

# 多変量解析①

## (次元削減とクラスタリング)

国立遺伝学研究所  
生命情報研究センター  
東 光一

# 次元削減とクラスタリング

## 0. 事前準備

```
In [ ]: %matplotlib inline

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(style='whitegrid')
pd.options.display.float_format = '{:.2f}'.format
pd.set_option('display.width', 800)

import matplotlib
import sklearn
print('pandas',pd.__version__)           # VM+Ubuntu: 0.23.4
print('numpy',np.__version__)            # VM+Ubuntu: 1.15.1
print('matplotlib',matplotlib.__version__) # VM+Ubuntu: 2.2.3
print('seaborn',sns.__version__)          # VM+Ubuntu: 0.9.0
print('scikit-learn',sklearn.__version__) # VM+Ubuntu: 0.19.2
```

この時間、中心的に扱うライブラリは numpy, pandas, sklearn (scikit-learn)

「事前準備のお願い」に沿って準備された方は大丈夫（↑のバージョンのはず）  
自分の環境で計算される方は、もしもこれらのライブラリが古い場合は動かないコードがあるかも

# この時間の演習？の内容

```
▶ In [2]: # データ読み込み  
df = pd.read_table('./input/count_tpm.tsv', index_col=0)  
print(df.head())  
print(len(df))
```

gene_id	batch_1	batch_2	batch_3	chemostat_1	chemostat_2	chemostat_3
gene_0001	0.00	0.73	3.13	0.00	0.00	0.50
gene_0002	0.00	0.00	0.00	0.00	0.00	0.00
gene_0003	0.00	0.00	0.00	0.00	0.00	0.00
gene_0004	0.00	0.00	0.00	0.00	0.00	0.00
gene_0005	0.95	2.80	4.97	4.69	4.37	8.66
5983						

昨日の演習で作った、  
TPM正規化されたRNA-seqデータ（5,983遺伝子 × 6サンプル）  
のみを入力として使って、このテーブル（行列）を色々変換したりして、  
何かおもしろいことを見つける。

ただ、pythonによる多変量解析ではほとんどの場合、

```
tsne = sklearn.manifold.TSNE(n_components=2)  
coords = tsne.fit_transform(std_df.transpose().values)
```

とか数行で済んでしまうので、プログラミングの演習というよりは、  
手法の紹介がメイン。

# この時間に扱う内容

## 教師なし学習 (Unsupervised learning)

異常検知

自己組織化マップ

ニューラルネット  
(GAN, VAEなど)

生成モデル、  
ノンパラメトリックベイズなど

この時間の内容  
PCA, MDS, NMF, LSI, 階層的  
クラスタリング、K-means、  
混合モデル、など

他のクラスタリング手法たくさん

隠れマルコフモデル  
独立成分分析

## 教師付き学習 (Supervised learning)

回帰

k-NN

SVM

Random forest

ニューラルネット  
(CNNなどmultilayer  
perceptronたくさん)

## 目的：

高次元空間に散らばっているデータの分布を推定して、  
人間が解釈しやすいなんらかの情報を抽出する

## 強化学習 (Reinforcement learning)

Q学習

SARSA

ニューラルネット  
(DQN、A3C...)

# 高次元データ

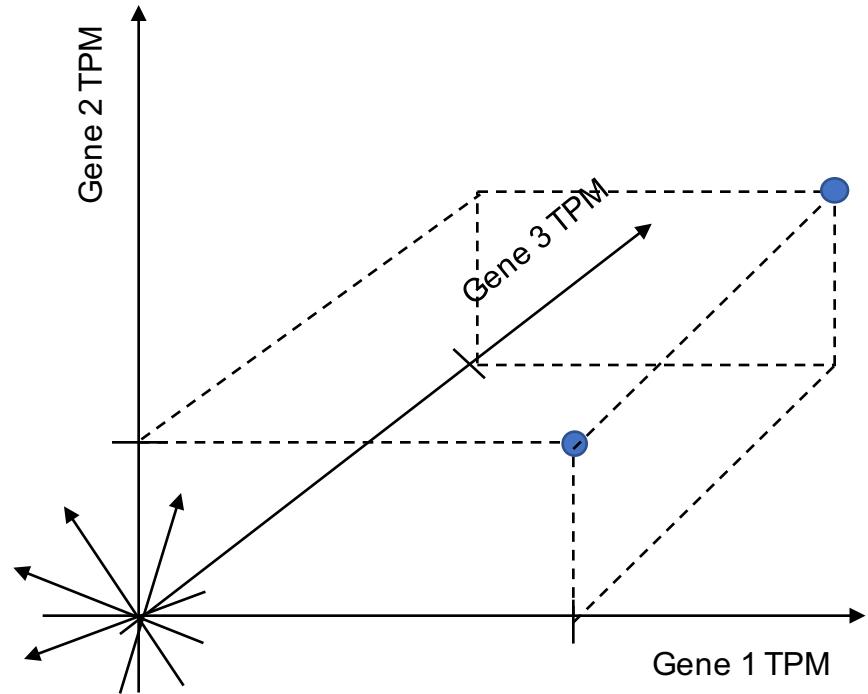
次元：空間内の特定の対象を記述するときに何個の変数が必要になるか

5,892次元  
(各軸は各遺伝子  
のTPM) で  
ひとつのサンプルが  
表現される

```
print(df)
```

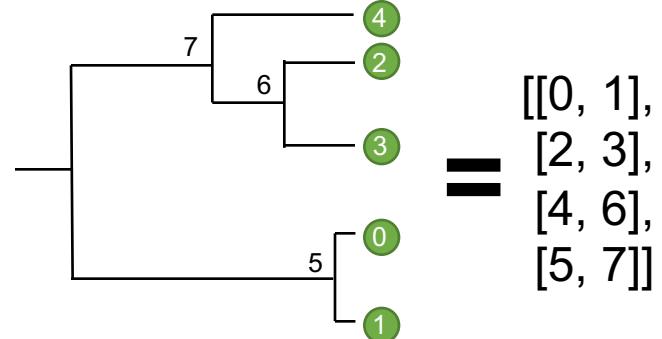
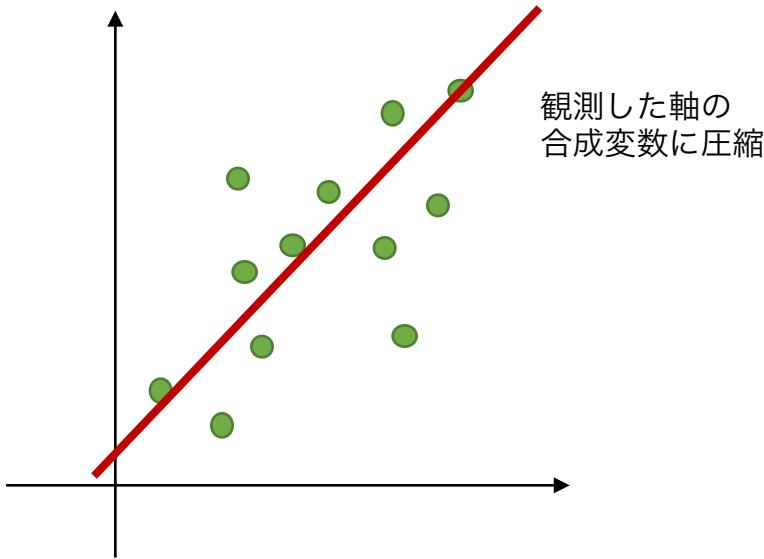
	batch_1	batch_2
gene_id		
gene_0001	0.00	0.73
gene_0005	0.95	2.80
gene_0009	1.46	1.24
gene_0010	8.55	7.96
gene_0011	17.15	12.23
gene_0012	181.63	204.34
gene_0013	90.20	95.14
gene_0014	33.77	37.64
gene_0015	20.85	21.19
gene_0016	14.97	13.26
gene_0017	10.21	8.34
gene_0018	79.28	63.09
gene_0019	17.76	17.43
gene_0020	96.32	81.33
gene_0021	17.62	16.56
gene_0022	13.15	14.12
gene_0023	74.91	71.71
gene_0024	54.26	70.47
gene_0025	506.14	473.79
gene_0026	29.58	31.38
gene_0027	224.12	232.68
gene_0028	28.32	31.08
gene_0029	46.65	47.58
gene_0030	88.56	71.44
gene_0031	3,231.45	3,503.06
gene_0032	19.43	14.34
gene_0033	15.73	13.27
gene_0034	101.58	104.38
gene_0035	161.17	188.18
gene_0036	18.90	19.46
...	...	...
gene_6370	115.10	117.70
gene_6371	41.24	35.03
gene_6372	18.94	18.23
gene_6373	53.76	52.08
gene_6374	23.65	22.51
gene_6375	19.56	17.48
gene_6376	33.75	37.80
gene_6377	125.31	133.67
gene_6378	86.54	69.47
gene_6379	147.65	147.59
gene_6380	8.11	6.77
gene_6381	29.01	28.99
gene_6382	24.51	21.39
gene_6383	478.36	445.00
gene_6384	78.59	76.42
gene_6385	22.95	24.41
gene_6386	39.26	45.19
gene_6387	289.39	271.47
gene_6388	5.90	6.10
gene_6389	7.67	8.21
gene_6390	19.28	9.62
gene_6391	9.86	6.87
gene_6392	11.13	9.44
gene_6393	56.91	49.02
gene_6394	29.60	28.93
gene_6395	9.66	8.14
gene_6396	4.76	6.47
gene_6397	7.06	9.11
gene_6398	0.00	2.59
gene_6399	5.36	6.84

[5892 rows x 6 columns]



# 次元削減とクラスタリング

高次元空間内のサンプルの分布に関する重要な特徴を残したまま、  
少ない変数でそれぞれのサンプルを記述する  
どういう情報を残すか、どんな変数に落とすかで様々なバリエーション



乱雑に分布しているように見えても、  
潜在変数（実験で直接は観測できなかった隠れた変数）  
の上では秩序立って分布しているかもしれない  
そういう隠れた関係を明らかにする。

(サンプル数 - 1) × 2 の行列  
(linkage matrix) に圧縮

ただし、教師なしで推定するため、得られた少ない変数が解釈しやすいとは限らない

# この時間に扱う内容

1. 次元削減
  - i. 行列分解による次元削減
    - a. 主成分分析 (Principal component analysis; PCA)
    - b. 非負値行列因子分解 (Non-negative matrix factorization; NMF)
    - c. 潜在意味解析 (Latent semantic indexing; LSI)
  - ii. 距離行列の最適化による次元削減
    - a. 多次元尺度構成法 (Multidimensional scaling; MDS)
    - b. その他の多様体学習
2. クラスタリング
  - i. 階層的クラスタリング
  - ii. クラスター「中心」との距離を最適化するクラスタリング
    - a. k-means
    - b. k-medoids
  - iii. 混合正規分布 (Gaussian mixture)

1. 次元削減
  - i. 行列分解による次元削減
    - a. 主成分分析 (Principal component analysis; PCA)
    - b. 非負値行列因子分解 (Non-negative matrix factorization; NMF)
    - c. 潜在意味解析 (Latent semantic indexing; LSI)
  - ii. 距離行列の最適化による次元削減
    - a. 多次元尺度構成法 (Multidimensional scaling; MDS)
    - b. その他の多様体学習
2. クラスタリング
  - i. 階層的クラスタリング
  - ii. クラスター「中心」との距離を最適化するクラスタリング
    - a. k-means
    - b. k-medoids
  - iii. 混合正規分布 (Gaussian mixture)

# まずはデータ読み込み

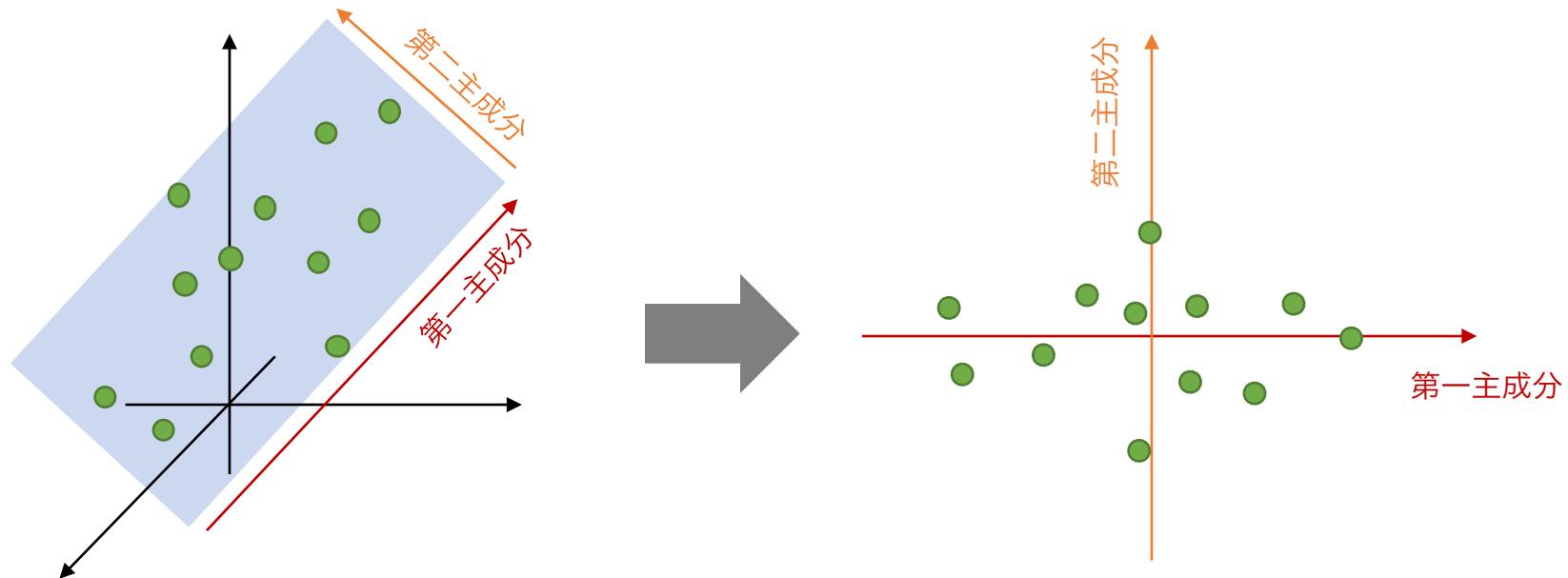
```
In [ ]: # データ読み込み
df = pd.read_table('./input/count_tpm.tsv', index_col=0)
print(df.head())
print(len(df))
```

```
In [ ]: # 全サンプルでTPMがゼロの遺伝子のレコードを削除
all_zero_index = df.index[df.sum(axis=1) == 0]
df = df.drop(all_zero_index)
print(df.head())
print(len(df))
```

```
In [ ]: # サンプルをプロットするときの色を設定
sample_colors = {'batch_1':'b', '# b: blue
                  'batch_2':'b',
                  'batch_3':'b',
                  'chemostat_1':'g', '# g: green
                  'chemostat_2':'g',
                  'chemostat_3':'g'}
colors = df.columns.map(sample_colors)
```

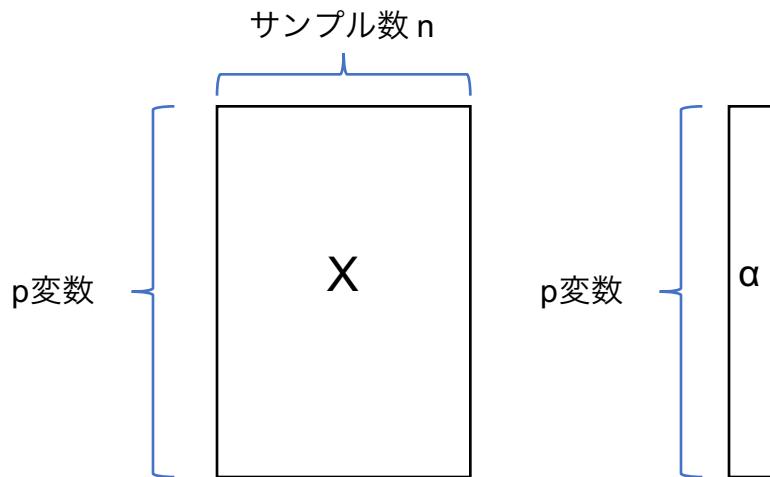
# 主成分分析 (Principal component analysis; PCA)

高次元空間にちらばっているデータでは、しばしば変数のいくつかは相関している。データの分散（ばらつき、広がり）が最大となるような合成変数（主成分）を取り出す。次に、その合成変数と直交する（無相関の）方向で分散が最大の合成変数を取り出す。このように作った合成変数でデータを表現すれば、データの広がり方に関する情報の損失が少なく、分布を少ない変数で表現できるはず、と考える。



# 主成分分析 (Principal component analysis; PCA)

主成分分析は、データの共分散行列の固有値分解で解ける。



$Var(X^T \alpha)$ を最大にするような係数ベクトル  $\alpha$  を求めたい。  
そのままだと無限にでかい  $\alpha$  が解になってしまうので、 $\alpha^T \alpha = 1$  という条件を設けて、ラグランジュの未定乗数法で極値を求める。  
 $X$  の共分散行列を  $\Sigma$  とすると、 $Var(X^T \alpha) = \alpha^T \Sigma \alpha$  なので、 $\alpha$  について以下の式を微分する。  
 $\alpha^T \Sigma \alpha - \lambda(\alpha^T \alpha - 1)$

$$\frac{d}{d\alpha} (\alpha^T \Sigma \alpha - \lambda(\alpha^T \alpha - 1)) = 0$$

$$\Sigma \alpha - \lambda \alpha = 0$$

$$\Sigma \alpha = \lambda \alpha$$

$\Sigma$  の固有方程式となっていて、  
結局、共分散行列  $\Sigma$  の固有値と固有ベクトルを求める問題となる。

# PCA実行

```
In [ ]: import sklearn.decomposition
```

```
In [ ]: # PCA実行  
pca = sklearn.decomposition.PCA()  
coords = pca.fit_transform(df.transpose().values)
```

第一、第二主成分についてサンプルの分布を可視化

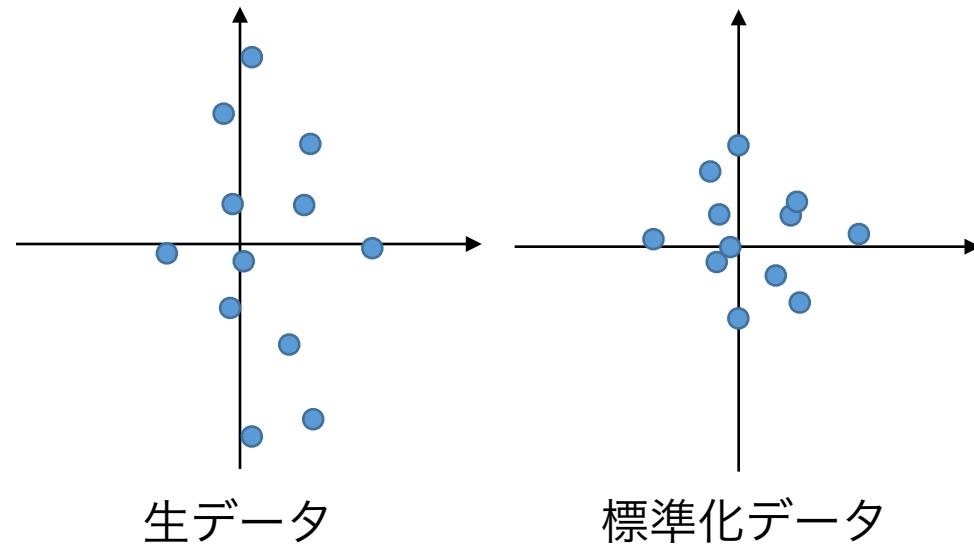
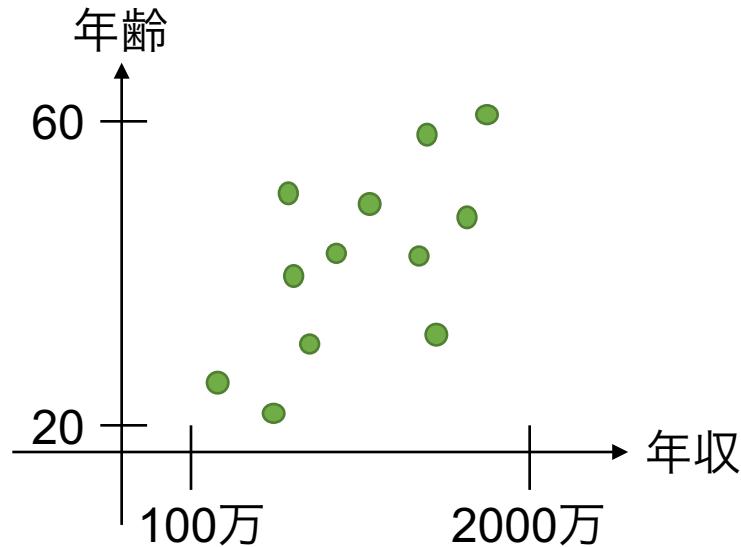
```
In [ ]: def scatter_plot(coords, sample_labels, colors, xlabel=None, ylabel=None, title=''):  
    fig = plt.figure()  
    ax = fig.add_subplot(111)  
    ax.scatter(coords[:, 0], coords[:, 1], color=colors)  
    for i, sample_label in enumerate(sample_labels):  
        ax.annotate(sample_label, xy=(coords[i, :2]), xytext=(10, 10),  
                   textcoords='offset points', color=colors[i],  
                   arrowprops={'arrowstyle': '-', 'edgecolor': colors[i]})  
    ax.set_xlabel(xlabel)  
    ax.set_ylabel(ylabel)  
    ax.set_title(title)  
    plt.show()
```

```
In [ ]: scatter_plot(coords, df.columns, colors, xlabel='PC1', ylabel='PC2')
```

# データの standardization をすべきか

一般的には、PCAの前にそれぞれの変数は標準化（平均を引いて標準偏差で割る）をするべき。

しかし、それはケースバイケース（と個人的に思う）



# データの standardization

In [ ]: # z-score 正規化

```
import sklearn.preprocessing
values = df.transpose().values
scaler = sklearn.preprocessing.StandardScaler(with_mean=True, with_std=True)
std_values = scaler.fit_transform(values)
std_df = pd.DataFrame(std_values.T, index=df.index, columns=df.columns)
```

In [ ]: # 標準化されたデータでPCA実行

```
pca = sklearn.decomposition.PCA()
coords = pca.fit_transform(std_df.transpose().values)
scatter_plot(coords, std_df.columns, colors, xlabel='PC1', ylabel='PC2')
```

1. 次元削減
  - i. 行列分解による次元削減
    - a. 主成分分析 (Principal component analysis; PCA)
    - b. 非負値行列因子分解 (Non-negative matrix factorization; NMF)
    - c. 潜在意味解析 (Latent semantic indexing; LSI)
  - ii. 距離行列の最適化による次元削減
    - a. 多次元尺度構成法 (Multidimensional scaling; MDS)
    - b. その他の多様体学習
2. クラスタリング
  - i. 階層的クラスタリング
  - ii. クラスター「中心」との距離を最適化するクラスタリング
    - a. k-means
    - b. k-medoids
  - iii. 混合正規分布 (Gaussian mixture)

# (Gene × Sample) マトリックスを「行列の積」と考える

$$X = WH$$

$H$

gene_id	batch_1	batch_2	batch_3	chemostat_1	chemostat_2	chemostat_3
gene_0001	0.00	0.73	3.13	0.00	0.00	0.50
gene_0005	0.95	2.80	4.97	4.69	4.37	8.66
gene_0009	1.46	1.24	1.57	3.23	3.78	3.88
gene_0010	8.55	7.96	8.13	159.95	159.68	147.90
gene_0011	17.15	12.23	13.44	147.00	166.23	154.75
gene_0012	181.63	204.34	218.85	257.89	215.60	206.21
gene_0013	90.20	95.14	92.46	60.59	78.58	67.68
gene_0014	33.77	37.64	34.13	34.73	26.97	25.38
gene_0015	20.85	21.19	23.28	44.51	49.07	44.10
gene_0016	14.97	13.26	18.13	45.52	36.81	36.78
gene_0017	10.21	8.34	8.31	1,337.87	1,265.58	1,144.99
gene_0018	79.28	63.09	60.38	112.31	95.41	97.15
gene_0019	17.76	17.43	16.20	24.06	24.14	23.72
gene_0020	96.32	81.33	85.61	168.79	202.96	176.82
gene_0021	17.62	16.56	16.76	36.83	30.75	29.76
gene_0022	13.15	14.12	13.58	19.28	18.46	17.16
gene_0023	74.91	71.71	67.36	117.62	108.65	93.93
gene_0024	54.26	70.47	59.71	152.03	132.92	159.03
gene_0025	506.14	473.79	542.97	419.71	309.40	332.20
gene_0026	29.58	31.38	32.84	41.05	36.73	35.90
gene_0027	224.12	232.68	229.41	173.00	137.48	158.43
gene_0028	28.32	31.08	32.56	33.44	29.78	30.22
gene_0029	46.65	47.58	49.21	42.37	40.41	30.49
gene_0030	88.56	71.44	78.78	109.56	95.19	114.47
gene_0031	3,231.45	3,503.06	3,759.14	1,606.89	2,098.60	2,027.29
gene_0032	19.43	14.34	12.22	9.61	5.12	13.79
gene_0033	15.73	13.27	12.72	18.15	18.65	18.23
gene_0034	101.58	104.38	113.27	78.07	64.10	64.05
gene_0035	161.17	188.18	200.75	144.30	136.18	131.85
gene_0036	18.90	19.46	19.15	27.04	23.80	26.12

$W$

•

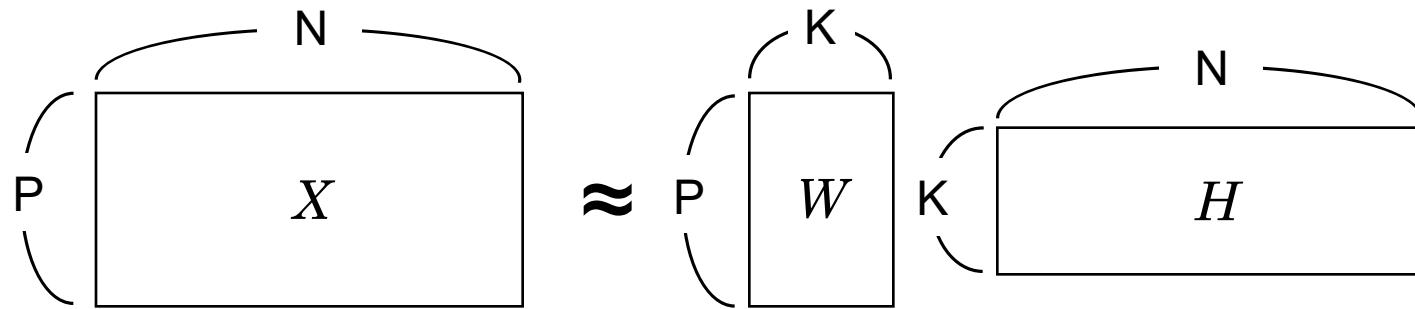
$X$

$W, H$ を見つけ出せれば、  
これらのマトリックスは以下のようにみなせる

各サンプルにおける コア遺伝子セットの 混合割合 $H$						
コア遺伝子セットの組成 $W$						
gene_id	batch_1	batch_2	batch_3	chemostat_1	chemostat_2	chemostat_3
gene_0001	0.00	0.73	3.13	0.00	0.00	0.50
gene_0005	0.95	2.80	4.97	4.69	4.37	8.66
gene_0009	1.46	1.24	1.57	3.23	3.78	3.88
gene_0010	8.55	7.96	8.13	159.95	159.68	147.90
gene_0011	17.15	12.23	13.44	147.00	166.23	154.75
gene_0012	181.63	204.34	218.85	257.89	215.60	206.21
gene_0013	90.20	95.14	92.46	60.59	78.58	67.68
gene_0014	33.77	37.64	34.13	34.73	26.97	25.38
gene_0015	20.85	21.19	23.28	44.51	49.07	44.10
gene_0016	14.97	13.26	18.13	45.52	36.81	36.78
gene_0017	10.21	8.34	8.31	1,337.87	1,265.58	1,144.99
gene_0018	79.28	63.09	60.38	112.31	95.41	97.15
gene_0019	17.76	17.43	16.20	24.06	24.14	23.72
gene_0020	96.32	81.33	85.61	168.79	202.96	176.82
gene_0021	17.62	16.56	16.76	36.83	30.75	29.76
gene_0022	13.15	14.12	13.58	19.28	18.46	17.16
gene_0023	74.91	71.71	67.36	117.62	108.65	93.93
gene_0024	54.26	70.47	59.71	152.03	132.92	159.03
gene_0025	506.14	473.79	542.97	419.71	309.40	332.20
gene_0026	29.58	31.38	32.84	41.05	36.73	35.90
gene_0027	224.12	232.68	229.41	173.00	137.48	158.43
gene_0028	28.32	31.08	32.56	33.44	29.78	30.22
gene_0029	46.65	47.58	49.21	42.37	40.41	30.49
gene_0030	88.56	71.44	78.78	109.56	95.19	114.47
gene_0031	3,231.45	3,503.06	3,759.14	1,606.89	2,098.60	2,027.29
gene_0032	19.43	14.34	12.22	9.61	5.12	13.79
gene_0033	15.73	13.27	12.72	18.15	18.65	18.23
gene_0034	101.58	104.38	113.27	78.07	64.10	64.05
gene_0035	161.17	188.18	200.75	144.30	136.18	131.85
gene_0036	18.90	19.46	19.15	27.04	23.80	26.12
	•		•		•	
						X

# 非負値行列因子分解 (Non-negative matrix factorization; NMF)

非負値行列を **2** つの非負値行列の積に分解するアルゴリズム



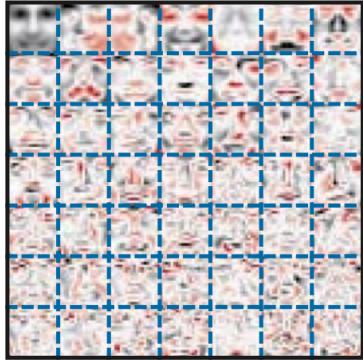
$K$ の大きさによって、元のデータ $X$ をどれだけ再現できるかを調節できる  
 $K$ が小さいと粗い近似にしかならないが、 $K$ が大きすぎると分解の意味が無い

つまり $K$ を許容できる範囲で小さめにとれば、次元圧縮ができる

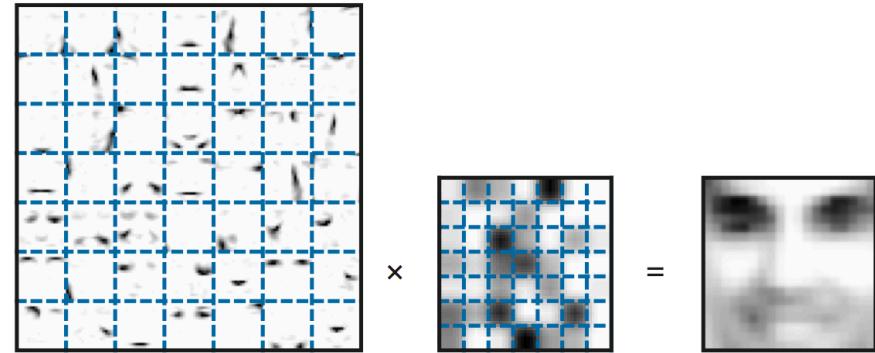
# 主成分分析との違い

↓ 基底画像  $\times$  そのサンプルの混合割合 = 再現されたデータ  
(赤=負値、黒=正值、白=ゼロ)

PCA



NMF



(D. D. Lee and H. S. Seung, "Learning the parts of objects with nonnegative matrix factorization," *Nature*, vol. 401, pp. 788–791, 1999.)

PCAは、基底を足したり引いたりする操作が可能なので、各基底の解釈が難しい。

NMFは、本質的に各基底の足し算しかできない。だがその制約によって、**局所的な特徴**を表現することができる。  
(NMFの基底画像は、「目」「鼻」「ひげ」「しわ」などとそれぞれ解釈できる)

「非負値行列データの構成要素もまた、非負値行列であるべき」という考え方方がNMF  
PCAなどで現れる負の係数は解釈が難しいが、NMFの基底は構成要素としての解釈が比較的容易。

# NMFのアルゴリズム

- 1) Kを設定
- 2) M×K行列 W、K×N行列 H を用意し、各値をランダムに設定する
- 3) 行列積  $WH = X_{\text{hat}}$  を計算する
- 4) 元の行列 X と  $X_{\text{hat}}$  との差をなんらかの距離尺度で計算する
- 5) 距離が収束していなかった場合、距離尺度に応じた更新式で W と H の各値を更新し、3) に戻る

Frobeniusノルム最小化の場合の更新式

$$w_{ik} \leftarrow w_{ik} \frac{(XH^T)_{ik}}{(WHH^T)_{ik}} \quad h_{kj} \leftarrow h_{kj} \frac{(W^T X)_{kj}}{(W^T WH)_{kj}}$$

cf) C言語実装：<https://github.com/khigashi1987/EU-NMF>

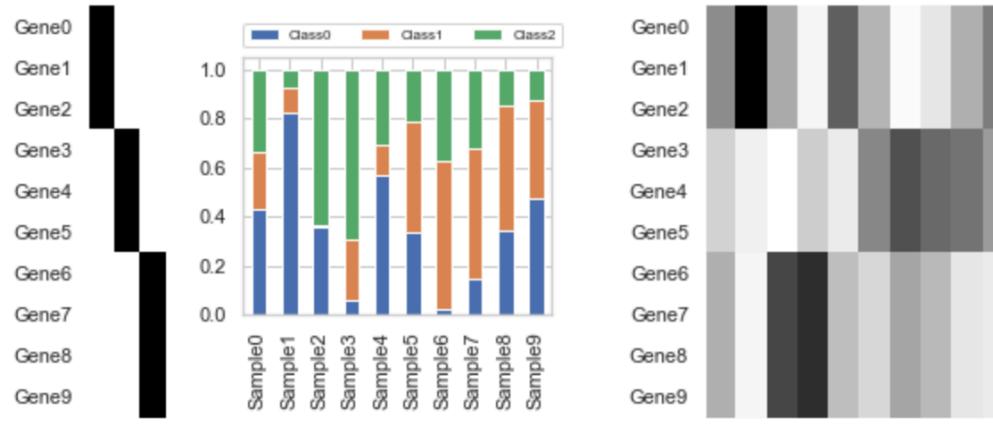
# 擬似データによるNMFのデモ

```
print('Original')
# class-0 は最初の3個の遺伝子を発現
genes = np.array([[1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
                  # class-1 はまんなか3個の遺伝子を発現
                  [0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0],
                  # class-2 はうしろ4個の遺伝子を発現
                  [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0]]).T
# 各サンプルがclass-0, class-1, class2をどういう割合で持っているか
samples = np.random.dirichlet(alpha=[1.0]*3, size=10).T
# 遺伝子発現テーブルはその掛け算で決まっている(と仮定する)
original_expression_data = np.dot(genes, samples)
plot_W_H_X(genes, samples, original_expression_data)

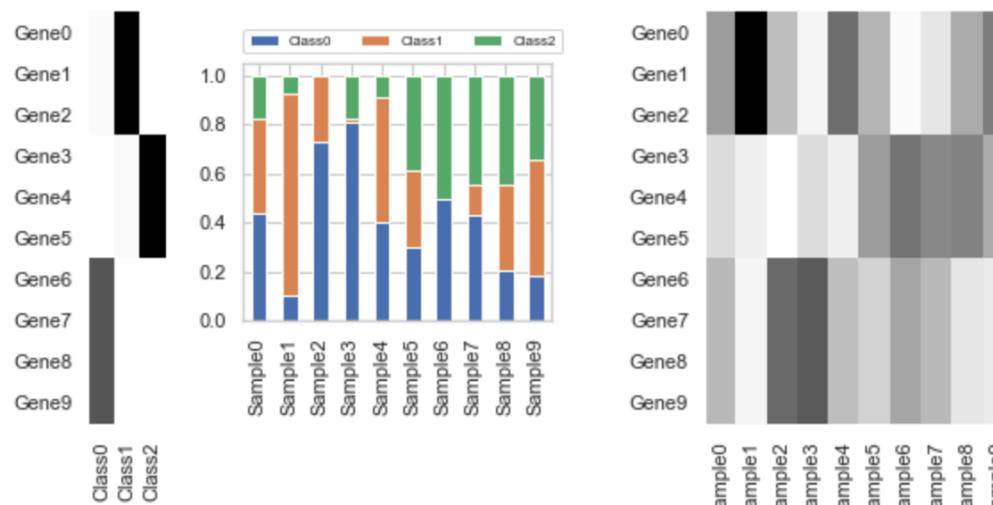
print('Reconstructed')
# 遺伝子発現テーブルだけを使って、クラスごとの発現パターンベクトル、サンプルごとのクラス割合を復元する
model = sklearn.decomposition.NMF(n_components=3)
W = model.fit_transform(original_expression_data)
W /= W.sum(axis=0)
H = model.components_
H /= H.sum(axis=0)
X = np.dot(W, H)
plot_W_H_X(W, H, X)
```

# 擬似データによるNMFのデモ

Original



Reconstructed



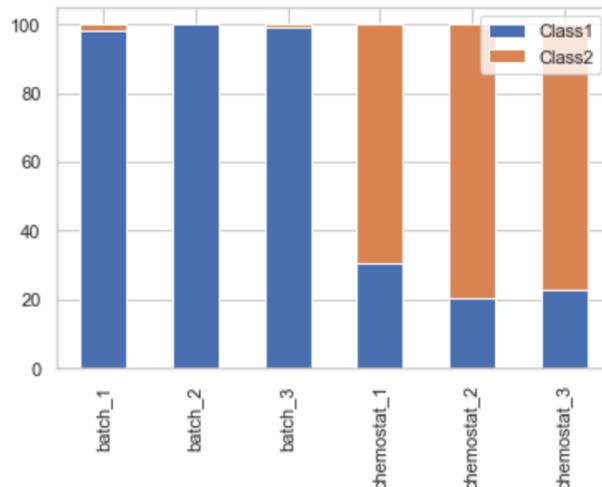
# NMF実行

```
In [ ]: model = sklearn.decomposition.NMF(n_components=2)
W = model.fit_transform(df.values)
H = model.components_
```

それぞれの因子が各サンプルにどれほど寄与しているか

```
In [16]: # 次元削減としての利用
H_percentage = 100.0 * H / H.sum(axis=0)
pd.DataFrame(H_percentage.T, index=df.columns, columns=['Class1', 'Class2']).plot.bar(stacked=True)
#各サンプル、もっとも値の高い要素に割り当てることでクラスタリングの代わりとしてしまうこともある。
```

```
Out[16]: <matplotlib.axes._subplots.AxesSubplot at 0x1a1b35cf98>
```



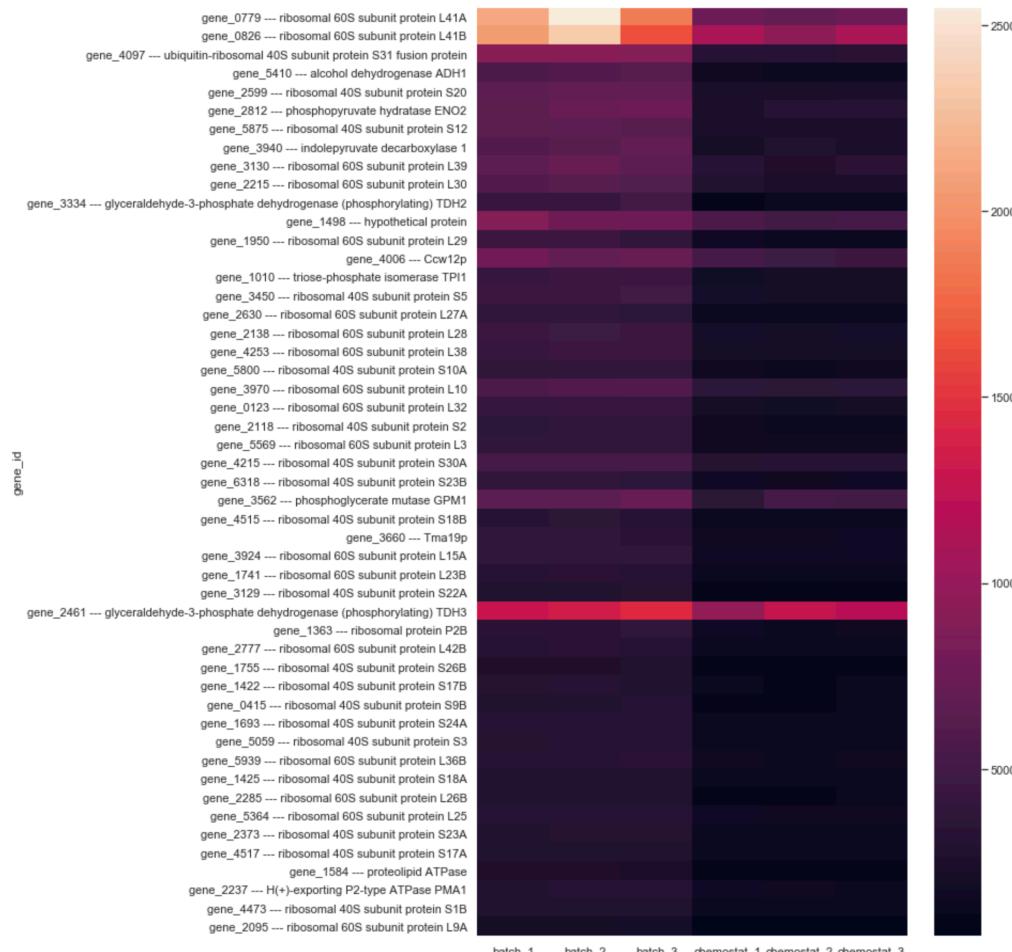
# NMF実行

それぞれの因子はどんな遺伝子セットで構成されているか

```
In [20]: # 因子1に強く寄与する遺伝子
topN = 50
top_factor1 = df.index[ np.argsort(W[:,0] - W[:,1])[:-1][:-topN] ]
gene_labels = top_factor1 + ' --- ' + gene_products.loc[top_factor1, 'product']

fig = plt.figure(figsize=(9,16))
sns.heatmap(df.loc[top_factor1, :], yticklabels=gene_labels)
```

Out[20]: <matplotlib.axes.\_subplots.AxesSubplot at 0x1a1d5ff438>



1. 次元削減
  - i. 行列分解による次元削減
    - a. 主成分分析 (Principal component analysis; PCA)
    - b. 非負値行列因子分解 (Non-negative matrix factorization; NMF)
    - c. 潜在意味解析 (Latent semantic indexing; LSI)
  - ii. 距離行列の最適化による次元削減
    - a. 多次元尺度構成法 (Multidimensional scaling; MDS)
    - b. その他の多様体学習
2. クラスタリング
  - i. 階層的クラスタリング
  - ii. クラスター「中心」との距離を最適化するクラスタリング
    - a. k-means
    - b. k-medoids
  - iii. 混合正規分布 (Gaussian mixture)

# 潜在意味解析 (Latent semantic indexing; LSI)

Latent semantic analysis (LSA) ともよばれる。

NMFと類似した行列分解を、非負値行列じゃない（正規化などを施した）

行列に対して実行したい

→ LSI

LSIは2つのステップからなる

## 1. TF-IDF変換

LSIにおいて変数を標準化する手法

TF (Term Frequency) と IDF (Inverse Document Frequency) の 2 つの重みで変数を調整する

## 2. 特異値分解 (Singular Value Decomposition; SVD)

固有値分解と似ているが、固有値分解が正方行列に対してのみ可能なのに対して、細長い行列でも可能な分解。

PCAではデータの共分散行列を分解したが、SVDではデータの行列そのものが分解できる。

# TF-IDF変換

TF: Term frequency. サンプルごとの変換。

サンプル内でその変数がどの程度の割合で存在しているか。

TPM正規化されている場合は特に変換が必要ないかもしれないが、今回はサンプル内でのmax TPMに対する割合とする。

IDF: Inverse document frequency. 変数ごとの変換。

その変数がどれほどサンプル間で共通して存在しているか。

全サンプルで共通して存在する変数は重要度が低いとみなして小さな重みにする。  
よく使われるのは、 $\log\left(\frac{\text{全サンプル数}}{\text{その変数が}>0\text{のサンプル数}}\right)$

サンプル  $j$  の変数  $i$  は以下のように変換される

$$TFIDF(x_{ij}) = TF(x_{ij}) \times IDF(x_j)$$

```
# TF-IDF 変換
# TF ... Term Frequency いくつかの流儀がある。ここではサンプルごとのmaxに対する割合
TF = df.values / df.values.max(axis=0)
# IDF ... Inverse Document Frequency これもいくつかの流儀あり。
n_samples = len(df.columns)
IDF = np.log2(1.0 + (float(n_samples) / df.values.astype(bool).sum(axis=1)))
# TF-IDF
TFIDF = TF * IDF[:, np.newaxis]
df_tfidf = pd.DataFrame(TFIDF, index=df.index, columns=df.columns)
```

# 特異値分解 (SVD)

$$P \begin{pmatrix} N \\ X \end{pmatrix}$$

=

$$P \begin{pmatrix} K \\ U \end{pmatrix}$$

左特異ベクトル

特異値が並んだ対角行列

右特異ベクトル

$$\begin{pmatrix} K \\ \sigma \end{pmatrix}$$

右側の2つの行列 ( $\sigma$ と $V$ )  
を掛け合わせてしまえば、  
NMFのときと同様、  
遺伝子をまとめた因子 ( $U$ ) と、  
サンプルごとの重み行列 ( $\sigma V$ )  
の積として表現できる

$$= P \begin{pmatrix} K \\ U \end{pmatrix} \begin{pmatrix} K \\ \sigma V \end{pmatrix}$$

# SVD実行

scikit-learnにもSVDがある（`sklearn.decomposition.TruncatedSVD`）が、どの変数にどの行列が入ってるかわかりにくいので、今回はnumpyのSVDを使う。  
sklearn版はTruncated SVDと呼ばれる近似手法で、特にテーブルのサイズが巨大なときに高速に計算できる。

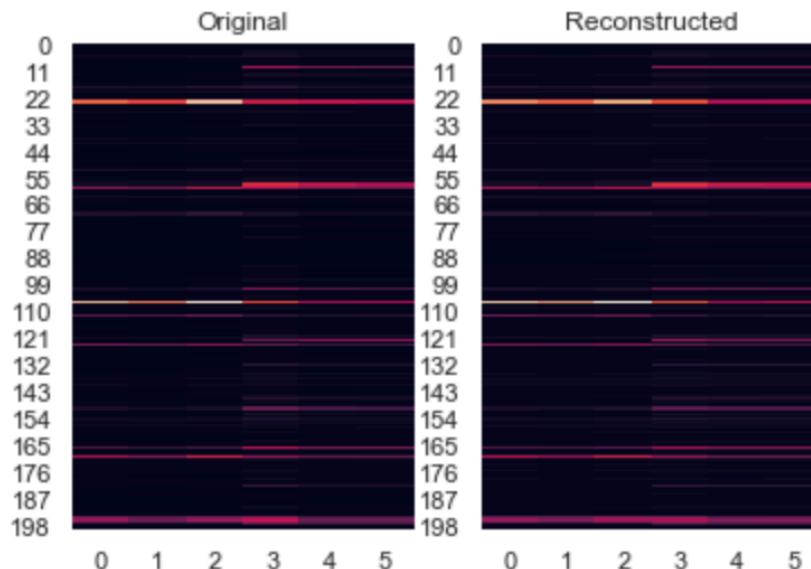
```
# SVD (Singular Value Decomposition; 特異値分解)を実行
import numpy.linalg
U, Sig, V = np.linalg.svd(df_tfidf.values, full_matrices=False)
Weights = np.dot(np.diag(Sig), V)

# U, Weights は元の行列を分解したものなので、UとWeightsの掛け算は元の行列を近似
np.allclose(df_tfidf.values, np.dot(U, Weights))
```

# 最初のいくつかの特異値で次元圧縮

```
U_reduced = U[:, :2]
W_reduced = np.dot(np.diag(Sig[:2]), V[:2, :])
```

```
fig = plt.figure()
ax1 = fig.add_subplot(121)
sns.heatmap(df_tfidf.values[:200,:], cbar=False, ax=ax1)
ax1.set_title('Original')
ax2 = fig.add_subplot(122)
sns.heatmap(np.dot(U_reduced, W_reduced)[:200,:], cbar=False, ax=ax2)
ax2.set_title('Reconstructed')
plt.show()
```

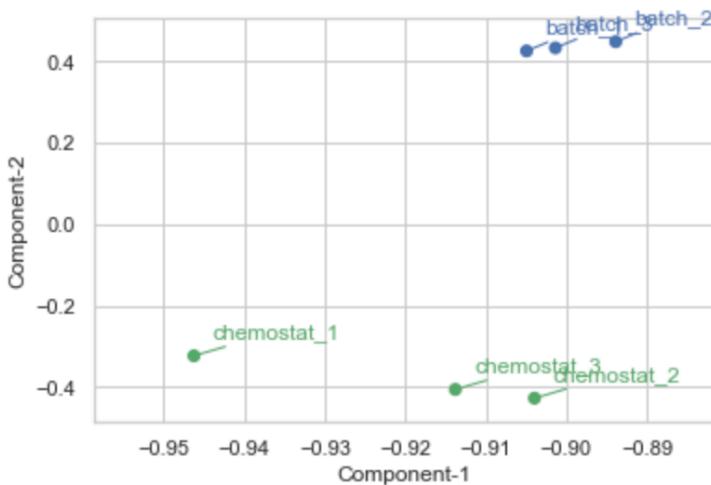


←最初の2つの特異ベクトルだけ取り出して  
データ行列を再構成すると、  
値のパターンがだいたい同じ

# 次元圧縮

=> 2群の分離に強く寄与する遺伝子はなにか？

```
# 長さ1のベクトルにノーマライズする
W_norm = W_reduced / np.sqrt((W_reduced**2).sum(axis=0))
# 2つのcomponentの「重み」空間でサンプルをプロット
scatter_plot(W_norm.T, df_tfidf.columns, colors, xlabel='Component-1', ylabel='Component-2')
```





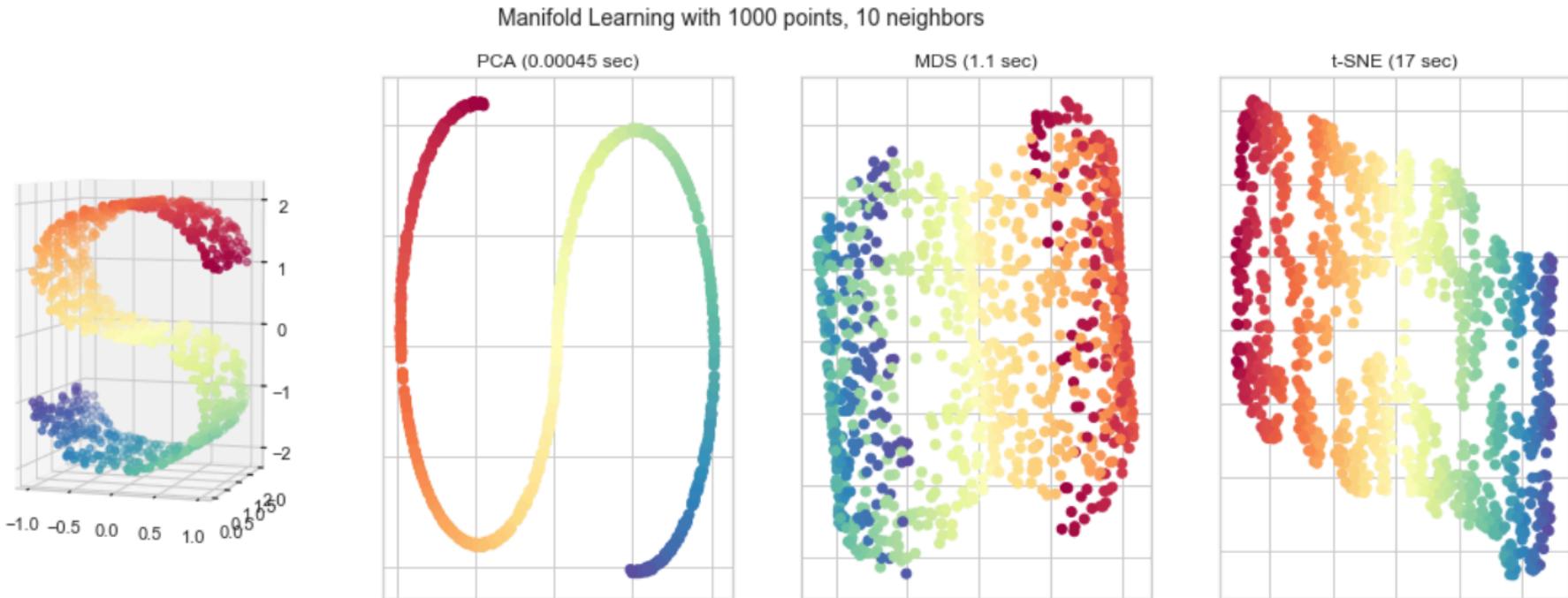
1. 次元削減
  - i. 行列分解による次元削減
    - a. 主成分分析 (Principal component analysis; PCA)
    - b. 非負値行列因子分解 (Non-negative matrix factorization; NMF)
    - c. 潜在意味解析 (Latent semantic indexing; LSI)
  - ii. 距離行列の最適化による次元削減
    - a. 多次元尺度構成法 (Multidimensional scaling; MDS)
    - b. その他の多様体学習
2. クラスタリング
  - i. 階層的クラスタリング
  - ii. クラスター「中心」との距離を最適化するクラスタリング
    - a. k-means
    - b. k-medoids
  - iii. 混合正規分布 (Gaussian mixture)

# 非線形次元圧縮

前節までの行列分解による手法は、高次元空間におけるデータの分布の線形な関係性だけを抽出していた（変数の一次結合で合成変数を作ったり、なんらかのベクトルの「重ね合わせ」でデータを表現したり）

だが、データは高次元空間で、ぐにゃぐにゃに曲がった部分空間に分布していたり、螺旋状に渦巻いていたりするかもしれない。そういうわけで、線形変換では捉えられない特徴を抽出するための手法が、本節の「距離行列に基づく次元圧縮」。

ここでは MDS と t-SNE を扱う。



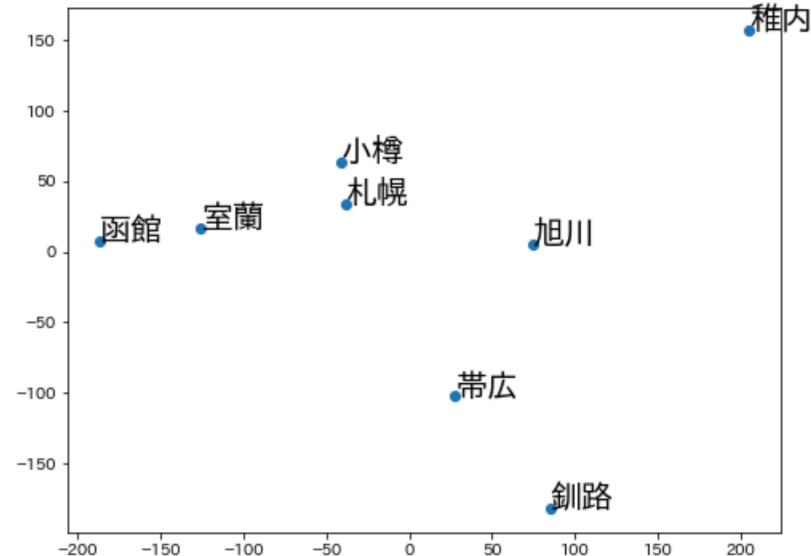
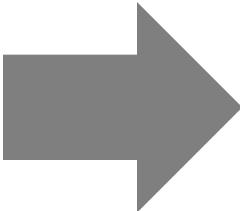
# 多次元尺度構成法 (Multidimensional scaling; MDS)

サンプル間の「距離行列」だけを受け取り（高次元データそのものは扱わない）、低次元空間におけるそれらの座標を出力する。

その際、**低次元空間における距離行列が、元の空間における距離行列をできるだけ近似する**ようにして低次元空間座標を推定する。

	札幌	旭川	稚内	釧路	帯広	室蘭	函館	小樽
札幌	0.0	115.0	274.0	249.0	152.0	88.0	152.0	30.0
旭川	115.0	0.0	202.0	188.0	118.0	200.0	263.0	130.0
稚内	274.0	202.0	0.0	358.0	313.0	360.0	413.0	266.0
釧路	249.0	188.0	358.0	0.0	98.0	290.0	330.0	277.0
帯広	152.0	118.0	313.0	98.0	0.0	195.0	240.0	180.0
室蘭	88.0	200.0	360.0	290.0	195.0	0.0	64.0	98.0
函館	152.0	263.0	413.0	330.0	240.0	64.0	0.0	159.0
小樽	30.0	130.0	266.0	277.0	180.0	98.0	159.0	0.0

入力：距離行列



出力：低次元空間座標

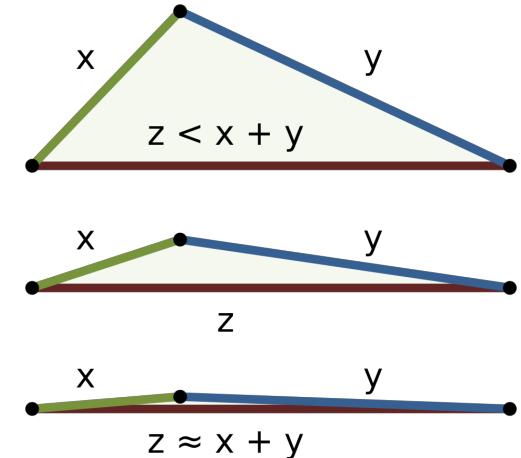
# 距離関数

MDSは任意の**距離関数**で測ったサンプル間距離行列を入力とできる。

距離関数  $d( , )$

2つのベクトルを引数として、以下の条件を満たす実数を返す関数

1.  $d(x, y) \geq 0$  (非負)
2.  $d(x, x) = 0$  (同一)
3.  $d(x, y) = d(y, x)$  (対称)
4.  $d(x, z) + d(y, z) \geq d(x, y)$  (三角不等式)



様々な距離関数

- ユークリッド距離 :

$$d(u, v) = \sqrt{\sum_i (u_i - v_i)^2}$$

- コサイン距離

$$d(u, v) = 1 - \frac{u \cdot v}{\|u\|_2 \|v\|_2}$$

- マンハッタン距離

$$d(u, v) = \sum_i |u_i - v_i|$$

- UniFrac距離 (マイクロバイオームデータ)

<https://ja.wikipedia.org/wiki/三角不等式>

しばしば使われる

Bray-Curtis非類似度指標は距離関数ではない  
(三角不等式を満たさない) ため注意

# いろいろなMDS

- Classical MDS (= Principal Coordinate Analysis; PCoA 主座標分析)  
距離行列の固有値分解に基づく手法  
距離ではない指標でPCoAを使うのは間違い（実用上は、固有値が負にならないベクトルを取り出すなら問題にはならないと言われてはいるが...）
- Metric MDS  
scikit-learn で実装されているのは、 Stress majorization （SMACOF algorithm）によって iterative に距離行列の差を最小化する手法
- Non-metric MDS (nMDS 非計量多次元尺度構成法)  
入力する「距離行列」（非類似度行列）が、距離関数の定義を満たしていない関数で計算されていた場合に適用可能なMDS。元の空間の距離行列と低次元空間の距離行列の値そのものを比較するのではなく、距離行列中の値の大小関係が一致するように最適化する（Isotonic regressionなど）。Bray-Curtis指標でも計算できる。

# MDS実行

```
import sklearn.manifold
```

```
# デフォルトではユークリッド距離でサンプル間距離行列を計算。この場合、数学的には主成分分析と等価。  
# ただしscikit-learnのMDS実装はiterativeに最適化するmetric MDSであるため(classical MDS=PCoAは固有値分解に基づく手法)。  
# ランダムな初期値の影響で実行のたびに結果が若干変わる  
mds = sklearn.manifold.MDS(n_components=2, dissimilarity='euclidean')  
coords = mds.fit_transform(std_df.transpose().values)  
scatter_plot(coords, std_df.columns, colors)
```

自分が計算した距離行列を入力にしてMDSをかけることもできる。

```
# 距離行列の計算  
from scipy.spatial.distance import pdist, squareform  
distance_matrix = squareform(pdist(std_df.transpose().values))  
print(distance_matrix)  
  
#自分で距離行列を作る場合(上の一行と同じ計算結果)  
##from scipy.spatial.distance import euclidean  
##values = std_df.transpose().values  
##distance_matrix_2 = []  
##for i in range(values.shape[0]):  
##    vec = []  
##    for j in range(values.shape[0]):  
##        vec.append(euclidean(values[i, :], values[j, :]))  
##    distance_matrix_2.append(vec)  
##print(np.array(distance_matrix_2))
```

```
mds = sklearn.manifold.MDS(n_components=2, dissimilarity='precomputed')  
coords = mds.fit_transform(distance_matrix)  
scatter_plot(coords, std_df.columns, colors)
```

# いろいろな距離関数で作った距離行列でMDS実行

Scipyの pdist 関数はいろんな距離関数を計算してくれる

```
# 相関係数で計算した距離(1 - 相関係数)
distance_matrix = squareform(pdist(std_df.transpose().values, 'correlation'))
mds = sklearn.manifold.MDS(n_components=2, dissimilarity='precomputed')
coords = mds.fit_transform(distance_matrix)
scatter_plot(coords, std_df.columns, colors, title='Correlation')

# Jaccard距離
distance_matrix = squareform(pdist(std_df.transpose().values, 'jaccard'))
mds = sklearn.manifold.MDS(n_components=2, dissimilarity='precomputed')
coords = mds.fit_transform(distance_matrix)
scatter_plot(coords, std_df.columns, colors, title='Jaccard distance')

# マンハッタン距離
distance_matrix = squareform(pdist(std_df.transpose().values, 'cityblock'))
mds = sklearn.manifold.MDS(n_components=2, dissimilarity='precomputed')
coords = mds.fit_transform(distance_matrix)
scatter_plot(coords, std_df.columns, colors, title='Manhattan distance')
```

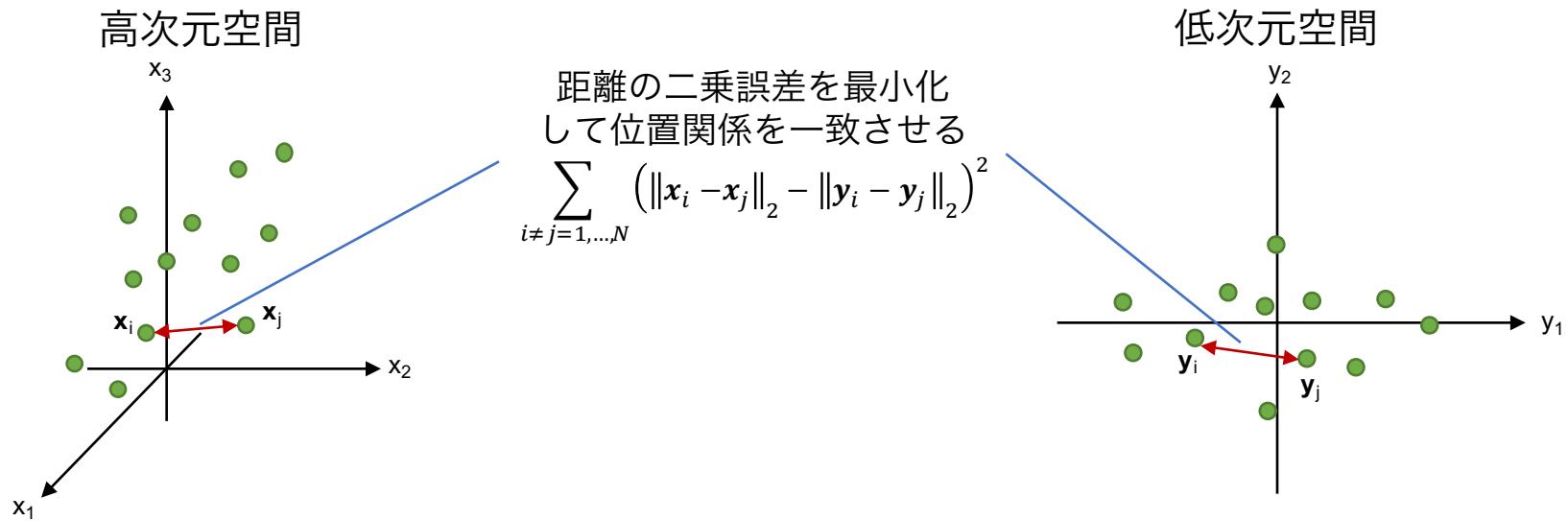
nMDSを計算するときは、sklearnのMDS関数でmetricオプションをFalseにする

```
# Bray-Curtis 非類似度指標による計算
distance_matrix = squareform(pdist(std_df.transpose().values, 'braycurtis'))
nmnds = sklearn.manifold.MDS(n_components=2, metric=False, dissimilarity='precomputed')
coords = nmnds.fit_transform(distance_matrix)
scatter_plot(coords, std_df.columns, colors, title='Bray-Curtis dissimilarity')
```

1. 次元削減
  - i. 行列分解による次元削減
    - a. 主成分分析 (Principal component analysis; PCA)
    - b. 非負値行列因子分解 (Non-negative matrix factorization; NMF)
    - c. 潜在意味解析 (Latent semantic indexing; LSI)
  - ii. 距離行列の最適化による次元削減
    - a. 多次元尺度構成法 (Multidimensional scaling; MDS)
    - b. その他の多様体学習
2. クラスタリング
  - i. 階層的クラスタリング
  - ii. クラスター「中心」との距離を最適化するクラスタリング
    - a. k-means
    - b. k-medoids
  - iii. 混合正規分布 (Gaussian mixture)

# 高次元の「距離」と低次元の「距離」をいかに比較するか？

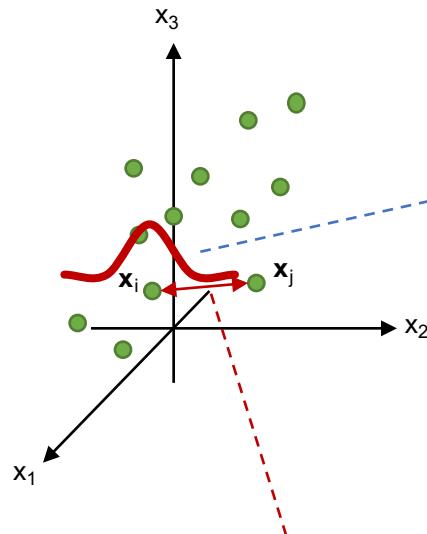
metric Multidimensional scaling



# 高次元の「距離」と低次元の「距離」をいかに比較するか？

Stochastic Neighbor Embedding (SNE; 確率的近傍埋め込み)

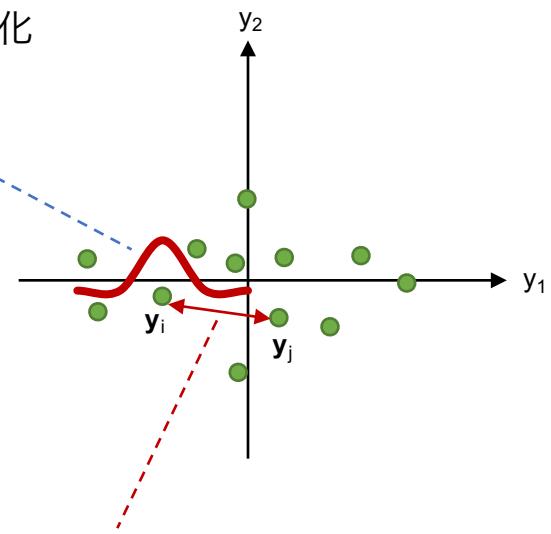
高次元空間



サンプル*i*とサンプル*j*の類似性を  
サンプル*i*を中心として減衰する  
正規分布の確率で測る

$$p_{j|i} = \frac{\exp(-\|x_i - x_j\|_2^2 / 2\sigma^2)}{\sum_{k \neq i} \exp(-\|x_i - x_k\|_2^2 / 2\sigma^2)}$$

低次元空間



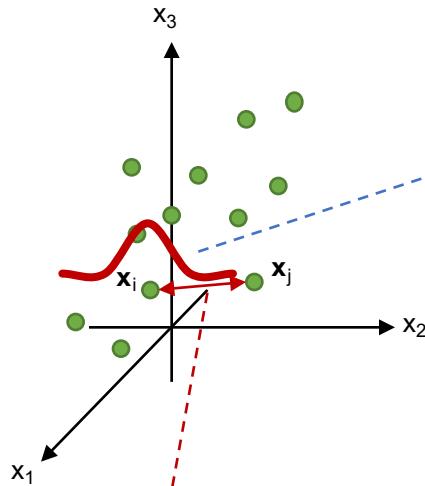
低次元側でも同様、類似性を  
正規分布の確率値とする

$$q_{j|i} = \frac{\exp(-\|y_i - y_j\|_2^2)}{\sum_{k \neq i} \exp(-\|y_i - y_k\|_2^2)}$$

# 高次元の「距離」と低次元の「距離」をいかに比較するか？

## t-Distributed Stochastic Neighbor Embedding (t-SNE; t分布型確率的近傍埋め込み)

高次元空間



サンプル*i*とサンプル*j*の類似性を  
サンプル*i*を中心として減衰する  
正規分布の確率で測る  
ただし*i*と*j*について対称

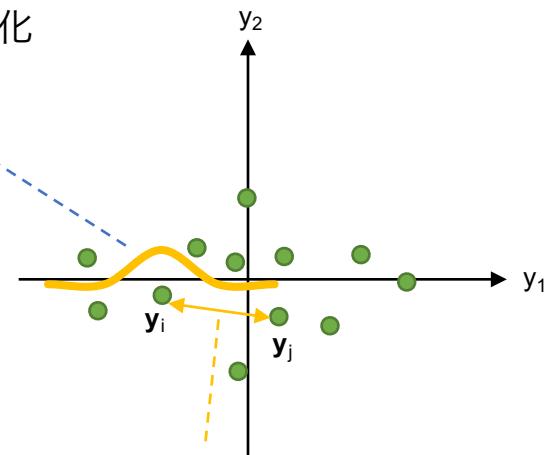
$$p_{ij} = \frac{\exp(-\|x_i - x_j\|_2^2 / 2\sigma^2)}{\sum_{k \neq l} \exp(-\|x_k - x_l\|_2^2 / 2\sigma^2)}$$

2018/11/21

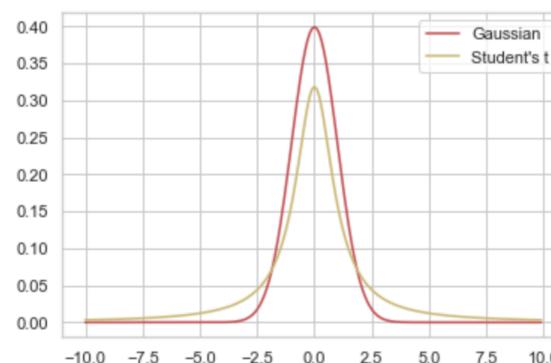
カルバック・ライブラ一情報量を最小化  
して2つの確率分布を近づける

$$KL(P \parallel Q) = \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}}$$

低次元空間



低次元側では、t分布で類似性を定義  
これにより、高次元空間と低次元空間  
の体積の違いによる影響が緩和される



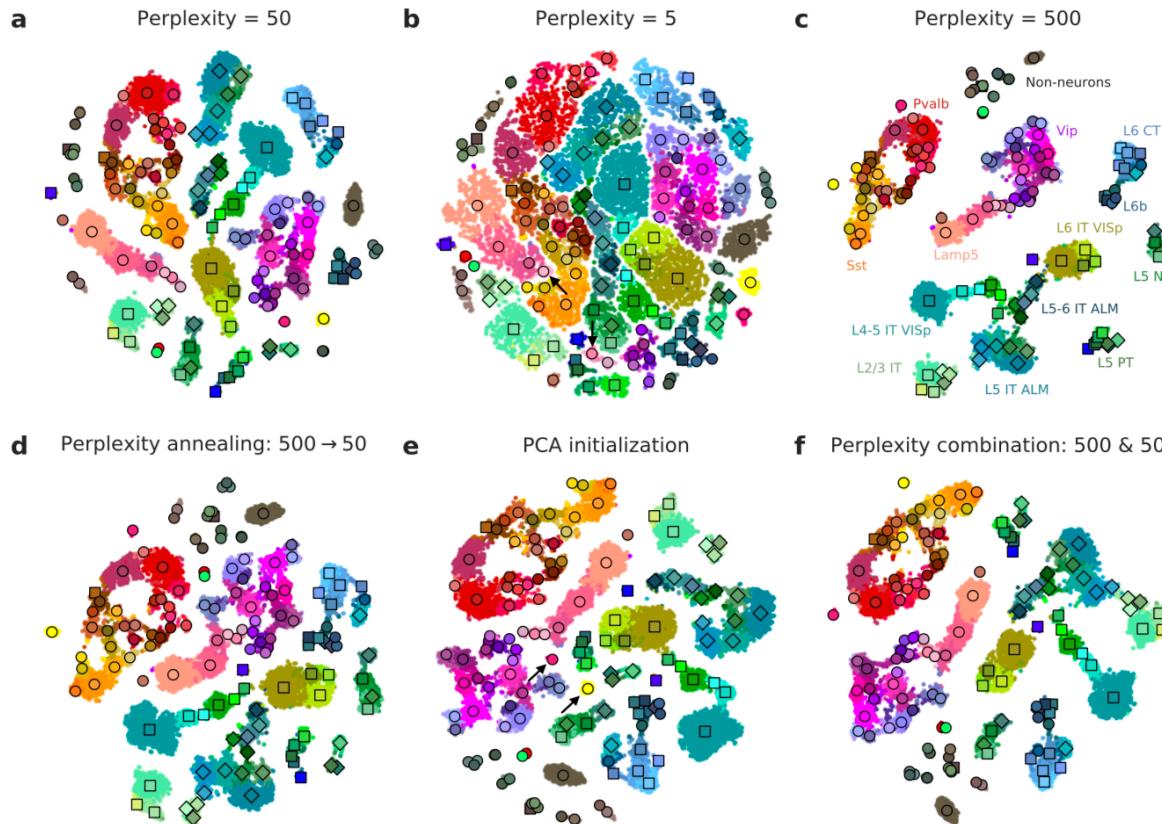
$$q_{ij} = \frac{(1 + \|y_i - y_j\|_2)^{-1}}{\sum_{k \neq l} (1 + \|y_k - y_l\|_2)^{-1}}$$

t分布は正規分布よりも「裾が重い」分布。  
そのため、高次元空間でわりと近い点が  
低次元空間では遠くに配置される。

# t-SNE実行

```
tsne = sklearn.manifold.TSNE(n_components=2)
coords = tsne.fit_transform(std_df.transpose().values)
scatter_plot(coords, std_df.columns, colors, title='t-SNE')
```

# パラメータや初期値の影響

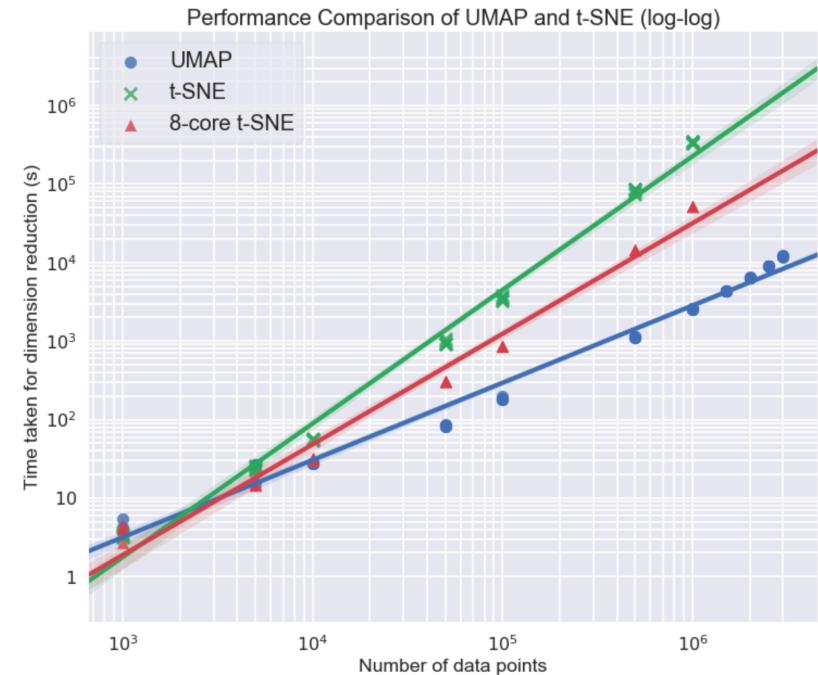
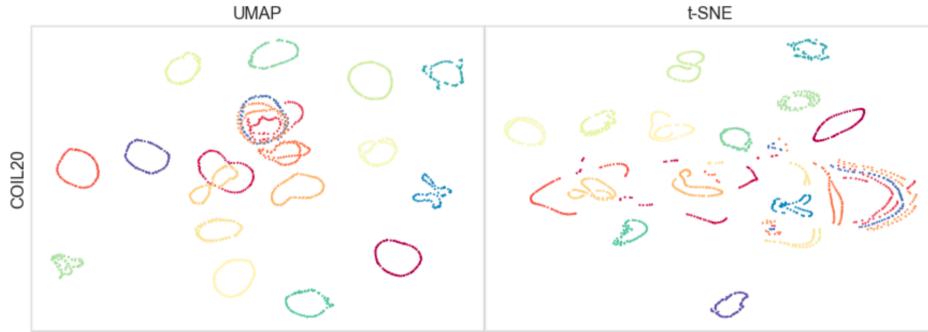


Kobak, Dmitry, and Philipp Berens.  
"The art of using t-SNE for single-cell transcriptomics."  
*bioRxiv* (2018): 453449.

# UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction

McInnes, Leland, and John Healy. arXiv:1802.03426 (2018).

t-SNEと同等の次元圧縮を、大規模なデータで高速に計算する最新の次元圧縮手法。  
今後流行るかも。



condaやpipで簡単にインストールして使える。

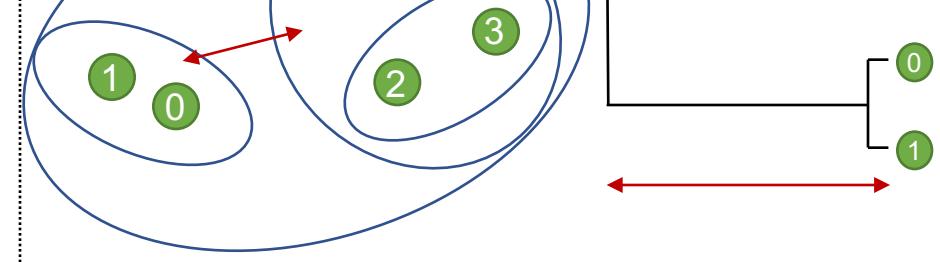
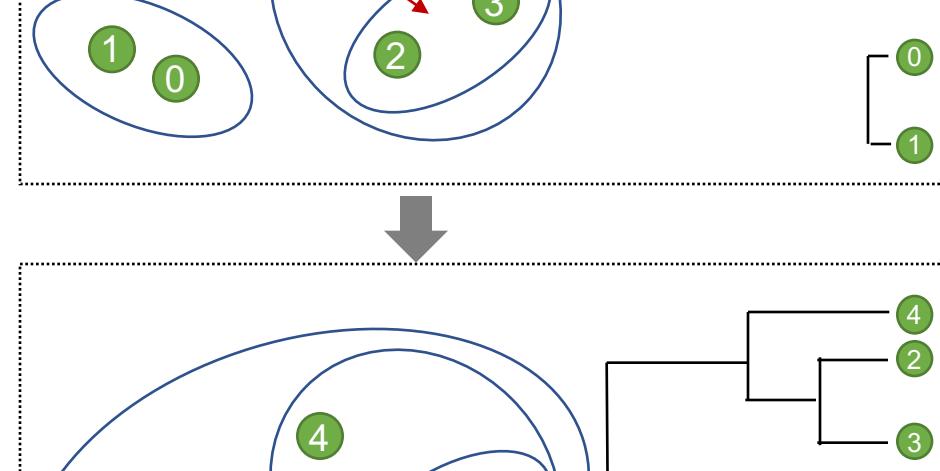
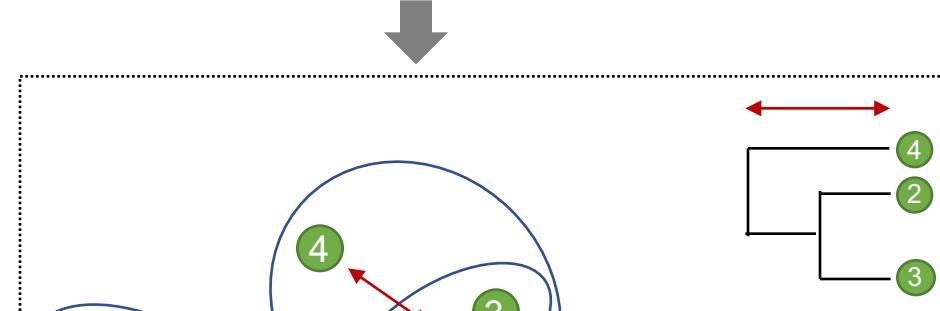
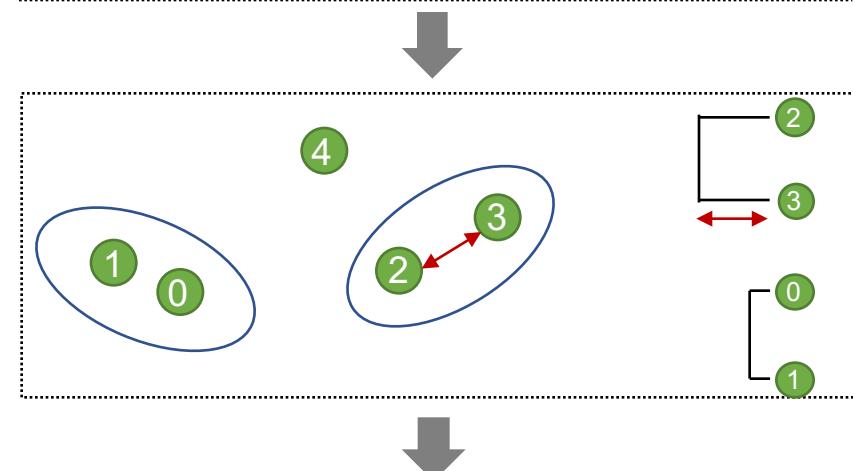
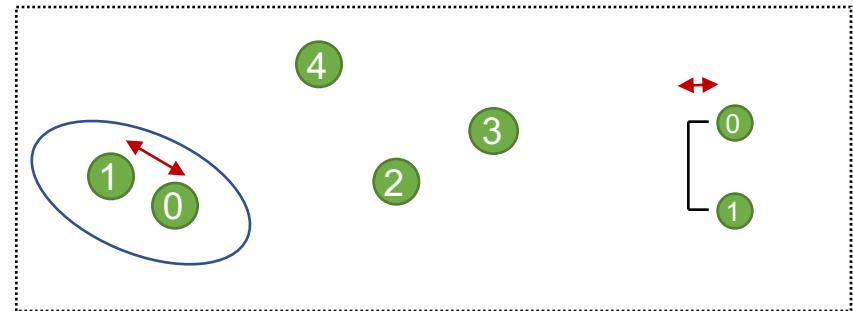
```
#conda install -c conda-forge umap-learn
import umap
reducer = umap.UMAP(n_components=2, n_neighbors=2)
coords = reducer.fit_transform(std_df.transpose().values)
scatter_plot(coords, std_df.columns, colors, title='UMAP')
```

1. 次元削減
  - i. 行列分解による次元削減
    - a. 主成分分析 (Principal component analysis; PCA)
    - b. 非負値行列因子分解 (Non-negative matrix factorization; NMF)
    - c. 潜在意味解析 (Latent semantic indexing; LSI)
  - ii. 距離行列の最適化による次元削減
    - a. 多次元尺度構成法 (Multidimensional scaling; MDS)
    - b. その他の多様体学習
2. クラスタリング
  - i. 階層的クラスタリング
  - ii. クラスター「中心」との距離を最適化するクラスタリング
    - a. k-means
    - b. k-medoids
  - iii. 混合正規分布 (Gaussian mixture)

# 階層的クラスタリング

ボトムアップ型のクラスタリング手法。

サンプル全体の中からもっとも距離が近い（類似性が高い）ペアを探して併合、  
その次に距離が近いペアを探して併合、、、と順次併合していく



# 階層的クラスタリングの実行

seabornのclustermapでは、pandasのデータフレームを与えると  
両側（今回の場合、サンプル、遺伝子それぞれ）で階層的クラスタリングを実行する

```
sns.clustermap(df, method='average', metric='correlation', col_colors=colors, figsize=(3, 16))
```

クラスタリング結果と各サンプルの値の関係性を比較するため、  
全体を対数変換したデータについても階層的クラスタリングと  
ヒートマップを表示してみる

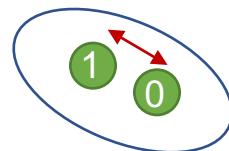
```
log_transform = lambda x: np.log10(x + 1.0)
sns.clustermap(df.apply(log_transform), method='average', metric='correlation', col_colors=colors, figsize=(3, 16))
```

# 階層的クラスタリングにおける「距離」

```
sns.clustermap(df, method='average', metric='correlation', col_colors=colors, figsize=(3, 16))
```

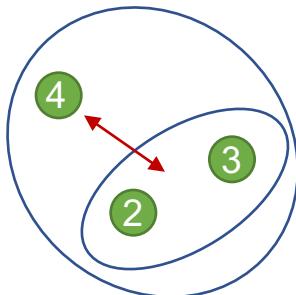
階層的クラスタリングでは、2種類の「距離」をどのように計算するか  
考える必要がある。

## 1. サンプル間の距離



clustermapにおける metric オプション。  
サンプル間の距離関数の設定。MDSのときの議論と同様。  
seabornのclustermapでは内部的にscipyのpdistを呼んでいる  
ので、pdistに実装されている距離関数を文字列で指定できる。

## 2. クラスタ間の距離

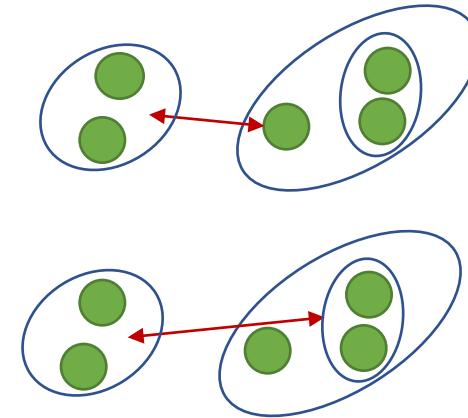


clustermapにおける method オプション。  
クラスタ間の距離関数の設定。  
サンプル間の距離は単純に各サンプルのデータで計算すればよ  
かったが、いくつかのサンプルが併合されたクラスタとサンプ  
ルの距離、あるいはクラスタとクラスタの距離はどのように定  
義すればいいのか？いろいろな手法がある。

# クラスタ間の距離

seabornのclustermapでは、内部的にscipyのscipy.cluster.hierarchy.linkageを呼んでいるため、scipyで定義されているクラスタ間距離指標を適用可能。  
たとえば以下のようなクラスタ間距離の計算手法がある。

1. Single linkage;  $D(U, V) = \min(\text{dist}(U_i, V_j))$   
2つのクラスタからなる距離の組み合わせの中で  
もっとも近いものをクラスタの距離とする方法。
2. Complete linkage;  $D(U, V) = \max(\text{dist}(U_i, V_j))$   
2つのクラスタからなる距離の組み合わせの中で  
もっとも遠いものをクラスタの距離とする方法。
3. Average linkage (UPGMA)  
$$D(U, V) = \sum_{i,j} \frac{\text{dist}(U_i, V_j)}{|U||V|}$$
  
各クラスタのサイズによる加重平均
4. Ward method  
(UとVを併合してできる大きなクラスタ内の距離の二乗の総和) から  
(UとVそれぞれのクラスタ内の距離の二乗の総和) を引いた値を  
クラスタ間距離とする。 (分散最小化基準)



# さまざまな階層的クラスタリングの実行

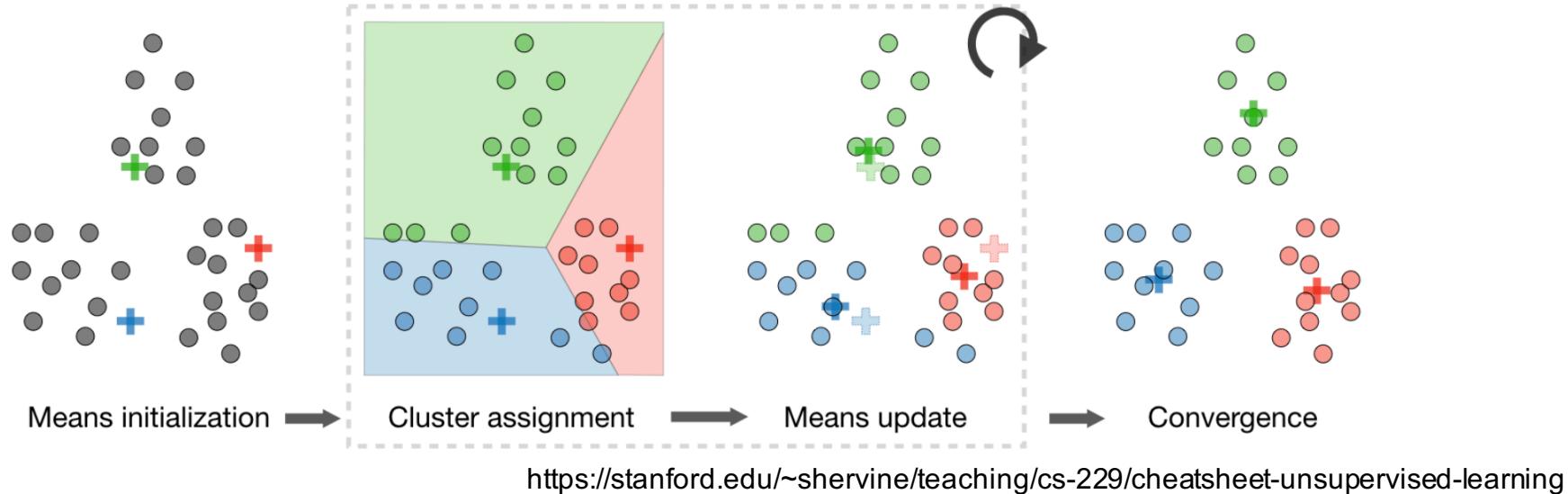
```
# 平均TPMのサンプル間分散がTop-20の遺伝子だけ抜き出し  
top20_df = df.loc[df.var(axis=1).sort_values(ascending=False).index[:20], :]
```

```
# サンプル間の距離計算手法、クラスター間の距離計算手法による違い  
sns.clustermap(top20_df, method='average', metric='correlation', col_colors=colors)  
sns.clustermap(top20_df, method='ward', metric='euclidean', col_colors=colors)
```

1. 次元削減
  - i. 行列分解による次元削減
    - a. 主成分分析 (Principal component analysis; PCA)
    - b. 非負値行列因子分解 (Non-negative matrix factorization; NMF)
    - c. 潜在意味解析 (Latent semantic indexing; LSI)
  - ii. 距離行列の最適化による次元削減
    - a. 多次元尺度構成法 (Multidimensional scaling; MDS)
    - b. その他の多様体学習
2. クラスタリング
  - i. 階層的クラスタリング
  - ii. クラスター「中心」との距離を最適化するクラスタリング
    - a. **k-means**
    - b. k-medoids
  - iii. 混合正規分布 (Gaussian mixture)

# k-means法

全体をいくつのクラスタに分割するかを最初に決めて、  
その分割を最適化するクラスタリング手法。



初期クラスタ中心をランダムに決める。  
中心との距離に基づいてサンプルをクラスタに分類、  
分類されたクラスタの平均を計算して、その位置を新たなクラスタ中心とし、  
そのクラスタ中心との距離に基づいてサンプルをクラスタに分類し、、、  
と分類が収束するまで繰り返す。

# k-means法の実行

```
import sklearn.cluster  
model = sklearn.cluster.KMeans(n_clusters=2)  
y = model.fit_predict(df.transpose().values)  
print(y)
```

# クラスタ数をいくつに設定すればいいのか

クラスタ数の決め方に正解はない。

いくつかのクラスタ数で実際にクラスタリングしてみて、できあがったクラスタの「良さ」を評価することで、なんとなく良さそうなクラスタ数を決める。

クラスタの評価手法として代表的なものは以下の2つ。

どちらも「同一クラスタ内のまとまり具合」と「別クラスタの分離具合」を評価。

## 1. Silhouette coefficient

サンプル*i*について、以下の2つを計算する。

$a(i)$  : サンプル*i*が属するクラスタ内の他のメンバーとの平均距離

$b(i)$  : サンプル*i*が属するクラスタともっとも近いクラスタのメンバーとの平均距離

これらを使って、サンプル*i*のシルエット係数は以下で計算できる。

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

他のクラスタと十分離れて、同じクラスタで十分近ければ、係数は1に近くなる。

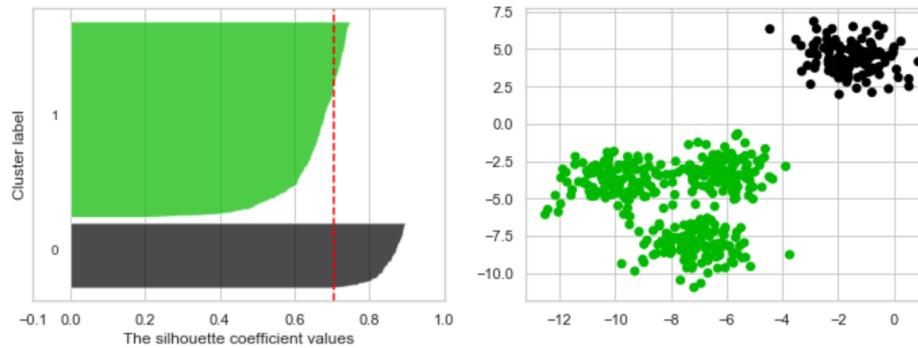
データ全体のクラスタの良さを見るためには、これらの係数の平均値を計算したり、値の分布を見たりする。

## 2. Calinski-Harabaz index

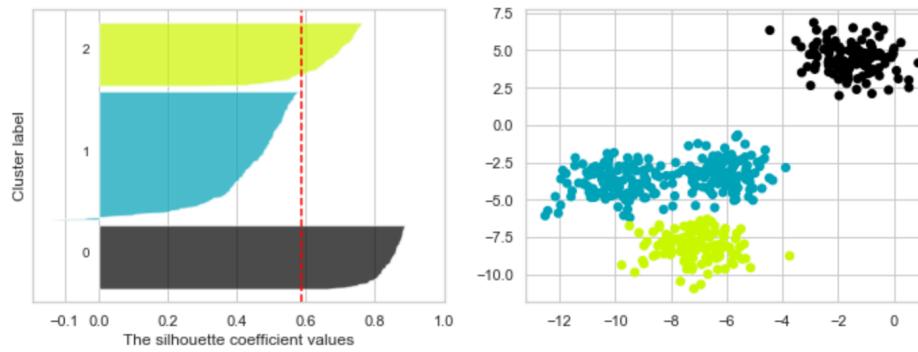
Silhouetteと同様の計算手法。クラスタ内分散とクラスタ間分散の計算から、データ全体のクラスタのまとまりと分離を評価。

# シルエット係数の計算

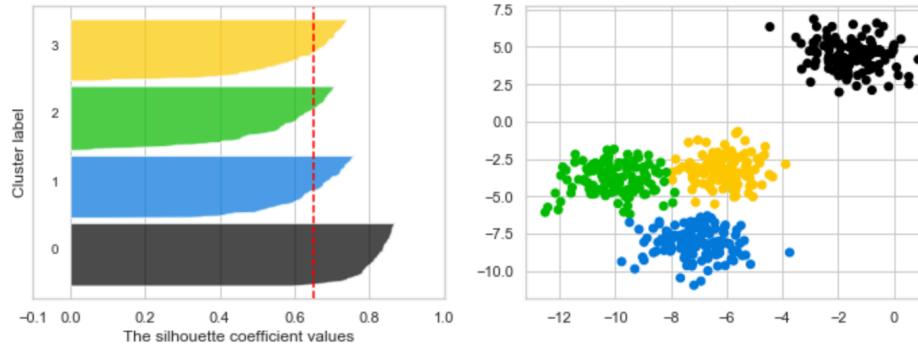
n\_clusters: 2 Average silhouette score = 0.7049787496083262



n\_clusters: 3 Average silhouette score = 0.5882004012129721



n\_clusters: 4 Average silhouette score = 0.6505186632729437



1. 次元削減
  - i. 行列分解による次元削減
    - a. 主成分分析 (Principal component analysis; PCA)
    - b. 非負値行列因子分解 (Non-negative matrix factorization; NMF)
    - c. 潜在意味解析 (Latent semantic indexing; LSI)
  - ii. 距離行列の最適化による次元削減
    - a. 多次元尺度構成法 (Multidimensional scaling; MDS)
    - b. その他の多様体学習
2. クラスタリング
  - i. 階層的クラスタリング
  - ii. クラスター「中心」との距離を最適化するクラスタリング
    - a. k-means
    - b. **k-medoids**
  - iii. 混合正規分布 (Gaussian mixture)

# k-meansの問題点

1. サンプル間の平均ベクトルを計算しなければならない関係上、データ（高次元ベクトル）全体を与えねばならず、また、ユークリッド距離空間での計算となる  
→好きな距離空間でのクラスタリングが行えない
2. クラスタの「中心」は平均ベクトルでいいのか？正規分布していない場合、平均の位置が意味をもつとは限らない

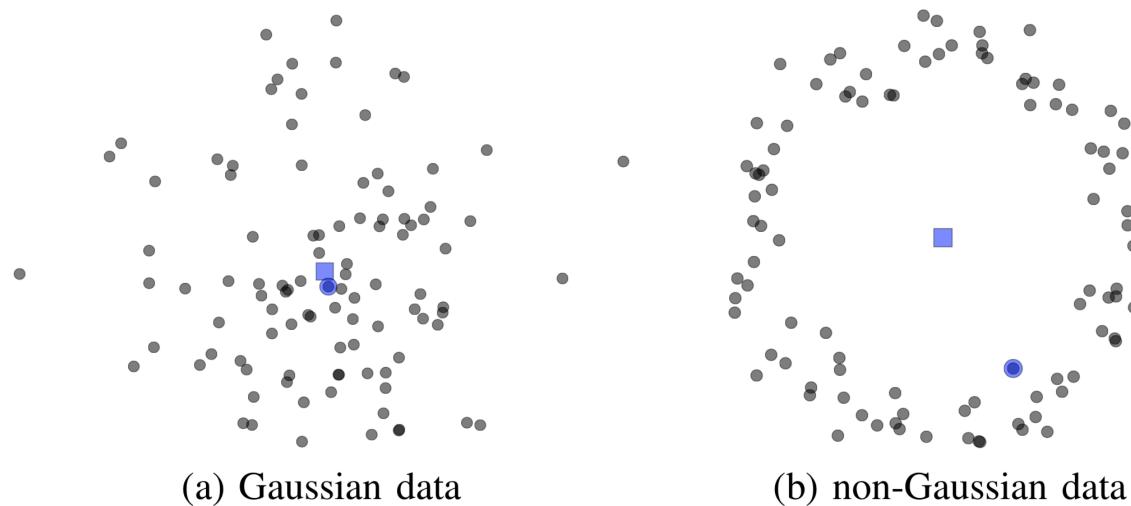


Fig. 1: Two data sets and their means ( $\square$ ) and medoids ( $\circ$ ).

[https://www.researchgate.net/publication/272351873\\_Numpy\\_SciPy\\_Recipes\\_for\\_Data\\_Science\\_k-Medoids\\_Clustering](https://www.researchgate.net/publication/272351873_Numpy_SciPy_Recipes_for_Data_Science_k-Medoids_Clustering)

# k-medoids法

平均 (mean) ではなく **medoid** (すべてのデータ点との距離の和が最小になるデータ点) を使う点だけがK-meansと異なる。

実データの平均を計算する必要がないため、実データを入力する必要がなく、データのサンプル間距離行列があればいい。→好きな距離関数を使える

```
def _k_medoids(self, X, n_clusters, max_iter, random_state, verbose):
    # 初期medoidsをランダムにn_clusters個選択。
    medoids = self._init_medoids(X, n_clusters, random_state)
    if self.verbose:
        print('initial medoids = ', medoids)
    new_medoids = np.copy(medoids)
    # ループ開始。medoidsが収束するまで繰り返す。
    for i in range(max_iter):
        # 各サンプルをもっとも近いmedoidsに割り当てる。
        assigned_cluster_labels = np.argmin(X[:, medoids], axis=1)
        for c in range(n_clusters): # クラスタごとに新たなmedoidsを選択。
            # クラスタc に割り当てられたサンプルのインデックスを取得。
            sample_indices_in_c = np.where(assigned_cluster_labels == c)[0]
            # クラスタc に割り当てられたサンプル内の距離行列を抽出。
            distance_matrix_of_c = X[np.ix_(sample_indices_in_c, sample_indices_in_c)]
            # クラスタc 内の各サンプルについて、同一クラスタ内サンプルとの平均距離を計算
            average_distance_in_c = np.mean(distance_matrix_of_c, axis=1)
            # 新たなmedoidは平均距離がもっとも小さいサンプル。
            new_medoid_index_in_c = np.argmin(average_distance_in_c)
            new_medoid_index = sample_indices_in_c[new_medoid_index_in_c]
            new_medoids[c] = new_medoid_index
        np.sort(new_medoids)
        if self.verbose:
            print('\titeration:', i, '\n\t\tnew medoids=', new_medoids)
        if np.array_equal(new_medoids, medoids):
            # medoidsのインデックスが更新されなかったらループ終了。
            break
    medoids = np.copy(new_medoids)
    assigned_cluster_labels = np.argmin(X[:, medoids], axis=1)
    return medoids, assigned_cluster_labels
```

# k-medoids法の実行

```
# K-medoidsクラスタリングの実行
distance_matrix = squareform(pdist(df.transpose().values, metric='correlation')) # 相関係数距離行列
n_clusters = 2

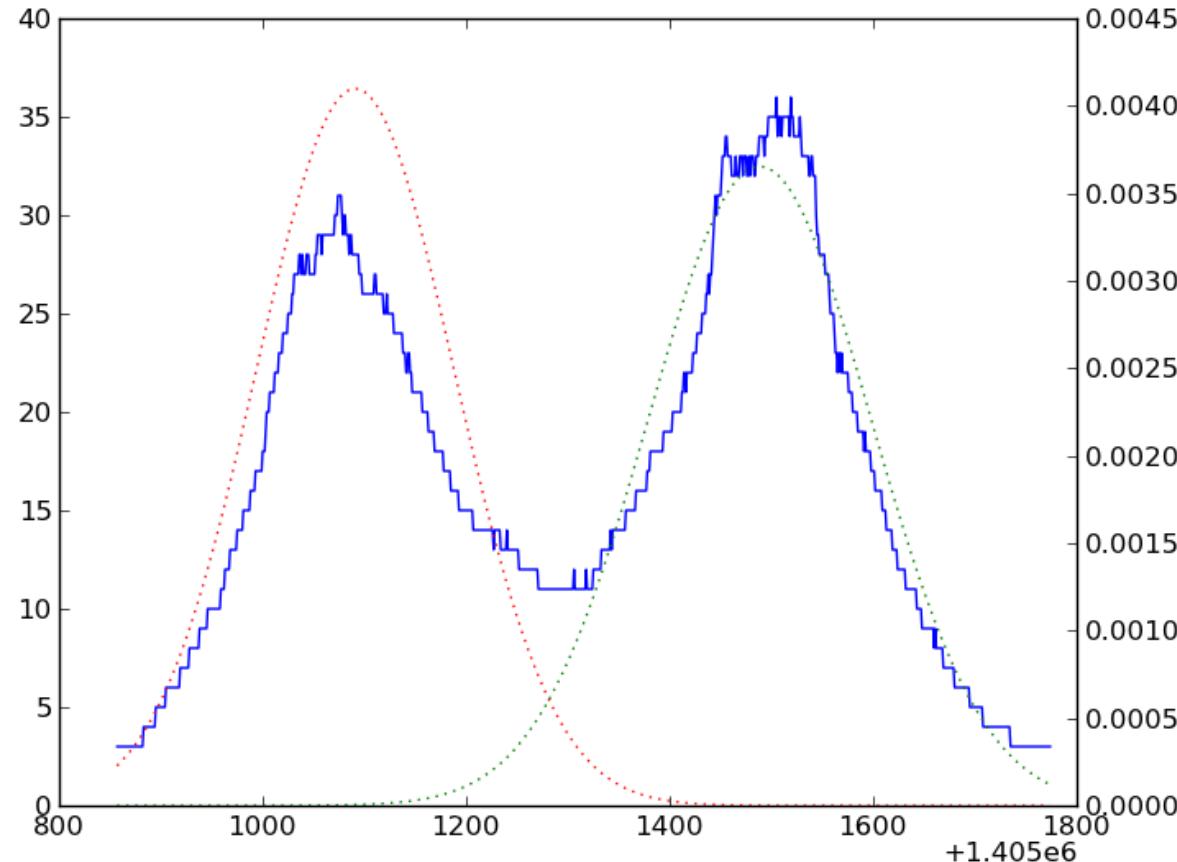
model = KMedoids(n_clusters=n_clusters, dissimilarity='precomputed', verbose=True)
model.fit(distance_matrix)
medoids = model.cluster_medoids_
labels = model.labels_
print('\nMedoids :',df.columns[medoids].values)
print('Clusters :')
for cluster in range(n_clusters):
    print('\tcluster-',cluster,':', df.columns[labels == cluster].values)
```

1. 次元削減
  - i. 行列分解による次元削減
    - a. 主成分分析 (Principal component analysis; PCA)
    - b. 非負値行列因子分解 (Non-negative matrix factorization; NMF)
    - c. 潜在意味解析 (Latent semantic indexing; LSI)
  - ii. 距離行列の最適化による次元削減
    - a. 多次元尺度構成法 (Multidimensional scaling; MDS)
    - b. その他の多様体学習
2. クラスタリング
  - i. 階層的クラスタリング
  - ii. クラスター「中心」との距離を最適化するクラスタリング
    - a. k-means
    - b. k-medoids
  - iii. 混合正規分布 (Gaussian mixture)

# 混合正規分布 (Gaussian mixture)

データの背後には、異なるパラメータ（平均、分散構造）を持ったいくつかの正規分布が存在し、観測されたデータはそれらが重ね合わさった混合正規分布から生成された、と考える（生成モデル）。

データ（サンプル）からそれらの正規分布のパラメータを推定する。



# 混合正規分布の最尤推定を実行

```
import sklearn.mixture
import matplotlib as mpl

n_samples = 500
C = np.array([[0., -0.1], [1.7, .4]])
X = np.r_[np.dot(np.random.randn(n_samples, 2), C),
          .7 * np.random.randn(n_samples, 2) + np.array([-6, 3])]

gmm = sklearn.mixture.GaussianMixture(n_components=2, covariance_type='full')
gmm.fit(X)

plt.figure()
ax = plt.subplot(111)
Y_ = gmm.predict(X)
cluster_colors = ['navy', 'turquoise']
for i, (mean, cov, color) in enumerate(zip(gmm.means_, gmm.covariances_, cluster_colors)):
    v, w = np.linalg.eigh(cov)
    if not np.any(Y_ == i):
        continue
    ax.scatter(X[Y_ == i, 0], X[Y_ == i, 1], .8, color=color)
    angle = np.arctan2(w[0][1], w[0][0])
    angle = 180. * angle / np.pi
    v = 2. * np.sqrt(2.) * np.sqrt(v)
    ell = mpl.patches.Ellipse(mean, v[0], v[1], 180. + angle, color=color)
    ell.set_clip_box(ax.bbox)
    ell.set_alpha(.5)
    ax.add_artist(ell)
```

# 混合正規分布のベイズ推論 (Variational Bayesian Gaussian Mixture Model; VBGMM. 変分混合正規分布)

尤度をEM法で最大化するのではなく、  
周辺尤度（モデルエビデンス）の下限を変分推論によって最大化する。

混合正規分布の事前分布を考えることで、  
正則化の効果がもたらされる（ベイズのご利益）

scikit-learnでは最近、

**`sklearn.mixture.BayesianGaussianMixture`**

が実装された。

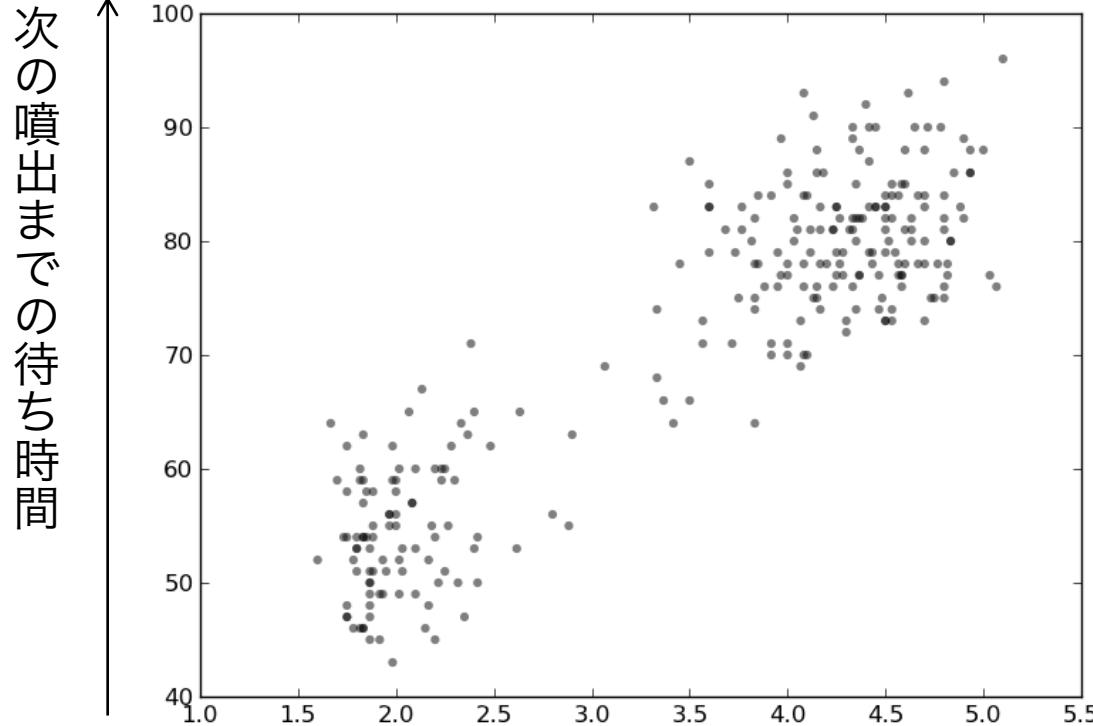
事前分布として、

1. ディリクレ分布
2. ディリクレ過程 (棒折り過程によるノンパラメトリックベイズ)

のいずれかが選べる。

今回扱うのはディリクレ分布。

# Old faithful 間欠泉データのクラスタリング

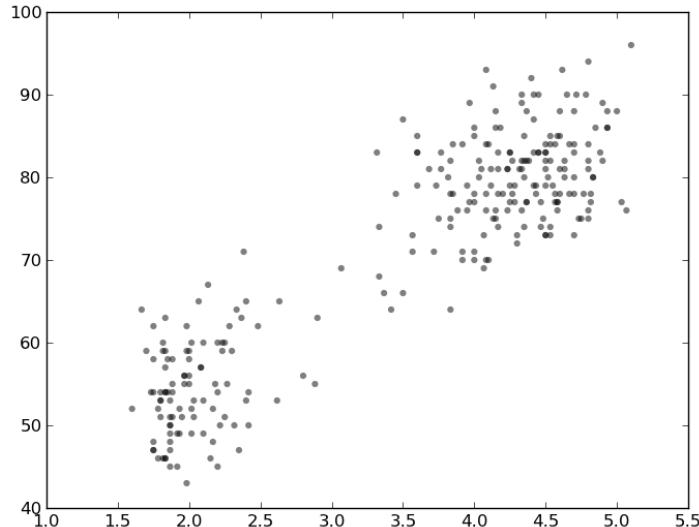


一回の噴出の持続時間

cf) C言語実装 : <https://github.com/khigashi1987/VB-GMM>

# 確率モデル

- 混合正規分布を仮定



N ... データ数  
K ... クラスタ数

X ... データベクトル  
Z ... そのデータが、どのクラスタに所属するかを示すラベル (K次元二値ベクトルで一要素だけが1)  
 $\pi$ ... 混合比  $\mu$  ... 平均パラメータ  $\Lambda$  ... 精度パラメータ  
 $m_0, \beta_0, W_0, v_0$  ... hyperparameter

## 尤度

$$P(Z | \pi) = \prod_{n=1}^N \prod_{k=1}^K \pi_k^{Z_{nk}}$$

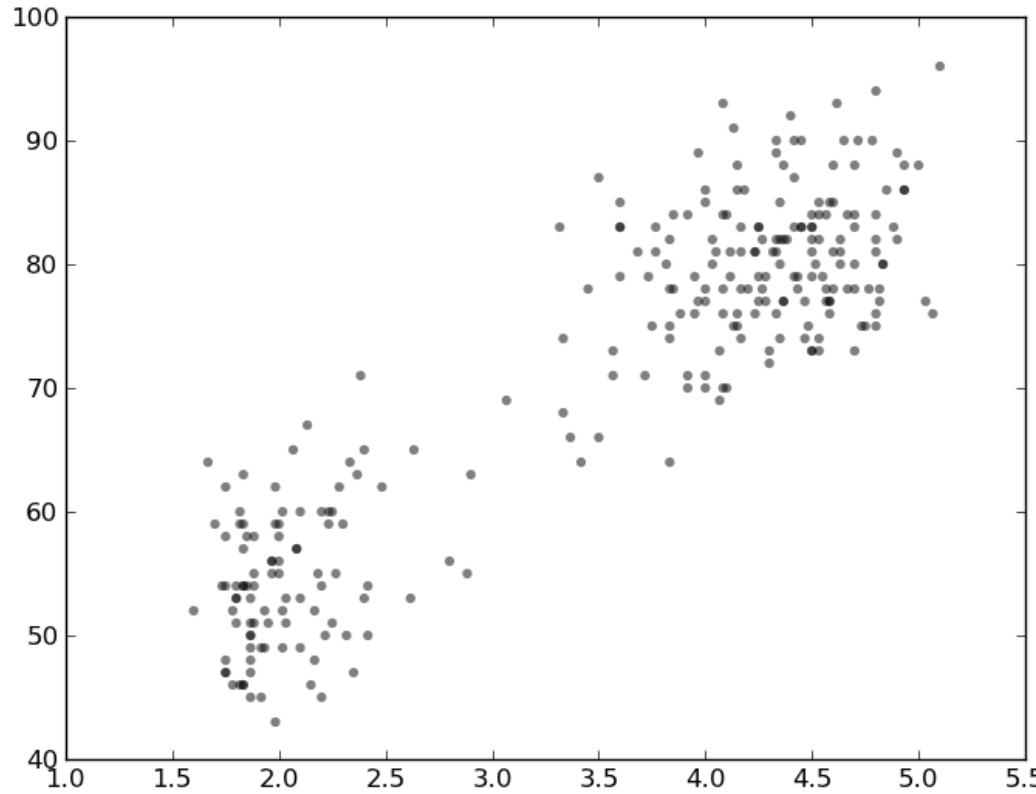
$$P(X | Z, \mu, \Lambda) = \prod_{n=1}^N \prod_{k=1}^K N(x_n | \mu_k, \Lambda_k^{-1})^{z_{nk}}$$

## 事前分布 (共役事前分布)

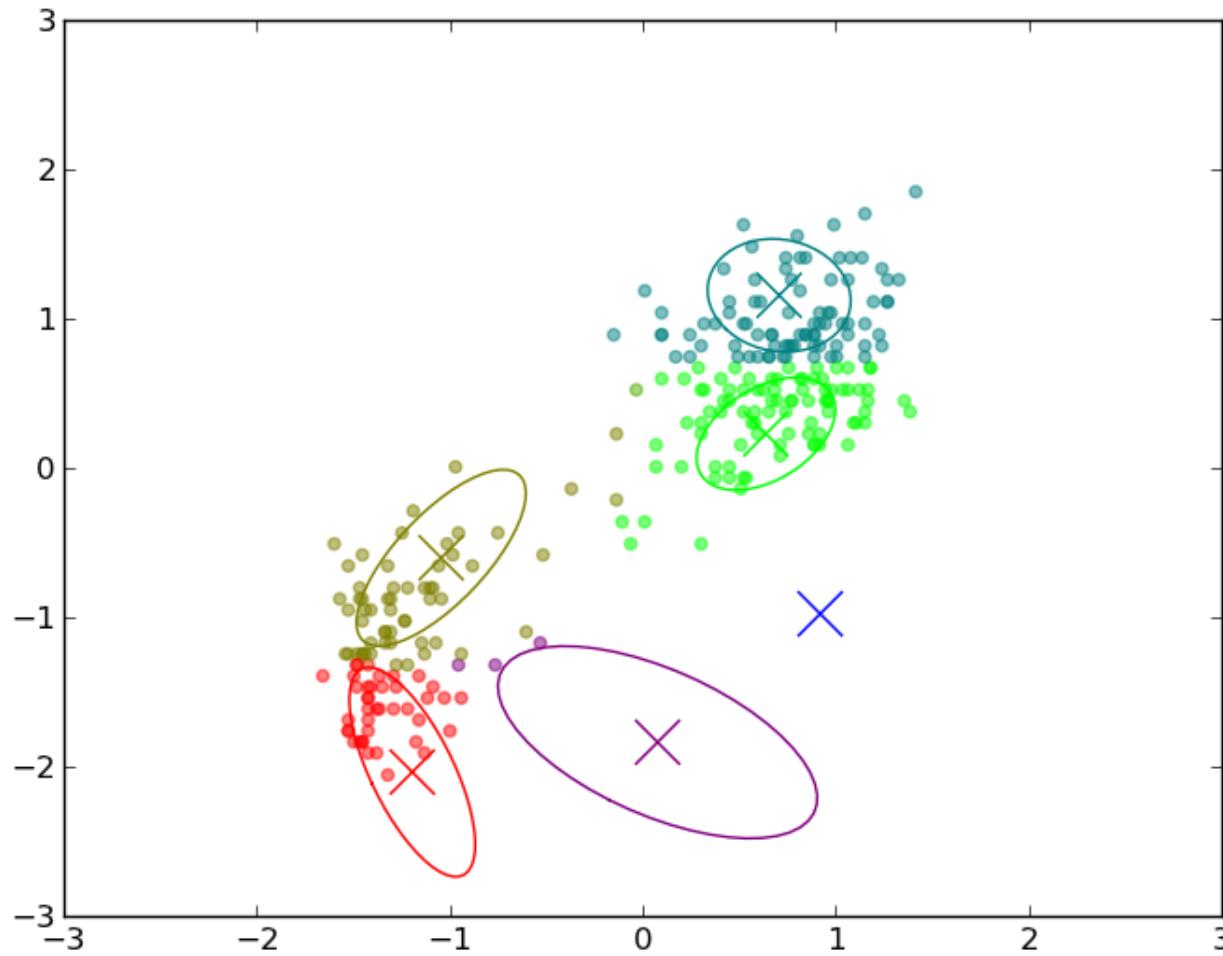
$$P(\pi) = Dir(\pi | \alpha_0)$$

$$P(\mu, \Lambda) = P(\mu | \Lambda)P(\Lambda) = \prod_{k=1}^K N(\mu_k | m_0, (\beta_0 \Lambda_k)^{-1}) W(\Lambda_k | W_0, v_0)$$

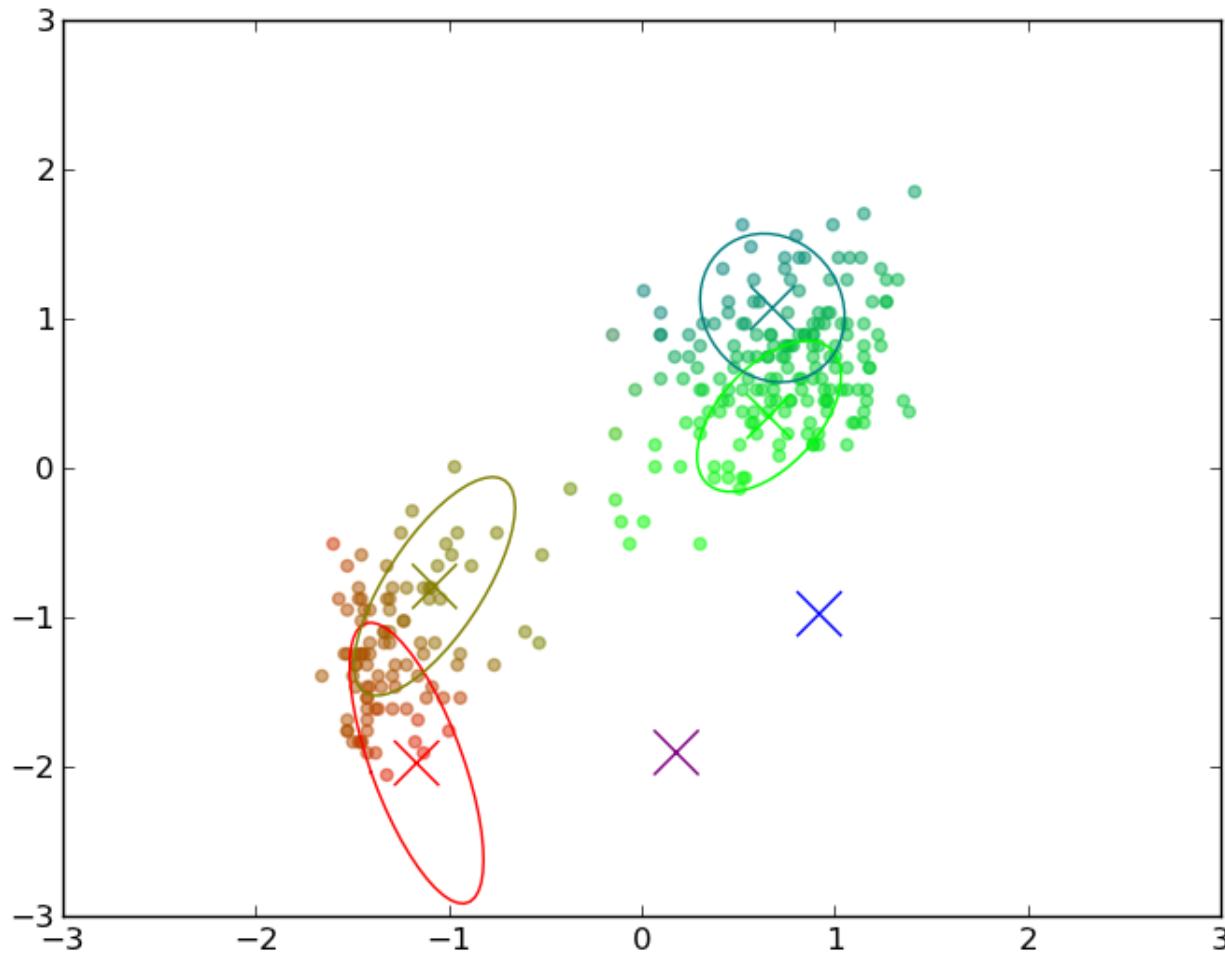
# 例



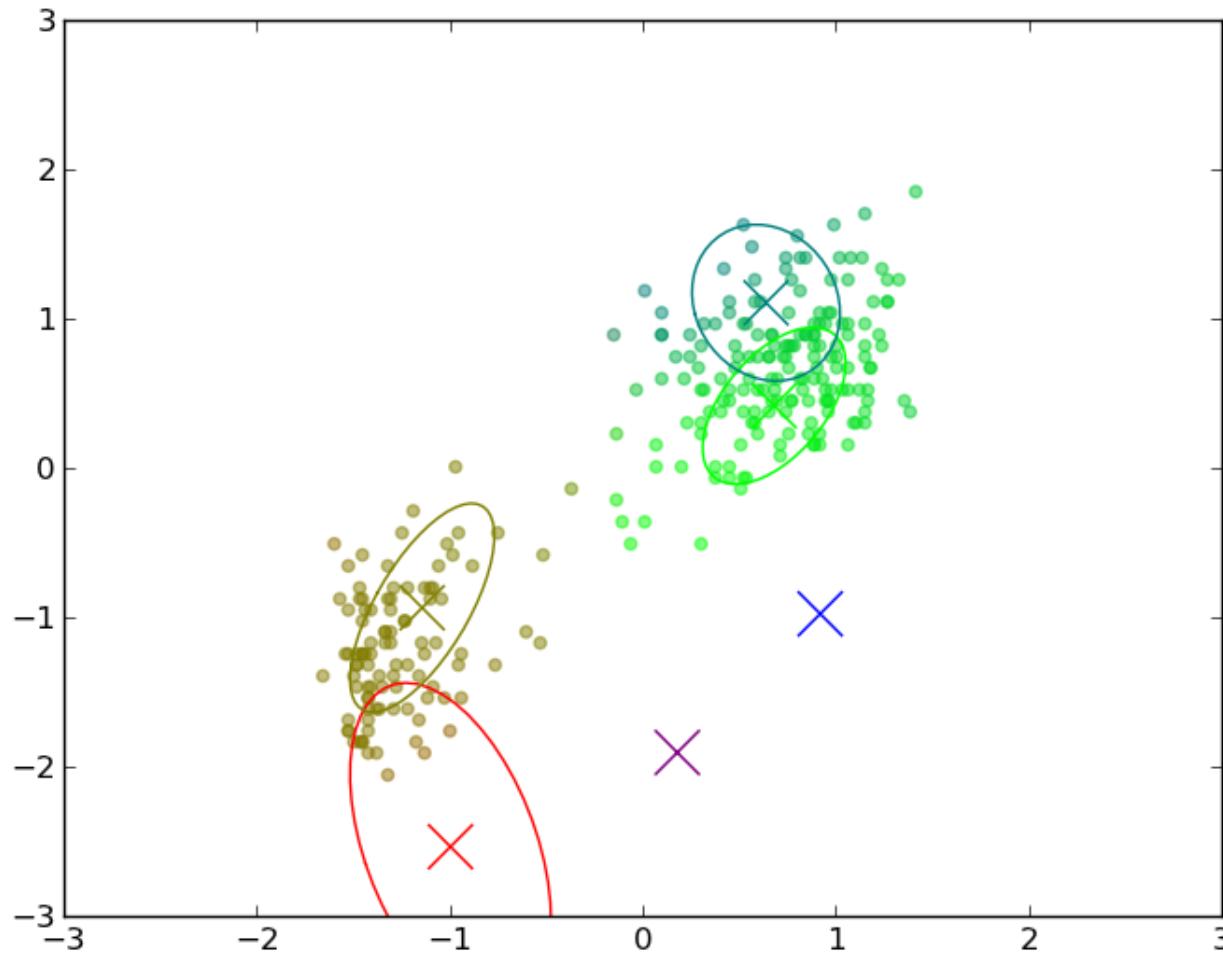
# STEP 1



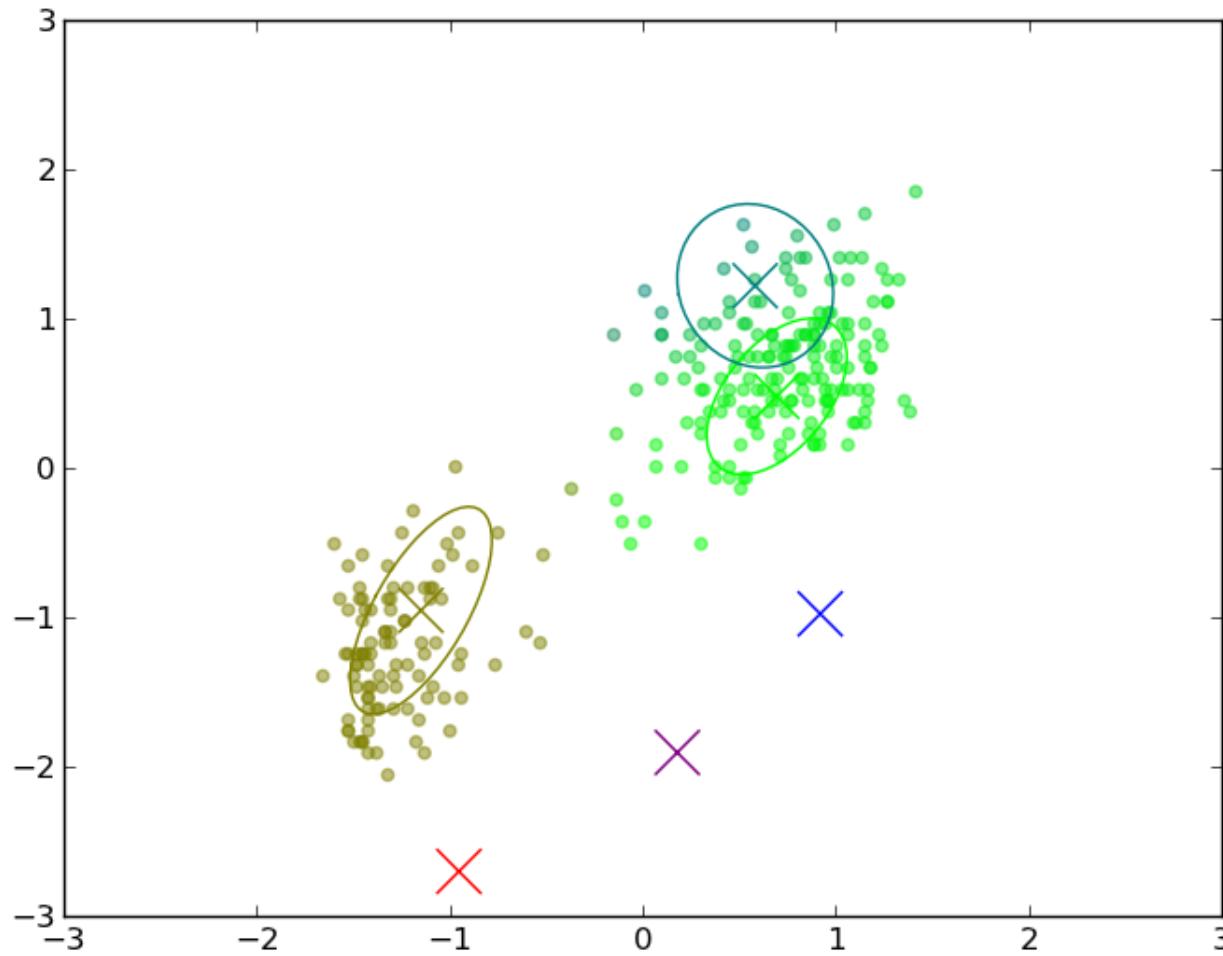
# STEP 5



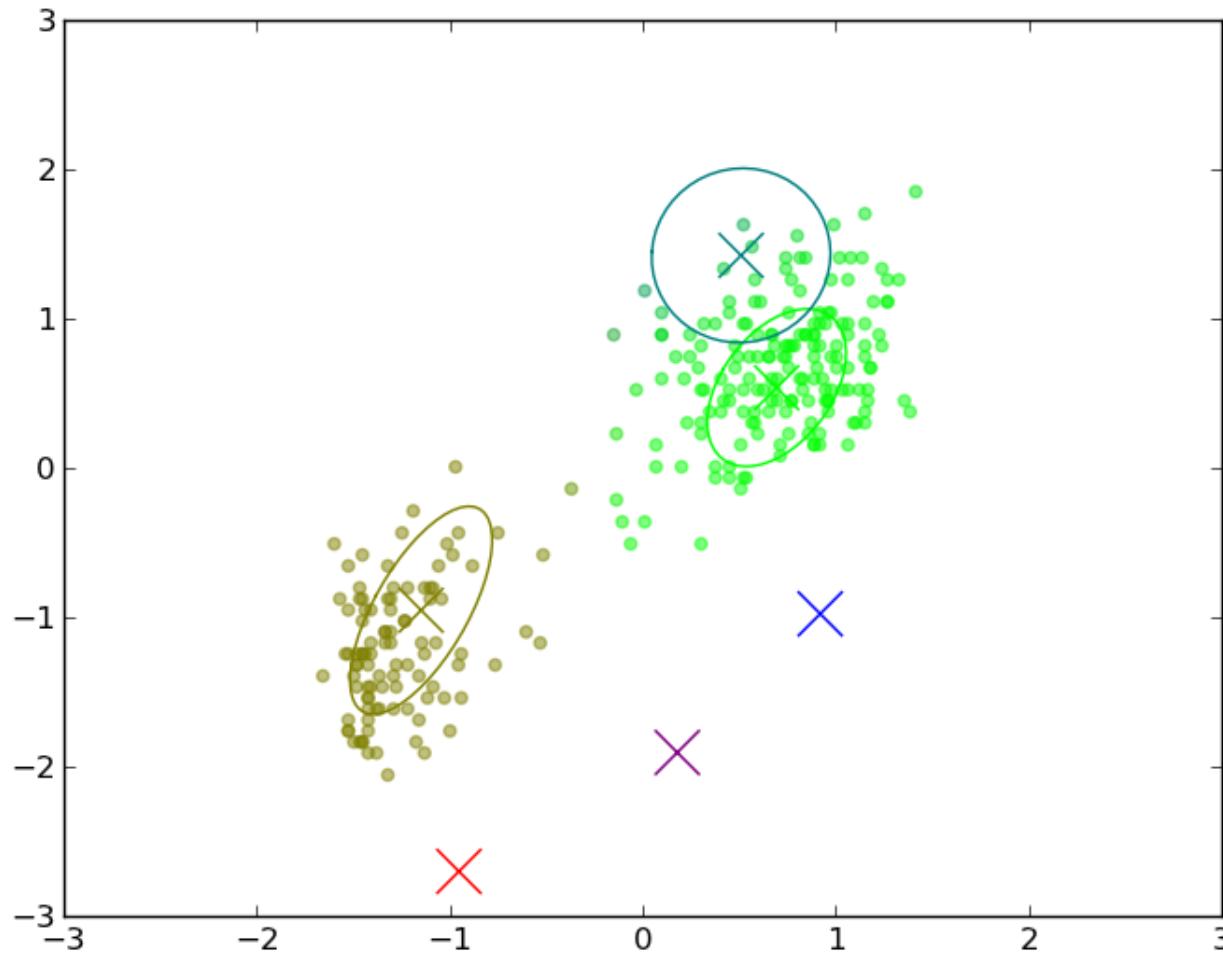
# STEP 10



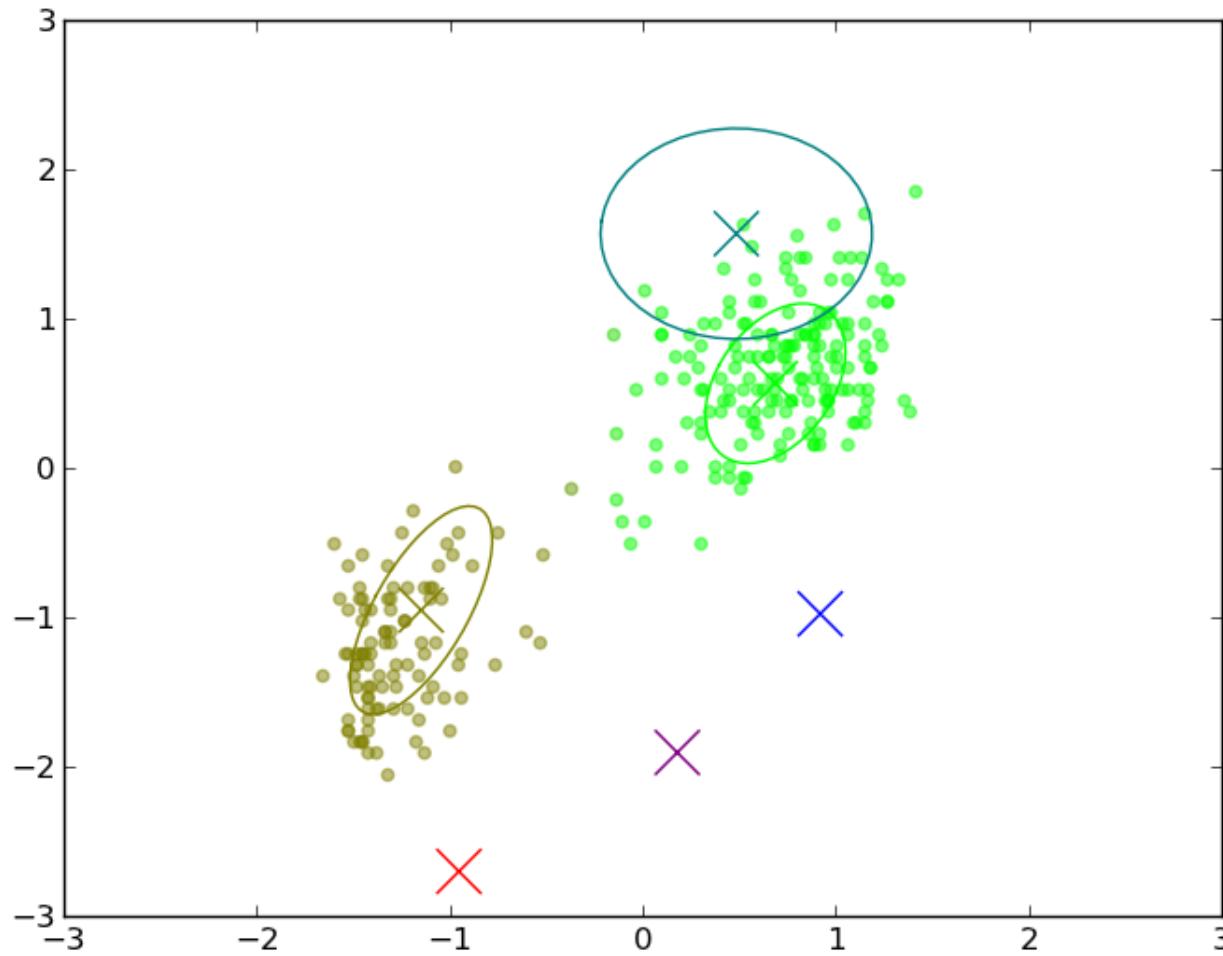
# STEP 15



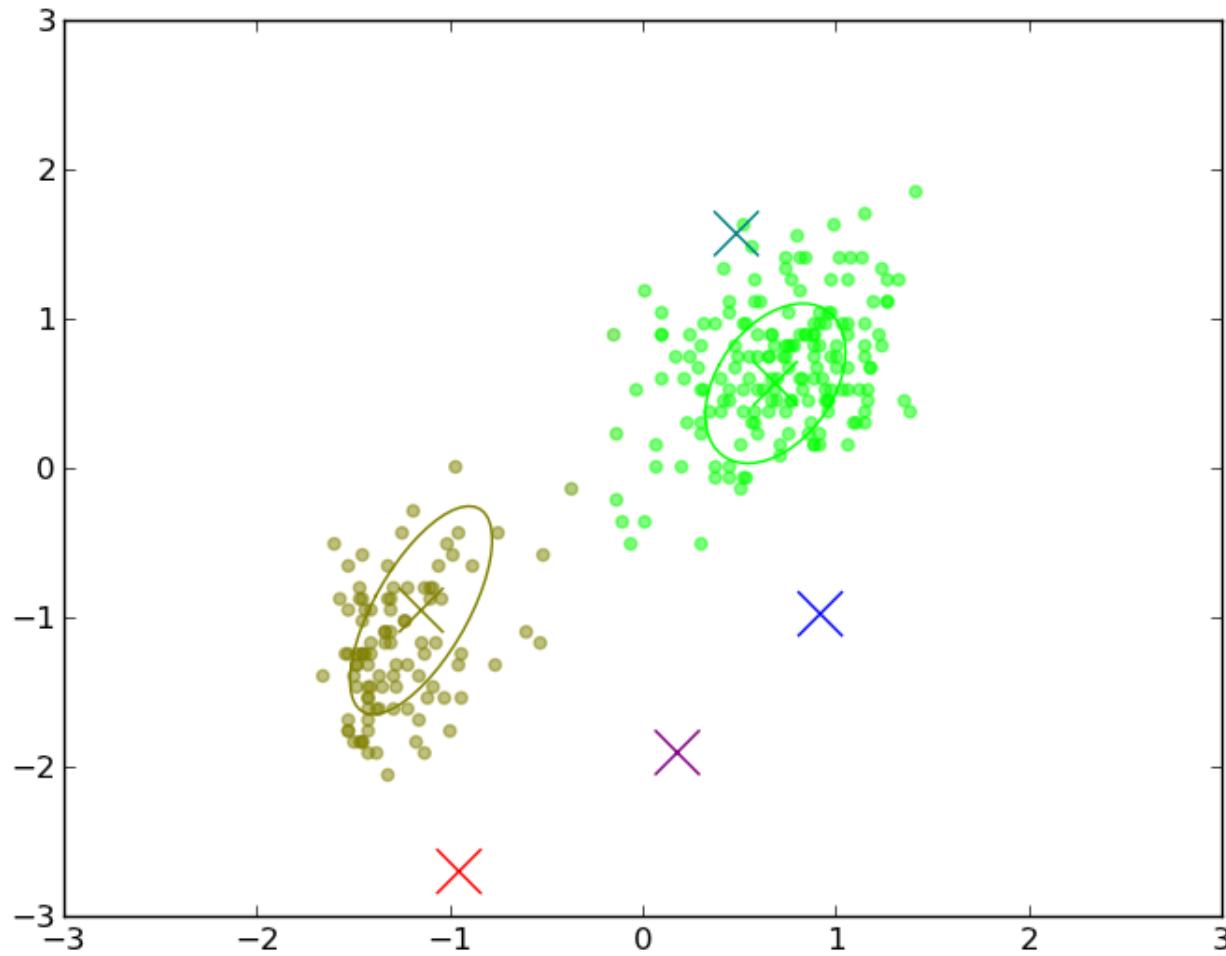
# STEP 20



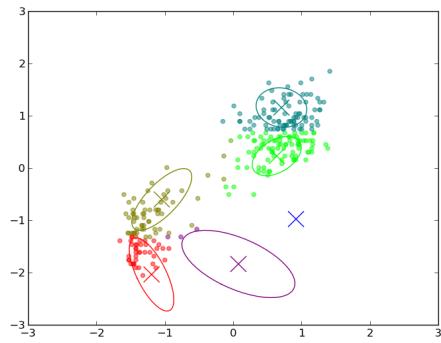
# STEP 25



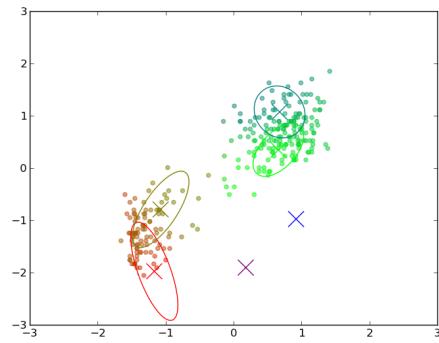
# STEP 30



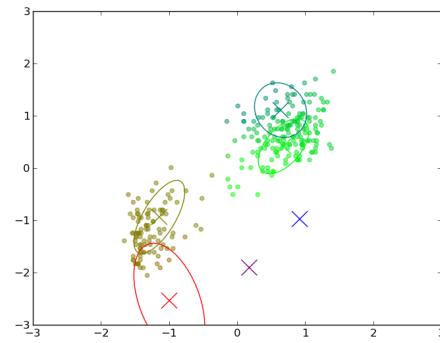
step 1



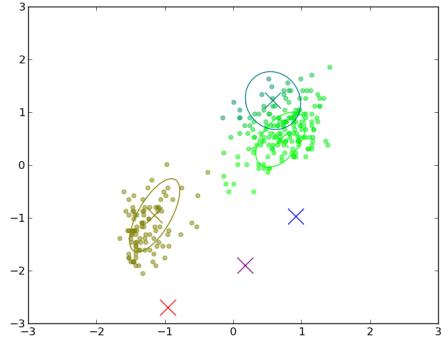
step 5



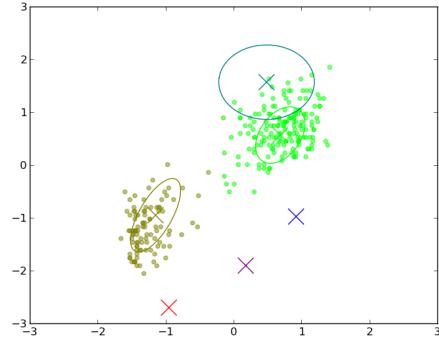
step 10



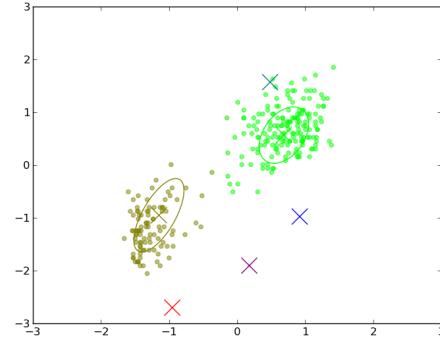
step 15



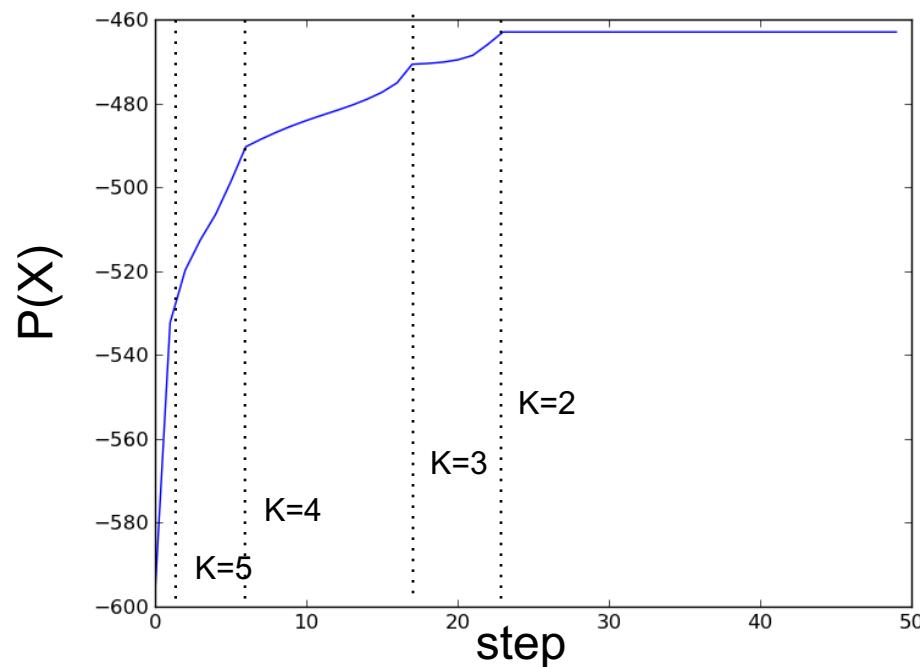
step 25



step 30



# ベイズ推論の正則化としての効果



必要のない要素は勝手に縮退して、適切なクラスタだけが生き残る  
(ディリクレ事前分布の効果)



多めのクラスタ数を与えて計算させればいい

# 混合正規分布のベイズ推論の実行

```
# サンプル間で分散の大きいトップ1000の遺伝子のみを使って推定
var1000_genes = std_df.var(axis=1).sort_values(ascending=False).index[:1000]
X = std_df.loc[var1000_genes, :].transpose().values

pca = sklearn.decomposition.PCA()
pca_coords = pca.fit_transform(X)
cluster_colors = np.array(['navy', 'turquoise', 'cornflowerblue', 'darkorange'])

for n_clusters in [1, 2, 3, 4]:
    vbgmm = sklearn.mixture.BayesianGaussianMixture(n_components=n_clusters,
                                                    weight_concentration_prior_type='dirichlet_distribution')
    vbgmm.fit(X)
    print('Number of clusters estimated:', n_clusters)
    for c in range(n_clusters):
        print('\tcluster-', c, ' weights =', vbgmm.weights_[c])
    y = vbgmm.predict(X)
    scatter_plot(pca_coords, std_df.columns, cluster_colors[y])
```