Data Science – IPT
Clustering

Mario Leston e Marcelo Rezende

19 de novembro de 2019

Definição

Uma partição de um conjunto $X = \{x_1, \ldots, x_n\}$ é uma coleção $\mathcal{C} = \{C_1, \ldots, C_k\}$ de subconjuntos de X dotada das seguintes propriedades:

- se $C_i \cap C_j = \emptyset$ se $i \neq j$, e
- $C_1 \cup C_2 \cup \cdots \cup C_k = X$, i.e, a união dos conjuntos que fazem parte de $\mathcal{C}$ coincide com X.

Cada $C \in \mathcal{C}$ é dito uma classe. Escrevemos $x \sim_{\mathcal{C}} y$ se x, y estão numa mesma classe de $\mathcal{C}$. É conveniente também encarar uma partição como uma função $p : [n] \to [k]$. Assim $C_i = \{m \in [n] \mid p(m) = i\}$ para cada $i \in [k]$.

Definição

Uma partição de um conjunto $X = \{x_1, \ldots, x_n\}$ é uma coleção $\mathcal{C} = \{C_1, \ldots, C_k\}$ de subconjuntos de X dotada das seguintes propriedades:

- se $C_i \cap C_j = \emptyset$ se $i \neq j$, e
- $C_1 \cup C_2 \cup \cdots \cup C_k = X$, i.e, a união dos conjuntos que fazem parte de $\mathcal{C}$ coincide com X.

Cada $C \in \mathcal{C}$ é dito uma classe. Escrevemos $x \sim_{\mathcal{C}} y$ se x, y estão numa mesma classe de $\mathcal{C}$. É conveniente também encarar uma partição como uma função $p : [n] \to [k]$. Assim $C_i = \{m \in [n] \mid p(m) = i\}$ para cada $i \in [k]$.

Definição

Uma partição de um conjunto $X = \{x_1, \dots, x_n\}$ é uma coleção $\mathcal{C} = \{C_1, \dots, C_k\}$ de subconjuntos de X dotada das seguintes propriedades:

- se $C_i \cap C_j = \emptyset$ se $i \neq j$, e
- $C_1 \cup C_2 \cup \cdots \cup C_k = X$, i.e, a união dos conjuntos que fazem parte de $\mathcal{C}$ coincide com X.

Cada $C \in \mathcal{C}$ é dito uma classe. Escrevemos $x \sim_{\mathcal{C}} y$ se x, y estão numa mesma classe de $\mathcal{C}$. É conveniente também encarar uma partição como uma função $p : [n] \to [k]$. Assim $C_i = \{m \in [n] \mid p(m) = i\}$ para cada $i \in [k]$.

Definição

Uma partição de um conjunto $X = \{x_1, \ldots, x_n\}$ é uma coleção $\mathcal{C} = \{C_1, \ldots, C_k\}$ de subconjuntos de X dotada das seguintes propriedades:

- se $C_i \cap C_j = \emptyset$ se $i \neq j$, e
- $C_1 \cup C_2 \cup \cdots \cup C_k = X$, i.e, a união dos conjuntos que fazem parte de $\mathcal{C}$ coincide com X.

Cada $C \in \mathcal{C}$ é dito uma classe. Escrevemos $x \sim_{\mathcal{C}} y$ se x, y estão numa mesma classe de $\mathcal{C}$. É conveniente também encarar uma partição como uma função $p : [n] \to [k]$. Assim $C_i = \{m \in [n] \mid p(m) = i\}$ para cada $i \in [k]$.

Definição

Dados:

- um conjunto finito X de $n \geq 1$ objetos,
- uma função distância (ou dissimilaridade) $\delta : X \times X \to \mathbb{R}$ tal que para cada $(x, y) \in X \times X$
  - $\delta(x, y) \geq 0$,
  - $\delta(x, y) = 0$ se e só se $x = y$, e
  - $\delta(x, y) = \delta(y, x)$,

Definição

Dados:

- um conjunto finito X de $n \geq 1$ objetos,
- uma função distância (ou dissimilaridade) $\delta : X \times X \to \mathbb{R}$ tal que para cada $(x, y) \in X \times X$
  - $\delta(x, y) \geq 0$,
  - $\delta(x, y) = 0$ se e só se $x = y$, e
  - $\delta(x, y) = \delta(y, x)$,

Problema

Encontrar uma partição $\mathcal{C}$ tal que:

- se $C \in \mathcal{C}$ e $c_1, c_2 \in C$, então c_1 e c_2 são "similares", e
- se $C_1 \neq C_2 \in \mathcal{C}$ e $c_1 \in C_1$ e $c_2 \in C_2$, então c_1 e c_2 não são "similares".
- Nada formal a noção de "similaridade".
- O que podemos fazer????

- Vamos tentar tornar mais precisa a noção de uma partição que aglomera objetos similares e separa objetos não-similares.
- Considere uma função π que recebe um conjunto finito X e uma função de dissimilaridade $\delta : X \times X \to \mathbb{R}$ e produz uma partição de X.
- Vamos chamar uma tal função de função de agrupamento.
- Vamos impor algumas condições “naturais” sobre π.

Invariante na escala

A função π é invariante na escala se para cada X finito e cada função de dissimilaridade δ e cada $\alpha > 0$,

$$\pi(X, \delta) = \pi(X, \alpha\delta).$$

Rica

A função π é rica se para cada X finito e cada partição $\mathcal{C}$ de X, existe uma função de dissimilaridade δ tal que $\pi(X, \delta) = \mathcal{C}$.

Consistente

A função π é consistente se para cada par de funções de dissimilaridade δ e δ' tal que para cada $x, y \in X$

- $x \sim_{\pi(X,\delta)} y \Rightarrow \delta'(x, y) \leqslant \delta(x, y)$, e
- $x \nsim_{\pi(X,\delta)} y \Rightarrow \delta'(x, y) \geqslant \delta(x, y)$.

Então $\pi(X, \delta) = \pi(X, \delta')$.

Teorema (Kleinberg)

Não existe função de agrupamento que é invariante na escala, rica, e consistente.

Prova.

- Suponha que tal função, π, exista.
- Tome um conjunto X com $|X| \geqslant 3$.
- π é rica donde existe δ tal que $\pi(X, \delta) = \{\{x\} \mid x \in X\}$.
- π é rica donde existe δ' tal que $\pi(X, \delta') \neq \pi(X, \delta)$.
- Escolha $\alpha > 0$ tal que $\alpha\delta' \geqslant \delta$ e ponha $\delta'' := \alpha\delta'$.
- Invariância de π implica que $\pi(X, \delta'') = \pi(X, \delta')$.
- (i) $x_1 \neq x_2 \in X \Rightarrow x_1 \not\sim_{\pi(X,\delta)} x_2$.
- $\delta'' \geqslant \delta$, π consistente, e (i) $\Rightarrow \pi(X, \delta'') = \pi(X, \delta)$.
- Contradição $\therefore$ uma tal π não existe.

$\square$

Dissimilaridades

- l_p-métrica: $\delta_p(x, y) = \left(\sum_{i=1}^{d} w_i |x_i - y_i|^p \right)^{1/p}$.
- Euclidiana: $p = 2$ e $w_i = 1$ para cada $i = 1, \ldots, d$.
- Manhattan: $p = 1$.
- l_∞: $\delta_\infty(x, y) := \max_{1 \leqslant i \leqslant d} w_i |x_i - y_i|$.

Extensões

Suponha que $C \subseteq \mathbb{R}^d$ é finito e não-vazio. Precisamos estender uma função de dissimilaridade δ para um vetor x em relação ao conjunto C. Algumas alternativas:

- $\delta(x, C) := \max_{y \in C} \delta(x, y)$.
- $\delta(x, C) := \min_{y \in C} \delta(x, y)$.
- Podemos também escolher um representante de $z \in C$ e por $\delta(x, C) := \delta(x, z)$.

Além disso, também precisamos estender δ para subcojuntos não-vazios e disjuntos de X. Pode ser feito de forma similar.

Categorias de algoritmos

- *Algoritmos sequenciais*: Produzem uma única partição tipicamente dependente da ordem na qual os vetores são apresentados.
- *Hierárquicos aglomerativos*: Produzem uma sequência de partições com de tamanho decrescente. Cada iteração consiste em unir duas classes distintas de uma partição em uma única classe. Produzem uma coleção encaixada de conjuntos.
- *Hierárquicos divisivos*: Produzem uma sequência de partições de tamanho crescente. Cada passo consiste na divisão de uma das classes de uma partição.

Categorias de algoritmos

- *Otimização*: Produzem uma partição que depende de uma certa função custo. Tais algoritmos tipicamente produzem um ótimo local e não global. Dividida em:
    1. *Rígida*: cada vetor pertence a um e só uma classe da partição.
    2. *Probabilística*: Cada vetor x é atribuído a uma classe da C da partição tal que $P(C|x)$ é máxima.
    3. *Fuzzy*: Cada vetor pertence a uma classe com um certo grau.
- *SVM*: Baseados em máquinas de suporte vetorial não-lineares, mapeando o espaço original para um espaço de dimensão maior.

O algoritmo recebe:

- vetores $x_1, \ldots, x_n$,
- dissimilaridade δ,
- tamanho máximo da partição, k, e
- limiar θ.

Algoritmo sequencial

```
def basic(x_1, ..., x_n, δ, k, θ):
    C_1 := {x_1};  m := 1
    for i in range(2, n):
        seja C_j tal que δ(x_i, C_j) = min{δ(x_i, C_ℓ) | ℓ ∈ {1, ..., m}}
        if δ(x_i, C_j) > θ and m < k:
            m := m + 1; C_m := {x_i}
        else:
            C_j := C_j ∪ {x_i}
    return {C_1, ..., C_m}
```

O algoritmo recebe:

- vetores $x_1, \ldots, x_n$,
- dissimilaridade δ,
- tamanho máximo da partição, k, e
- limiar θ.

Algoritmo sequencial

```
def two_pass(x_1, ..., x_n, δ, k, θ):
    C_1 := {x_1};  m := 1;  Y := ∅
    for i in range(2, n):
        seja C_j tal que δ(x_i, C_j) = min{δ(x_i, C_ℓ) | ℓ ∈ {1, ..., m}}
        if δ(x_i, C_j) > θ and m < k:
            m := m + 1; C_m := {x_i}
        else:
            Y := Y ∪ {x_i}
    for y in Y:
        seja C_j tal que δ(y, C_j) = min{δ(y, C_ℓ) | ℓ ∈ {1, ..., m}}
        C_j := C_j ∪ {y}.
    return {C_1, ..., C_m}
```

Partições encaixadas

Sejam $\mathcal{C}_1$ e $\mathcal{C}_2$ partições de um conjunto X. Dizemos que $\mathcal{C}_1$ está encaixada em $\mathcal{C}_2$ se cada classe de $\mathcal{C}_1$ é um subconjunto de uma classe de $\mathcal{C}_2$, i.e,

$$C \in \mathcal{C}_1 \Rightarrow \exists D \in \mathcal{C}_2 : C \subseteq D.$$

Escrevemos $\mathcal{C}_1 \sqsubseteq \mathcal{C}_2$ se $\mathcal{C}_1$ está encaixada em $\mathcal{C}_2$.

O algoritmo recebe

- pontos $x_1, \ldots, x_n$, e
- uma função de similaridade, δ, estendida,

e devolve uma coleção de partições $\mathcal{C}_1, \ldots, \mathcal{C}_n$ tal que $\mathcal{C}_i \sqsubseteq \mathcal{C}_{i+1}$ para $i = 1, \ldots, n-1$.

Hierárquico

```
def hierarquico(x_1,...,x_n,δ):
    𝒞_1 := {{x_i} | i = 1,...,n};  m := 1
    for _ in range(n − 1):
        sejam C_i e C_j em 𝒞 tais que δ(C_i, C_j) é mínimo
        𝒞_{m+1} := (𝒞_m − {C_i, C_j}) ∪ {C_i ∪ C_j}
        m := m + 1
    return 𝒞_1,...,𝒞_m
```

Um algoritmo elementar

```
def clusters(x_1, ..., x_n, δ, k):
    𝒞 := {{x_i} | i = 1, ..., n}
    suponha que X × X = {e_0, ..., e_{m-1}} e
    δ(e_{i-1}) ≤ δ(e_i) para cada i ∈ {1, ..., m-1}
    for i in range(m):
        if k = |𝒞|: break
        sejam a e b as pontas de e_i
        sejam C_a e C_b em 𝒞 tais que a ∈ C_a e b ∈ C_b
        if C_a ≠ C_b:
            𝒞 := (𝒞 − {C_a, C_b}) ∪ {C_a ∪ C_b}
     return 𝒞
```

Definição

O baricentro (ou centróide) de um conjunto finito X de $n \geq 1$ pontos em $\mathbb{R}^d$ é o ponto $\frac{1}{|X|}\sum_{x\in X} x$.

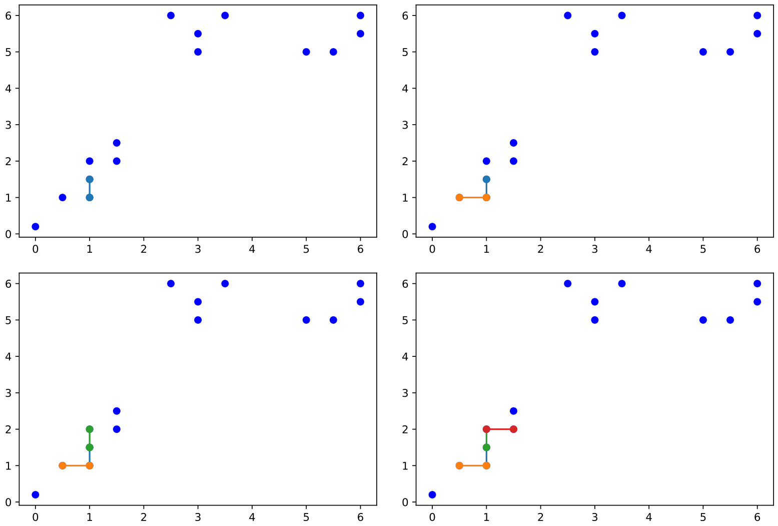
Exemplo

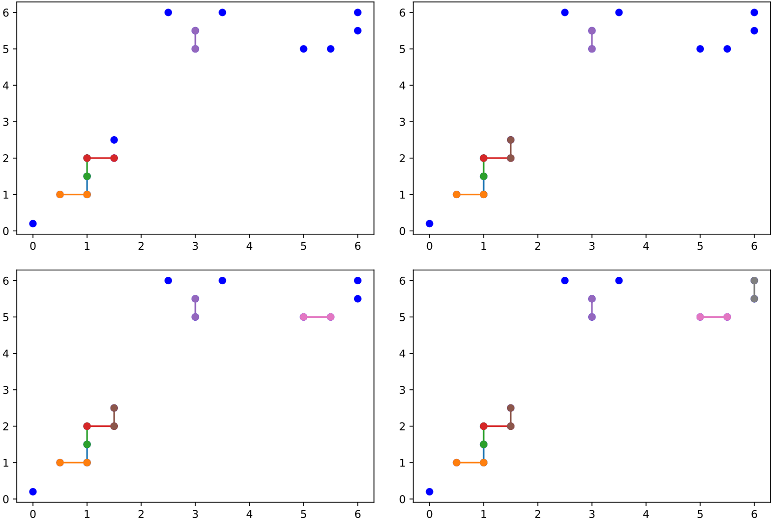
O baricentro do conjunto $\{(1,2),(2,5),(3,2)\}$ é o ponto

$$\frac{1}{3}[(1,2)+(2,5)+(3,2)] = \frac{1}{3}(6,9) = (2,3).$$

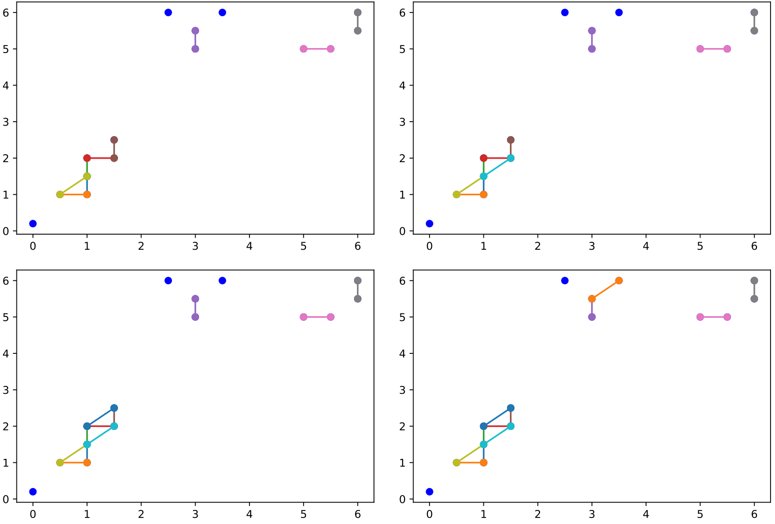
Classificando

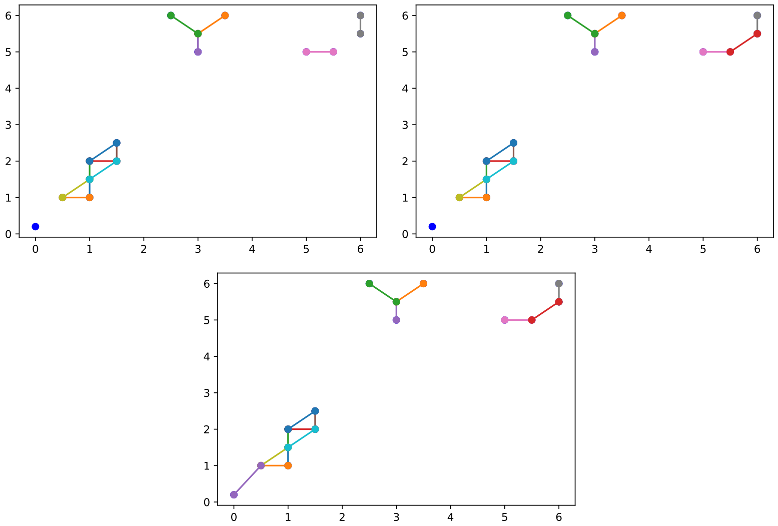
Seja $\{C_1,\ldots,C_k\}$ uma partição de X e $c_1,\ldots,c_k$ os centróides de $C_1,\ldots,C_k$. Para cada ponto x, devolva i tal que $\delta(c_i,x) \leq \delta(c_j,x)$ para cada $j \in \{1,\ldots,k\}$.

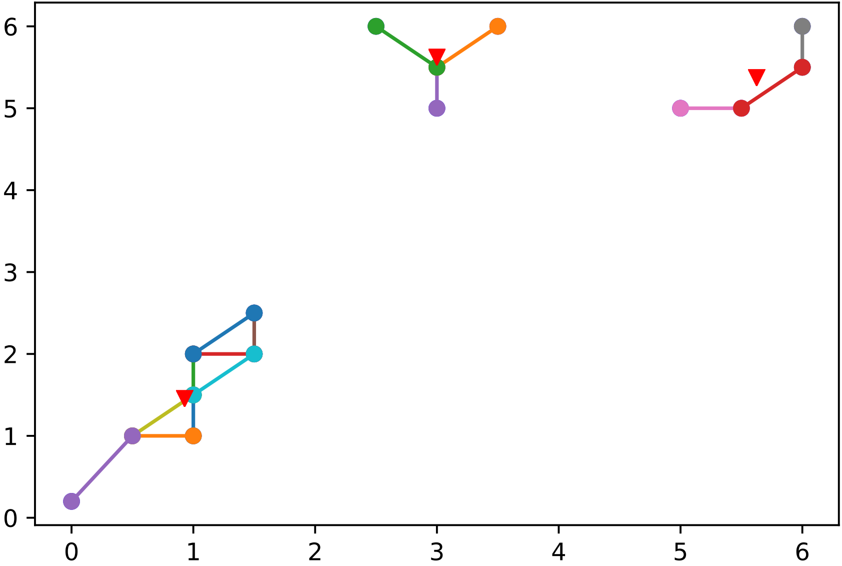

0 1 2 3 4 5 6
0 1 2 3 4 5 6



0
1
2
3
4
5
6


0 1 2 3 4 5 6


0
1
2
3
4
5
6


0
1
2
3
4
5
6
0
1
2
3
4
5
6

class UF

```python
class UF:
  def __init__(self, elems: np.ndarray) -> None:
      self._rank = [0] * len(elems)
      self._repr = [i for i in range(len(elems))]
      self._size = len(elems)

  def find(self, elem: int) -> int:
      e, rpr = elem, self._repr
      while rpr[e] != e:
          e = rpr[e]
      while rpr[elem] != elem:
          elem, rpr[elem] = rpr[elem], e
      return e
```

Continuação class UF

```
def union(self, x: int, y: int) -> None:
    e, f = self.find(x), self.find(y)
    if e == f:
        return
    rpr, rank = self._repr, self._rank
    if rank[f] < rank[e]:
        e, f = f, e
    rpr[e] = f
    if rank[f] == rank[e]:
        rank[f] += 1
    self._size -= 1

def size(self):
    return self._size
```

Implementação do algoritmo

```
def lex_pairs(xs):
    for i in range(len(xs)):
        for k in range(i + 1, len(xs)): yield xs[i], xs[k]
 class SimpleClustering:
    def __init__(self, elems, dist, nclusters):
        self._size = len(elems)
        edges = sorted([(dist(elems[i], elems[k]), i, k)
                   for (i, k) in lex_pairs(list(range(len(elems))))])
        self._uf = uf = UF(elems)
        for (c, i, k) in edges:
            if uf.size() == nclusters: break
            uf.union(i, k)

    def clusters(self):
        cls, uf = defaultdict(set), self._uf
        for i in range(self._size):
            r = uf.find(i)
            if r not in cls: cls[r].add(r)
            cls[r].add(i)
        return cls
```

Otimização

Suponha que $X = \{x_1, \ldots, x_n\} \subseteq \mathbb{R}^d$ é um conjunto finito e $\delta : X \times X \to \mathbb{R}$ é uma função de dissimilaridade. Seja $k \in \mathbb{N}$. Queremos encontrar uma partição $p : [n] \to [k]$ e pontos $c_1, \ldots, c_k$ em $\mathbb{R}^d$ tais que

$$J(p, c_1, \ldots, c_k) := \sum_{i=1}^{n} \delta(x_i, c_{p(i)})$$

é mínimo.

- Má notícia: o problema é NP-difícil.
- Difícil até de aproximar.
- Ideia: atacar o problema com uma heurística.
- Logo, não há garantias quanto à qualidade da solução.
- Nem tão grave, já que, como vimos, é difícil fornecer uma definição para a noção de agrupamento.

- Má notícia: o problema é NP-difícil.
- Difícil até de aproximar.
- Ideia: atacar o problema com uma heurística.
- Logo, não há garantias quanto à qualidade da solução.
- Nem tão grave, já que, como vimos, é difícil fornecer uma definição para a noção de agrupamento.

- Má notícia: o problema é NP-difícil.
- Difícil até de aproximar.
- Ideia: atacar o problema com uma heurística.
- Logo, não há garantias quanto à qualidade da solução.
- Nem tão grave, já que, como vimos, é difícil fornecer uma definição para a noção de agrupamento.

- Má notícia: o problema é NP-difícil.
- Difícil até de aproximar.
- Ideia: atacar o problema com uma heurística.
- Logo, não há garantias quanto à qualidade da solução.
- Nem tão grave, já que, como vimos, é difícil fornecer uma definição para a noção de agrupamento.

- Má notícia: o problema é NP-difícil.
- Difícil até de aproximar.
- Ideia: atacar o problema com uma heurística.
- Logo, não há garantias quanto à qualidade da solução.
- Nem tão grave, já que, como vimos, é difícil fornecer uma definição para a noção de agrupamento.

- Má notícia: o problema é NP-difícil.
- Difícil até de aproximar.
- Ideia: atacar o problema com uma heurística.
- Logo, não há garantias quanto à qualidade da solução.
- Nem tão grave, já que, como vimos, é difícil fornecer uma definição para a noção de agrupamento.

Heurística k-means

A ideia é resolver dois problemas "fáceis". O primeiro deles consiste em fixar $\mathbf{c} := (c_1, \ldots, c_k)$ e determinar $p : [n] \to [k]$ que minimize $J(p, c_1, \ldots, c_k)$, ou seja, queremos minimizar a função

$$J_{\mathbf{c}}(p) = J(p, c_1, \ldots, c_k).$$

O segundo consiste em fixar $p : [n] \to [k]$ e determinar $c_1, \ldots, c_k$ que minimizem $J(p, c_1, \ldots, c_k)$, ou seja, queremos minimizar a função

$$J_p(c_1, \ldots, c_k) = J(p, c_1, \ldots, c_k).$$

Minimizando $J_{\mathbf{c}}$

Dado $\mathbf{c} = (c_1, \ldots, c_k)$, candidatos a "centros" de cada classe de uma partição, temos que encontrar uma partição $p^* : [n] \to [k]$ tal que $J_{\mathbf{c}}(p^*)$ é mínimo. Eis uma tal partição:

$$p^*(i) = \operatorname{argmin}\{\delta(x_i, c_j) \mid j \in [k]\}.$$

para cada $i \in [n]$.

Observação

Admitimos que o mínimo acima é atingido para no máximo um único j para cada i. Se este não for o caso, escolha um menor j.

Minimizando $J_{\mathbf{c}}$

Dado $\mathbf{c} = (c_1, \ldots, c_k)$, candidatos a "centros" de cada classe de uma partição, temos que encontrar uma partição $p^* : [n] \longrightarrow [k]$ tal que $J_{\mathbf{c}}(p^*)$ é mínimo. Eis uma tal partição:

$$p^*(i) = \operatorname{argmin}\{\delta(x_i, c_j) \mid j \in [k]\}.$$

para cada $i \in [n]$.

Observação

Admitimos que o mínimo acima é atingido para no máximo um único j para cada i. Se este não for o caso, escolha um menor j.

Minimizando J_p

Dado uma partição $p : [n] \to [k]$, encontrar centros $\mathbf{c}^* = (c_1^*, \ldots, c_k^*)$ tais que $J_p(\mathbf{c}^*)$ é mínimo.

- Vamos supor que δ é o quadrado da dissimilaridade euclidiana.
- Neste caso, J_p é convexa.
- Logo, um mínimo local também é global.

Minimizando J_p

Ora,

$$
\begin{aligned}
J_p(c_1, \ldots, c_k) &= \sum_{i=1}^{n} \delta(x_i, c_{p(i)}) \\
&= \sum_{j \in [k]} \sum_{i \in p^{-1}(j)} \delta(x_i, c_j) \\
&= \sum_{j \in [k]} \sum_{i \in p^{-1}(j)} \sum_{e=1}^{d} (x_{ie} - c_{je})^2
\end{aligned}
$$

Logo, é suficiente mostrar como minimizar um dos componentes

$$
\tau(c_j) = \sum_{i \in p^{-1}(j)} \sum_{e=1}^{d} (x_{ie} - c_{je})^2
$$

da soma acima.

Minimizando J_p

Ora,

$$\begin{aligned} J_p(c_1, \ldots, c_k) &= \sum_{i=1}^{n} \delta(x_i, c_{p(i)}) \\ &= \sum_{j \in [k]} \sum_{i \in p^{-1}(j)} \delta(x_i, c_j) \\ &= \sum_{j \in [k]} \sum_{i \in p^{-1}(j)} \sum_{e=1}^{d} (x_{ie} - c_{je})^2 \end{aligned}$$

Logo, é suficiente mostrar como minimizar um dos componentes

$$\tau(c_j) = \sum_{i \in p^{-1}(j)} \sum_{e=1}^{d} (x_{ie} - c_{je})^2$$

da soma acima.

Minimizando J_p

Fixe $f \in [d]$. Então

$$\partial_{jf}\tau(c_j) = \sum_{i \in p^{-1}(j)} -2(x_{if} - c_{jf}).$$

Como τ é convexa, um ponto c_j é um mínimo se o gradiente em c_j é zero. Logo,

$$\sum_{i \in p^{-1}(j)} -2(x_{if} - c_{jf}) = 0.$$

Segue daí que

$$c_{jf} = \frac{1}{|p^{-1}(j)|} \sum_{i \in p^{-1}(j)} x_{if}.$$

Baricentro

Suponha que $C \subseteq \mathbb{R}^d$ é finito e não-vazio. O baricentro de C é o ponto

$$\frac{1}{|C|} \sum_{x \in C} x.$$

Assim, $\mathbf{c}^* = (c_1^*, \ldots, c_k^*)$ que minimiza J_p é tal que c_j^* é o baricentro de

$$C_j := \{x_i \mid i \in p^{-1}(j)\}.$$

Baricentro

Suponha que $C \subseteq \mathbb{R}^d$ é finito e não-vazio. O baricentro de C é o ponto

$$\frac{1}{|C|} \sum_{x \in C} x.$$

Assim, $\mathbf{c}^* = (c_1^*, \ldots, c_k^*)$ que minimiza J_p é tal que c_j^* é o baricentro de

$$C_j := \{x_i \mid i \in p^{-1}(j)\}.$$

Para cada $\mathbf{c} = (c_1, \ldots, c_k)$, vamos escrever

$$\mathsf{optJc}(\mathsf{c})$$

para denotar a partição p^* que minimiza $J_{\mathbf{c}}$, e

$$\mathsf{optJp}(p)$$

para denotar c^* que minimiza J_p.

heurística k-means

```
def kmeans(k, x_1, ..., x_n):
    escolha aleatoriamente c := (c_1, ..., c_k)
    while True:
        c' := (optJp ∘ optJc)(c)
        if c = c': return c
        c := c'
```

Observações

- Pode ser necessária uma normalização (padronização) dos dados. Por exemplo, via *z-score*:

$$u_{ij} := \frac{x_{ij} - b_j}{\sigma_j},$$

onde b é o baricentro de $x_1, \ldots, x_n$ e

$$\sigma_j := \left[\frac{1}{n-1}\sum_{i=1}^{n}(x_{ij} - b_j)^2\right]^{\frac{1}{2}}$$

é a variância do j-ésimo atributo.

- O algoritmo é guloso e depende das condições iniciais.
- É sensitivo a *outliers*.
- O tamanho da partição, k, deve ser conhecido a priori.
- O tamanho da partição produzida pode ser menor que k.

Definição

Dada uma partição p e uma dissimilaridade δ, a dissimilaridade de $i \in [n]$ em relação ao cluster $\ell \in [k]$ é o número

$$a(i, p^{-1}(\ell)) = \frac{\sum_{j \in p^{-1}(\ell)} \delta(x_i, x_j)}{|p^{-1}(\ell)|}$$

A dissimilaridade (p, δ)-média é o número

$$a(i) = a(i, p^{-1}(p(i)))$$

para cada $i \in [n]$.
Um vizinho de $i \in [n]$ é um cluster $p^{-1}(\ell) \neq p^{-1}(i)$ tal que $a(i, p^{-1}(\ell))$ é mínimo. Ponha $b(i) = a(i, p^{-1}(\ell))$ para cada $i \in [n]$.

Definição

A silhueta de $i \in [n]$ se $|p^{-1}(p(i))| \geq 2$ é o número

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}}.$$

Se $|p^{-1}(p(i))| = 1$, então $s(i) = 0$.

Note que $-1 \leq s(i) \leq 1$ para cada $i \in [n]$. Se $s(i)$ é perto de 1, então i está bem classificado; se $s(i)$ é próximo de zero, não está claro qual o melhor cluster para i; e se $s(i)$ for próximo de -1, então i está mal classificado.

Definição

A silhueta média de um cluster $C = p^{-1}(j)$ para $j \in [k]$ é o número

$$s(C) = \frac{\sum_{i \in C} s(i)}{|C|}.$$

Uma silhueta média acima de 0.7 indica que k é possivelmente um bom número de clusters.