Implementation of SSVD biclustering algorithm

Rumo Zhang, Xige Huang

Github: https://github.com/RumoZhang/STA663-SSVD

In [77]:
```
import scipy.linalg as la
import numpy as np
import seaborn as sns
import pandas as pd
from sklearn.decomposition import SparsePCA
```

Abstract

Sparse singular value decomposition (SSVD) is proposed as a SVD-based tool for biclustering or identifying interpretable row-column associations of the data matrices, especially for real-world data with high-dimension and low sample size (DHLSS). Specifically, SSVD provides low-rank, checkerboard structuered matrix approximation to the data through obtaining the sparse (contaning many zero entries) left- and right-singular vectors. The iterative SSVD algorithm utilizes the connection between SSVD and penalized regression using BIC. Optimization of the algorithm is conducted with the use of numba. A simulated data set, the lung cancer data introduced in the research paper, and a data set of gene expressions in study of Golub(1999) are used to illustrate the performance of the SSVD algorithm, along with comparative analysis with competing algorithms of standard SVD method and SPCA.

Package installation instruction is included at the end of the report

Key words: Biclustering; Sparse SVD; Adaptive Lasso penalty; PCA

Background

This project is a python realization of the algorithm proposed in *Biclustering via Sparse Singular Value Decomposition* by Lee, Shen, Huang, and Marron.

High-dimension and low sample size (DHLSS) data proposed a lot of challenge in statistical prediction in many fields. Among some of the useful tools when dealing with high-dimensional data, biclustring methods refers to a collection of unsupervised learning tools that simultaneously clusters the sets of samples (rows) and sets of variables (columns) in the data matrices to identify groups that are significantly associated. Lee, Shen, Huang, and Marron introduced SSVD as a new tool for biclustering. SSVD biclustering can be applied to fields include text mining, gene expression analysis, and numerous other biomedical analyzing tasks to detect important associations among the clusters.

In their research paper, Lee, Shen, Huang, and Marron utilizes special structure of SSVD to make the algorithm more efficient than the prior ones. Specifically, they suggest that selecting the penalty parameters in the adaptive Lasso regressions is equivalent to selecting

the degrees of sparsity in the singular vectors; the sparsity implies selection of relevant rows and columns when forming a low rank approximation to the data matrix.

In addition, SSVD can be more rigorous as a biclustering tool comparing to others such as SVD and PCA. (**?????**) As shown later in the comparative analysis section, SSVD has obvious advantages of detecting sparse structure when compared with SVD, and imposing sparse structure in both the penalized and unpenalized direction when comparing to the PCA methods.

The Algorithm

Overview

Let X be a n × d data matrix whose rows represent observations and columns represent variables. With the SVD decomposition of X, the first $K \leq r$ rank-one matrices in the summation is the closest rank-K approximation to X:

$$\mathbf{X} = \mathbf{UDV}^T = \sum_{k=1}^{r} s_k \boldsymbol{u}_k \boldsymbol{v}_k^T, \quad \mathbf{X}^{(K)} \equiv \sum_{k=1}^{K} s_k \boldsymbol{u}_k \boldsymbol{v}_k^T$$

The proposed SSVD algorithm is based on the idea above, but with sparsity-inducing penalities to make u_k and v_k sparse. The rank-one matrix $s_k u_k v_k^T$, with a checkerboard structure, is referred to as the first SSVD layer, and the sum of the first m layers provides the best sparse rank-m approximation to the data matrices. For the main purpose of this project and simplification, we will only describe the algorithm to compute SSVD, more details of the connection with variable selection for penalized regressions are presented in the original paper.

The iterative algorithm to compute SSVD layer

Step 0: Choose non-negative parameters γ_1 and γ_2

Step 1: Obtain the SVD composition of the n x d data matrix $\mathbf{X}$, denote by $\mathbf{X} = \mathbf{u}s\mathbf{v}^T$

Step 2:

1. Calculate $\boldsymbol{w}_2$ by

$$\boldsymbol{w}_2 \equiv (w_{2,1}, \ldots, w_{2,d})^T = |\hat{\tilde{v}}|^{-\gamma_2} = |\mathbf{X}^T u|^{-\gamma_2}$$

Obtain $\hat{\sigma^2}$, the OLS estimate of the error variance from the penalized regression

$$\left\|\mathbf{X} - \boldsymbol{u}\tilde{\boldsymbol{v}}^T\right\|_F^2 + s\sum_{j=1}^{d} w_{2,j}\,|v_j|$$

Calculate λ_v from $\lambda_v = \underset{\lambda_v}{argmin}\,\mathrm{BIC}(\lambda_v)$, where

$$\mathrm{BIC}(\lambda_v) = \frac{\|\mathbf{Y} - \hat{\mathbf{Y}}\|^2}{nd \cdot \hat{\sigma}^2} + \frac{\log(nd)}{nd}\hat{df}\,(\lambda_v)$$

2. Set $\tilde{v}_j = \text{sign}\left\{\left(\mathbf{X}^T\boldsymbol{u}\right)_j\right\}\left(\left|\left(\mathbf{X}^T\boldsymbol{u}\right)_j\right| - \lambda_v w_{2,j}/2\right)_+$, with λ_v from step2.1, $j = 1, \ldots, d$
3. Let $\tilde{v} = (\tilde{v}_1, \ldots, \tilde{v}_d)^T$, and $\boldsymbol{v}_{\text{new}} = \tilde{\boldsymbol{v}}/\|\tilde{\boldsymbol{v}}\|$

Note:

- $\mathbf{Y} = \left(\boldsymbol{x}_1^T, \ldots, \boldsymbol{x}_d^T\right)^T \in R^{nd}$ with x_j being the jth column of $\mathbf{X}$
- $\hat{\tilde{v}}$ is the OLS estimate of $\tilde{v}$

Step 3:

1. Calculate $\boldsymbol{w}_1$ by

$$\boldsymbol{w}_1 \equiv (w_{1,1}, \ldots, w_{1,n})^T = |\hat{\tilde{u}}|^{-\gamma_1} = |\mathbf{X}v|^{-\gamma_2}$$

Obtain $\hat{\sigma^2}$, the OLS estimate of the error variance from the penalized regression

$$\left\|\mathbf{X} - \tilde{\boldsymbol{u}}\boldsymbol{v}^T\right\|_F^2 + s\sum_{i=1}^{n} w_{1,i}\,|u_i|$$

Calculate λ_u from $\lambda_u = \underset{\lambda_u}{argmin}\,\text{BIC}(\lambda_u)$, where

$$\text{BIC}(\lambda_u) = \frac{\|\mathbf{Z} - \hat{\mathbf{Z}}\|^2}{nd \cdot \hat{\sigma}^2} + \frac{\log(nd)}{nd}\hat{df}\,(\lambda_u)$$

2. Set $\tilde{u}_i = \text{sign}\left\{(\mathbf{X}\boldsymbol{v}_{\text{new}})_i\right\}\left(\left|(\mathbf{X}\boldsymbol{v}_{\text{new}})_j\right| - \lambda_v w_{u,i}/2\right)_+$, with λ_u from step3.1, $i = 1, \ldots, n$
3. Let $\tilde{u} = (\tilde{u}_1, \ldots, \tilde{u}_n)^T$, and $\boldsymbol{u}_{\text{new}} = \tilde{\boldsymbol{u}}/\|\tilde{\boldsymbol{u}}\|$

Note:

- $\mathbf{Z} = \left(\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(n)}\right)^T \in R^{nd}$ with x_i^T being the ith row of $\mathbf{X}$
- $\hat{\tilde{u}}$ is the OLS estimate of $\tilde{u}$

Step 4: Repeat step 2 and step 3 with $\boldsymbol{u} = \boldsymbol{u}_{\text{new}}$, $\boldsymbol{v} = \boldsymbol{v}_{\text{new}}$ until convergence

Step 5: Obtain the $\boldsymbol{u}$, $\boldsymbol{v}$, s = $\boldsymbol{u}^T\mathbf{X}\boldsymbol{v}$ at convergence

Subsequent layers can be extracted sequentially from the residual matrices after removing the preceding layers. For our package, we choose to use $\gamma_1 = \gamma_2 = 2$ suggested by Zou (2006) as an default for the function `SSVD_single` .

Optimization of the algorithm

The implementation of the original algorithm

In [78]:
```
def SSVD(x, gamma1 = 2, gamma2 = 2, tol = 1e-6, max_iter = 50):
    """The SSVD algorithm for a single layer"""

    n, d = x.shape

    # Step1
    U, S, V = la.svd(x)
```

```
    Vt = V.T
    iters = 0
    converge_diff_u = tol + 1
    converge_diff_v = tol + 1

    U_old = U[:, 0]
    S_old = S[0]
    V_old = V[0, :]

    V_new = np.zeros(d)
    U_new = np.zeros(n)

    # Step 2
    while(converge_diff_u > tol and converge_diff_v > tol and iters < max_ite

        iters += 1
        #update v
        V_hat = np.zeros(d)
        Xt_U = x.T @ U_old
        omega_2 = np.abs(Xt_U) ** (-gamma2)
        error_var = np.abs(np.sum(x ** 2) - np.sum(Xt_U**2))/(n*d-d)
        lambda_2 = np.unique(np.append(np.abs(Xt_U / omega_2), 0))
        lambda_2.sort()

        min_bic = np.Inf

        for l in lambda_2:

            ## Find all v's
            term1 = Xt_U / abs(Xt_U)
            term2 = abs(Xt_U) - l * omega_2 / 2
            term2 *= term2 >= 0
            V_hat = term1 * term2

            ## Choose the best v based on bic
            bic = np.sum((x - U_old.reshape((-1, 1)) @
                          V_hat.reshape((1, -1)))**2) / error_var + np.sum(V_
            if bic < min_bic:
                V_new = V_hat
                min_bic = bic

        s = np.linalg.norm(V_new)
        V_new = V_new / s

        #update U
        U_hat = np.zeros(n)
        X_V = x @ V_old

        omega_1 = np.abs(X_V) ** (-gamma1)
        error_var = np.abs(np.sum(x ** 2) - np.sum(X_V**2))/(n*d-n)
        lambda_1 = np.unique(np.append(np.abs(X_V / omega_1), 0))
        lambda_1.sort()

        min_bic = np.Inf

        for l in lambda_1:

            term1 = X_V / abs(X_V)
            term2 = abs(X_V) - l * omega_1 / 2
            term2 *= term2 >= 0
            U_hat = term1 * term2

            ## Choose the best v based on bic
            bic = np.sum((x - U_hat.reshape((-1, 1)) @
                          V_old.reshape((1, -1)))**2)/error_var + np.sum(U_ha
```

```
            if bic < min_bic:
                U_new = U_hat
                min_bic = bic

        s = np.linalg.norm(U_new)
        U_new = U_new / s

        converge_diff_u = np.sqrt(np.sum((U_new - U_old) ** 2))
        converge_diff_v = np.sqrt(np.sum((V_new - V_old) ** 2))
        U_old = U_new
        V_old = V_new

    return U_new, S_old, V_new, iters
```

Below is the multi-layer SSVD algorithm implementation.

In [79]:

```
def SSVD_multi_layer(x, layers):
    """The SSVD algorithm for multiple layers"""

    n, d = x.shape
    all_layers_u = np.zeros((n, layers))
    all_layers_v = np.zeros((d, layers))
    all_layers_s = np.zeros(layers)

    for i in range(layers):
        u_new, s_new, v_new, iters = SSVD(x)
        layer = s_new * u_new.reshape((-1, 1)) @ v_new.reshape((1, -1))
        all_layers_u[:, i] = u_new
        all_layers_v[:, i] = v_new
        all_layers_s[i] = s_new
        x = x - layer

    return all_layers_u, all_layers_v
```

In [80]:

```
from numba import jit
import timeit
import multiprocessing as mp
from multiprocessing import Pool
from functools import partial
import numba
```

Since our algorithm could be applied in high dimensional data environment, we want to optimize our function in terms of run-time. In this section, we experimented with the `nopython` and the `object` mode in `numba` . We also tested the `multiprocessing` package.

In [81]:

```
@jit
def best_lmd_jit(l, x, x_given, given, omega, n, d, sigma_2, update):
    """Selecting lambda value using object mode in numba"""

    result = np.zeros(len(l))

    for i in range(len(l)):

        term1 = x_given / np.absolute(x_given)
        term2 = np.absolute(x_given) - l[i] * omega / 2
        term2 *= term2 >= 0
        temp = term1 * term2

        ind = np.where(temp!=0)
```

```
        if update == 'u':

            result[i] = np.sum((x - np.reshape(temp, (n, 1)) @
                                np.reshape(given, (1, d)))**2) / sigma_2 + np
        else:
            result[i] = np.sum((x - np.reshape(given, (n, 1))
                                @ np.reshape(temp, (1, d)))**2) / sigma_2 + n

    return np.argmin(result)
```

In [82]:

```
@numba.jit(nopython=True, warn = False)
def best_lmd_numba(l, x, x_given, given, omega, n, d, sigma_2, update):
    """Selecting lambda value using nopython mode in numba"""

    result = np.zeros(len(l))

    for i in range(len(l)):

        term1 = x_given / np.absolute(x_given)
        term2 = np.absolute(x_given) - l[i] * omega / 2
        term2 *= term2 >= 0
        temp = term1 * term2

        ind = np.where(temp!=0)
        sum_sq = 0
        if update == 'u':

            for j in range(d):
                for k in range(n):
                    sum_sq += (x[k, j] - temp[k] * given [n]) ** 2
            result[i] = sum_sq / sigma_2 + + np.sum(temp!=0) * np.log(n * d)

        else:

            for j in range(d):
                for k in range(n):
                    sum_sq += (x[k, j] - temp[k] * given [n]) ** 2
            result[i] = sum_sq / sigma_2 + + np.sum(temp!=0) * np.log(n * d)

    return np.argmin(result)
```

In [83]:

```
def SSVD_op(optm, x, gamma1 = 2, gamma2 = 2, tol = 1e-6, max_iter = 50):
    """The SSVD algorithm, adjusted to utilize the numba package"""

    n, d = x.shape

    # Step1
    U, S, V = np.linalg.svd(x)
    Vt = V.T
    iters = 0
    converge_diff_u = tol + 1
    converge_diff_v = tol + 1

    U_old = U[:, 0]
    S_old = S[0]
    V_old = V[0, :]

    V_new = np.zeros(d)
    U_new = np.zeros(n)

    # Step 2
    while(converge_diff_u > tol and converge_diff_v > tol and iters < max_ite
```

```
        iters += 1
        #update v
        V_hat = np.zeros(d)
        Xt_U = x.T @ U_old
        omega_2 = np.abs(Xt_U) ** (-gamma2)
        error_var = np.abs(np.sum(x ** 2) - np.sum(Xt_U**2))/(n*d-d)
        lambda_2 = np.unique(np.append(np.abs(Xt_U / omega_2), 0))
        lambda_2.sort()

        best_bic_v = optm(lambda_2, x, Xt_U, U_old, omega_2, n, d, error_var,
        V_new = (Xt_U / abs(Xt_U)) * (abs(Xt_U) - best_bic_v * omega_2 / 2) *
        s = np.linalg.norm(V_new)
        V_new = V_new / s

        #update U
        U_hat = np.zeros(n)
        X_V = x @ V_old

        omega_1 = np.abs(X_V) ** (-gamma1)
        error_var = np.abs(np.sum(x ** 2) - np.sum(X_V**2))/(n*d-n)
        lambda_1 = np.unique(np.append(np.abs(X_V / omega_1), 0))
        lambda_1.sort()

        best_bic_u = optm(lambda_1, x, X_V, V_old, omega_1, n, d, error_var,
        U_new = (X_V / abs(X_V)) * (abs(X_V) - best_bic_u * omega_1 / 2) * ((
        s = np.linalg.norm(U_new)
        U_new = U_new / s

        converge_diff_u = np.sqrt(np.sum((U_new - U_old) ** 2))
        converge_diff_v = np.sqrt(np.sum((V_new - V_old) ** 2))
        U_old = U_new
        V_old = V_new

    return U_new, S_old, V_new, iters
```

For the comparative purpose, we construct a simulation dataset for 1-layer SSVD decomposition. The details of this dataset will be discussed in the `Simulation` section.

In [84]:

```
u_tilde = np.concatenate((np.arange(3, 11)[::-1],
                          np.ones(17) * 2,
                          np.zeros(75)))
u = u_tilde / np.linalg.norm(u_tilde)
v_tilde = np.concatenate((np.array([10, -10, 8, -8, 5, -5]),
                              np.ones(5) * 3,
                              np.ones(5) * (-3),
                              np.zeros(34)))
v = v_tilde / np.linalg.norm(v_tilde)
s = 50
x_sim = s * u.reshape((-1, 1)) @ v.reshape((1, -1))
x_sim += np.random.normal(size = (x_sim.shape))
```

In [85]:

```
%timeit -r3 SSVD(x_sim)
```

```
91.9 ms ± 645 µs per loop (mean ± std. dev. of 3 runs, 10 loops each)
```

In [86]:

```
%timeit -r3 SSVD_op(best_lmd_jit, x_sim)
```

```
41.7 ms ± 2.4 ms per loop (mean ± std. dev. of 3 runs, 1 loop each)
```

Application to simulated data sets

Rank 1 simulation

In order to closely examine the performance of our algorithm, we decided to carry out simulation studies. In this section, we will start by repeating a rank-1 simulation study suggested by Lee et al. (2010). In this designed simulation dataset, the target matrix X has the dimension of 100×50. In a rank-1 approximation by our SSVD algorithm, the corresponding vectors u and v are designed to have 25 and 16 nonzero entries, respectively. The scaling eigenvalue associated with this case is set to be 50. The final dataset also has error terms from a standard normal distribution added. And the entire data generating process is repeated 100 times.

Ideally, we expect our algorithm to correctly detect the zero and nonzero entries from the original dataset. In this case, we collected the following statistics to evaluate its performance: number of zero entries, number of correctly specified zero entries and number of correctly specified nonzero entries.

In [87]:
```
def rank1_approx(M = 100):
    """Carry out a rank-1 approximation simulation study"""

    u_tilde = np.concatenate((np.arange(3, 11)[::-1],
                          np.ones(17) * 2,
                          np.zeros(75)))
    u = u_tilde / np.linalg.norm(u_tilde)
    v_tilde = np.concatenate((np.array([10, -10, 8, -8, 5, -5]),
                              np.ones(5) * 3,
                              np.ones(5) * (-3),
                              np.zeros(34)))
    v = v_tilde / np.linalg.norm(v_tilde)
    s = 50
    x_sim_rank1 = s * u.reshape((-1, 1)) @ v.reshape((1, -1))

    v_nonzero_ind = np.where(v != 0)
    v_zero_ind = np.where(v == 0)
    u_nonzero_ind = np.where(u != 0)
    u_zero_ind = np.where(u == 0)

    v_zeros = np.zeros(M)
    v_true_zeros = np.zeros(M)
    v_true_nonzeros = np.zeros(M)

    u_zeros = np.zeros(M)
    u_true_zeros = np.zeros(M)
    u_true_nonzeros = np.zeros(M)


    for i in range(M):
        noise = np.random.normal(size = (x_sim_rank1.shape))
        u_appr, s_, v_appr, iters = SSVD(x_sim_rank1 + noise)
        v_zeros[i] = np.sum(v_appr == 0)
        u_zeros[i] = np.sum(u_appr == 0)
        v_true_zeros[i] = np.sum(v_appr[v_zero_ind] == 0)
        u_true_zeros[i] = np.sum(u_appr[u_zero_ind] == 0)
        v_true_nonzeros[i] = np.sum(v_appr[v_nonzero_ind] != 0)
        u_true_nonzeros[i] = np.sum(u_appr[u_nonzero_ind] != 0)

    return np.array([v_zeros, v_true_zeros, v_true_nonzeros, u_zeros, u_true_
```

In [88]:
```
result = rank1_approx()
```

```
result = np.mean(result, axis = 1)
```

In [89]:

```
print("Average of correctly specified zero entries on u is", result[4] / resu
print("Average of correctly specified nonzero entries on u is", result[5] / 2
print("Average of correctly specified zero entries on v is", result[1] / resu
print("Average of correctly specified nonzero entries on v is", result[2] / 1
```

```
Average of correctly specified zero entries on u is 0.9972918077183481
Average of correctly specified nonzero entries on u is 0.992
Average of correctly specified zero entries on v is 1.0
Average of correctly specified nonzero entries on v is 1.0
```

As we can read from our result, the simulation study on our rank-1 approximation is fairly accurate, with the biclustering identification rate close to 100% on both vector u and v. Comparing with the table in Lee et al. (2010), we can observe that results are close.

Rank-3 simulation

Continuing from the rank-1 approximation setup, we also want to generate a dataset with rank-3. In this simulation, we choose to use a dataset that is moderately sparse, and examine the performance of algorithm on each layer.

In [90]:

```
A = np.random.poisson(1, (15,20)) * np.random.randint(0, 10, (15,20))
pd.DataFrame(A).head()
```

Out[90]:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	6	8	8	0	0	15	5	0	6	9	4	2	0	1	0	9	0	8	0	1
1	0	0	8	2	36	0	0	0	0	0	0	0	6	0	0	0	4	12	2	0
2	0	16	0	0	7	1	7	14	0	6	15	9	9	7	0	18	16	12	0	0
3	7	0	0	0	8	12	0	0	5	9	7	27	1	5	0	8	10	0	2	0
4	2	0	1	1	7	0	12	6	0	0	5	16	0	0	8	6	0	27	9	0

In [91]:

```
SSVD_multi_layer(A, 3)
```

Out[91]:

```
(array([[-0.16507616,  0.        , -0.        ],
        [-0.23220542,  0.53765752,  0.        ],
        [-0.31593966,  0.        , -0.        ],
        [-0.24745302,  0.        ,  0.        ],
        [-0.23318267,  0.        , -0.        ],
        [-0.21918123,  0.        ,  0.98810544],
        [-0.24639009,  0.        ,  0.        ],
        [-0.30752164, -0.        ,  0.        ],
        [-0.35134003,  0.14282498, -0.        ],
        [-0.17735999, -0.        ,  0.        ],
        [-0.17430107,  0.        , -0.        ],
        [-0.3494012 ,  0.52614918,  0.        ],
        [-0.24461678,  0.        , -0.        ],
        [-0.19415487, -0.19970338,  0.        ],
        [-0.30853769, -0.61140087, -0.1537779 ]]),
 array([[-0.12099676,  0.        , -0.        ],
        [-0.23613237,  0.        , -0.        ],
        [-0.16436702, -0.        ,  0.        ],
        [-0.11203678, -0.        ,  0.        ],
        [-0.47872182,  0.83451883, -0.38016974],
        [-0.21064092,  0.        , -0.        ],
        [-0.24280566,  0.        ,  0.6449811 ],
        [-0.15035262,  0.        ,  0.3397161 ],
```

```
[-0.13955544, -0.        , -0.        ],
[-0.25429609, -0.50870314,  0.        ],
[-0.22036297,  0.        ,  0.18815182],
[-0.29374039, -0.21165878,  0.        ],
[-0.17333773, -0.        , -0.        ],
[-0.12431028,  0.        ,  0.        ],
[-0.2213231 ,  0.        ,  0.        ],
[-0.23496495,  0.        ,  0.        ],
[-0.28924804,  0.        ,  0.        ],
[-0.22403407, -0.        ,  0.        ],
[-0.14900805,  0.        ,  0.        ],
[-0.10095454,  0.        ,  0.53727294]]))
```

As we can see from our result, when the target dataset is moderately sparse, the first layer of our SSVD algorithm does not perform biclustering. On the other hand, the second and the third layer have more zero entries and bicluster the dataset.

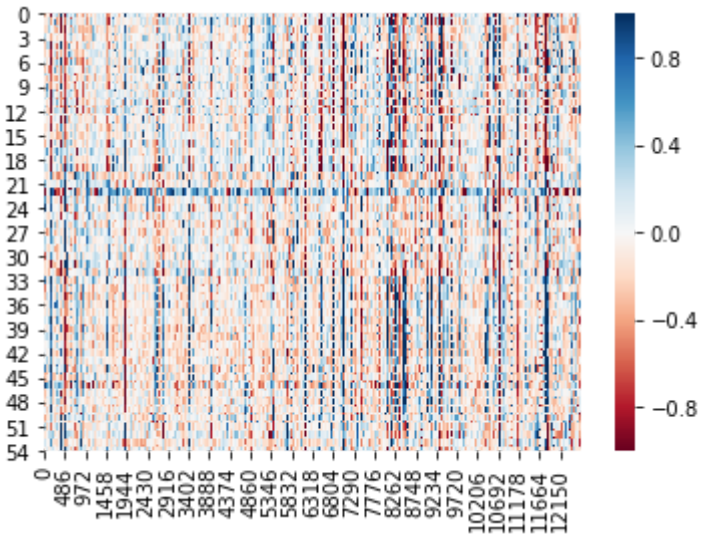
Applications to real-world data set

The lung cancer data

After some simulation studies on our algorithm, we want to apply it to real datasets to check its performance. Firstly, we choose the Lung Cancer dataset used in the paper Lee et al. (2010). The dataset contains 12625 rows and 56 columns, representing gene expressions and subjects, respectively. Notice from the heatmap below that the raw gene data appear to be unorganized. Our expected outcome of applying our SSVD algorithm to this dataset is to find clusters in the gene subjects.

In [92]:
```
Lung_Cancer_data = np.loadtxt('./data/LungCancerData.txt')
X_Lung = Lung_Cancer_data.T

sns.heatmap(X_Lung, vmin=-1, vmax=1, cmap = 'RdBu')
pass
```

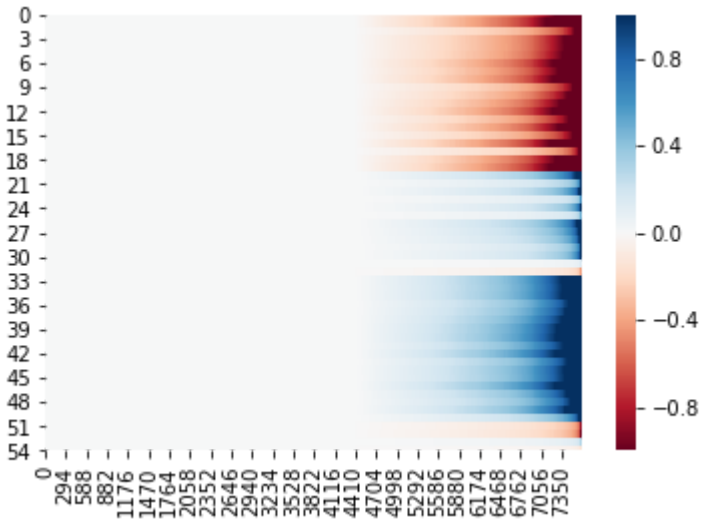


In [93]:
```
result_Lung = SSVD(X_Lung)
u, s, v, i = result_Lung
```

As suggested in Lee et al. (2010), we unselected 5000 genes in the white area. Then we rescaled the data to a $[-1, 1]$ interval for better visualization.

In [94]:
```
Lung_SSVD = s * u.reshape((-1, 1)) @ np.sort(np.abs(v))[5000:].reshape((1, -1
```

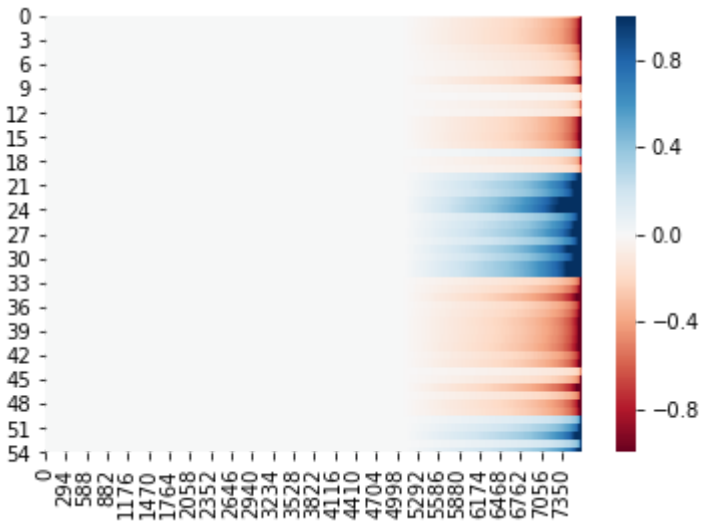
```
sns.heatmap(Lung_SSVD, vmin=-1, vmax=1, cmap = 'RdBu')
pass
```



Note that the original gene data has four subjects. In other words, the genes are supposed to be grouped in 4 clusters. In our plot, we can observe that our gene data are grouped in 4 clusters at subjects #20, #30 and #51. Thus, the rank-1 SSVD by our algorithm is indeed useful in this dataset. We then proceed to the second layer of SSVD.

In [95]:
```
result_Lung2 = SSVD(X_Lung - s * u.reshape((-1, 1)) @ v.reshape((1, -1)))
u2, s2, v2, i2 = result_Lung2
```

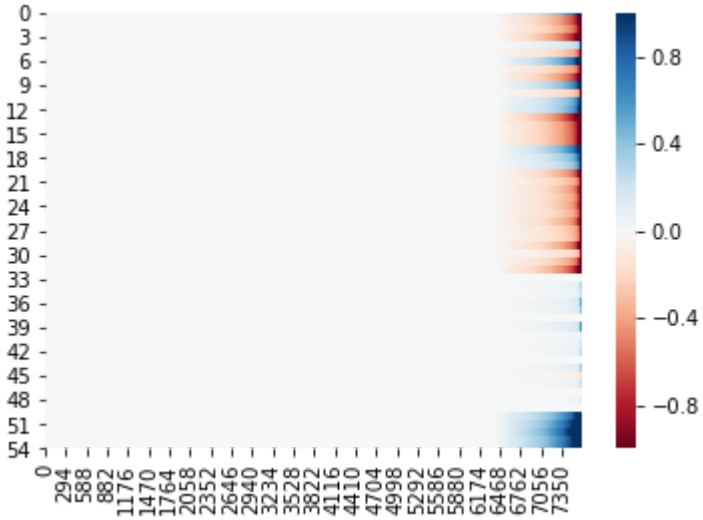
In [96]:
```
Lung_SSVD2 = s2 * u2.reshape((-1, 1)) @ np.sort(np.abs(v2))[5000:].reshape((1
sns.heatmap(Lung_SSVD2, vmin=-1, vmax=1, cmap = 'RdBu')
pass
```



In the heatmap of the second layer, we can notice that the number of genes used to cluster data is less than of from the first layer. Also, we can notice that the difference between the seond and the third cluster, though not very clear in the first layer, is obvious in this layer as they are different in signs.

In [97]:
```
result_Lung3 = SSVD(X_Lung - s * u.reshape((-1, 1)) @ v.reshape((1, -1)) -
                    s2 * u2.reshape((-1, 1)) @ v2.reshape((1, -1)))
u3, s3, v3, i3 = result_Lung3
```

In [98]:
```
Lung_SSVD3 = s3 * u3.reshape((-1, 1)) @ np.sort(np.abs(v3))[5000:].reshape((1
sns.heatmap(Lung_SSVD3, vmin=-1, vmax=1, cmap = 'RdBu')
pass
```



At layer 3, we can notice that the number of genes used to cluster data is significantly less than that of the first two layers. We can also observe that in this layer, only the last group are being clearly clustered by our algorithm, while the first two groups are blur in our graph.

Another dataset from other sources we want to use is used in a proof-of-concept study published in 1999 by Golub et al. Essentially, there are two underlying clusters in the gene data. Similarly, we want our SSVD algorithm to locate these two clusters. Below are a snapshot of our data, and a heatmap of the raw data before clustering.

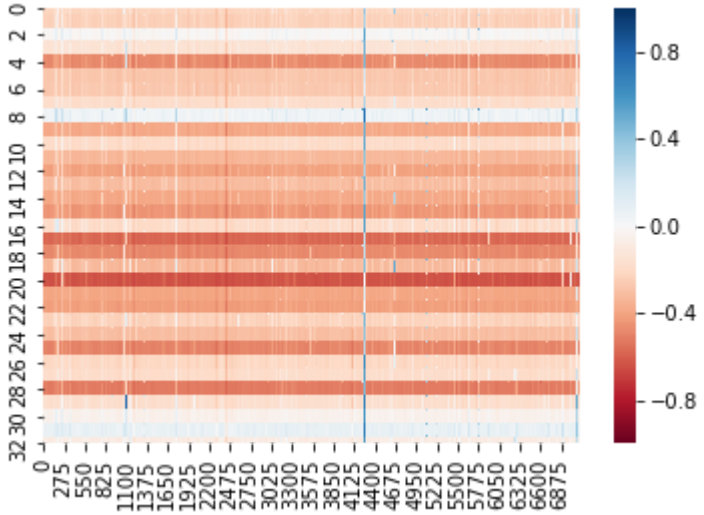
In [99]:
```
gene_expr = pd.read_csv('./data/GolubGeneExpression.csv', sep=',', index_col=
gene_expr.head()
```

Out[99]:

	0	1	2	3	4	5	6	7
0	-0.229573	-0.227176	-0.223444	-0.217709	-0.232755	-0.243086	-0.213349	-0.228080
1	-0.246431	-0.243636	-0.240586	-0.228557	-0.251726	-0.257487	-0.254522	-0.247660
2	-0.018574	-0.017597	-0.026938	-0.004634	-0.029436	-0.039357	-0.014628	-0.029110
3	-0.151755	-0.150989	-0.137168	-0.146394	-0.162111	-0.168165	-0.141070	-0.156058
4	-0.486836	-0.487789	-0.485332	-0.473096	-0.493055	-0.495762	-0.481320	-0.487639

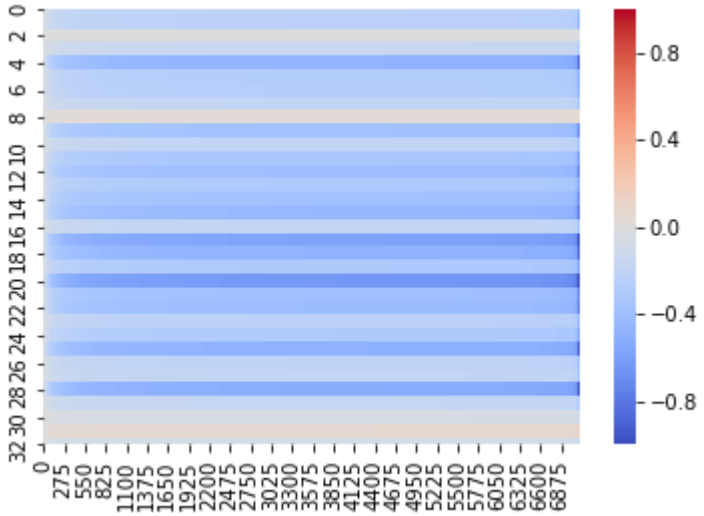
5 rows × 7129 columns

In [100…
```
x_gene = np.array(gene_expr)
sns.heatmap(x_gene, vmin=-1, vmax=1, cmap = 'RdBu')
pass
```



In [101…
```
result_Gene = SSVD(x_gene)
u, s, v, i = result_Gene
```

In [102…
```
Gene_SSVD = s * u.reshape((-1, 1)) @ np.sort(np.abs(v)).reshape((1, -1))
sns.heatmap(Gene_SSVD, vmin=-1, vmax=1, cmap = 'coolwarm')
pass
```



In the originial dataset, the first 27 genes and the last 8 genes belong to two different groups. In our heatmap, we can roughly see a clustering effect around gene #28, where the color underneath that has a warmer color. However, we should also notive that the warm color appears at other genes, indicating that the algorithm fails to correctly indentify those genes.

Comparative analysis with competing algorithms

In the comparative analysis, we want to compare the bicluster performance in terms of accuracy between our SSVD algorithm and some similar models: the standard SVD method and the Sparse Principle Component Analysis (SPCA). The statistics we choose in the analysis are suggested by Lee et al. and are the same in the simulation section: number of zero entries, number of correctly specified zero entries, number of correctly specified nonzero entries and the misclasscification rate.

For consistency, we keep the default weight parameters $\gamma_1 = \gamma_2 = 2$ in the SSVD and the SPCA algorithms.

In [103…

```
u_tilde = np.concatenate((np.arange(3, 11)[::-1],
                      np.ones(17) * 2,
                      np.zeros(75)))
u = u_tilde / np.linalg.norm(u_tilde)
v_tilde = np.concatenate((np.array([10, -10, 8, -8, 5, -5]),
                          np.ones(5) * 3,
                          np.ones(5) * (-3),
                          np.zeros(34)))
v = v_tilde / np.linalg.norm(v_tilde)
s = 50
x_sim = s * u.reshape((-1, 1)) @ v.reshape((1, -1))
```

In [104…

```
def num_zeros(approx):

    return np.sum(approx == 0)

def num_crct_zeros(approx, orig):

    ind = np.where(approx == 0)
    return np.sum(orig[ind] == 0)

def num_crct_nzeros(approx, orig):

    ind = np.where(approx != 0)
    return np.sum(orig[ind] != 0)
```

In [105…

```
def all_approx(M = 100):


    u_tilde = np.concatenate((np.arange(3, 11)[::-1],
                      np.ones(17) * 2,
                      np.zeros(75)))
    u = u_tilde / np.linalg.norm(u_tilde)
    v_tilde = np.concatenate((np.array([10, -10, 8, -8, 5, -5]),
                              np.ones(5) * 3,
                              np.ones(5) * (-3),
                              np.zeros(34)))
    v = v_tilde / np.linalg.norm(v_tilde)
    s = 50
    x_sim = s * u.reshape((-1, 1)) @ v.reshape((1, -1))

    zeros = np.zeros((M, 6))
    crct_zeros = np.zeros((M, 6))
    crct_nzeros = np.zeros((M, 6))
    n_all = np.concatenate((np.ones(3) * 100, np.ones(3) * 50))

    for i in range(M):
        noise = np.random.normal(size = x_sim.shape)
        ssvd_u, ssvd_s, ssvd_v, ssvd_i = SSVD(x_sim + noise)
        svd_u, svd_s, svd_v = la.svd(x_sim + noise)
        svd_u = svd_u[:, 0]
        svd_v = svd_v[0, :]
        spca_model = SparsePCA(n_components=1)
        spca_model.fit(x_sim + noise)
        spca_v = spca_model.components_[0]
        spca_model.fit((x_sim + noise).T)
        spca_u = spca_model.components_[0]
```

```
        list_of_approx_u = [ssvd_u, svd_u, spca_u]
        list_of_approx_v = [ssvd_v, svd_v, spca_v]

        zeros[i, :] = np.array(list(map(num_zeros, list_of_approx_u + list_of_
        crct_zeros[i, :] = np.array(list(map(num_crct_zeros, list_of_approx_u
                                         list(map(num_crct_zeros, list_of_appro
        crct_nzeros[i, :] = np.array(list(map(num_crct_nzeros, list_of_approx_
                                         list(map(num_crct_nzeros, list_of_appr


    return np.array([np.mean(zeros, axis = 0),
                     np.mean(crct_zeros, axis = 0),
                     np.mean(crct_zeros, axis = 0) / np.mean(zeros, axis = 0),
                     np.mean(crct_nzeros, axis = 0),
                     np.mean(crct_nzeros, axis = 0) / (n_all - np.mean(zeros, a
                     ((np.mean(zeros, axis = 0) - np.mean(crct_zeros, axis = 0)
                      (n_all - np.mean(zeros, axis = 0) - np.mean(crct_nzeros,
                      ])
```

In [106…

```
result = all_approx()
result_pd = pd.DataFrame(result.T)
result_pd.columns = ['Avg # of zeros',
                     'Avg # of correctly identified zeros',
                     'Percentage of correctly identified zeros',
                     'Avg # of correctly identified nonzeros',
                     'Percentage of correctly identified nonzeros',
                     'Misclassification Rate'
                    ]
result_pd= result_pd.reindex([0, 3, 1, 4, 2, 5])
result_pd.index = ['SSVD_u', 'SSVD_v', 'SVD_u', 'SVD_v', 'SPCA_u', 'SPCA_v']
result_pd
```

Out[106…

	Avg # of zeros	Avg # of correctly identified zeros	Percentage of correctly identified zeros	Avg # of correctly identified nonzeros	Percentage of correctly identified nonzeros	Misclassification Rate
SSVD_u	74.00	73.70	0.995946	24.70	0.950000	0.0160
SSVD_v	32.32	32.32	1.000000	16.00	0.904977	0.0336
SVD_u	0.00	0.00	NaN	25.00	0.250000	0.7500
SVD_v	0.00	0.00	NaN	16.00	0.320000	0.6800
SPCA_u	51.37	51.36	0.999805	24.99	0.513880	0.2365
SPCA_v	23.04	23.04	1.000000	16.00	0.593472	0.2192

In the comparative run on our simulated dataset, the biclustering by the our SSVD method has the overall minimum value of the misclasscification rate. More specifically, it performs consistently in both vectors u and v. Notice that the rate of nonzero entries identification is slighly lower than that of zero entries identification. Thus, a valid convern about this algorithm is that whether it generally shrink entries to zero more than it should.

The standard SVD algorithm does not perform well in our standard. By a closer look on our numbers, we found that the first layer of a SVD decomposition are all nonzeros. Hence, the standard SVD algorithm cannot bicluster data by shrinking some entries to zero.

The paper Lee et al. claims that the SPCA method is a the most proper tool in biclustering. And our result confirms with that claim. In the table, we can notice that though it has a high

idenfitication rate on zeros, its idenfitication rate on nonzeros is only about 50%, making mistake half of the times. Hence, it is not comparable to the SSVD algorithm in terms of biclustering.

Discussion/conclusion

In this paper, we introduced an algorithm performing SSVD decomposition. To illustrate the usage of the algorithm, we experimented on a simulated dataset in which we added sparsity on the left and right vectors, u and v. By decomposing the dataset, the algorithm selects important entries on both columns and rows. Hence, we can expect this algorithm to be a useful tool in biclustering. According to our simulation and our applications on real datasets, our SSVD algorithm is successful in biclustering our datasets. The evidence is provided by the table summarizing the identification rate and the misclassification rate, which the SSVD algorithm has high scores. Also, by visualizing our result with heatmaps we can observe that the biclustering effect generally agrees with the underlying structure of our original dataset. Thus, in our study, the SSVD algorithm is indeed useful in biclustering datasets.

Given, the algorithm's consistent performance, we can expect it to be applied in biomedical researches. According to the paper by Xie et al. (2018), the biclustering technique is useful in analyzing various omics data to generate system-level understanding. The biclustering technique can also be applied to gene datasets to categorize them into one or multiple functions. We believe that the SSVD algorithm has the potency to be generated for these domains.

Admittedly, there are some limitations to our algorithm. Based on our study, a disadvantage of it is the long computational time. While running our model, the selection of penalizing parameter often take time to complete, especially when the target dataset has high dimension. A potential solution to this problem is to limit the candidate of penalizing parameters or to modify the selection threshold BIC. Another aspect that we want to improve is its performance while applied to higher-rank datasets. We noticed that the algorithm takes multiple layers to bicluster the dataset. And our future direction is to identify the connection between the number of layers in the algorithm and the structure of the target dataset.

Package Installation Instruction

The package can be installed using `pip install git+https://github.com/RumoZhang/STA663-SSVD.git#egg=STA663-SSVD`

The package contains two functions, which can be loaded using:

- `from SSVD.functions import SSVD_single`
- `from SSVD.functions import SSVD_multi_layer`

Contribution

Rumo Zhang: initial implementation of the algorithm, implementation to simulated and real dataset, comparitive analysis, discussion

Xige Huang: abstract, background, algorithm, maintainance of github page, package installation

Reference

Lee, Mihee, et al. "Biclustering via sparse singular value decomposition." Biometrics 66.4 (2010): 1087-1095.

Juan Xie, Anjun Ma, Anne Fennell, Qin Ma, Jing Zhao, It is time to apply biclustering: a comprehensive review of biclustering applications in biological and biomedical data, Briefings in Bioinformatics, Volume 20, Issue 4, July 2019, Pages 1450–1465

Golub, Todd R., et al. "Molecular classification of cancer: class discovery and class prediction by gene expression monitoring." science 286.5439 (1999): 531-537.

Zou, H. (2006). The adaptive lasso and its oracle properties. Journal of the American Statistical Association 101, 1418-1429.

Data:

Kaggle: https://www.kaggle.com/crawford/gene-expression