STA 663 Final Project
Implementation of Biclustering via Sparse Singular Value Decomposition

Yijun Jiang, Xingyu Yan

May 1, 2017

Abstract

Sparse singular value decomposition (SSVD) is considered to be an emerging exploratory analysis tool for biclustering. It generates low-rank and checkerboard-structured matrix approximations to original data by forcing both the left and right singular vectors to have many zero entries. Sparsity-inducing regularization penalties are imposed to the least squares regression to generate sparse singular vectors. In this paper, we implement SSVD algorithm adopted from paper *Biclustering via Sparse Singular Value Decomposition*. We explain the mathmatical basis and establish the algorithm step-by-step. Then we encode the algorithm in Python and optimize it using profiling, Just-In-Time Compilation and Cython. Furthermore, we use the lung cancer data and two simulated datasets (rank-1 and rank-2 simulation) as applications of SSVD. We provide data visualization such as image plots on lung cancer data and also utilize two simulated datasets to compare SSVD with other existing biclustering methods.
KEY WORDS: Adaptive Lasso; Biclustering: Singular Value Decomposition; Dimension Reduction

1 Introduction

When we deal with datasets in real life, high dimensionality has become a very common feature in various applications. Sometimes, we may encounter data that are high-dimension low sample size (HDLSS), for instance, the gene expression analysis in the paper we chose. In this case, we prefer using unsupervised learning to find interpretable structures in the data.

The paper we selected is "Biclustering via Sparse Singular Value Decomposition", which describes biclustering methods, a collection of unsupervised learning tools that identify distinctive "checkerboard" patterns in data matrices. This method has become very important recently that many previous studies were conducted, such as comprehensive survey of existing biclustering algorithms for biological data analysis by Madeira and Oliveira (2004) and Biclustering in data mining by Busygin, Prokopyev and Pardalos (2008). In the paper we chose, the author introduces sparse singular value decomposition (SSVD), a new tool for biclustering.

In the paper, the author uses two datasets, microarray gene expressions and nutrition datasets as applications. However, we have no access to the second dataset; Instead, we use microarray gene expression and two simulated datasets to tesify the algorithm and compare it with other methods.

Let $\mathbf{X}$ be a n by d data matrix whose rows represent samples and columns represent variables. The singular value decomposition of $\mathbf{X}$ can be written as

$$\mathbf{X} = \sum_{k=1}^{r} s_k \mathbf{u}_k \mathbf{v_k}^T$$

SVD decomposes $\mathbf{X}$ into a summation of rank-one matrices $s_k \mathbf{u}_k \mathbf{v}_k^T$, each of which is defined as an SVD layer. We always pay attention to the SVD layers corresponding to significant s_k values and

interprete SVD with small s_k as noise. If we use first $K \leq r$ rank-one matrices, we can estimate $\mathbf{X}$ as:

$$\mathbf{X} \approx \sum_{k=1}^{K} s_k \mathbf{u}_k \mathbf{v}_k^T$$

And we want to find the approximation that minimizes the squared Frobenius norm.

$$\arg \min_{\mathbf{X}^* \in \mathcal{A}_x} tr(\mathbf{X} - \mathbf{X}^*)(\mathbf{X} - \mathbf{X}^*)^T$$

The proposed SSVD finds out a low-rank matrix approximation to $\mathbf{X}$ with the constraint that $\mathbf{u}_k$ and $\mathbf{v_k}$ are sparse. We add sparsity-inducing penalties and now the rank-one matrix $s_k \mathbf{u}_k \mathbf{v}_k$ is defined as an SSVD layer, which has a checkerboard structure. This makes SSVD suitable for biclustering.

2 Methodology

In this section, we only focus on the first SSVD layer $s\mathbf{u}_1\mathbf{v}_1^T$. The subsequent layers can be extracted from the residual matrix after removing the preceding layers. The first SSVD layer $s_1\mathbf{u}_1\mathbf{v}_1^T$ is the best rank-one matrix approximation of $\mathbf{X}$ under the Frobenius norm with sparsity-inducing penalties,

$$argmin_{s,\mathbf{u},\mathbf{v}} \|\mathbf{X} - s\mathbf{u}\mathbf{v}\|_F^2 + \lambda_u P_1(s\mathbf{u}) + \lambda_v P_2(s\mathbf{v}) \tag{1}$$

where $P_1(s\mathbf{u})$ and $P_2(s\mathbf{v})$ are sparsity-inducing penalties and λ_u and λ_v are penalty parameters providing trade-off between the loss function $\|\mathbf{X} - s\mathbf{u}\mathbf{v}\|_F^2$ and the penalties. When $\lambda_u = \lambda_v = 0$, no penalties are imposed and we obtain plain SVD layers. If λ_u and λ_v go to infinity, $\mathbf{u}$ and $\mathbf{v}$ will go to zero.

2.1 Penalized Sum of Squares

We use adaptive lasso penalties where data-driven weights are added to the penalty terms. For fixed $\mathbf{u}$, minimization of 1 with respective to $(s, \mathbf{v})$ is equivalent to minimization with respect to $\tilde{\mathbf{v}} = s\mathbf{v}$ of

$$\|\mathbf{X} - s\mathbf{u}\mathbf{v}\|_F^2 + \lambda_v P_2(\tilde{\mathbf{v}}) = \|\mathbf{Y} - (\mathbf{I}_d \otimes \mathbf{u})\tilde{\mathbf{v}}\|^2 + \lambda_v P_2(\tilde{\mathbf{v}}) \tag{2}$$

where $\mathbf{Y} = (\mathbf{x}_x^T, ..., \mathbf{x}_d^T)^T$ with $\mathbf{x}_j$ being the jth column of $\mathbf{X}$ and $\otimes$ being the Kronecker product. The right-hand side is the minimization of a penalized linear regression with $(\mathbf{I}_d \otimes \mathbf{u})$ as the design matrix, $\tilde{\mathbf{v}}$ as the coefficient vector, and $P_2(\tilde{\mathbf{v}})$ as the penalty term.

Similarly, for fixed $\mathbf{v}$, minimization of 1 with respective to $(s, \mathbf{u})$ is equivalent to minimization with respective to $\tilde{\mathbf{u}} = s\mathbf{u}$ of

$$\|\mathbf{X} - s\mathbf{u}\mathbf{v}\|_F^2 + \lambda_u P_1(\tilde{\mathbf{u}}) = \|\mathbf{Z} - (\mathbf{I}_n \otimes \mathbf{v})\tilde{\mathbf{u}}\|^2 + \lambda_u P_1(\tilde{\mathbf{u}}) \tag{3}$$

where $\mathbf{Z} = (\mathbf{x}_{(1)}, ..., \mathbf{x}_{(n)})^T$ with $\mathbf{x}_{(i)}^T$ being the ith row of $\mathbf{X}$. The right-hand side is also the minimization of sum of squares with $P_1(\tilde{\mathbf{u}})$ as the penalty.

The adaptive lasso penalties take the form of $P_1(\tilde{\mathbf{u}}) = s\sum_{i=1}^{n} w_{1,i}|u_i|$, and $P_2(\tilde{\mathbf{v}}) = s\sum_{j=1}^{d} w_{2,j}|v_j|$, where $w_{1,i}$ and $w_{2,j}$ are weights depending on the data. When $w_{1,i} = w_{2,j} = 1$, we obtain the lasso penalties. The weights $w_{2,j}$ can be defined by

$$\mathbf{w_2} = (w_{2,1}, ..., w_{2,d})^T = |\hat{\tilde{\mathbf{v}}}|^{-\gamma_2}$$

where $\hat{\tilde{\mathbf{v}}}$ is the OLS estimate of $\tilde{\mathbf{v}}$, which in this setting is

$$\{(\mathbf{I}_d \otimes \mathbf{u})^T(\mathbf{I}_d \otimes \mathbf{u})\}^{-1}(\mathbf{I}_d \otimes \mathbf{u})^T\mathbf{Y} = \mathbf{X}^T\mathbf{u}$$

.

The weights $w_{1,i}$ can be defined by

$$\mathbf{w_1} = (w_{1,1}, ..., w_{1,n})^T = |\hat{\tilde{\mathbf{u}}}|^{-\gamma_1}$$

where $\hat{\tilde{\mathbf{u}}}$ is the OLS estimate of $\tilde{\mathbf{u}}$, which is

$$\{(\mathbf{I}_n \otimes \mathbf{v})^T(\mathbf{I}_n \otimes \mathbf{v})\}^{-1}(\mathbf{I}_n \otimes \mathbf{v})^T\mathbf{Z} = \mathbf{X}\mathbf{v}$$

.

There are some choices for γ_1 and γ_2. If $\gamma_1 = \gamma_2 = 0$, then we go back to lasso penalties. The authors of this paper proposed to use $\gamma_1 = \gamma_2 = 0.5$ or 2. We choose to use $\gamma_1 = \gamma_2 = 2$ in the implementation.

2.2 Selection of Penalty Parameters

We select λ_v and λ_u based on BIC. For fixed $\mathbf{u}$, we define

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

where $\hat{df}(\lambda_v)$ is the degree of sparsity (the number of zeros) of $\mathbf{v}$ with λ_v as the penalty parameter and $\hat{\sigma}^2$ is the OLS estimated mean squared error from 2.

For fixed $\mathbf{v}$, we define

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

where $\hat{df}(\lambda_u)$ is the degree of freedom of $\mathbf{u}$ with λ_u as the penalty parameter and $\hat{\sigma}^2$ is the OLS estimated mean squared error from 3.

2.3 The Iterative Algorithm

With the adaptive lasso penalties, the minimization of 1 can be written as

$$\|\mathbf{X} - s\mathbf{u}\mathbf{v}^T\|_F^2 + \lambda_u \sum_{i=1}^{n} w_{1,i}|u_i| + s\lambda_v \sum_{j=1}^{d} w_{2,j}|v_j| \tag{4}$$

We can alternatively minimize 4 with respective to $\mathbf{u}$ and $\mathbf{v}$. For fixed $\mathbf{u}$, minimization of 4 is equivalent to

$$\|\mathbf{X} - \mathbf{u}\tilde{\mathbf{v}}^T\|_F^2 + s\lambda_v \sum_{j=1}^{d} w_{2,j}|\tilde{v_j}| = \|\mathbf{X}\|_F^2 + \sum_{j=1}^{d}\{\tilde{v_j}^2 - 2\tilde{v_j}(\mathbf{X}^T\mathbf{u})_j + \lambda_v w_{2,j}|\tilde{v_j}|\} \tag{5}$$

5 has a closed-form solution

$$\tilde{v_j} = sign\{(\mathbf{X}^T\mathbf{u})_j\}(|(\mathbf{X}^T\mathbf{u})_j| - \lambda_v w_{2,j}/2)$$

.

For fixed $\mathbf{v}$, minimization of 4 is equivalent to

$$\|\mathbf{X} - \tilde{\mathbf{u}}\mathbf{v}^T\|_F^2 + \lambda_u \sum_{i=1}^{n} w_{1,i}|\tilde{u_i}| = \|\mathbf{X}\|_F^2 + \sum_{i=1}^{n}\{\tilde{u_i}^2 - 2\tilde{u_i}(\mathbf{X}\mathbf{v})_i + \lambda_u w_{1,i}|\tilde{u_i}|\} \tag{6}$$

6 also has a closed-form solution

$$\tilde{u_i} = sign\{(\mathbf{X}\mathbf{v})_i\}(|(\mathbf{X}\mathbf{v})_i| - \lambda_u w_{1,i}/2)$$

.

Thus the minimization of 4 can be applied with respect to $\mathbf{v}$ and $\mathbf{u}$ iteratively until convergence. The selection of penalty parameters can be nested within the iterative algorithm. Finally, the iterative SSVD procedure with model fitting and parameter selection are shown below.

- Apply the standard SVD to X. Let $\{s_{old}, \mathbf{u}_{old}, \mathbf{v}_{old}\}$ denote the first SVD triplet.
- Set $\tilde{v_j} = sign\{(\mathbf{X}^T\mathbf{u_{old}})_j\}(|\ (\mathbf{X}^T\mathbf{u_{old}})_j\ |) - \lambda_v\omega_{2,j}/2)_+$, $j = 1, 2, ..., d$. λ_v is the minimizer of $BIC(\lambda_v) = \frac{\|\mathbf{Y}-\hat{\mathbf{Y}}\|^2}{nd\hat{\sigma}^2} + \frac{log(nd)}{nd}\hat{df}(\lambda_v)$. Let $\tilde{\mathbf{v}} = (\tilde{v}_1, ..., \tilde{v}_d)^T$, $s = \|\tilde{\mathbf{v}}\|$, and $\mathbf{v}_{new} = \tilde{\mathbf{v}}/s$.
- Set $\tilde{u_j} = sign\{(\mathbf{X}^T\mathbf{v_{new}})_i\}(|\ (\mathbf{X}^T\mathbf{v_{new}})_i\ |) - \lambda_u\omega_{1,i}/2)_+$, $i = 1, 2, ..., n$. λ_u is the minimizer of $BIC(\lambda_u) = \frac{\|\mathbf{Z}-\hat{\mathbf{Z}}\|^2}{nd\hat{\sigma}^2} + \frac{log(nd)}{nd}\hat{df}(\lambda_u)$. Let $\tilde{\mathbf{u}} = (\tilde{u}_1, ..., \tilde{u}_n)^T$, $s = \|\tilde{\mathbf{u}}\|$, and $\mathbf{u}_{new} = \tilde{\mathbf{u}}/s$.
- Set $\mathbf{u} = \mathbf{u}_{new}$, $\mathbf{v} = \mathbf{v}_{new}$, and $s = \mathbf{u}_{new}^T\mathbf{X}\mathbf{v}_{new}$ at convergence.

3 Profiling and Optimization

3.1 Profiling

In order to check bottlenecks, we profile the code using `%prun` decorator on the lung cancer data (56×12625) to be discussed in the Application section. The most time-consuming functions are shown in Table 1. It is not surprising to see that our main function, `ssvd`, takes more than 80% of the time because nearly all the computations are done in this function. The numpy version SVD function, which are used to set initial values, also takes a relative long time. The `thresh` function, which solves the penalized minimization problems, are called a large number of times, but each call takes small amount of time. Numpy vectorized functions, such as `numpy.sum` or `numpy.zeros`, are relatively fast, although being called for many times. In order to improve the runtime, we need to work on optimizing our main `ssvd` function and the `thresh` function.

function	Number of Calls	Total Time (s)	Time per Call (s)
ssvd	1	252.90	252.90
numpy.linalg.svd	1	14.959	14.959
thresh	76098	2.583	0.000
numpy.sum	152209	0.780	0.000
numpy.zeros	76086	0.780	0.000
math.log	76086	0.138	0.000

Table 1: Profiling of the initial code.

3.2 Just-In-Time Compilation

We use the Just-In-Time(JIT) compiler from the package `numba` by simply adding the `@JIT` decorator. It generates optimized machine code on the fly, in contrast to other compilers, such as Cython, that compile code to machine language ahead of time. The runtime comparison is shown in Table 2. The runtime does not improve a lot, which may convey that most part of the code fails to compile to machine code.

3.3 Cythonizing

We use Cython which combines Python with C to generate optimized code. We start with our pure Python function, run it through Cython with the annotate flag, and rewrite the code that are highlighted in yellow which indicates Python interaction. The runtime comparison is also shown in Table 2. The cythonized code is around 25% faster than the pure Python code. This result is not as good as we expect, probability because we could not remove all the Python interaction in yellow. Given more time, we may consider

	Run Time (ms)
pure python	128
Just-In-Time	102
Cythonized Code	93.9

Table 2: Runtime comparison (on a 56×100 matrix).

4 Application

4.1 Simulation

In this section, we use two simulation studies to investigate the performance of SSVD and then compare them with other related methods, which will be discussed in later sections. For the first simulation test, we use rank-1 signal matrix $\mathbf{X}^*$ to investigate the performance of SSVD. Here, we define

$$\mathbf{X}^* = s\mathbf{u}\mathbf{v}^T$$

where $s = 50$, and

$\hat{\mathbf{u}} = [10, 9, 8, 7, 6, 5, 4, 3, r(2, 17), r(0, 75)]^T$
$\hat{\mathbf{v}} = [10, -10, 8, -8, 5, -5, r(3, 5), r(-3, 5), r(0, 34)]^T$
$\mathbf{u} = \frac{\hat{\mathbf{u}}}{\|\hat{\mathbf{u}}\|}$,$\mathbf{v} = \frac{\hat{\mathbf{v}}}{\|\hat{\mathbf{v}}\|}$

Here $r(a, b)$ denotes a vector of length b, whose entries are all equal to a. $\mathbf{X}^*$ here is a 100 by 50 matrix. Then we calculate $\mathbf{u}$ and $\mathbf{v}$, which have 25 and 16 nonzero entries respectively. Then we generate $\mathbf{X}$ as the sum of $\mathbf{X}^*$ and the noise matrix ϵ which are randomly sampled from a standard normal distribution. Then we repeat this procedure 100 times and apply SSVD function in each iteration. For SSVD, we utilize BIC to choose the degree of sparsity and use weight parameters $\gamma_1 = \gamma_2 = 2$ to define weight vectors. After running the algorithm for 100 iterations, we collect the average number of zeros, the average proportion of correctly classified zeros and nonzeros and the misclassification error rate in two directions: $\mathbf{u}$ and $\mathbf{v}$ respectively. We found out that SSVD correctly classifies most elements.

For the second simulation test, we change to do higher rank approximation. In this case , we use a rank-2 true signal matrix $\hat{\mathbf{X}}^*$ (defined below) to investigate the performance of SSVD. Here, we define

$$\hat{\mathbf{X}}^* = s_1 \mathbf{u}_1 \mathbf{v}_1^T + s_2 \mathbf{u}_2 \mathbf{v}_2^T$$

where $s_1 = 1000$ and $s_2 = 100$,and

$\hat{\mathbf{u}}_1 = [r(20, 2), r(10, 4), r(3, 8), r(1, 16), r(0, 70)]^T$
$\hat{\mathbf{v}}_1 = [r(1, 20), r(0, 30)]^T$
$\hat{\mathbf{u}}_2 = [r(0, 6), 5, -5, r(0, 6), r(10, 4), r(-10, 4), r(0, 8), r(30, 6), r(0, 14)]^T$
$\hat{\mathbf{v}}_2 = [r(0, 10), r(1, 5), r(-1, 5), r(0, 30)]^T$
$\mathbf{u}_1 = \frac{\hat{\mathbf{u}}_1}{\|\hat{\mathbf{u}}_1\|}, \mathbf{v}_1 = \frac{\hat{\mathbf{v}}_1}{\|\hat{\mathbf{v}}_1\|}, \mathbf{u}_2 = \frac{\hat{\mathbf{u}}_2}{\|\hat{\mathbf{u}}_2\|}, \mathbf{v}_2 = \frac{\hat{\mathbf{v}}_2}{\|\hat{\mathbf{v}}_2\|}$

We can find that $\mathbf{u}_1$ and $\mathbf{u}_2$ are orthogonal, as well as $\mathbf{v}_1$ and $\mathbf{v}_2$. Here $\mathbf{X}^*$ is a 100 by 50 matrix of rank 2. Also, we define ϵ the same as above and then we generate $\mathbf{X}$ as the sum of $\mathbf{X}^*$ and ϵ. This simulation is again repeated 100 times.

SSVD produces its first sparse singular vector $\hat{\mathbf{u}}$ and $\hat{\mathbf{v}}$ from the original data $\mathbf{X}$. Then we calculate the residual matrix $\mathbf{X} - \hat{s}_1 \hat{\mathbf{u}}_1 \hat{\mathbf{v}}_1^T$ and apply it to SSVD again to obtain the the second sparse singular vector $\hat{\mathbf{u}}_2$ and $\hat{\mathbf{v}}_2$. We can then estimate $\mathbf{X}^* = \hat{s}_1 \hat{\mathbf{u}}_1 \hat{\mathbf{v}}_1^T + \hat{s}_2 \hat{\mathbf{u}}_2 \hat{\mathbf{v}}_2^T$.

After running the algorithm for 100 iterations, we collect the average number of zeros in each direction : u_1, v_1, u_2, v_2 respectively and also the average proportion of correctly classified zeros and nonzeros as well as misclassification error rate. We found out that SSVD correctly classified the majority of elements in each direction.

4.2 Lung Cancer Data

We tested SSVD algorithm using microarray gene expression data. The dataset is high-dimensional but with small sample size, consisting of expression levels of 12625 genes from 56 subjects (56 $\times$ 12625 matrix). In the dataset, each column represents expression levels of a particular gene and each row represents a subject. These subjects are either without cancer (Normal) or with one of the three types of cancer, including carcinoid tumors (Carcinoid), colon metastases (Colon), and Small cell carcinoma (SmallCell). The subjects are grouped together according to their cancer types.

Our goal is to identify rows and columns in the matrix that are highly related, or "checkerboard" patterns in the data. In other words, we try to find groups of genes that are significantly expressed for certain types of cancer. As SSVD is an unsupervised learning algorithm, we use the information of cancer types only for interpretation and evaluation purpose.

The first three SSVD layers $\hat{s_k} \hat{\mathbf{u}}_k \hat{\mathbf{v}}_k^T, k = 1, 2, 3$ are obtained using the SSVD algorithm mentioned above. We only focus on the first three layers because the first three singular values are much bigger than the others. The results are shown in Figure 1, where each panel corresponds to different layer. All entries of each layer are first scaled by the maximum absolute value of the entries, so that they fall into the range $[-1, 1]$. Genes are then reordered according to values of

right sparse singular vectors $\hat{\mathbf{v}}_k$, while subject are reordered according to values of the left sparse singular vectors $\hat{\mathbf{u}}_k$ within each cancer type. The horizontal dotted lines differentiates groups of caner types. The white area within the two vertical dotted lines represents zeroed-out genes, where entires of layers are zero. In each panel, 8000 genes with values of zero are omitted for better visualization.

Figure 1 clearly illustrates the "checkerboard" patterns from biclustering. For exmaple, in the first layer the first 1463 genes are positively expressed for Carcinoid and SmallCell, but negatively expressed for Normal and Colon. On the contrary, genes from 10883 to the last are negatively expressed for Carcinoid and SmallCell, but positively expressed for Normal and Colon. Genes between 1463 to 10883 are zeroed-out, indicating the sparsity imposed by SSVD. Similarly, the second layer also shows the grouping of genes and cancer types, highlighting the contrast between Carcinoid/Normal and Colon/SmallCell. The same patterns also appear in the third layer.

The grouping can also be seen in Figure 2, where the left sparse singular vectors $\hat{\mathbf{u}}_k (k = 1, 2, 3)$ are plotted against each other. The first panel shows a clear separation of the four cancer types, even though cancer types information are not used for training. The second and the third panel reveal not only clusters of different cancer types, but also two sub-clusters for Carcinoid.

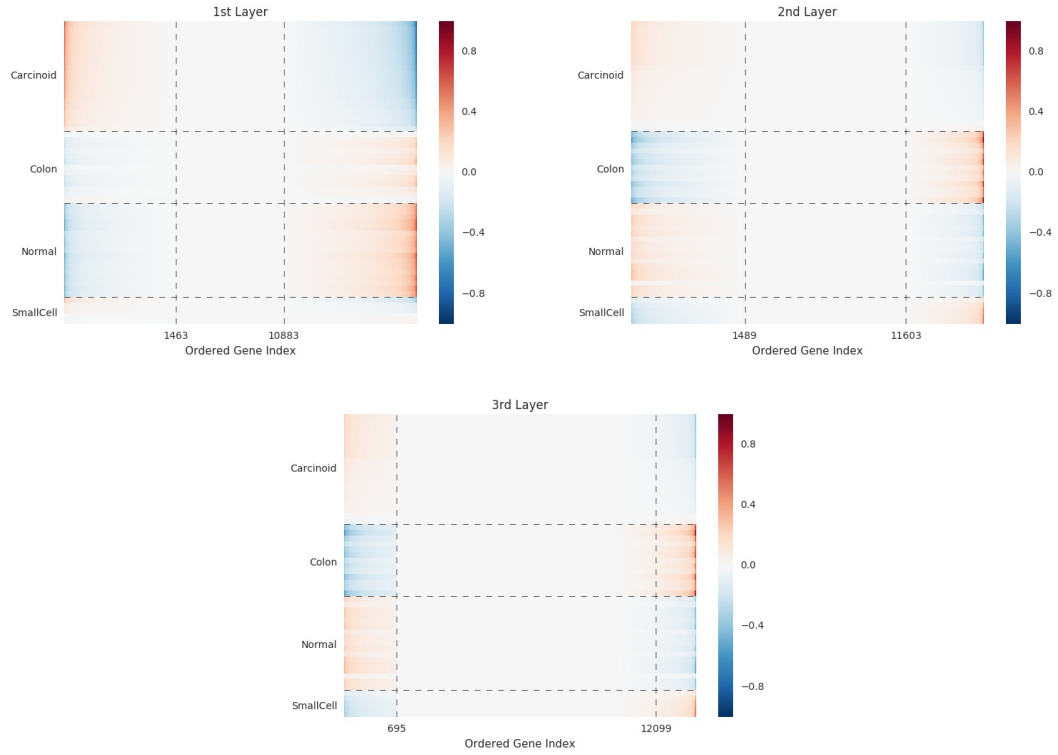

Figure 1: Image plots of the first three SSVD layers.

5 Comparative Analysis

Besides SSVD, we applied other methods to the simulated data to compare their performance as well. For the rank-1 approximation, we first consider using Plaid and RoBic biclustering method. These two methods are based on SVD and assume that the data matrix can be estimated by a similar structure provided by SVD. For Plaid, we assume that

$$\mathbf{X}_{ij} = \sum_{K=1}^{K} \theta_{ijk}\rho_{ik}\kappa_{jk} = \sum_{k=1}^{K}(\mu_k + \alpha_{ik} + \beta_{jk})\rho_{ik}\kappa_{jk}$$

Where K is the number of layers (biclusters), ρ_{ik} is an indicator that it is 1 if row i is in the kth bicluster, and κ_{jk} is 1 if column j is in the kth bicluster.θ_{ijk} denotes the contribution of the kth

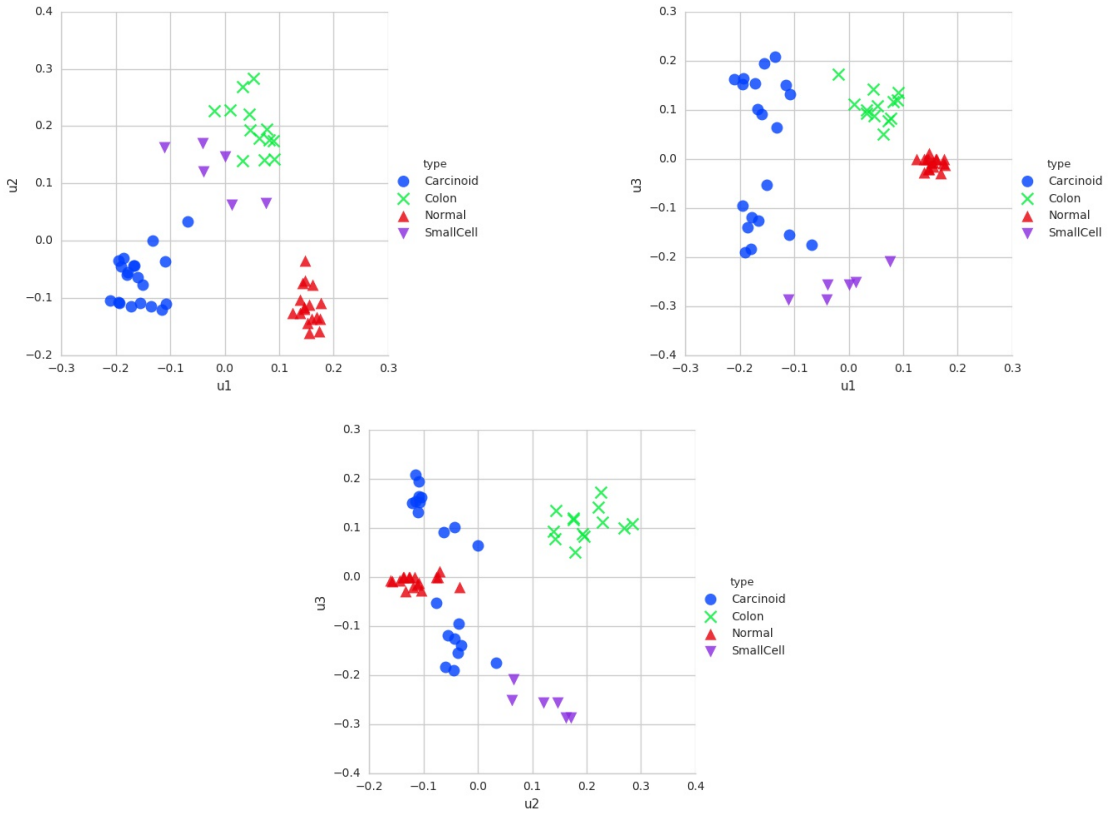

Figure 2: Scatterplots of the entries of the first three left sparse singular vectors $\hat{\mathbf{u}}_k (k = 1, 2, 3)$

bicluster. Plaid utilizes an iterative method to get layers and all parameters can be approximated by maximizing the reduction in sum of squares. On the other hand, in SSVD, we assume there is a multiplicative structure wihin the Kth bicluster, which balances the goodness-of-fit and the sparsity.

For RoBiC,we begin with the best rank-1 approximation of the data matrix obtained by SVD. Then we sort entries of the first left singular vector in a decreasing order in terms of absolute values, which are latter fit by a hinge function. The same procedure is applied to the first right singular vector as well to generate the first column cluster. RoBiC obtains the first bicluster from the data and then calculate the residual matrix by substracing biclusters from the data matrix and use it to obtain the next bicluster. However, The hinge function here imposes an unnecessarily strict constraint which may limit the sparse structure to be detected. However, since there are no packages available for RoBic and Plaid right now, it is very hard for ourselves to implement algorithms. Instead, we compared SSVD with the basic SVD.

Based on above tables, we conclude that SSVD performs much better than SVD. It is reasonable since SVD failed to identify any sparse matrix. Also, even if we could not test RoBiC in this paper, we can predict it would perform better than SVD, but worse than SSVD, since it will detect too many zeros due to hinge model. RoBic has the trouble differentiating the small nonzero entries from zeros.

For rank-2 approximation, we again choose SSVD and SVD. Since the existing packages for Plaid and RoBiC either lack an automatic procedure to generate outputs or only return locations of detected biclusters without estimating the model. AS a result, We only run SSVD and SVD on simulated data and analyze Plaid and RoBic from theoretical perspective.

Based on Table 3, Table 4, we found out that SVD again fails and SSVD outperforms in both layers: it identifies almost all zero/nonzero elements. On the other hand, RoBiC is expected to

perform worse than SSVD; RoBiC cannot detect very small nonzero elements of $\mathbf{u}_1$, since the hinge function imposes very strict constraints.

Method	Direction	Average number of zeros	Correctly classified zeros	Correctly classified nonzeros	Misclassification error rate
SSVD	u	74.31	98.8	99.16	1.11
SSVD	v	33.76	99.29	100.0	0.48
SVD	u	0.0	0.0	100.0	75.0
SVD	v	0.0	0.0	100.0	83.74

Table 3: Rank-1 simulation

Method	Direction	Average number of zeros	Correctly classified zeros	Correctly classified nonzeros	Misclassification error rate
SSVD	u1	70.0	100.0	100.0	0.0
SSVD	v1	30.0	100.0	100.0	0.0
SSVD	u2	83.85	99.82	100.0	9.27
SSVD	v2	39.96	99.9	100.0	11.48
SVD	u1	0.0	0.0	100.0	70.0
SVD	v1	0.0	0.0	100.0	78.98
SVD	u2	0.0	0.0	100.0	91.93
SVD	v2	0.0	0.0	100.0	90.06

Table 4: Rank-2 simulation

5.1 Conclusion

In this paper, we implement the sparse singular value decomposition (SSVD) in Python, apply it on both simulated datasets and a real data set, and compare it with other biclustering methods. We find that SSVD is able to correctly identify zero entries and thus provide dimension reduction for high-dimension low sample size data. Since sparsity is imposed on rows and columns simultaneously, SSVD offers us a great new tool for biclusteering.

Although SSVD can converge within 5 to 10 iterations, it is actually very slow for large matrices. To improve efficiency in terms of runtime, we use JIT and Cython to optimize our code. Source code, test code, and reproducible examples can be found https://github.com/XingyuYan/SSVD. Further improvement can be made using C++ optimization.

Reference

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

2. Lee, Mihee, et al. "Web-based Supplementary Materials for Biclustering via Sparse Singular Value Decomposition."