3-1_DimensionalityReduction_Clustering

2018 年 11 月 20 日

行列分解による線形次元削減、多様体学習による非線形次元削減、およびクラスタリング

1 事前準備

In [1]:
```
%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
```

```
pandas 0.23.4
numpy 1.15.4
matplotlib 3.0.1
seaborn 0.9.0
scikit-learn 0.20.0
```

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))
```

```
           batch_1  batch_2  batch_3  chemostat_1  chemostat_2  chemostat_3
gene_id
```

```
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
```

In [3]:
```
# 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))
```

```
           batch_1  batch_2  batch_3  chemostat_1  chemostat_2  chemostat_3
gene_id
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
5892
```

In [4]:
```
# 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)
```

2 次元削減

2.1 行列分解に基づく次元削減

2.1.1 PCA (Principal Component Analysis; 主成分分析)

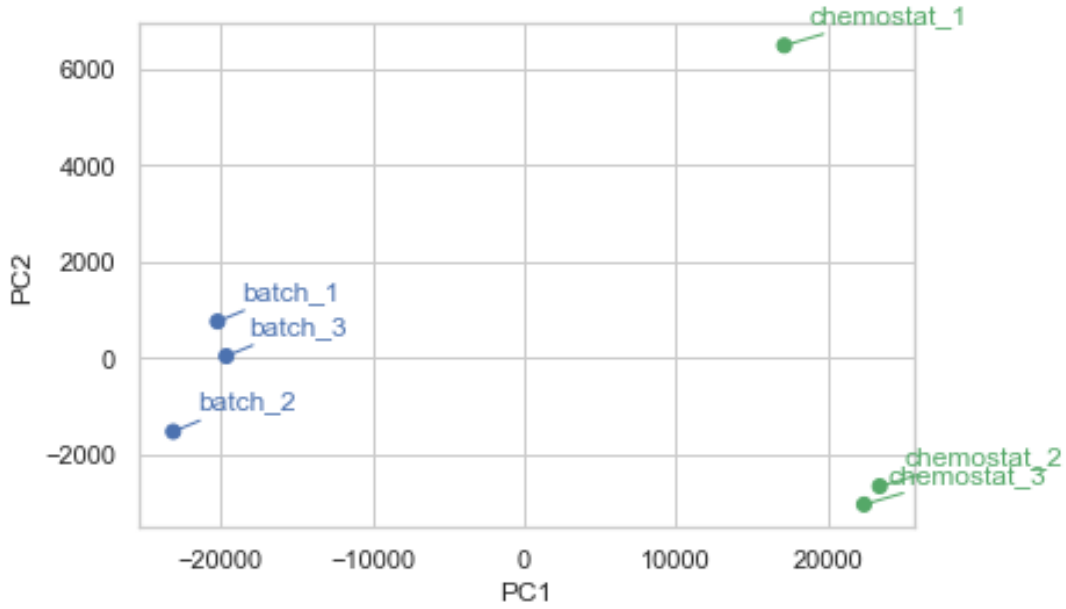
In [5]:
```
import sklearn.decomposition
```

In [6]:
```
# PCA 実行
pca = sklearn.decomposition.PCA()
coords = pca.fit_transform(df.transpose().values)
```

In [7]:
```
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 [8]:
```
scatter_plot(coords, df.columns, colors, xlabel='PC1', ylabel='PC2')
```



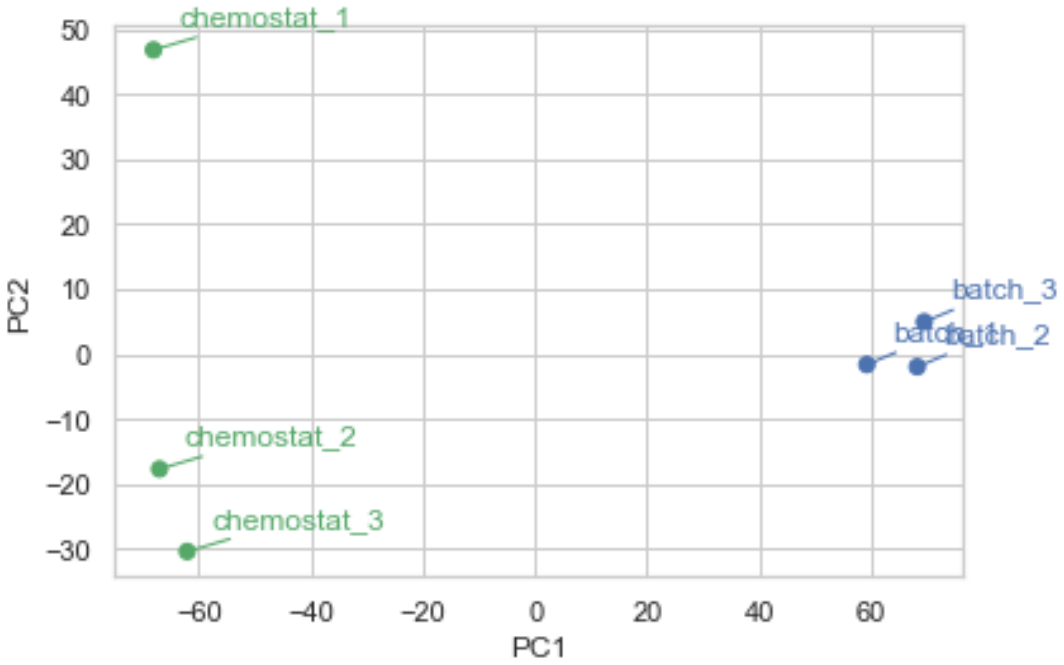
In [9]:
```
# z-score 標準化
import sklearn.preprocessing
values = df.transpose().values
scaler = sklearn.preprocessing.StandardScaler(with_mean=True, with_std=True)
std_values = scaler.fit_transform(values)
std_df = pd.DataFrame(std_values.T, index=df.index, columns=df.columns)

print('\nRaw values')
print('TPM Average :')
print(df.mean(axis=1)[:10])
print('\nTPM Standard deviation :')
print(df.std(ddof=0, axis=1)[:10])
```

```
print('\n\nStandardized values')
print('TPM Average :')
print(std_df.mean(axis=1)[:10])
print('\nTPM Standard deviation :')
print(std_df.std(ddof=0, axis=1)[:10])

#おんなじことを自分で計算する場合
#values = df.transpose().values
#std_values = (values - values.mean(axis=0)) / values.std(axis=0)
#std_df2 = pd.DataFrame(std_values.T, index=df.index, columns=df.columns)
#print('\n\nStandardized values 2')
#print('TPM Average :')
#print(std_df2.mean(axis=1)[:10])
#print('\nTPM Standard deviation:')
#print(std_df2.std(ddof=0, axis=1)[:10])
```

```
Raw values
TPM Average :
gene_id
gene_0001     0.73
gene_0005     4.41
gene_0009     2.53
gene_0010    82.03
gene_0011    85.13
gene_0012   214.09
gene_0013    80.77
gene_0014    32.10
gene_0015    33.83
gene_0016    27.58
dtype: float64

TPM Standard deviation :
gene_id
gene_0001    1.11
gene_0005    2.35
gene_0009    1.12
gene_0010   73.92
gene_0011   71.10
gene_0012   22.93
gene_0013   13.01
gene_0014    4.40
gene_0015   12.19
gene_0016   12.55
```

```
dtype: float64


Standardized values
TPM Average :
gene_id
gene_0001    0.00
gene_0005   -0.00
gene_0009   -0.00
gene_0010   -0.00
gene_0011   -0.00
gene_0012    0.00
gene_0013    0.00
gene_0014   -0.00
gene_0015    0.00
gene_0016   -0.00
dtype: float64

TPM Standard deviation :
gene_id
gene_0001   1.00
gene_0005   1.00
gene_0009   1.00
gene_0010   1.00
gene_0011   1.00
gene_0012   1.00
gene_0013   1.00
gene_0014   1.00
gene_0015   1.00
gene_0016   1.00
dtype: float64
```

```
In [10]: # 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')
```




```
In [11]: # 各主成分の「寄与率」
         print(['{:.2f}%'.format(x*100) for x in pca.explained_variance_ratio_])
```

```
['73.55%', '9.78%', '7.09%', '5.54%', '4.04%', '0.00%']
```

2.1.2 NMF (Non-negative Matrix Factorization; 非負値行列因子分解)

X ~ W * H

X: non-negative matrix. (n, m)

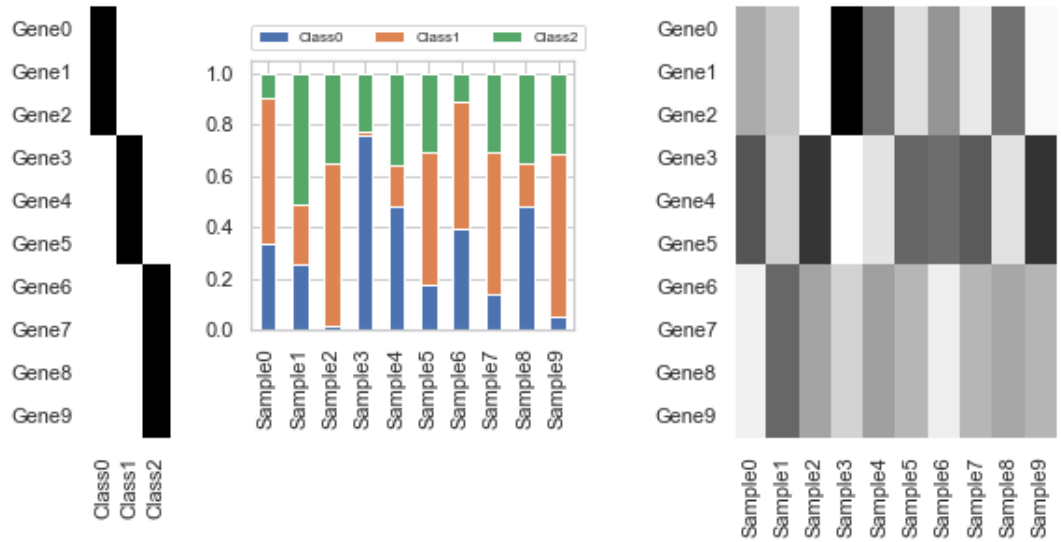
W: non-negative matrix. (n, k)

H: non-negative matrix. (k, m)

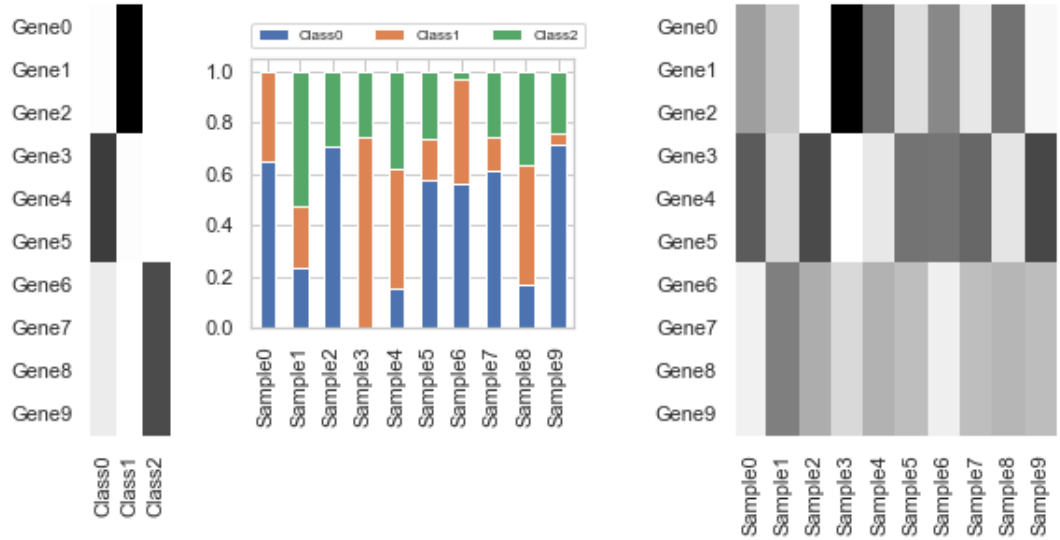
```
In [12]: # 擬似データで NMF のデモ


         def plot_W_H_X(W, H, X):
             classes = ['Class%d'%x for x in range(W.shape[1])]
             genes = ['Gene%d'%x for x in range(W.shape[0])]
             samples = ['Sample%d'%x for x in range(H.shape[1])]
             fig = plt.figure()
             ax1 = plt.axes([0.0, 0.1, 0.1, 0.8])
             sns.heatmap(pd.DataFrame(W, index=genes, columns=classes), cmap='Greys', cbar=False, ax
             ax2 = plt.axes([0.2, 0.3, 0.4, 0.5])
             pd.DataFrame(H, index=classes, columns=samples).transpose().plot.bar(stacked=True, ax=a
             plt.legend(loc=(0., 1.04), ncol=3, fontsize=8)
             ax3 = plt.axes([0.8, 0.1, 0.4, 0.8])
```

```
        sns.heatmap(pd.DataFrame(X, index=genes, columns=samples), cmap='Greys', cbar=False)
        plt.show()

    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)
```

```
Original
```



Reconstructed

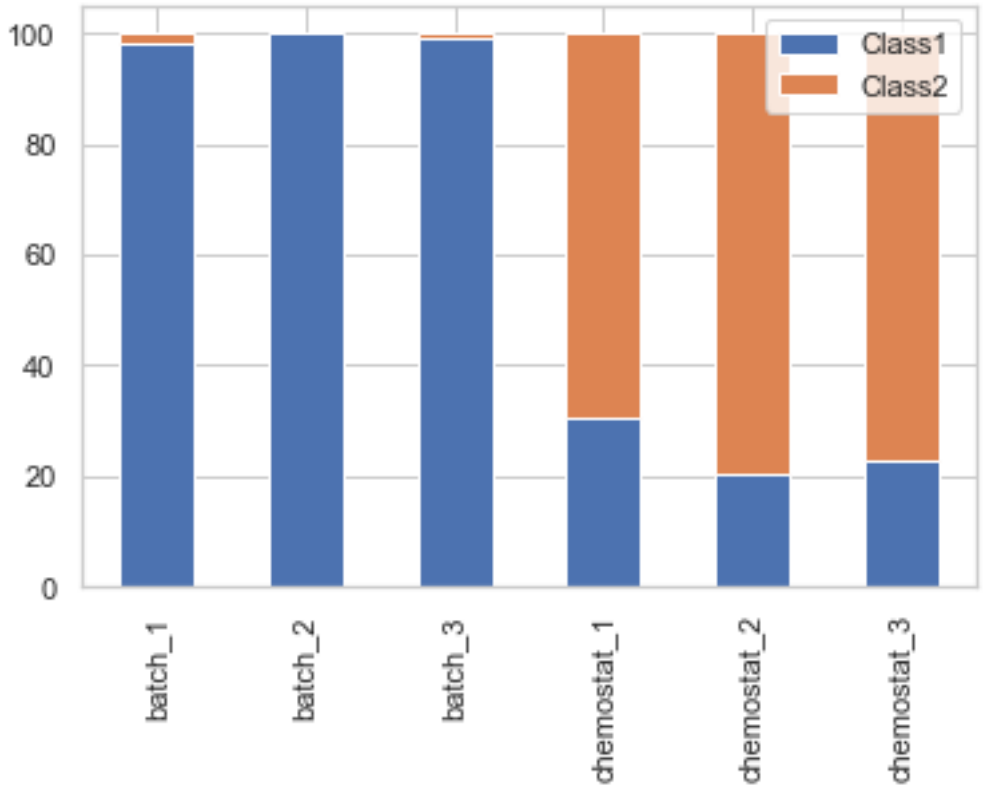


In [13]:
```
model = sklearn.decomposition.NMF(n_components=2)
W = model.fit_transform(df.values)
H = model.components_

print('Original shape=',df.values.shape)
print('W shape =',W.shape)
print('H shape =',H.shape)
```

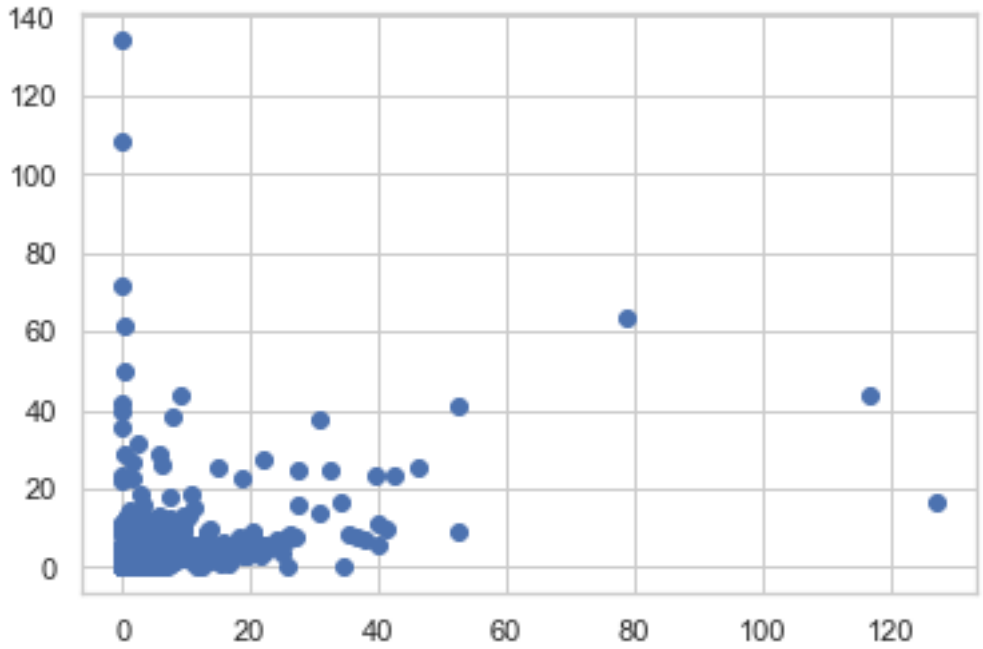
```
Original shape= (5892, 6)
W shape = (5892, 2)
H shape = (2, 6)
```

In [14]:
```
# 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(stack
#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[14]: <matplotlib.axes._subplots.AxesSubplot at 0x1a16566f60>



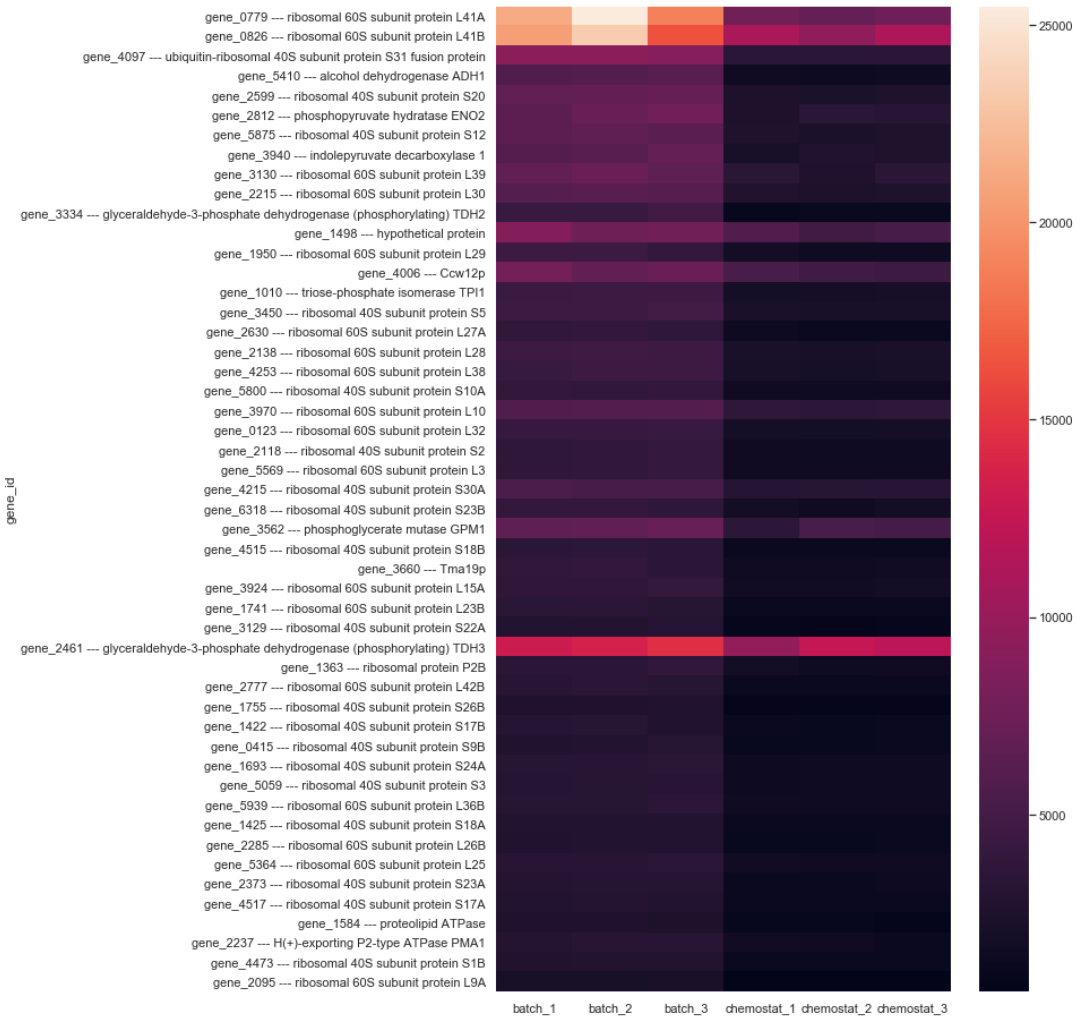
In [15]:
```
# それぞれの因子に強く寄与している遺伝子はなにか？
fig = plt.figure()
ax = fig.add_subplot(111)
ax.scatter(W[:,0], W[:,1])
plt.show()
```



In [16]:
```
# 遺伝子の product 情報をロード
gene_products = pd.read_table('./input/gene_id_product.tsv', index_col=0, names=["gene_id",
```

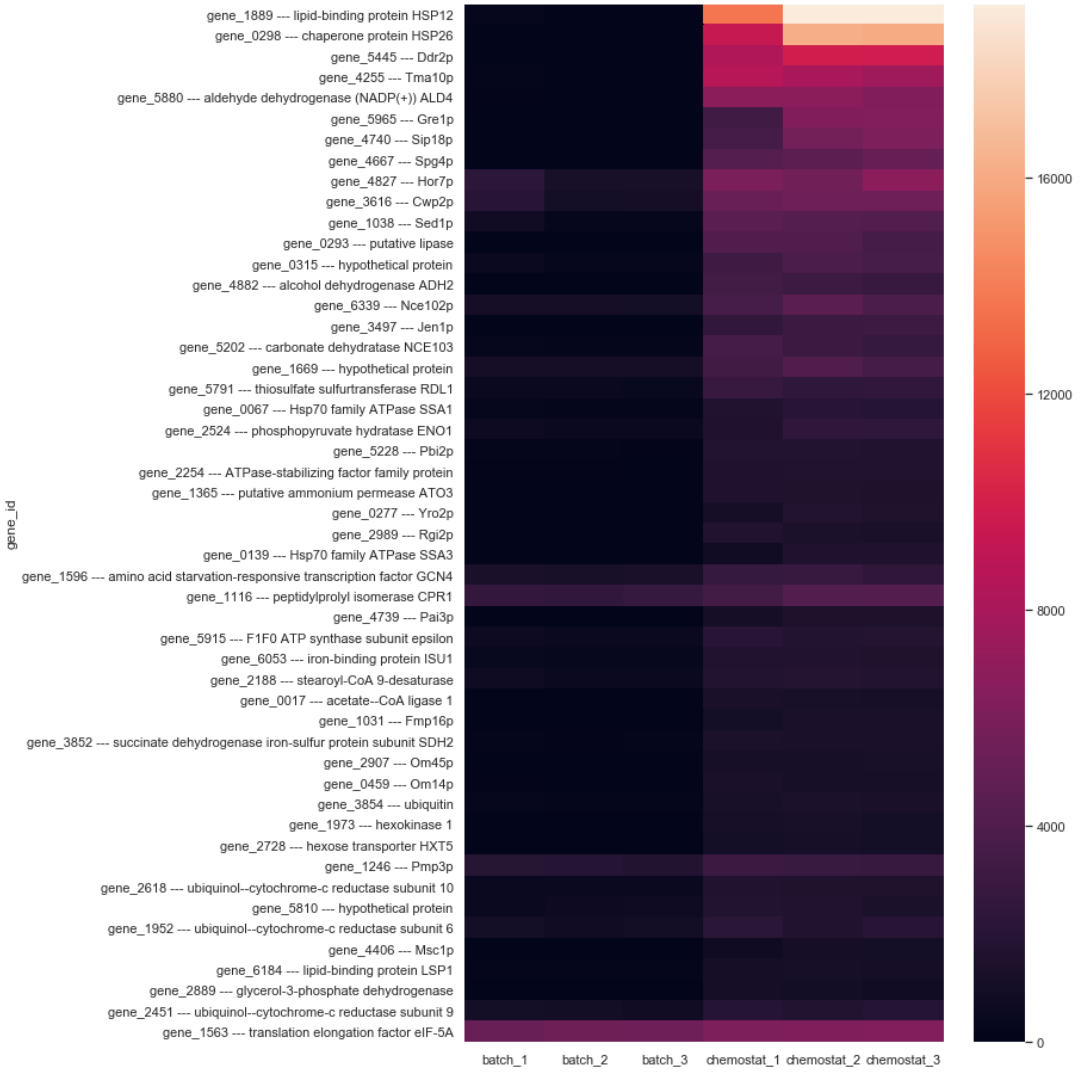
In [17]:
```
# 因子１　に強く寄与する遺伝子
topN = 50
top_factor1 = df.index[ np.argsort(W[:,0] - W[:,1])[::-1][:topN] ]
gene_labels = top_factor1 + ' --- ' + gene_products.loc[top_factor1, 'product']

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

Out[17]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a16589f28>
```



In [18]:
```
# 因子 2  に強く寄与する遺伝子
top_factor2 = df.index[ np.argsort(W[:,0] - W[:,1])[:topN] ]
gene_labels = top_factor2 + ' --- ' + gene_products.loc[top_factor2, 'product']

fig = plt.figure(figsize=(9,16))
sns.heatmap(df.loc[top_factor2, :], yticklabels=gene_labels)
```

Out[18]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a1772ff60>
```



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

In [19]:
```
# 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)
```

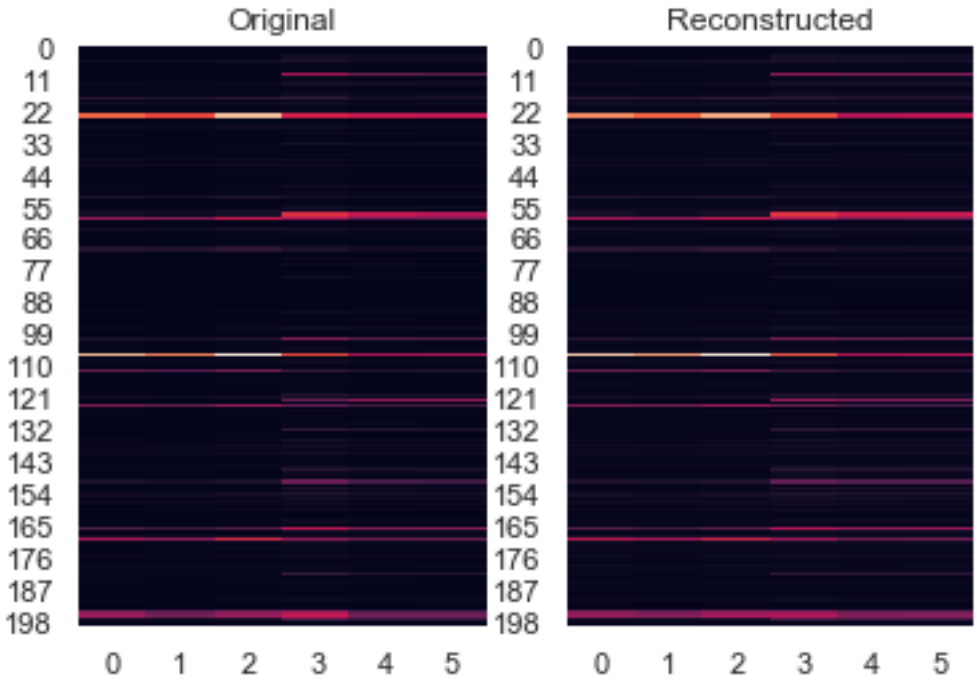
In [20]:
```
# SVD (Singular Value Decomposition; 特異値分解) を実行
import numpy.linalg
U, Sig, V = np.linalg.svd(df_tfidf.values, full_matrices=False)
Weights = np.dot(np.diag(Sig), V)
```

```
# 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))
```

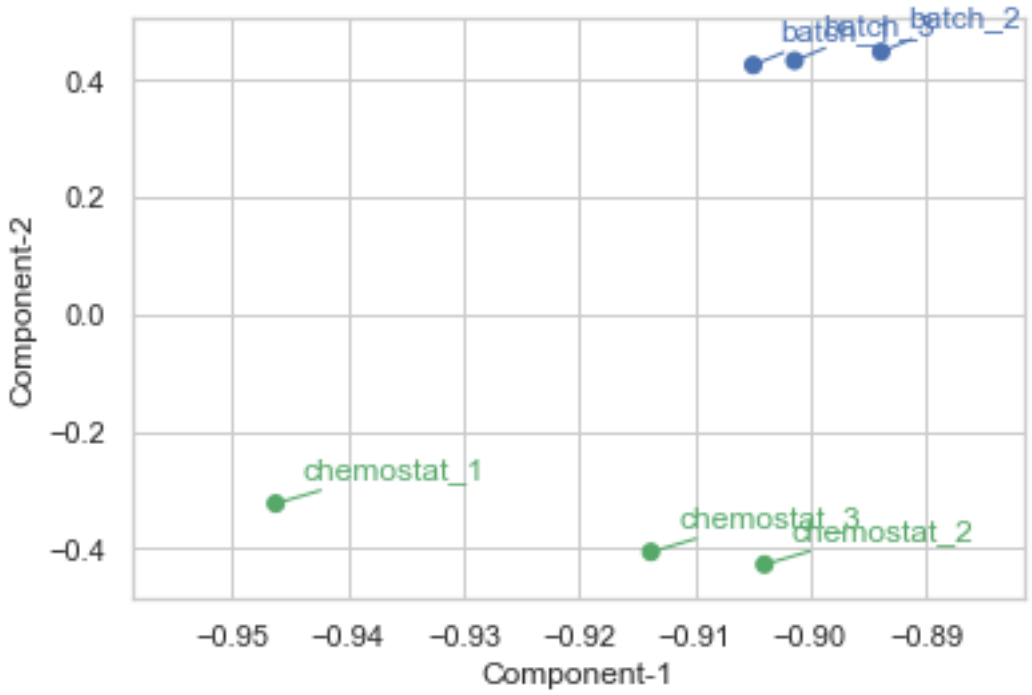
Out[20]: True

```
In [21]: U_reduced = U[:, :2]
         W_reduced = np.dot(np.diag(Sig[:2]), V[:2, :])
```

```
In [22]: 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()
```




```
In [23]: # 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'
```



In [24]:
```
# batch と chemostat の違いは、　Component-2 がプラスに寄与するかマイナスに寄与するか。
# Components-2 にどういう遺伝子の合成ベクトルなのか、係数が大きい遺伝子をいくつか見てみる
C2_genes = pd.DataFrame(U_reduced[:, 1], index=df_tfidf.index, columns=['weights'])
# 係数の絶対値が大きいトップ 50 の遺伝子を取得
topN = 50
C2_genes['abs weights'] = np.abs(C2_genes['weights'].values)
C2_genes = C2_genes.sort_values(by=['abs weights'], ascending=False)
C2_genes = C2_genes.head(topN)
C2_genes = C2_genes.sort_values(by=['weights'])
C2_genes['labels'] = C2_genes.index + ' --- ' + gene_products.loc[C2_genes.index, 'product'
C2_genes.plot.barh(y='weights', x='labels', figsize=(9, 16))
```

Out[24]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a181619e8>
```

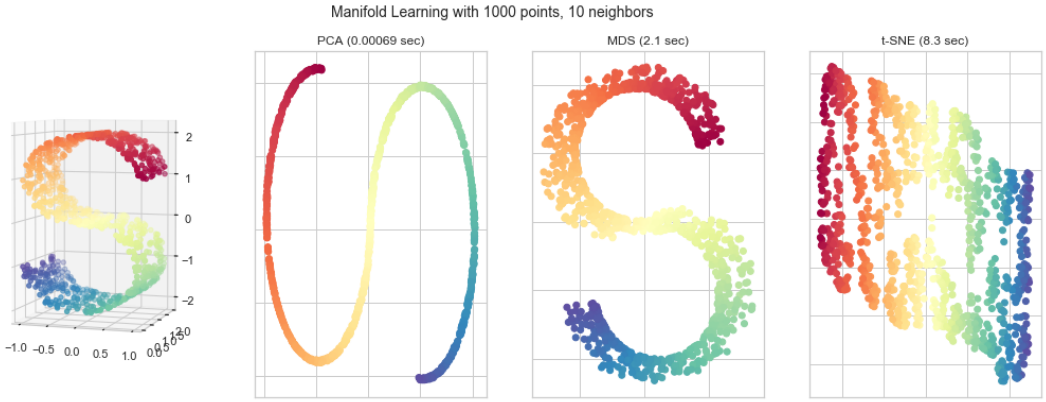
2.2 距離行列の最適化に基づく次元削減

In [25]:
```
#非線形次元圧縮、各手法の比較
#他の手法は以下の URL を参照
#https://scikit-learn.org/stable/auto_examples/manifold/plot_compare_methods.html

from time import time
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.ticker import NullFormatter
from sklearn import manifold, datasets
Axes3D

n_points = 1000
X, color = datasets.samples_generator.make_s_curve(n_points, random_state=0)
n_neighbors = 10
n_components = 2
```

```
fig = plt.figure(figsize=(18, 6))
plt.suptitle("Manifold Learning with %i points, %i neighbors" % (1000, n_neighbors), fontsi

ax = fig.add_subplot(141, projection='3d')
ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=color, cmap=plt.cm.Spectral)
ax.view_init(4, -72)

t0 = time()
pca = sklearn.decomposition.PCA()
Y = pca.fit_transform(X)
t1 = time()
print("PCA: %.2g sec" % (t1 - t0))
ax = fig.add_subplot(142)
plt.scatter(Y[:, 0], Y[:, 1], c=color, cmap=plt.cm.Spectral)
plt.title("PCA (%.2g sec)" % (t1 - t0))
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())
plt.axis('tight')

t0 = time()
mds = manifold.MDS(n_components, max_iter=100, n_init=1)
Y = mds.fit_transform(X)
t1 = time()
print("MDS: %.2g sec" % (t1 - t0))
ax = fig.add_subplot(143)
plt.scatter(Y[:, 0], Y[:, 1], c=color, cmap=plt.cm.Spectral)
plt.title("MDS (%.2g sec)" % (t1 - t0))
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())
plt.axis('tight')

t0 = time()
tsne = manifold.TSNE(n_components=n_components, init='pca', random_state=0)
Y = tsne.fit_transform(X)
t1 = time()
print("t-SNE: %.2g sec" % (t1 - t0))
ax = fig.add_subplot(144)
plt.scatter(Y[:, 0], Y[:, 1], c=color, cmap=plt.cm.Spectral)
plt.title("t-SNE (%.2g sec)" % (t1 - t0))
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())
plt.axis('tight')
```

```
        plt.show()
```

```
PCA: 0.00069 sec
MDS: 2.1 sec
t-SNE: 8.3 sec
```



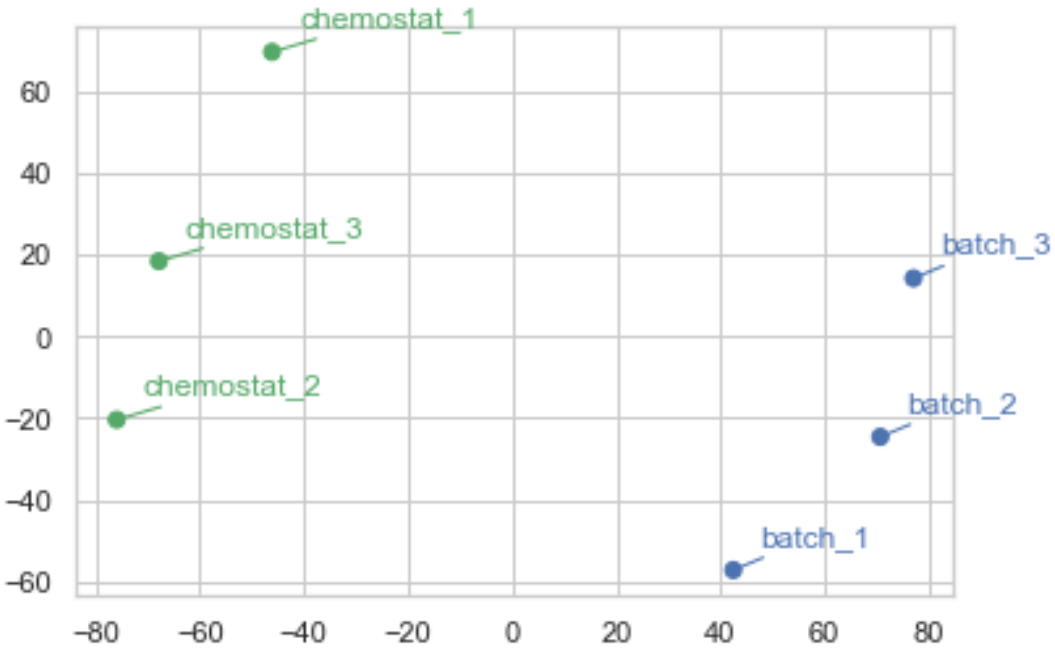
2.2.1 MDS (Multidimensional scaling; 多次元尺度構成法)

■metric MDS = PCoA (Principal Coordinate Analysis; 主座標分析) 構成する「距離行列」が、距離関数の要件を満たしている場合に適用可能な手法。

Bray-Curtis dissimilarity など、三角不等式の要件を満たさない非類似性指標があるので注意。こういう場合は PCoA ではなく nMDS を使う。

```
In [26]: import sklearn.manifold
```

```
In [27]: # 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
         # 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)
```

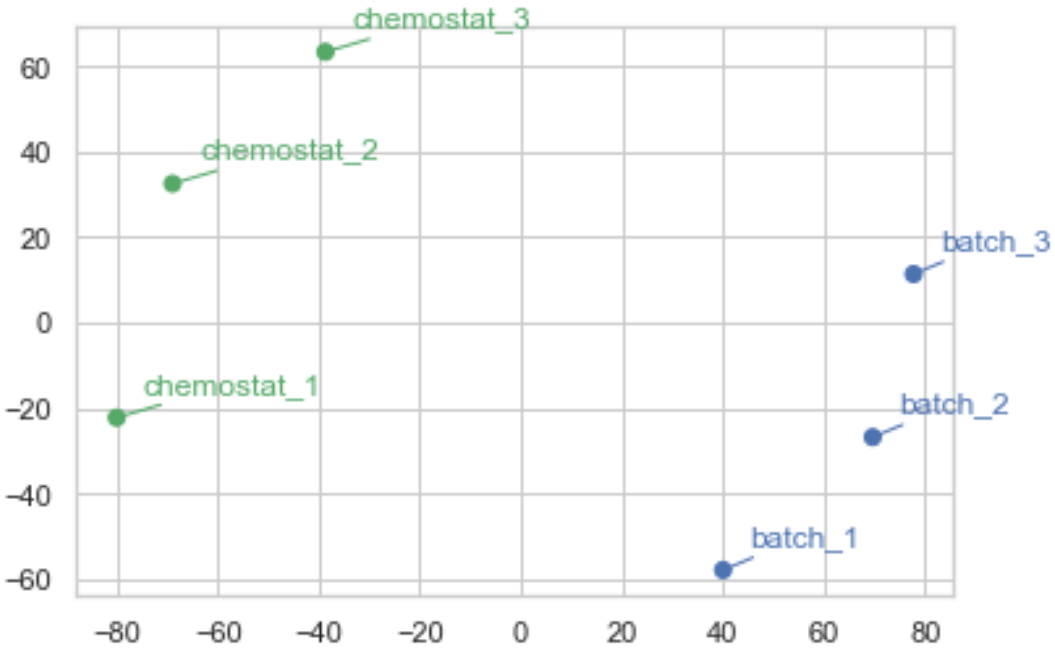


In [28]:
```
# 距離行列の計算
from scipy.spatial.distance import pdist, squareform
distance_matrix = squareform(pdist(std_df.transpose().values))
print(distance_matrix)

#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))
```

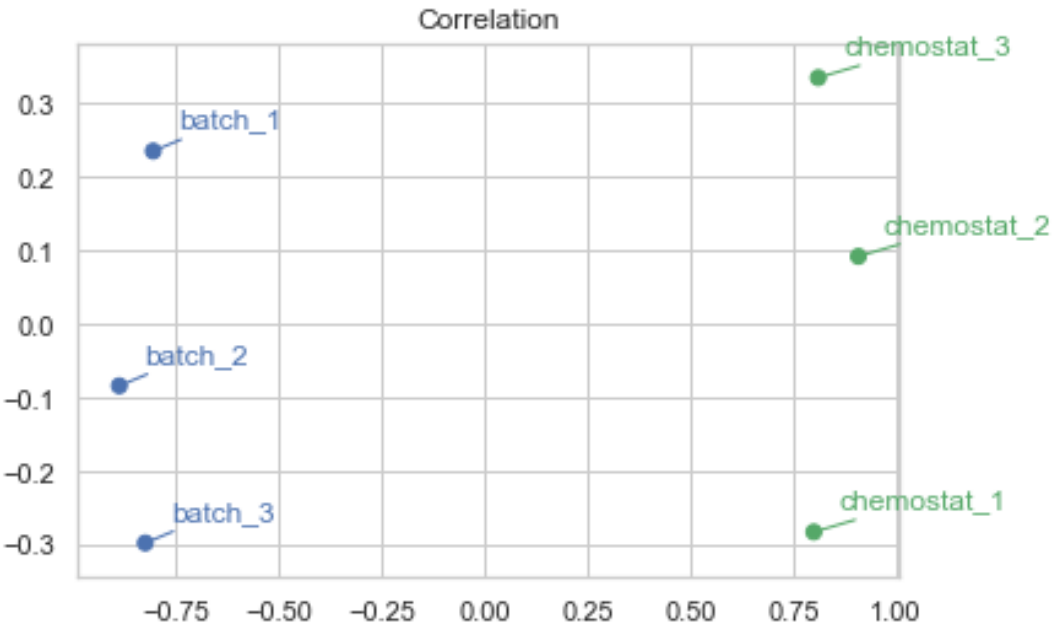
```
[[  0.          61.3886715   69.82784729 141.72006779 139.27185618
  133.98870325]
 [ 61.3886715    0.          56.11255212 148.032513   143.22201069
  140.71825784]
 [ 69.82784729  56.11255212   0.         149.09664857 146.08961444
  143.53272672]
 [141.72006779 148.032513   149.09664857   0.          76.11398372
   80.8830421 ]
 [139.27185618 143.22201069 146.08961444  76.11398372   0.
```

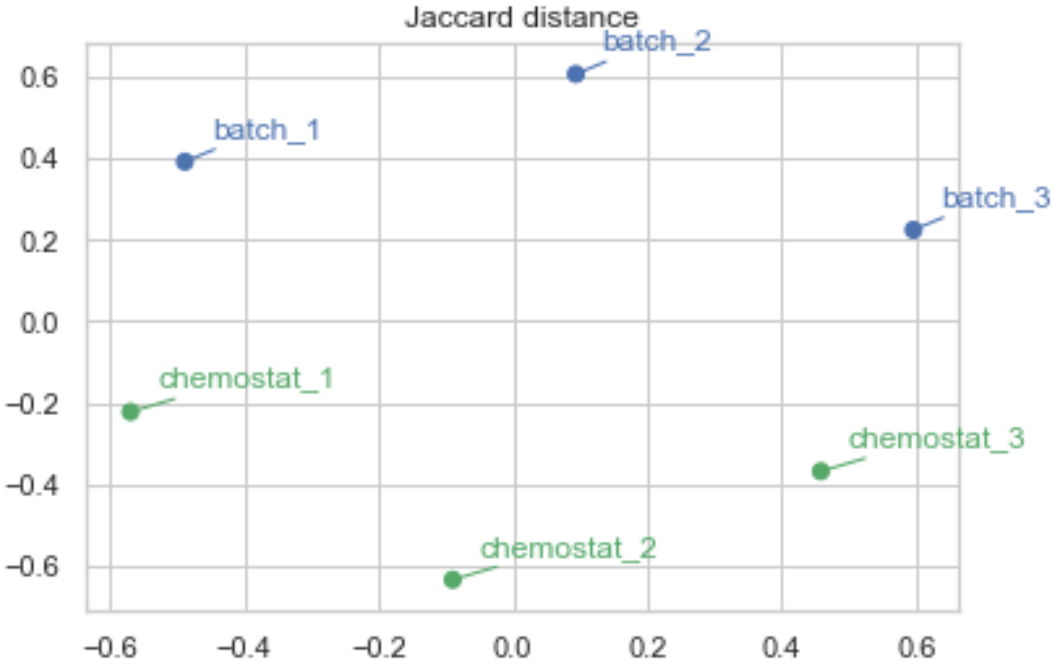
```
  64.40382016]
 [133.98870325 140.71825784 143.53272672  80.8830421   64.40382016
   0.        ]]
```

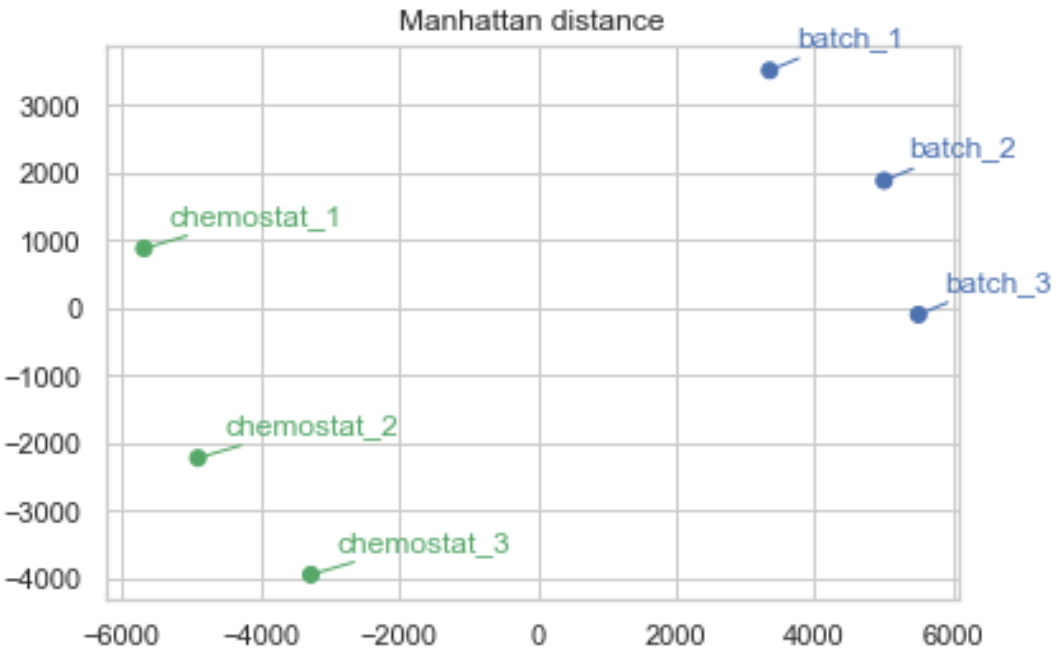
In [29]:
```
mds = sklearn.manifold.MDS(n_components=2, dissimilarity='precomputed')
coords = mds.fit_transform(distance_matrix)
scatter_plot(coords, std_df.columns, colors)
```



In [30]:
```
# 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')
```

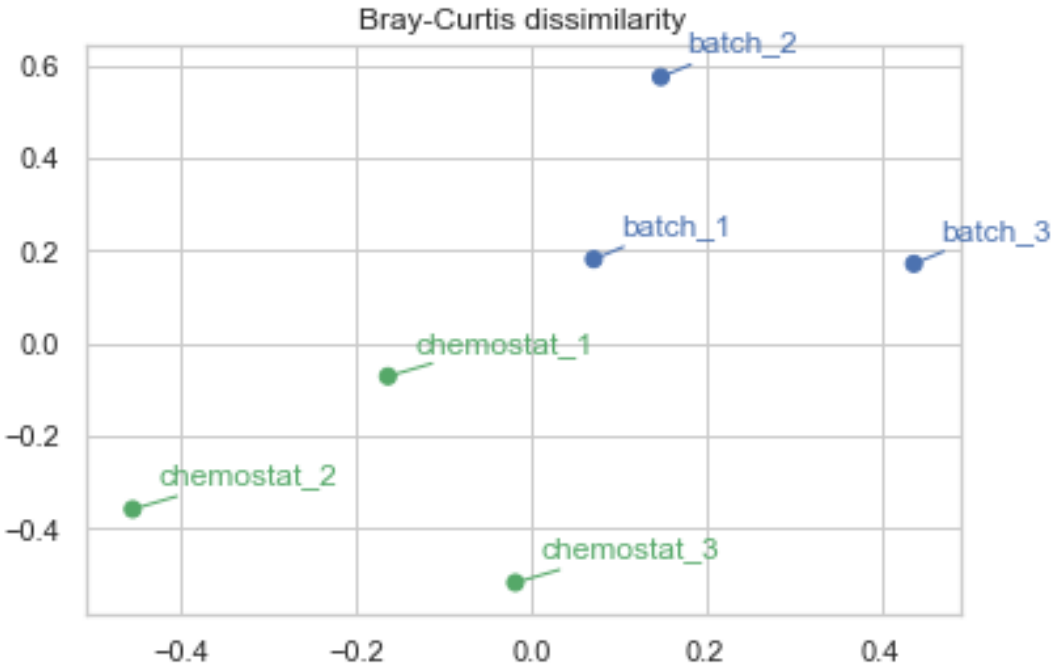

Correlation
chemostat_3
batch_1
chemostat_2
batch_2
batch_3
chemostat_1
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00


Jaccard distance
batch_2
batch_1
batch_3
chemostat_1
chemostat_3
chemostat_2
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.6
−0.4
−0.2
0.0
0.2
0.4
0.6



■non-metric MDS (非計量多次元尺度構成法)　構成する「距離行列」が、距離関数の要件を満たさない場合に適用する。

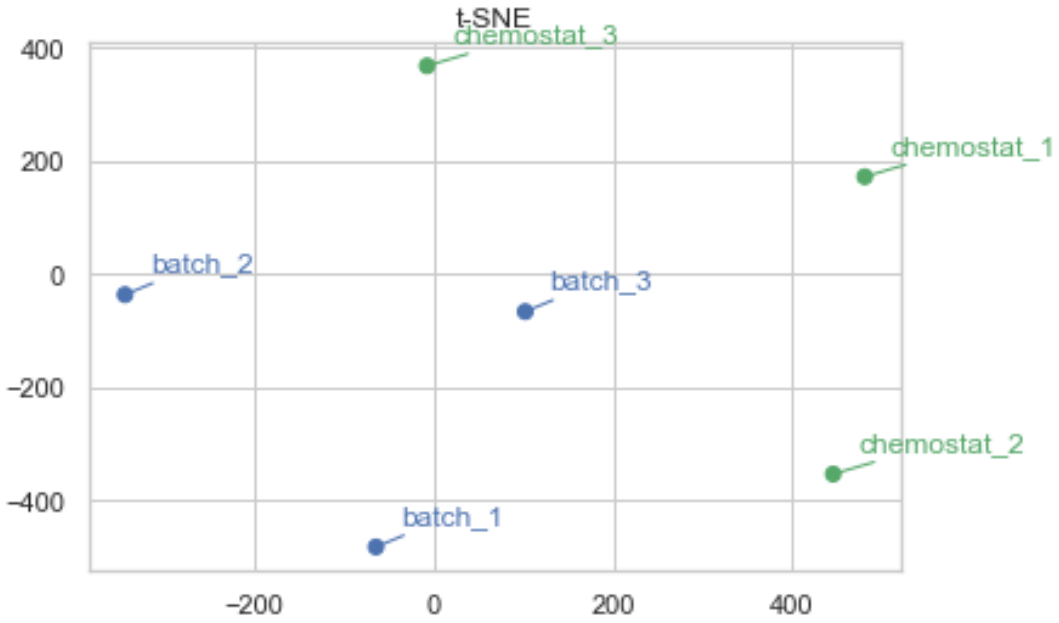
```
In [31]: # Bray-Curtis 非類似度指標による計算
         distance_matrix = squareform(pdist(std_df.transpose().values, 'braycurtis'))
         nmds = sklearn.manifold.MDS(n_components=2, metric=False, dissimilarity='precomputed')
         coords = nmds.fit_transform(distance_matrix)
         scatter_plot(coords, std_df.columns, colors, title='Bray-Curtis dissimilarity')
```



2.2.2　多様体学習

■t-SNE (t-distributed Stochastic Neighbor Embedding)

```
In [32]: 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')
```

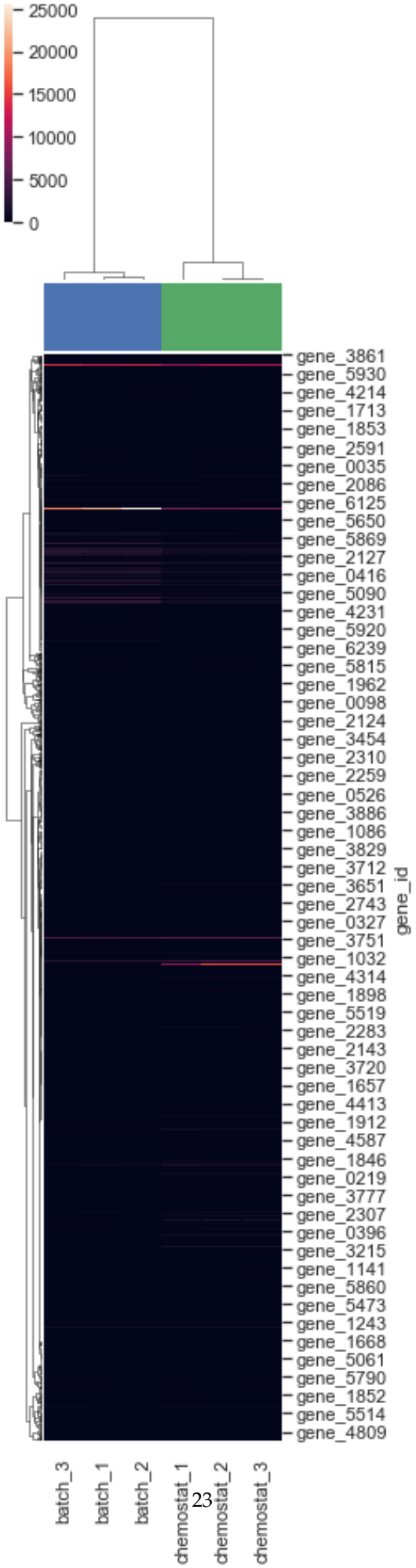


3　クラスタリング

3.1　階層的クラスタリング

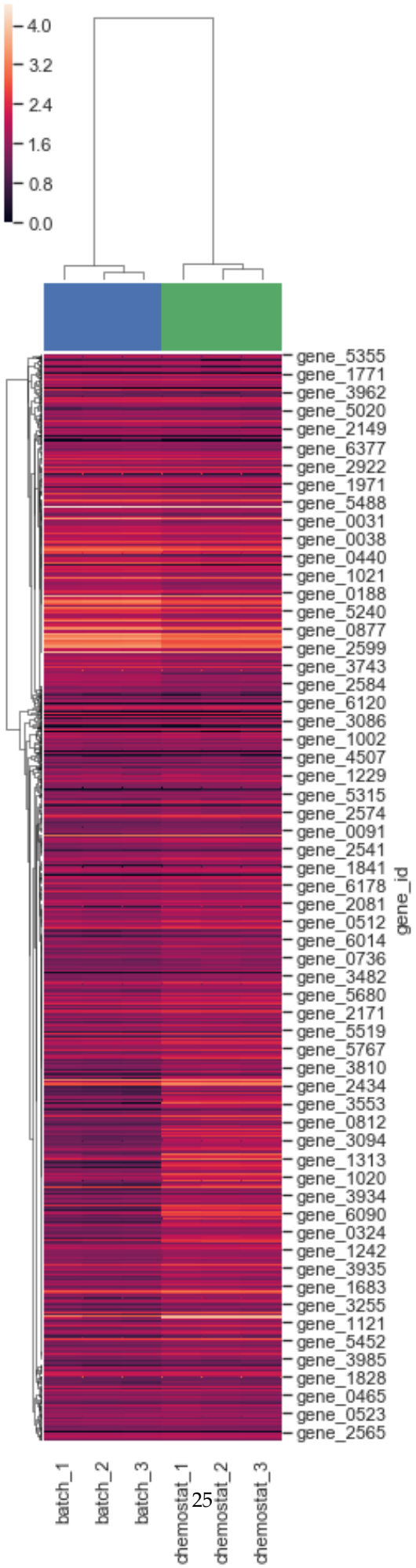
```
In [33]: sns.clustermap(df, method='average', metric='correlation', col_colors=colors, figsize=(3, 1
```

```
Out[33]: <seaborn.matrix.ClusterGrid at 0x1a181114e0>
```


25000
20000
15000
10000
5000
0
gene_3861
gene_5930
gene_4214
gene_1713
gene_1853
gene_2591
gene_0035
gene_2086
gene_6125
gene_5650
gene_5869
gene_2127
gene_0416
gene_5090
gene_4231
gene_5920
gene_6239
gene_5815
gene_1962
gene_0098
gene_2124
gene_3454
gene_2310
gene_2259
gene_0526
gene_3886
gene_1086
gene_3829
gene_3712
gene_3651
gene_2743
gene_0327
gene_3751
gene_1032
gene_4314
gene_1898
gene_5519
gene_2283
gene_2143
gene_3720
gene_1657
gene_4413
gene_1912
gene_4587
gene_1846
gene_0219
gene_3777
gene_2307
gene_0396
gene_3215
gene_1141
gene_5860
gene_5473
gene_1243
gene_1668
gene_5061
gene_5790
gene_1852
gene_5514
gene_4809
gene_id
batch_3
batch_1
batch_2
chemostat_1
chemostat_2
chemostat_3

In [34]:
```
log_transform = lambda x: np.log10(x + 1.0)
sns.clustermap(df.apply(log_transform), method='average', metric='correlation', col_colors=
```

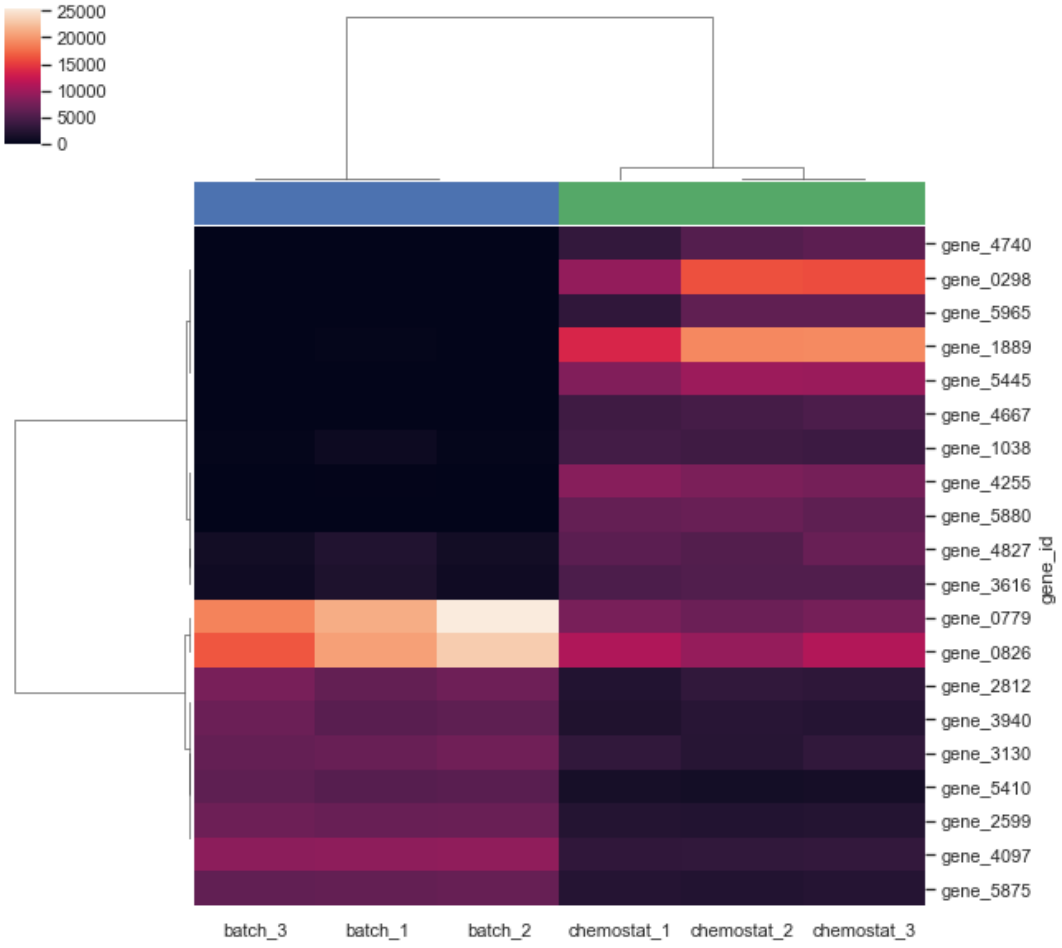
Out[34]:
```
<seaborn.matrix.ClusterGrid at 0x1a15f94748>
```

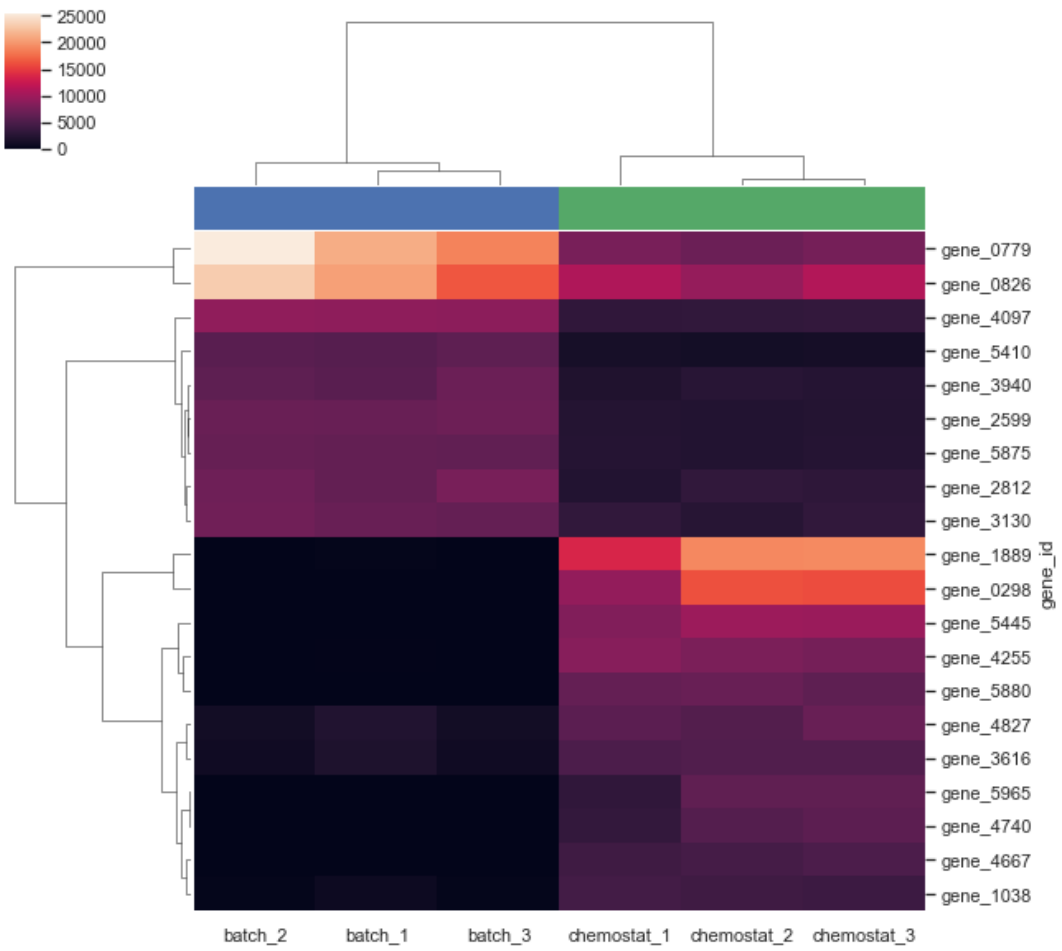

4.0
3.2
2.4
1.6
0.8
0.0
gene_5355
gene_1771
gene_3962
gene_5020
gene_2149
gene_6377
gene_2922
gene_1971
gene_5488
gene_0031
gene_0038
gene_0440
gene_1021
gene_0188
gene_5240
gene_0877
gene_2599
gene_3743
gene_2584
gene_6120
gene_3086
gene_1002
gene_4507
gene_1229
gene_5315
gene_2574
gene_0091
gene_2541
gene_1841
gene_6178
gene_2081
gene_0512
gene_6014
gene_0736
gene_3482
gene_5680
gene_2171
gene_5519
gene_5767
gene_3810
gene_2434
gene_3553
gene_0812
gene_3094
gene_1313
gene_1020
gene_3934
gene_6090
gene_0324
gene_1242
gene_3935
gene_1683
gene_3255
gene_1121
gene_5452
gene_3985
gene_1828
gene_0465
gene_0523
gene_2565
gene_id
batch_1
batch_2
batch_3
chemostat_1
chemostat_2
chemostat_3

In [35]:
```
# 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], :]
```

In [36]:
```
# 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)
```

Out[36]: <seaborn.matrix.ClusterGrid at 0x1a1a705c88>





3.2 K-means & K-medoids （クラスタ「中心」との距離に基づくクラスタリング）

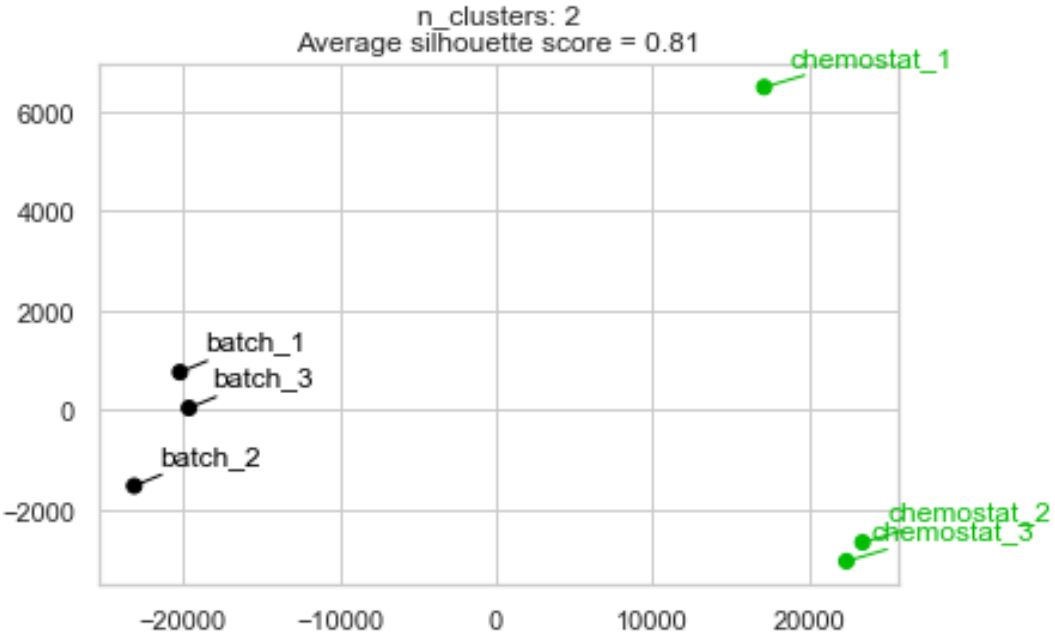
3.2.1 K-means clustering

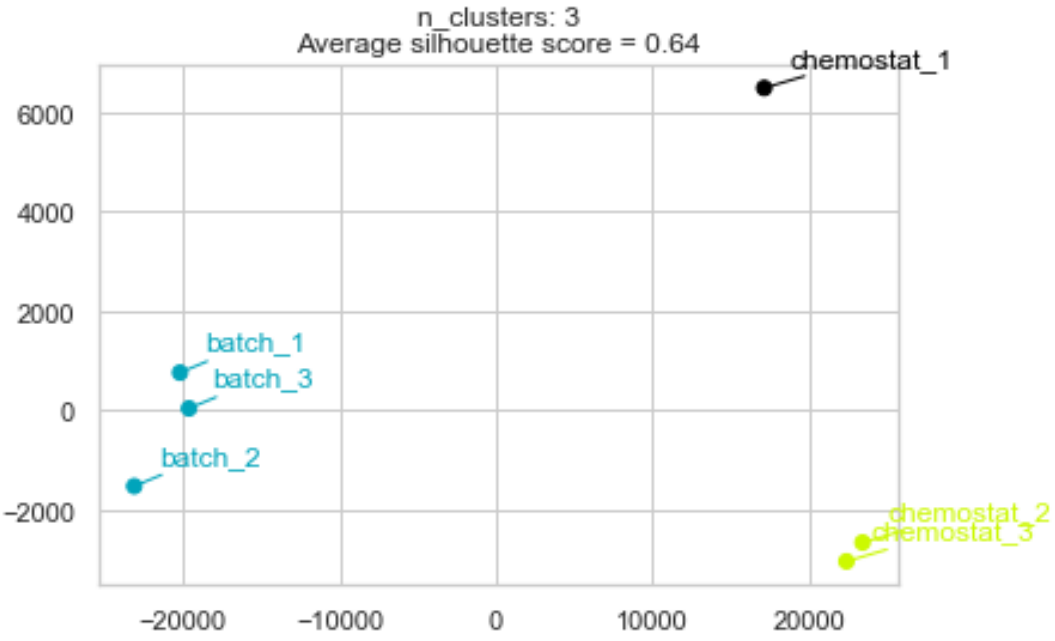
```
In [37]: import sklearn.cluster
         model = sklearn.cluster.KMeans(n_clusters=2)
         y = model.fit_predict(df.transpose().values)
         print(y)
```

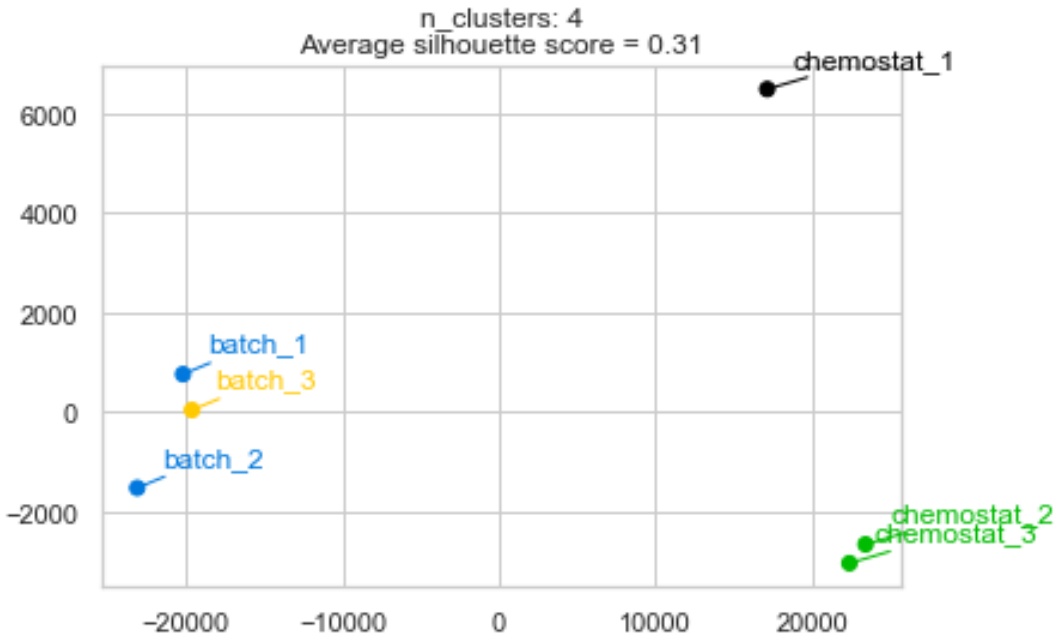
```
[1 1 1 0 0 0]
```

```
In [38]: # silhouette 解析による適切なクラスター数の推定
         from sklearn.metrics import silhouette_score
         import matplotlib.cm as cm

         X = df.transpose().values
         pca = sklearn.decomposition.PCA()
         pca_coords = pca.fit_transform(X)
```

```
for n_clusters in [2, 3, 4]:
    model = sklearn.cluster.KMeans(n_clusters=n_clusters)
    y = model.fit_predict(X)
    silhouette_avg = silhouette_score(X, y)
    title = 'n_clusters: {}\nAverage silhouette score = {:.2f}'.format(n_clusters, silhouet
    cluster_colors = cm.nipy_spectral(y.astype(float) / n_clusters)
    scatter_plot(pca_coords, std_df.columns, cluster_colors, title=title)
```



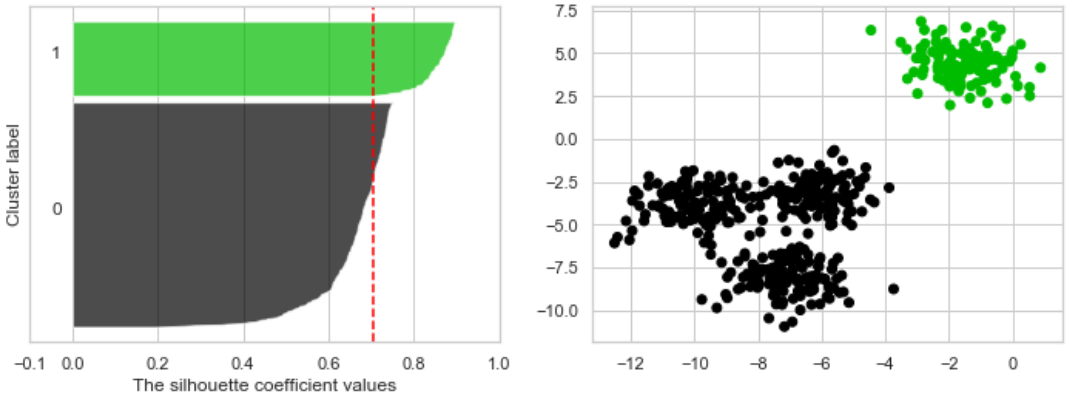




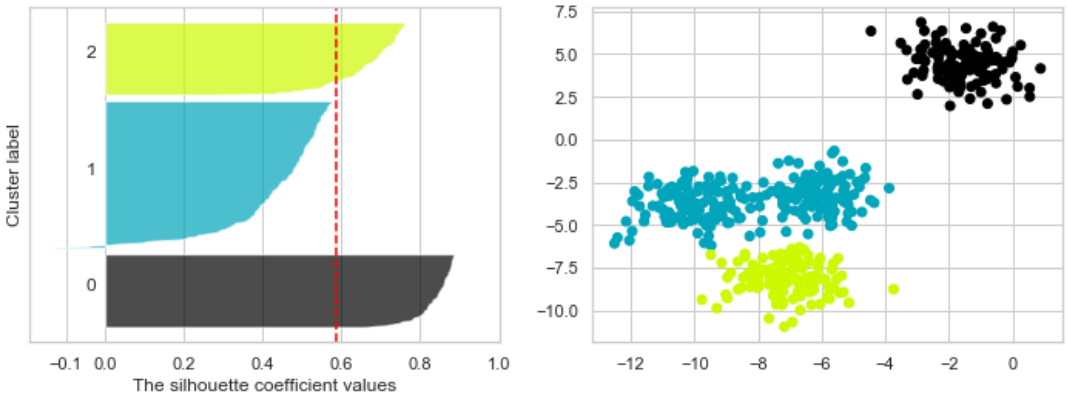
In [39]:
```
# もっとサンプル数がたくさんある場合は、サンプルごとのシルエットスコアの分布を比較できる
# https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.htm
from sklearn.datasets import make_blobs
from sklearn.metrics import silhouette_samples, silhouette_score
import matplotlib.cm as cm

X, y = make_blobs(n_samples=500, n_features=2, centers=4, cluster_std=1, center_box=(-10.0,

def plot_silhouette(n_clusters):
    fig = plt.figure(figsize=(12,4))
    ax1 = fig.add_subplot(121)
    ax2 = fig.add_subplot(122)
    model = sklearn.cluster.KMeans(n_clusters=n_clusters)
    cluster_labels = model.fit_predict(X)
    silhouette_avg = silhouette_score(X, cluster_labels)
    print('n_clusters:',n_clusters,' Average silhouette score =',silhouette_avg)
    sample_silhouette_values = silhouette_samples(X, cluster_labels)
    y_lower = 10
    for i in range(n_clusters):
        ith_cluster_silhouette_values = np.sort(sample_silhouette_values[cluster_labels ==
        size_cluster_i = ith_cluster_silhouette_values.shape[0]
        y_upper = y_lower + size_cluster_i
        cluster_colors = cm.nipy_spectral(float(i) / n_clusters)
```

```
            ax1.fill_betweenx(np.arange(y_lower, y_upper), 0, ith_cluster_silhouette_values, fa
            ax1.text(-0.05, y_lower + 0.5 * size_cluster_i, str(i))
            y_lower = y_upper + 10  # 10 for the 0 samples
        ax1.set_xlabel("The silhouette coefficient values")
        ax1.set_ylabel("Cluster label")
        ax1.axvline(x=silhouette_avg, color="red", linestyle="--")
        ax1.set_yticks([])
        ax1.set_xticks([-0.1, 0, 0.2, 0.4, 0.6, 0.8, 1])
        cluster_colors = cm.nipy_spectral(cluster_labels.astype(float) / n_clusters)
        ax2.scatter(X[:, 0], X[:, 1], c=cluster_colors)
        plt.show()

    for n_clusters in [2, 3, 4]:
        plot_silhouette(n_clusters)
```

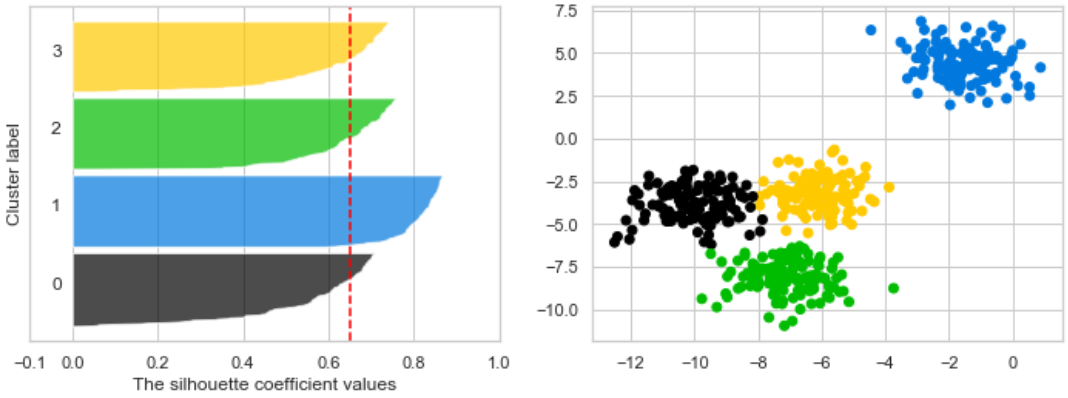
```
n_clusters: 2  Average silhouette score = 0.7049787496083262
```




```
n_clusters: 3  Average silhouette score = 0.5882004012129721
```



```
n_clusters: 4  Average silhouette score = 0.6505186632729437
```



3.2.2 K-medoids clustering

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、距離行列があればいい

→ ユークリッド距離である必要はなく、任意の距離関数を使える。

In [40]:
```
# K-medoids clustering を scikit-learn の使用感っぽく使えるように実装した。
# エラーハンドリングとか全然してないので注意。また、初期値依存性も高い。sklearn の KMeans メソッドは初期
# 参考：https://www.researchgate.net/publication/272351873_NumPy_SciPy_Recipes_for_Data_Scie

import numpy as np
from scipy.spatial.distance import pdist, squareform

class KMedoids():
    """K-Medoids clustering

    Parameters
    ----------
    n_clusters : int, optional, default: 2
        Number of clusters to form.
    max_iter : int, optional, default: 100
        Maximum number of iterations of the k-medoids algorithm.
    verbose : boolean, optional, default: False
        Verbosity mode.
    random_state : int, RandomState instance or None, optional, default: None
    dissimilarity : 'euclidean' | 'precomputed', optional, default: 'euclidean'
        Dissimilarity measure to use:
```

```
        - 'euclidean':
            Pairwise Euclidean distances between points in the dataset.
        - 'precomputed':
            Pre-computed dissimilarities are passed.

    Attributes
    ----------
    cluster_medoids_ : array, [n_clusters]
        Indices of cluster medoids
    labels_ :
        Labels of each point
    """
    def __init__(self, n_clusters=2, max_iter=100, verbose=False,
                 random_state=None, dissimilarity='euclidean'):
        self.n_clusters = n_clusters
        self.max_iter = max_iter
        self.verbose = verbose
        self.random_state = random_state
        self.dissimilarity = dissimilarity

    def _init_medoids(self, X, n_clusters, random_state):
        n_samples = X.shape[0]
        init_indices = random_state.choice(n_samples, n_clusters, replace=False)
        return np.sort(init_indices)

    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

    def fit(self, X):
        """Compute k-medoids clustering.
        Parameters
        ----------
        X : array, shape (n_samples, n_features) or (n_samples, n_samples)
            Input data. If ``dissimilarity == 'precomputed'``, the input should
            be the dissimilarity matrix.
        """
        if self.dissimilarity == 'precomputed':
            self.dissimilarity_matrix_ = X
        elif self.dissimilarity == 'euclidean':
            self.dissimilarity_matrix_ = squareform(pdist(X))
        else:
            raise ValueError("dissimilarity must be 'precomputed' or 'euclidean'.")
        if self.random_state is None or self.random_state is np.random:
            self.random_state = np.random.mtrand._rand
        elif isinstance(self.random_state, int):
            self.random_state = np.random.RandomState(self.random_state)
        elif isinstance(self.random_state, np.random.RandomState):
            self.random_state = np.random.RandomState
        else:
            raise ValueError("%r cannot be used to seed a numpy.random.RandomState" % self.
        self.cluster_medoids_, self.labels_ = \
            self._k_medoids(self.dissimilarity_matrix_, self.n_clusters, self.max_iter, sel
        return self

    def fit_predict(self, X):
        """Compute k-medoids clustering and predict cluster index for each sample.
        """
        return self.fit(X).labels_
```

```
In [41]: # 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
         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)
```

```
initial medoids = [1 4]
        iteration: 0
                new medoids= [0 5]
        iteration: 1
                new medoids= [0 5]

Medoids : ['batch_1' 'chemostat_3']
Clusters :
        cluster- 0 :  ['batch_1' 'batch_2' 'batch_3']
        cluster- 1 :  ['chemostat_1' 'chemostat_2' 'chemostat_3']
```

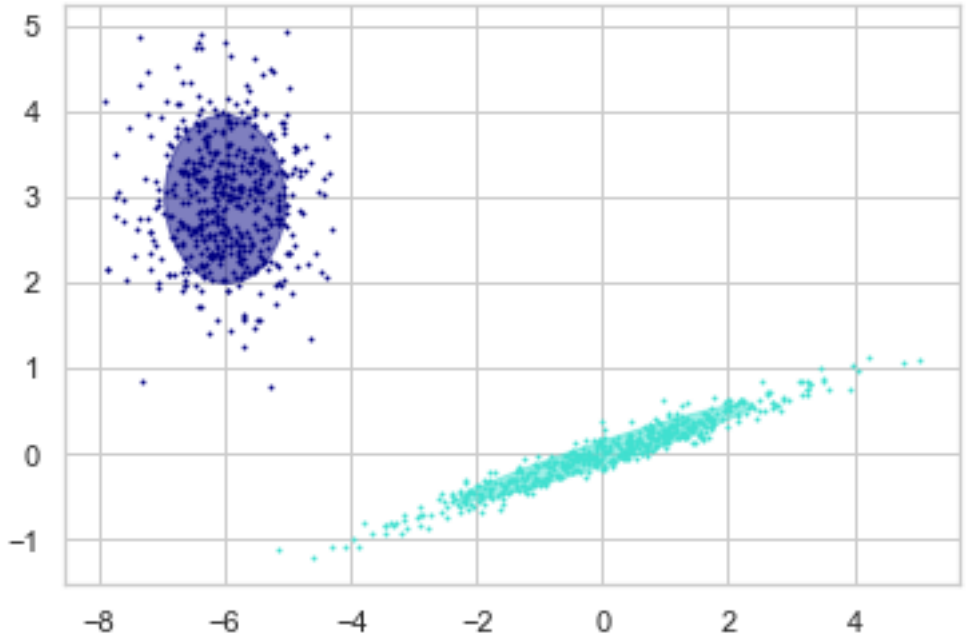
3.3 混合正規分布（確率分布に基づくクラスタリング）

実データが、なんからの確率分布がいくつか混ざった混合確率分布から生成されたものであると仮定する手法。

微生物群集構造データではここ数年、Dirichlet Multinomial Mixture がよく使われている。

```
In [42]: 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)
```

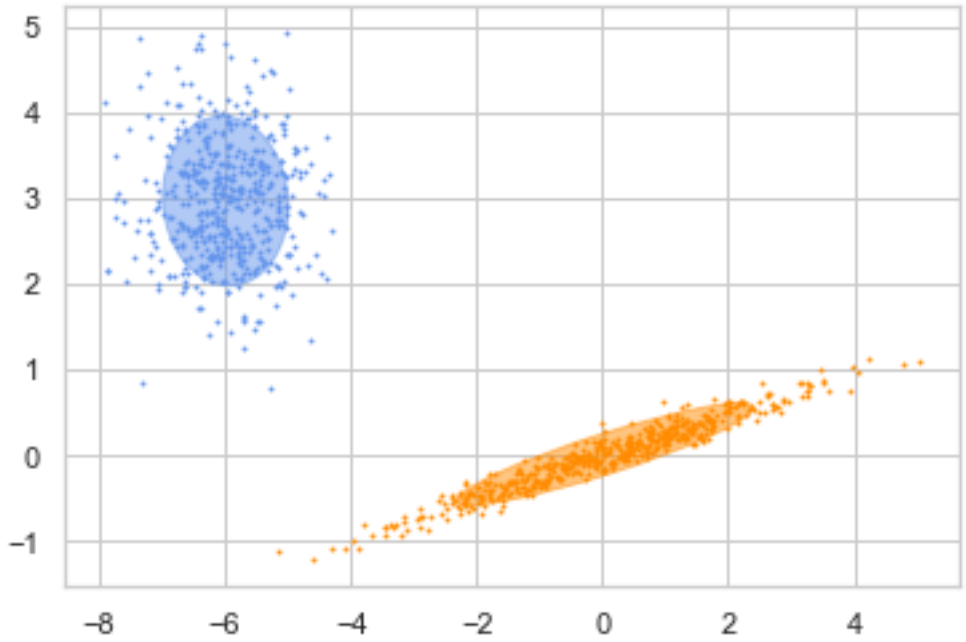
```
Y_ = gmm.predict(X)
cluster_colors = ['navy', 'turquoise']
for i, (mean, cov, color) in enumerate(zip(gmm.means_, gmm.covariances_, cluster_colors)):
    v, w = np.linalg.eigh(cov)
    if not np.any(Y_ == i):
        continue
    ax.scatter(X[Y_ == i, 0], X[Y_ == i, 1], .8, color=color)
    angle = 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)
```



In [43]:
```
import itertools
cluster_colors = itertools.cycle(['navy', 'turquoise', 'cornflowerblue', 'darkorange', 'gra

#  n_compoments=10 とすることで、実際の構造よりかなり大きな 10 個ぶんのクラスタを仮定して推論をはじめ
# 最終的には、2 つ以外のクラスタが推論の過程で勝手につぶれてくれる
vbgmm = sklearn.mixture.BayesianGaussianMixture(n_components=10, max_iter=300,
                                                covariance_type='full', weight_concentratio
vbgmm.fit(X)

for i, cluster_weight in enumerate(vbgmm.weights_):
```

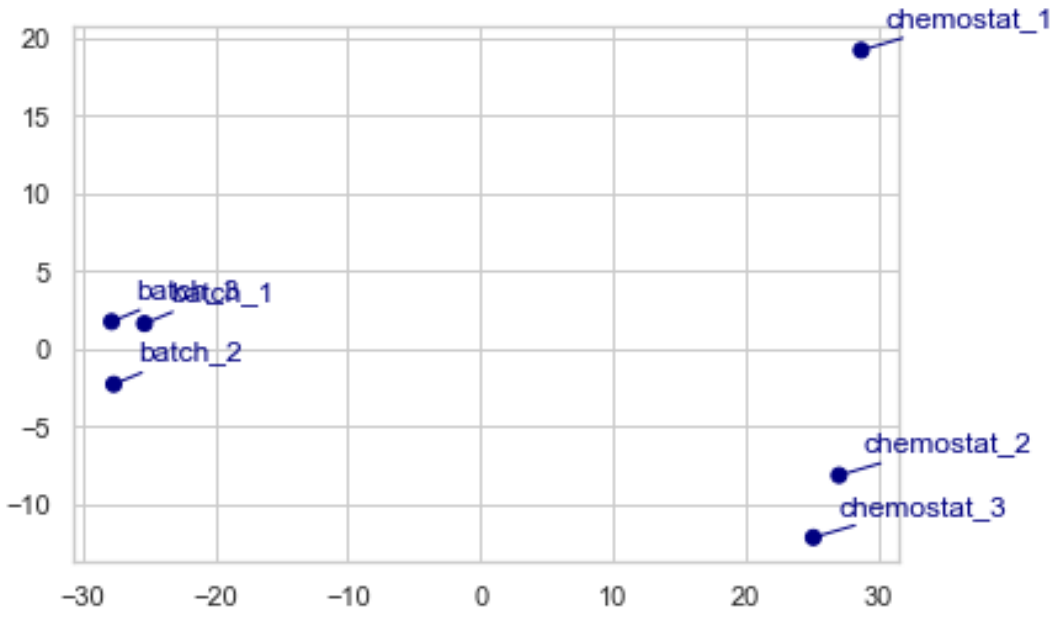
```
            print('Cluster-', i, ' weight =', cluster_weight)

        plt.figure()
        ax = plt.subplot(111)
        Y_ = vbgmm.predict(X)
        for i, (mean, cov, color) in enumerate(zip(vbgmm.means_, vbgmm.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)
```

```
Cluster- 0  weight = 9.990108501802354e-05
Cluster- 1  weight = 9.990108501800333e-05
Cluster- 2  weight = 0.49960039630663616
Cluster- 3  weight = 0.49960039500975884
Cluster- 4  weight = 9.990108501800333e-05
Cluster- 5  weight = 9.990108501800333e-05
Cluster- 6  weight = 9.990108502979147e-05
Cluster- 7  weight = 9.990108501800333e-05
Cluster- 8  weight = 9.990108846700025e-05
Cluster- 9  weight = 9.990108501800333e-05
```

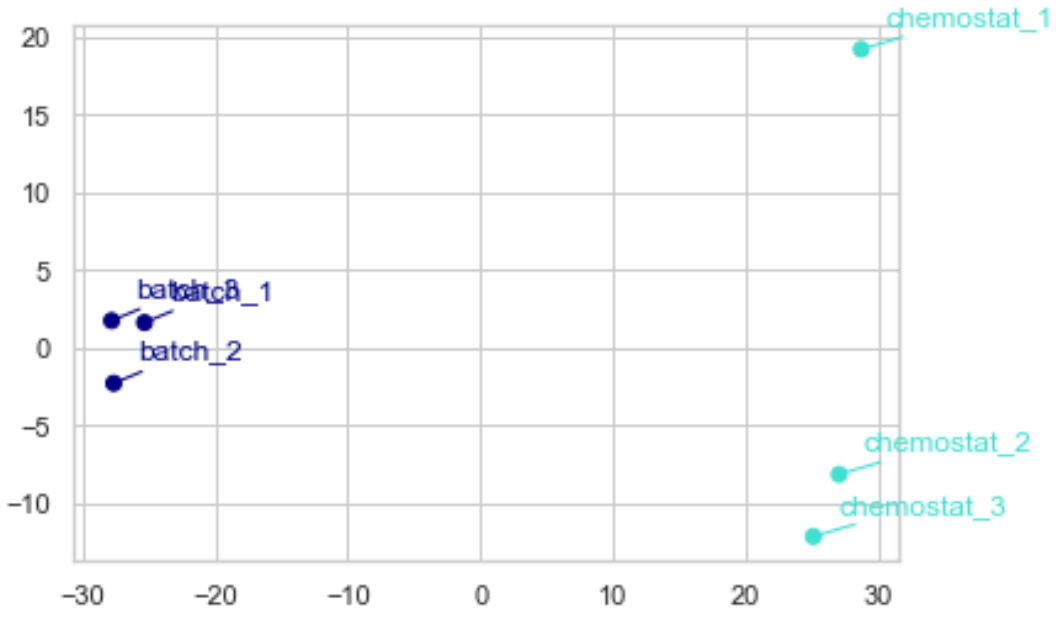


In [44]:
```
# サンプル間で分散の大きいトップ 1000 の遺伝子のみを使って推定
var1000_genes = std_df.var(axis=1).sort_values(ascending=False).index[:1000]
X = std_df.loc[var1000_genes, :].transpose().values

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='dirich
    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])
```

```
Number of clusters estimated: 1
        cluster- 0  weights = 1.0
```

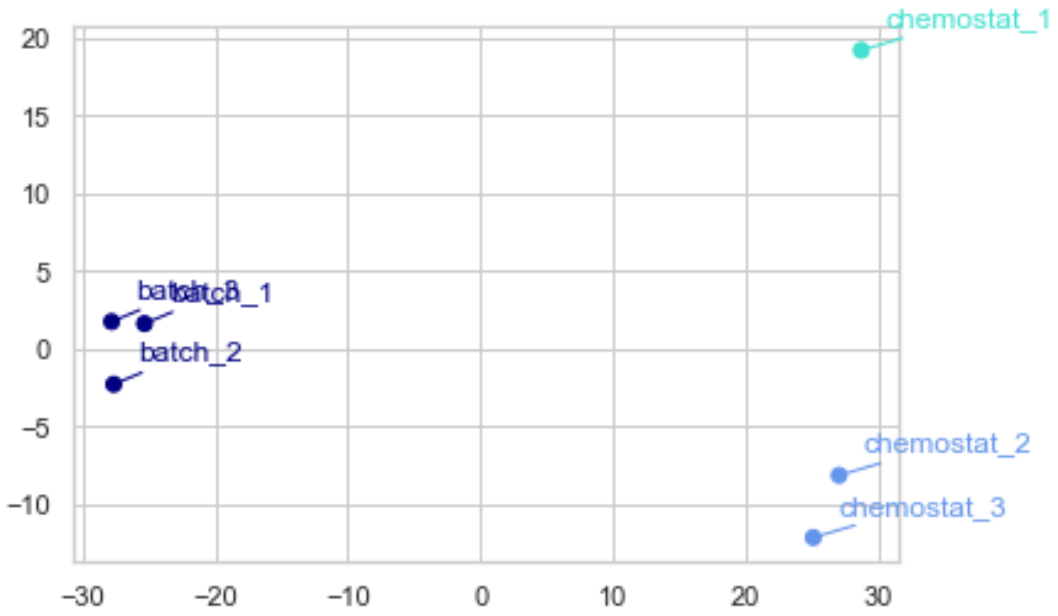



```
Number of clusters estimated: 2
        cluster- 0  weights = 0.5
        cluster- 1  weights = 0.5
```

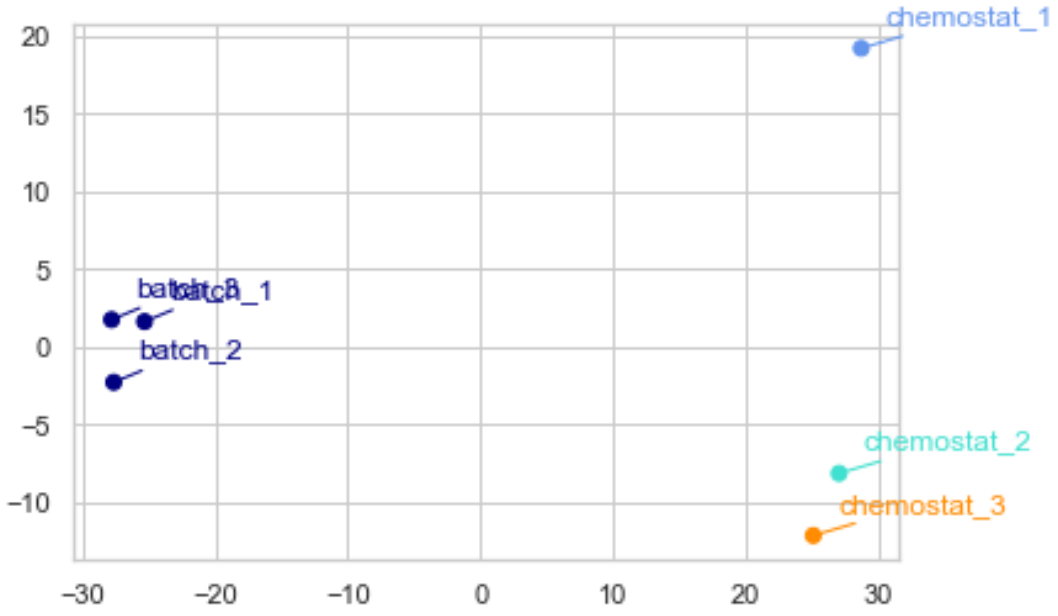



```
Number of clusters estimated: 3
        cluster- 0  weights = 0.47619047619047605
        cluster- 1  weights = 0.19047619047619058
```

```
        cluster- 2  weights = 0.3333333333333333
```




```
Number of clusters estimated: 4
        cluster- 0  weights = 0.464285714285714
        cluster- 1  weights = 0.17857142857142866
        cluster- 2  weights = 0.17857142857142866
        cluster- 3  weights = 0.17857142857142866
```



In [ ]: