Applications of SVD

MOTIVATION

The singular value decomposition or SVD is a real or complex matrix factorization in linear algebra. SVD has been widely used in various areas, such as Moore-Penrose inverse, linear system of equations, PCA, and image compression. In this project, we are going to understand some intuitive interpretations of SVD and explore some useful applications of SVD in different fields.

1. Diagonalization

Definition: Suppose $\mathbf{A}$ is a $n \times n$ matrix, where A has n linearly independent eigenvectors x, then

$$A = XDX^{-1}$$

where

$$A = \begin{bmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_n \end{bmatrix}$$

and

$$X = \begin{bmatrix} \overrightarrow{x_1} & \overrightarrow{x_2} & \dots & \overrightarrow{x_n} \end{bmatrix}$$

As shown above, the columns of X are the linearly independent normalized eigenvectors $\overrightarrow{x_1}, \overrightarrow{x_2}, \dots, \overrightarrow{x_n}$ of the matrix A (which guarantees that X^{-1} exists) and D is a diagonal matrix with the eigenvalues $\lambda_1, \lambda_2, \dots, \lambda_n$ of the matrix A.

- If a $n \times n$ symmetric matrix $\mathbf{A}$ has n distinct eigenvalues, then A is diagonalizable.

Proof: Suppose λ, u and μ, v are eigenpairs of the matrix:

$$Au = \lambda u$$

and

$$Av = \mu v$$

Then since A is symmetric $\Rightarrow A^T = A$

$$\begin{gathered} Au = \lambda u \\ v.Au = \lambda v.u \\ A^T v.u = \lambda v.u \\ Av.u = \lambda v.u \\ \mu v u = \lambda v.u \\ (\mu - \lambda) v.u = 0 \end{gathered}$$

Considering $\mathbf{A}$ has n distinct eigenvalues $\Rightarrow \mu - \lambda \neq 0 \Rightarrow v.u = 0$

Therefore, eignvectors are orthogonal $\rightarrow \mathbf{A}$ is diagonalizable.

- If a $n \times n$ matrix $\mathbf{A}$ has less than n linearly independent eigenvectors, then A is not diagonalizable which is called defective.

- Considering the indenitty matrix:
$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & 1 \end{bmatrix}$$
which has eigenvalues $\lambda_1 = \lambda_2 = \cdots = \lambda_n = 1$, and n linearly independent eigenvectors:
$U = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & 1 \end{bmatrix}$. Even though the matrix I is diagonalizable but does not have distinct eigenvalues. As we can see, $\lambda = 1$ is an eigenvalue with algebraic multiplicity equal to n and $\lambda = 1$ is an eigenvalue with geometric multiplicity equal to n

- If geometirc multiplicity < algebraic multiplicity, the eigenvalue is called defective.

2 Singular Value Decomposition

Singular Value decomposition generalies the diagonalization. The **singular value decomposition** (SVD) is a factorization of a $m \times n$ matrix A into

$$A = U\Sigma V^T$$

$$A = \begin{bmatrix} \vdots & \dots & \vdots \\ u_1 & \dots & u_m \\ \vdots & \dots & \vdots \end{bmatrix} \begin{bmatrix} \sigma_1 & & & \\ & \ddots & & \\ & & \sigma_n & \\ & & 0 & \\ & & \vdots & \\ & & 0 & \end{bmatrix} \begin{bmatrix} \dots & v_1^T & \dots \\ \vdots & \vdots & \vdots \\ \dots & v_n^T & \dots \end{bmatrix}$$

where U is a $m \times m$ orthogonal matrix, V^T is a $n \times n$ orthogonal matrix ($\sigma_1 \geq \sigma_2 \geq \sigma_3 \geq \dots$), and Σ is a $m \times n$ diagonal matrix.

2.1 First, Assume that A is a Square Matrix (m = n)

The matrix A has the singluar value decomposition $A = U\Sigma V^T$ and $A^T = (U\Sigma V^T)^T = V\Sigma^T U^T$, then we can have:

First Case (A^TA)**:**

$$A^TA = (V\Sigma^T U^T)(U\Sigma V^T) = V\Sigma^T U^T U\Sigma V^T = V\Sigma^T I\Sigma V^T = V\Sigma^T\Sigma V^T = V\Sigma^2 V^T$$
$$A^TA = V\Sigma^2 V^T$$

Considering the definition of diagonalization, we have $Z = XDX^{-1} = XDX^T$

Let $C = A^TA$, we can get $C = V\Sigma^2 V^T \Rightarrow$

$$V = \begin{bmatrix} \vdots & \dots & \vdots \\ v_1 & \dots & v_n \\ \vdots & \dots & \vdots \end{bmatrix}, \Sigma^2 = \begin{bmatrix} \sigma_1^2 & & \\ & \ddots & \\ & & \sigma_n^2 \end{bmatrix} = \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix}, V^T = \begin{bmatrix} \dots & v_1^T & \dots \\ \vdots & \vdots & \vdots \\ \dots & v_n^T & \dots \end{bmatrix}$$

where $\forall i$, we will have $\lambda_i = \sigma_i^2$ and $\sigma_i = \sqrt{\lambda_i}$

Therefore, by looking at the eigenpairs with respect to A^TA, we can conclude that:

- Columns of V are the eigenvectors of A^TA.
- Σ^2 are the eigenvalues of A^TA.

Second Case (AA^T):

$$AA^T = (U\Sigma V^T)(V\Sigma^T U^T) = U\Sigma^T V^T V\Sigma U^T = U\Sigma^T I\Sigma U^T = U\Sigma^T\Sigma U^T = U\Sigma^2 U^T$$
$$AA^T = U\Sigma^2 U^T$$

Let $D = A^T A$, we can get $D = U\Sigma^2 U^T \Rightarrow$

$$U = \begin{bmatrix} \vdots & \dots & \vdots \\ u_1 & \dots & u_n \\ \vdots & \dots & \vdots \end{bmatrix}, \Sigma^2 = \begin{bmatrix} \sigma_1^2 & & \\ & \ddots & \\ & & \sigma_n^2 \end{bmatrix} = \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix}, U^T = \begin{bmatrix} \dots & u_1^T & \dots \\ \vdots & \vdots & \vdots \\ \dots & u_n^T & \dots \end{bmatrix}$$

where $\forall i$, we will have $\lambda_i = \sigma_i^2$ and $\sigma_i = \sqrt{\lambda_i}$

Similarly, at this time we can conclude that:

- Columns of U are the eigenvectors of AA^T.
- Σ^2 are the eigenvalues of AA^T.

2.2 The Steps to compute the SVD of the Matrix A

There are four key steps to compute the SVD of the matrix which is shown below. We are going to explain each step using a random example.

Before computing the SVD of the matrix A, we aer going to generate a random 4×4 matrix A.

In [2]:
```
import numpy as np
import numpy.linalg as la
```

In [3]:
```
random_state = 42
np.random.seed(random_state)
n = 4
A = np.random.randn(n, n)
A
```

Out[3]:
```
array([[ 0.49671415, -0.1382643 ,  0.64768854,  1.52302986],
       [-0.23415337, -0.23413696,  1.57921282,  0.76743473],
       [-0.46947439,  0.54256004, -0.46341769, -0.46572975],
       [ 0.24196227, -1.91328024, -1.72491783, -0.56228753]])
```

STEP 1: Find the transpose of the matrix A and compute the $A^T A$

For this step, we can directly calculate the transpose of the matrix and multiply it by A using the knowledge of matrix multiplication.

In [4]:
```
# the transpose of A
A_transpose = A.T
A_transpose
```

Out[4]:
```
array([[ 0.49671415, -0.23415337, -0.46947439,  0.24196227],
       [-0.1382643 , -0.23413696,  0.54256004, -1.91328024],
       [ 0.64768854,  1.57921282, -0.46341769, -1.72491783],
       [ 1.52302986,  0.76743473, -0.46572975, -0.56228753]])
```

In [5]:
```
#A^T·A
ATA = A_transpose.dot(A)
ATA
```

Out[5]:
```
array([[ 0.58050469, -0.73151355, -0.24786425,  0.65940888],
       [-0.73151355,  4.02894983,  2.589515  ,  0.43286178],
       [-0.24786425,  2.589515  ,  6.10351105,  3.38411893],
       [ 0.65940888,  0.43286178,  3.38411893,  3.44164748]])
```

STEP 2: Calculate the eigenpairs of A^TA and the V matrix and check its orthonormality.

For this step, we can use linear algebra package in the Python to calculate the eigenvalues and eigenvectors,

where eigenvals, eigenvecs = la.eig(A^TA), eigenvecs $= \begin{bmatrix} | & \dots & | \\ v_1 & \dots & v_n \\ | & \dots & | \end{bmatrix}$, and eigenvals = $\begin{bmatrix} \lambda_1 & \lambda_2 & \dots & \lambda_n \end{bmatrix}$

In [6]:
```
eigenvals, eigenvecs = la.eigh(ATA)
eigenvals
```

Out[6]:
```
array([0.11672408, 0.84637788, 3.70764037, 9.48387072])
```

In [7]:
```
V = eigenvecs
V
```

Out[7]:
```
array([[ 0.83813082,  0.46067627, -0.29132142,  0.02111699],
       [ 0.01667468,  0.47881258,  0.77530415, -0.41153852],
       [ 0.2888667 , -0.54399272, -0.08592964, -0.783099  ],
       [-0.46240103,  0.51243322, -0.55376115, -0.4657747 ]])
```

Check that eigenvectors are orthonormal:

In [13]:
```
V.T @ V
```

Out[13]:
```
array([[ 1.00000000e+00, -8.32667268e-17,  5.55111512e-17,
        -2.77555756e-17],
       [-8.32667268e-17,  1.00000000e+00, -1.24900090e-16,
         1.11022302e-16],
       [ 5.55111512e-17, -1.24900090e-16,  1.00000000e+00,
        -7.63278329e-17],
       [-2.77555756e-17,  1.11022302e-16, -7.63278329e-17,
         1.00000000e+00]])
```

STEP 3: Calcualte the Σ matrix

For this step, we can construct the diagonal matrix to get Σ.

In [9]:
```
Σ = np.diag(np.sqrt(eigenvals))
Σ
```

Out[9]:
```
array([[0.34164905, 0.        , 0.        , 0.        ],
       [0.        , 0.91998798, 0.        , 0.        ],
       [0.        , 0.        , 1.9255234 , 0.        ],
       [0.        , 0.        , 0.        , 3.07958938]])
```

STEP 4: Calcualte the U matrix

Considering that $A = U\Sigma V^T$, we can get

$$AV = U\Sigma V^T V$$

Since $V^T V = I$ and $\Sigma^T \Sigma = I$

$$AV = U\Sigma$$
$$AV\Sigma^{-1} = U\Sigma\Sigma^{-1} = U$$
$$U = AV\Sigma^{-1}$$

In [12]:
```
U = A @ V @ la.inv(Σ)
U
```

Out[12]:
```
array([[-0.30191507,  0.64211164, -0.59773398, -0.37316755],
       [-0.28929081, -0.74544218, -0.3500296 , -0.48796113],
       [-0.8867168 ,  0.06190224,  0.44410852,  0.11255683],
       [-0.19720909, -0.1678592 , -0.56829657,  0.78100632]])
```

Check the orthogonality of U:

In [14]:
```
U @ U.T
```

Out[14]:
```
array([[ 1.00000000e+00, -1.11022302e-16, -2.12330153e-15,
        -8.32667268e-16],
       [-1.11022302e-16,  1.00000000e+00, -2.35922393e-16,
        -1.66533454e-16],
       [-2.12330153e-15, -2.35922393e-16,  1.00000000e+00,
        -7.49400542e-16],
       [-8.32667268e-16, -1.66533454e-16, -7.49400542e-16,
         1.00000000e+00]])
```

STEP 5: Compare with the Python SVD.

In [15]:
```
U1, Σ1, V1t = la.svd(A)
```

In [16]:
```
U1, U
```

Out[16]:
```
(array([[-0.37316755, -0.59773398,  0.64211164, -0.30191507],
        [-0.48796113, -0.3500296 , -0.74544218, -0.28929081],
        [ 0.11255683,  0.44410852,  0.06190224, -0.8867168 ],
        [ 0.78100632, -0.56829657, -0.1678592 , -0.19720909]]),
 array([[-0.30191507,  0.64211164, -0.59773398, -0.37316755],
        [-0.28929081, -0.74544218, -0.3500296 , -0.48796113],
        [-0.8867168 ,  0.06190224,  0.44410852,  0.11255683],
        [-0.19720909, -0.1678592 , -0.56829657,  0.78100632]]))
```

In [17]:
```
Σ1, Σ
```

Out[17]:
```
(array([3.07958938, 1.9255234 , 0.91998798, 0.34164905]),
 array([[0.34164905, 0.        , 0.        , 0.        ],
        [0.        , 0.91998798, 0.        , 0.        ],
        [0.        , 0.        , 1.9255234 , 0.        ],
        [0.        , 0.        , 0.        , 3.07958938]]))
```

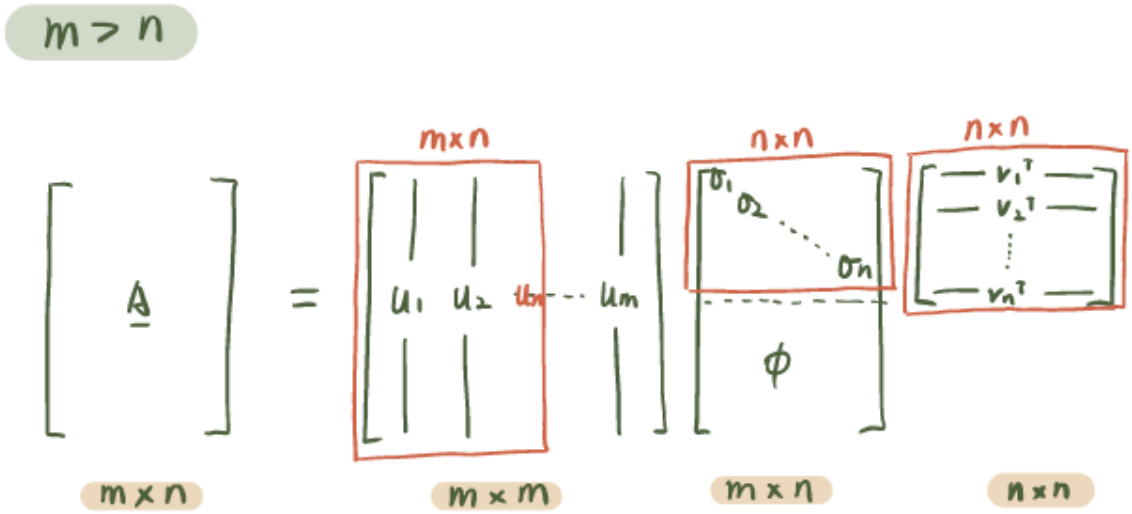
In [18]:
```
V1t.T, V
```

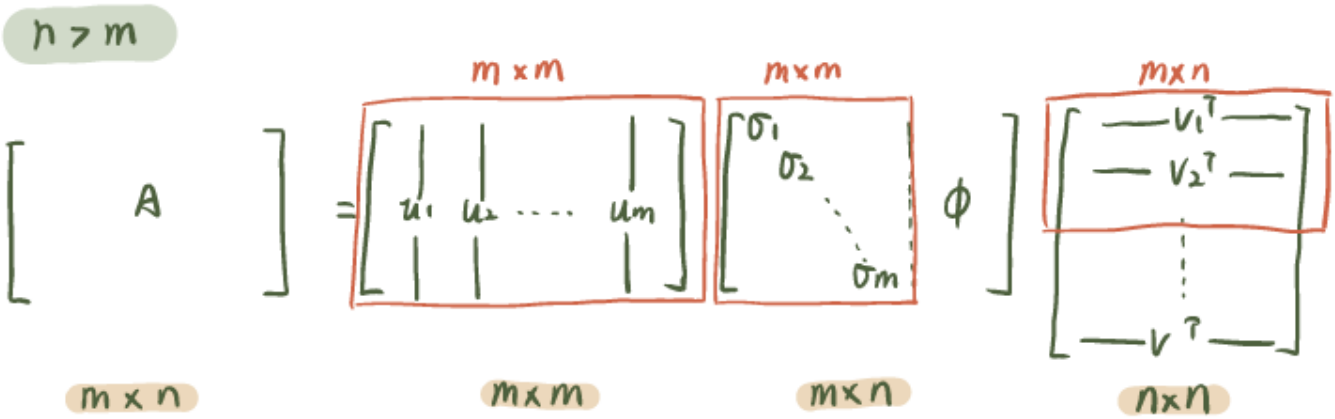
Out[18]:
```
(array([[ 0.02111699, -0.29132142,  0.46067627,  0.83813082],
        [-0.41153852,  0.77530415,  0.47881258,  0.01667468],
        [-0.783099  , -0.08592964, -0.54399272,  0.2888667 ],
        [-0.4657747 , -0.55376115,  0.51243322, -0.46240103]]),
 array([[ 0.83813082,  0.46067627, -0.29132142,  0.02111699],
        [ 0.01667468,  0.47881258,  0.77530415, -0.41153852],
        [ 0.2888667 , -0.54399272, -0.08592964, -0.783099  ],
        [-0.46240103,  0.51243322, -0.55376115, -0.4657747 ]]))
```

As we can see above we can get the same result using two different methods. In addition, the matrices U and V are not singular due to orthogonality.

2.3 Second, Assume that A is not a Square Matrix (m > n or m < n)

Reduced SVD





First Case(m > n):

In this case, we can see that there are n singular values. The Σ is a $n \times n$ diagonal matrix with positive real entries, U is a $m \times n$ matrix with orthonormal columns, and V is a $n \times n$ matrix with orthonormal columns.

Second Case(m < n):

In this case, we can see that there are m singular values. The Σ is a $m \times m$ diagonal matrix with positive real entries, U is a $m \times m$ matrix with orthonormal columns, and V is a $m \times n$ matrix with orthonormal columns.

The singular value decomposition of the $m \times n$ matirx A is caleed as **`Reduced Singular Value Decomposition`** :

$$A = U\Sigma V^T$$

where A is a $m \times n$ matrix, U is a $m \times k$ matrix, Σ is a $k \times k$ matrix, V^T ia a $k \times n$ matrix, and k = min(m, n).

Therefore, we can conclude that:

Full Rank : $A = \underset{m\times m}{U}\ \underset{m\times n}{\Sigma}\ \underset{n\times n}{V^T}$

Reduced Rank : $A = \underset{m\times k}{U}\ \underset{k\times k}{\Sigma}\ \underset{k\times n}{V^T}$ where k = min(m, n)

3. The Computation Cost of SVD

The Computational cost of SVD is calculated by a two-step process. In the first step, assume that $m \geq n$, the matrix is reduced to a bidiagonal matrix using QR decomposition and Householder reflections which in overall take $2mn^2 + 2n^3$. In the second step, the iterative method is applied to compute the SVD of the bidiagonal matrix, where the cost is O(n) which is less expensive than the first step.

As shown above, the cost of the SVD is proportional to $\beta(mn^2 + n^3)$ and the β is ranging from 4 to 10 or more with respect to different algorithm.

In conclusion,

Full Rank $m = n$: cost $\rightarrow n \cdot n^2 + n^3 = 2n^3$

Reduced Rank : $m > n$: cost $\geq 2n^3$

4. SVD APPLICATIONS

4.1 Pseudo-inverse and SVD

In linear algebra, for any $m \times n$ matrix A, there exists a unique $n \times m$ matrix $A^\dagger$ which can be computed using `Singular Value Decomposition` . Consider A be any an $m \times n$ of rank r with a singular value decomposition $A = U\Sigma V^T$ with nonzero singular value $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r$. Then we can get the pseudoinverse of A:

$$A^\dagger = V\Sigma U^\star$$

Where the $\Sigma^\dagger$ is a $n \times n$ matrix which can be computed by taking the reciprocal of all non-zero elements and then leaving all zero value alone.

Example 1: Suppose $A = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ -1 & -1 \end{bmatrix}$, we can use Python to calculate SVD which is equal to

$$\begin{bmatrix} -0.577350269 & 0.816496581 & 0 \\ -0.577350269 & -0.408248290 & -0.707106781 \\ -0.577350269 & 0.408248290 & -0.707106781 \end{bmatrix} \begin{bmatrix} \sqrt{6} & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} -0.70710678 & 0.70710678 \\ -0.70710678 & -0.70710678 \end{bmatrix}^T$$

Then we can calculate the $A^\dagger$ by using the equation above:

$$A^\dagger = \begin{bmatrix} -0.70710678 & 0.70710678 \\ -0.70710678 & -0.70710678 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{6}} & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} -0.577350269 & 0.816496581 & 0 \\ -0.577350269 & -0.408248290 & -0.7071067 \\ -0.577350269 & 0.408248290 & -0.7071067 \end{bmatrix}$$

$$A^\dagger = \begin{bmatrix} \frac{1}{6} & \frac{1}{6} & -\frac{1}{6} \\ \frac{1}{6} & \frac{1}{6} & -\frac{1}{6} \end{bmatrix}$$

which is the same as below:

```
In [84]: A = np.array([[1, 1], [1, 1], [-1, -1]])
         la.pinv(A)
```

```
Out[84]: array([[ 0.16666667,  0.16666667, -0.16666667],
                [ 0.16666667,  0.16666667, -0.16666667]])
```

4.2 Systems of Linear Equations and SVD

Consider a system of linear equations $Ax = b$ where A is an $m \times n$ matrix and $b \in F^m$, there are three possibilities for this system:

- it has no solutions
- it has a unique solutions
- it has infinitely many solutions

By Theorem,

- If $Ax = b$ is consistent and A is invertible, then $A^{-1} = A^{\dagger}$. The solution can be written as $x = A^{\dagger}b$, where x is the unique solution to the system having minimum norm.
- f $Ax = b$ is inconsistent or A is not invertible, then $A^{\dagger}b$ still exists. The solution can be written as $x = A^{\dagger}b$, where x is the unique best approximation to the system having minimum norm.

Example 2: Consider two linear systems

$$\begin{cases} x_1 + x_2 = 2 \\ x_1 + x_2 = 2 \\ -x_1 - x_2 = -2 \end{cases} and \begin{cases} x_1 + x_2 = 1 \\ x_1 + x_2 = 2 \\ -x_1 - x_2 = 2 \end{cases}$$

Obviously, the first system has infinitely many solutions. Let $A = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ -1 & -1 \end{bmatrix}$ which is the coefficient matrix of the system and $b = \begin{bmatrix} 2 \\ 2 \\ 2 \end{bmatrix}$. Using the example 1 above, $A^{\dagger} = \begin{bmatrix} \frac{1}{6} & \frac{1}{6} & -\frac{1}{6} \\ \frac{1}{6} & \frac{1}{6} & -\frac{1}{6} \end{bmatrix}$. Thus,

$x = A^{\dagger}b = \begin{bmatrix} \frac{1}{6} & \frac{1}{6} & -\frac{1}{6} \\ \frac{1}{6} & \frac{1}{6} & -\frac{1}{6} \end{bmatrix} \begin{bmatrix} 2 \\ 2 \\ 2 \end{bmatrix} = \begin{bmatrix} \frac{1}{3} \\ \frac{1}{3} \end{bmatrix}$, which is the solution of minimal norm.

On the other hand, the second system which is inconsistent has no solution. However,

$x = A^{\dagger}b = \begin{bmatrix} \frac{1}{6} & \frac{1}{6} & -\frac{1}{6} \\ \frac{1}{6} & \frac{1}{6} & -\frac{1}{6} \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 2 \end{bmatrix} = \begin{bmatrix} \frac{1}{6} \\ \frac{1}{6} \end{bmatrix}$ is the "best approximation" to this solution with minimum norm.

4.3 Image Compression and SVD

4.3.1 Low-Rank Approximation from the SVD

Assume that $m > n$, for any $m \times n$ matrix, there exists another way to represent the SVD:

$$A = \begin{bmatrix} \vdots & \dots & \vdots \\ u_1 & \dots & u_m \\ \vdots & \dots & \vdots \end{bmatrix} \begin{bmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_n \\ & & 0 \\ & & \vdots \\ & & 0 \end{bmatrix} \begin{bmatrix} \dots & v_1^T & \dots \\ \vdots & \vdots & \vdots \\ \dots & v_n^T & \dots \end{bmatrix}$$

$$A = \begin{bmatrix} \vdots & \dots & \vdots \\ u_1 & \dots & u_m \\ \vdots & \dots & \vdots \end{bmatrix} \begin{bmatrix} \dots & \sigma_1 v_1^T & \dots \\ \vdots & \vdots & \vdots \\ \dots & \sigma_n v_n^T & \dots \end{bmatrix} = \sigma_1 u_1 v_1^T + \sigma_2 u_2 v_2^T + \ldots + \sigma_n u_n v_n^T$$

where $\sigma_1 \geq \sigma_2 \geq \sigma_3 \geq \ldots \geq 0$.

Therefore, in order to make **the best rank-k approximation** of A, where $k \leq \min(m,n)$, we have to minimize $\underset{A_k}{min} \|A - A_k\|$ s.t rank$(A_k) \leq k$. Then we keep the top k right singular vectors corresponding to the first k rows of V^T which is denoted as V_k^T. Similarly, we also keep the top k left singular vectors corresponding to the first k columns of U which is denoted as U_k. Last, we only keep the top k singular values and then we can get:

$$A_k = \sigma_1 u_1 v_1^T + \sigma_2 u_2 v_2^T + \ldots + \sigma_k u_k v_k^T$$

where $\sigma_1 \geq \sigma_2 \geq \sigma_3 \geq \ldots \geq 0$ and the norm of the difference between A and A_k is equal to $\|A - A_k\|_2 = \| \sum_{i=k+1}^{n} \sigma_i u_i v_i^T \| = \sigma_{k+1}$.

4.3.2 Image Compression

Images from the real world have full rank, but there are only a small number of the singular values in the SVD of images are significantly large. In image compression, a low-rank approximation of the matrix is applied to approximate images. Suppose the $m \times n$ matrix A is a 2-d image, instead of storing all m rows and n columns of the original matrix, the image is compressed by selecting r singular vectors and values. Then the compressed image A is approximated as the product of a handful of r columns and r rows which is known as the low-rank approximation.

In the image processing, insteads of using three channel of color(Red, Green, Blue), the grayscale image is preferred in which the only colors are the shades of gray. The reason is that the gray is the color in which all RGB have equal intensity and therefore rather than using three intensities to speicify each pixel it's only necessary to include a single intensity value. In addition, in the real world, grayscale intensity is stores as 8-bit int which could provide 256 different shades of gray. Therefore, given that the image capture hardwares are only able to support 8-bits images, grayscale images are very popular and sufficient for many image processings.

Example 3: There is an example to illustarte how SVD applies to the image compression.

First Step: Import required packages and images and convert the image to grayscale.

In [211]:
```
import matplotlib.pyplot as plt
%matplotlib inline
from PIL import Image
```

In [212]:
```
with Image.open("Kitten.jpg") as img:
    rgb_img = np.array(img)
print(rgb_img.shape)
plt.figure(figsize = (30, 20))
plt.title("Original Image", size = 35)
plt.imshow(rgb_img)
```

```
(3024, 4032, 3)
```

Out[212]:
```
<matplotlib.image.AxesImage at 0x1d52d7180f0>
```

In [213]:
```
A = np.sum(rgb_img, axis=-1)
print(A.shape)
plt.figure(figsize = (30, 20))
plt.title("Before Compression", size = 35)
plt.imshow(A, cmap = 'gray')
```

```
(3024, 4032)
```

Out[213]:
```
<matplotlib.image.AxesImage at 0x1d52d775ba8>
```

Second Step: Perform SVD to the grayscale image.

In [214]:
```
U, Σ, Vt = la.svd(A)
print("U: ", U.shape)
print("Σ: ", Σ.shape)
print("Vt: ", Vt.shape)
```

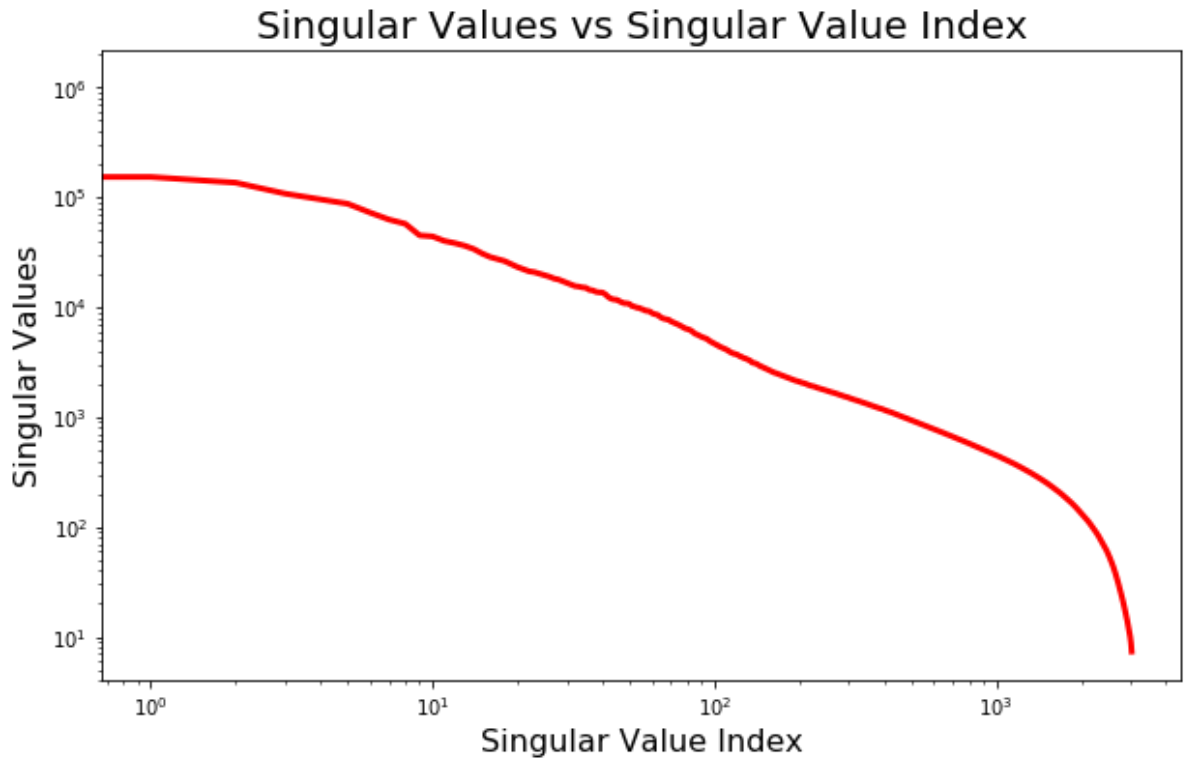
```
U:  (3024, 3024)
Σ:  (3024,)
Vt:  (4032, 4032)
```

Third Step: Analyze the relationship between singular values and its indices.

Analyze how the the singular values change with respect to the singular value index.

In [215]:
```
plt.figure(figsize = (10, 6))
plt.loglog(Σ, lw=3, color = 'red')
plt.xlabel('Singular Value Index', size = 16)
plt.ylabel('Singular Values', size = 16)
plt.title("Singular Values vs Singular Value Index", size = 20)
```

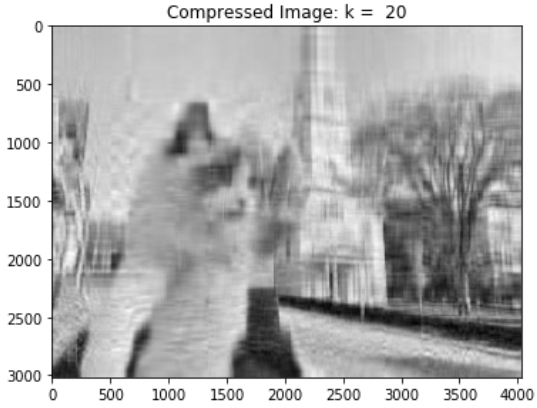
Out[215]:
```
Text(0.5, 1.0, 'Singular Values vs Singular Value Index')
```

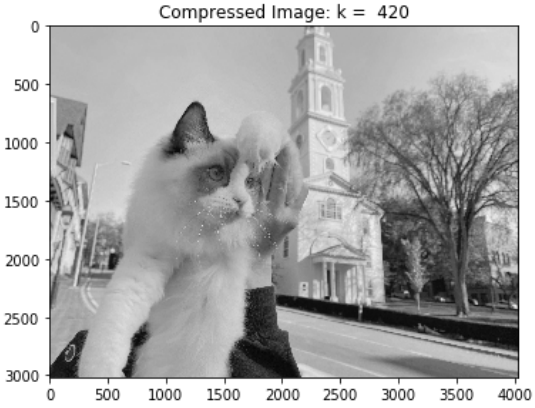


Forth Step: Select k significant singular values and vectors.

In [216]:
```
index = 1
fig = plt.figure(figsize = (15,10))
for i in range(20, 1500, 400):
    compressed_img = np.matrix(U[:, :i]) * np.diag(Σ[:i]) * np.matrix(Vt[:i
,:])
    plt.subplot(2, 2, index)
    plt.imshow(compressed_img, cmap = 'gray')
    title = "Compressed Image: k =  %s" %i
    plt.title(title)
    print("k = ", i)
    print("original size: %d" % rgb_img.size)
    compressed_size = U[:,:i].size + Σ[:i].size + Vt[:i,:].size
    print("compressed size: %d" % compressed_size)
    print("ratio: %f" % (compressed_size / rgb_img.size))
    index += 1

plt.show()
```

```
k =  20
original size: 36578304
compressed size: 141140
ratio: 0.003859
k =  420
original size: 36578304
compressed size: 2963940
ratio: 0.081030
k =  820
original size: 36578304
compressed size: 5786740
ratio: 0.158201
k =  1220
original size: 36578304
compressed size: 8609540
ratio: 0.235373
```





4.4 PCA and SVD

`Principal component analysis (PCA)` and SVD are both widely used in the machine learning and analysis of multivariate data in order to reduce dimensionality. For PCA, it's always used to reduce the feature space but keeps the most relevant and important information about features. It calculates the directions of maximum variance in high-dimensional data and projects it into a smaller dimensional subspace. PCA and SVD has a close relationship: we can use the SVD to conduct principal comonent analysis and vice versa.

Example 4: There is an example to illustarte how SVD and PCA are combined to reduce the dimension of a dataset while keeping important information about features.

First Step: Import required packages and datasets and perform EDA.

In [245]:
```
import seaborn as sns
import pandas as pd
df = sns.load_dataset("iris")
print(df.shape)
df.head()
```

```
(150, 5)
```

Out[245]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

In [246]:
```
idx = df.columns
d = {'mean':df.mean(), 'Standard Deviation': df.std()}
df1 = pd.DataFrame(data = d, index = idx)
df1
```

Out[246]:

	mean	Standard Deviation
sepal_length	5.843333	0.828066
sepal_width	3.057333	0.435866
petal_length	3.758000	1.765298
petal_width	1.199333	0.762238
species	NaN	NaN

Second Step: Standardize the dataset to zero mean and compute the svd of this dataset.

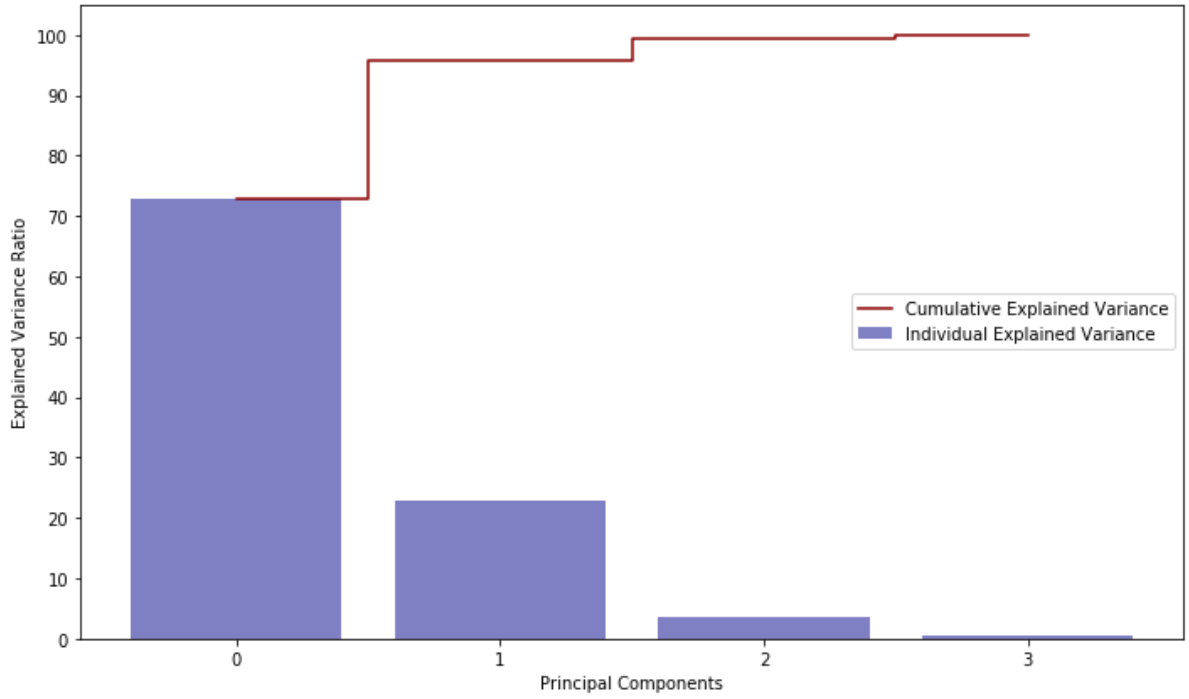
In [247]:
```
#standardize the dataset
df_shifted = df.iloc[:,:4] - df.iloc[:,:4].mean()
df_standardized = df_shifted / df.iloc[:,:4].std()

#compute the SVD
U, Σ, Vt = la.svd(df_standardized, full_matrices = False)
print("U: ", U.shape)
print("Σ: ", Σ.shape, Σ)
print("Vt: ", Vt.shape)
print("Variances: ", Σ**2)
```

```
U:  (150, 4)
Σ:  (4,) [20.85320538 11.67007028  4.6761923   1.75684679]
Vt:  (4, 4)
Variances:  [434.85617466 136.19054025  21.86677446   3.08651063]
```

Third Step: Analyze the relationship between principal component and vairances about each feature.

In [248]:
```
tot = sum(Σ**2)
variance_ratio = [(i / tot)*100 for i in Σ**2]
variance_cumulative= np.cumsum(variance_ratio)
print(variance_ratio)
print(variance_cumulative)

fig = plt.figure(figsize = (10,6))
plt.bar(range(4), variance_ratio, alpha=0.5, align='center', label='Individual
Explained Variance', color = 'darkblue')
plt.step(range(4), variance_cumulative, where='mid', label='Cumulative Explain
ed Variance', color = 'darkred')
plt.ylabel('Explained Variance Ratio')
plt.xlabel('Principal Components')
plt.xticks(np.arange(0, 4, step=1))
plt.yticks(np.arange(0, 101, step=10))
plt.legend()
plt.tight_layout()
```

```
[72.9624454132999, 22.85076178670173, 3.668921889282877, 0.5178709107154799]
[ 72.96244541  95.8132072   99.48212909 100.        ]
```



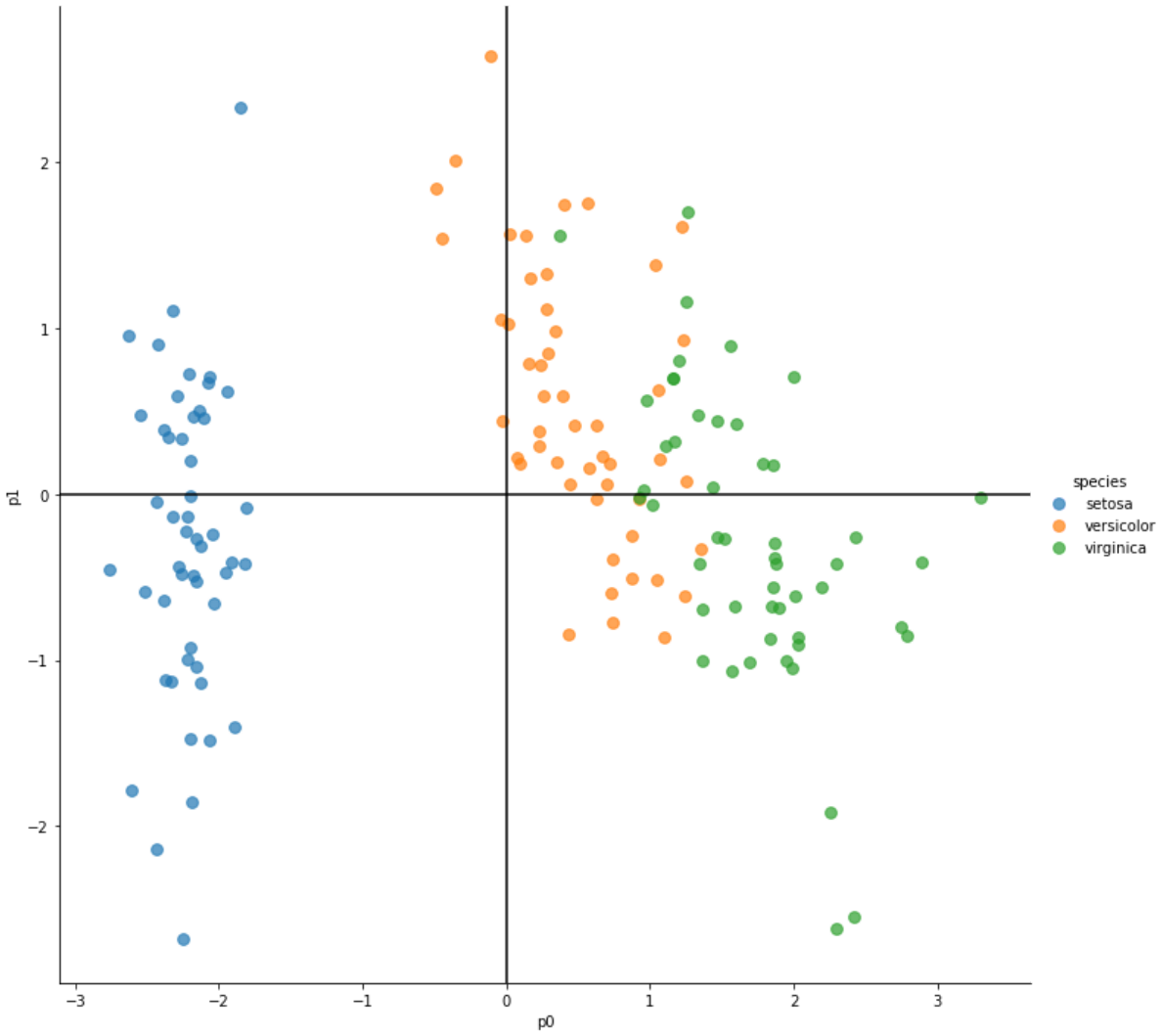
This plot shows that around 73% of the variance can be explained by the first principal component. Around 95.8% of the variance can be explained by the first two principal components. Therefore, the remaining principal components contains few information about features which could be ignored.

Forth Step: Perform PCA.

In [262]:
```
V = Vt.T
Zstar = df_standardized@V[:,:2]

fig = plt.figure(figsize = (10,6))
df['p0'] = Zstar.iloc[:,0]
df['p1'] = Zstar.iloc[:,1]
fig = plt.figure(figsize = (10,6))
g1 = sns.lmplot('p0', 'p1', df, hue='species', fit_reg=False, size=10, scatter
_kws={'alpha':0.7,'s':60})
ax = g1.axes[0,0]
ax.axhline(y=0, color='k')
ax.axvline(x=0, color='k')
```

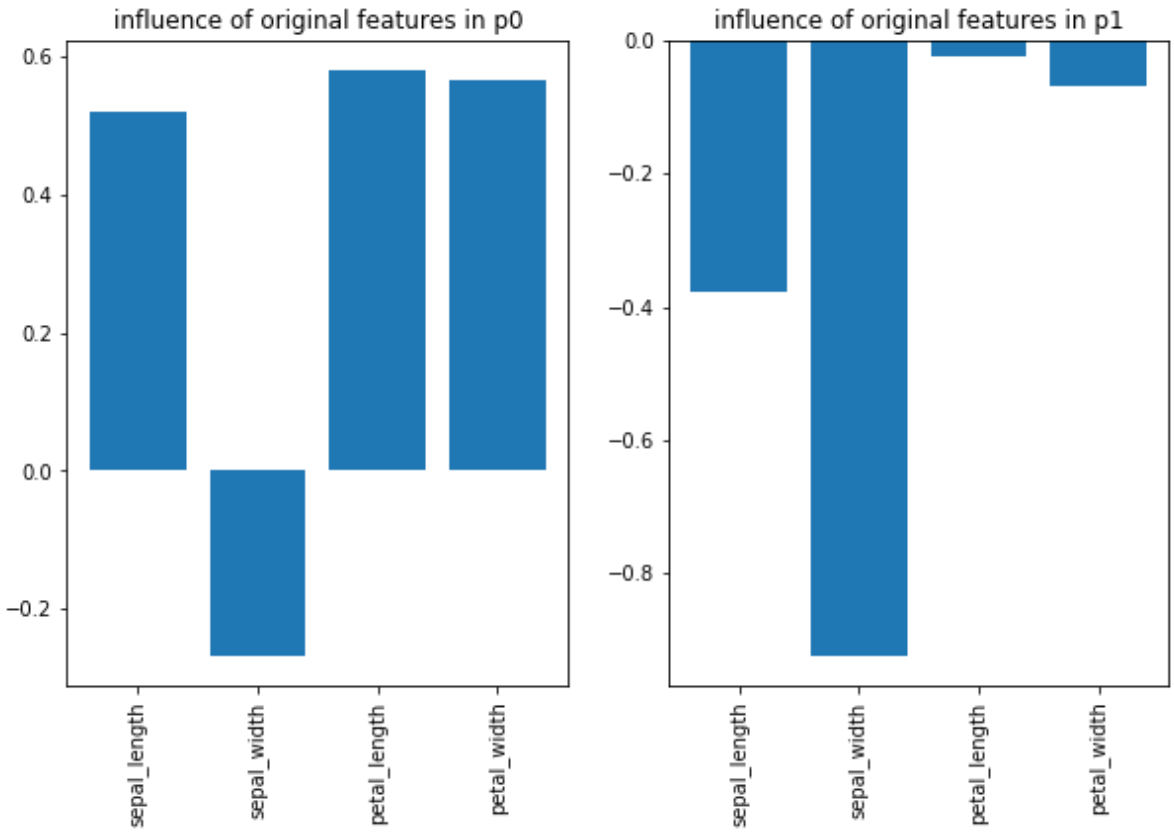
```
D:\Anaconda\lib\site-packages\seaborn\regression.py:546: UserWarning: The `si
ze` paramter has been renamed to `height`; please update your code.
  warnings.warn(msg, UserWarning)
```

Out[262]:
```
<matplotlib.lines.Line2D at 0x1d5314bc908>

<Figure size 720x432 with 0 Axes>

<Figure size 720x432 with 0 Axes>
```



In [266]:
```
plt.figure(figsize = (10,6))
plt.subplot(1, 2, 1)
plt.bar(df.columns[:4],V[:,0])
plt.xticks(rotation=90)
plt.title('influence of original features in p0')

plt.subplot(1, 2, 2)
plt.bar(df.columns[:4],V[:,1])
plt.xticks(rotation=90)
plt.title('influence of original features in p1')
```

Out[266]: Text(0.5, 1.0, 'influence of original features in p1')



REFERENCE

- CS357: Spring 2019. (n.d.). Retrieved November 21, 2020, from https://relate.cs.illinois.edu/course/cs357-s19/page/demos/ (https://relate.cs.illinois.edu/course/cs357-s19/page/demos/)

- Singular value decomposition. (2020, November 09). Retrieved November 21, 2020, from https://en.wikipedia.org/wiki/Singular_value_decomposition (https://en.wikipedia.org/wiki/Singular_value_decomposition)

- John. (2018, May 05). Computing SVD and pseudoinverse. Retrieved November 21, 2020, from https://www.johndcook.com/blog/2018/05/05/svd/ (https://www.johndcook.com/blog/2018/05/05/svd/)

- Grayscale Images. (n.d.). Retrieved November 21, 2020, from https://homepages.inf.ed.ac.uk/rbf/HIPR2/gryimage.htm (https://homepages.inf.ed.ac.uk/rbf/HIPR2/gryimage.htm)

- Putalapattu, R. (2017, August 20). Jupyter, python, Image compression and svd - An interactive exploration. Retrieved November 21, 2020, from https://medium.com/@rameshputalapattu/jupyter-python-image-compression-and-svd-an-interactive-exploration-703c953e44f6 (https://medium.com/@rameshputalapattu/jupyter-python-image-compression-and-svd-an-interactive-exploration-703c953e44f6)

- Wang, Z. (2019, September 05). PCA and SVD explained with numpy. Retrieved November 21, 2020, from https://towardsdatascience.com/pca-and-svd-explained-with-numpy-5d13b0d2a4d8 (https://towardsdatascience.com/pca-and-svd-explained-with-numpy-5d13b0d2a4d8)