Quiz navigation

i i 1 2 3 i 4 i 5 6

Finish review

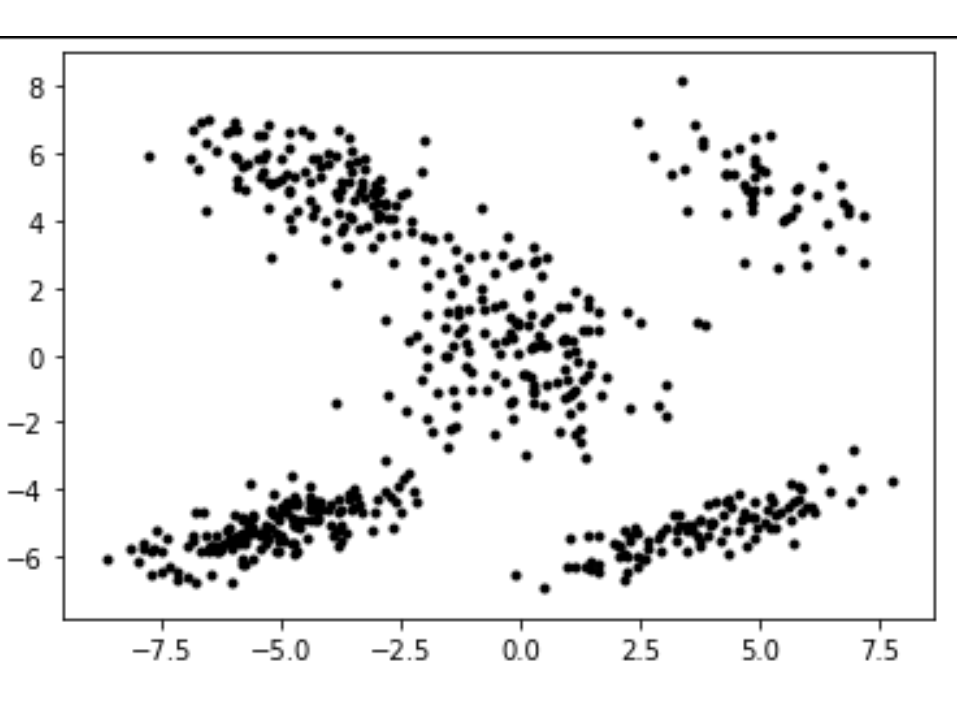
Started on	Monday, 14 December 2020, 9:41 AM
State	Finished
Completed on	Saturday, 9 January 2021, 7:42 PM
Time taken	26 days 10 hours
Grade	**20.00** out of 20.00 (**100**%)

Information

Flag question

O tema deste projeto é muito relevante no mundo da Informática, sendo usado em áreas como a Prospeção de Dados (em inglês, *Data Mining*), na compressão de dados, ou em *Machine Learning*.

Seja a seguinte imagem que representa um conjunto de pontos 2D gerados por um processo aleatório desconhecido:

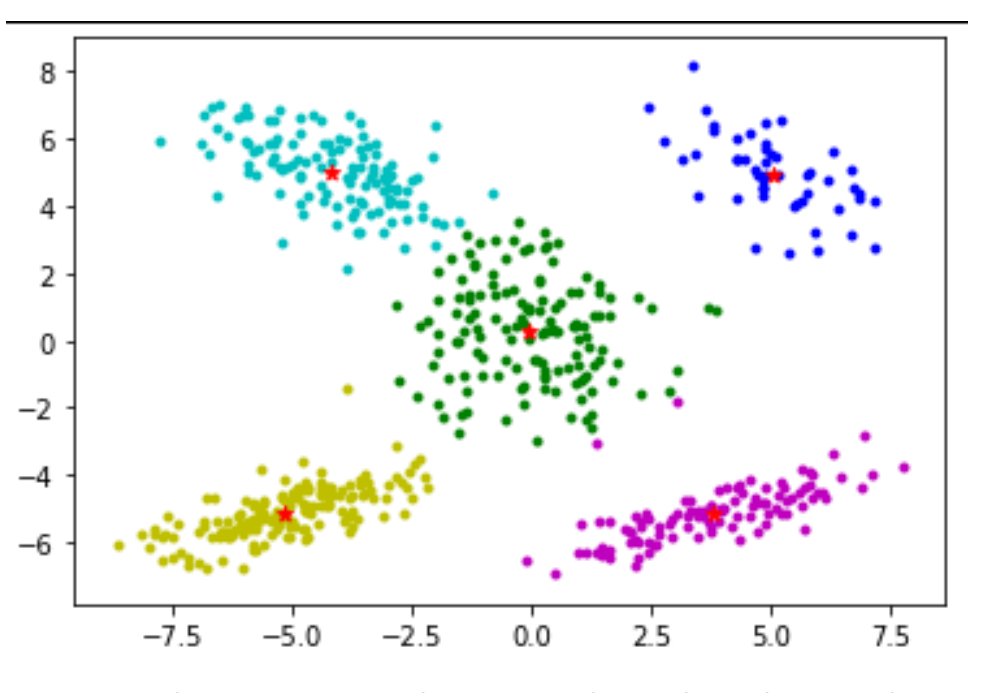


Observando estes pontos percebemos que existem diferentes aglomerados de pontos. Isto leva-nos à ideia de querer obter estes aglomerados, ou seja, distribuir os pontos dados por diferentes grupos.

Vamos imaginar, por exemplo, que estes dados dizem respeito a clientes de uma empresa, onde o eixo dos xx corresponde ao dinheiro gasto no ano fiscal anterior, e o eixo dos yy ao número de transacções efetuadas (sendo o ponto 0,0 a média destas medidas). Seria interessante criar categorias diferentes de clientes a partir dos aglomerados para que a empresa tivesse estratégias de *marketing* diferentes para cada categoria.

Há de imediato o problema de saber quantos aglomerados devemos considerar. Estes aglomerados não existem excepto como expressão do modelo que estamos a tentar criar. Convinha ser o computador a dar-nos uma estimativa do melhor número k de aglomerados (até porque se o problema tiver mais de duas dimensões, deixamos de conseguir visualizar os dados) e, quando soubéssemos esse valor de k, classificar os vários clientes pelas respectivas categorias.

No exemplo da figura anterior, uma solução poderia dizer que existem cinco aglomerados e a distribuição dos pontos pelos aglomerados seria a seguinte:



As estrelas vermelhas correspondem ao centro de massa de cada aglomerado, vamos designá-los por *centróides*.

Vamos desenvolver um programa Python para resolver este tipo de problema.

Information

Flag question

Para experimentarem o vosso código, está disponível a função `criarPontos(ns, seed)` que dada uma lista de inteiros `ns` e uma semente aleatória, gera `len(ns)` aglomerados, onde o aglomerado i possui `ns[i]` pontos. A geração dos dados é aleatória mas baseada no valor da semente dada (ou seja, para o mesmo valor de semente são criados sempre os mesmos pontos).

Os pontos da imagem anterior foram criados pela execução de

```
criarPontos([140, 50, 100, 160, 120], 101)
```

Este código está disponível para *download* na página da disciplina, para desenvolverem o projeto no Spyder ou em Jupyter.

Não é necessário entender o funcionamento desta função, apenas a devem usar para criar conjuntos de pontos de modo a testar a vossa solução.

Question **1**

Correct

Mark 1.50 out of 1.50

Flag question

Defina a função `distancia` que, dados dois pontos 2D, devolve a sua distância euclidiana.

Considere que os pontos são representados por pares de valores `float`.

For example:

Test	Result
`print(distancia((0.0,3.0), (4.0,0.0)))`	5.0

Answer: (penalty regime: 0 %)

Reset answer

```
import math
def distancia(pt1, pt2):
    um=(float(pt1[0]-pt2[0]))
    um2=um*um
    dois=(float(pt1[1]-pt2[1]))
    dois2=dois*dois
    soma= um2 + dois2
    raiz=soma**0.5
    return raiz
```

	Test	Expected	Got	
✓	`print(distancia((0.0,3.0), (4.0,0.0)))`	5.0	5.0	✓

Passed all tests! ✓

Correct

Marks for this submission: 1.50/1.50.

Question **2**

Correct

Mark 2.50 out of 2.50

Flag question

Vamos assumir que já tomámos a decisão de ter k centróides e temos, de momento, uma lista de candidatos.

Se nos derem um ponto `pt` dos dados originais, queremos saber qual o centróide mais perto de `pt` (de acordo com a distância Euclidiana).

Para tal, defina a função `sugerirCentroide(centros, pt)` que recebe uma lista de centróides e um ponto e devolve o índice da lista onde se encontra o centróide mais perto do ponto.

Se dois ou mais centróides estiverem a igual distância, devolver o de menor índice.

For example:

Test	Result
`centroides = [(10,10), (20,10), (0,0)]` `print(sugerirCentroide(centroides, (4,4)))`	2
`centroides = [(10,10), (0,0)]` `print(sugerirCentroide(centroides, (5,5)))`	0

Answer: (penalty regime: 0 %)

Reset answer

```
def sugerirCentroide(centros, pt):
    distancias=[]
    k=len(centros)
    if k==1:
        return 0
    else:
        for m in centros:
            distancias.append(distancia(m, pt))
            minimo=min(distancias)
            indice=distancias.index(minimo)
        return(indice)
```

	Test	Expected	Got	
✓	`centroides = [(10,10), (20,10), (0,0)]` `print(sugerirCentroide(centroides, (4,4)))`	2	2	✓
✓	`centroides = [(10,10), (0,0)]` `print(sugerirCentroide(centroides, (5,5)))`	0	0	✓

Passed all tests! ✓

Correct

Marks for this submission: 2.50/2.50.

Question **3**

Correct

Mark 3.00 out of 3.00

Flag question

Defina a função `encontrarCentroMassa(pts)` que dado uma lista de pontos 2D devolve um par com as coordenadas do centro de massa destes pontos.

Considere todos os pontos com igual 'massa' quando realizar os cálculos.

For example:

Test	Result
`print(encontrarCentroMassa([(0,0),(6,6),(0,6)]))`	(2.0, 4.0)

`lista = []`
`for i in range(100):`
`  pts = criarPontos([100,150], i)`
`  lista.extend(encontrarCentroMassa(pts))`
`print(round(sum(lista),3))`
	598.486

Answer: (penalty regime: 0 %)

Reset answer

```
def encontrarCentroMassa(pts):
    k=len(pts)
    somax=sum([pt[0] for pt in pts])
    somay=sum([pt[1] for pt in pts])
    x=somax/k
    y=somay/k
    return (x,y)
```

	Test	Expected	Got	
✓	`print(encontrarCentroMassa([(0,0),(6,6),(0,6)]))`	(2.0, 4.0)	(2.0, 4.0)	✓

✓	`lista = []`
	`for i in range(100):`
	`  pts = criarPontos([100,150], i)`
	`  lista.extend(encontrarCentroMassa(pts))`
	`print(round(sum(lista),3))`
		598.486	598.486	✓

Passed all tests! ✓

Correct

Marks for this submission: 3.00/3.00.

Information

Flag question

Agora precisamos descrever um algoritmo para criar os aglomerados.

Assuma que sabemos o número k de aglomerados e que nos dão a lista de pontos do problema.

O algoritmo deve ter os seguintes passos:

1. Definir os primeiros k pontos da lista como centróides (inicialização)
2. Repetir os seguintes passos:
    1. Associar cada ponto da lista ao centróide mais perto, isto vai produzir k aglomerados à volta dos centróides
    2. Uma vez criados os aglomerados, calcular os respetivos centros de massa; estes serão os novos centróides

O ciclo definido pelos passos 2.1-2.2 vai ter de terminar. Há duas condições possíveis de paragem:

- Se os novos centróides estão muito perto dos centróides anteriores (ou seja, o algoritmo convergiu). Para tal vamos somar as distâncias da cada novo centróide com a sua versão anterior. Esta soma terá de ser menor que um valor de tolerância dado;
- Se executámos o ciclo demasiadas vezes (ou o algoritmo não convergiu ou está muito lento).

Quando o ciclo terminar devolvemos uma lista com a versão atual dos centróides.

Question **4**

Correct

Mark 7.00 out of 7.00

Flag question

De acordo com o algoritmo descrito, defina a função `aglomerar(k, pts, tol, maxIter)` que recebe o número de aglomerados, a lista de pontos, a tolerância máxima que define a convergência do algoritmo e o número máximo de iterações permitidas para o cálculo dos centróides.

A função deve devolver uma lista de pares contendo os centróides propostos.

For example:

Test	Result

`pts = criarPontos([140, 50, 100, 160, 120], 101)`
`centros = aglomerar(1, pts)`
`print([ (round(x,2),round(y,2)) for (x,y) in centros ])`
	[(−1.23, −0.8)]

`pts = criarPontos([140, 50, 100, 160, 120], 101)`
`centros = aglomerar(2, pts)`
`centros.sort()`
`print([ (round(x,2),round(y,2)) for (x,y) in centros ])`
	[(−4.27, −0.61), (2.9, −1.0

Answer: (penalty regime: 0 %)

Reset answer

```
import doctest
def aglomerar(k, pts, tol=0.001, maxIter=500):
    """
    Função que calcula aglomerados, devolvendo uma lista de paress dos c
    k=numero de aglomerados
    requires : type(k) is int
    requires : k > 0
    """
    aglomerados=[]
    centroides=[]
    for valor in range(0,k): #1.Definir os primeiros k pontos da lista c
        aglomerados.append([]) #consoante o k faz k vezes [] onde esta o
        inicializa=pts[valor]#inicializaçao
        centroides.append(inicializa)
```

	Test	Expected

✓	`pts = criarPontos([140, 50, 100, 160, 120], 101)`
	`centros = aglomerar(1, pts)`
	`print([ (round(x,2),round(y,2)) for (x,y) in centros ])`
		[(−1.23, −0.8)]

✓	`pts = criarPontos([140, 50, 100, 160, 120], 101)`
	`centros = aglomerar(2, pts)`
	`centros.sort()`
	`print([ (round(x,2),round(y,2)) for (x,y) in centros ])`
		[(−4.27, −0.61), (2.9,

Passed all tests! ✓

Correct

Marks for this submission: 7.00/7.00.

Information

Flag question

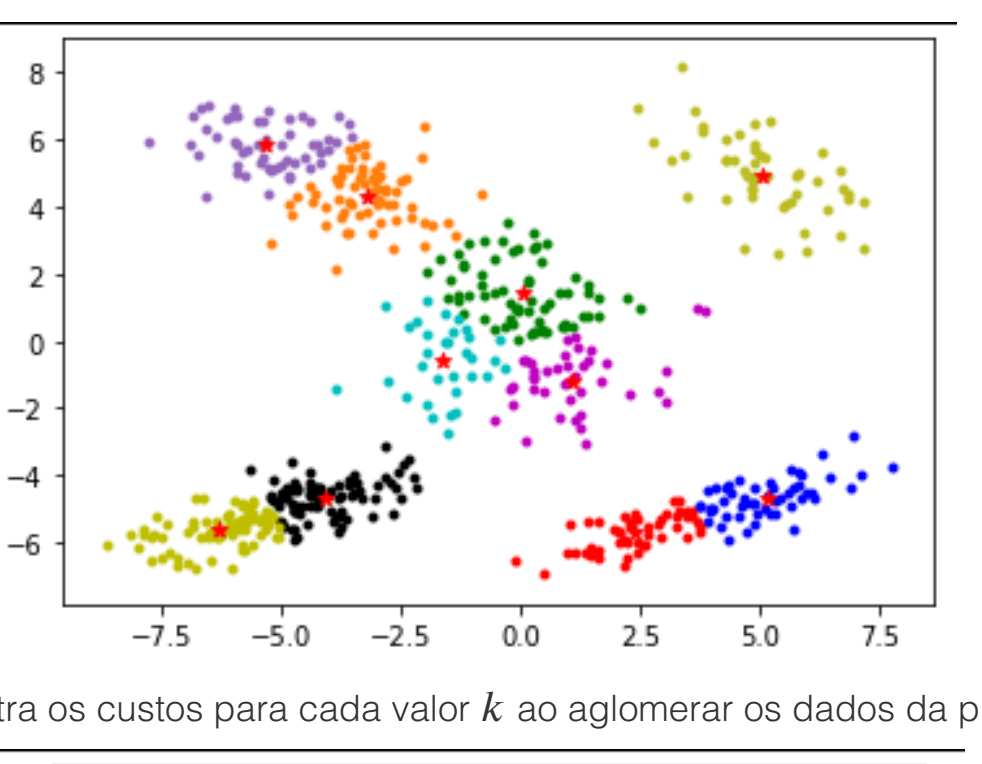
Já temos quase todas as peças para efetuar a aglomeração. O que nos falta é uma função que nos sugira o número k de aglomerados. Até agora temos assumido que este número existe. Nas próximas questões, vamos resolver este último passo.

Uma primeira solução seria variar k entre várias propostas (por exemplo, entre 2 e 10). Depois, para cada k, calcularíamos os centróides com a função anterior e calculávamos a soma dos quadrados das distâncias de todos os pontos aos seus respetivos centróides. Esta soma de distâncias seria interpretada como um *custo* de cada proposta. Quanto menor o custo, melhor.

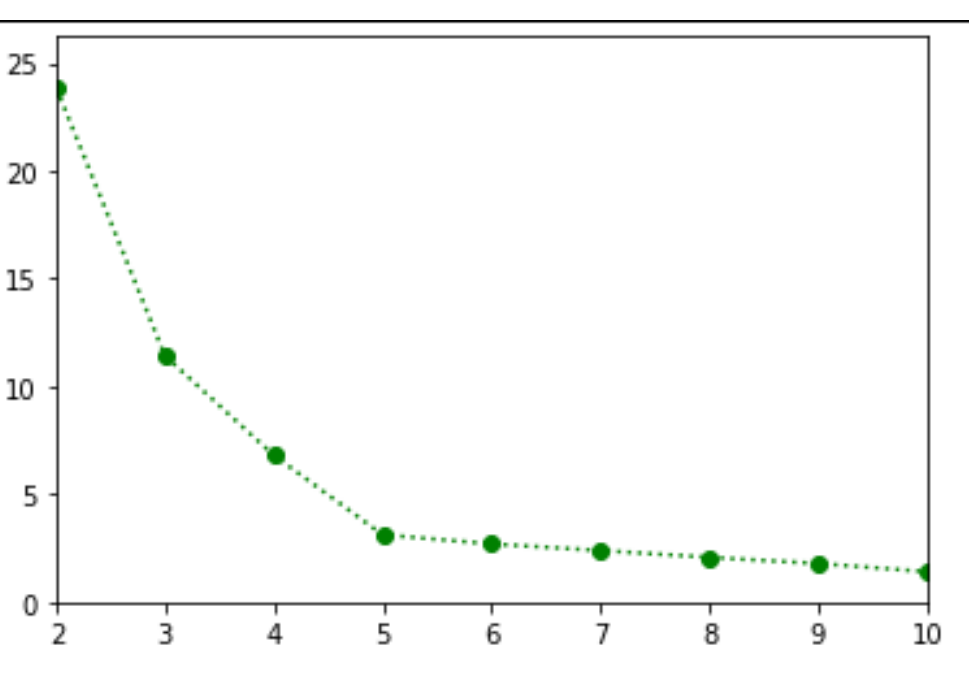
> nota: usamos os quadrados das distâncias para reforçar o custo de pontos que estejam muito longe do respetivo centróide.

A opção k escolhida seria, entre as opções testadas, aquela com menor custo.

Esta solução é fácil de implementar mas tem um problema: quantos mais aglomerados melhor (da perspetiva do custo). Se houver mais aglomerados, as somas das distâncias vão ser cada vez mais pequenas. Vejamos um exemplo dos dados iniciais com 10 aglomerados:



O gráfico seguinte mostra os custos para cada valor k ao aglomerar os dados da primeira figura:

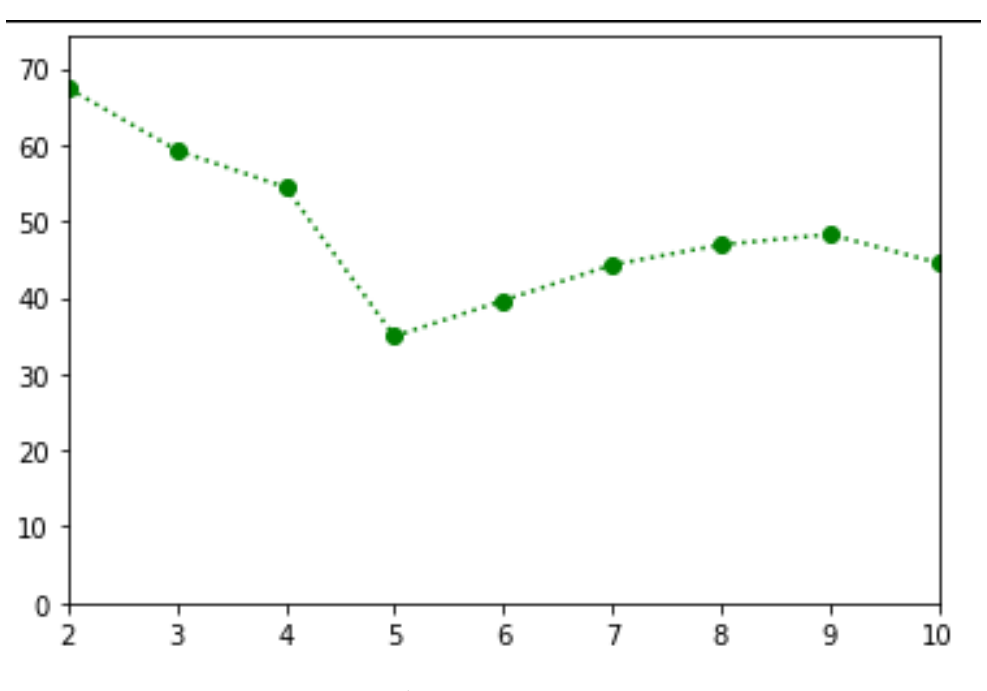


Isto ocorre porque um modelo mais complexo (muitos aglomerados) tem muito mais por onde se adaptar aos dados que um modelo menos complexo (poucos aglomerados). Neste caso, o algoritmo está a criar demasiados grupos entre os dados, o que não nos interessa.

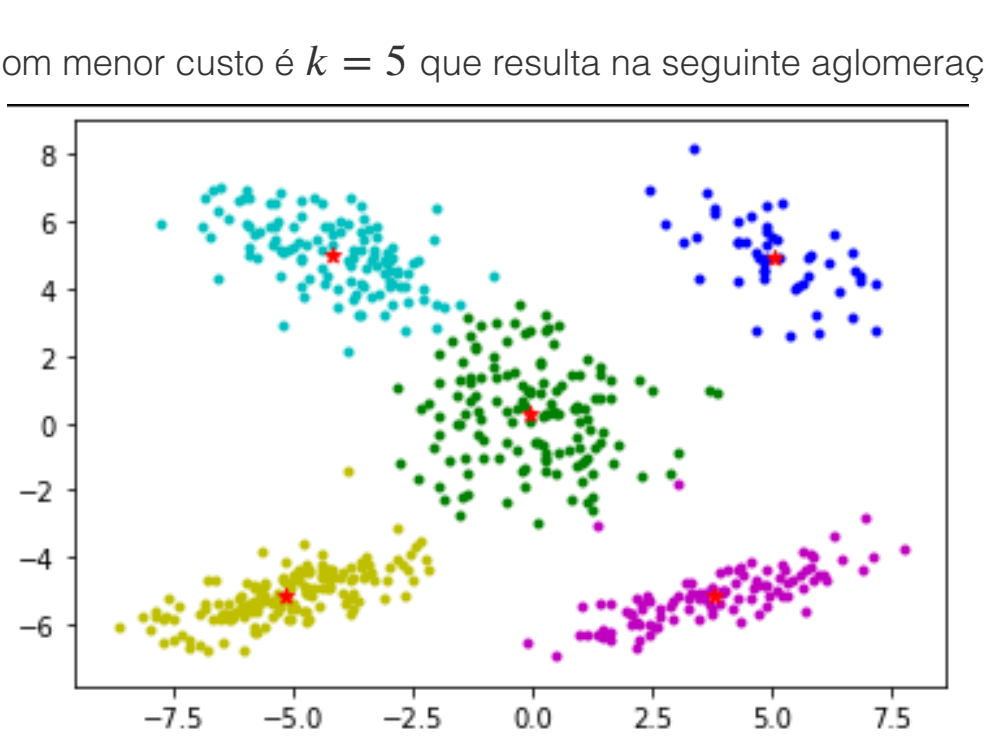
Então o que fazer? Bem, podemos penalizar a complexidade do modelo. Ou seja, juntar ao cálculo do custo um factor que considere o valor de k. Assim, um modelo mais complexo tem de ser mesmo muito melhor que um modelo mais simples, para conseguir recuperar da penalização.

No nosso caso vamos multiplicar a soma das distâncias pelo factor $k^{1.5}$, que é um compromisso entre uma penalização linear (multiplicar por k) e uma penalização quadrática (multiplicar por k^2).

Com este critério, o gráfico dos custos fica o seguinte:



Ou seja, o valor de k com menor custo é $k = 5$ que resulta na seguinte aglomeração:



Muito melhor :-)

Question **5**

Correct

Mark 3.00 out of 3.00

Flag question

De acordo com o texto anterior, defina a função `custear(centros, pts)` que recebe uma lista de centróides e a lista com os pontos originais, e devolve a soma dos quadrados das distâncias entre cada ponto e o centróide mais próximo.

Repare que ainda não estamos a incorporar a penalização.

For example:

Test	Result

`pts = criarPontos([140, 50, 100, 160, 120], 101)`
`centros = aglomerar(2, pts)`
`print(round(custear(centros, pts),3))`
	13589.5

`pts = criarPontos([140, 50, 100, 160, 120], 101)`
`centros = aglomerar(4, pts)`
`print(round(custear(centros, pts),3))`
	3878.02

Answer: (penalty regime: 0 %)

Reset answer

```
def custear(centros, pts):
    tamanho=len(pts)
    final=[]
    for j in range(tamanho):
        minimoIndice=sugerirCentroide(centros,pts[j])
        distanciaMenor=distancia(pts[j], centros[minimoIndice])
        final.append(distanciaMenor**2)
        soma=sum(final)
    return soma

```

	Test	Expected	Got	

✓	`pts = criarPontos([140, 50, 100, 160, 120], 101)`
	`centros = aglomerar(2, pts)`
	`print(round(custear(centros, pts),3))`
		13589.5	13589.5	✓

✓	`pts = criarPontos([140, 50, 100, 160, 120], 101)`
	`centros = aglomerar(4, pts)`
	`print(round(custear(centros, pts),3))`
		3878.02	3878.02	✓

Passed all tests! ✓

Correct

Marks for this submission: 3.00/3.00.

Question **6**

Correct

Mark 3.00 out of 3.00

Flag question

Defina a função `sugerirK(pts, minK, maxK)` que recebe a lista dos pontos iniciais, e dois inteiros que definem o intervalo de procura do valor k.

A função deve devolver o k que minimiza o custo definido no texto acima.

For example:

Test	Result

`pts = criarPontos([140, 50, 100, 160, 120], 101)`
`k = sugerirK(pts)`
`print(k)`
	5

Answer: (penalty regime: 0 %)

Reset answer

```
def sugerirK(pts, minK=2, maxK=10):
  """
  requires: minK >= 2
  requires: minK < maxK
  """
  lista1=[]
  lista2=[]
  for k in range(minK, maxK):
      centros=aglomerar(k,pts)
      distancias=custear(centros,pts)
      distancias2=int(distancias *(k**1.5))
      lista2.append(distancias2)
      minimo=min(lista2)
      indice=lista2.index(minimo)
```

	Test	Expected	Got	

✓	`pts = criarPontos([140, 50, 100, 160, 120], 101)`
	`k = sugerirK(pts)`
	`print(k)`
		5	5	✓

Passed all tests! ✓

Correct

Marks for this submission: 3.00/3.00.

Finish review