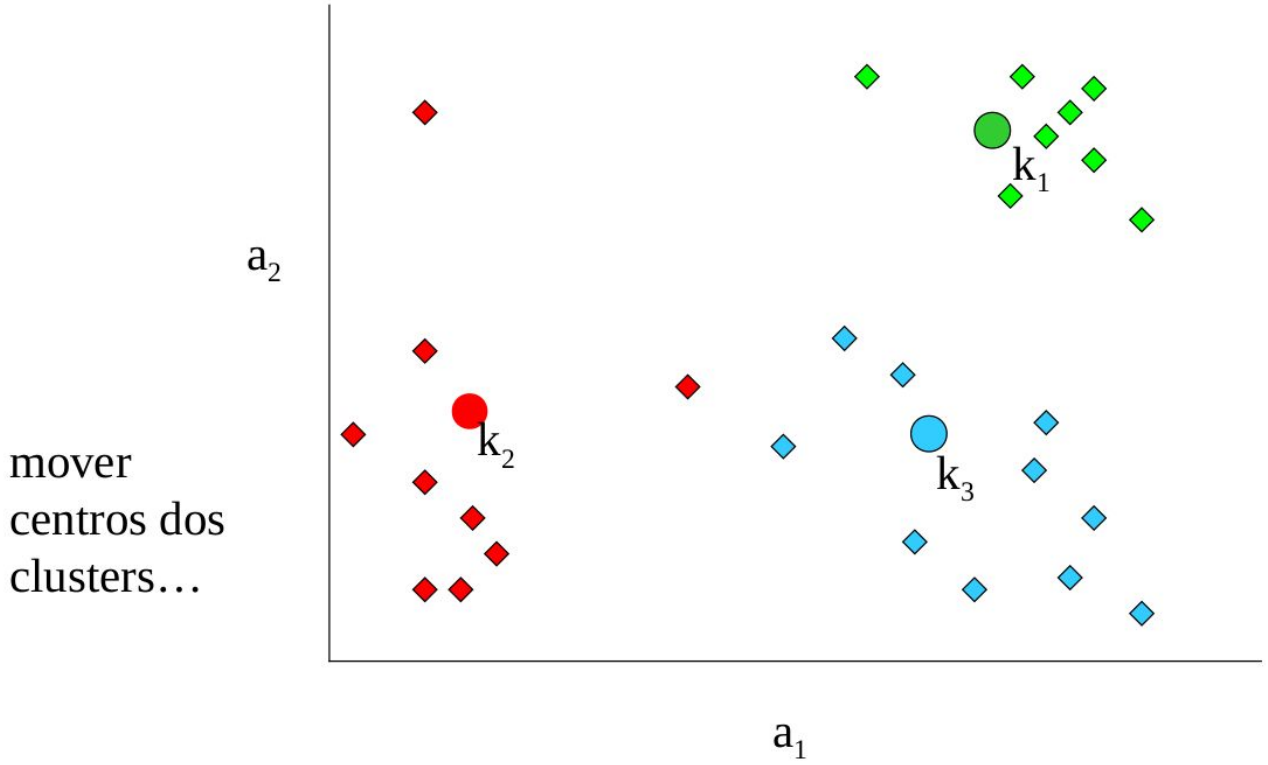
re-calcular
vetores
médios



Baseado no curso de Gregory Piatetsky-Shapiro, disponível em http://www.kdnuggets.com

Agrupamento

k-Means:



Baseado no curso de Gregory Piatetsky-Shapiro, disponível em http://www.kdnuggets.com

Agrupamento

- Que diferenças e/ou (des)vantagens, há entre Kmeans e DBScan?

Agrupamento: métricas de avaliação

- Não é tão claro como na classificação, em que temos um gabarito para conferir o quão bom o classificador foi.
- Geralmente é necessário comparar o agrupamento de diferentes algoritmos – ou de um mesmo algoritmo com diferentes parâmetros - e ver qual faz mais sentido.
- Existem métricas que servem de guia.

Métricas de avaliação: silhueta

- Quanto maior, em geral, melhor.
- O valor da silhueta varia de –1 a 1, e indica o quão similar um objeto é em relação ao seu próprio grupo (coesão), comparado aos demais grupos (separação).
- Ideia: quanto maior, melhor.

Métricas de avaliação: silhueta

- Para cada objeto:
  - a = distância média do objeto para demais elementos de seu grupo.
  - l = lista. Cada elemento é a distância média do objeto para os demais grupos.
  - b = menor distância em l.
  - Se a == b: 0
  - Se a < b: 1 - a/b
  - Se a > b: b/a - 1
- Retorna média das silhuetas.

Métricas de avaliação: silhueta

```haskell
silhouette :: [Point] -> Double
silhouette points = sum scores / fromIntegral (length points)
    where
        scores = [silhouette' x points | x <- points ]

silhouette' :: Point -> [Point] -> Double
silhouette' x points = if a < b then (1 - (a / b)) else if a > b then ((b/a)- 1) else 0
    where
        filteredGroup = map fst $ filter (\p -> snd p == label) points
        label = snd x
        x' = fst x
        a = (sum $ map (euclidean x') filteredGroup) / fromIntegral ((length filteredGroup)-1)
        otherGroups = groupBy ((==) `on` snd) $ sortBy (comparing snd) (filter (\p -> snd p /= label) points) :: [[Point]]
        distancesClusters = map (meanDistanceCluster x) otherGroups
        b = minimum distancesClusters
        meanDistanceCluster x cluster = (sum $ map (euclidean x') (map fst cluster) ) / fromIntegral (length cluster)
```

Métricas de avaliação: silhueta

- Um valor alto para silhueta indica que os objetos estão bem encaixados em seus grupos. Um valor baixo pode indicar que o número de grupos está demasiadamente alto ou baixo.

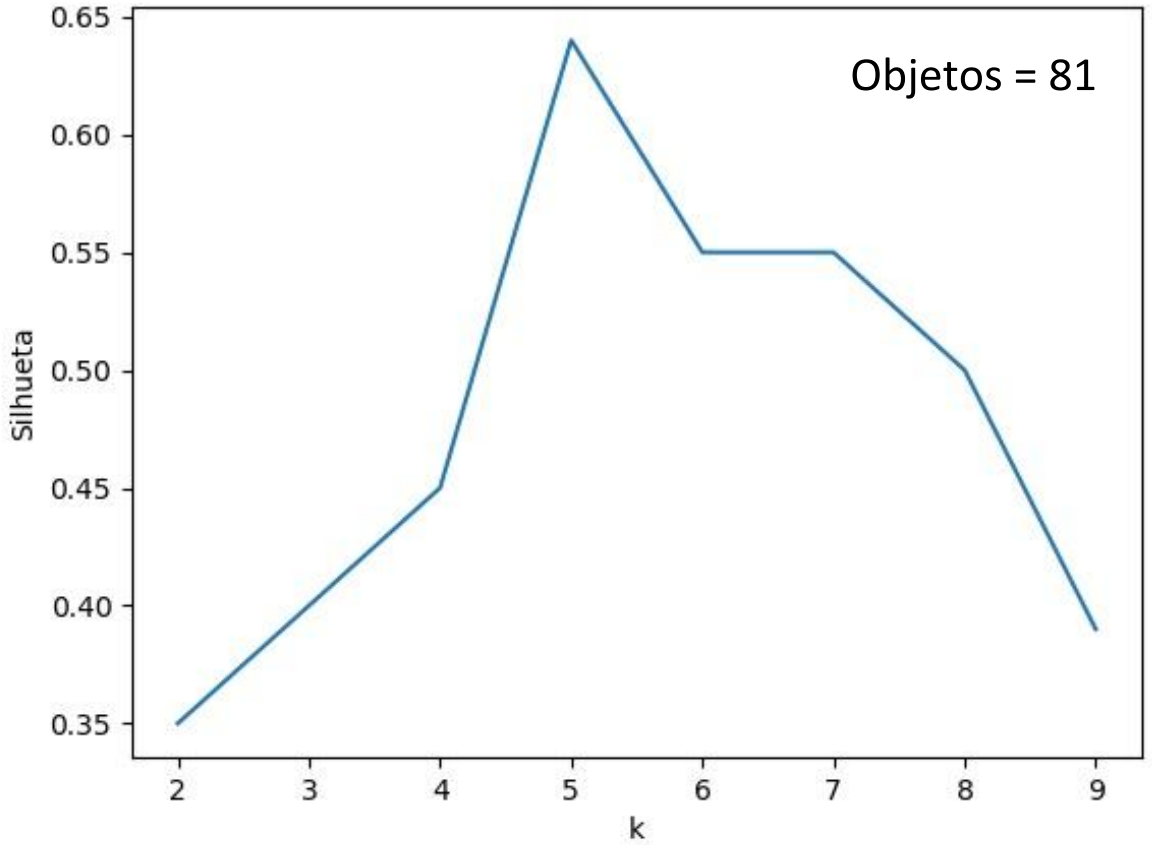
Métricas de avaliação: silhueta

- Um valor alto para silhueta indica que os objetos estão bem encaixados em seus grupos. Um valor baixo pode indicar que o número de grupos está demasiadamente alto ou baixo.
- É comum portanto usar a silhueta para determinar o valor de K do Kmeans.
  - Executa-se o Kmeans com vários valores de k. Seleciona-se aquele com a maior silhueta.

Métricas de avaliação: silhueta

- Um valor alto para silhueta indica que os objetos estão bem encaixados em seus grupos. Um valor baixo pode indicar que o número de grupos está demasiadamente alto ou baixo.
- É comum portanto usar a silhueta para determinar o valor de K do Kmeans.
  - Executa-se o Kmeans com vários valores de k. Seleciona-se aquele com a maior silhueta.
  - Regra do dedão: começamos com k=2 e vamos até k = sqrt(N), sendo N o número de objetos.

Métricas de avaliação: silhueta



Regressão

- Classificação determina uma classe a um objeto.
- Classe é nominal.
- Algoritmos de regressão funcionam de forma semelhante. Diferença é que preveem não uma classe, mas um número.
- Ideia é gerar uma fórmula que preveja um valor numérico a partir dos dados conhecidos.

Regressão: exemplos de aplicação

- Prever preço de imóvel.

Regressão: exemplos de aplicação

- Prever preço de imóvel.
- Estimar preço futuro de uma ação.

Regressão: exemplos de aplicação

- Prever preço de imóvel.
- Estimar preço futuro de uma ação.
- Estimar desempenho acadêmico futuro (notas) de estudantes.

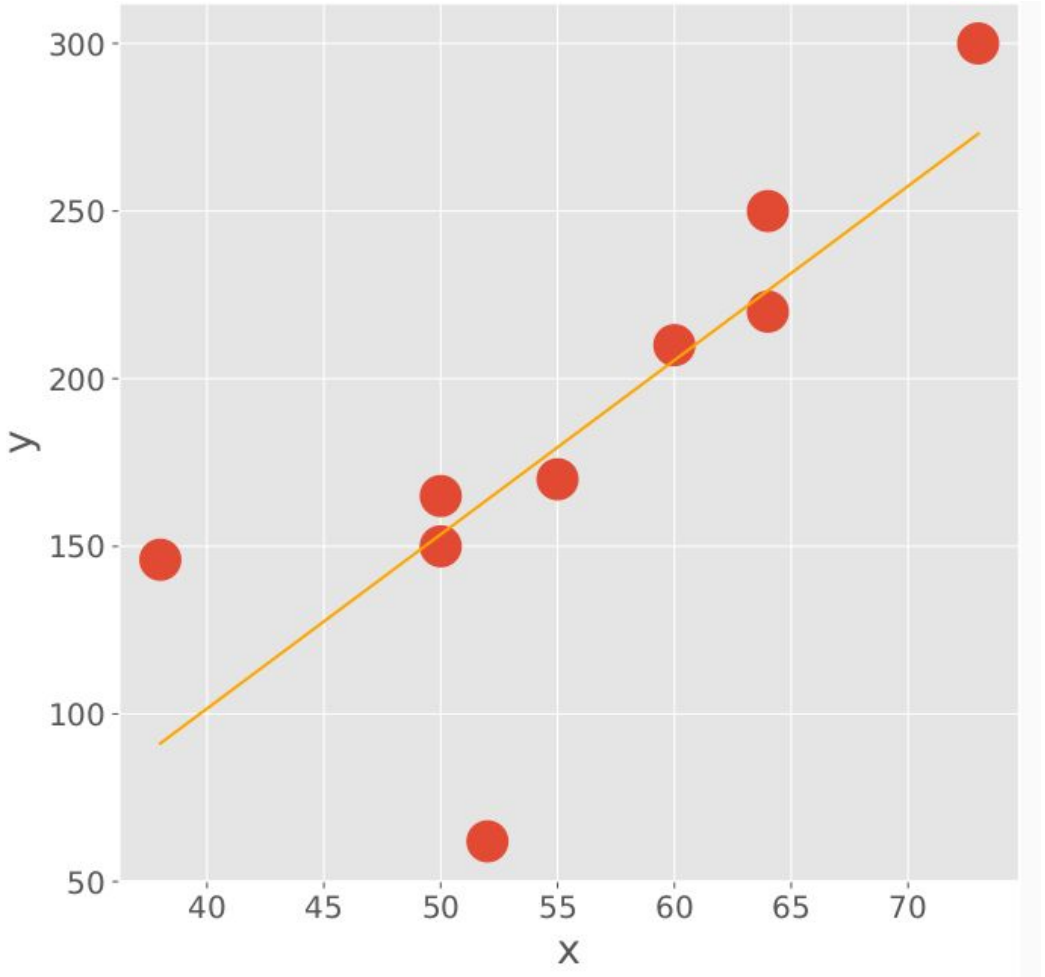
Regressão: exemplos de aplicação

- Prever preço de imóvel.
- Estimar preço futuro de uma ação.
- Estimar desempenho acadêmico futuro (notas) de estudantes.
- Prever a nota que um filme vai receber.

Regressão: exemplos de aplicação

- Prever preço de imóvel.
- Estimar preço futuro de uma ação.
- Estimar desempenho acadêmico futuro (notas) de estudantes.
- Prever a nota que um filme vai receber.
- Prever quando uma pessoa vai morrer.

Regressão Linear



A partir de um conjunto de pontos, determinar qual função linear melhor se encaixa nos pontos.

Dado um conjunto com N objetos {(**Xi,** *Yi*)}, queremos encontrar uma função f(Xi) que aproxime Wi . Xi + Wn = Yi.

Fazemos isso multiplicando cada atributo dos objetos por um peso Wi.

Os pesos são ajustados de forma a minimizar o erro da função.

Ajustando os pesos

- Primeiro, calcula-se o custo.
  - Custo é o erro que nosso modelo, com os pesos atuais, obtém ao tentar estimar Yi.
- Queremos então minimizar esse custo.
- Calculamos o vetor gradiente sobre os pesos.
- Atualizamos os pesos.
- Processo é repetido até um critério de parada (convergência, iterações)

Código

```
apply :: [Double] -> [Double] -> [Double]
apply weights points = map (\(w,x) -> w*x) (zip weights points)


gradDesc :: [[Double]] -> [Double] -> Double -> Int -> [Double]
gradDesc x y α it = gradDesc' x y w0 α it
  where
    w0 = take (length' $ head x) [0.01,0.01..]

gradDesc' :: [[Double]] -> [Double] -> [Double] -> Double -> Int -> [Double]
gradDesc' x y w α it
  | convergiu = w
  | otherwise = gradDesc' x y w' α (it-1)
  where
    w'        = w .+. (α *. nabla)
    nabla     = mediaVetor $ map (\(yi, xi) -> yi *. xi) $ zip (y .-. y') x
    y'        = map (dotprod w) x
    convergiu = (erro' < 1e-6) || (it==0)
    erro'     = erro x y w

erro x y w = mean $ (y .-. y') .^ 2
  where
    y' = map (dotprod w) x

mean :: [Double] -> Double
mean l = (sum l) / (length' l)

polyfeats :: Int -> [[Double]] -> [[Double]]
polyfeats k x = map (\xi -> poly xi !! k) x
  where
    poly x' = foldr f ([1] : repeat []) x'
    f x''   = scanl1 $ (++) . map (*x'')
```

Ajuste dos pesos é feito durante o treinamento.

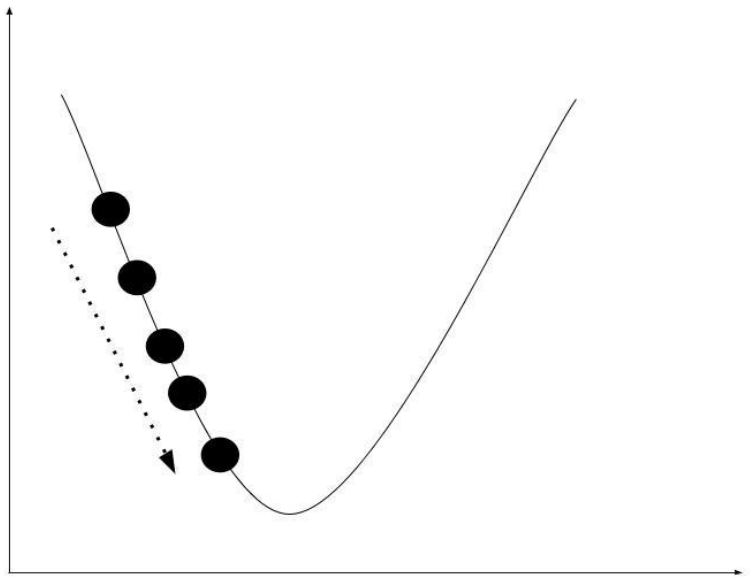
Função gradDesc' realiza diversas chamadas recursivas para si mesma, até convergência ou um valor limite de iterações ser atingido.

Ajustando os pesos

```haskell
gradDesc' :: [[Double]] -> [Double] -> [Double] -> Double -> Int -> [Double]
gradDesc' x y w α it
  | convergiu = w
  | otherwise = gradDesc' x y w' α (it-1)
  where
    w'        = w .+. (α *. nabla)
    nabla     = mediaVetor $ map (\(yi, xi) -> yi *. xi) $ zip (y .-. y') x
    y'        = map (dotprod w) x
    convergiu = (erro' < 1e-6) || (it==0)
    erro'     = erro x y w
```

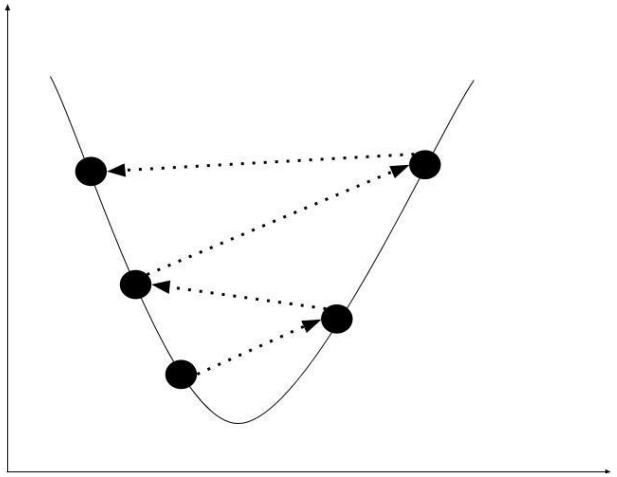
Ajustando os pesos

```
gradDesc' :: [[Double]] -> [Double] -> [Double] -> Double -> Int -> [Double]
gradDesc' x y w α it
  | convergiu = w
  | otherwise = gradDesc' x y w' α (it-1)
  where
    w'        = w .+. (α *. nabla)
    nabla     = mediaVetor $ map (\(yi, xi) -> yi *. xi) $ zip (y .-. y') x
    y'        = map (dotprod w) x
    convergiu = (erro' < 1e-6) || (it==0)
    erro'     = erro x y w
```



Ajustando os pesos

```
gradDesc' :: [[Double]] -> [Double] -> [Double] -> Double -> Int -> [Double]
gradDesc' x y w α it
  | convergiu = w
  | otherwise = gradDesc' x y w' α (it-1)
  where
    w'        = w .+. (α *. nabla)
    nabla     = mediaVetor $ map (\(yi, xi) -> yi *. xi) $ zip (y .-. y') x
    y'        = map (dotprod w) x
    convergiu = (erro' < 1e-6) || (it==0)
    erro'     = erro x y w
```



Código

```haskell
apply :: [Double] -> [Double] -> [Double]
apply weights points = map (\(w,x) -> w*x) (zip weights points)


gradDesc :: [[Double]] -> [Double] -> Double -> Int -> [Double]
gradDesc x y α it = gradDesc' x y w0 α it
  where
    w0 = take (length' $ head x) [0.01,0.01..]

gradDesc' :: [[Double]] -> [Double] -> [Double] -> Double -> Int -> [Double]
gradDesc' x y w α it
  | convergiu = w
  | otherwise = gradDesc' x y w' α (it-1)
  where
    w'        = w .+. (α *. nabla)
    nabla     = mediaVetor $ map (\(yi, xi) -> yi *. xi) $ zip (y .-. y') x
    y'        = map (dotprod w) x
    convergiu = (erro' < 1e-6) || (it==0)
    erro'     = erro x y w

erro x y w = mean $ (y .-. y') .^ 2
  where
    y' = map (dotprod w) x

mean :: [Double] -> Double
mean l = (sum l) / (length' l)

polyfeats :: Int -> [[Double]] -> [[Double]]
polyfeats k x = map (\xi -> poly xi !! k) x
  where
    poly x' = foldr f ([1] : repeat []) x'
    f x''   = scanl1 $ (++) . map (*x'')
```

Com os pesos calibrados, chamamos a função apply nos dados de teste.

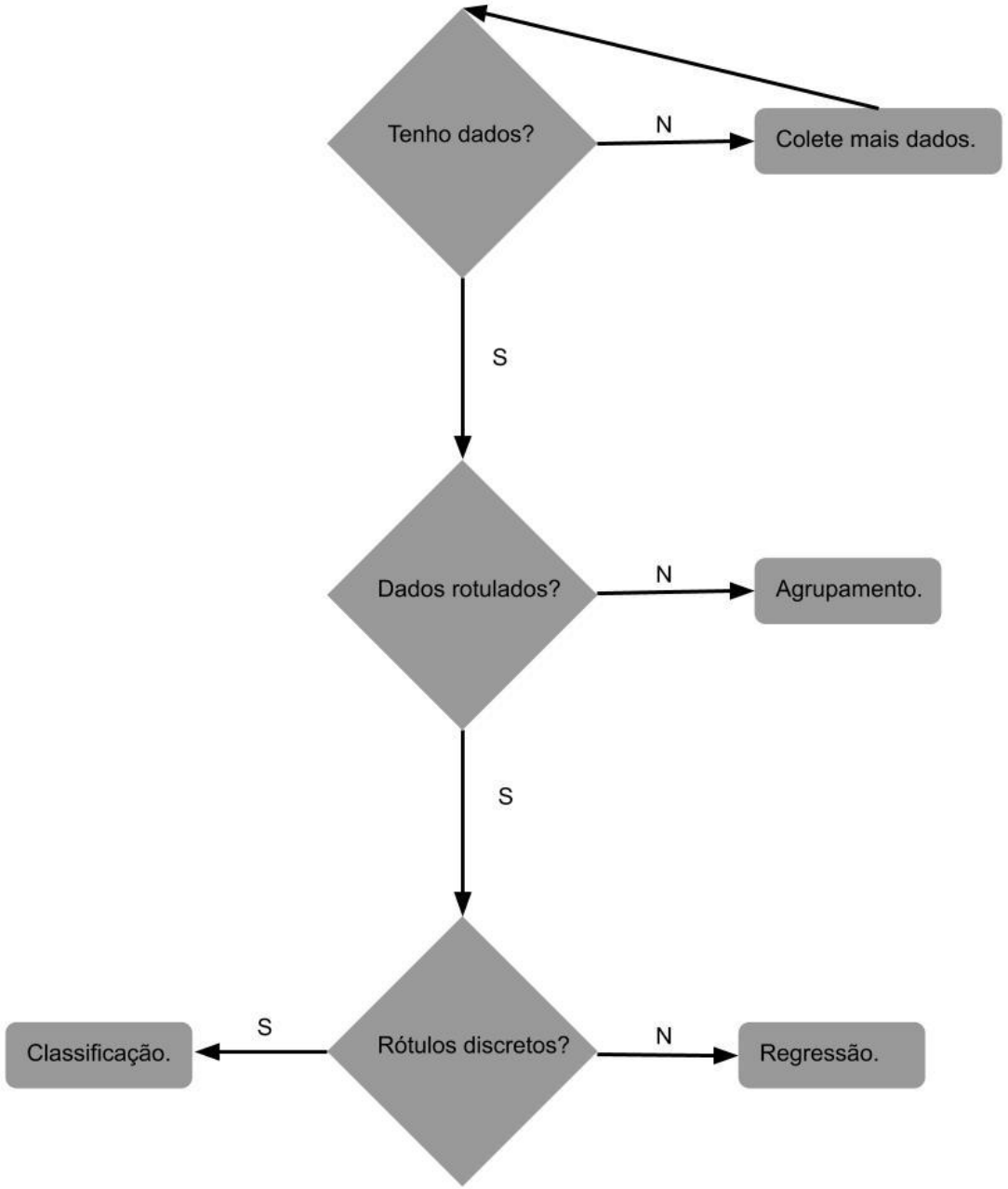
Métricas de avaliação

- Como não temos classes discretas, mas valores no espaço contínuo para estimar, dificilmente acertaremos muitas previsões.
- Root-mean-square error (RMSE) é uma medida para a diferença entre os valores preditos e reais.
- Ao contrário das medidas de classificação, que sempre queremos maximizar, o RMSE ideal é 0.
- RMSE = 0 significa que nosso modelo acertou todos os casos de teste na mosca.

Métricas de avaliação

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y})^2}$$

```haskell
rmse :: [Double] -> [Double] -> Double
rmse ytrue ypred = sqrt (s / n)
    where
        s = sum $ map (\(y, y') -> (y - y')**2 ) (zip ytrue ypred)
        n = fromIntegral $ length ytrue
```


Tenho dados?
N
Colete mais dados.
S
Dados rotulados?
N
Agrupamento.
S
Rótulos discretos?
S
Classificação.
N
Regressão.

Mais:

- HLearn: A Machine Learning Library for Haskell
  - https://izbicki.me/public/papers/tfp2013-hlearn-a-machine-learning-library-for-haskell.pdf
  - https://github.com/mikeizbicki/HLearn
- Biblioteca do professor Fabricio Olivetti
  - https://github.com/folivetti/BIGDATA
- Mining Frequent Patterns with Functional Programming
  - https://waset.org/publications/6340/-mining-frequent-patterns-with-functional-programming
- Models for machine learning and data mining in functional programming
- Introduction to Data Mining, Pang-Ning Tan (tem na biblioteca)