

STA 663 Final Project

Aihua Li, Yuxuan Chen

04/27/2021

Abstract

In this report, we try to realize biclustering by utilizing Sparse Singular Value Decomposition (SSVD). SSVD is a tool for biclustering by seeking the low-rank matrix approximation with sparsed left and right singular vectors of original matrix. Note that a sparsity-inducing penalty is incorporated to get sparsity. We will describe this algorithm in detail and try to optimize the process by implementing cython. After optimization, applications of this algorithm on a simulated dataset and a real tumor dataset. Moreover, comparative analysis with simulated dataset are also conducted to compare the performance of SSVD, SVD and SPCA. The full repository is available on <https://github.com/YuxuanMonta/STA663-final-project-AY>.

Key words: SSVD, Sparsity, Simulation, BIC, Penalty, Approximation

1 Introduction to biclustering via SSVD

The research we focus on is *Biclustering via Sparse Singular Value Decomposition* written by Mihee Lee, Haipeng Shen, Jianhua Z. Huang, and J.S. Marron from University of North Carolina at Chapel Hill [1].

In the paper, as what is shown in the title, a new exploratory analysis tool for biclustering is introduced. This tool is also designed to seek the association between the column and row of data matrices with high dimension (singularity). A checkerboard structured matrix approximation to the original data matrices will be developed through this tool, achieved by sparsifying both the left and right singular vectors.

There are increasing number of dataset with high dimension in real life. In most case, those data with relatively higher dimension and smaller sample size is defined as "high-dimension low sample size (HDLSS)" in the paper. This kind of data could be a problem when being used to do classical statistical data analysis. Biclustering method, a collection of unsupervised learning tools, is developed to solve this kind of problem. Under this background, SSVD is proposed in this paper as a new tool of biclustering method. Besides SSVD, other algorithms like SVD and Sparse PCA (SPCA) can be also helpful when encountering the same problem. One typical application of SSVD algorithm is the microarray gene expression analysis, a limited number of individuals with numerous genes. SSVD can help to identify the groups of genes for different kind of diseases. Moreover, in text categorization, SSVD can be used to seek the pattern of words for different documents.

One advantage of SSVD compared to SVD is that SSVD can be useful to find sparsified structure of HDLSS data. Moreover, while Sparse PCA "only imposes sparse structure on one direction", SSVD can detect block structures in data matrices. However, because iteration method is needed to realize this algorithm, the computation cost can be expensive.

In our search, SSVD algorithm will be applied to get the approximation of sparse matrices of both simulated data set and real data set. Additionally, we will also compare SSVD algorithm with SVD algorithm and SPCA algorithm.

2 Algorithm

In order to realise SSVD algorithm, a low-rank matrix approximation matrix (rank-k) to the original data matrix should be first computed. This rank-k matrix here is the combination of top k ranks SSVD layer. Then in the iteration process, each step of iteration can generate a SSVD layer from the residual matrix of previous layer, based on a penalized sum-of-squares criterion where the degree of sparsity in each iteration depends on the BIC. This iteration process will continue until convergence. Afterward, the computed left and right singular vector of the SSVD can help to group the row and column space. Then an approximation matrix can be generated by sorting the singular vector in each group.

In the following part, we only focus on the rank-1 matrix approximation via SSVD.

Details of SSVD algorithm

- Step 1: Initialization

Apply the standard SVD to X and get the initial SVD decomposition u_0, s_0, v_0 . Note that $X = u_0 s_0 v_0^T$.

- Step2: Update

1. update v :

- For each λ_v , get $BIC(\lambda_v) = \frac{\|Y - \hat{Y}\|^2}{nd \cdot \hat{\sigma}_v^2} + \frac{\log(nd)}{nd} \hat{d}f(\lambda_v)$. Note $\hat{d}f(\lambda_v)$ is the degree of sparsity of v with λ_v as the penalty parameter, and $\hat{\sigma}_v^2$ is the OLS estimate of the error variance from $\|X - u\tilde{v}\|_F^2 + \lambda_v P_2(\tilde{v})$, where $\tilde{v} = sv$.
- Get the λ_v that minimizes $BIC(\lambda_v)$ (penalty).
- Set $\tilde{v}_j = \text{sign}(X^T u_0)_j (|(X^T u_0)_j| - \lambda_v w_{2,j}/2)_+$, $\{j = 1, \dots, d\}$, where $w_2 = |\hat{v}|^{-\gamma_2}$ and $\hat{v} = X^T u$
- Set $v_{new} = \frac{\tilde{v}}{s_v}$, where $\tilde{v} = (\tilde{v}_1, \dots, \tilde{v}_d)^T$ and $s_v = \|\tilde{v}\|$

2. update u :

- For each λ_u , get $BIC(\lambda_u) = \frac{\|Y - \hat{Y}\|^2}{nd \cdot \hat{\sigma}_u^2} + \frac{\log(nd)}{nd} \hat{d}f(\lambda_u)$. Note $\hat{d}f(\lambda_u)$ is the degree of sparsity of u with λ_u as the penalty parameter, and $\hat{\sigma}_u^2$ is the OLS estimate of the error variance from $\|X - \tilde{u}v\|_F^2 + \lambda_u P_1(\tilde{u})$, where $\tilde{u} = su$.
- Get the λ_u that minimizes $BIC(\lambda_u)$ (penalty).
- Set $\tilde{u}_i = \text{sign}(X^T v_0)_i (|(X^T v_0)_i| - \lambda_u w_{1,i}/2)_+$, $\{i = 1, \dots, n\}$, where $w_1 = |\hat{u}|^{-\gamma_1}$ and $\hat{u} = X^T v$
- Set $u_{new} = \frac{\tilde{u}}{s_u}$, where $\tilde{u} = (\tilde{u}_1, \dots, \tilde{u}_n)^T$ and $s_u = \|\tilde{u}\|$

3. Iteration:

Set $u_0 = u_{new}$ and $v_0 = v_{new}$ and go back to Step 2 (1) and Step 2 (2) until both the distances between u_0 and u_{new} as well as v_0 and v_{new} are smaller than tolerance.

- Step 3: Return $u = u_{new}$, $v = v_{new}$ and $s = u_{new}^T X v_{new}$ at convergence.

3 Coding and optimization

Starting from a most basic version of Python plain code, we further make 4-step improvement/modification. As shown in table 1, we have compared the computing time of the three key functions `update_uv()`, `SSVD_layer()`, and `SSVD()` in all the 5 versions. In this section, we will introduce the key idea of the optimization procedure. All codes are included in the appendix.

Version 1: The most basic plain codes

In the baseline version, we implement the SSVD algorithm according to the algorithm description. Most of the Python codes can be directly related to the mathematics in the paper, so that the codes is highly understandable.

Table 1: Average computing time comparisons in 10 runs 10 loops each (ms)

	<code>update_uv()</code>	<code>SSVD_layer()</code>	<code>SSVD()</code>	Description
Version1	351.8	1448.64	1268.01	basic
Version2	8.03	36.91	55.84	by linear algebra
Version3	3.84	15.52	15.12	by broadcasting
Version4	4.49	20.17	20.34	modify broadcasting
Version5	3.97	8.56	8.47	by Cython

Note that starting from this baseline version, there are three key functions:

- `update_uv()`, to update the vector u and v given the current values.
- `SSVD_layer()`, to compute the SSVD decomposition at the current layer.
- `SSVD()`, to compute the SSVD decomposition at all desired layers.

For the detailed full codes, see appendix A.

Version 2: Improve version 1 by linear algebra tricks

The baseline version, although intuitive, is terribly slow. After profiling the codes, a noticeable result is that the "ols calculation", (i.e., computing quantities like $\hat{\sigma}^2$), which involves the Kronecker product, dramatically slows down the computation, because working with the ols stuff indeed increases the matrix dimension from n or d into $n \times d$. Therefore, the computation of the ols are replaced in this version. As can be seen from table 1, this modification greatly improves the computing efficiency.

For the detailed full codes, see appendix B.

Version 3: Improve version 2 by broadcasting

Another thing that slows down the computation is the searching for the optimal λ from a given grid. In previous versions, it is done by loop with `map()`, while in this version, broadcasting is applied to replace all the loops. As a result, with the help of broadcasting, we further speed up the computation by half, as shown in table 1.

For the detailed full codes, see appendix C.

Version 4: Modify the broadcasting in version 3

Although broadcasting is a powerful tool to optimize the codes, there is a potential problem of its application in our context. In fact, one step that apply the broadcasting is to compute the outer product of vectors; then, conceivably, under the broadcasting, "outer product * lambda grid" can result in a quite huge matrix if the original data is of high dimension. Concerning this potential problem, in this version, we consider appropriately combine the broadcasting and the map loop. It turns out that the performance of this version is slightly worse than version 3 for the simulation data of moderate dimension, as shown in table 1; but overall, the performance is somewhat comparable.

For the detailed full codes, see appendix D.

Version 5: Cythonize version 3

Based on the previous results, we decided to work with version 3, and further improve it by Cython. It will be too tedious to explain all the details about how we cythonize the codes, but we'd like to mention some key points:

1. First of all, we found that Cython *doesn't always* bring improvement to the codes, especially compared with the broadcasting in Numpy.

In fact, we have tried to "fully" cythonize all the codes with as much parallelization as possible, and it turns out to be totally comparable, (or sometimes even slightly worse,) to the non-cythonized version. A possible reason is that there can be high overhead of parallelization, which sometimes exceed its benefits, especially compared to the powerful broadcasting in Numpy (which is already written in C).

Therefore, we convert the codes into Cython line by line, and see if each-line modification can make improvements. Only those that can actually speed up the codes will be kept.

2. Another noticeable point is that in the plain python version, everytime we update the vectors u and v , we require new spaces in the computer to store the updated vectors. Then, Cython allows us to always use the same storage for the "old" and "new" vectors. We have found that this is a step greatly improving the efficiency.

As shown in table 1, in fact, the performance of the Cython version is comparable to the version 3 in the function `update_uv()` (again, demonstrating the usefulness of basic Numpy), but Cython significantly improves the performance of the functions `SSVD_layer()` and `SSVD` (by saving storage).

For the detailed full codes, see appendix E.

4 Simulation

In this section, we realize the simulation in the original paper to demonstrate the clustering performance of our codes.

More specifically, we consider rank-1 true signal matrix $X^* = suv^T \in \mathbb{R}^{100 \times 50}$. Let $s = 50$, and

$$\begin{aligned}\tilde{u} &= [10, 9, 8, 7, 6, 5, 4, 3, r(2, 17), r(0, 75)]^T, & u &= \tilde{u}/\|\tilde{u}\|, \\ \tilde{v} &= [10, -10, 8, -8, 5, -5, r(3, 5), r(-3, 5), r(0, 34)]^T, & v &= \tilde{v}/\|\tilde{v}\|,\end{aligned}\tag{1}$$

where $r(a, b)$ denotes a vector of length b , whose entries are all a . Then, the data matrix X is generated by X^* plus a noise matrix ϵ , and here we consider standard normal noise.

Figure 1 shows the checkerboard plot (aka heatmap, SSVD first layer plot) of the true data decomposition, and figure 2 and 3 show the outputs of the 5 versions of codes.

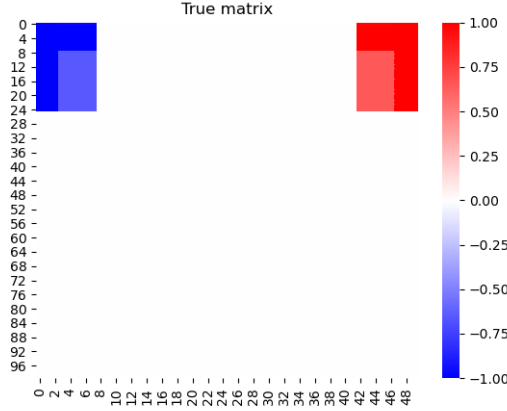


Figure 1: SSVD first layer plot of the true decomposition

As a result, only about first 25 entries are non-zero, which accords to our true normalized vector u with length 100 and last 75 data are all 0s.

Moreover, the original pattern can also be observed from the layer 1 approximation plot above, indicating SSVD reaches a good approximation of the true data by getting rid of the influence of noises.

5 Real data application

In this part, the real data sets we use is a gene expression cancer RNA-Seq data set. It is part of the RNA-Seq (HiSeq) PANCAN data set. This is a random extraction of gene expressions of patients having different types of tumor: BRCA, KIRC, COAD, LUAD and PRAD. (Data source: <http://archive.ics.uci.edu/ml/datasets/gene+expression+cancer+RNA-Seq>, see[2])

This dataset provides 801 subjects and 20531 genes. You may find the data in ‘data.csv’ and the 5 groups of gene in ‘labels.csv’.

In the real data set application process, we may use only 5000 genes of all 801 subjects instead of the whole dataset given the consideration of computation cost. The application are realized by the version 5 codes (i.e., Cython)

The subject grouping can be seen above from the three scatterplots among the first three sparse left singular vectors. The first vector and the second vector suggest that tumor KIRC are identified. The second vector and the third vector reach the same results. The first vector and the third vector cannot separate the data well.

Therefore the three vectors may not provide good separation of the five tumor types. Our initial guess is that this could be because the data itself and our guess is proved in next step.

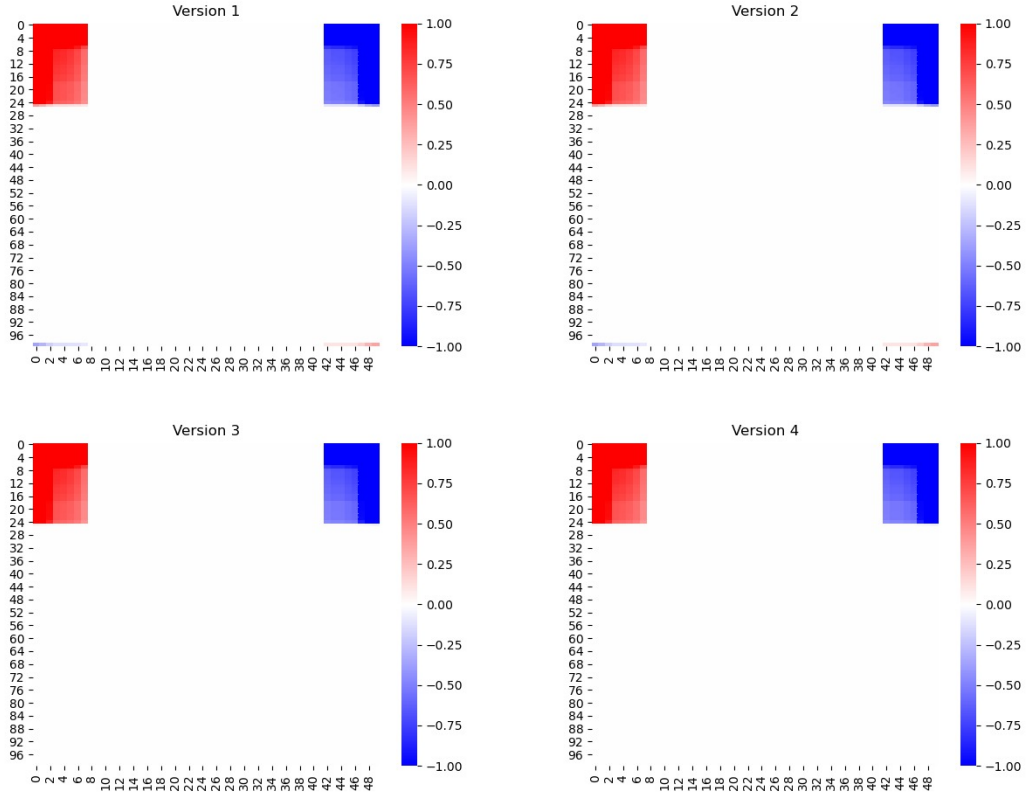


Figure 2: SSVD first layer plot of the outputs of version 1 to version 4

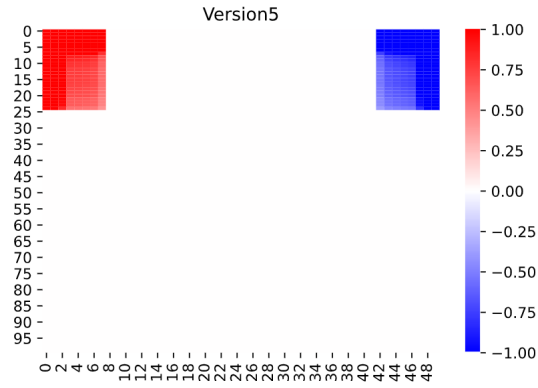


Figure 3: SSVD first layer plot of the outputs of version 5

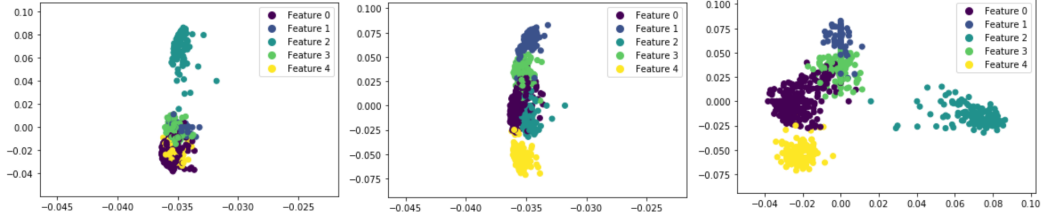


Figure 4: Tumor data: scatter plots of the first three entries in the SSVD decomposition. Left: u_1 vs u_2 , middle: u_1 vs u_3 , right: u_2 vs u_3

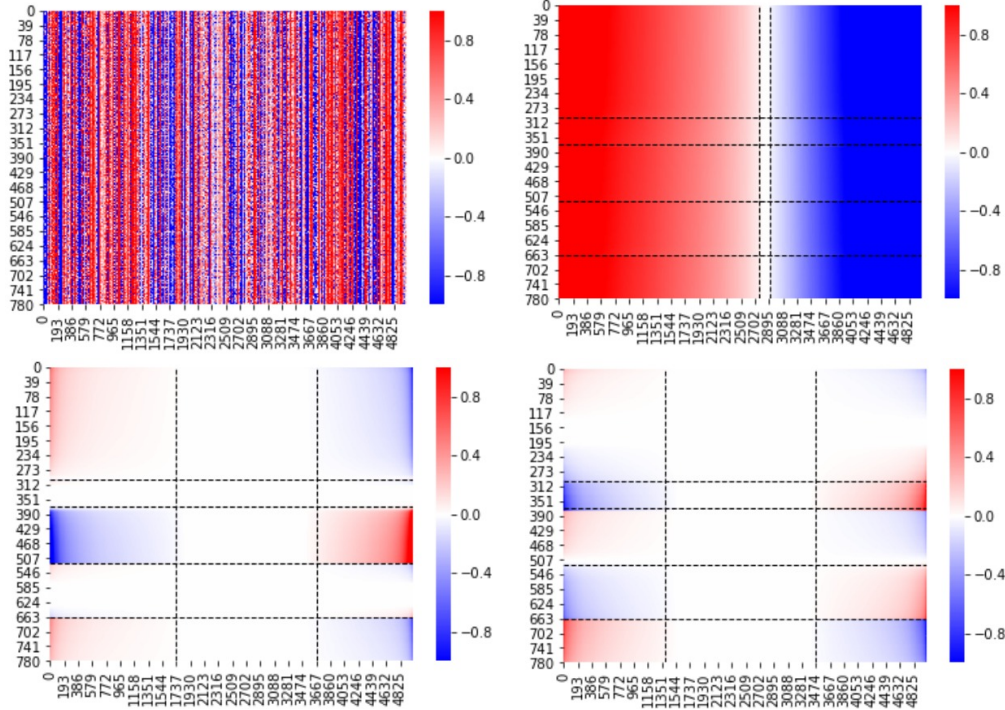


Figure 5: SSVD layer plot of the tumor data. Left top: heatmap of original dataset, right top: first layer of SSVD, left bottom: second layer of SSVD, right bottom: third layer of SSVD

In figure 5, the first 3 SSVD layers are clearly plotted. Based on the layer 1 plot, we can see a similar pattern for each tumor group. Thus, we will interpret the plot by simply ignoring the group. In this way, the figure can suggest that genes ordered from about 0 to round 2750 positively expressed for all tumors, while genes ordered from 2900 negatively expressed for all tumors. Layer 2 suggests a contrast between 'BRCA'/'PRAD' and 'KIRC', also zeros out the other two types. Layer 3 suggests a approximately contrast between 'BRCA'/'PRAD'/'KIRC' and 'COAD'/'LUAD'.

By combining with the heatmap of original polt, it suggest that the SSVD algorithm works well for

identifying the approximation of original dataset.

6 Comparative analysis

In this comparative analysis part, we will compare SSVD with SVD and Sparse PCA algorithm. Note that SPCA here uses 2 as weight parameter.

Rank 1 decomposition of this three methods will be compared. The true u and v are defined by the code in the next cell and normal distributed noises are incorporated to get test data.

For accuracy, we will compare four indicators.

1. Average number of zeros in u and v
2. Average number of correctly identified 0s in u and v
3. Average number of correctly identified non-0s in u and v
4. Rate of correctly identified all the positions of 0s AND non-0s (Correct classification rate) in u and v

Figure 6, 7 summarize the comparison results.

	Avg. # of 0s	Avg. # of correctly identified 0s	Avg. # of correctly identified non0s	Rate of correctly identified all 0s
u_ssvd	74.91	74.91	25.0	0.91
v_ssvd	33.88	33.88	16.0	0.88
u_svd	0.00	0.00	25.0	0.00
v_svd	0.00	0.00	16.0	0.00
u_spca	75.00	75.00	25.0	1.00
v_spca	34.00	34.00	16.0	1.00

Figure 6: Accuracy comparisons between algorithms

	Time(sec)
Time of SPCA	11.873480
Time of SSVD	3.345996
Time of SVD	0.603197

Figure 7: Running time comparisons between algorithms

Accuracy

The correct classification rate of SPCA indicates that SPCA successfully identifies all the positions of 0s and non-0s in both u and v . The correct classification rate of SSVD is also acceptable in both u and v . Moreover, the average number of 0s, average number of correctly identified 0s and average number of correctly identified non0s of SPCA and SSVD are all excellent for both u and v

However, given the nature of SVD algorithm, the correct classification rate of SVD is 0 for both u and v , indicating there is no 0s in the approximations of all layers by also referring the 0s in column 1 and column 2 for both u and v . Nevertheless, the average number of correctly identified non-0s of all methods are perfect.

Running Time

As shown in the last table above, SPCA has the longest running time all SVD has least running time.

7 Conclusion and discussion

After going through this project, SSVD is clearly an useful tool to develop a low rank (rank = 1 for most of time in this project) matrix approximation for those HDLSS datasets. It has relatively high accuracy and acceptable efficiency. For the usage of SSVD, this project mainly focuses on microarray gene expression analysis, and a good approximation is obtained.

There are some limitations of our product. Even though Cython has greatly improved the computation efficiency, a full C version is still worth trying to see if the computation can be fully optimized. Besides, in the tumor data application, for illustration purpose, it is sufficient to consider up to 5000 genes. Although, as we discussed before, the SSVD by the current 5000 genes has a powerful classification ability, for future study, we may expect to use the full data set to have a more detailed investigation. Last but not least, maybe a new sparsity-inducing penalty instead of the adaptive lasso penalty (weight parameter is 2 in this project) can be introduced.

8 References

- [1] Lee, M., Shen, H., Huang, J., and Marron, J. (2010). Biclustering via sparse singular value decomposition. *Biometrics* **66**, 1087-1095.
- [2] UCI machine Learning Repository: Gene EXPRESSION CANCER rna-seq data set. (n.d.). Retrieved April 28, 2021, from <http://archive.ics.uci.edu/ml/datasets/gene+expression+cancer+RNA-Seq>

Appendix A Version 1

Basic plain codes

```
1  ## This the most basic realization of the SSVD algorithm.
2  ## Codes directly correspond to the paper context.
3
4  import numpy as np
5  import scipy.linalg as la
6
7
8  def get_tilde(lam, tilde_hat, w):
9      """Get the tilde vector given the lambda, tilde_hat_vector, and w."""
10     part1 = np.sign(tilde_hat)
11     part2 = np.abs(tilde_hat)-lam*w/2
12     part2[part2<np.zeros(part2.shape)] = 0
13     tilde = np.multiply(part1, part2)
14     return tilde
15
16 def get_BIC(lam, w, tilde_hat, fixed_vec, response, sigma2_hat, nd):
17     """ Compute BIC given the lambda.
18     lam: scalar
19     For v:
20         w = w2: (d, 1)
21         tilde_hat = v_tilde_hat: (d, 1)
22         fixed_vec = u_old: (n, 1)
23         response = Y: (nd, 1)
24         sigma2_hat = sigma2_hat_v: scalar
25         nd: scalar, =n*d
26     For u:
27         w = w1: (n, 1)
28         tilde_hat = u_tilde_hat: (n, 1)
29         fixed_vec = v_new: (d, 1)
30         response = Z: (nd, 1)
31         sigma2_hat = sigma2_hat_u: scalar
32         nd: scalar, =n*d
33     Return: BIC, scalar
34     """
35     df = np.sum(np.abs(tilde_hat) > lam*w/2)
36     tilde = get_tilde(lam, tilde_hat, w) # get the tilde vector under the current lambda
37     response_hat = np.kron(np.eye(tilde.shape[0]), fixed_vec) @ tilde
38     BIC = np.linalg.norm(response-response_hat)**2/(nd*sigma2_hat)+np.log(nd)/nd*df
39     return BIC
40 def select_lam(lam_grid, w, tilde_hat, fixed_vec, response, sigma2_hat, nd):
41     """ Select lambda given a lambda grid based on BIC.
42     lam_grid: (S,)
43     For v:
44         w = w2: (d, 1)
45         tilde_hat = v_tilde_hat: (d, 1)
46         fixed_vec = u_old: (n, 1)
47         response = Y: (nd, 1)
48         sigma2_hat = sigma2_hat_v: scalar
49         nd: scalar, =n*d
50     For u:
51         w = w1: (n, 1)
52         tilde_hat = u_tilde_hat: (n, 1)
53         fixed_vec = v_new: (d, 1)
54         response = Z: (nd, 1)
55         sigma2_hat = sigma2_hat_u: scalar
56         nd: scalar, =n*d
57     Return: scalar, the optimal lambda
58     """
59     BICs = list(map(lambda lam: get_BIC(lam, w, tilde_hat, fixed_vec, response, sigma2_hat, nd),
60                     lam_grid))
61     return lam_grid[np.argmin(BICs)]
62 def update_uv(u_old, v_old, X, Y, Z, gamma1, gamma2, lam_grid):
63     """Update u and v once given the current u and v.
```

```

63 Input:
64     u_old: (n, 1)
65     v_old: (d, 1)
66     X: (n, d)
67     Y: (nd, 1)
68     Z: (nd, 1)
69     gamma1, gamma2: scalar, tuning parameter
70     lam_grid: ndarray, a grid of lambdas to be selected
71 Return:
72     u_new: (n, 1)
73     v_new: (d, 1)
74     lambda_u, lambda_v: scalar, the optimal lambda under the current u and v
75 """
76 n, d = X.shape
77 nd = n*d
78
79 ## Update v using current u
80
81 # ols for v, use current u
82 v_tilde_hat = X.T @ u_old # (d, 1), fixed ols estimate
83 Y_hat = (np.kron(np.eye(d), u_old) @ v_tilde_hat).reshape((-1,1)) # (nd, 1), ols estimate
84 sigma2_hat_v = ((Y-Y_hat).T @ (Y-Y_hat) / (nd-d))[0][0] # scalar, fixed ols estimate
85
86 # select lambda_v
87 w2 = np.abs(v_tilde_hat)**(-gamma2) # (d, 1)
88 lambda_v = select_lam(lam_grid, w2, v_tilde_hat, u_old, Y, sigma2_hat_v, nd)
89
90 # update v
91 v_tilde = get_tilde(lambda_v, v_tilde_hat, w2) # (d, 1)
92
93 if np.all(v_tilde == 0): # full shrinkage at v
94     v_new = np.zeros(v_old.shape)
95     u_new = np.zeros(u_old.shape)
96     lambda_u = 0
97 else:
98     v_new = v_tilde / np.linalg.norm(v_tilde)
99
100 ## Update u using current v
101
102 # ols for u, use current v
103 u_tilde_hat = X @ v_new # (n, 1), fixed ols estimate
104 Z_hat = np.kron(np.eye(n), v_new) @ u_tilde_hat # (nd, 1), ols estimate
105 sigma2_hat_u = ((Z-Z_hat).T @ (Z-Z_hat) / (nd-n))[0][0] # scalar, fixed ols estimate
106
107 # select lambda_u
108 w1 = np.abs(u_tilde_hat)**(-gamma1)
109 lambda_u = select_lam(lam_grid, w1, u_tilde_hat, v_new, Z, sigma2_hat_u, nd)
110
111 # update u
112 u_tilde = get_tilde(lambda_u, u_tilde_hat, w1) # (n, 1)
113 u_new = u_tilde / np.linalg.norm(u_tilde) if np.linalg.norm(u_tilde) != 0 else u_tilde
114
115 return u_new, v_new, lambda_u, lambda_v
116
117
118 def SSVd_layer(X, lam_grid, gamma1, gamma2, max_iter=5000, tol=1e-6):
119     """Get the sparse SVD layer given the data matrix X at a SVD layer and the tuning parameters
    grid.
120     Input:
121         X: (n, d), can be the original data matrix or the residual matrix
122         lam_grid: ndarray, a grid of lambdas to be selected
123         gamma1, gamma2: scalar, tuning parameter
124         max_iter: integer, the maximum iteration times
125         tol: float, tolerance to stop the iteration
126     Return: n_iter, u, v, s, lambda_u, lambda_v
127         n_iter: number of iterations
128         u: (n, 1), the final u at convergence
129         v: (d, 1), the final v at convergence
130         s: scalar, the singular value at convergence

```

```

131         lambda_u, lambda_v: the optimal tuning parameter at convergence
132         """
133         # SVD
134         U, _, VT = la.svd(X)
135         # prepare vector Y and Z, which are fixed after given X
136         Y = X.T.reshape((-1,1)) # (nd, 1)
137         Z = X.reshape((-1,1)) # (nd, 1)
138
139         # initial value
140         u_old = U[:,0][:,None]
141         v_old = VT[0][:,None]
142
143         for i in range(max_iter):
144             u_new, v_new, lambda_u, lambda_v = update_uv(u_old, v_old, X, Y, Z, gamma1, gamma2,
145                 lam_grid)
146             if np.linalg.norm(u_new-u_old) < tol and np.linalg.norm(v_new-v_old) < tol: # achieve the
147                 tolerance
148                 break
149                 if np.all(u_new == 0) or np.all(v_new == 0): # full shrinkage (i.e., all zeros in the
150                     vector)
151                     print("Warning: Full shrinkage has been achieved. Iterations stops. No further
152                         decomposition. The desired number of layers may not be achieved. ")
153                     break
154                     u_old, v_old = u_new, v_new
155             n_iter = i+1 # number of iterations
156             u, v = u_new, v_new # the final u and v at convergence
157             s = (u_new.T @ X @ v_new)[0][0]
158             if n_iter == max_iter:
159                 print("Warning: The maximum iteration has been achieved. Please consider increasing '
160                     max_iter'.")
161             return n_iter, u, v, s, lambda_u, lambda_v
162
163 def SSVD(X, num_layer, lam_grid, gamma1, gamma2, max_iter=5000, tol=1e-6):
164     """Get the SSVD given the data matrix X and the desired number of SSVD layers.
165     Input:
166         X: (n, d), the original data matrix
167         num_layer: desired number of SSVD layers
168         lam_grid: ndarray, a grid of lambdas to be selected
169         gamma1, gamma2: scalar, tuning parameter
170         max_iter: integer, the maximum iteration times
171         tol: float, tolerance to stop the iteration
172     Return: n_iters, us, vs, ss, lambda_us, lambda_vs
173         n_iters: (num_layer,), number of iterations for each layer
174         us: (n, num_layer), the final u at convergence for each layer
175         vs: (d, num_layer), the final v at convergence for each layer
176         ss: (num_layer,), the singular value at convergence for each layer
177         lambda_us, lambda_vs: (num_layer,), the optimal tuning parameter at convergence for each
178         layer
179     """
180     n, d = X.shape
181     n_iters = np.zeros(num_layer, dtype = int)
182     ss = np.zeros(num_layer)
183     lambda_us = np.zeros(num_layer)
184     lambda_vs = np.zeros(num_layer)
185     us = np.zeros((n, num_layer))
186     vs = np.zeros((d, num_layer))
187     # initial value
188     u = np.zeros((n, 1)); v = np.zeros((d, 1)); resi_mat = X; s = 0
189     for i in range(num_layer):
190         resi_mat = resi_mat - s*u@v.T
191         n_iter, u, v, s, lambda_u, lambda_v = SSVD_layer(resi_mat, lam_grid, gamma1, gamma2,
192             max_iter, tol)
193         n_iters[i] = n_iter
194         ss[i] = s
195         lambda_us[i] = lambda_u
196         lambda_vs[i] = lambda_v
197         us[:,i] = u[:,0]
198         vs[:,i] = v[:,0]
199         if np.all(u == 0) or np.all(v == 0): # full shrinkage (i.e., all zeros in the vector)

```

```

193         break
194     return n_iters, us, vs, ss, lambda_us, lambda_vs

```

Appendix B Version 2

First-step improvement by linear algebra

```

1  ## This is the first-step optimization, based on SSVDversion1.
2  ## Based on linear algebra,
3  ## computation about about Y, Z, and the Kronecker product are replaced.
4
5  import numpy as np
6  import scipy.linalg as la
7
8  def get_tilde(lam, tilde_hat, w):
9      """Get the tilde vector given the lambda, tilde_hat_vector, and w."""
10     part1 = np.sign(tilde_hat)
11     part2 = np.abs(tilde_hat) - lam*w/2
12     part2[part2 < np.zeros(part2.shape)] = 0
13     tilde = np.multiply(part1, part2)
14     return tilde
15
16 def get_BIC(lam, w, tilde_hat, fixed_vec, sigma2_hat, nd, X, fixed_name):
17     """ Compute BIC given the lambda.
18     lam: scalar
19     For v:
20         w = w2: (d, 1)
21         tilde_hat = v_tilde_hat: (d, 1)
22         fixed_vec = u_old: (n, 1)
23         response = Y: (nd, 1)
24         sigma2_hat = sigma2_hat_v: scalar
25         nd: scalar, =n*d
26     For u:
27         w = w1: (n, 1)
28         tilde_hat = u_tilde_hat: (n, 1)
29         fixed_vec = v_new: (d, 1)
30         response = Z: (nd, 1)
31         sigma2_hat = sigma2_hat_u: scalar
32         nd: scalar, =n*d
33     Return: BIC, scalar
34     """
35     df = np.sum(np.abs(tilde_hat) > lam*w/2)
36     tilde = get_tilde(lam, tilde_hat, w)
37     SSE = np.sum((X - np.outer(fixed_vec, tilde))**2) if fixed_name == "u" else\
38         np.sum((X - np.outer(tilde, fixed_vec))**2)
39     BIC = SSE/(nd*sigma2_hat) + np.log(nd)/nd*df
40     return BIC
41
42 def select_lam(lam_grid, w, tilde_hat, fixed_vec, sigma2_hat, nd, X, fixed_name):
43     """ Select lambda given a lambda grid based on BIC.
44     lam_grid: (S,)
45     For v:
46         w = w2: (d, 1)
47         tilde_hat = v_tilde_hat: (d, 1)
48         fixed_vec = u_old: (n, 1)
49         response = Y: (nd, 1)
50         sigma2_hat = sigma2_hat_v: scalar
51         nd: scalar, =n*d
52     For u:
53         w = w1: (n, 1)
54         tilde_hat = u_tilde_hat: (n, 1)
55         fixed_vec = v_new: (d, 1)
56         response = Z: (nd, 1)
57         sigma2_hat = sigma2_hat_u: scalar
58         nd: scalar, =n*d

```

```

57 Return: scalar, the optimal lambda
58 """
59 BICs = list(map(lambda lam: get_BIC(lam, w, tilde_hat, fixed_vec, sigma2_hat, nd, X, fixed_name
60 ), lam_grid))
61 return lam_grid[np.argmin(BICs)]
62
63 def update_uv(u_old, v_old, X, gamma1, gamma2, lam_grid):
64     """Update u and v once given the current u and v.
65     Input:
66         u_old: (n, 1)
67         v_old: (d, 1)
68         X: (n, d)
69         Y: (nd, 1)
70         Z: (nd, 1)
71         gamma1, gamma2: scalar, tuning parameter
72         lam_grid: ndarray, a grid of lambdas to be selected
73     Return:
74         u_new: (n, 1)
75         v_new: (d, 1)
76         lambda_u, lambda_v: scalar, the optimal lambda under the current u and v
77     """
78     n, d = X.shape
79     nd = n*d
80
81     ## Update v using current u
82
83     # ols for v, use current u
84     v_tilde_hat = X.T @ u_old # (d, 1), fixed ols estimate
85     SSE_v = np.sum((X - np.outer(u_old, v_tilde_hat))**2) # scalar, SSE_v = (Y-Y_hat).T @ (Y-Y_hat)
86     sigma2_hat_v = SSE_v / (nd-d) # scalar, fixed ols estimate
87
88     # select lambda_v
89     w2 = np.abs(v_tilde_hat)**(-gamma2) # (d, 1)
90     lambda_v = select_lam(lam_grid, w2, v_tilde_hat, u_old, sigma2_hat_v, nd, X, "u")
91
92     # update v
93     v_tilde = get_tilde(lambda_v, v_tilde_hat, w2) # (d, 1)
94
95     if np.all(v_tilde == 0): # full shrinkage at v
96         v_new = np.zeros(v_old.shape)
97         u_new = np.zeros(u_old.shape)
98         lambda_u = 0
99     else:
100         v_new = v_tilde / np.linalg.norm(v_tilde)
101
102     ## Update u using current v
103
104     # ols for u, use current v
105     u_tilde_hat = X @ v_new # (n, 1), fixed ols estimate
106     SSE_u = np.sum((X.T - np.outer(v_new, u_tilde_hat))**2)
107     sigma2_hat_u = SSE_u / (nd-n) # scalar, fixed ols estimate
108
109     # select lambda_u
110     w1 = np.abs(u_tilde_hat)**(-gamma1)
111     lambda_u = select_lam(lam_grid, w1, u_tilde_hat, v_new, sigma2_hat_u, nd, X, "v")
112
113     # update u
114     u_tilde = get_tilde(lambda_u, u_tilde_hat, w1) # (n, 1)
115     u_new = u_tilde / np.linalg.norm(u_tilde) if np.linalg.norm(u_tilde) != 0 else u_tilde
116
117     return u_new, v_new, lambda_u, lambda_v
118
119 def SSVD_layer(X, lam_grid, gamma1, gamma2, max_iter=5000, tol=1e-6):
120     """Get the sparse SVD layer given the data matrix X at a SVD layer and the tuning parameters
121     grid.
122     Input:
123         X: (n, d), can be the original data matrix or the residual matrix

```

```

123     lam_grid: ndarray, a grid of lambdas to be selected
124     gamma1, gamma2: scalar, tuning parameter
125     max_iter: integer, the maximum iteration times
126     tol: float, tolerance to stop the iteration
127 Return: n_iter, u, v, s, lambda_u, lambda_v
128     n_iter: number of iterations
129     u: (n, 1), the final u at convergence
130     v: (d, 1), the final v at convergence
131     s: scalar, the singular value at convergence
132     lambda_u, lambda_v: the optimal tuning parameter at convergence
133 """
134 # SVD
135 U, _, VT = la.svd(X)
136
137 # initial value
138 u_old = U[:,0][:,None]
139 v_old = VT[0][:,None]
140
141 for i in range(max_iter):
142     u_new, v_new, lambda_u, lambda_v = update_uv(u_old, v_old, X, gamma1, gamma2, lam_grid)
143     if np.linalg.norm(u_new-u_old) < tol and np.linalg.norm(v_new-v_old) < tol: # achieve the
        tolerance
144         break
145     if np.all(u_new == 0) or np.all(v_new == 0): # full shrinkage (i.e., all zeros in the
        vector)
146         print("Warning: Full shrinkage has been achieved. Iterations stops. No further
            decomposition. The desired number of layers may not be achieved. ")
147         break
148     u_old, v_old = u_new, v_new
149     n_iter = i+1 # number of iterations
150     u, v = u_new, v_new # the final u and v at convergence
151     s = (u_new.T @ X @ v_new)[0][0]
152     if n_iter == max_iter:
153         print("Warning: The maximum iteration has been achieved. Please consider increasing '
            max_iter'.")
154     return n_iter, u, v, s, lambda_u, lambda_v
155
156 def SSVD(X, num_layer, lam_grid, gamma1, gamma2, max_iter=5000, tol=1e-6):
157     """Get the SSVD given the data matrix X and the desired number of SSVD layers.
158     Input:
159         X: (n, d), the original data matrix
160         num_layer: desired number of SSVD layers
161         lam_grid: ndarray, a grid of lambdas to be selected
162         gamma1, gamma2: scalar, tuning parameter
163         max_iter: integer, the maximum iteration times
164         tol: float, tolerance to stop the iteration
165     Return: n_iters, us, vs, ss, lambda_us, lambda_vs
166         n_iters: (num_layer,), number of iterations for each layer
167         us: (n, num_layer), the final u at convergence for each layer
168         vs: (d, num_layer), the final v at convergence for each layer
169         ss: (num_layer,), the singular value at convergence for each layer
170         lambda_us, lambda_vs: (num_layer,), the optimal tuning parameter at convergence for each
            layer
171     """
172     n, d = X.shape
173     n_iters = np.zeros(num_layer, dtype = int)
174     ss = np.zeros(num_layer)
175     lambda_us = np.zeros(num_layer)
176     lambda_vs = np.zeros(num_layer)
177     us = np.zeros((n, num_layer))
178     vs = np.zeros((d, num_layer))
179     # initial value
180     u = np.zeros((n, 1)); v = np.zeros((d, 1)); resi_mat = X; s = 0
181     for i in range(num_layer):
182         resi_mat = resi_mat - s*u@v.T
183         n_iter, u, v, s, lambda_u, lambda_v = SSVD_layer(resi_mat, lam_grid, gamma1, gamma2,
            max_iter, tol)
184         n_iters[i] = n_iter
185         ss[i] = s

```



```

186     lambda_us[i] = lambda_u
187     lambda_vs[i] = lambda_v
188     us[:,i] = u[:,0]
189     vs[:,i] = v[:,0]
190     if np.all(u == 0) or np.all(v == 0): # full shrinkage (i.e., all zeros in the vector)
191         break
192     return n_iters, us, vs, ss, lambda_us, lambda_vs

```

Appendix C Version 3

Second-step improvement by broadcasting

```

1  ## This is the second-step optimisation, based on SSVDversion2.
2  ## Broadcasting are applied when selecting optimal lambda from a given grid.
3  ## Function get_tilde(), get_BIC(), and select_lam() are no longer necessary.
4
5  import numpy as np
6  import scipy.linalg as la
7
8  def update_uv(u_old, v_old, X, gamma1, gamma2, lam_grid):
9      """Update u and v once given the current u and v.
10     Input:
11         u_old: (n, 1)
12         v_old: (d, 1)
13         X: (n, d)
14         Y: (nd, 1)
15         Z: (nd, 1)
16         gamma1, gamma2: scalar, tuning parameter
17         lam_grid: ndarray, a grid of lambdas to be selected
18     Return:
19         u_new: (n, 1)
20         v_new: (d, 1)
21         lambda_u, lambda_v: scalar, the optimal lambda under the current u and v
22     """
23     n, d = X.shape
24     nd = n*d
25     S = len(lam_grid)
26
27     # initialize
28     v_new = np.zeros(v_old.shape)
29     u_new = np.zeros(u_old.shape)
30     lambda_u = lambda_v = 0
31
32     ## Update v using current u
33
34     # ols for v, use current u
35     v_tilde_hat = X.T @ u_old # (d, 1), fixed ols estimate
36     SSE_v = np.sum((X - np.outer(u_old, v_tilde_hat))**2) # scalar, SSE_v = (Y-Y_hat).T @ (Y-Y_hat)
37     sigma2_hat_v = SSE_v / (nd-d) # scalar, fixed ols estimate
38
39     # select lambda_v
40     w2 = np.abs(v_tilde_hat)**(-gamma2) # (d, 1)
41     dfs_v = np.sum(np.abs(v_tilde_hat) > lam_grid*w2/2, axis=0) # (S,), df for each lambda
42     part2 = v_tilde_hat - lam_grid*w2/2; part2[part2<0] = 0
43     part1 = np.sign(v_tilde_hat)
44     v_tilde_br = part1 * part2 # (d, S), each column is v_tilde under each lambda
45     outer_prods = np.outer(u_old, v_tilde_br.T) # (n, d*S), each chunk of size (n, d) is the outer
46     # product mat under each lambda
47     outer_prods = np.array(np.hsplit(outer_prods, S)) # (S, n, d)
48     SSEs_v = ((X-outer_prods)**2).sum(axis = (1,2)) # (S,), ||X-uv_tilde^T||_F^2=||Y-Y_hat||^2 for
49     # each lambda
50     BICs_v = SSEs_v/(nd*sigma2_hat_v) + np.log(nd)/nd*dfs_v # (S,), BIC for each lambda

```

```

49 lambda_v = lam_grid[np.argmin(BICs_v)]
50
51 # update v
52 part1 = np.sign(v_tilde_hat)
53 part2 = np.abs(v_tilde_hat)-lambda_v*w2/2; part2[part2<0] = 0
54 if not np.all(part2 == 0): # not full shrinkage at v
55     v_tilde = np.multiply(part1, part2) # (d, 1)
56     v_new = v_tilde / np.linalg.norm(v_tilde)
57
58 ## Update u using current v
59
60 # ols for u, use current v
61 u_tilde_hat = X @ v_new # (n, 1), fixed ols estimate
62 SSE_u = np.sum((X.T - np.outer(v_new, u_tilde_hat))**2)
63 sigma2_hat_u = SSE_u / (nd-n) # scalar, fixed ols estimate
64
65 # select lambda_u
66 w1 = np.abs(u_tilde_hat)**(-gamma1) # (n, 1)
67 dfs_u = np.sum(np.abs(u_tilde_hat) > lam_grid*w1/2, axis=0) # (S,), df for each lambda
68 part2 = u_tilde_hat - lam_grid*w1/2; part2[part2<0] = 0
69 part1 = np.sign(u_tilde_hat)
70 u_tilde_br = part1 * part2 # (n, S), each column is u_tilde under each lambda
71 outer_prods = np.outer(u_tilde_br.T, v_new) # (n*S, d), each chunk of size (n, d) is the
outer product mat under each lambda
72 outer_prods = np.array(np.vsplit(outer_prods, S)) # (S, n, d)
73 SSEs_u = ((X-outer_prods)**2).sum(axis = (1,2)) # (S,), ||X-u_tildev^T||_F^2=||Z-Z_hat||^2
for each lambda
74 BICs_u = SSEs_u/(nd*sigma2_hat_u) + np.log(nd)/nd*dfs_u # (S,), BIC for each lambda
75 lambda_u = lam_grid[np.argmin(BICs_u)]
76
77 # update u
78 part1 = np.sign(u_tilde_hat)
79 part2 = np.abs(u_tilde_hat)-lambda_u*w1/2; part2[part2<0] = 0
80 if not np.all(part2 == 0): # not full shrinkage at u
81     u_tilde = np.multiply(part1, part2) # (n, 1)
82     u_new = u_tilde / np.linalg.norm(u_tilde)
83
84 return u_new, v_new, lambda_u, lambda_v
85
86
87 def SSVD_layer(X, lam_grid, gamma1, gamma2, max_iter=5000, tol=1e-6):
88     """Get the sparse SVD layer given the data matrix X at a SVD layer and the tuning parameters
grid.
89     Input:
90         X: (n, d), can be the original data matrix or the residual matrix
91         lam_grid: ndarray, a grid of lambdas to be selected
92         gamma1, gamma2: scalar, tuning parameter
93         max_iter: integer, the maximum iteration times
94         tol: float, tolerance to stop the iteration
95     Return: n_iter, u, v, s, lambda_u, lambda_v
96         n_iter: number of iterations
97         u: (n, 1), the final u at convergence
98         v: (d, 1), the final v at convergence
99         s: scalar, the singular value at convergence
100         lambda_u, lambda_v: the optimal tuning parameter at convergence
101     """
102     # SVD
103     U, _, VT = la.svd(X)
104
105     # initial value
106     u_old = U[:,0][:,None]
107     v_old = VT[0][:,None]
108
109     for i in range(max_iter):
110         u_new, v_new, lambda_u, lambda_v = update_uv(u_old, v_old, X, gamma1, gamma2, lam_grid)
111         if np.linalg.norm(u_new-u_old) < tol and np.linalg.norm(v_new-v_old) < tol: # achieve the
tolerance
112             break

```

```

113     if np.all(u_new == 0) or np.all(v_new == 0): # full shrinkage (i.e., all zeros in the
        vector)
114         print("Warning: Full shrinkage has been achieved. Iterations stops. No further
            decomposition. The desired number of layers may not be achieved. ")
115         break
116         u_old, v_old = u_new, v_new
117     n_iter = i+1 # number of iterations
118     u, v = u_new, v_new # the final u and v at convergence
119     s = (u_new.T @ X @ v_new)[0][0]
120     if n_iter == max_iter:
121         print("Warning: The maximum iteration has been achieved. Please consider increasing '
            max_iter'.")
122     return n_iter, u, v, s, lambda_u, lambda_v
123
124 def SSVD(X, num_layer, lam_grid, gamma1, gamma2, max_iter=5000, tol=1e-6):
125     """Get the SSVD given the data matrix X and the desired number of SSVD layers.
126     Input:
127         X: (n, d), the original data matrix
128         num_layer: desired number of SSVD layers
129         lam_grid: ndarray, a grid of lambdas to be selected
130         gamma1, gamma2: scalar, tuning parameter
131         max_iter: integer, the maximum iteration times
132         tol: float, tolerance to stop the iteration
133     Return: n_iters, us, vs, ss, lambda_us, lambda_vs
134         n_iters: (num_layer,), number of iterations for each layer
135         us: (n, num_layer), the final u at convergence for each layer
136         vs: (d, num_layer), the final v at convergence for each layer
137         ss: (num_layer,), the singular value at convergence for each layer
138         lambda_us, lambda_vs: (num_layer,), the optimal tuning parameter at convergence for each
            layer
139     """
140     n, d = X.shape
141     n_iters = np.zeros(num_layer, dtype = int)
142     ss = np.zeros(num_layer)
143     lambda_us = np.zeros(num_layer)
144     lambda_vs = np.zeros(num_layer)
145     us = np.zeros((n, num_layer))
146     vs = np.zeros((d, num_layer))
147     # initial value
148     u = np.zeros((n, 1)); v = np.zeros((d, 1)); resi_mat = X; s = 0
149     for i in range(num_layer):
150         resi_mat = resi_mat - s*u@v.T
151         n_iter, u, v, s, lambda_u, lambda_v = SSVD_layer(resi_mat, lam_grid, gamma1, gamma2,
            max_iter, tol)
152         n_iters[i] = n_iter
153         ss[i] = s
154         lambda_us[i] = lambda_u
155         lambda_vs[i] = lambda_v
156         us[:,i] = u[:,0]
157         vs[:,i] = v[:,0]
158         if np.all(u == 0) or np.all(v == 0): # full shrinkage (i.e., all zeros in the vector)
159             break
160     return n_iters, us, vs, ss, lambda_us, lambda_vs

```

Appendix D Version 4

A comparable modication combining broadcasting and map

```

1 ## This is a modified version of SSVDversion3.
2 ## The broadcasting may result in extremely huge matrix (as we have to compute the outer product),
3 ## which can be counterproductive if it indeed slows the computation.
4 ## In this version, we appropriately combine broadcasting and the map loop.
5

```

```

6 import numpy as np
7 import scipy.linalg as la
8
9
10 def get_outer_prod(tilde_vec, fixed_vec, fixed_name):
11     """Compute the outer product according to the formula. """
12     if fixed_name == "u":
13         return np.outer(fixed_vec, tilde_vec)
14     else:
15         return np.outer(tilde_vec, fixed_vec)
16
17
18 def update_uv(u_old, v_old, X, gamma1, gamma2, lam_grid):
19     """Update u and v once given the current u and v.
20     Input:
21         u_old: (n, 1)
22         v_old: (d, 1)
23         X: (n, d)
24         Y: (nd, 1)
25         Z: (nd, 1)
26         gamma1, gamma2: scalar, tuning parameter
27         lam_grid: ndarray, a grid of lambdas to be selected
28     Return:
29         u_new: (n, 1)
30         v_new: (d, 1)
31         lambda_u, lambda_v: scalar, the optimal lambda under the current u and v
32     """
33     n, d = X.shape
34     nd = n*d
35     S = len(lam_grid)
36
37     # initialize
38     v_new = np.zeros(v_old.shape)
39     u_new = np.zeros(u_old.shape)
40     lambda_u = lambda_v = 0
41     SSEs_v = SSEs_u = np.zeros(S)
42
43     ## Update v using current u
44
45     # ols for v, use current u
46     v_tilde_hat = X.T @ u_old # (d, 1), fixed ols estimate
47     SSE_v = np.sum((X - np.outer(u_old, v_tilde_hat))**2) # scalar, SSE_v = (Y-Y_hat).T @ (Y-Y_hat)
48     sigma2_hat_v = SSE_v / (nd-d) # scalar, fixed ols estimate
49
50     # select lambda_v
51     w2 = np.abs(v_tilde_hat)**(-gamma2) # (d, 1)
52     dfs_v = np.sum(np.abs(v_tilde_hat) > lam_grid*w2/2, axis=0) # (S,), df for each lambda
53     part2 = v_tilde_hat - lam_grid*w2/2; part2[part2<0] = 0
54     part1 = np.sign(v_tilde_hat)
55     v_tilde_br = part1 * part2 # (d, S), each column is v_tilde under each lambda
56
57     SSEs_v = list(map(lambda tilde_vec: np.sum((X - get_outer_prod(tilde_vec, u_old, "u"))**2),
58                     v_tilde_br.T)) # (S,)
59     BICs_v = SSEs_v/(nd*sigma2_hat_v) + np.log(nd)/nd*dfs_v # (S,), BIC for each lambda
60     lambda_v = lam_grid[np.argmin(BICs_v)]
61
62     # update v
63     part1 = np.sign(v_tilde_hat)
64     part2 = np.abs(v_tilde_hat)-lambda_v*w2/2; part2[part2<0] = 0
65     if not np.all(part2 == 0): # not full shrinkage at v
66         v_tilde = np.multiply(part1, part2) # (d, 1)
67         v_new = v_tilde / np.linalg.norm(v_tilde)
68
69     ## Update u using current v
70
71     # ols for u, use current v
72     u_tilde_hat = X @ v_new # (n, 1), fixed ols estimate
73     SSE_u = np.sum((X.T - np.outer(v_new, u_tilde_hat))**2)

```

```

73     sigma2_hat_u = SSE_u / (nd-n) # scalar, fixed ols estimate
74
75     # select lambda_u
76     w1 = np.abs(u_tilde_hat)*(-gamma1) # (n, 1)
77     dfs_u = np.sum(np.abs(u_tilde_hat) > lam_grid*w1/2, axis=0) # (S,), df for each lambda
78     part2 = u_tilde_hat - lam_grid*w1/2; part2[part2<0] = 0
79     part1 = np.sign(u_tilde_hat)
80     u_tilde_br = part1 * part2 # (n, S), each column is u_tilde under each lambda
81     SSEs_u = list(map(lambda tilde_vec: np.sum((X - get_outer_prod(tilde_vec, v_new, "v"))**2),
    u_tilde_br.T)) # (S,)
82     BICs_u = SSEs_u/(nd*sigma2_hat_u) + np.log(nd)/nd*dfs_u # (S,), BIC for each lambda
83     lambda_u = lam_grid[np.argmin(BICs_u)]
84
85     # update u
86     part1 = np.sign(u_tilde_hat)
87     part2 = np.abs(u_tilde_hat)-lambda_u*w1/2; part2[part2<0] = 0
88     if not np.all(part2 == 0): # not full shrinkage at u
89         u_tilde = np.multiply(part1, part2) # (n, 1)
90         u_new = u_tilde / np.linalg.norm(u_tilde)
91
92     return u_new, v_new, lambda_u, lambda_v
93
94
95 def SSVD_layer(X, lam_grid, gamma1, gamma2, max_iter=5000, tol=1e-6):
96     """Get the sparse SVD layer given the data matrix X at a SVD layer and the tuning parameters
    grid.
97     Input:
98     X: (n, d), can be the original data matrix or the residual matrix
99     lam_grid: ndarray, a grid of lambdas to be selected
100    gamma1, gamma2: scalar, tuning parameter
101    max_iter: integer, the maximum iteration times
102    tol: float, tolerance to stop the iteration
103    Return: n_iter, u, v, s, lambda_u, lambda_v
104    n_iter: number of iterations
105    u: (n, 1), the final u at convergence
106    v: (d, 1), the final v at convergence
107    s: scalar, the singular value at convergence
108    lambda_u, lambda_v: the optimal tuning parameter at convergence
109    """
110    # SVD
111    U, _, VT = la.svd(X)
112
113    # initial value
114    u_old = U[:,0][:,None]
115    v_old = VT[0][:,None]
116
117    for i in range(max_iter):
118        u_new, v_new, lambda_u, lambda_v = update_uv(u_old, v_old, X, gamma1, gamma2, lam_grid)
119        if np.linalg.norm(u_new-u_old) < tol and np.linalg.norm(v_new-v_old) < tol: # achieve the
    tolerance
120            break
121        if np.all(u_new == 0) or np.all(v_new == 0): # full shrinkage (i.e., all zeros in the
    vector)
122            print("Warning: Full shrinkage has been achieved. Iterations stops. No further
    decomposition. The desired number of layers may not be achieved. ")
123            break
124        u_old, v_old = u_new, v_new
125    n_iter = i+1 # number of iterations
126    u, v = u_new, v_new # the final u and v at convergence
127    s = (u_new.T @ X @ v_new)[0][0]
128    if n_iter == max_iter:
129        print("Warning: The maximum iteration has been achieved. Please consider increasing '
    max_iter'.")
130    return n_iter, u, v, s, lambda_u, lambda_v
131
132 def SSVD(X, num_layer, lam_grid, gamma1, gamma2, max_iter=5000, tol=1e-6):
133     """Get the SSVD given the data matrix X and the desired number of SSVD layers.
134     Input:
135     X: (n, d), the original data matrix

```

```

136     num_layer: desired number of SSVD layers
137     lam_grid: ndarray, a grid of lambdas to be selected
138     gamma1, gamma2: scalar, tuning parameter
139     max_iter: integer, the maximum iteration times
140     tol: float, tolerance to stop the iteration
141     Return: n_iters, us, vs, ss, lambda_us, lambda_vs
142     n_iters: (num_layer,), number of iterations for each layer
143     us: (n, num_layer), the final u at convergence for each layer
144     vs: (d, num_layer), the final v at convergence for each layer
145     ss: (num_layer,), the singular value at convergence for each layer
146     lambda_us, lambda_vs: (num_layer,), the optimal tuning parameter at convergence for each
147     layer
148     """
149     n, d = X.shape
150     n_iters = np.zeros(num_layer, dtype = int)
151     ss = np.zeros(num_layer)
152     lambda_us = np.zeros(num_layer)
153     lambda_vs = np.zeros(num_layer)
154     us = np.zeros((n, num_layer))
155     vs = np.zeros((d, num_layer))
156     # initial value
157     u = np.zeros((n, 1)); v = np.zeros((d, 1)); resi_mat = X; s = 0
158     for i in range(num_layer):
159         resi_mat = resi_mat - s*u@v.T
160         n_iter, u, v, s, lambda_u, lambda_v = SSVD_layer(resi_mat, lam_grid, gamma1, gamma2,
161         max_iter, tol)
162         n_iters[i] = n_iter
163         ss[i] = s
164         lambda_us[i] = lambda_u
165         lambda_vs[i] = lambda_v
166         us[:,i] = u[:,0]
167         vs[:,i] = v[:,0]
168         if np.all(u == 0) or np.all(v == 0): # full shrinkage (i.e., all zeros in the vector)
169             break
170     return n_iters, us, vs, ss, lambda_us, lambda_vs

```

Appendix E Version 5

Final version optimized by Cython based on version 3

```

1  ## This is the final version optimized by Cython.
2  ## The following should be run in Jupyter notebook.
3
4  import cython
5  %load_ext cython
6
7  %%cython
8
9  import numpy as np
10 import scipy.linalg as la
11 import cython
12 from cython.parallel import parallel, prange
13
14 cdef extern from "math.h":
15     double log(double x) nogil
16     double pow(double x, double y) nogil
17     double fabs(double x) nogil
18     double sqrt(double x)
19     double isless(double x, double y) nogil
20     double fmax(double x, double y)
21     double fma(double x, double y, double z) nogil
22
23

```

```

24 cdef double vector_dist_sq(double[:, :] u, double[:, :] v):
25     """Squared Euclidean distance between two vectors. Can also compute the squared norm of a
    vector if set one of the input being zero vector. """
26     cdef int i
27     cdef double s = 0
28     for i in range(u.shape[0]):
29         s += pow(u[i,0] - v[i,0], 2)
30     return s
31
32 @cython.boundscheck(False)
33 @cython.wraparound(False)
34 cdef matrix_multiply(double[:, :] u, double[:, :] v, double[:, :] res, double c = 1):
35     """Matrix multiplication, equivalent to c*u@v."""
36     cdef int i, j, k
37     cdef int m = u.shape[0], n = u.shape[1], p = v.shape[1]
38     with cython.nogil, parallel():
39         for i in prange(m): # parallel
40             for j in prange(p): # parallel
41                 res[i,j] = 0
42                 for k in range(n): # serial
43                     res[i,j] += u[i,k] * v[k,j]
44                 res[i,j] = fma(res[i,j], c, 0)
45
46 @cython.boundscheck(False)
47 @cython.wraparound(False)
48 cdef elementwise_multiply(double[:, :] u, double[:, :] v, double[:, :] res):
49     """Element multiplication of two matrices. """
50     cdef int i, j
51     cdef int m = u.shape[0], n = u.shape[1]
52     with cython.nogil, parallel():
53         for i in prange(m): # parallel
54             for j in prange(n): # parallel
55                 res[i,j] = u[i,j] * v[i,j]
56
57 @cython.boundscheck(False)
58 @cython.wraparound(False)
59 cdef double[:, :] vec_outer_prod(double[:, :] u, double[:, :] v, double c = 1):
60     """Outer product between two vectors, equivalent to c*u@v.T."""
61     cdef int n = u.shape[0], m = v.shape[0]
62     cdef int i, j
63     cdef double[:, :] res = np.zeros((n, m))
64     with cython.nogil, parallel():
65         for i in prange(n): # parallel
66             for j in prange(m): # parallel
67                 res[i,j] = u[i,0] * v[j,0] * c
68     return res
69
70
71 @cython.boundscheck(False)
72 @cython.wraparound(False)
73 cdef double[:, :] get_w(double[:, :] tilde_hat, double gamma):
74     """Get the w vector, equivalent to elementwise |tilde_hat|(-gamma)"""
75     cdef int i, l = tilde_hat.shape[0]
76     cdef double[:, :] w = np.zeros((l, 1))
77     with cython.nogil, parallel():
78         for i in prange(l): # parallel
79             w[i,0] = pow(fabs(tilde_hat[i,0]), -gamma)
80     return w
81
82
83
84 @cython.boundscheck(False)
85 @cython.wraparound(False)
86 cdef double[:, :] get_part2(double[:, :] tilde_hat, double[:, :] w, double lam):
87     """Compute the part2 in the updating formula. """
88     cdef int i, l = tilde_hat.shape[0]
89     cdef double ele
90     cdef double[:, :] part2 = np.zeros((l,1))

```

```

92     with cython.nogil, parallel():
93         for i in prange(1): # parallel
94             ele = fabs(tilde_hat[i,0])-lam*w[i,0]/2
95             if ele > 0:
96                 part2[i,0] = ele
97     return part2
98 cdef int is_full_shinkage(double[:,:] v):
99     """Judge if the input vector if fully shrunk to 0. Return 1 if fully shunk, and 0 otherwise.
100     """
101     cdef int l = v.shape[0]
102     cdef int flag = 1
103
104     for i in range(1):
105         if v[i,0] != 0:
106             flag = 0
107             break
108     return flag # flag = 1 if all elements are 0, = 0 otherwise
109
110 @cython.boundscheck(False)
111 @cython.wraparound(False)
112 cdef get_vec_new(double[:,:] tilde_vec, double[:,:] vec_new):
113     """Get the vec_new according to tilde_vec/norm(tilde_vec). """
114     cdef int i, l = tilde_vec.shape[0]
115     cdef double norm = sqrt(vector_dist_sq(tilde_vec, np.zeros((1, 1))))
116     with cython.nogil, parallel():
117         for i in prange(1): # parallel
118             vec_new[i,0] = tilde_vec[i,0] / norm
119
120
121 cpdef update_uv3c(double[:,:] u_old, X, gamma1, gamma2, lam_grid, double[:,:] u_new, double[:,:]
122     v_new):
123     """Update u and v once given the current u and v."""
124     cdef int n = X.shape[0], d = X.shape[1], S = lam_grid.shape[0]
125     cdef int nd = n*d
126
127     cdef double[:,:] v_tilde_hat = np.zeros((d,1)), v_tilde = np.zeros((d,1))
128     cdef double[:,:] u_tilde_hat = np.zeros((n,1)), u_tilde = np.zeros((n,1))
129
130     cdef double lambda_u = 0, lambda_v = 0
131
132     ## Update v using current u
133
134     # ols for v, use current u
135     matrix_multiply(X.T, u_old, v_tilde_hat) # v_tilde_hat: (d, 1), fixed ols estimate
136     SSE_v = np.sum((X - vec_outer_prod(u_old, v_tilde_hat))*2) # scalar, SSE_v = (Y-Y_hat).T @ (Y
137     -Y_hat)
138     sigma2_hat_v = SSE_v / (nd-d) # scalar, fixed ols estimate
139
140     # select lambda_v
141     w2 = get_w(v_tilde_hat, gamma2) # (d, 1)
142     dfs_v = np.sum(np.abs(v_tilde_hat) > lam_grid*w2/2, axis=0) # (S,), df for each lambda
143     part2 = v_tilde_hat - lam_grid*w2/2; part2[part2<0] = 0
144     part1 = np.sign(v_tilde_hat)
145     v_tilde_br = part1 * part2 # (d, S), each column is v_tilde under each lambda
146     outer_prods = np.outer(u_old, v_tilde_br.T) # (n, d*S), each chunk of size (n, d) is the outer
147     product mat under each lambda
148     outer_prods = np.array(np.hsplit(outer_prods, S)) # (S, n, d)
149     SSEs_v = ((X-outer_prods)**2).sum(axis = (1,2)) # (S,), ||X-uv_tilde^T||_F^2=||Y-Y_hat||^2 for
150     each lambda
151     BICs_v = SSEs_v/(nd*sigma2_hat_v) + np.log(nd)/nd*dfs_v # (S,), BIC for each lambda
152     lambda_v = lam_grid[np.argmin(BICs_v)]
153
154     # update v
155     part1 = np.sign(v_tilde_hat)
156     part2 = get_part2(v_tilde_hat, w2, lambda_v)
157
158     if is_full_shinkage(part2) == 0: # not full shrinkage at v

```



```

156     elementwise_multiply(part1, part2, v_tilde) # v_tilde: (d, 1)
157     get_vec_new(v_tilde, v_new) # v_new: (d, 1)
158
159     ## Update u using current v
160
161     # ols for u, use current v
162     matrix_multiply(X, v_new, u_tilde_hat) # u_tilde_hat: (n, 1), fixed ols estimate
163     SSE_u = np.sum((X.T - vec_outer_prod(v_new, u_tilde_hat))**2)
164     sigma2_hat_u = SSE_u / (nd-n) # scalar, fixed ols estimate
165
166     # select lambda_u
167     w1 = get_w(u_tilde_hat, gamma1) # (n, 1)
168     dfs_u = np.sum(np.abs(u_tilde_hat) > lam_grid*w1/2, axis=0) # (S,), df for each lambda
169     part2 = u_tilde_hat - lam_grid*w1/2; part2[part2<0] = 0
170     part1 = np.sign(u_tilde_hat)
171     u_tilde_br = part1 * part2 # (n, S), each column is u_tilde under each lambda
172     outer_prods = np.outer(u_tilde_br.T, v_new) # (n*S, d), each chunk of size (n, d) is the
outer product mat under each lambda
173     outer_prods = np.array(np.vsplit(outer_prods, S)) # (S, n, d)
174     SSEs_u = ((X-outer_prods)**2).sum(axis = (1,2)) # (S,), ||X-u_tildev^T||_F^2=||Z-Z_hat||^2
for each lambda
175     BICs_u = SSEs_u/(nd*sigma2_hat_u) + np.log(nd)/nd*dfs_u # (S,), BIC for each lambda
176     lambda_u = lam_grid[np.argmin(BICs_u)]
177
178     # update u
179     part1 = np.sign(u_tilde_hat)
180     part2 = get_part2(u_tilde_hat, w1, lambda_u)
181     if is_full_shrinkage(part2) == 0: # not full shrink at u
182         elementwise_multiply(part1, part2, u_tilde) # u_tilde: (n, 1)
183         get_vec_new(u_tilde, u_new) # u_new: (n, 1)
184
185     return lambda_u, lambda_v
186
187
188 cpdef SSVD_layer3c(X, lam_grid, gamma1, gamma2, max_iter=5000, tol=1e-6):
189     """Get the sparse SVD layer given the data matrix X at a SVD layer and the tuning parameters
grid."""
190     cdef int n = X.shape[0], d = X.shape[1]
191     # SVD
192     U, _, VT = la.svd(X)
193
194     # initial value
195     cdef double[:, :] u_old = U[:, 0][:, None], v_old = VT[0][:, None]
196     cdef double[:, :] u_new = np.zeros((n,1)), v_new = np.zeros((d,1))
197     cdef int i # number of iterations
198
199     for i in range(max_iter):
200         lambda_u, lambda_v = update_uv3c(u_old, X, gamma1, gamma2, lam_grid, u_new, v_new) #
update u_new, v_new
201         if isless(fmax(vector_dist_sq(u_new, u_old), vector_dist_sq(v_new, v_old)), pow(tol, 2)):
# achieve the tolerance
202             break
203         if fmax(is_full_shrinkage(u_new), is_full_shrinkage(v_new)): # full shrinkage (i.e., all
zeros in the vector)
204             print("Warning: Full shrinkage has been achieved. Iterations stops. No further
decomposition. The desired number of layers may not be achieved. ")
205             break
206         u_old, v_old = u_new, v_new
207     u, v = u_new, v_new # the final u and v at convergence
208     s = (u.T @ X @ v)[0][0]
209     n_iter = i+1
210     if n_iter == max_iter:
211         print("Warning: The maximum iteration has been achieved. Please consider increasing '
max_iter'")
212     return n_iter, np.array(u), np.array(v), s, lambda_u, lambda_v
213
214
215 cpdef SSVD3c(X, num_layer, lam_grid, gamma1, gamma2, max_iter=5000, tol=1e-6):
216     """Get the SSVD given the data matrix X and the desired number of SSVD layers."""

```

```

217 n, d = X.shape
218 n_iters = np.zeros(num_layer, dtype = int)
219 ss = np.zeros(num_layer)
220 lambda_us = np.zeros(num_layer)
221 lambda_vs = np.zeros(num_layer)
222 us = np.zeros((n, num_layer))
223 vs = np.zeros((d, num_layer))
224 # initial value
225 cdef double[:, :] res = np.zeros((n, d))
226 cdef double s = 0
227 resi_mat = X
228 u = np.zeros((n, 1)); v = np.zeros((d, 1))
229 for i in range(num_layer):
230     resi_mat = resi_mat - s * np.outer(u, v)
231     n_iter, u, v, s, lambda_u, lambda_v = SSVD_layer3c(resi_mat, lam_grid, gamma1, gamma2,
max_iter, tol)
232     n_iters[i] = n_iter
233     ss[i] = s
234     lambda_us[i] = lambda_u
235     lambda_vs[i] = lambda_v
236     us[:, i] = u[:, 0]
237     vs[:, i] = v[:, 0]
238     if np.all(u == 0) or np.all(v == 0): # full shrinkage (i.e., all zeros in the vector)
239         break
240 return n_iters, us, vs, ss, lambda_us, lambda_vs

```