

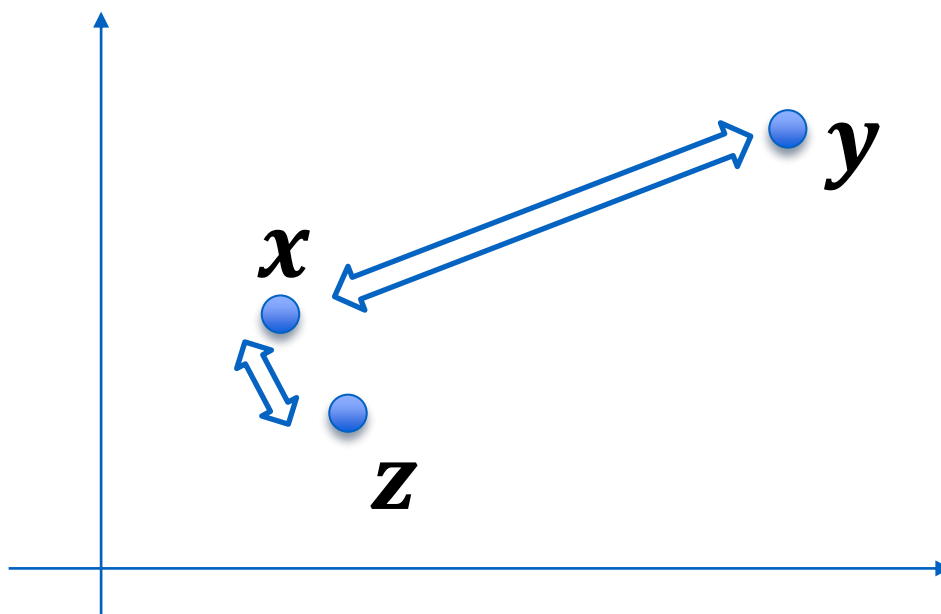
# 「距離とクラスタリング」

九州大学 大学院システム情報科学研究所  
情報知能工学部門  
データサイエンス実践特別講座  
末廣大貴, Diego Thomas, 正井克俊

# データ（ベクトル）間の距離、類似度

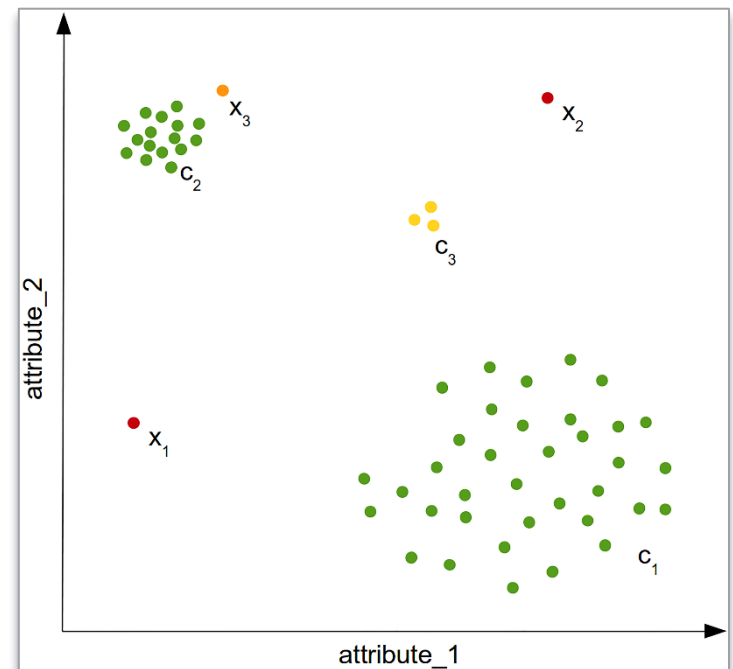
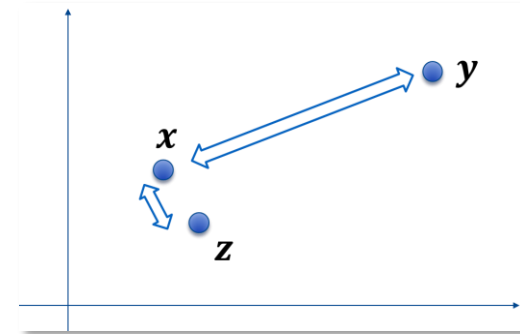
# データ解析における「距離」とは？

- データ解析における「距離」
  - 要するにデータ間の類似度（似てない具合）
  - 距離が小さい2データは「似ている」
  - 単位がある場合もない場合も



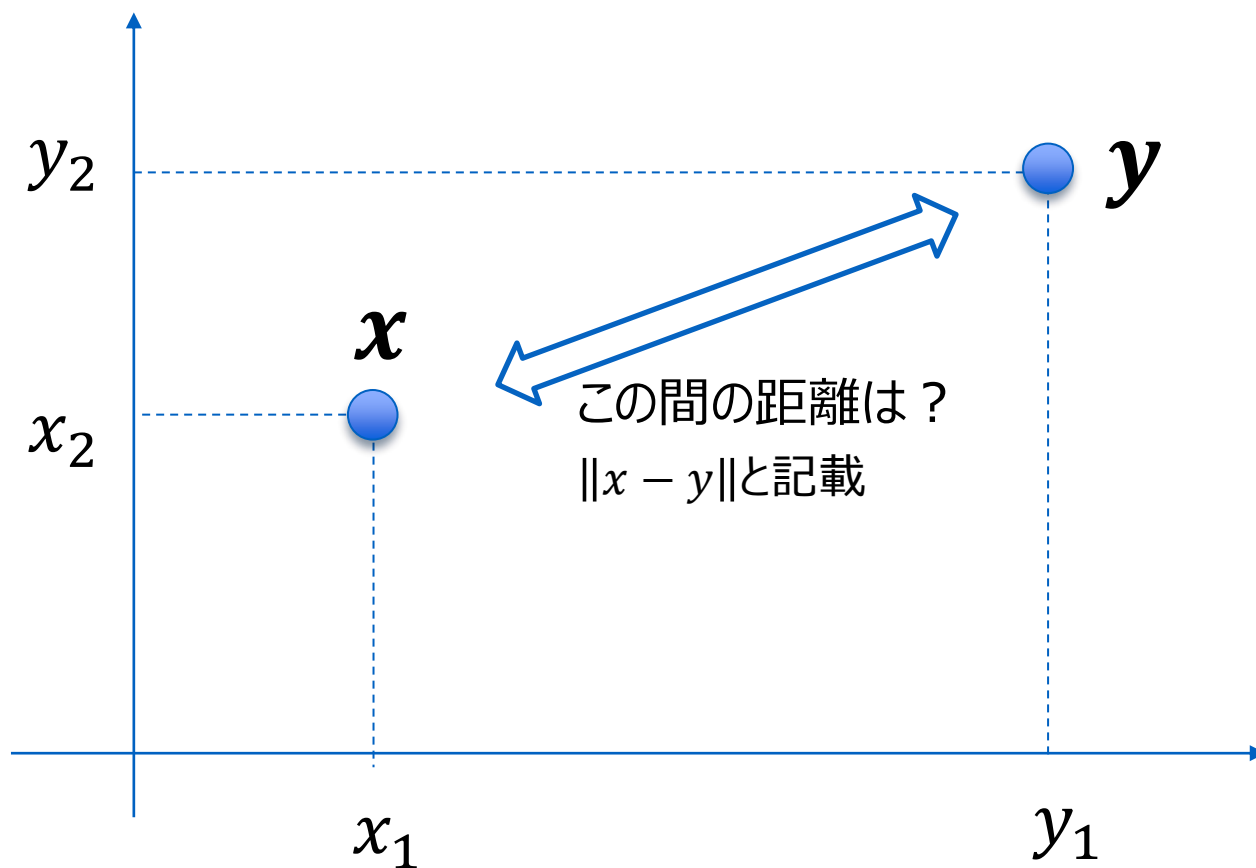
# 距離がわかると何に使えるか？ 実は超便利！

- データ間の「類似度」を定量的に比較できる
  - 「xとyは全然違う/結構似ている」「xとyは28ぐらい違う」
  - 「xにとっては、yよりもzのほうが似ている」
- データ集合のグルーピングができる
  - 似たものとおしでグループを作る
- データの異常度が測れる
  - 他に似たデータがたくさんあれば正常、一つもなければ異常



# ユークリッド距離 (1)

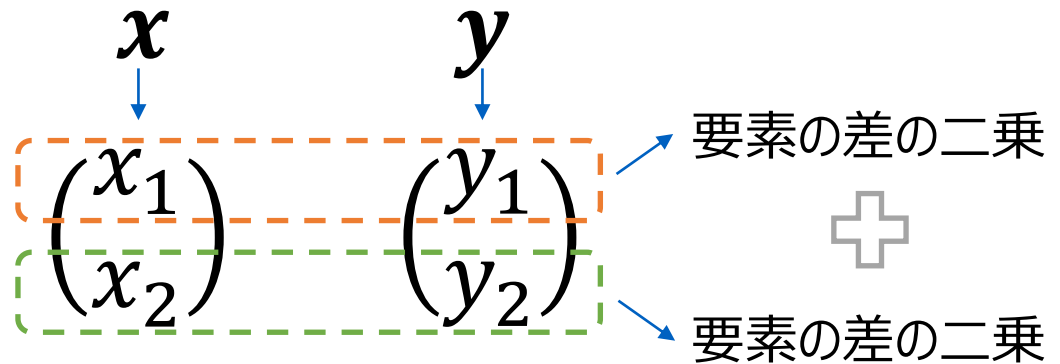
- 地図上の2点  $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ ,  $y = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$



# ユークリッド距離 (2)

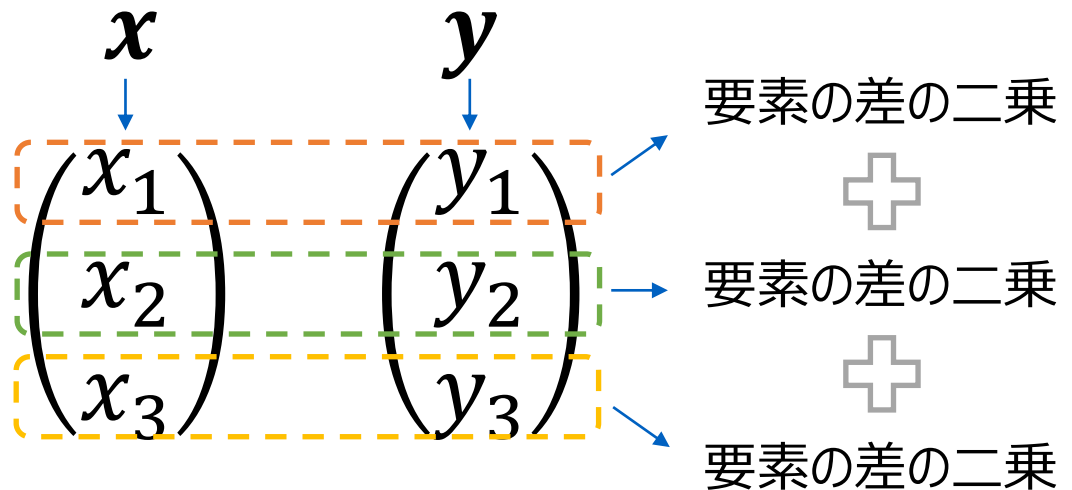
## ● 2次元の場合

$x$ と $y$ の距離の二乗  $\equiv$   
 $(x_1 - y_1)^2 + (x_2 - y_2)^2$



## ● 3次元の場合

$x$ と $y$ の距離の二乗  $\equiv$   
 $(x_1 - y_1)^2 + (x_2 - y_2)^2 + (x_3 - y_3)^2$



# ユークリッド距離 (3)

- $D$ 次元の場合

$x$ と $y$ の距離の二乗  $\equiv$

$$\begin{pmatrix} x_1 \\ \vdots \\ x_D \end{pmatrix} \quad \begin{pmatrix} y_1 \\ \vdots \\ y_D \end{pmatrix}$$

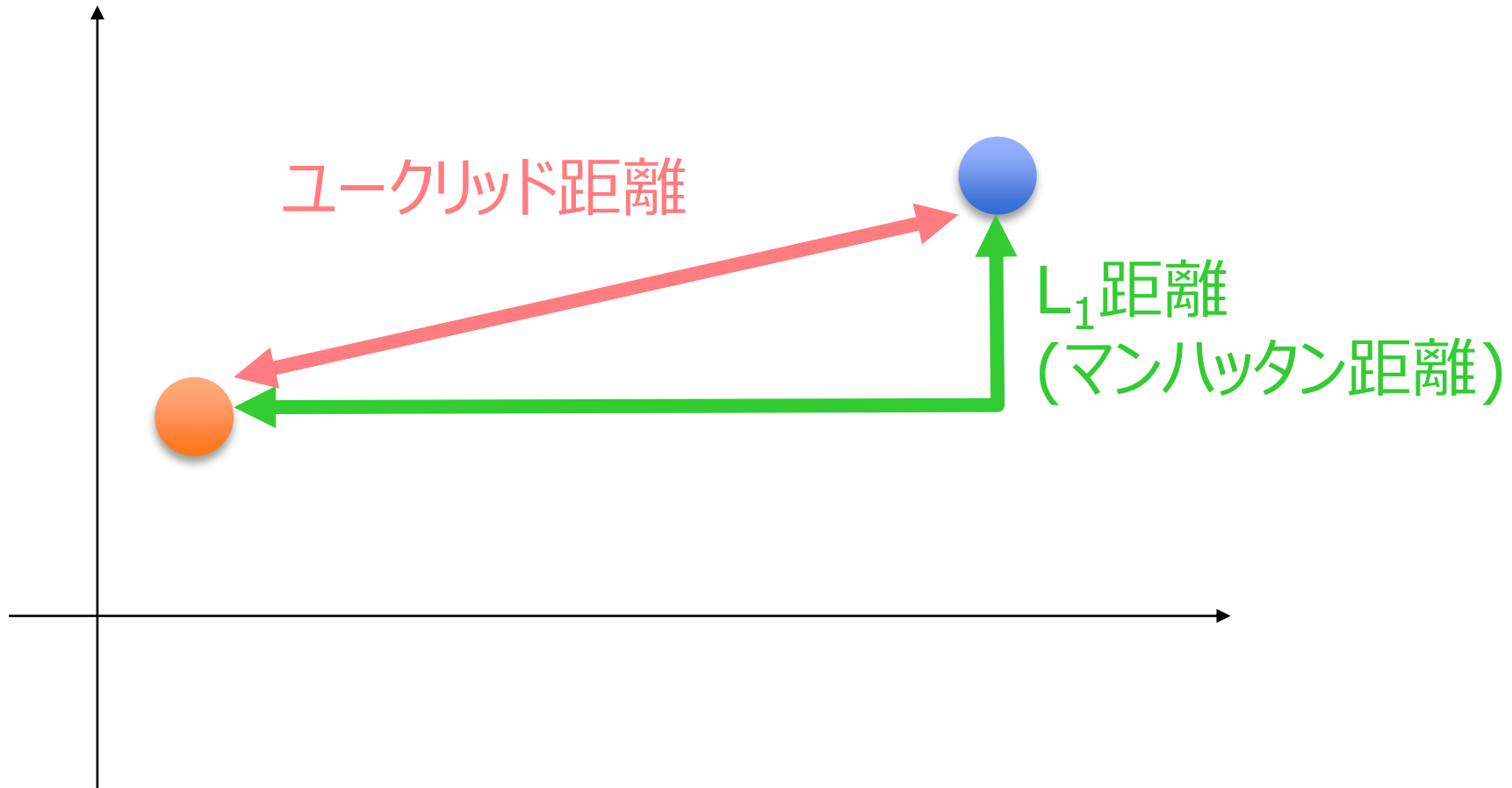
要素の差の二乗  
+  
+  
+  
要素の差の二乗

というわけで、何次元ベクトルでも距離は計算可能

もちろん1次元ベクトル(数値)間の距離も計算可能

$$(x_1 - y_1)^2$$

# マンハッタン距離



# マンハッタン？

- 斜めには行けない街
  - 平安京距離
  - 平城京距離
  - 札幌距離
- でもいいかも
- 「市街地距離」と呼ばれることも



# マンハッタン距離:プログラミング (7)

● N次元の場合

$$\mathbf{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_N \end{pmatrix}, \quad \mathbf{y} = \begin{pmatrix} y_1 \\ \vdots \\ y_N \end{pmatrix}$$

$$\mathbf{x} \text{ と } \mathbf{y} \text{ の距離} = \underbrace{|x_1 - y_1|} + \cdots + \underbrace{|x_N - y_N|}$$

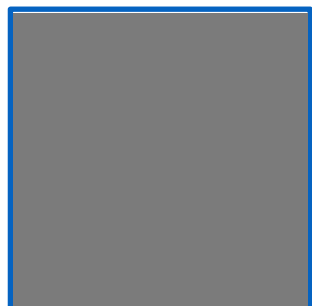
各要素の差の絶対値の和

`np.linalg.norm(x-y, ord=1)` で計算できる

※ ユークリッド距離が「L2」ノルム、  
マンハッタン距離が「L1」ノルムとも呼ばれる

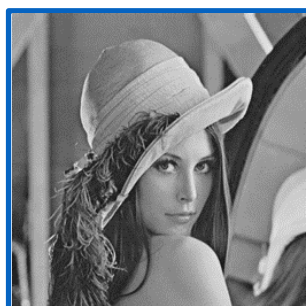
# ユークリッド距離で測る類似度

256 × 256



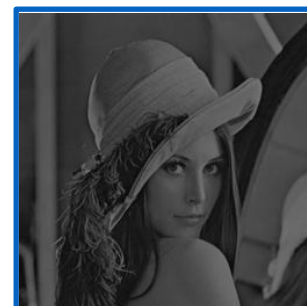
65536次元  
ベクトル  $z$

256 × 256

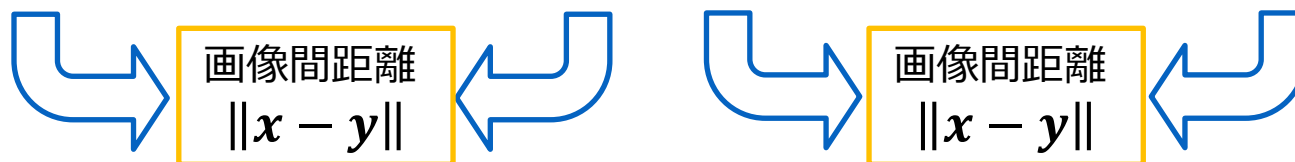


65536次元  
ベクトル  $x$

256 × 256



65536次元  
ベクトル  $y$



12249.03

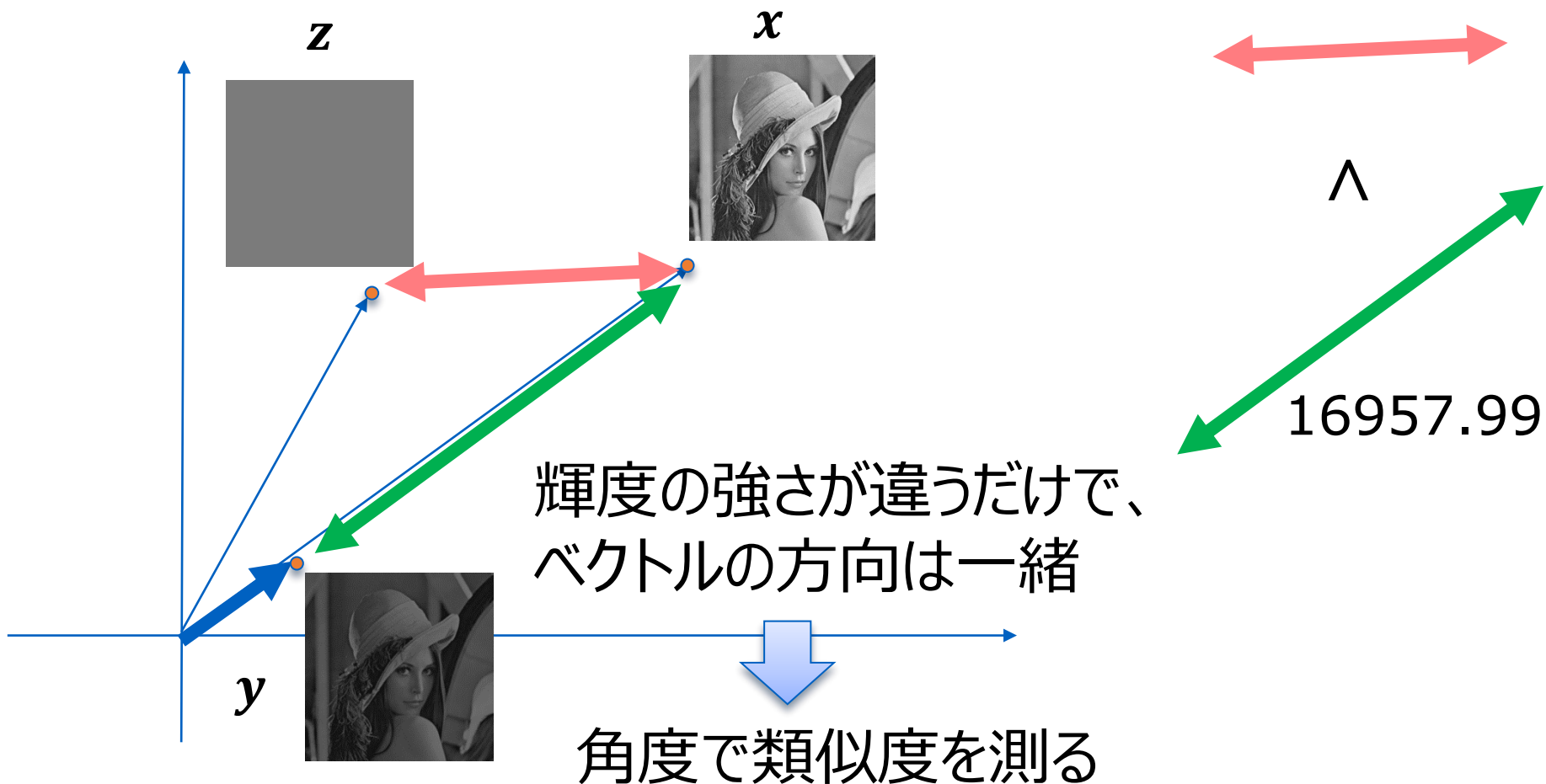
<

16957.99

人の目には  $z$  より  $y$  の方が  $x$  に類似しているが、  
ユークリッド距離だと、 $z$  の方が  $x$  からの距離が近い

# 他の類似度

イメージ図



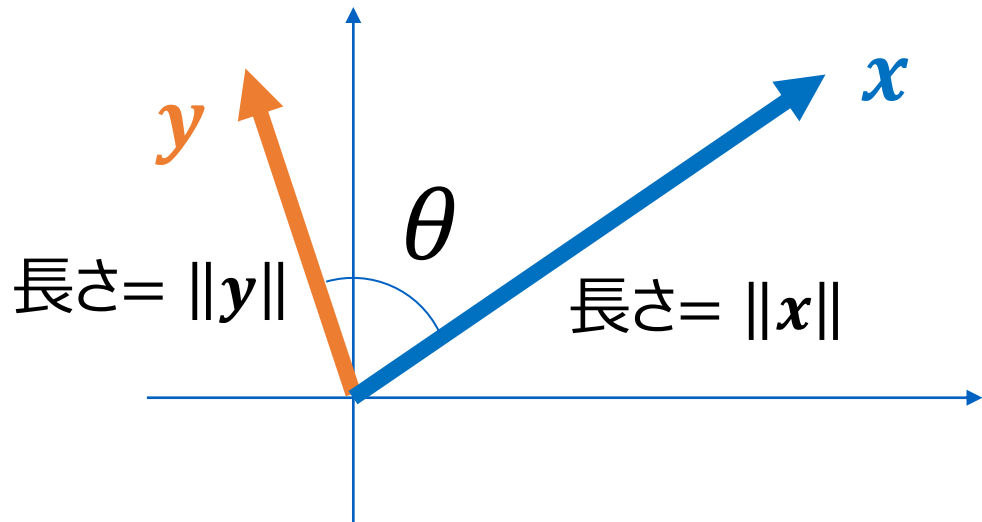
# 内積

- 内積は角度を考慮した類似度

$$\mathbf{x} \cdot \mathbf{y} = \|\mathbf{x}\| \|\mathbf{y}\| \cos \theta$$

$$\cos \theta = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|}$$

各成分2乗して足して  
ルートとったやつ



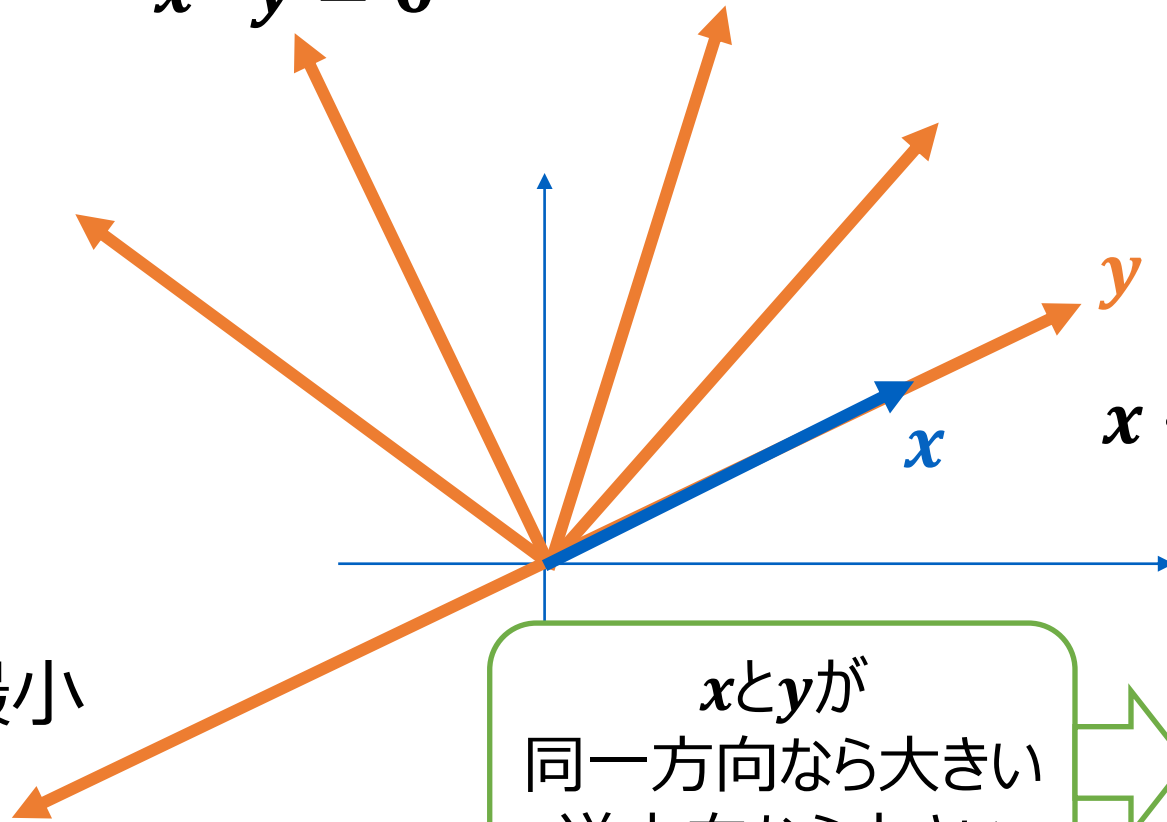
# 内積

- 一定の大きさのベクトル $y$ を回転させながら $x$ と内積を取ると...

$$x \cdot y = 0$$

$$x \cdot y \rightarrow \text{最小}$$

$$x \cdot y \rightarrow \text{最大}$$



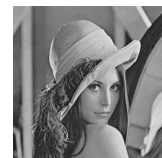
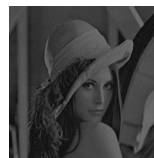
$x$ と $y$ が  
同一方向なら大きい  
逆方向なら小さい

$x$ と $y$ の  
類似度に  
使えそう!

# 正規化相関（コサイン類似度）

- 正規化相関

$$\cos \theta = \frac{x \cdot y}{\|x\| \|y\|}$$



```
import numpy as np
```

```
# xとyの定義
```

```
# 1. ベクトルの定義
```

```
x = np.array([60, 180]) # ベクトルx
```

```
y = np.array([60, 150]) # ベクトルy
```

```
# 2. コサイン類似度の計算
```

```
dot_product = np.dot(x, y) # 内積 x・y
```

```
norm_x = np.linalg.norm(x) # ||x|| (ベクトルxの大きさ)
```

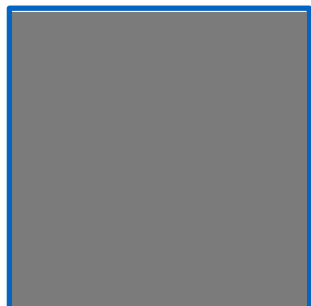
```
norm_y = np.linalg.norm(y) # ||y|| (ベクトルyの大きさ)
```

```
# コサイン類似度
```

```
cosine_similarity = dot_product / (norm_x * norm_y)
```

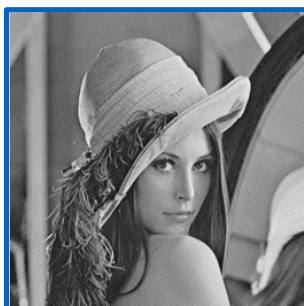
# 正規化相関で測る類似度

256 × 256



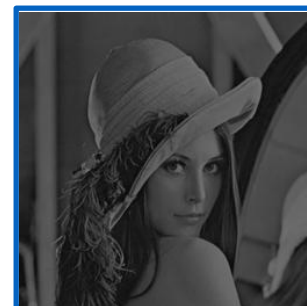
65536次元  
ベクトル  $z$

256 × 256

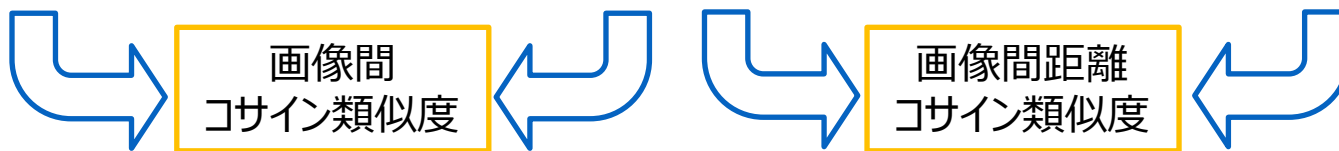


65536次元  
ベクトル  $x$

256 × 256



65536次元  
ベクトル  $y$



0.932

<

1.0

# まとめ: numpy を使えば簡単

```
a=np.array([1,2,3,4,5])  
b=np.array([2,2,3,3,4])
```

- ユークリッド距離

```
np.linalg.norm(a-b)  
1.7320508075688772
```

- マンハッタン距離

```
np.linalg.norm(a-b, ord=1)  
3.0
```

- 内積

```
np.dot(a,b)  
47
```

- コサイン

```
np.dot(a,b)/(np.linalg.norm(a)*np.linalg.norm(b))  
0.9778941948273628
```

練習 1 : 2つのデータ間の各類似度 :  
ユークリッド距離、マンハッタン距離、内積、コサイン類似度  
を求めよう

$$\mathbf{x} = \begin{pmatrix} 3 \\ 5 \end{pmatrix}, \mathbf{y} = \begin{pmatrix} 6 \\ 1 \end{pmatrix} \text{ のとき 各類似度は？}$$

$$\mathbf{x} = \begin{pmatrix} 3 \\ 5 \\ 2 \end{pmatrix}, \mathbf{y} = \begin{pmatrix} 6 \\ 1 \\ 2 \end{pmatrix} \text{ のとき 各類似度は？}$$

# 距離を使ったちょっと高度な可視化（1）

- csvファイル" height\_weight.csv"を読み込み、「横軸：身長、縦軸：体重」としてプロットして、分布の形状を確かめたい。target\_pointを別の色で表示してみたい。

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
# 1. データの読み込み
data_dir = './Data/'
df = pd.read_csv(data_dir + "height_weight.csv") # CSVファイルを読み込む
df = df.rename(columns={"Ht": "Height", "Wt": "Weight"}) # 列名を直感的に変更
data_array = np.array(df) # DataFrameをNumPy配列に変換
```

こうすればpandasで読み込んだデータが一発でnumpyのデータセット（複数のベクトル）にできます

```
# -----  
# 2. 基準点 (例: 身長180cm, 体重75kg)  
# -----  
target_point = np.array([180, 75]) # 比較の基準となる点を定義  
  
# -----  
# 3. 散布図で分布を可視化  
# -----  
# 全データを散布図で表示 (青丸)  
df.plot(kind='scatter', x='Height', y='Weight', c='blue', marker='o', s=30)  
# 基準点を赤い+で強調表示  
plt.scatter(target_point[0], target_point[1],  
c='red', marker='+', s=100, label='Target') # cは色の指定、markerは形の指定、sは大きさの指定  
plt.xlabel("Height (cm)") # x軸ラベル  
plt.ylabel("Weight (kg)") # y軸ラベル  
plt.title("Height-Weight Distribution with Distance from Target") # タイトル  
plt.legend() # 凡例を表示  
plt.show()
```



## 距離を使ったちょっと高度な可視化（2）

- target\_point=(180,75)からの各点へのユークリッド距離をそれぞれ算出し、最も距離が遠い順に並び替えて、距離に応じて色を変えて表示

```
# 4. 各データ点と基準点との距離を計算
```

```
distances = []
```

```
for row in data_array: # row[2] = Height, row[3] = Weight
```

```
    distance = np.linalg.norm(target_point - row[[2, 3]])
```

```
    distances.append(distance) # ユークリッド距離を計算
```

```
# 5. 距離が大きい順に並べ替え
```

```
sorted_indices = np.argsort(distances)[::-1]
```

```
# 大きい順に並べ替えたインデックス、[::-1]は逆順にするスライス
```

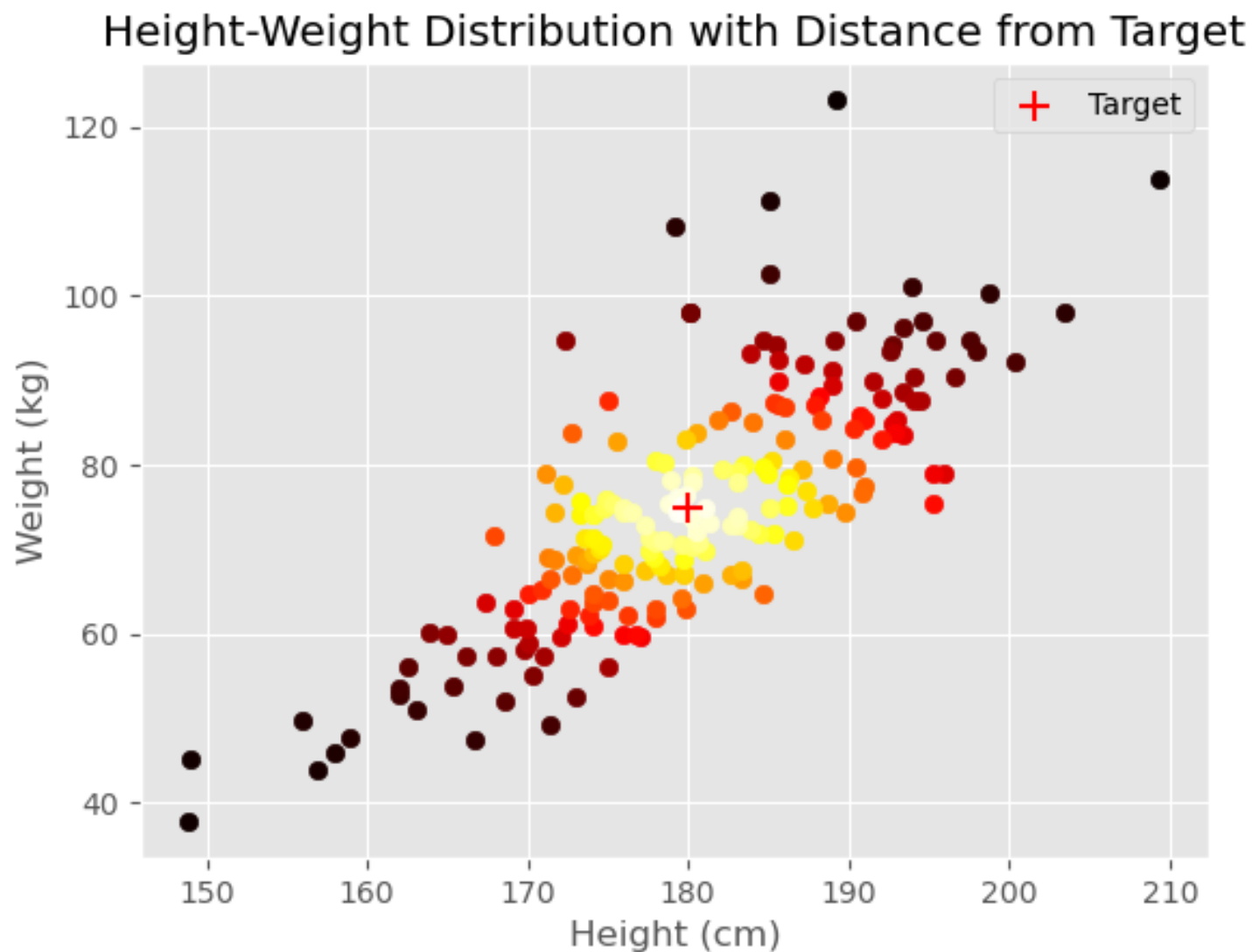
例 : `np.argsort([2,1,3])=[1,0,2]`

# 距離を使ったちょっと高度な可視化（2）

## ● 距離に応じて色を変えて表示

```
# -----  
# 6. 距離に応じて色を変えて表示  
# -----  
# enumerate(sorted_indices) を使うことで  
# count = ループが何番目か (0, 1, 2, ...)  
# idx = sorted_indices の中身 (データの行番号)  
# を同時に取り出せる  
for count, idx in enumerate(sorted_indices):  
    row = data_array[idx]  
    # 距離の順位に応じて色を決定 (ホットカラーマップを使用)  
    color = cm.hot(count / len(data_array))  
    plt.scatter(row[2], row[3], color=color)  
  
# 基準点をもう一度表示 (上書きして見やすく)  
plt.scatter(target_point[0], target_point[1],  
            c='red', marker='+', s=100, label = "Target")
```

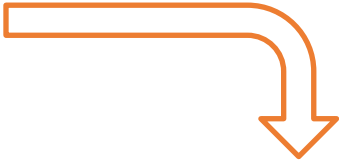
# 可視化結果



# K-Means

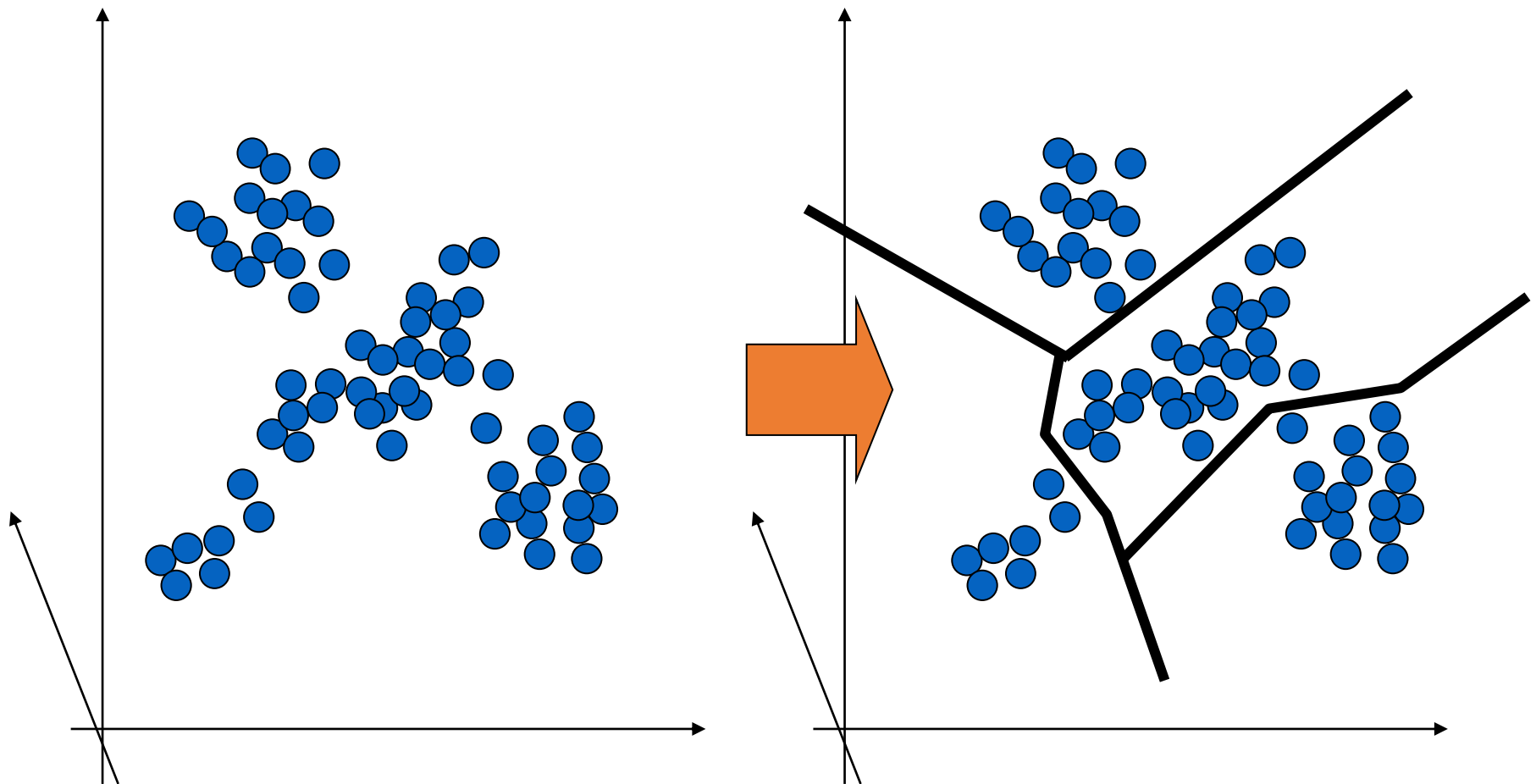
データのクラスタリング(分類)

# データ集合をグルーピングする

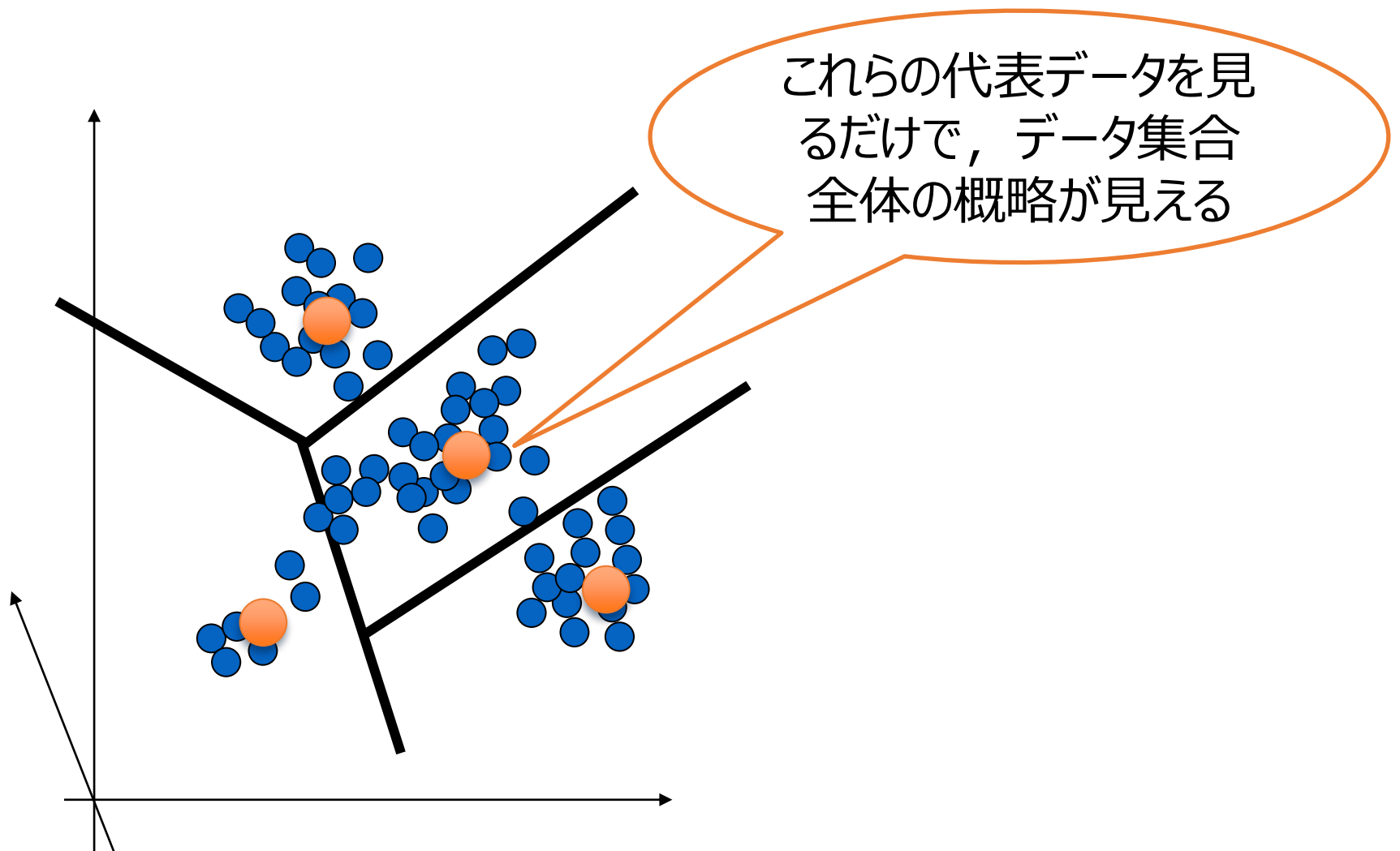
- 我々は「距離」や「類似度」を手に入れた 
- 結果, 「与えられたデータ集合」を「それぞれ似たデータからなる幾つかのグループに分ける」ことが可能に！

クラスタリング(clustering) =  
データの集合をいくつかの部分集合に分割する(グルーピング)

- 各部分集合 = 「クラスタ」と呼ばれる

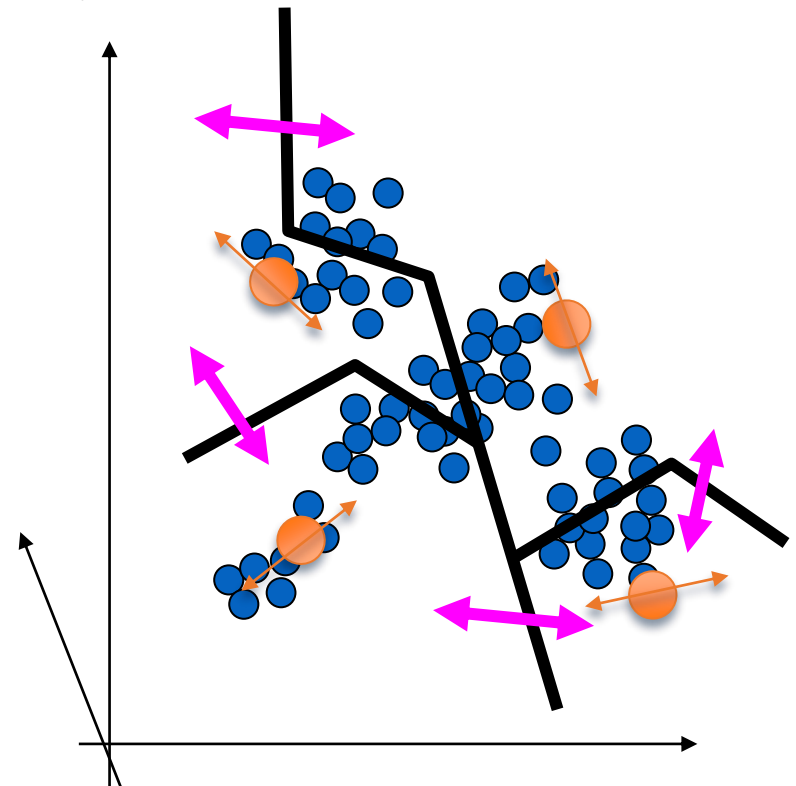
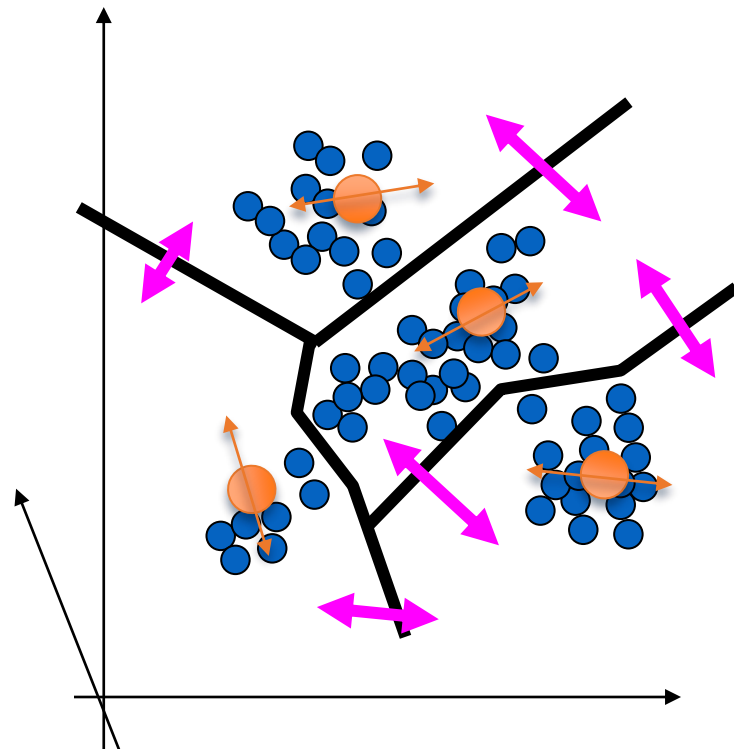


# 各クラスタから代表的なデータを選ぶと...



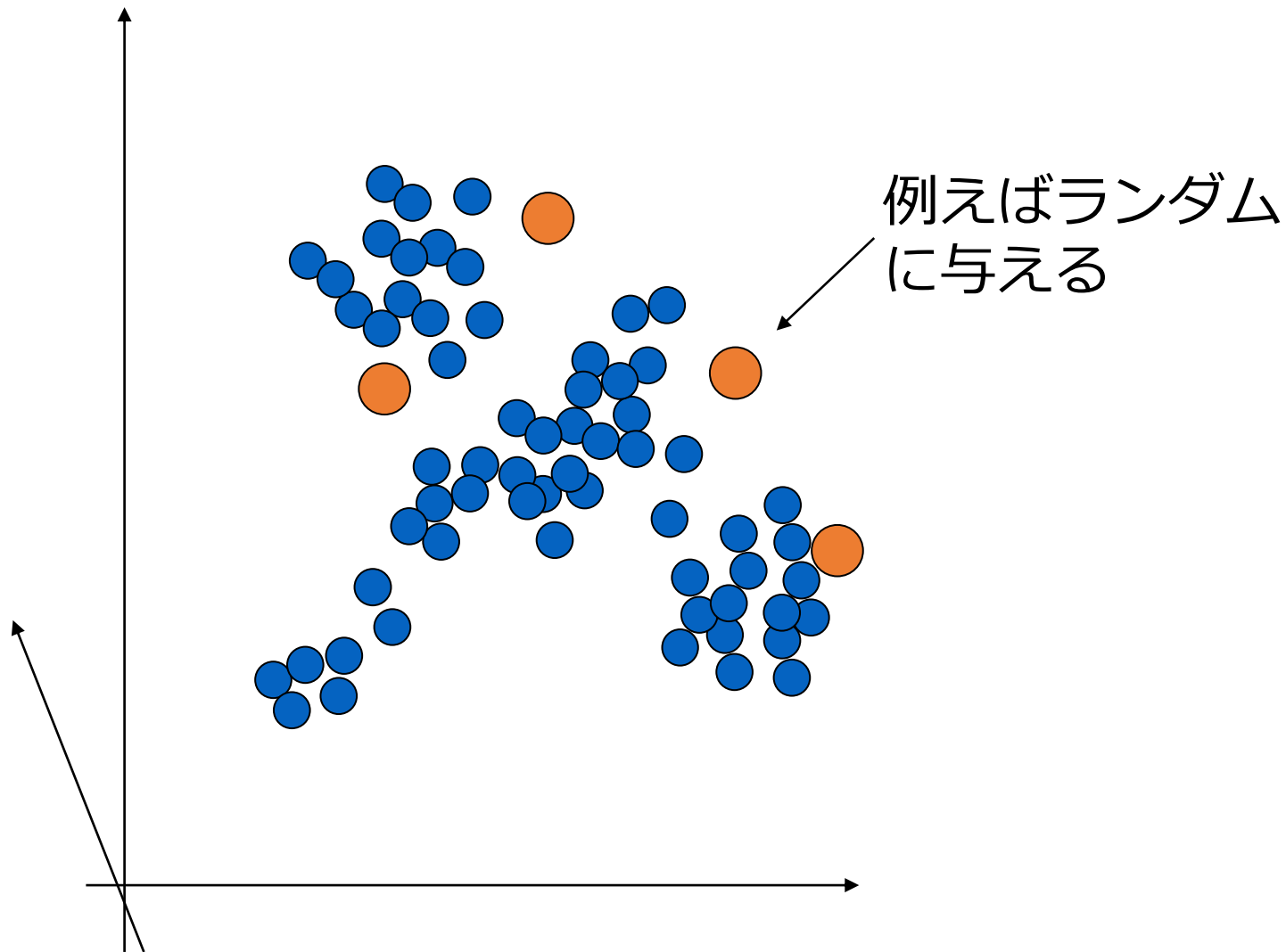
# どういふ分割がよいのか？

- $N$ 個のデータを  $K$ 個に分割する方法はおよそ  $K^N$ 通り
- 100個のデータを10分割→およそ  $10^{100}$ 通り



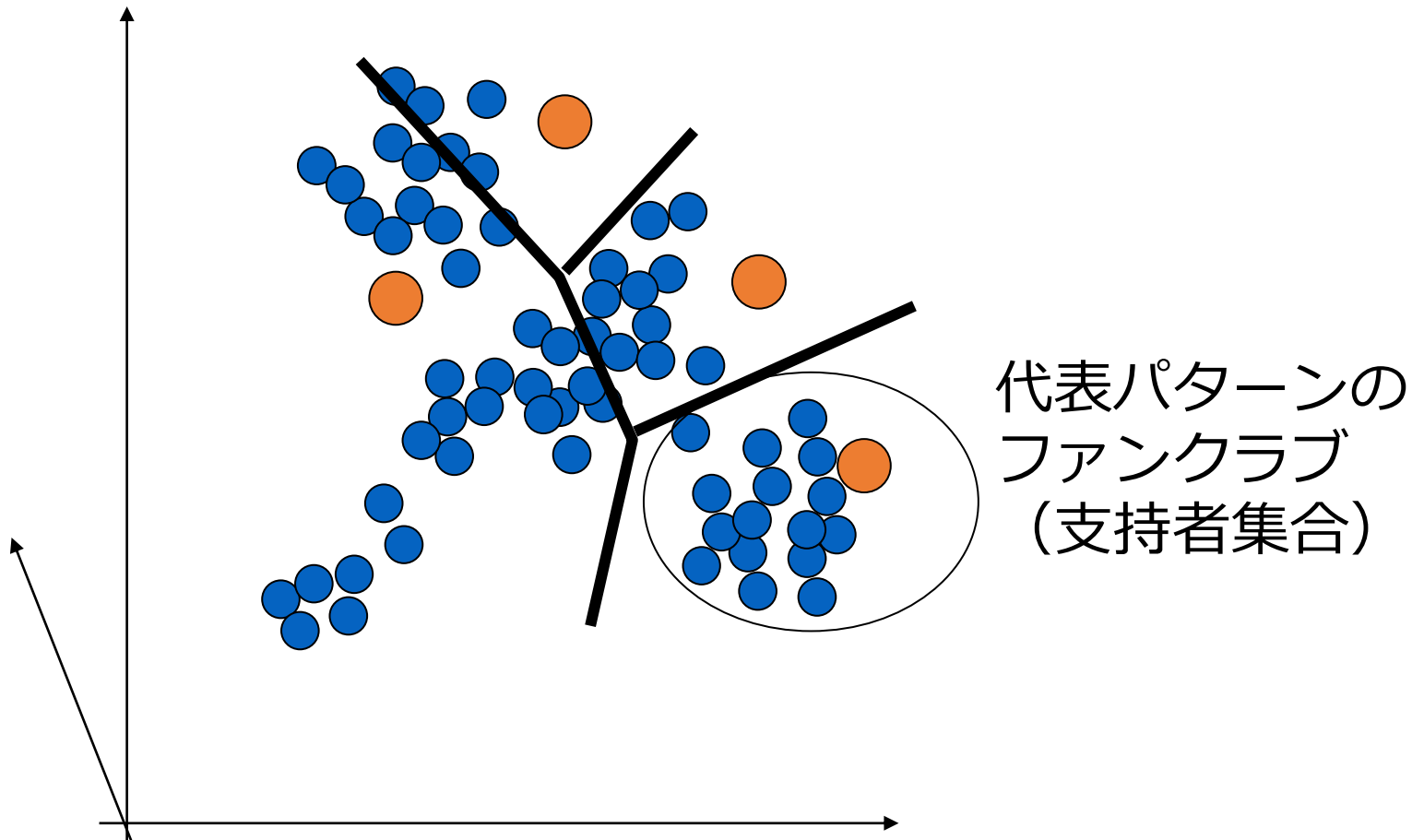
- 近くにあるデータが、なるべく同じ部分集合になるように

# 代表的なクラスタリング法： K-means法 (0) 初期代表パターン

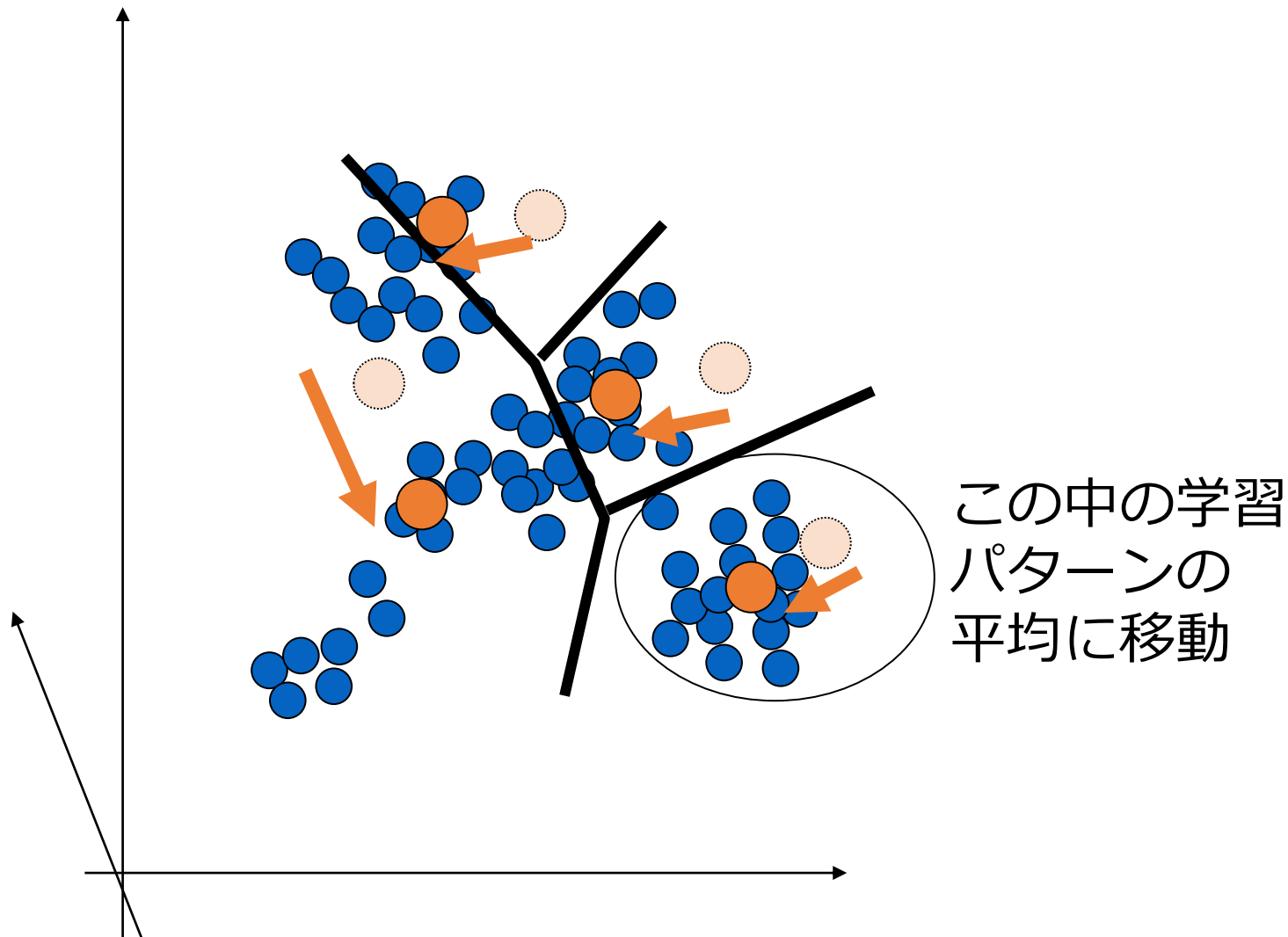


# 代表的なクラスタリング法： K-means法（1）学習パターン分割

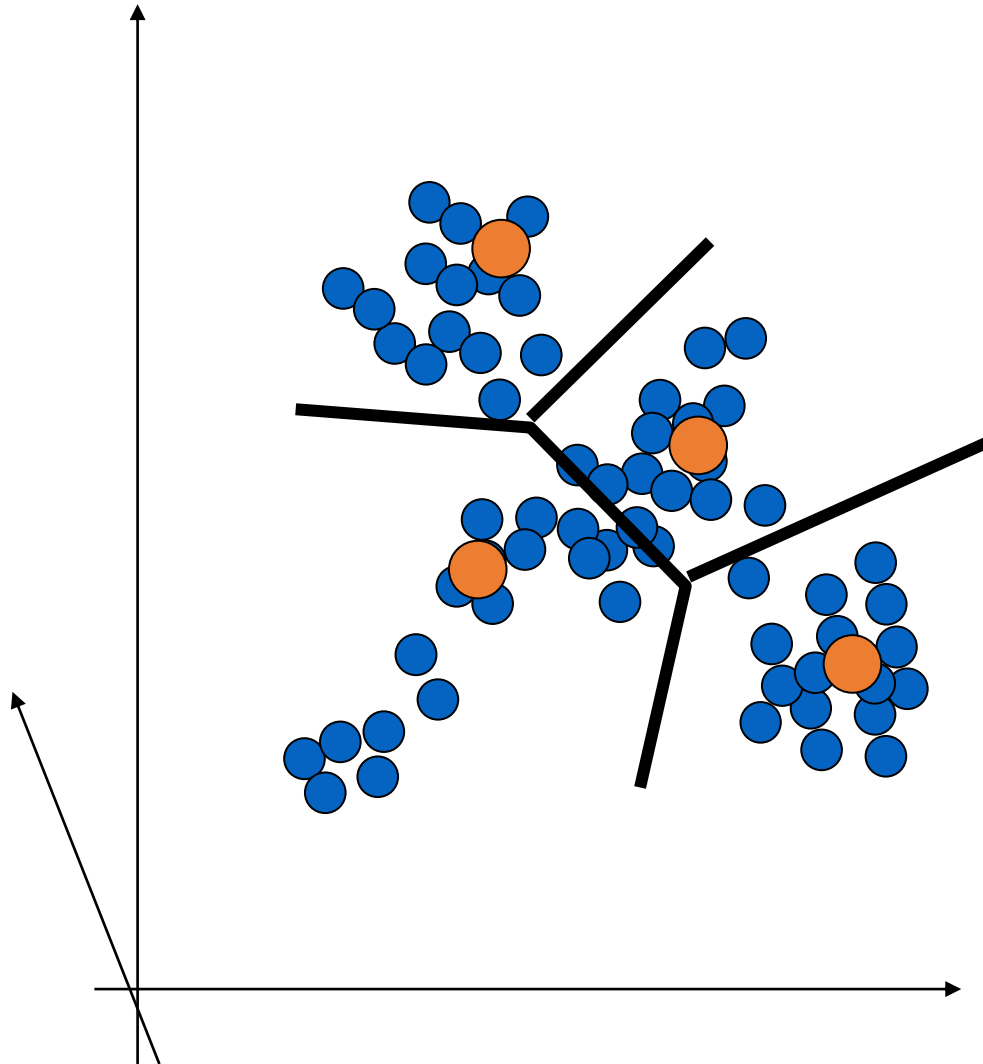
- 各データ点は「どの代表パターンに一番近いか」をユークリッド距離で計算して、近い重心に所属



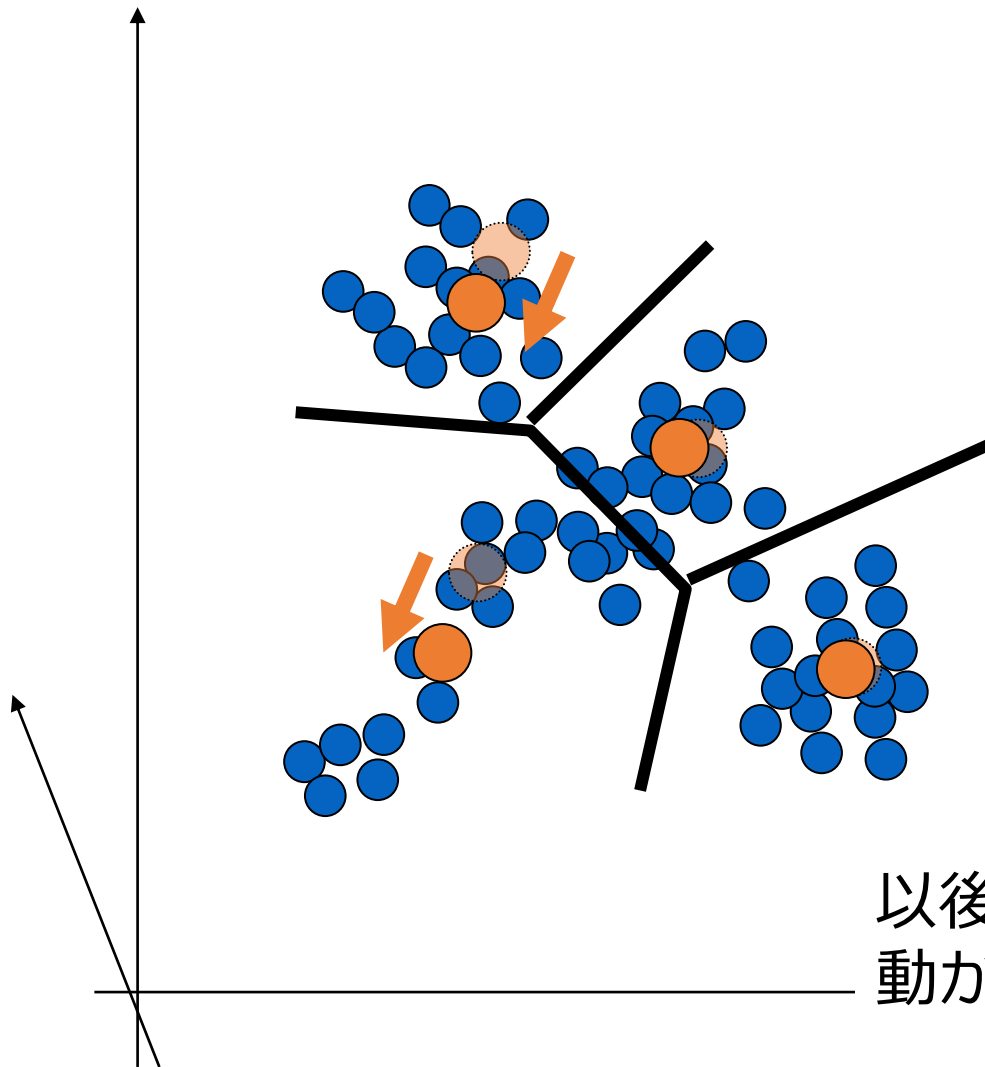
# 代表的なクラスタリング法： K-means法（2）代表パターン更新



# 代表的なクラスタリング法： K-means法（1）学習パターン分割



# 代表的なクラスタリング法： K-means法（2）代表パターン更新



以後、代表パターンが  
動かなくなるまで反復

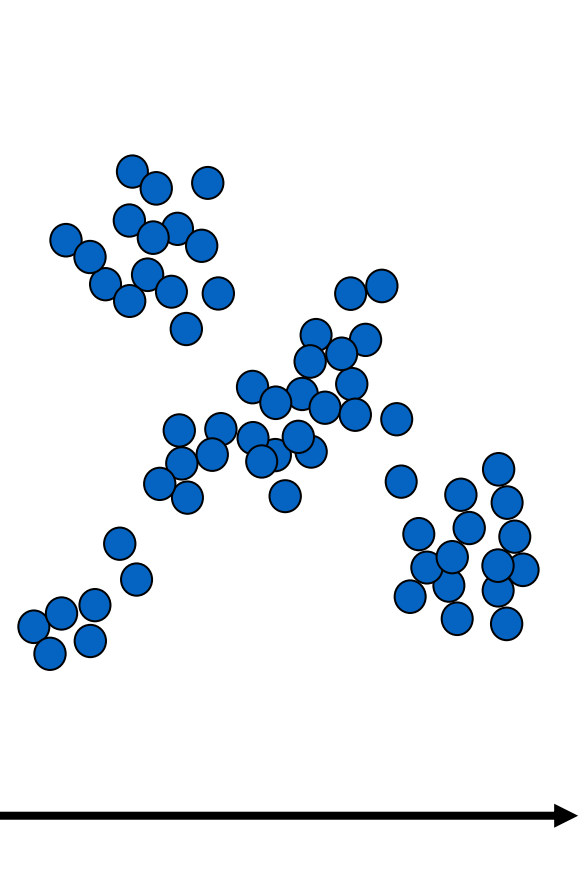
# K-means : プログラミング

- ステップ0 :
  - データ読み込み
  - パラメータ (クラスタ数  $K$  など) の決定
  - 初期代表点の決定
- ステップ1 :
  - 各データサンプルと代表点との距離を計算して、データサンプルがどの代表点のグループに所属するかを決定する関数を作る
- ステップ2:
  - 所属したグループの平均値を用いて、代表点の位置を更新
- ステップ3:
  - ステップ1とステップ2を代表点が (ほぼ) 動かなくなるまで繰り返す

# K-means : プログラミング

- ステップ0 :
  - データ読み込み
  - パラメータ (クラスタ数 K など) の決定

```
# -----  
# 1) データの読み込み  
# -----  
data_dir = "./Data/"  
df = pd.read_csv(data_dir + "data.csv", header=None) # CSV  
を読み込む (ヘッダなし)  
data = df.values.astype(float) # NumPy配列に変換  
  
# -----  
# 2) パラメータの設定  
# -----  
K = 4          # クラスタ数  
tol = 0.1      # 収束判定の閾値 (中心の移動がこれ未満なら終了)  
max_iter = 100 # 最大繰り返し回数  
np.random.seed(0) # 乱数シード固定 (同じ結果を得るため)
```

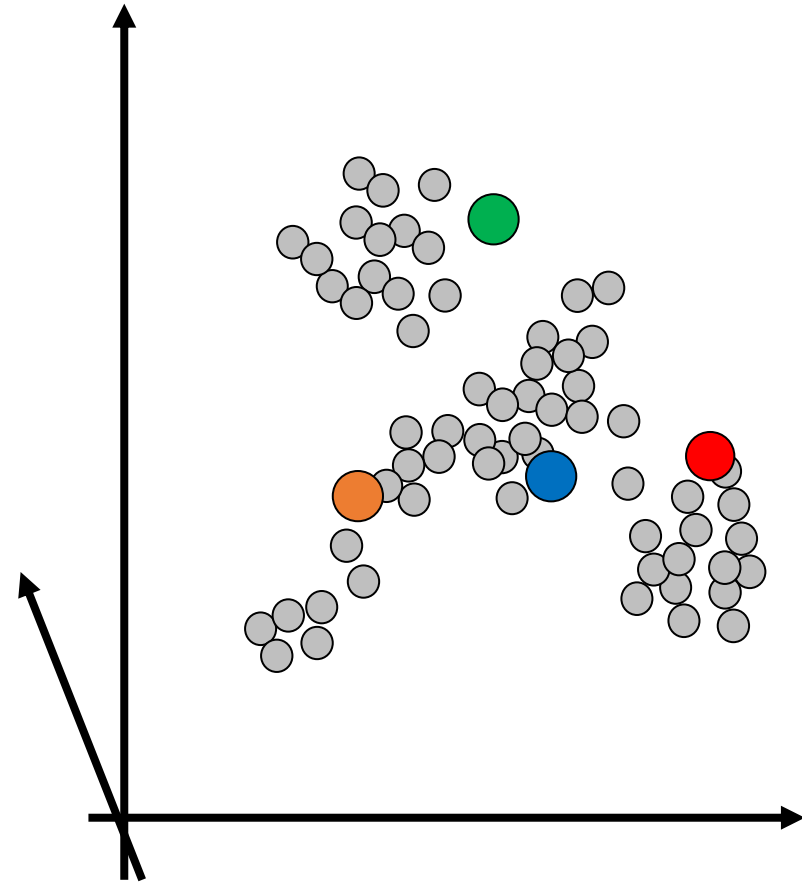


# K-means : プログラミング

- ステップ0 :
  - 初期代表点の決定 (ランダム)

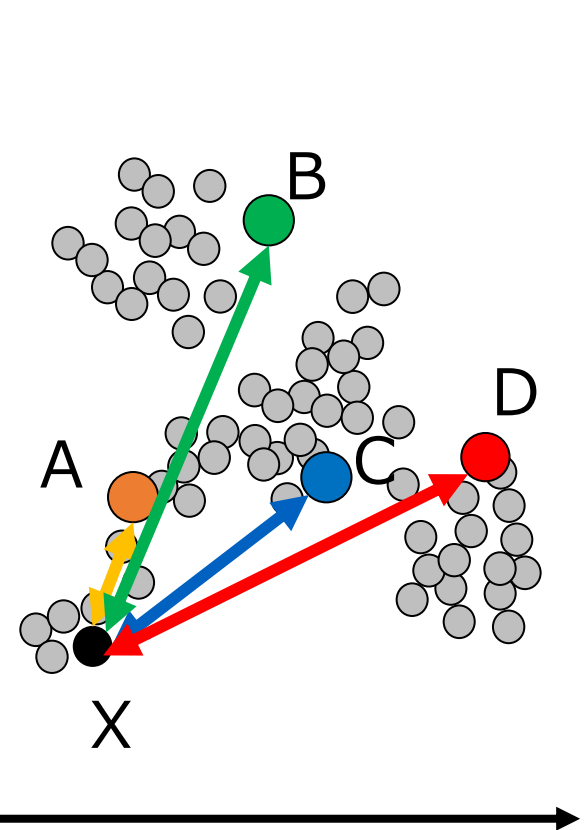
```
x_min = np.min(data[:, 0]) # xの最小値
x_max = np.max(data[:, 0]) # xの最大値
y_min = np.min(data[:, 1]) # yの最小値
y_max = np.max(data[:, 1]) # yの最大値
```

```
# K個の中心を、データ範囲内の一様乱数で作る
centers = [] # 中心を入れるリスト (あとで配列にする)
for i in range(K): # K回くり返してK個用意
    cx = np.random.uniform(x_min, x_max)
    # x座標をランダムに1つ取る
    cy = np.random.uniform(y_min, y_max)
    # y座標をランダムに1つ取る
    centers.append([cx, cy]) # [x, y] の形で保存
centers = np.array(centers) # NumPy配列に変換
```



# K-means : プログラミング

- ステップ 1 :
  - 代表点の支持者決定  
⇒ 最も近い代表点を支持する  
距離を活用！
  - ある点Xに着目
    - 各代表点までの距離を算出

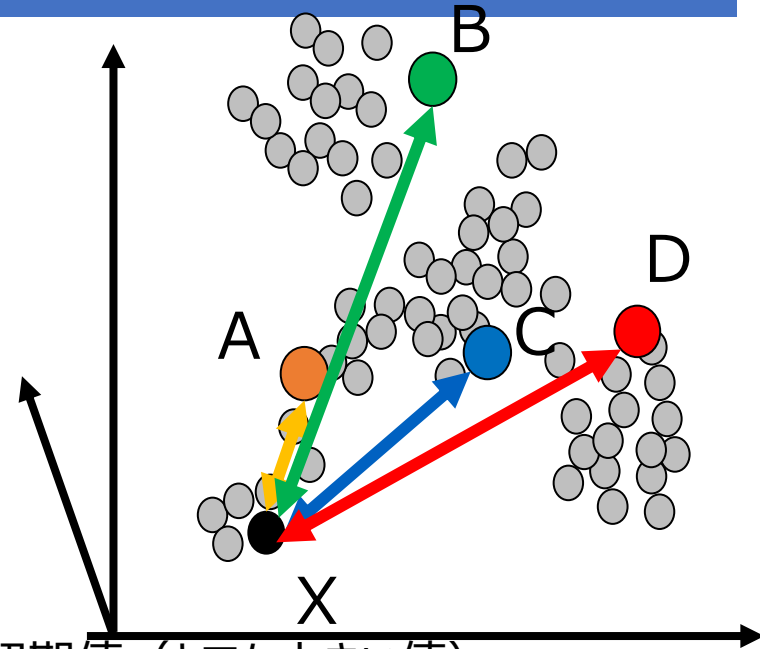


# XとAの距離の計算

```
Ad = numpy.linalg.norm(X-A)
```

# K-means : プログラミング

- ステップ 1 :
  - ある点Xに着目
  - 距離が最も近い代表点を決定



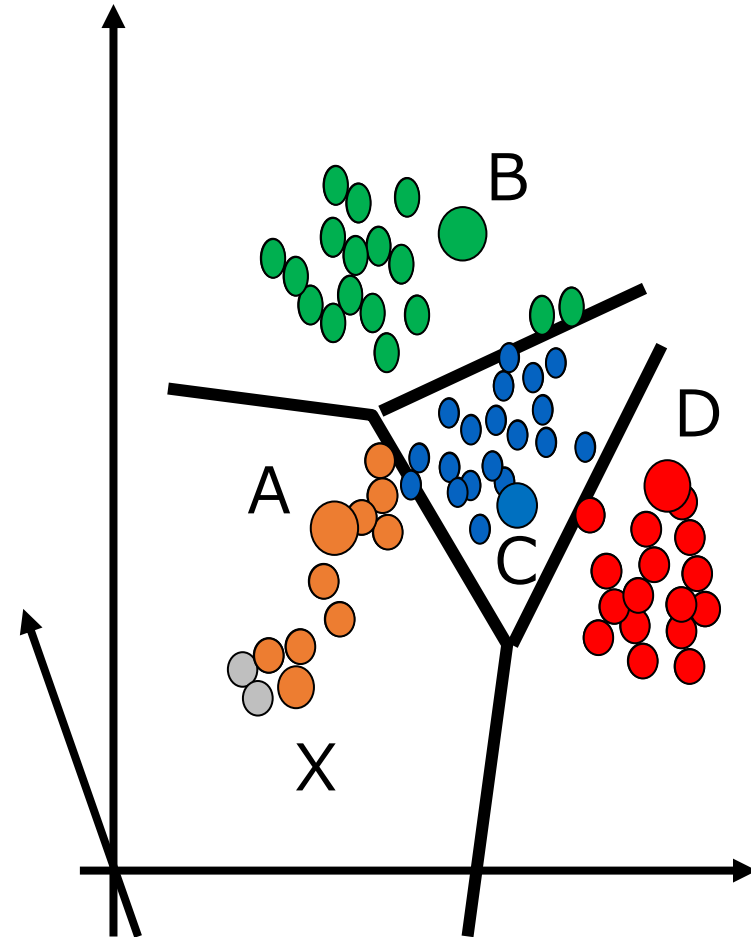
```
def nearest_center(point, centers):  
    min_distance = float("inf") # 最小距離の初期値 (とても大きい値)  
    min_index = 0 # 最も近い中心の番号を覚えておく変数  
    for i in range(len(centers)):  
        # すべての中心について距離を測る  
        center = centers[i] # i番目の中心  
        distance = np.linalg.norm(point - center) # ユークリッド距離  
        if distance < min_distance: # もし今の距離が最小より小さければ  
            min_distance = distance # 最小距離を更新  
            min_index = i # 中心の番号も更新  
    return min_index # 一番近い中心の番号を返す
```

# K-means : プログラミング

- ステップ 1 :
  - 全ての点で代表点を決定

```
def assign_clusters(data, centers):  
    labels = [] # 各点のクラスタ番号を入れるリスト  
    for i in range(len(data)): # 全データ点について  
        point = data[i] # i番目の点  
        cid = nearest_center(point, centers)  
        # 最も近い中心の番号  
        labels.append(cid) # 結果を保存  
    return labels # 長さNのリスト (要素は0..K-1)
```

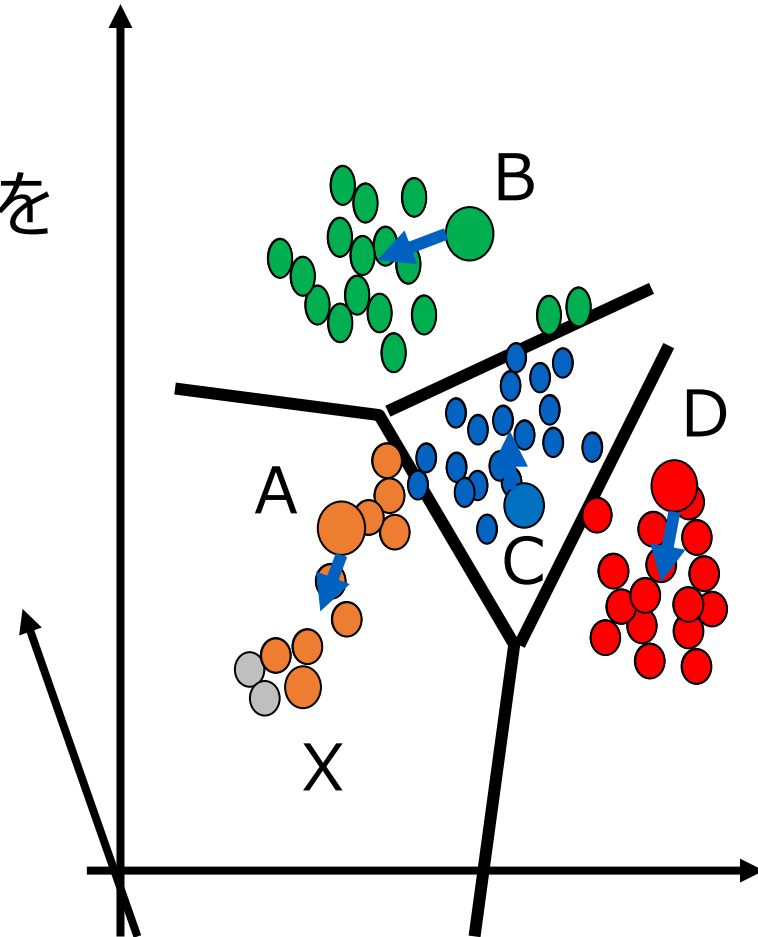
ある点がどの代表点に  
一番近いか？  
→どのクラスタに属するか？



# K-means : プログラミング

- ステップ2 : 代表点を更新
  - 各クラスターの支持者群の平均値を代表点とする。

```
def update_centers(data, labels, K):  
    new_centers = [] # 新しい中心を入れるリスト  
    for k in range(K): # 各クラスターごとに処理  
        members = [] # クラスターkに属する点を集める  
        for i in range(len(data)): # すべての点を確認  
            if labels[i] == k: # 点がクラスターkなら  
                Members.append(data[i]) # その点を追加  
        if len(members) > 0: # メンバーが1つ以上あれば  
            members = np.array(members) # 配列に変換  
            mean_xy = np.mean(members, axis=0)  
            # x,yそれぞれの平均を計算  
        else:  
            # 空クラスター対策 : データからランダムに1点を中心として選ぶ  
            mean_xy = data[np.random.randint(0, len(data))]  
            new_centers.append(mean_xy) # 新しい中心を追加  
    return np.array(new_centers) # K×2の配列で返す
```



# K-means : プログラミング

- アルゴリズム

- ステップ0

- 繰り返し

- クラスタの割り当て

- 代表点の更新

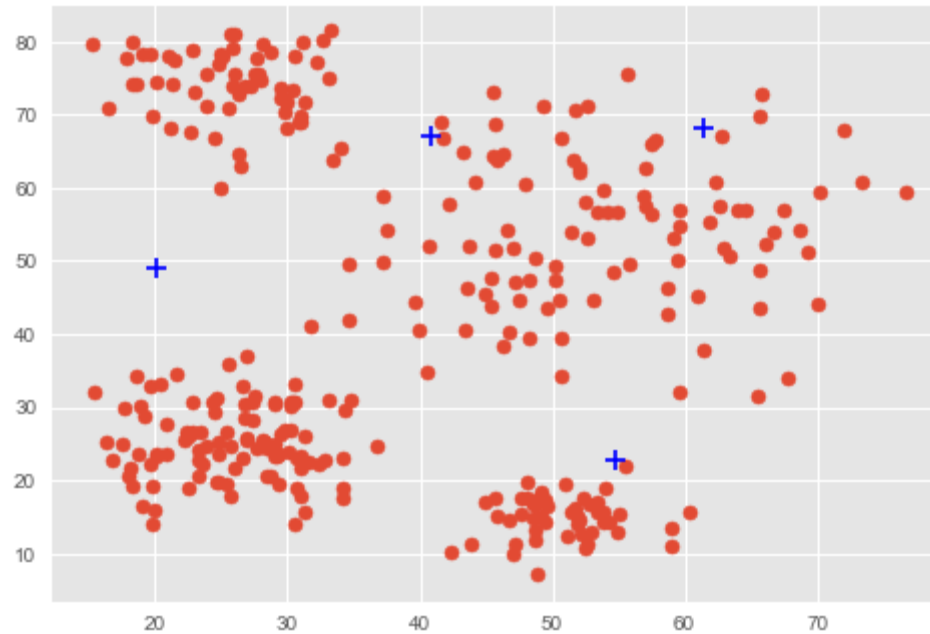
- 終了チェック

```
for it in range(max_iter): # 最大 max_iter 回繰り返し
    labels = assign_clusters(data, centers) # (a) 割り当て
    new_centers = update_centers(data, labels, K) # (b) 代表点の更新

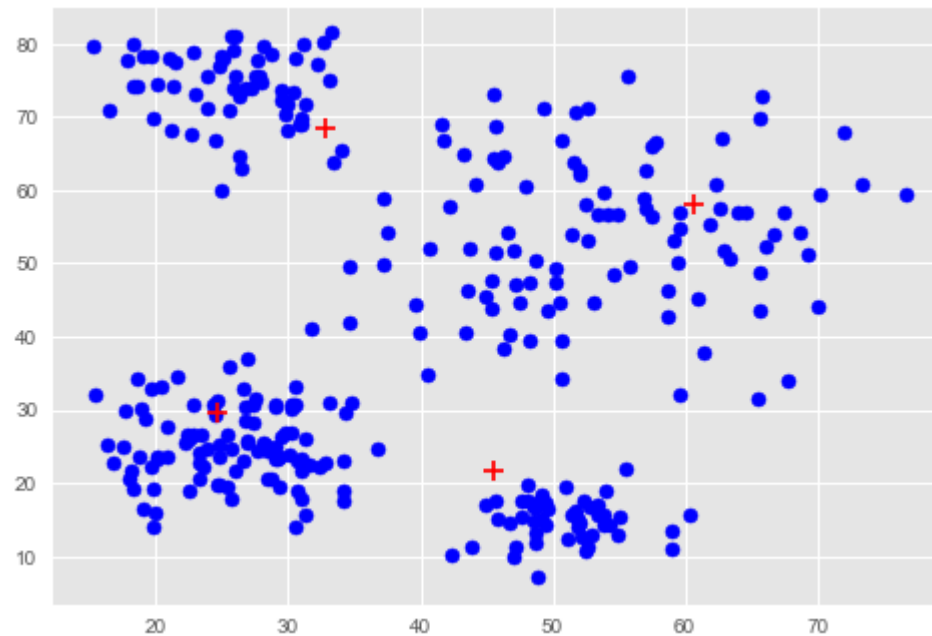
    # (c) 収束判定 : 各中心の移動距離を計算して最大値を見る
    max_shift = 0.0 # 最大移動距離の初期値
    for j in range(K): # すべてのクラスタ中心で確認
        shift = np.linalg.norm(new_centers[j] - centers[j]) # j番目の移動量
        if shift > max_shift: # もし今の方が大きければ更新
            max_shift = shift
    print(f"Iter {it+1:02d} | 最大移動距離 = {max_shift:.4f}")
    centers = new_centers # 中心を更新して次へ

if max_shift < tol: # 中心がほとんど動かなくなったら
    break # ループを抜ける
```

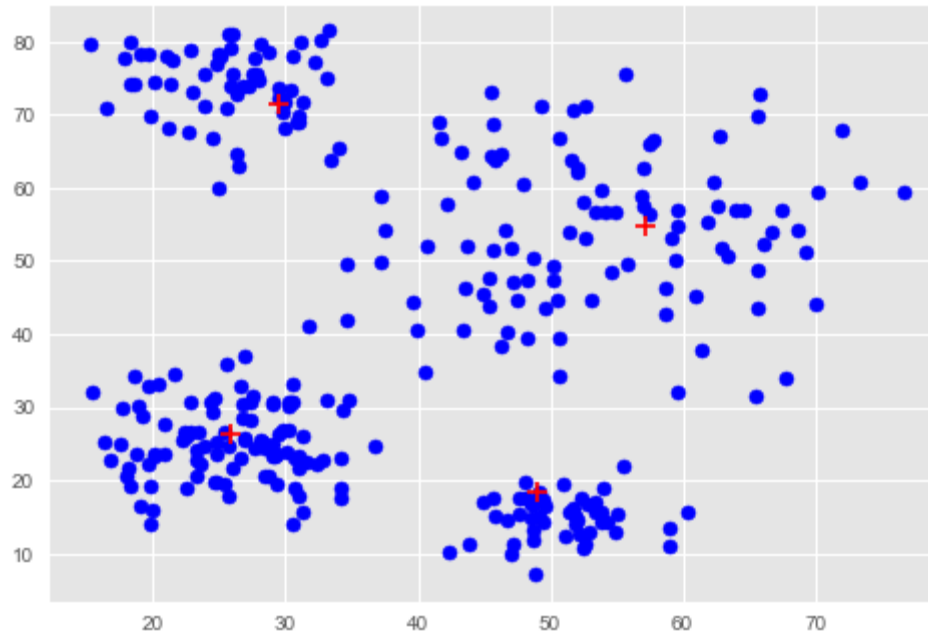
# K-meansクラスタリング



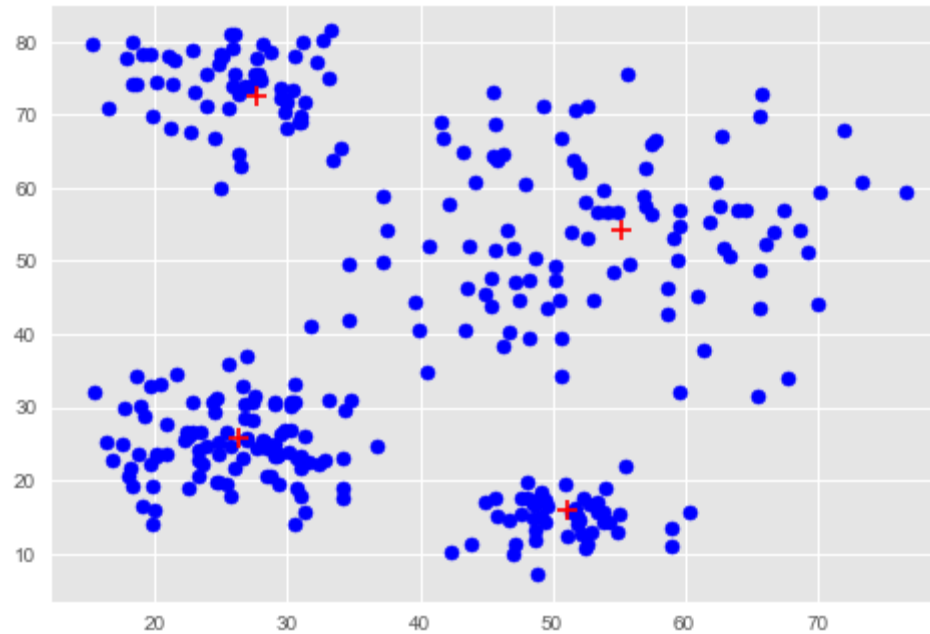
# K-meansクラスタリング



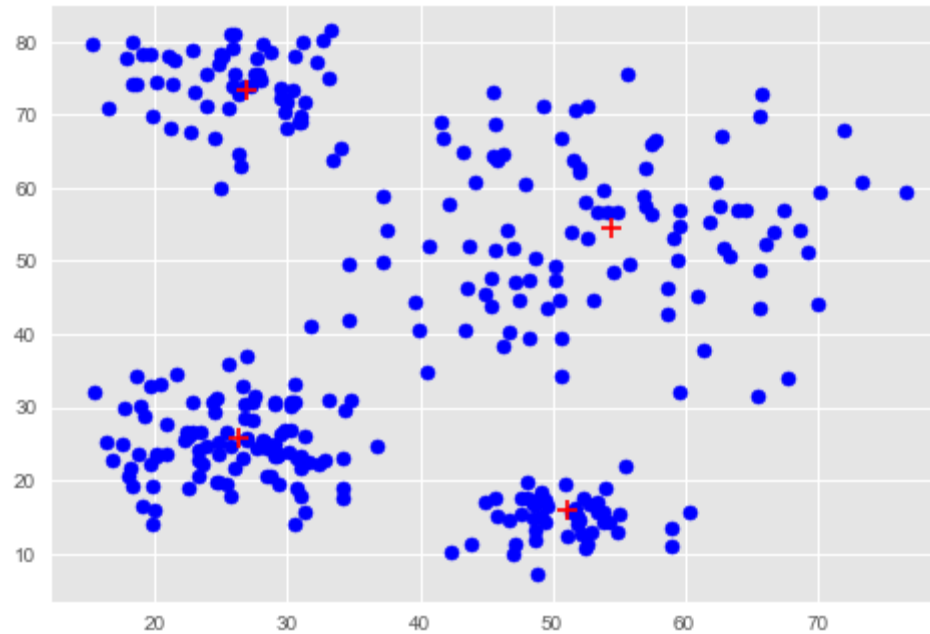
# K-meansクラスタリング



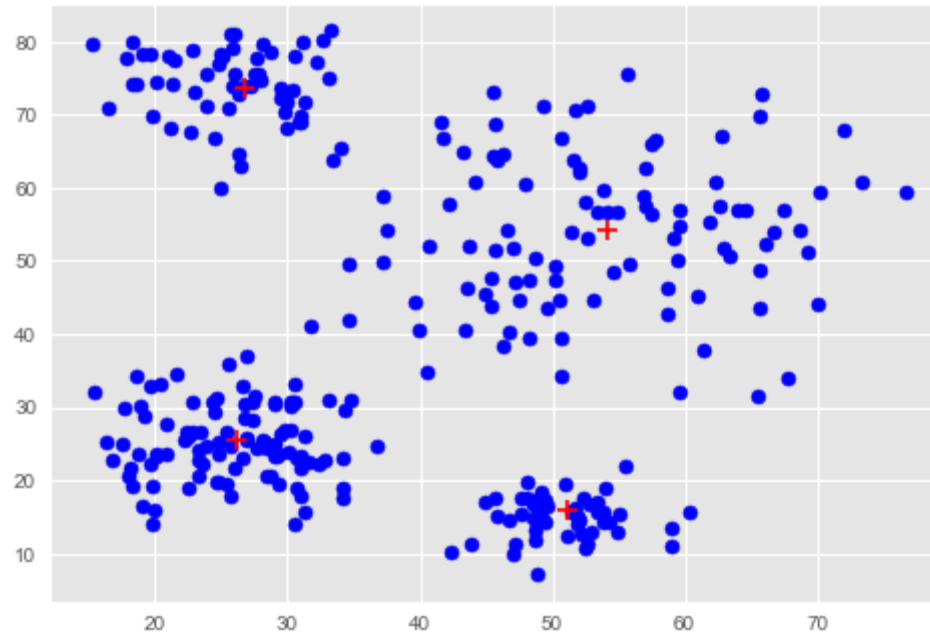
# K-meansクラスタリング



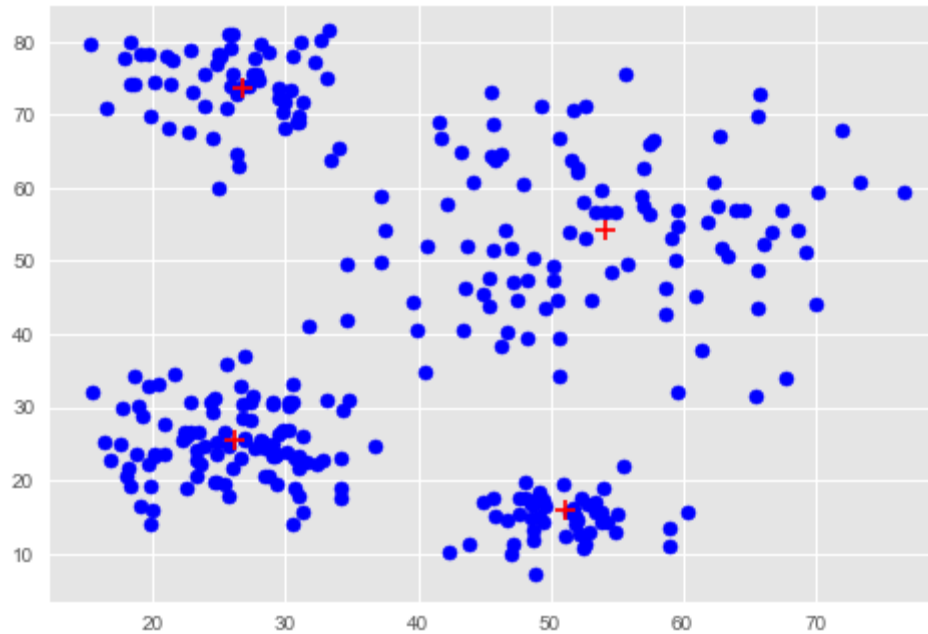
# K-meansクラスタリング



# K-meansクラスタリング



# K-meansクラスタリング



# ライブラリを使った K-Means クラスタリング

```
from sklearn.cluster import KMeans
km_model = KMeans(n_clusters=4) #K=4のkmeansモデルの初期化
km_model.fit(data) #実際にデータをクラスタリング
```

```
In [564]: km_model.cluster_centers_
Out[564]:
array([[26.15670902, 25.59839271],
       [53.97625751, 54.52132754],
       [51.44211711, 16.33067878],
       [26.76710588, 73.68744256]])
```

クラスタの中心たち

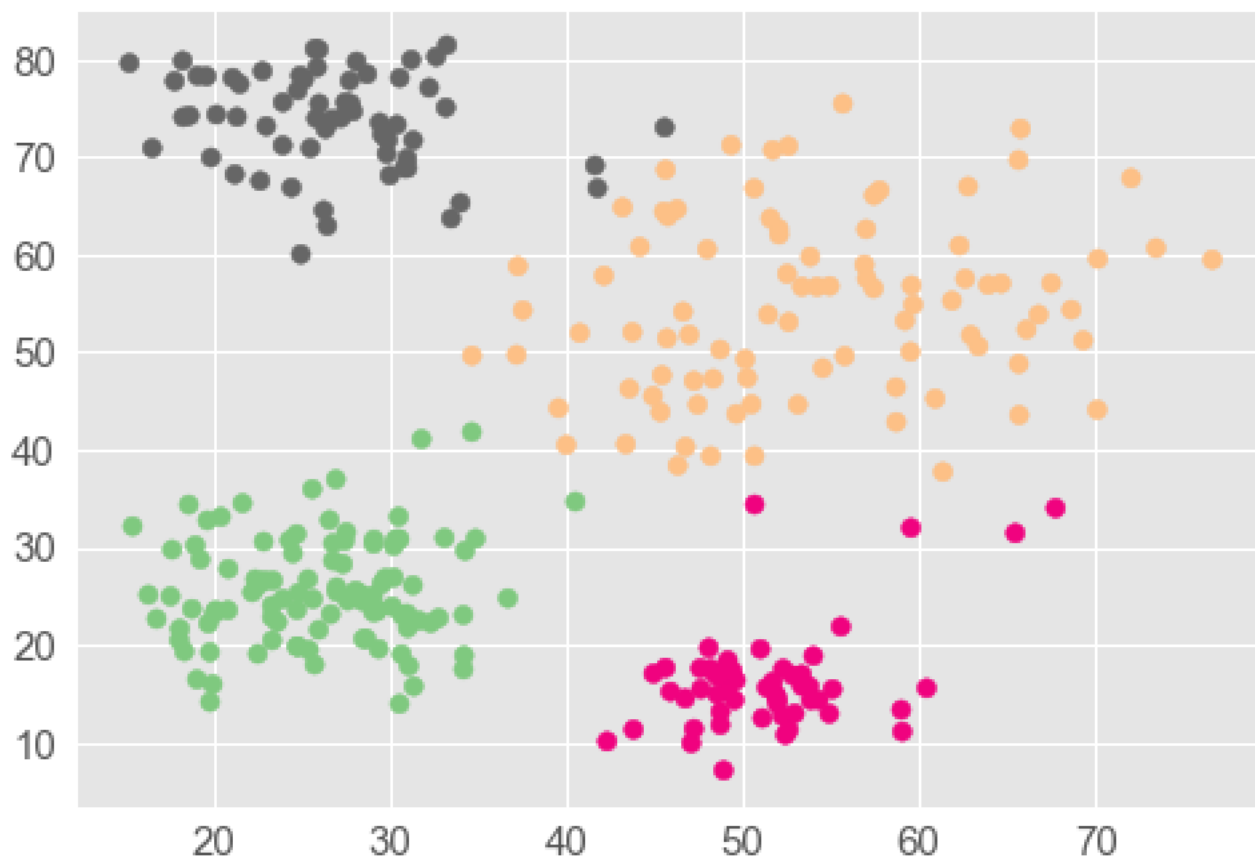
```
In [567]: km_model.labels_
Out[567]: array([0, 0, 0, ..., 3, 3, 3], dtype=int32)
```

各データがどのクラスタか  
(クラスタ0～クラスタ3)

# 可視化も簡単

```
labels=km_model.labels_ # 各データがどのクラスタかが入っている
```

```
plt.scatter(vlist[:,0], vlist[:,1], c=labels,cmap=cm.Accent)
```



# 演習

# 演習 1

- csvファイル" height\_weight.csv"を読み込み、 $A=(180,75)$ からの各点への正規化相間（類似度）をそれぞれ算出し、類似度の小さい順に並び替え、正規化相間の大きさに色を変えてプロットせよ
- ヒント) 

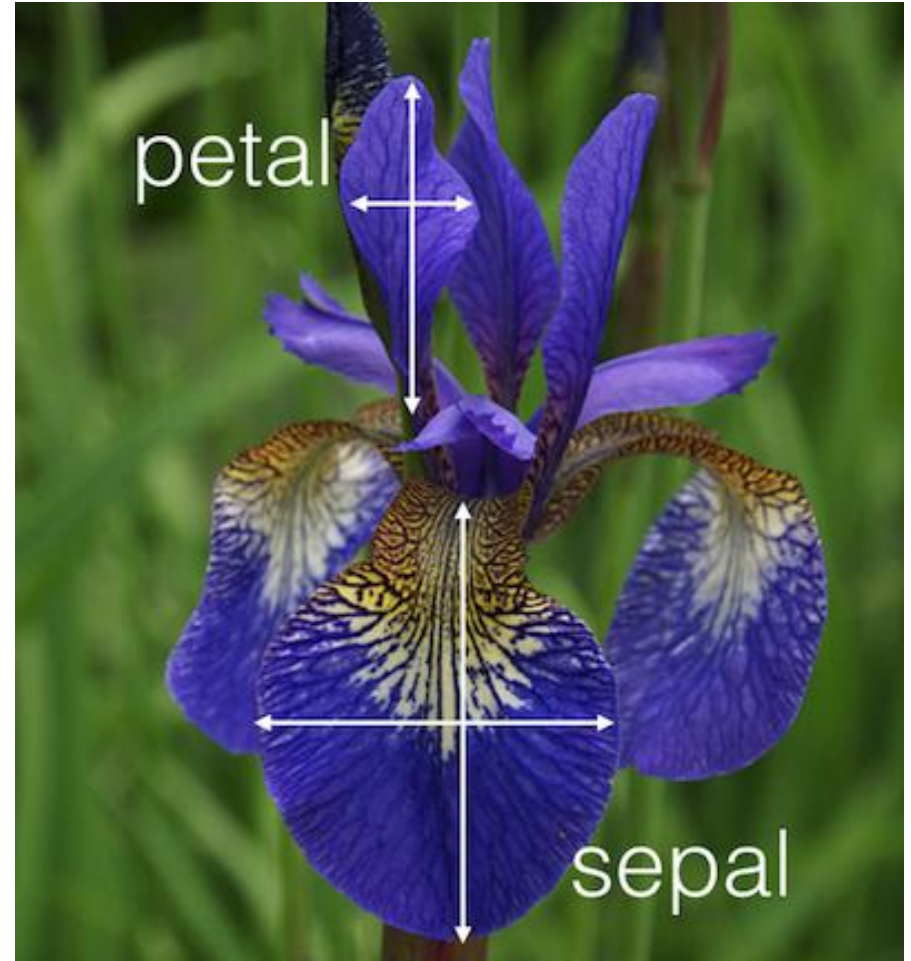
```
import matplotlib.cm as cm
cm.hot
```

## 演習 2

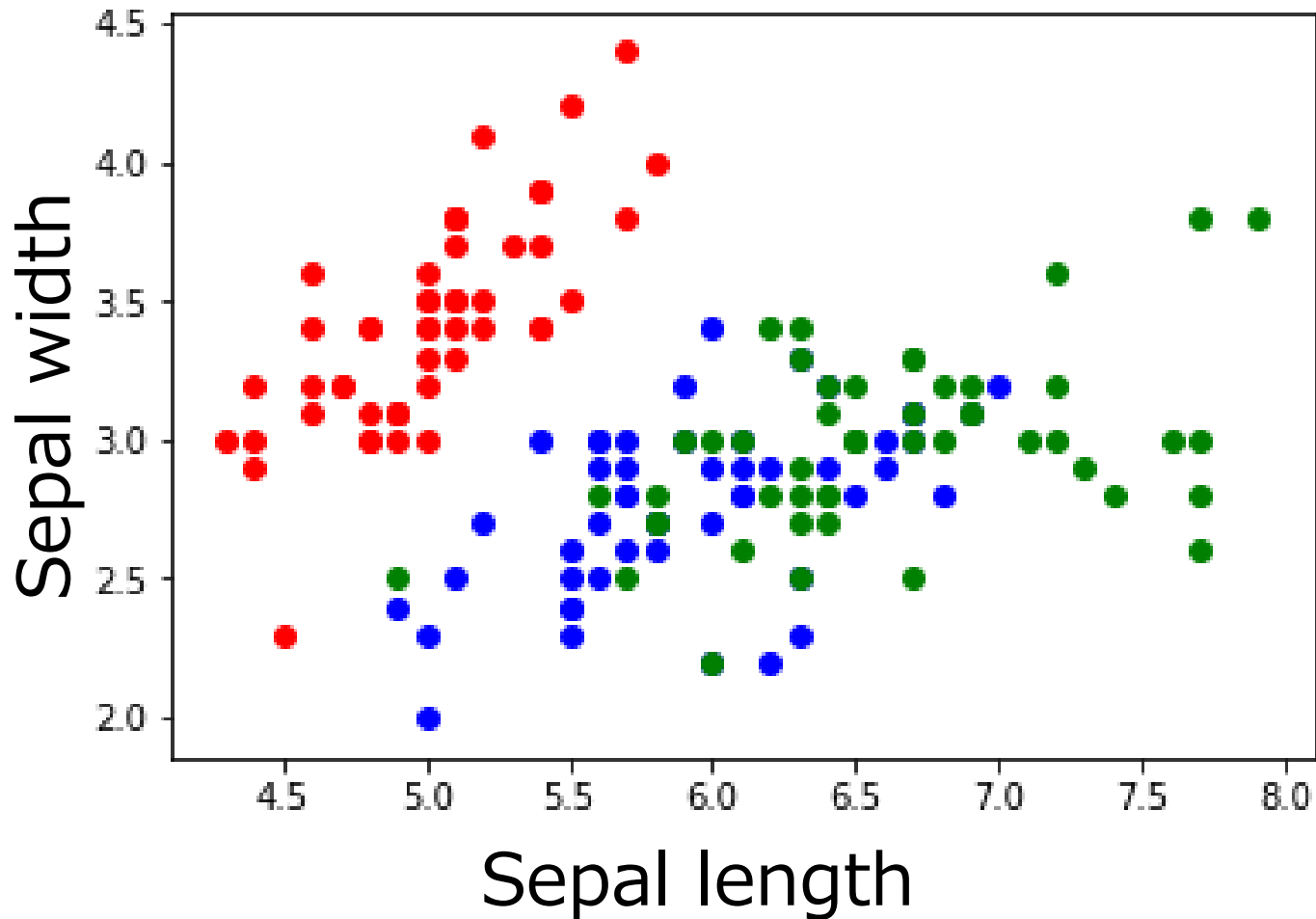
- クラスタリングの対象となるcsvデータ“data.csv”を読み込み：
  - $K=2, 3, 4$  でそれぞれ K-means クラスタリングせよ
  - 余裕がある人は自分で K-means を実装せよ
- 注意：data.csvはヘッダ（項目名）がないので、`pd.read_csv(ファイル名, header=None)`とすること

# アイリスデータ (1)

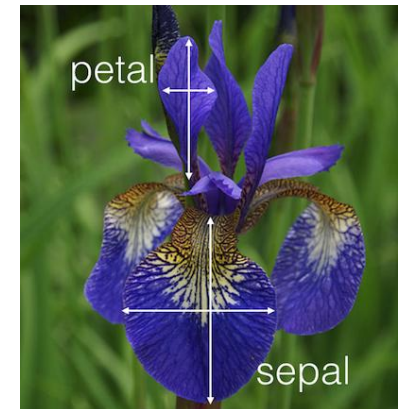
- 3クラス
  - Iris-setosa
  - Iris-versicolor
  - Iris-virginica
- 指標
  - sepal length
  - sepal width
  - petal length
  - petal width



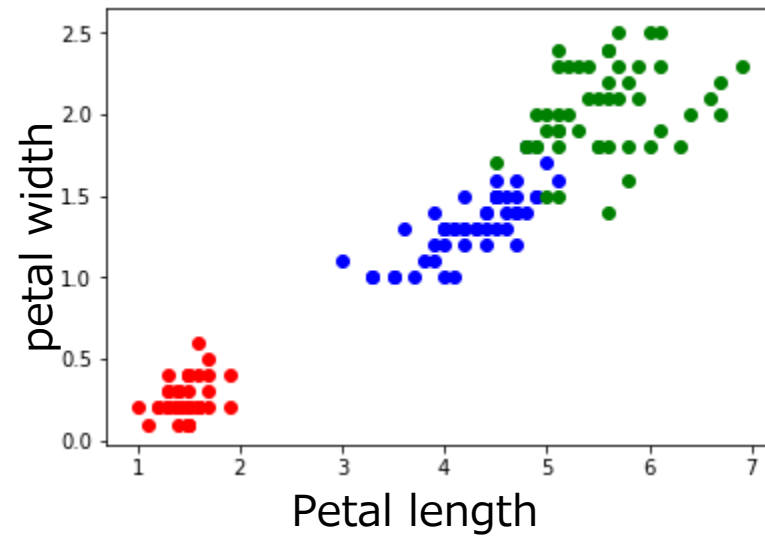
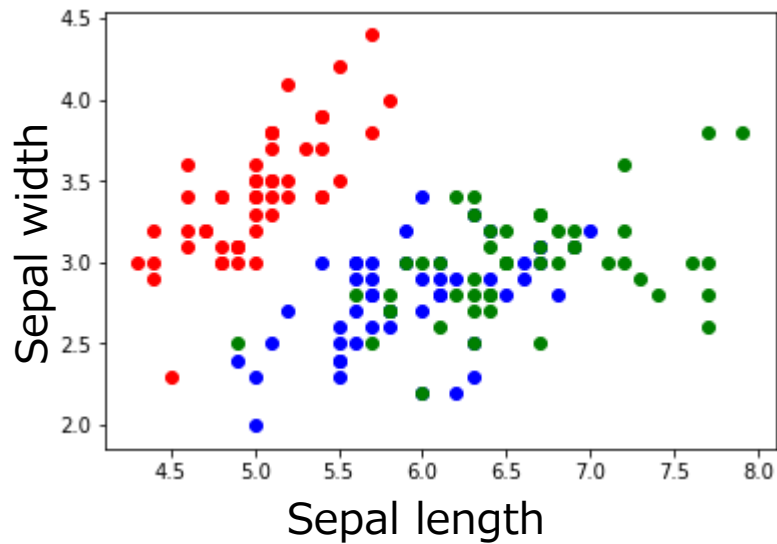
# アイリスデータ (2)



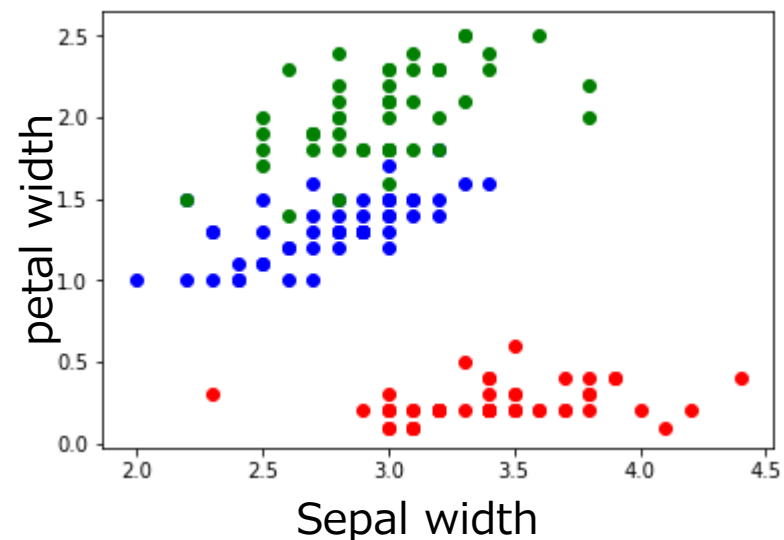
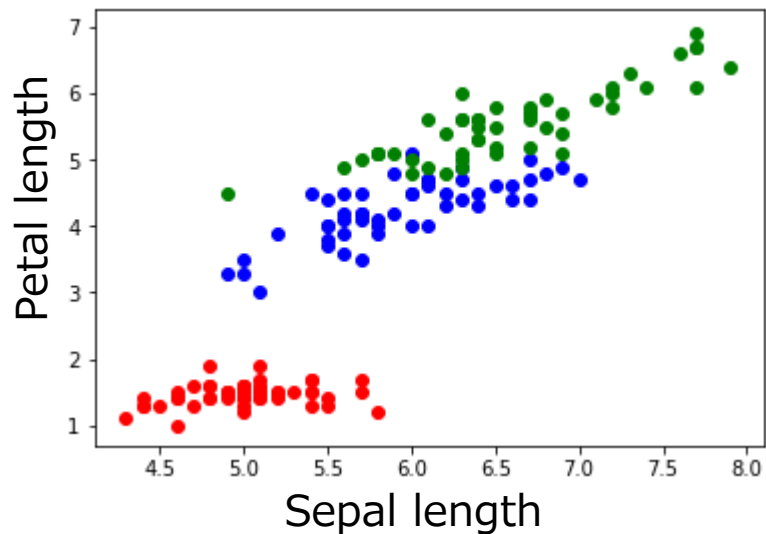
- Iris-setosa
- Iris-versicolor
- Iris-virginica



# アイリスデータ (3)



- setosa
- versicolor
- virginica



# 演習 3

- “iris.csv”を読み込み、以下の2つの指標だけで、 $K=3$ でクラスタリング可能か可視化せよ
- 指標 1
  - sepal length
  - sepal width
- 指標 2
  - sepal width
  - petal width

0列目と1列目を取り出す

実際のクラスと  
見比べてみよう