STA 663 Final Project
Sparse Singular Value Decomposition for Biclustering

Kai Wang, Yi Zhu

May 2, 2018

Abstract

Sparse singular value decomposition(SSVD) has been considered as a new exploratory analysis tool for biclustering. SSVD identifies row-column associations of data matrices by seeking low-rank and "checkerboard" structured matrix approximations to data matrices. Such matrix approximations are achieved by introducing sparsity to the left and right matrices of singular vectors after decomposition. In this paper, We explain the mathematical basis of and implement the SSVD algorithm for biclustering proposed by Lee et al.(2010)[1] in paper *Biclustering via Sparse Singular Value Decomposition* in Python. The algorithm is then optimized using Just-In-Time Compilation and Parallel Processing. Furthermore, the validity of the algorithm is examined by applying the algorithm to lung cancer data, MNIST database and simulated data while the performance of the algorithm is tested with these data sets.
KEY WORDS: Sparse singular value decomposition; Biclustering; Optimization; Adaptive LASSO; Spectral Biclustering.

1 Background

In real life, high dimensionality has gradually become a common feature for data in a variety of applications. Data sets with more variables than observations are now very important in many fields. For example, data sets in genetics, medical imaging and text recognition usually have much higher dimension than the sample size. Such type of data sets is described by "High Dimension Low Sample Size(HDLSS)". It is perceived that, with the development of technology, HDLSS data would be more often encountered in future applications. While classical multivariate analysis fails in situations involving HDLSS data, unsupervised learning tools, such as biclustering, become important to handle such data in terms of finding interpretable structures of the data sets.

Biclustering refers to a class of algorithms or learning tools which allow simultaneous clustering of rows and columns of data matrices and enable identification of "checkerboard" patterns in matrices. Biclustering has become so important and popular that many algorithms have been proposed, such as Iterative Signature Algorithm(ISA), Order Preserving Submatrix Algorithm (OPSM) and Spectral Biclustering. In the paper *Biclustering via Sparse Singular Value Decomposition*, Lee et al. (2010)[1] propose SSVD as a new tool for biclusering. In terms of application, the author uses SSVD to microarray gene data to identify sets of genes which are significantly expressed for certain type of cancers.

In terms of advantages and disadvantages, Lee et al.(2010)[1] mentions that one advantage of SSVD over Sparse Principal Component Analysis(SPCA) is that SSVD discovers block structures in data matrices. However, in the paper *Sparse Biclustering of Transposable Data*, Tan,Kean Ming, and Daniela M. Witten.(2014)[2] point out that SSVD would only return a series of sparse singular vectors instead of explicit clusters labels for rows and columns of the data matrix. Thus, a *post hoc* approach must be taken in order to interpret the singular vectors.

Before proceeding to the algorithm of biclusering via SSVD, we first introduce the concept of Singular Value Decomposition(SVD). Let X be a $n \times d$ data matrix where n is number of samples while d is number of variables. The SVD of X can then be written as:

$$X = UDV^T = \sum_{k=1}^{r} s_k u_k t_k^T$$

where r is the rank of X, $U = (u_1, u_2, ..., u_r)$ is a matrix of orthonormal left singular vectors, $V = (v_1, v_2, ..., v_r)$ is a matrix of orthonormal right singular vectors, $D = \text{diag}(s_1, s_2, ..., s_r)$ is a diagonal matrix with positive singular values in descending order($s_1 > s_2 > ... > s_r$). SVD decomposes a matrix into a summation of $s_k u_k t_k^T$, each of which is called a layer. While the values of the first K singular values are significant while the other singular values are close to 0 or not significant compared to the first K values, we can take the sum of the first K layers to approximate the original matrix. The obtained matrix:

$$X \approx X^{(K)} = \sum_{k=1}^{K} s_k u_k v_k^T$$

is called a rank-K approximation to X and in fact is the closet rank-K approximation to X in terms of minimizing the squared Frobenius norm:

$$X^{(K)} = \underset{X^* \in A_K}{\text{argmin}} ||X - X^*||_F^2 = \underset{X^* \in A_K}{\text{argmin}} \{(X - X^*)(X - X^*)^T\}$$

where A_K is the set of all $n \times d$ matrices of rank K.

2 Algorithm

In this section, we present the algorithm for extracting the first SSVD layer $s_1 u_1 v_1^T$ of a data matrix. As indicated by Lee et al. (2010)[1], subsequent layers can be extracted sequentially by applying the same algorithm to the residual matrices after removing preceding layers. As we introduced in the end of the background section, SVD yields a closest matrix approximation to the data matrix in terms of minimizing the squared Frobenius norm. However, in order to introduce sparsity to the matrices of singular vectors, we need to minimize the triplets (s, u, v) according to the following criterion:

$$||X - suv^T||_F^2 + \lambda_u P_1(su) + \lambda_v P_2(sv)$$

where $P_1(su)$ and $P_2(sv)$ are sparsity-inducing penalty terms, λ_u and λ_v are two nonnegative penalty parameters that balance the goodness-of-fit measure $||X - suv^T||$ and the penalty terms.

2.1 LASSO

In order to find appropriate sparsity-inducing penalty terms, the idea of LASSO regularization, which performs feature selection in terms of shrinking some parameters to exactly 0, is used:

- For fixed u, minimize $||X - u\tilde{v}^T||_F^2 + \lambda_v P_2(\tilde{v}) = ||Y - (I_d \otimes u)\tilde{v}||^2 + \lambda_v P_2(\tilde{v})$ with respect to $\tilde{v} = sv$ where $Y = (x_1^T, ..., x_d^T) \in R^{nd}$($x_j$: jth column of X; $\otimes$: Kronecker product).
- For fixed v, minimize $||X - u\tilde{v}^T||_F^2 + \lambda_u P_1(\tilde{u}) = ||Z - (I_n \otimes v)\tilde{u}||^2 + \lambda_u P_1(\tilde{u})$ with respect to $\tilde{u} = su$ where $Z = (x_{(1)}, ..., x_{(N)})^t \in R^{nd}$($x_{(i)}^T$: ith row of X; $\otimes$: Kronecker product).

2.2 Penalty Terms: $P_1(\tilde{u} = su)$ and $P_2(\tilde{v} = sv)$

In the algorithm, adaptive LASSO, a broader class of penalties, is adopted where data-driven weights are added:

$$P_1(\tilde{u}) = s \sum_{i=1}^{n} w_{1,i}|u_i| \text{ and } P_2(\tilde{v}) = s \sum_{j=1}^{d} w_{2,j}|v_j|$$

where the weights can be chosen from

$$w_2 = (w_{2,1}, ..., w_{2,d})^T = |\hat{\tilde{v}}|^{\gamma_2} \text{ and } w_1 = (w_{1,i}, ..., w_{1,n})^T = |\hat{\tilde{u}}|^{\gamma_1}$$

where the OLS estimators of $\tilde{u}$ and $\tilde{v}$ can be calculated as follows: $\hat{\tilde{v}} = X^T u$ and $\hat{\tilde{u}} = Xv$ and γ_2 and γ_1 are nonnegative known parameters usually be chosen from $\{0, 1, 0.5, 2\}$ corresponding to different criterion.With the above penalty terms, the process presented in 2.1 becomes:

- For fixed u, minimize $||X - u\tilde{v}^T||_F^2 + \lambda_v \sum_{j=1}^d w_{2,j}|\tilde{v}_j| = ||X||_F^2 + \sum_{j=1}^d [\tilde{v}_j^2 - 2\tilde{v}_j^2(X^T u)_j + \lambda_v w_{2,j}|\tilde{v}_j|]$
- For fixed v, minimize $||X - u\tilde{v}^T||_F^2 + \lambda_u \sum_{i=1}^n w_{1,i}|\tilde{u}_i| = ||X||_F^2 + \sum_{i=1}^n [\tilde{u}_i^2 - 2\tilde{u}_i^2(Xv)_i + \lambda_u w_{1,i}|\tilde{u}_i|]$

By the LEMMA mentioned by Lee et al. (2010)[1]: The minimizer of $\beta^2 + 2y\beta + 2\lambda|\beta|$ is $\hat{\beta} = sign(y)(|y| - \lambda)_+$, with $X^T u$ and Xv be y; $\lambda_v w_{2,j}/2$ and $\lambda_u w_{1,i}/2$ be λ, the above process becomes:

- Let $\tilde{v}_j = sign\{(X^T u)_j\}(|(X^T u)_j| - \lambda_v w_{2,j}/2)_+$; let$s = ||\tilde{v}||$ and $v = \frac{\tilde{v}}{s}$(separate out the scaling).
- Let $\tilde{u}_i = sign\{(Xv)_i\}(|(Xv)_i| - \lambda_u w_{1,i}/2)_+$; let$s = ||\tilde{u}||$ and $u = \frac{\tilde{u}}{s}$(separate out the scaling).

2.3 Penalty Parameters: λ_u and λ_v

BIC is used in the algorithm to select the optimal number of nonzero coefficient in LASSO regression:

- For fixed u, define BIC$(\lambda_v) = \frac{||Y - \hat{Y}||^2}{nd \cdot \hat{\sigma}}^2 + \frac{log(nd)}{nd}\hat{df}(\lambda_v)$ where $\hat{df}(\lambda_v)$ is the degree of sparsity of v with λ_v as the penalty parameter and $\hat{\sigma}^2$ is the OLS estimator of the error variance.
- For fixed v, define BIC$(\lambda_u) = \frac{||Z - \hat{Z}||^2}{nd \cdot \hat{\sigma}}^2 + \frac{log(nd)}{nd}\hat{df}(\lambda_u)$ where $\hat{df}(\lambda_u)$ is the degree of sparsity of u with λ_u as the penalty parameter and $\hat{\sigma}^2$ is the OLS estimator of the error variance.

2.4 SSVD algorithm

With all the above specifications above, the overall algorithm of SSVD is presented step by step as follows:

Step 1: Apply SVD to X, get triplet$\{s_{old}, u_{old}, v_{old}\}$.

Step 2: Update v: let $\tilde{v}_j = sign\{(X^T u_{old})_j\}(|(X^T u_{old})_j| - \lambda_v w_{2,j}/2)_+$; let $\tilde{v} = (\tilde{v}_1, ..., \tilde{v}_d)^T$ and $s = ||\tilde{v}||$; let $v_{new} = \frac{\tilde{v}}{s}$ (BIC$(\lambda_v) = \frac{||Y - \hat{Y}||^2}{nd \cdot \hat{\sigma}}^2 + \frac{log(nd)}{nd}\hat{df}(\lambda_v)$)

Step 3: Update u: let $\tilde{u}_j = sign\{(Xv_{old})_i\}(|(Xv_{old})_i| - \lambda_u w_{1,i}/2)_+$; let $\tilde{u} = (\tilde{u}_1, ..., \tilde{u}_n)^T$ and $s = ||\tilde{u}||$; let $u_{new} = \frac{\tilde{u}}{s}$ (BIC$(\lambda_u) = \frac{||Z - \hat{Z}||^2}{nd \cdot \hat{\sigma}}^2 + \frac{log(nd)}{nd}\hat{df}(\lambda_u)$)

Step 4: Let $u_{old} = u_{new}$, keep updating u and v till convergence.

Step 5: Let $u = u_{new}$, $v = v_{new}$ and $s = u_{new}^T X v_{new}$ at convergence.

3 Optimization and Performance

3.1 Profiling

We use the %prun decorator to profile our code's performance on our simulated data(2). By doing so, we can identify computational bottlenecks in our code and try to optimize with methods such as JIT and Cython.

Function	Number Of Calls	Total Time(s)	Average Time(s)
ssvd	1	0.047	0.047
numpy.sum	1225	0.019	0.000
numpy.linalg.svd	1	0.017	0.017
thresh	608	0.012	0.000
sum	1225	0.009	0.000
numpy.zeros	608	0.005	0.000
math.log	600	0.003	0.000

Table 1: Profiling of initial code.

From the profiling of our code, we can see ssvd function takes more than 90% of runtime, which is reasonable since most of our numerical operations are performed in this function. Also, thresh function are called 608 times and could be optimized. Other numpy functions are already very efficient, and we will focus on our user-defined-function. We will use JIT from numba package and Cython to optimize the performance of our functions.

3.2 JIT Compilation

We can use the Just-In-Time(JIT) complier from numba package to optimize our functions by adding @JIT decorator to our code.JIT complier will generate optimized machine code on the fly. From Table 2, we can see JIT does not improve much in terms of runtime.

3.3 Parallel Processing

We utliize IPython cluster to distribute the computational workload. Although the SSVD algorithm is embrassingly parallel since it has a few steps that require sequential update, we can still improve the runtime within those steps where no dependency exists. By using ipyparallel package, we significantly improve runtime.

In addition, we experimented with thread pool to distribute the computation of SSVD function across multiple threads, which also speeds up the runtime significantly.

Method	Wall time(ms)
Pure Python	43.8
JIT	39.1
IPyParallel	5.15
ThreadPool	4.43

Table 2: Runtime Comparison.

4 Application

In this section, we use a simulated data set to evaluate the performance of SSVD algorithm and compare it to other related methods. In addition, we apply our implemented SSVD algorithm to the lung cancer data set in the paper and recover the plots presented in the paper. Furthermore, we also find a MNIST(Modified National Institute of Standards and Technology) data set which contains hand-written digits. We apply SSVD to the data set and try to interpret the results and evaluate its performance.

4.1 Simulated Data Set

We use rank-1 signal matrix X^* to evaluate the performance of SSVD. We define

$$X^* = suv^T$$

where $s = 50$, $u = \frac{\hat{u}}{\|u\|}$, $v = \frac{\hat{v}}{\|v\|}$ and

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

$\hat{v} = [10, -10, 8, -8, 5, -5, r(3, 5), r(-3, 5), r(0, 34)]^{T^{\cdot}}$

Note that $r(x, y)$ denotes a vector with y entries of value x. Hence, X^* is a 100 by 50 matrix. We calculate u and v, which have 25 and 16 non-zero entries respectively. We can generate X as the sum of X^* and the random noise ϵ which is sampled from a normal distribution. Then, we repeat this process 100 times and use SSVD function in each iteration. In SSVD algorithm, we use BIC to choose the sparsity we need and use weight parameters $\gamma_1 = \gamma_2 = 2$ for weight vectors. After 100 iterations, we calculate the average number of zeros, the average proportion of correctly classified zeros and non-zeros, and the misclassification rate in two directions. From our results, it is evident that SSVD can classify most elements correctly.

4.2 Lung Cancer Data

Lung cancer data set(X) is an HDLSS data set containing 56(n) samples or subjects and 12625(d) genes for each subject. Each subject belongs to one of the following categories of cancer { Normal, Carcinoid, Colon, SmallCell}. But the informations of cancer types are not used in our algorithm. They are only used to interpret the checkerboard structure achieved by SSVD(unsupervised). This is because, as we mentioned before, SSVD would only return a series of sparse singular vectors instead of explicit clusters labels for rows and columns of the data matrix.

In this section, we mainly want to validate our implementation of the SSVD algorithm by regenerating the plots presented in the original paper. With this particular data set, the first three layers have been extracted by SSVD since the first three singular values are much larger than the others.

The same as what indicated in the paper, the matrices of the three layers are modified for better visualization: columns are sorted according to values of v, rows are sorted according to the values of u for each cancer category, and the 8000 genes in the middle white area are excluded for a better view. The generated plots are as follows:

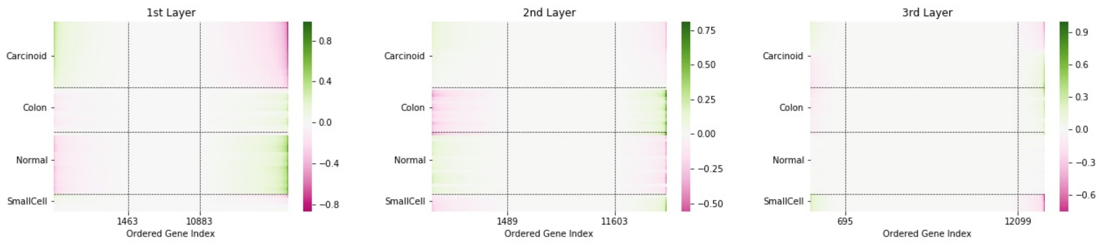

Figure 1: Output of SSVD on lung cancer data

As we can see from the above images, we have successfully find checkerboard patterns in the three extracted layers. We can see in the first layer, the first 1463 genes are mostly positively expressed for Carcinoid while negatively expressed for Normal and Colon. And we can the opposite patterns for the last 1742 genes. In the subsequent layers, we can also see contrasts between different groups.

4.3 MNIST Database

MNIST database is a large database of handwritten digits.This database is widely used for training and testing various machine learning algorithms. This database contains images information for hand-written Arabic numbers from 0 to 9 on a 28×28 field. Each row corresponds to each observation of an Arabic number while each column corresponds to grey levels at a particular pixel position.

In our case, we extracted 25 observations for the number 4 and 25 observations for number 9 from the database, excluding columns(pixel positions) that are all 0 for all rows(439 columns remaining). And then we applied SSVD to the modified data set. Since the first singular value is much larger than the other, we only extract the first layer. The result is visualized similarly as we have done with the lung cancer data set. And the following plots are generated:

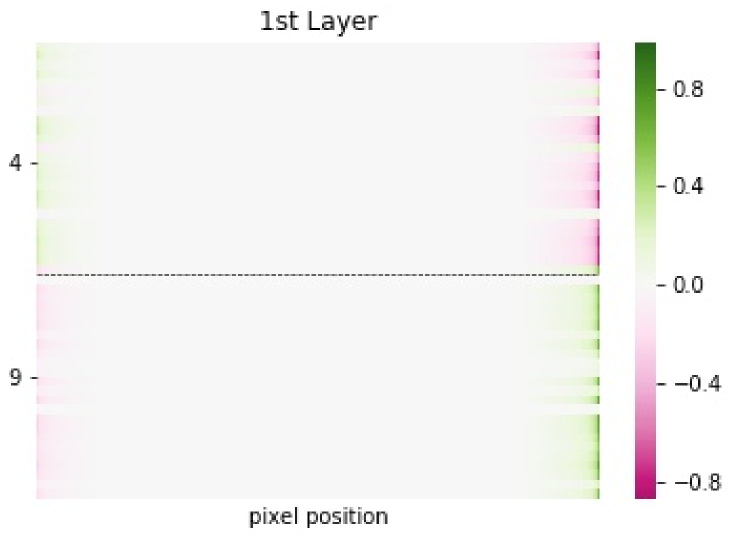

Figure 2: Output of SSVD on simulated data

As we can see from the above image, we have successfully find checkerboard patterns in the three extracted layers although is pattern is not extremely clear. The pattern is not very clear since the dimension is not very high and a lot of columns have been zeroed out. However, We can still see in the first layer, there is relative obvious contrasts for the first and last couples of pixel positions.

5 Comparative Analysis

Besides SSVD, we also utilized other methods to our simulated data to compare their performance. For rank-1 approximation, we first consider Plaid to perform biclustering, which is a SVD-based method. It assume that the data matrix can be estimated by a SVD-like structure.
For Plaid, we assume that

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

where K is the number of layers, ρ_{ik} is an indicator of whether row i is in the kth layer, and κ_{jk} is an indicator of whether column j is in the Kth layer. θ_{ijk} denotes the contribution of kth layer. In addition to methods investigated by Lee, et al, we utilize the spectral biclustering method (Kluger, 2003) implemented in Sci-Kit Learn package. We show the performance of each method on our simulated data in the following table.

Method	direction	Avg.Proportion of correctly identified zeros	Avg. Proportion of correctly identified zeros	Misclassification rate
SSVD	**u**	98.60%	99.12%	1.27%
	v	99.62%	100.00%	0.26%
SVD	**u**	0.00%	100.00%	75.0%
	v	0.00%	100.00%	82.58%
RoBiC	**u**	100.00%	37.60%	15.60%
	v	99.94%	53.25%	15.00%
Plaid	**u**	100.00%	37.40%	15.65%
	v	52.74%	85.88%	36.66%

Table 3: Comparison between SSVD, SVD, RoBiC and Plaid. Adapted from "Case 1: Comparison of the performance among SSVD, RoBic, Plaid, SVD, SPCA(u), and SPCA(v)" by Lee, Mihee.

6 Conclusion

In this paper, we implemented the Sparse Singular Value Decomposition algorithm proposed by Lee, et al in Python and tested it with both simulated data and real-life data sets. We find that compared to other methods, particularly SVD-related ones, SSVD has the advantage of correctly identify zeros, thus very useful for dimension reduction for high dimensional low sample size data. However, the SSVD algorithm has its own limitation in the sense that its penalty terms force low values to zero and hence some structures will disappear. Nonetheless, for many data sets, SSVD is very useful for biclustering because of its low misclassification rate. Additional improvement can be made in our Python code to improve performance. For example, use Cython/C++ to write our code and parallelize the intermediate steps of computation. Also, we can use approximate SVD to speed up the computation for large data matrix since our algorithm does not require exact singular values. In addition, our implementation can be found in Python package "ssvd".

References

[1] Lee, Mihee, et al. "Biclustering via Sparse Singular Value Decomposition" Biometrics 66.4(2010):1087-1095

[2] Tan, Kean Ming, and Daniela M.Witten. "Sparse Biclustering of Transposable Data." *Journal of computational and graphical statistics: a joint publication of American Statistical Association, Institute of Mathematical Statistics, Interface Foundation of North America.* 23.4(2014):985-1008.PMC. Web.26 Apr.2018.