Singular Value Decomposition (SVD)

Vien Bui and Sumanth Bail

I. INTRODUCTION

In our search for a project topic, we found Singular Value Decomposition (SVD) popping up in numerous applications. We realized that this is a fundamental topic of study that gives insight into a very diverse range of problems from dimensionality reduction to digital forensics. This was our motivation to dig a little deeper into SVD by looking at its various definitions and a couple of its applications to get a better understanding and maybe whet our appetite for more, way past this course.

In this submission we would like present different "forms" of SVD and focus on three of its real-life applications: SVD in finding pseudo-inverse matrix and applying that matrix in solving linear regression problem, SVD in image compression, and SVD in image watermarking. Source code demonstrating the applications is also attached with the submission for verification purpose.

II. DEFINITION

1. Definition

Given an $m \times n$ matrix A, the factorization $A = UDV^T$ is called the Singular Value Decomposition of A where:

- D is an $m \times n$ matrix whose:
  - Top n rows contain $diag(\sigma_1, \sigma_2, \dots, \sigma_n)$ and bottom $m - n$ rows are zeros when $m \geq n$
  - Left m columns contain $diag(\sigma_1, \sigma_2, \dots, \sigma_m)$ and right $n - m$ columns are zeros when $m < n$

  σ_i are called *singular values*.
- U is an $m \times m$ matrix, $U = [u_1, u_2, \dots, u_m]$, which is called *the left singular vector matrix* of A
- V is an $n \times n$ matrix, $V = [v_1, v_2, \dots, v_n]$, which is called *the right singular vector matrix* of A.

In U, u_i are *orthogonal unit vectors* where $u_i u_i = 1$ and $u_i u_{i'} = 0$. Therefore, U is an *orthogonal matrix* where $U^T U = UU^T = I$. The same discussion applies to V.

E.g.: using NumPy's SVD library, we can compute the SVD of an arbitrary matrix with parameter "full_matrices" equals to "True".

The SVD of a 3*2 matrix:

```python
import numpy as np

A = np.array([[3,2],[2,3],[2,-2]])
print('-----Matrix A-------')
print(A)

print('-----SVD: U, D, Vt-------')
[U,D,Vt] = np.linalg.svd(A, full_matrices=True)
print(U)
print(D)
print(Vt)
```

```
-----Matrix A-------
[[ 3  2]
 [ 2  3]
 [ 2 -2]]
-----SVD: U, D, Vt-------
[[-7.07106781e-01  2.35702260e-01 -6.66666667e-01]
 [-7.07106781e-01 -2.35702260e-01  6.66666667e-01]
 [-1.66533454e-16  9.42809042e-01  3.33333333e-01]]
[5. 3.]
[[-0.70710678 -0.70710678]
 [ 0.70710678 -0.70710678]]
```

The SVD of a 2*3 matrix:

```python
import numpy as np

A = np.array([[3,2,2],[2,3,-2]])
print('-----Matrix A-------')
print(A)

print('-----SVD: U, D, Vt-------')
[U,D,Vt] = np.linalg.svd(A, full_matrices=True)
print(U)
print(D)
print(Vt)
```

```
-----Matrix A-------
[[ 3  2  2]
 [ 2  3 -2]]
-----SVD: U, D, Vt-------
[[-0.70710678 -0.70710678]
 [-0.70710678  0.70710678]]
[5. 3.]
[[-7.07106781e-01 -7.07106781e-01 -5.55111512e-17]
 [-2.35702260e-01  2.35702260e-01 -9.42809042e-01]
 [-6.66666667e-01  6.66666667e-01  3.33333333e-01]]
```

Demonstration of *orthogonal unit vectors* and *orthogonal matrix* properties:

```
print("-----U'.U and U.U'------")
print(np.matrix.round(U.T.dot(U))) #matrix.round: avoid NumPy's imprecision
print(np.matrix.round(U.dot(U.T)))

print("-----Vt'.Vt and Vt.Vt'------")
print(np.matrix.round(Vt.T.dot(Vt)))
print(np.matrix.round(Vt.dot(Vt.T)))

print('----u characteristics---------')
print(U[:,0].dot(U[:,0])) # U's columns are unit vectors
print(U[:,0].dot(U[:,1])) # U's columns are orthogonal to each other
```

```
-----U'.U and U.U'------
[[1. 0.]
 [0. 1.]]
[[1. 0.]
 [0. 1.]]
-----Vt'.Vt and Vt.Vt'------
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
[[ 1.  0.  0.]
 [ 0.  1. -0.]
 [ 0. -0.  1.]]
----u characteristics---------
1.0
1.1102230246251565e-16
```

2. SVD Forms

The above definition is the **Full version** of SVD. There are several smaller versions of the SVD that are often computed. Let $U_t = [u_1, u_2, \dots, u_t]$ be an $m \times t$ matrix of the first t left singular vectors, $V_t = [v_1, v_2, \dots, v_t]$ be an $n \times t$ matrix of the first t right singular vectors, and $D_t = diag(\sigma_1, \sigma_2, \dots, \sigma_t)$ be a $t \times t$ matrix of the first t singular values. Consider the case $m > n$ (the case of $m < n$ is very similar), we can make the following definitions:

Thin SVD: $A = U_n D_n V_n^T$ is the thin SVD of A. The thin SVD is much smaller and faster to compute than then full SVD when $m \gg n$.

E.g.:

```
A = np.array([[3,2],[2,3],[2,-2]])
print('-----Matrix A-------')
print(A)

print('-----SVD: U, D, Vt-------')
[U,D,Vt] = np.linalg.svd(A, full_matrices=False)
print(U)
print(D)
print(Vt)
```

```
-----Matrix A-------
[[ 3  2]
 [ 2  3]
 [ 2 -2]]
-----SVD: U, D, Vt-------
[[-7.07106781e-01  2.35702260e-01]
 [-7.07106781e-01 -2.35702260e-01]
 [-1.66533454e-16  9.42809042e-01]]
[5. 3.]
[[-0.70710678 -0.70710678]
 [ 0.70710678 -0.70710678]]
```

Compact SVD: $A = U_r D_r V_r^T$ is the compact SVD of A where r is the number of positive singular values and $r < n$.

E.g.:

```
A = np.array([[3, 2, 4], [-1, 1, 2], [9, 5, 10]])
print('-----A matrix of rank 2 (not full rank)-------')
print(A)
[U,D,Vt] = np.linalg.svd(A, full_matrices=False)
print('-----SVD: U, D, Vt-------')
print(np.matrix.round(U,2))
print(np.matrix.round(D,2))
print(np.matrix.round(Vt,2))
print('-----Compact SVD: Ur, Dr, Vtr-------')
Ur = U[:,:2]
Dr = D[:2]
Vtr = Vt[:2,:]
print(Ur.dot(np.diag(Dr).dot(Vtr)))
```

```
-----A matrix of rank 2 (not full rank)-------
[[ 3  2  4]
 [-1  1  2]
 [ 9  5 10]]
-----SVD: U, D, Vt-------
[[-0.35 -0.16 -0.92]
 [-0.08 -0.98  0.2 ]
 [-0.93  0.14  0.33]]
[15.37  2.21  0.  ]
[[-0.61 -0.35 -0.71]
 [ 0.79 -0.27 -0.55]
 [ 0.   -0.89  0.45]]
-----Compact SVD: Ur, Dr, Vtr-------
[[ 3.  2.  4.]
 [-1.  1.  2.]
 [ 9.  5. 10.]]
```

Truncated SVD: $A_t = U_t D_t {V_t}^T$ is the rank-t truncated SVD of A, where $t < r$. $A_t = U_t$ is the best rank-t approximation of A, i.e.: among all rank-t matrices B, $\|A - A_t\|_F \leq \|A - B\|_F$. The truncated SVD is much smaller to store and cheaper to compute than the compact SVD when we choose $t \ll r$, and is the most common form of the SVD computed in applications.

Below is a summary of SVD forms:

	Size			
SVD Forms	**Matrix A**	**Matrix U**	**Matrix D**	**Matrix V^T**
Full	m*n	m*m	m*n	n*n
Thin	m*n	m*n	n*n	n*n
Compact ($r < n$)	m*n	m*r	r*r	r*n
Truncated ($t \ll r$)	m*n	m*t	t*t	t*n

III. APPLICATIONS

The first and most important application of SVD is to reduce the dimensionality of data. However, we will explore other three different applications of SVD to demonstrate its potential.

1. Pseudoinverse Matrix & Linear Regression

The Moore-Penrose pseudoinverse A^+ of an arbitrary matrix A is generalization of the inverse matrix of A. That is, when A has full rank, i.e.: there exists an inverse matrix A^{-1} of A, A^+ is A^{-1}. The pseudoinverse can be computed by $A^+ = VD^+U^T$ where with U, D and V respectively the left singular vector matrix, the singular value matrix and the right singular vector matrix of A. D^+ is the pseudoinverse of D and can be calculated by taking the reciprocal of the non-zero values of D, then transposing the matrix (in numerical computation, for example in NumPy, only values larger than some small tolerance are taken to be non-zero, and the others are replaced by zeros).

E.g.:

```python
A = np.array([[7, 2], [3, 4], [5, 3]])
U, D, Vt = np.linalg.svd(A, full_matrices=True)

# Pseudoinverse calculation
D_plus = np.zeros(A.shape)
# The following line assumes that there are no zero values
# in D for quick calculation
D_plus[:D.shape[0],:D.shape[0]] = np.diag(1/D)
D_plus = D_plus.T
A_plus = Vt.T.dot(D_plus.dot(U.T))

print("--------Pseudoinverse by SVD------")
print(A_plus)
print("--------Pseudoinverse by NumPy's pinv method------")
print(np.linalg.pinv(A))
```

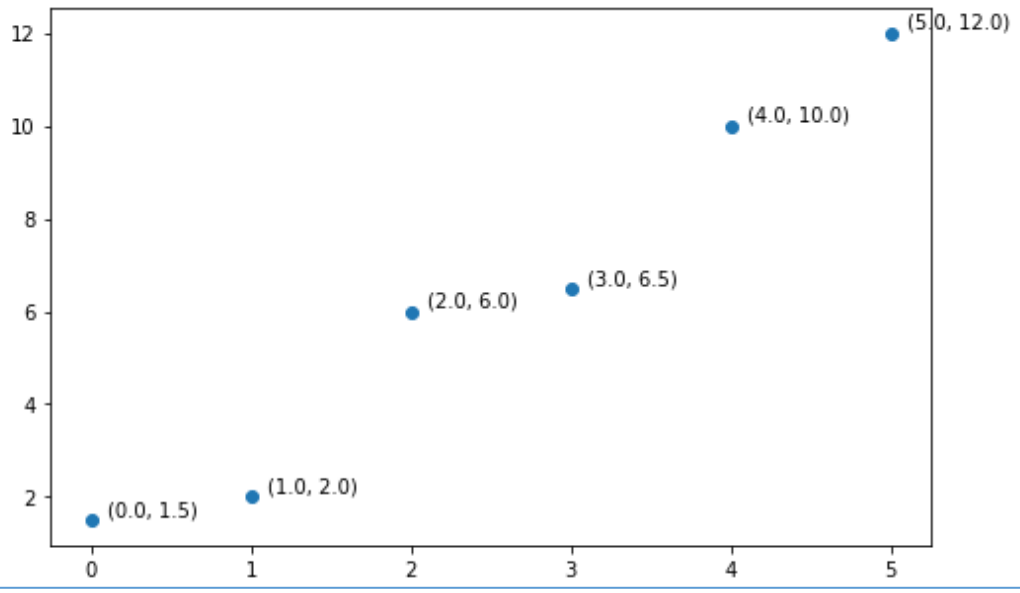
```
--------Pseudoinverse by SVD------
[[ 0.16666667 -0.10606061  0.03030303]
 [-0.16666667  0.28787879  0.06060606]]
--------Pseudoinverse by NumPy's pinv method------
[[ 0.16666667 -0.10606061  0.03030303]
 [-0.16666667  0.28787879  0.06060606]]
```

In a linear regression problem, the pseudoinverse provides a least-squares solution to the system of linear equations. To demonstrate the use of pseudoinverse, let’s consider the following example: we will try to find a line going through all these points.

```
import matplotlib.pyplot as plt
%matplotlib inline

dataset = np.array([[0, 1.5], [1, 2], [2, 6], [3, 6.5], [4, 10], [5, 12]])

plt.rcParams["figure.figsize"] = [8,5]
plt.scatter(dataset[:, 0], dataset[:, 1])
for i in np.arange(dataset.shape[0]):
    plt.annotate('('+str(dataset[i][0]) + ', ' + str(dataset[i][1]) + ')',
                 (dataset[:,0][i]+0.1, dataset[:,1][i]+0.1))
plt.show()
```



Let *s* be the slope and *i* be the intercept of that line. We need to find *s* and *i* so that:

$$\begin{cases} 0s + i = 1.5 \\ 1s + i = 2 \\ 2s + i = 6 \\ 3s + i = 6.5 \\ 4s + i = 10 \\ 5s + i = 12 \end{cases}$$

We can rewrite the above requirement as:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \\ 2 & 1 \\ 3 & 1 \\ 4 & 1 \\ 5 & 1 \end{bmatrix} \begin{bmatrix} s \\ i \end{bmatrix} = \begin{bmatrix} 1.5 \\ 2 \\ 6 \\ 6.5 \\ 10 \\ 12 \end{bmatrix}$$

or $A\vec{x} = \vec{b}$.

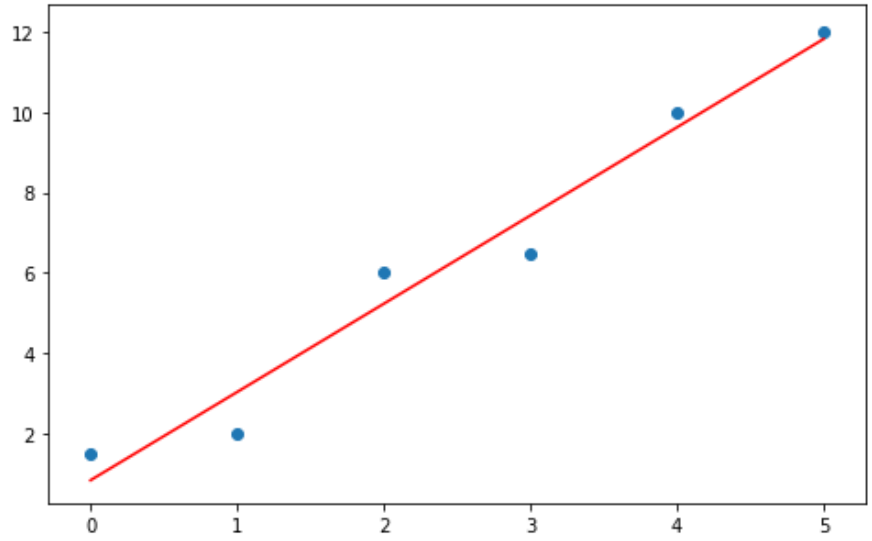
In general, the vector $\vec{x}$ that solves that system does not exist. The pseudoinverse solves the "least-squares" problem as follows: we have $\left\|A\vec{x} - \vec{b}\right\|_2 \geq \left\|A\vec{z} - \vec{b}\right\|_2$ where $\vec{z} = A^+\vec{b}$. The square root of sum of square errors $\left\|A\vec{x} - \vec{b}\right\|_2$ is minimum when $\vec{x} = \vec{z}$, i.e.: $\vec{x} = A^+\vec{b}$. Therefore, using the pseudoinverse of A, we can quickly solve the problem of linear regression.

```python
# Construct matrix A and vector b
A = dataset.copy()
A[:,1] = 1
b = dataset[:,1]

# Calculate A_plus
A_plus = np.linalg.pinv(A)

# Calculate vector x
x = A_plus.dot(b)

# Plot the line
lon = np.linspace(0,5,100)
lat = x[0]*lon + x[1]
plt.plot(lon, lat, '-r')
plt.scatter(dataset[:, 0], dataset[:, 1])
plt.show()
```



2. Image Compression

As an application of truncated version, SVD can be used to extract the most important features from images. As a result, the resulting image's size will be much lesser that the original size. The idea is to apply rank-t truncated SVD on the matrix of pixels of an image.

E.g.:

Here is an original image of grayscale:

Here are the results of different *t* values:

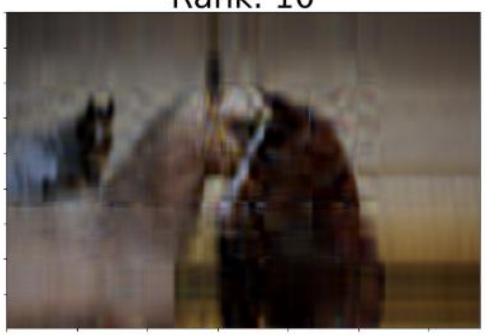
Only with the first 90 singular values, we can reconstruct a quality compressed image with the size dropped for more than 3 times (4,685kb to 1,400kb).

We can also do the same thing to color images. This time, instead of applying SVD to only 1 matrix, we apply SVD to 3 matrices of the 3 channels (R, G, B). Here is a color image:

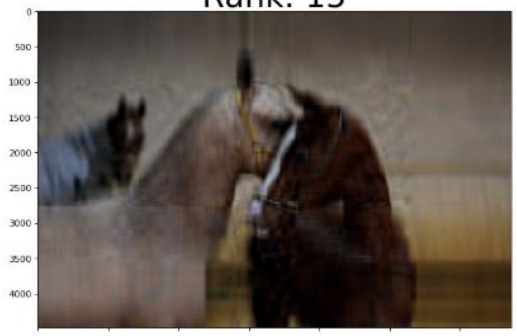
Here are the results of different *t* values:

Rank: 5

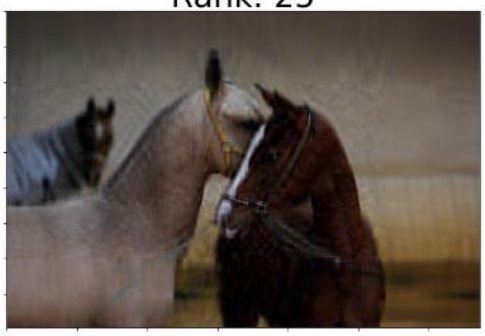
Rank: 10



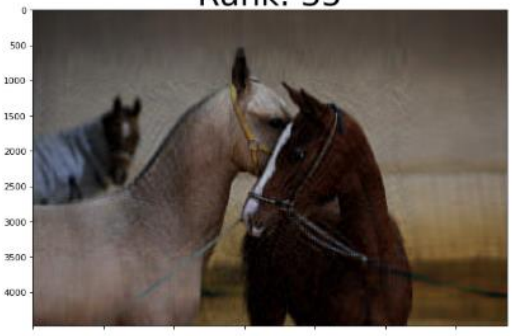
Rank: 15



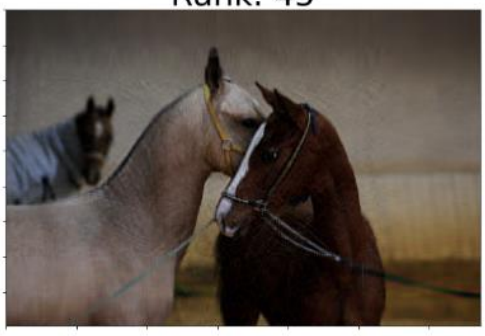
Rank: 25



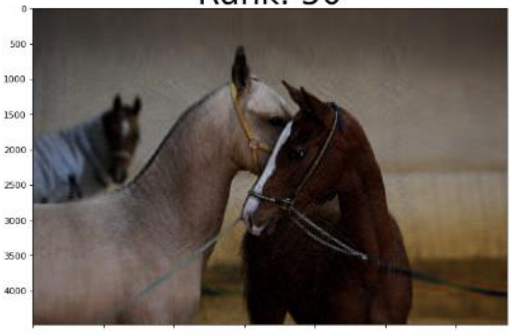
Rank: 35



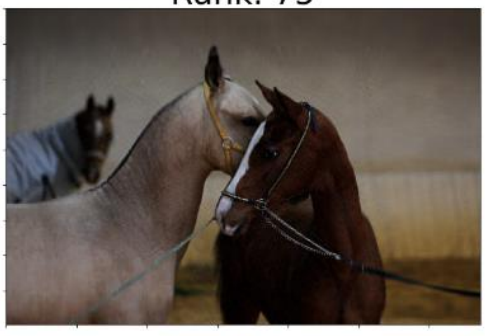
Rank: 45



Rank: 50



Rank: 75



3. Image Watermarking

Another interesting application of SVD is image watermarking in digital forensics, which is based on both full (or thin) and truncated version of SVD. The basic idea is, for copyright protection reason, we need to find a way to embed a watermark image into a target image so that the watermarked image is basically the same as the original target image (the difference cannot be discovered by naked eyes) and later on the watermark image can be extracted from the watermarked image to prove the ownership.

Embedding is carried out by first getting the SVD for the host image X and watermark image W as follows:

$$X = U_h S_h V_h^T$$

$$W = U_w S_w V_w^T$$

Where S_h and S_w are singular values of host image and watermark image respectively.

Then we calculate the singular values of the watermarked image by adding the first *k* singular values of the watermark image to the first *k* singular values of the host image:

$$S_m(i) = S_h(i) + \alpha * S_w(i) \qquad for\ 1 \le i \le k$$

α is a scaling factor which is adjustable by user to increase (or decrease) the watermarked image fidelity and decrease (or increase) the robustness of watermark protection. *k* is user defined and could be chosen adaptively based on the energy distribution in both host and the watermark image.

Finally, the watermarked image *Y* is constructed from the modified singular values S_m as:

$$Y = U_h S_m V_h^T$$

The process of extraction the watermark image from the watermarked image are done in the reverse ordering. First obtain the SVD of the watermarked image *Y* and the host image *X*:

$$Y = U_m S'_m V_m^T$$

$$X = U_h S_h V_h^T$$

Then extract the embedded singular values S'_w:

$$S'_w(i) = \frac{S'_m(i) - S_h(i)}{\alpha} \qquad for\ 1 \le i \le k$$

Finally, reconstruct the watermark image *W'* by obtaining the inverse of the SVD:

$$W' = U_w S'_w V_w^T$$

Below is a demonstration of this algorithm with $\alpha = 0.2$ and $k = 100$.

Host image

Watermark image

Watermarked image

Reconstructed watermark image

This method of image watermarking was an early-stage method of watermarking. Later research found that this method is subjected to a severe false positive problem: a different watermark can be extracted from the watermarked image by using different singular vector matrices. Other better versions of SVD based image watermarking have been developed later. However, this method serves the purpose of this paper to demonstrate the potential of SVD in Image Forensic domain. In essence, we can use SVD to embed information into an image without being discovered by naked eyes. The resulting image is basically the same as the original host image. And the most interesting thing is the embedded information can be perfectly extracted with minor error. For a better understanding, please see the attached source code.

IV. CONCLUSION

In this project we discussed using SVD for finding pseudoinverse matrix, image compression using SVD on 2D and 3D arrays and talked about image watermarking. We learnt a lot working on this project and can say that this is just the tip of the iceberg for us when it comes to SVD.

V. REFERENCES

[1] http://www.netlib.org/utk/people/JackDongarra/etemplates/node43.html

[2] https://en.wikipedia.org/wiki/Moore%E2%80%93Penrose_inverse

[3] https://hadrienj.github.io/posts/Deep-Learning-Book-Series-2.9-The-Moore-Penrose-Pseudoinverse/

[4] https://arxiv.org/ftp/arxiv/papers/1211/1211.7102.pdf

[5] https://www.mdpi.com/1999-5903/9/3/45/pdf