Implementation and Optimization of Biclustering via Sparse Singular Value Decomposition

Cathy Shi and Eric Tay

April 27, 2021

Abstract

Sparse singular value decomposition (SSVD) is an effective algorithm that can be used as a exploratory analysis tool for biclustering or identifying interpretable row–column associations within high-dimensional data matrices. As an unsupervised learning algorithm, SSVD identifies significant row–column associations within high-dimensional data matrices. It first extracts layers from the singular value decomposition (SVD) result and forces each layer's singular vector to be sparse through an iterative algorithm. Hence it achieves desired biclustering effect, which is to simultaneously identify distinctive "checkerboard" patterns in data matrices, or sets of rows and columns in the matrices that are associated. This paper focuses on first implementing the algorithm as it was proposed in the paper *Biclustering via Sparse Singular Value Decomposition* by Mihee Lee, et. al (1) and then its optimization and application to both simulated data and real-world datasets. Code to reproduce the findings in this paper can be found at our GitHub repository.

Keywords: Adaptive lasso; Biclustering; Dimension reduction; Sparse singular value decomposition

1 Background

For this project, we will be implementing, optimizing and discussing the Sparse Singular Value Decomposition (SSVD) algorithm introduced in (1). We shall first introduce the concept of the algorithm as described in (1). Let $\mathbf{X}$ be a $n \times d$ data matrix, whose rows represent samples and columns represent variables. The singular value decomposition (SVD) of $\mathbf{X}$ can be written as

$$\mathbf{X} = \mathbf{UDV}^T = \sum_{k=1}^{r} \mathbf{s}_k \mathbf{u}_k \mathbf{v}_k^T,$$

where r is the rank of $\mathbf{X}$, $\mathbf{U} = (\mathbf{u}_1, \ldots, \mathbf{u}_r)$ is a matrix of orthonormal left singular vectors, $\mathbf{V} = (\mathbf{v}_1, \ldots, \mathbf{v}_r)$ is a matrix of orthonormal right singular vectors, $\mathbf{D} = \text{diag}(s_1, \ldots, s_r)$ is a diagonal matrix with positive singular values $s_1 \geq \cdots \geq s_r$ on its diagonal. In other words, SVD decomposes $\mathbf{X}$ into a summation of rank one matrices $s_k \mathbf{u}_k \mathbf{v}_k^T$, which we define as the k-th SVD layer. In addition, if we take the first $K \leq r$ rank-one matrices in the summation, we obtain the following rank-K approximation to $\mathbf{X}$:

$$\mathbf{X} \sim \mathbf{X}^{(K)} = \sum_{k=1}^{r} \mathbf{s}_k \mathbf{u}_k \mathbf{v}_k^T.$$

The proposed SSVD seeks a low-rank approximation to $\mathbf{X}$ as in $\mathbf{X}^{(K)}$, but with the requirement that $\mathbf{u}_k$ and $\mathbf{v}_k$ are sparse. The sparsity property implies that each SSVD layer has a checkerboard structure, which makes the algorithm suitable for biclustering. Specifically, for the k-th SSVD layer, the rows (or samples) with nonzero u_{ki}s are naturally clustered together, as well as those columns (or variables) with nonzero v_{kj}s . Hence, the k-th layer simultaneously links sets of samples and sets of variables together to reveal some desirable sample-variable associations.

Biclustering has a number of applications in biomedicine, text mining, marketing, and dimensionality reduction (2), and (1) also provides concrete examples of how the algorithm

can be used in identifying groups of coregulated genes for different cancer types, or for clustering of different subgroups of foods based on nutrient content. Compared to other algorithms like SVD, Plaid (3), and RoBiC (4), SSVD has shown to have significantly more success in detecting sparsity in both $\mathbf{X}$ (when noise is added) and in its singular vectors, when applied to simulations proposed in (1). When applied to microarray gene expression data, SSVD also gave more meaningful biclusters and better low-rank approximations of the data than Plaid or RoBiC (1). However, a disadvantage of the algorithm is that it is more computationally intensive than SVD, and may therefore only be more suitable when the data does indeed have an underlying structure of sparse singular vectors. Another disadvantage is that the algorithm does not have a guarantee of convergence, which we have experienced especially when extracting more SSVD layers.

2 Algorithm

The algorithm focuses on extracting the first SSVD layer ($\mathbf{u}, \mathbf{v}$ and s), extracting subsequent layers sequentially from the residual matrices after removing the preceding layers (1). In extracting each layer, the algorithm introduces an adaptive lasso penalty to the traditional rank-1 SVD decomposition, which encourages sparsity in the singular vectors it extracts. To obtain the singular vectors that minimize the objective, the authors of (1) then propose an efficient iterative algorithm that alternately updates $\mathbf{u}$ and $\mathbf{v}$ until convergence.

In greater detail, the algorithm seeks to minimize

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

where, following (5), $\mathbf{w}_1 = |\mathbf{X}\mathbf{v}|^{-\gamma_1}, \mathbf{w}_2 = |\mathbf{X}^T\mathbf{u}|^{-\gamma_2}$, where γ_1 and γ_2 can be chosen, or selected using the BIC criterion, and λ_u and λ_v are selected at each step using the BIC

criterion. To estimate the optimal solution, the authors propose the following algorithm (for fixed $\gamma_1 = \gamma_2 = 2$).

The SSVD Algorithm

- For i in $1, 2, \ldots,$ *num_layers_to_extract*,
  - Apply the SVD to $\mathbf{X}$ and extract $\mathbf{u}_{old}, s_{old}, \mathbf{v}_{old}$, which correspond to the largest singular value.
  - Define $\mathbf{Y} = (\mathbf{x}_1^T, \ldots, \mathbf{x}_d^T) \in \mathbb{R}^{nd}$ with $\mathbf{x}_j$ being the j-th column of $\mathbf{X}$.
  - Define $\mathbf{Z} = (\mathbf{x}_{(1)}, \ldots, \mathbf{x}_{(n)})^T \in \mathbb{R}^{nd}$ with $\mathbf{x}_{(i)}^T$ being the i-th row of $\mathbf{X}$.
  - While True:
    - Set $\mathbf{w}_2 = |\mathbf{X}^T \mathbf{u}_{old}|^{-\gamma_2}$.
    - For each candidate λ_v:
      - Set $\hat{df}(\lambda_v)$ as the number of $(\mathbf{X}^T \mathbf{u}_{old})_j$s that are bigger than $\lambda_v w_{2,j}/2$.
      - Set $\tilde{v}_j = sign\{(\mathbf{X}^T \mathbf{u}_{old})_j\}(|(\mathbf{X}^T \mathbf{u}_{old})_j| \lambda_v w_{2,j}/2)_+, \ j = 1, ..., d.$
      - Set $\hat{\mathbf{Y}} = (\mathbf{I}_d \otimes \mathbf{u}_{old})\tilde{\mathbf{v}}$, where $\otimes$ denotes the Kronecker product.
      - Set $\hat{\sigma}$ to be the OLS estimate of error variance of $||\mathbf{Y} - (\mathbf{I}_d \otimes \mathbf{u}_{old})\tilde{\mathbf{v}}||^2$, whereby the OLS estimate of $\tilde{\mathbf{v}} = \mathbf{X}^T \mathbf{u}_{old}$.
      - Compute $\text{BIC}(\lambda_v) = \frac{||\mathbf{Y} - \hat{\mathbf{Y}}||^2}{nd \cdot \hat{\sigma}^2} + \frac{\log(nd)}{nd} \hat{df}(\lambda_v)$.
    - Select λ_v as the minimizer of $\text{BIC}(\lambda_v)$, and define $\tilde{\mathbf{v}}$ accordingly.
    - Set $\mathbf{v}_{new} = \tilde{\mathbf{v}}/||\tilde{\mathbf{v}}||_2$.
    - Set $\mathbf{w}_1 = |\mathbf{X}\mathbf{v}_{new}|^{-\gamma_1}$.
    - For each candidate λ_u:
      - Set $\hat{df}(\lambda_u)$ as the number of $(\mathbf{X}\mathbf{v}_{new})_i$s that are bigger than $\lambda_u w_{1,i}/2$.

· Set $\tilde{u}_i = sign\{(\mathbf{X}\mathbf{v}_{new})_i\}(|(\mathbf{X}\mathbf{v}_{new})_i| - \lambda_u w_{1,i}/2)_+,\ i = 1, ..., n.$

· Set $\hat{\mathbf{Z}} = (\mathbf{I}_n \otimes \mathbf{v}_{new})\tilde{\mathbf{u}}$.

· Set $\hat{\sigma}$ to be the OLS estimate of error variance of $||\mathbf{Z} - (\mathbf{I}_n \otimes \mathbf{v}_{new})\tilde{\mathbf{u}}||^2$, whereby the OLS estimate of $\tilde{\mathbf{u}} = \mathbf{X}\mathbf{v}_{new}$.

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

∗ Select λ_u as the minimizer of BIC(λ_u), and define $\tilde{\mathbf{u}}$ accordingly.

∗ Set $\mathbf{u}_{new} = \tilde{\mathbf{u}}/||\tilde{\mathbf{u}}||_2$.

∗ $\mathbf{u}_{old} = \mathbf{u}_{new}, \mathbf{v}_{old} = \mathbf{v}_{new}$.

∗ Exit loop at convergence

– Set $\mathbf{u}_i = \mathbf{u}_{new}, \mathbf{v}_i = \mathbf{v}_{new}, s_i = \mathbf{u}_{new}^T \mathbf{X}\mathbf{v}_{new}$.

– Set $\mathbf{X} = \mathbf{X} - s_i \mathbf{u}_i \mathbf{v}_i^T$

3 Performance optimization

In this section, we first implemented the algorithm as it was written above. Then we profiled the code and used JIT (just-in-time) compilation.

3.1 Original Implementation

We first implement SSVD as per Section 2, with the following additional details:

- γ_1 and γ_2 were set to be 2 as in (1).
- Since (1) does not specify how to select a set of candidate λ_us and λ_vs, if these candidates are not passed in as a parameter, we simply choose λ_us and λ_vs that would correspond exactly to different values of $\hat{df}(\lambda_u)$ and $\hat{df}(\lambda_v)$.

- Since (1) does not specify how to check for convergence, we define convergence to be when $||\mathbf{u}_{new} - \mathbf{u}_{old}||_2$ and $||\mathbf{v}_{new} - \mathbf{v}_{old}||_2$ are smaller than a tolerance value of $1e-3$.

In our implementation, we also create two helper functions, `BIC_v` and `BIC_u`, which will be referenced in later sections.

3.2 Optimization with `sparsesvd`

Given that the SSVD algorithm only requires the first singular vectors for the initialization of each layer, we use `sparsesvd` instead of `numpy.linalg.svd`.

3.3 Profiling

The profiling result is shown below. The algorithm itself consists of two functions and other function calls are of build-in functions. We optimized the two functions we wrote that are embedded in the main algorithm. Although both functions perform decently well, we chose to optimize them separately with JIT compiler in the next subsection.

3.4 Optimization with Numba

Numba is a JIT compiler that can translate Python code into fast machine code. Since Numba is designed to be used with NumPy arrays and functions, we chose to this JIT compiler to optimize the algorithm due to the NumPy arrays the algorithm uses (6). We first used JIT decorator on `BIC_u` and `BIC_v` separately to speed them up. We then tried different options of "numba.jit" decorator to find get optimal version in terms of timing.

The "numba.jit" decorator triggers generation and execution of compiled code when the function is first called. We did a comparison of using different Numba decorator's options.

Since Numba internally converts code to native code and we chose the option of "nopython=True", the algorithm's syntax is adapted to Numba's environment. The "cache=True"

```
         1256173 function calls (1203913 primitive calls) in 11.290 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1340    0.006    0.000    0.133    0.000 <__array_function__ internals>:2(amin)
      670    0.003    0.000    0.075    0.000 <__array_function__ internals>:2(argmin)
    52260    0.262    0.000    0.731    0.000 <__array_function__ internals>:2(dot)
    52260    0.338    0.000    3.210    0.000 <__array_function__ internals>:2(norm)
    50250    0.239    0.000    2.280    0.000 <__array_function__ internals>:2(sum)
      100    0.001    0.000    0.834    0.008 <__array_function__ internals>:2(svd)
    33500    2.691    0.000    6.568    0.000 <ipython-input-53-d2672f629fac>:11(BIC_u)
      100    0.327    0.003   11.283    0.113 <ipython-input-53-d2672f629fac>:3(ssvd)
    16750    1.547    0.000    3.222    0.000 <ipython-input-53-d2672f629fac>:4(BIC_v)
        1    0.007    0.007   11.290   11.290 <string>:1(<module>)
    53030    0.168    0.000    0.291    0.000 _asarray.py:16(asarray)
      670    0.001    0.000    0.001    0.000 fromnumeric.py:1189(_argmin_dispatcher)
      670    0.002    0.000    0.069    0.000 fromnumeric.py:1193(argmin)
    50250    0.061    0.000    0.061    0.000 fromnumeric.py:2087(_sum_dispatcher)
    50250    0.314    0.000    1.769    0.000 fromnumeric.py:2092(sum)
     1340    0.015    0.000    0.015    0.000 fromnumeric.py:2671(_amin_dispatcher)
     1340    0.005    0.000    0.109    0.000 fromnumeric.py:2676(amin)
      670    0.004    0.000    0.017    0.000 fromnumeric.py:42(_wrapit)
      670    0.006    0.000    0.067    0.000 fromnumeric.py:55(_wrapfunc)
    51590    0.585    0.000    1.438    0.000 fromnumeric.py:73(_wrapreduction)
    51590    0.124    0.000    0.124    0.000 fromnumeric.py:74(<dictcomp>)
      100    0.000    0.000    0.000    0.000 linalg.py:111(get_linalg_error_extobj)
      100    0.000    0.000    0.001    0.000 linalg.py:116(_makearray)
    52460    0.196    0.000    0.265    0.000 linalg.py:121(isComplexType)
      200    0.001    0.000    0.001    0.000 linalg.py:134(_realType)
      100    0.001    0.000    0.002    0.000 linalg.py:144(_commonType)
      100    0.000    0.000    0.000    0.000 linalg.py:1454(_svd_dispatcher)
      100    0.828    0.008    0.833    0.008 linalg.py:1458(svd)
      100    0.000    0.000    0.000    0.000 linalg.py:203(_assert_stacked_2d)
    52260    0.060    0.000    0.060    0.000 linalg.py:2312(_norm_dispatcher)
    52260    0.987    0.000    2.572    0.000 linalg.py:2316(norm)
    52260    0.091    0.000    0.091    0.000 multiarray.py:707(dot)
    50920    0.182    0.000    0.182    0.000 {built-in method builtins.abs}
     1340    0.003    0.000    0.003    0.000 {built-in method builtins.all}
        1    0.000    0.000   11.290   11.290 {built-in method builtins.exec}
```

option allows the timing to exclude the compilation time, so the timing's standard deviation of overall algorithm is large. The "nogil=True" option allows Numba to release Python's global interpreter lock (GIL) because GIL is no longer necessary when Numba doesn't run the Python code on Python objects (7). We also considered using "parallel=True" option, but it is not suitable for our main function nor the two penalization functions `BIC_v` and `BIC_u`. For `BIC_v` and `BIC_u`, the overhead cost in multi-threading makes the functions to run slower for small arrays, so we chose to not add this option. It also does not work well with cross iteration dependencies, which is the case with our main function on two levels: first, the function as a whole updates $\mathbf{u}$ and $\mathbf{v}$ in iterations before convergence of the final results; second, race conditions would occur in some places where the multiple parallel threads simultaneously update the same memory address.

We also tried writing functions and optimize them with other packages, such as `cython` and `sparsesvd` to use in this Numba-optimized version, but due the limited available libraries that "numba.jit" under "nopython=True" supports, we cannot use them.

Hence, we finally decided to use Numba with options "nopython=True, nogil=True, cache=True" as the final version for optimization.

4 Application to simulated data sets

We apply our SSVD algorithm to 2 simulation studies described in (1).

4.1 Simulation 1

The first simulation study considers a rank-1 true signal matrix $\mathbf{X}^* = s\mathbf{u}\mathbf{v}^T, s = 50, \tilde{\mathbf{u}} = [10, 9, 8, 7, 6, 5, 4, 3, r(2, 17), r(0, 75)]^T, \mathbf{u} = \tilde{\mathbf{u}}/||\tilde{\mathbf{u}}||, \tilde{\mathbf{v}} = [10, 10, 8, 8, 5, 5, r(3, 5), r(3, 5), r(0, 34)]^T$, $\mathbf{v} = \tilde{\mathbf{v}}/||\tilde{\mathbf{v}}||$. A data matrix $\mathbf{X}$ is then generated as the sum of $\mathbf{X}^*$ and a noise matrix, whose elements are randomly sampled from the standard normal distribution. The simulation is carried out 100 times, and for the SSVD algorithm, SVD algorithm, and Spectral Biclustering (SB) algorithm (8), we track the number of correctly identified zeros in $\mathbf{u}$ and $\mathbf{v}$, and report the misclassification rate. Given that SB only clusters the entries of $\mathbf{u}$ and $\mathbf{v}$, we explore all permutations of labels and report the lowest misclassification rate in Table 1. We also time the different variants of our algorithm and report the improvement in performance in Table 2.

4.2 Simulation 2

The second simulation study considers a 50×100 true signal matrix $\mathbf{X}^*$ whose elements are given by $X^*_{i,j} = T_{i,j}\mathbb{1}_{\{|T_{i,j}|>1\}}$, where

Algorithm	Singular vector	Misclassification rate (%)
SSVD	**u**	1.27
SSVD	**v**	0.28
SC	**u**	47.11
SC	**v**	37.74
SVD	**u**	75.00
SVD	**v**	68.00

Table 1: *Comparison of performance among SSVD, SVD and SB.*

Algorithm	Runtime	Multiplicative speed-up (3 s.f.)
Original	2.87 s ± 227 ms	1
Original + sparsesvd	2.49 s ± 142 ms	1.15
Numba	711 ms ± 44.5 ms	4.04

Table 2: *Comparison of performance among different variants of SSVD.*

$$T_{i,j} = \begin{cases} \frac{24^2-(i-25)^2-(j-50)^2}{100}, & \text{if } 26 \leq j \leq 75, \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

This setup is more complicated because $\mathbf{X}^*$ no longer has a multiplicative structure as assumed by SSVD, nor an additive structure. Again, a data matrix $\mathbf{X}$ is then generated as the sum of $\mathbf{X}^*$ and a noise matrix, whose elements are randomly sampled from the standard normal distribution. The simulation is carried out 100 times, and for the SSVD algorithm, SVD algorithm, and SB algorithm, we track the number of correctly identified zeros in $\mathbf{X}$ across an increasing number of singular vectors extracted/considered, and plot the classification accuracy in Figure 1. Given that SB only clusters the entries of $\mathbf{u}$ and $\mathbf{v}$, we explore all permutations of labels and plot the highest classification accuracy.

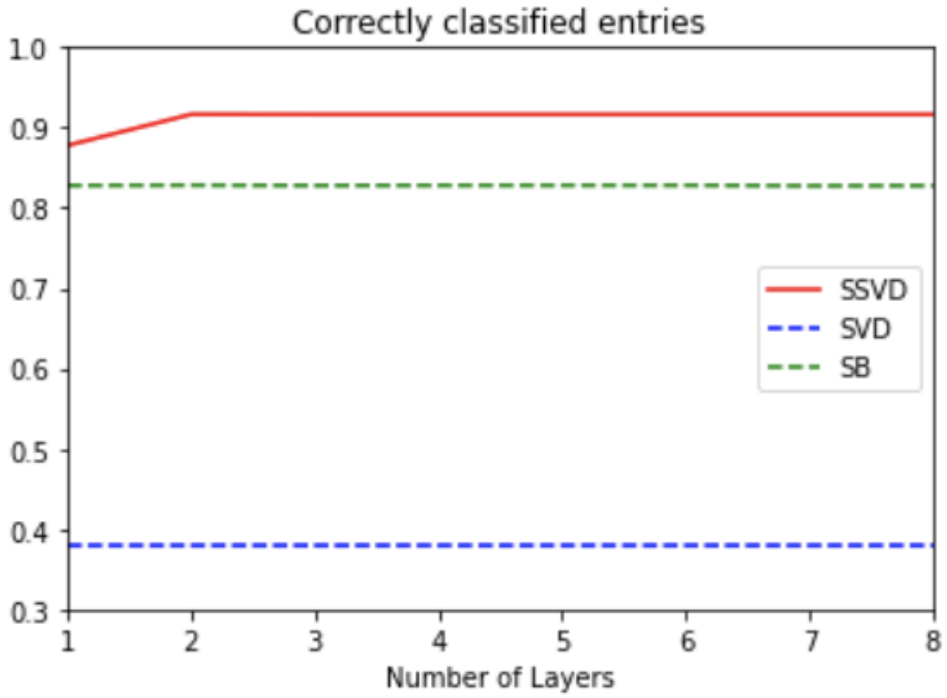

Figure 1: *Percentage of correctly classified entries for each method, across the number of layers extracted/considered.*

5 Application to real data sets

5.1 Lung Cancer Data

We test the algorithm using the microarray gene expression data used in the paper (9), which we obtain from (10). The data consists of expression levels of 12,625 genes, measured from 56 subjects. These subjects are known to be either normal subjects (Normal) or patients with one of the following three types of cancer: pulmonary carcinoid tumors (Carcinoid), colon metastases (Colon), and Small cell carcinoma (SmallCell). The data can be viewed as a $56 \times 12{,}625$ matrix (X), whose rows represent the subjects, grouped together according to the cancer type, and the columns correspond to the genes. The goal is to then identify sets of biologically relevant genes that are significantly expressed for certain cancer

types (1).

As per (1), we extract the first 3 SSVD layers, since they have much higher singular values. We also reproduce a similar plot of each SSVD layer in Figure 2. As done in (1), to better visualize the gene grouping, the columns of the k-th layer are rearranged based on an ascending ordering of the entries of $\mathbf{v}_k$. The dotted horizontal lines in each panel reveal the four cancer types of the subjects. The white vertical area corresponds to those zeroed-out genes, which reveals the effect of the sparsity regularization. (In each panel, 8000 zeroed-out genes are excluded when plotting, and the boundaries of the white areas are indicated.)

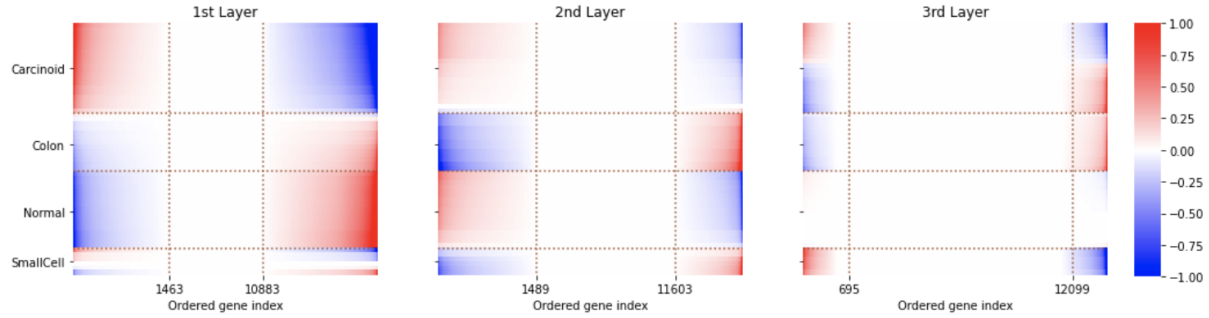

Figure 2: *Image plots of the first 3 SSVD layers.*

We also similarly plot scatterplots of the entries of the first three left sparse singular vectors $\mathbf{u}_k$ in Figure 3.

Finally, we measure the time to extract the first SSVD layer with each variant of our algorithm, and report the results in Table 3. We only extract the first layer given that the dataset is large and the time for the algorithm to run is extensive.

Algorithm	CPU user time	CPU sys time	CPU total time	Wall time
Original	10min 19s	2min 41s	13min 1s	7min 7s
Original + sparsesvd	10min 21s	2min 33s	12min 54s	7min 39s
Numba	9min 26s	2min 15s	11min 42s	6min 46s

Table 3: *Comparison of performance among different variants of SSVD on Lung Cancer Data.*

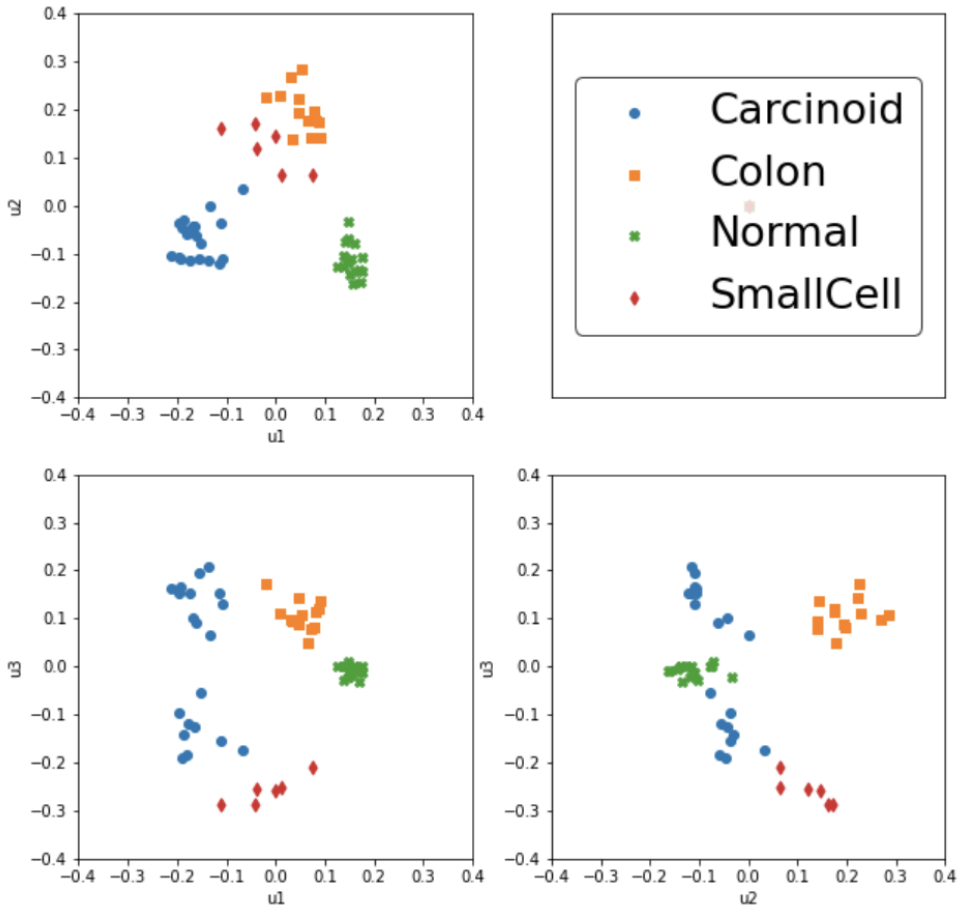

Figure 3: *Scatterplots of the entries of the first three left sparse singular vectors* $\mathbf{u}_k$.

5.1.1 Interpretation of Results

First, we note that the SSVD performs gene selection claimed in (1). With reference to Figure 2, we note that the number of genes selected in each layer is much less than the total number of 12,625, due to the sparsity constraint placed on the $\mathbf{v}_k$s.

Next, we note from the checkerboard structure of Figure 2, that the SSVD algorithm has indeed been able to uncover significant differences in gene expression between different sample subgroups, even though the algorithm is unsupervised. For example, in the first layer, there is a significant difference in gene expression for first 1463 genes and last 1742

genes, between the Normal and Carcinoid group. The SSVD algorithm is therefore useful in uncovering associations between expression in certain genes and cancer.

Furthermore, Figure 3 also demonstrates that differences between subgroups are reflected in the entries of $\mathbf{u}_k$, showing that the SSVD algorithm is indeed effective in clustering, and could aid in identifying the propensity of one contracting cancer. Interestingly, Figure 3 also shows 2 different clusters of those in the Carcinoid group, which could be an interesting direction for biological research.

5.2 Company Financial Fraud Dataset

The company financial fraud dataset (11) is a dataset consisting of 13 companies and 35 financial indices. The financial indices are indices summarized from both academic papers and industry experiences that are effective in evaluating whether a company has made fraudulent report about their financial situations to maintain a relatively satisfactory stock price level or avoid delisting warnings. The value assigned to each company under each index is a percentage calculated by comparing the companies' actual report with many other companies. Out of the 13 companies, 7 of them had fraud reports and 6 of them didn't. The data we used was scaled to have values between 0 and 1 as the companies' score for each index, and independently, we found that when the values were the closer to 0 or 1, the more likely the company had fraudulent report. To aid SSVD performance, we transformed each entry in the data with the transformation $f(x) = |cos(180x)|$, such that in general, larger values would correspond to higher probability of fraud. In Section 5.1, we also create image plots of the first 3 SSVD layers and scatterplots of the entries of the first three left sparse singular vectors $\mathbf{u}_k$ in Figures 4 and 5 respectively.

Additionally, we time how long it takes for each variant of our algorithm to extract all 3 SSVD layers in Table 4.

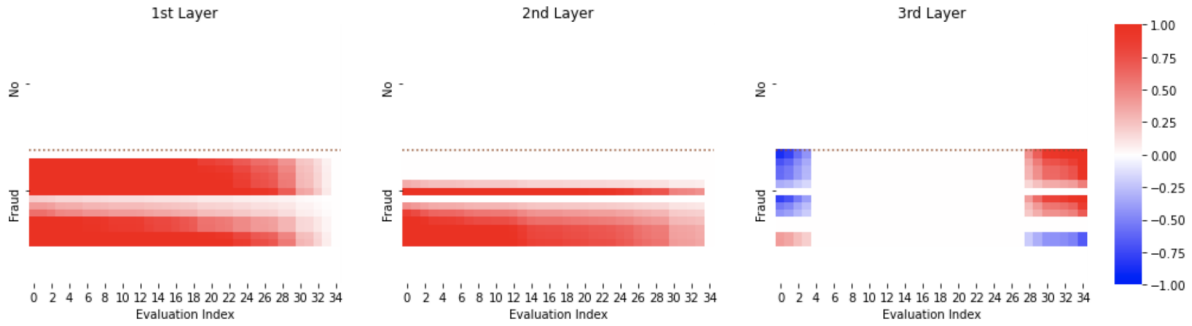

Figure 4: *Image plots of the first 3 SSVD layers.*

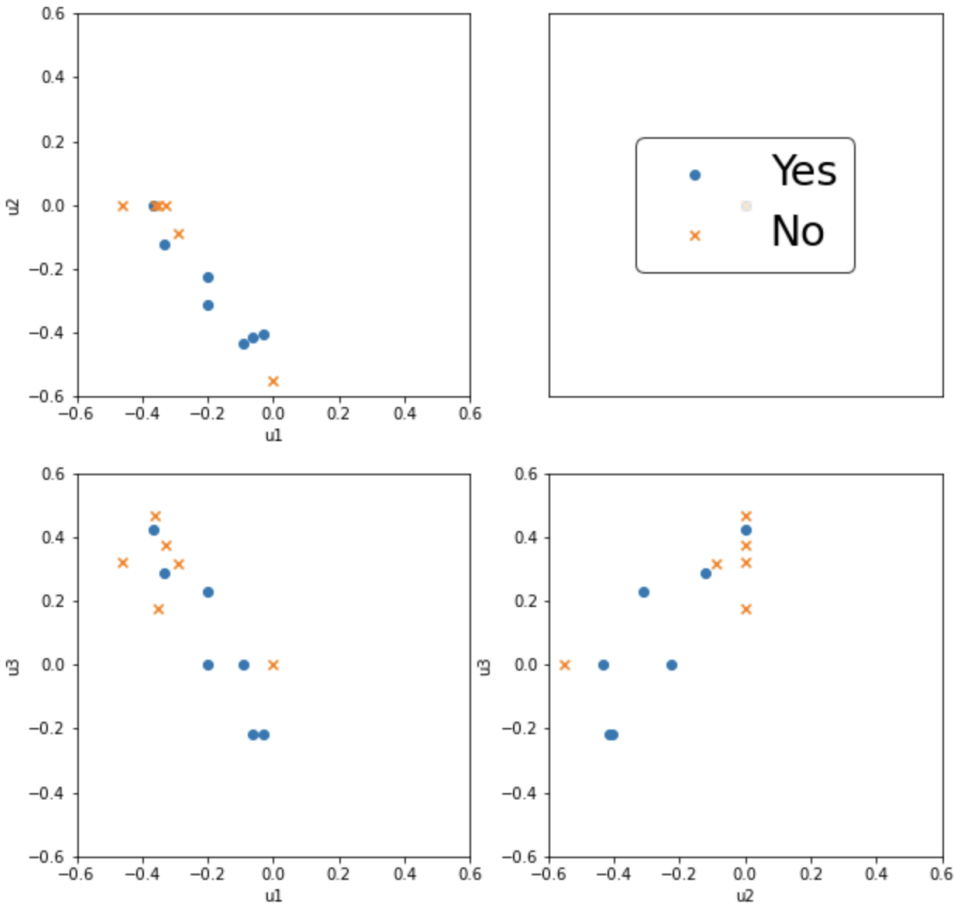

Figure 5: *Scatterplots of the entries of the first three left sparse singular vectors* $\mathbf{u}_k$.

Algorithm	Runtime	Multiplicative speed-up (3 s.f.)
Original	38.6ms $\pm$ 513μs	1
Original + sparsesvd	40.8ms $\pm$ 2.16μs	0.946
Numba	4.79 ms $\pm$ 89.1μs	8.058

Table 4: *Comparison of performance among different variants of SSVD on Company Financial Fraud Data.*

5.2.1 Interpretation of Results

In Figure 4, we see that the algorithm is able to very effectively uncover fraud overall, with significant difference between fraudulent and non-fraudulent companies uncovered in layers 1 and 2. In the scatterplots of $\mathbf{u}_3$ in Figure 5, that the SSVD algorithm was also had some success in clustering fraudulent companies together, although this clustering was not as effective as in the lung cancer data since there are some overlapping of the two groups, and we suspect that this is because the data matrix is much smaller (dimension of 13*35) comparing to the lung cancer one (dimension of 56*12625).

6 Comparative analysis with competing algorithms

With reference to the results in Sections 4.1 and 4.2, we note that the SSVD algorithm outperforms both the SVD and SB algorithm in terms of accuracy. Unsurprisingly, the SVD algorithm does poorly on both simulations, given that it has no assumption of sparsity. In comparison the SB algorithm does better, but the SSVD algorithm showed the greatest success in detecting sparsity in both the underlying components (Section 4.1), and the data itself (Section 4.2). This is a reflection of the successful adaption of the SVD for the detection of sparsity.

7 Usable Package

This algorithm is uploaded to PyPI's platform and can be installed via the following command:

```
pip install -i https://test.pypi.org/simple/ SSVD-pkg-cathy10
from SSVD_pkg import algorithms
```

Both the original version and the optimized version via Numba can be called using `algorithms.ssvd_original` and `algorithms.ssvd_new`, respectively by passing in the matrix of interest and get the values of decomposition u, v, s of one SSVD layer. For more detailed descriptions, visit our package on PyPI.

8 Discussion

8.1 Efficacy of Algorithm

Overall, this algorithm is very useful in identifying interpretable row–column associations within high-dimensional data matrices. SSVD addresses the overfitting problem in the SVD by forcing the singular vectors $\mathbf{u}_k$ and $\mathbf{v}_k$ to be sparse, and the iterative process of computing those sparse singular vectors achieves biclustering at an accuracy that outperforms other similar algorithms with its significantly lower misclassification rate, on both simulation 1 and 2, even when the data matrix used in simulation 2 does not have the multiplicative/additive structure assumed by the SSVD algorithm. As mentioned previously, SSVD is also an efficient exploratory analysis tool, which can be widely applied in many fields, such as text mining, marketing, dimensionality reduction, etc., and as our applications on real datasets concretely show, in biomedicine and fraud detection.

Nevertheless, there are a few limitations of the SSVD algorithm which the original

paper did not mention. As mentioned before, it is more computationally intensive than the traditional SVD, and there is no guarantee of convergence. In addition, as an unsupervised learning algorithm, in order for the algorithm to effectively perform biclustering, the input data matrix has to be relatively large. Moreover, we suspect that SSVD is less effective in biclustering data whereby extreme values should be clustered together. Prior to our data transformation, the SSVD algorithm did not seem to observe noticeable differences between the fraudulent and non-fraudulent companies (up to the 3rd SSVD layer). However, after we applied our transformation, which led extreme values to be larger than moderate values, the SSVD algorithm was significantly more successful. The need to cluster extreme values together is a situation that occurs often in real life, and a direction for future work could therefore be to improve the SSVD algorithm to handle data that is more complicated than those presented in (1).

8.2 Optimization Results

With the optimizations described in Section 3 and their application of both simulations (Section 4) and real world data (Section 5), we are able to get satisfactory results for our final optimization algorithm. Comparing to the original SSVD, the overall performance of the algorithm utilizing `sparsesvd` is comparable to the original one, but the Numba version can achieve a multiplicative speed-up ranging from 4 folds to more than 8 folds depending on the dataset used, with the most pronounced difference occurring when the dataset is not extremely large (as in the lung cancer data).

9 Conclusion

The SSVD algorithm is effective in biclustering, with the best performance on the simulations we experimented with among competing algorithms. On our applications on

real-world datasets, the algorithm is also able to successfully identify significant row–column associations within high-dimensional data matrices. In our paper, we implemented this algorithm and optimized its performance to reduce computational speed mainly via the JIT compilation through Numba, which is ultimately able to speed up the computation by 4 to 8 folds. To aid the use of this algorithm, we have developed the SSVD package via PyPI platform, which contains the original implementation of SSVD, along with the optimizations we have proposed in this paper. For future steps, we could improve this algorithm to be able to take matrices with more complicated data structures as mentioned in Section 8.

References

[1] Lee et al. (2010). Biclustering via Sparse Singular Value Decomposition. *Biometrics*, 66(1), 1087-1095.

[2] Busygin et al. (2010). Biclustering in data mining. *Computers & Operations Research*, 35, 2964-2987.

[3] L. Lazzeroni and A. Owen (2002). Plaid models for gene expression data. *Statistica Sinica*, 12, 61-86.

[4] N. Asgarian and R. Greiner (2007). Using Rank-One Biclusters to Classify Microarray Data. *Using Rank-One Biclusters to Classify Microarray Data*, 00(00), 1-7.

[5] H. Zou (2006). The adaptive lasso and its oracle properties. *Journal of the American Statistical Association*, 101, 1418–1429.

[6] Anaconda (2017). Parallel Python with Numba and ParallelAccelerator. Retrieved from https://www.anaconda.com/blog/parallel-python-with-numba-and-parallelaccelerator.

[7] Anaconda (n.d.). Automatic Parallelization with @jit. Retrieved from https://numba.pydata.org/numba-doc/dev/user/parallel.html.

[8] Y. Kluger et al. (2003). Spectral Biclustering of Microarray Cancer Data: Co-clustering Genes and Conditions. *Genome Research*, 13, 703–716.

[9] A. Bhattacharjee et al. (2001). Classification of human lung carcinomas by mRNA expression profiling reveals distinct adenocarcinoma subclasses. *Proceedings of the National Academy of Sciences of the United States of America*, 98(24), 13790-13795.

[10] H. Sun (n.d.). AutoDecoder, a Noise-Resistant Bicluster Recognition Model. Retrieved from http://grafia.cs.ucsb.edu/autodecoder/dataset.html.

[11] WindDB (n.d.). Retrieved from https://www.wind.com.cn/NewSite/edb.html

10 Appendix

We list the authors and their respective contributions below:

- Cathy Shi
  - Optimized the algorithm with Numba and produced the package for the SSVD algorithm.
  - Sourced and transformed the Company Financial Fraud dataset, and applied the SSVD algorithm to it.
  - Wrote the final report.
- Eric Tay
  - Reproduced the original algorithm, and implemented the `sparsesvd` optimization.
  - Applied the algorithm and other comparative algorithms on Simulation 1, Simulation 2 and the Lung Cancer Dataset.
  - Wrote the final report.