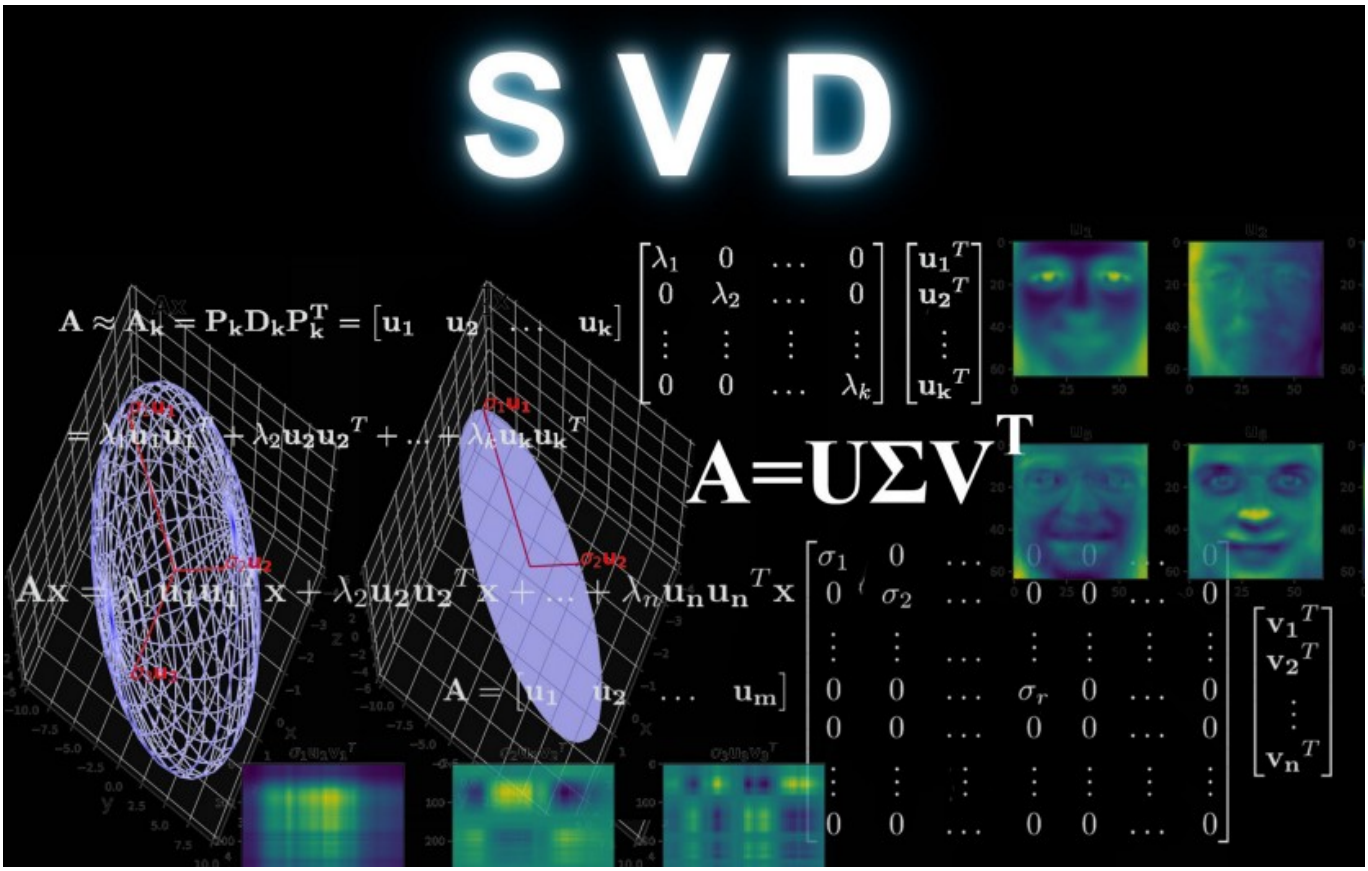
Understanding Singular Value Decomposition and its Application in Data Science

Learn about the intuition behind Singular Value Decomposition



In linear algebra, the Singular Value Decomposition (SVD) of a matrix is a factorization of that matrix into three matrices. It has some interesting algebraic properties and conveys important geometrical and theoretical insights about linear transformations. It also has some important applications in data science. In this article, I will try to explain the mathematical intuition behind SVD and its geometrical meaning. Instead of manual calculations, I will use the Python libraries to do the calculations and later give you some examples of using SVD in data science applications. In this article, bold-face lower-case letters (like **a**) refer to vectors. Bold-face capital letters (like **A**) refer to matrices, and italic lower-case letters (like *a*) refer to scalars.

To understand SVD we need to first understand the *Eigenvalue Decomposition* of a matrix. We can think of a matrix **A** as a transformation that acts on a vector **x** by multiplication to produce a new vector **Ax**. We use [**A**]ij or *aij* to denote the element of matrix **A** at row *i* and column *j*. If **A** is an $m \times p$ matrix and **B** is a $p \times n$ matrix, the matrix product **C**=**AB** (which is an $m \times n$ *matrix*) is defined as:

$$[\mathbf{C}]_{ij} = c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}$$

For example, the rotation matrix in a 2-d space can be defined as:

$$\mathbf{A} = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix}$$

This matrix rotates a vector about the origin by the angle θ (with counterclockwise rotation for a positive θ). Another example is the stretching matrix **B** in a 2-d space which is defined as:

$$\mathbf{B} = \begin{bmatrix} k & 0 \\ 0 & 1 \end{bmatrix}$$

This matrix stretches a vector along the *x*-axis by a constant factor *k* but does not affect it in the *y*-direction. Similarly, we can have a stretching matrix in *y*-direction:

$$\mathbf{C} = \begin{bmatrix} 1 & 0 \\ 0 & k \end{bmatrix}$$

As an example, if we have a vector

$$\mathbf{x} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

then **y=Ax** is the vector which results after rotation of **x** by θ, and **Bx** is a vector which is the result of stretching **x** in the *x*-direction by a constant factor *k*.

Listing 1 shows how these matrices can be applied to a vector **x** and visualized in Python. We can use the NumPy arrays as vectors and matrices.

```
# Listing 1

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import colors
import math as mt
```

```
from numpy import linalg as LA
from mpl_toolkits.mplot3d import Axes3D
from sklearn.datasets import fetch_olivetti_faces
%matplotlib inline

x=np.array([1,0]) # Original vector
theta = 30 * mt.pi / 180 # 30 degress in radian
A = np.array([[np.cos(theta), -np.sin(theta)],[np.sin(theta), np.cos(theta)]]) # Rotation matrix
B = np.array([[3,0],[0,1]]) # Stretching matrix

Ax = A @ x  # y1 is the rotated vector
Bx = B @ x  # y2 is the stretched vector

# Reshaping and storing both x and Ax in t1 to be plotted as vectors
t1 = np.concatenate([x.reshape(1,2), Ax.reshape(1,2)])
# Reshaping and storing both x and Bx in t2 to be plotted as vectors
t2 = np.concatenate([x.reshape(1,2), Bx.reshape(1,2)])
origin = [0], [0] # origin point

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10,15))
plt.subplots_adjust(wspace=0.4)

# Plotting t1
ax1.quiver(*origin, t1[:,0], t1[:,1], color=['b', 'g'], width=0.013, angles='xy', scale_units='x
ax1.set_xlabel('x', fontsize=14)
ax1.set_ylabel('y', fontsize=14)
ax1.set_xlim([-0.5,1.5])
ax1.set_ylim([-0.5,1])
ax1.set_aspect('equal')
ax1.grid(True)
ax1.set_axisbelow(True)
ax1.set_title("Rotation transform")
ax1.axhline(y=0, color='k')
ax1.axvline(x=0, color='k')
ax1.text(1, 0.1, "$\mathbf{x}$", fontsize=16)
ax1.text(0.8, 0.6, "$\mathbf{Ax}$", fontsize=16)

# Plotting t2
ax2.quiver(*origin, t2[:,0], t2[:,1], color=['b', 'g'], width=0.013, angles='xy', scale_units='x
ax2.set_xlabel('x', fontsize=14)
ax2.set_ylabel('y', fontsize=14)
ax2.set_xlim([-0.5,3.5])
ax2.set_ylim([-1.5,1.5])
ax2.set_aspect('equal')
ax2.grid(True)
ax2.set_axisbelow(True)
ax2.set_title("Stretching transform")
ax2.axhline(y=0, color='k')
```

```
55  ax2.axvline(x=0, color='k')
56  ax2.text(1, 0.2, "$\mathbf{x}$", fontsize=16)
57  ax2.text(3, 0.2, "$\mathbf{Bx}$", fontsize=16)
58
59  plt.show()
```

Here the rotation matrix is calculated for $\theta=30^0$ and in the stretching matrix $k=3$. **y** is the transformed vector of **x**. To plot the vectors, the `quiver()` function in `matplotlib` has been used. Figure 1 shows the output of the code.

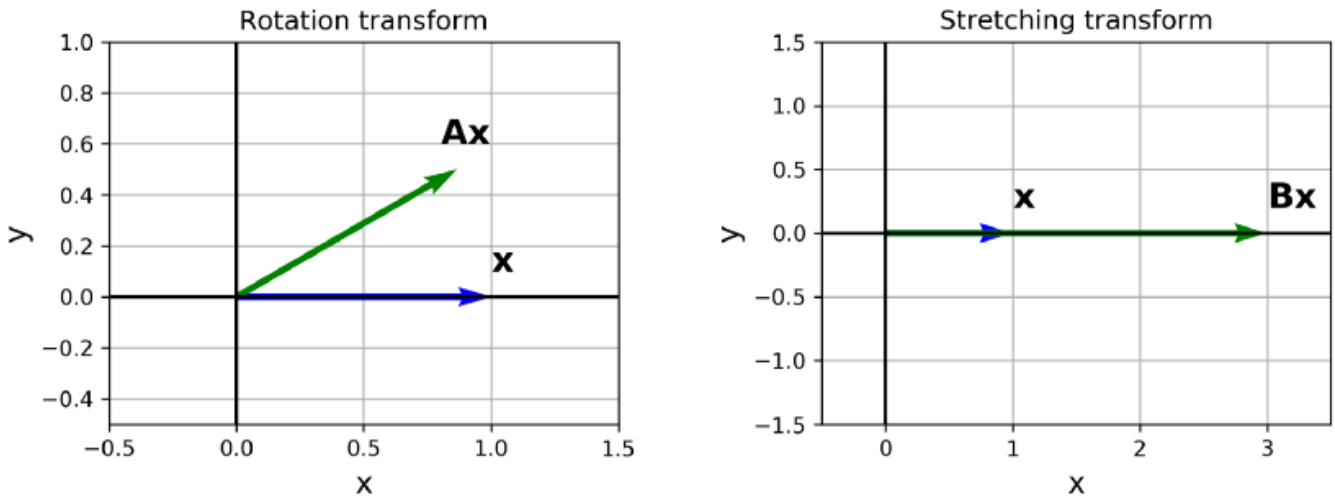


Figure 1

The matrices are represented by a 2-d array in NumPy. We can use the `np.matmul(a,b)` function to the multiply matrix `a` by `b` However, it is easier to use the `@` operator to do that. The vectors can be represented either by a 1-d array or a 2-d array with a shape of `(1,n)` which is a row vector or `(n,1)` which is a column vector.

Now we are going to try a different transformation matrix. Suppose that

$$\mathbf{A} = \begin{bmatrix} 3 & 2 \\ 0 & 2 \end{bmatrix}$$

However, we don't apply it to just one vector. Initially, we have a circle that contains all the vectors that are one unit away from the origin. These vectors have the general form of

$$\mathbf{x} = \begin{bmatrix} x_i \\ y_i \end{bmatrix} \quad where \;\; x_i^2 + y_i^2 = 1$$

$$\begin{bmatrix} y_n \end{bmatrix}$$

Now we calculate $\mathbf{t}=\mathbf{Ax}$. So $\mathbf{t}$ is the set of all the vectors in $\mathbf{x}$ which have been transformed by $\mathbf{A}$. Listing 2 shows how this can be done in Python.

```
# Listing 2

# Creating the vectors for a circle and storing them in x
xi1 = np.linspace(-1.0, 1.0, 100)
xi2 = np.linspace(1.0, -1.0, 100)
yi1 = np.sqrt(1 - xi1**2)
yi2 = -np.sqrt(1 - xi2**2)

xi = np.concatenate((xi1, xi2),axis=0)
yi = np.concatenate((yi1, yi2),axis=0)
x = np.vstack((xi, yi))

# getting a sample vector from x
x_sample1 = x[:, 65]
x_sample2 = x[:, 100]

A = np.array([[3, 2],
              [0, 2]])

t = A @ x  # Vectors in t are the transformed vectors of x

t_sample1 = t[:, 65]
t_sample2 = t[:, 100]
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10,15))

plt.subplots_adjust(wspace=0.4)

# Plotting x
ax1.plot(x[0,:], x[1,:], color='b')
ax1.quiver(*origin, x_sample1[0], x_sample1[1], color=['b'], width=0.012, angles='xy', scale_uni
ax1.quiver(*origin, x_sample2[0], x_sample2[1], color=['r'], width=0.012, angles='xy', scale_uni
ax1.set_xlabel('x', fontsize=14)
ax1.set_ylabel('y', fontsize=14)
ax1.set_xlim([-4,4])
ax1.set_ylim([-4,4])
ax1.set_aspect('equal')
ax1.grid(True)
ax1.set_axisbelow(True)
ax1.set_title("Original vectors")
ax1.axhline(y=0, color='k')
ax1.axvline(x=0, color='k')
ax1.text(0.3, 1.2, "$\mathbf{x_1}$", color='b', fontsize=14)
```

```
ax1.text(1.2, 0.2, "$\mathbf{x_2}$", color='r', fontsize=14)

# Plotting t
ax2.plot(t[0, :], t[1, :], color='b')
ax2.quiver(*origin, t_sample1[0], t_sample1[1], color=['b'], width=0.012, angles='xy', scale_uni
ax2.quiver(*origin, t_sample2[0], t_sample2[1], color=['r'], width=0.012, angles='xy', scale_uni

ax2.set_xlabel('x', fontsize=14)
ax2.set_ylabel('y', fontsize=14)
ax2.set_xlim([-4,4])
ax2.set_ylim([-4,4])
ax2.set_aspect('equal')
ax2.grid(True)
ax2.set_axisbelow(True)
ax2.set_title("New vectors after transformation")
ax2.axhline(y=0, color='k')
ax2.axvline(x=0, color='k')
ax2.text(2.5, 2.3, "$\mathbf{t_1}$", color='b', fontsize=14)
ax2.text(2.6, 0.4, "$\mathbf{t_2}$", color='r', fontsize=14)
plt.savefig('2.png', dpi=300, bbox_inches='tight')

plt.show()
```

Figure 2 shows the plots of **x** and **t** and the effect of transformation on two sample vectors **x1** and **x2** in **x**.

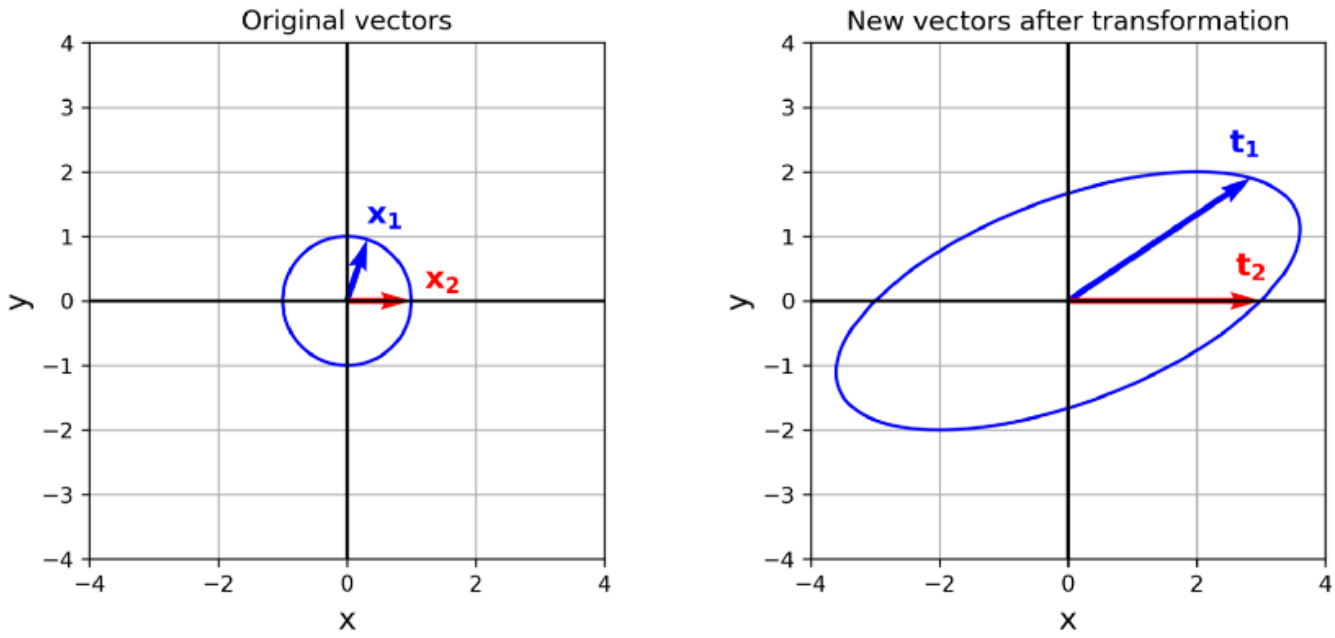


Figure 2

The initial vectors (**x**) on the left side form a circle as mentioned before, but the transformation matrix somehow changes this circle and turns it into an ellipse.

The sample vectors **x1** and **x2** in the circle are transformed into **t1** and **t2** respectively. So:

$$\mathbf{t_1} = \mathbf{A}\mathbf{x_1} \qquad \mathbf{t_2} = \mathbf{A}\mathbf{x_2}$$

Eigenvalues and Eigenvectors

A vector is a quantity which has both magnitude and direction. The general effect of matrix **A** on the vectors in **x** is a combination of rotation and stretching. For example, it changes both the direction and magnitude of the vector **x1** to give the transformed vector **t1**. However, for vector **x2** only the magnitude changes after transformation. In fact, **x2** and **t2** have the same direction. Matrix **A** only stretches **x2** in the same direction and gives the vector **t2** which has a bigger magnitude. The only way to change the magnitude of a vector without changing its direction is by multiplying it with a scalar. So if we have a vector $\mathbf{u}$, and λ is a scalar quantity then $\lambda\mathbf{u}$ has the same direction and a different magnitude. So for a vector like **x2** in figure 2, the effect of multiplying by **A** is like multiplying it with a scalar quantity like λ.

$$\mathbf{t_2} = \mathbf{A}\mathbf{x_2} = \lambda\mathbf{x_2}$$

This is not true for all the vectors in **x**. In fact, for each matrix **A,** only some of the vectors have this property. These special vectors are called the eigenvectors of **A** and their corresponding scalar quantity λ is called an eigenvalue of **A** for that eigenvector. So the eigenvector of an $n \times n$ matrix **A** is defined as a nonzero vector $\mathbf{u}$ such that:

$$\mathbf{A}\mathbf{u} = \lambda\mathbf{u}$$

where λ is a scalar and is called the eigenvalue of **A**, and $\mathbf{u}$ is the eigenvector corresponding to λ. In addition, if you have any other vectors in the form of $a\mathbf{u}$ where a is a scalar, then by placing it in the previous equation we get:

$$\mathbf{A}(a\mathbf{u}) = a\mathbf{A}\mathbf{u} = a\lambda\mathbf{u} = \lambda(a\mathbf{u})$$

which means that any vector which has the same direction as the eigenvector $\mathbf{u}$ (or the opposite direction if a is negative) is also an eigenvector with the same corresponding

eigenvalue.

For example, the eigenvalues of

$$\mathbf{B} = \begin{bmatrix} -1 & 1 \\ 0 & -2 \end{bmatrix}$$

are $\lambda 1 = -1$ and $\lambda 2 = -2$ and their corresponding eigenvectors are:

$$\mathbf{u_1} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad \mathbf{u_2} = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

and we have:

$$\mathbf{Bu_1} = \lambda_1 \mathbf{u_1} \qquad \mathbf{Bu_2} = \lambda_2 \mathbf{u_2}$$

This means that when we apply matrix **B** to all the possible vectors, it does not change the direction of these two vectors (or any vectors which have the same or opposite direction) and only stretches them. So for the eigenvectors, the matrix multiplication turns into a simple scalar multiplication. Here I am not going to explain how the eigenvalues and eigenvectors can be calculated mathematically. Instead, I will show you how they can be obtained in Python.

```
# Listing 3
B = np.array([[-1, 1],[0, -2]])
lam, u = LA.eig(B)
print("lam=", np.round(lam, 4))
print("u=", np.round(u, 4))
```

SVD_Listing3.py hosted with ♡ by GitHub view raw

We can use the `LA.eig()` function in NumPy to calculate the eigenvalues and eigenvectors. It returns a tuple. The first element of this tuple is an array that stores the eigenvalues, and the second element is a 2-d array that stores the corresponding eigenvectors. In fact, in Listing 3 the column `u[:,i]` is the eigenvector corresponding to the eigenvalue `lam[i]`. Now if we check the output of Listing 3, we get:

```
lam= [-1. -2.]
```

```
u= [[ 1.      -0.7071]
    [ 0.       0.7071]]
```

You may have noticed that the eigenvector for λ=*-1* is the same as **u1**, but the other one is different. That is because `LA.eig()` returns the normalized eigenvector. A normalized vector is a unit vector whose length is 1. But before explaining how the length can be calculated, we need to get familiar with the transpose of a matrix and the dot product.

Transpose

The transpose of the column vector **u (**which is shown by **u** superscript T) is the row vector of **u** (in this article sometimes I show it as **u**^T). The transpose of an $m \times n$ matrix **A** is an $n \times m$ matrix whose columns are formed from the corresponding rows of **A**. For example if we have

$$\mathbf{C} = \begin{bmatrix} 5 & 4 & 2 \\ 7 & 1 & 9 \end{bmatrix}$$

then the transpose of **C** is:

$$\mathbf{C^T} = \begin{bmatrix} 5 & 7 \\ 4 & 1 \\ 2 & 9 \end{bmatrix}$$

So the transpose of a row vector becomes a column vector with the same elements and vice versa. In fact, the element in the *i*-th row and *j*-th column of the transposed matrix is equal to the element in the *j*-th row and *i*-th column of the original matrix. So

$$[\mathbf{A^T}]_{ij} = [\mathbf{A}]_{ji}$$

In NumPy you can use the `transpose()` method to calculate the transpose. For example to calculate the transpose of matrix `C` we write `C.transpose().` We can also use the transpose attribute `T`, and write `C.T` to get its transpose. The transpose has some important properties. First, the transpose of the transpose of **A** is **A**. So:

$$(\mathbf{A^T})^\mathbf{T} = \mathbf{A}$$

In addition, the transpose of a product is the product of the transposes in the reverse order.

$$(\mathbf{AB})^T = \mathbf{B}^T\mathbf{A}^T$$

To prove it remember the matrix multiplication definition:

$$[\mathbf{AB}]_{ij} = \sum_{k=1}^{p} a_{ik}b_{kj}$$

and based on the definition of matrix transpose, the left side is:

$$[(\mathbf{AB})^T]_{ij} = [\mathbf{AB}]_{ji} = \sum_{k=1}^{p} a_{jk}b_{ki}$$

and the right side is

$$[\mathbf{B^T A^T}]_{ij} = \sum_{k=1}^{p} [\mathbf{B^T}]_{ik}[\mathbf{A^T}]_{kj} = \sum_{k=1}^{p} [\mathbf{B}]_{ki}[\mathbf{A}]_{jk} = \sum_{k=1}^{p} a_{jk}b_{ki}$$

so both sides of the equation are equal.

Dot product

If we have two vectors **u** and **v**:

$$\mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix} \qquad \mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

The dot product (or inner product) of these vectors is defined as the transpose of **u** multiplied by **v**:

$$\mathbf{u}.\mathbf{v} = \mathbf{u}^T\mathbf{v} = [u_1 u_2 \cdots u_n] \begin{bmatrix} v_1 \\ v_2 \\ \vdots \end{bmatrix} = u_1v_1 + u_2v_2 + \cdots + u_nv_n$$

$$\begin{bmatrix} \\ \vdots \\ v_n \end{bmatrix}$$

Based on this definition the dot product is commutative so:

$$\mathbf{u}.\mathbf{v} = \mathbf{v}.\mathbf{u}$$

Partitioned matrix

When calculating the transpose of a matrix, it is usually useful to show it as a partitioned matrix. For example, the matrix

$$\mathbf{C} = \begin{bmatrix} 5 & 4 & 2 \\ 7 & 1 & 9 \end{bmatrix}$$

can be also written as:

$$\mathbf{C} = \begin{bmatrix} \mathbf{u1} & \mathbf{u2} & \mathbf{u3} \end{bmatrix}$$

where

$$\mathbf{u_1} = \begin{bmatrix} 5 \\ 7 \end{bmatrix} \quad \mathbf{u_2} = \begin{bmatrix} 4 \\ 1 \end{bmatrix} \quad \mathbf{u_3} = \begin{bmatrix} 2 \\ 9 \end{bmatrix}$$

So we can think of each column of **C** as a column vector, and **C** can be thought of as a matrix with just one row. Now to write the transpose of **C**, we can simply turn this row into a column, similar to what we do for a row vector. The only difference is that each element in **C** is now a vector itself and should be transposed too.

Now we know that

$$\mathbf{u_1^T} = \begin{bmatrix} 5 & 7 \end{bmatrix} \qquad \mathbf{u_2^T} = \begin{bmatrix} 4 & 1 \end{bmatrix} \qquad \mathbf{u_3^T} = \begin{bmatrix} 2 & 9 \end{bmatrix}$$

So:

$$\mathbf{C^T} = \begin{bmatrix} \mathbf{u_1}^T \\ \mathbf{u_2}^T \\ \mathbf{u_3}^T \end{bmatrix} = \begin{bmatrix} 5 & 7 \\ 4 & 1 \\ 2 & 9 \end{bmatrix}$$

Now each row of the **C**^T is the transpose of the corresponding column of the original matrix **C**.

Now let matrix **A** be a partitioned column matrix and matrix **B** be a partitioned row matrix:

$$\mathbf{A} = \begin{bmatrix} \mathbf{a_1} & \mathbf{a_2} & \dots & \mathbf{a_p} \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} \mathbf{b_1}^T \\ \mathbf{b_2}^T \\ \vdots \\ \mathbf{b_p}^T \end{bmatrix}$$

where each column vector **ai** is defined as the *i*-th column of **A**:

$$\mathbf{a_i} = \begin{bmatrix} a_{1i} \\ a_{2i} \\ \vdots \\ a_{mi} \end{bmatrix}$$

Here for each element, the first subscript refers to the row number and the second subscript to the column number. So **A** is an $m \times p$ matrix. In addition, **B** is a $p \times n$ matrix where each row vector in **bi**^T is the *i*-th row of **B**:

$$\mathbf{b_i}^T = \begin{bmatrix} b_{i1} & b_{i2} & \dots & b_{in} \end{bmatrix}$$

Again, the first subscript refers to the row number and the second subscript to the column number. Please note that by convection, a vector is written as a column vector. So to write a row vector, we write it as the transpose of a column vector. So **bi** is a column vector, and its transpose is a row vector that captures the *i*-th row of **B**. Now we can calculate **AB**:

$$\mathbf{C} = \mathbf{AB} = \begin{bmatrix} \mathbf{a_1} & \mathbf{a_2} & \dots & \mathbf{a_p} \end{bmatrix} \begin{bmatrix} \mathbf{b_1}^T \\ \mathbf{b_2}^T \\ \vdots \\ \mathbf{b_p}^T \end{bmatrix} =$$

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1p} \\ a_{21} & a_{22} & \dots & a_{2p} \\ . & . & . & . \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ . & . & . & . \end{bmatrix} =$$

$$\begin{bmatrix} \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mp} \end{bmatrix} \begin{bmatrix} \vdots & \vdots & \vdots & \vdots \\ b_{p1} & b_{p2} & \dots & b_{pn} \end{bmatrix}$$

$$\begin{bmatrix} \sum_{i=1}^{p} a_{1i}b_{i1} & \sum_{i=1}^{p} a_{1i}b_{i2} & \dots & \sum_{i=1}^{p} a_{1i}b_{in} \\ \sum_{i=1}^{p} a_{2i}b_{i1} & \sum_{i=1}^{p} a_{2i}b_{i2} & \dots & \sum_{i=1}^{p} a_{2i}b_{in} \\ \vdots & \vdots & \vdots & \vdots \\ \sum_{i=1}^{p} a_{mi}b_{i1} & \sum_{i=1}^{p} a_{mi}b_{i2} & \dots & \sum_{i=1}^{p} a_{mi}b_{in} \end{bmatrix} =$$

$$\begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} & \dots & a_{11}b_{1n} \\ a_{21}b_{11} & a_{21}b_{12} & \dots & a_{21}b_{1n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1}b_{11} & a_{m1}b_{12} & \dots & a_{m1}b_{1n} \end{bmatrix} + \begin{bmatrix} a_{12}b_{21} & a_{21}b_{22} & \dots & a_{21}b_{2n} \\ a_{22}b_{21} & a_{22}b_{22} & \dots & a_{22}b_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m2}b_{21} & a_{m2}b_{22} & \dots & a_{m2}b_{2n} \end{bmatrix} +$$

$$\dots + \begin{bmatrix} a_{1p}b_{p1} & a_{1p}b_{p2} & \dots & a_{1p}b_{pn} \\ a_{2p}b_{p1} & a_{2n}b_{n2} & \dots & a_{2p}b_{pn} \\ \vdots & \vdots & \vdots & \vdots \\ a_{mp}b_{p1} & a_{mp}b_{p2} & \dots & a_{mp}b_{pn} \end{bmatrix} =$$

$$\mathbf{a_1}\mathbf{b_1}^T + \mathbf{a_2}\mathbf{b_2}^T + \dots + \mathbf{a_p}\mathbf{b_p}^T$$

so the product of the *i*-th *column* of **A** and the *i*-th row of **B** gives an $m \times n$ matrix, and all these matrices are added together to give **AB** which is also an $m \times n$ matrix. In fact, we can simply assume that we are multiplying a row vector **A** by a column vector **B**. As a special case, suppose that **x** is a column vector. Now we can calculate **Ax** similarly:

$$\mathbf{Ax} = \begin{bmatrix} \mathbf{a_1} & \mathbf{a_2} & \dots & \mathbf{a_p} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \end{bmatrix} =$$

$$x_1\mathbf{a_1} + x_2\mathbf{a_2} + \dots + x_p\mathbf{a_p}$$

So **Ax** is simply a linear combination of the columns of **A**.

To calculate the dot product of two vectors `a` and `b` in NumPy, we can write `np.dot(a,b)` if both are 1-d arrays, or simply use the definition of the dot product and write `a.T @ b` .

Now that we are familiar with the transpose and dot product, we can define the length (also called the 2-norm) of the vector **u** as:

$$||\mathbf{u}|| = \sqrt{\mathbf{u}.\mathbf{u}} = \sqrt{\mathbf{u}^T\mathbf{u}} = \sqrt{u_1^2 + u_2^2 + \dots + u_n^2}$$

To normalize a vector **u**, we simply divide it by its length to have the normalized vector **n**:

$$\mathbf{n} = \frac{\mathbf{u}}{||\mathbf{u}||}$$

The normalized vector **n** is still in the same direction of **u,** but its length is 1. Now we can normalize the eigenvector of λ=-2 that we saw before:

$$\mathbf{u_2} = \begin{bmatrix} -1 \\ 1 \end{bmatrix} \qquad ||\mathbf{u_2}|| = \sqrt{(-1)^2 + 1^2} = \sqrt{2}$$

$$\mathbf{n} = \frac{\mathbf{u_2}}{||\mathbf{u_2}||} = \begin{bmatrix} \frac{-1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \approx \begin{bmatrix} -0.7071 \\ 0.7071 \end{bmatrix}$$

which is the same as the output of Listing 3. As shown before, if you multiply (or divide) an eigenvector by a constant, the new vector is still an eigenvector for the same eigenvalue, so by normalizing an eigenvector corresponding to an eigenvalue, you still have an eigenvector for that eigenvalue.

But why eigenvectors are important to us? As mentioned before an eigenvector simplifies the matrix multiplication into a scalar multiplication. In addition, they have some more interesting properties. Let me go back to matrix **A** that was used in Listing 2 and calculate its eigenvectors:

$$\mathbf{A} = \begin{bmatrix} 3 & 2 \\ 0 & 2 \end{bmatrix}$$

As you remember this matrix transformed a set of vectors forming a circle into a new set forming an ellipse (Figure 2). We will use `LA.eig()` to calculate the eigenvectors in Listing 4.

```
# Listing 4
A = np.array([[3, 2],
              [0, 2]])
lam, u = LA.eig(A)
print("lam=", np.round(lam, 4))
print("u=", np.round(u, 4))
```

SVD_Listing4.py hosted with ♡ by GitHub view raw

The output is :

```
lam= [3. 2.]
u= [[ 1.      -0.8944]
    [ 0.       0.4472]]
```

So we have two eigenvectors:

$$\mathbf{u_1} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad \mathbf{u_2} = \begin{bmatrix} -0.8944 \\ 0.4472 \end{bmatrix}$$

and the corresponding eigenvalues are:

$$\lambda_1 = 3 \quad \lambda_2 = 2$$

Now we plot the eigenvectors on top of the transformed vectors:

```
# Listing 5

t = A @ x   # Vectors in t are the transformed vectors of x

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10,15))

plt.subplots_adjust(wspace=0.4)

# Plotting x
ax1.plot(x[0,:], x[1,:], color='b')
ax1.quiver(*origin, u[0,:], u[1,:], color=['b'], width=0.012, angles='xy', scale_units='xy', sca
ax1.set_xlabel('x', fontsize=14)
ax1.set_ylabel('y', fontsize=14)
ax1.set_xlim([-4,4])
ax1.set_ylim([-4,4])
ax1.set_aspect('equal')
ax1.grid(True)
ax1.set_title("Original vectors")
ax1.axhline(y=0, color='k')
ax1.axvline(x=0, color='k')
ax1.text(1, 0.3, "$\mathbf{u_1}$", fontsize=14)
ax1.text(-1.6, 0.5, "$\mathbf{u_2}$", fontsize=14)
```

```
ax1.text(0.3, 1.3, "$\mathbf{x}$", color='b', fontsize=14)

# Plotting t
ax2.plot(t[0, :], t[1, :], color='b')
ax2.quiver(*origin, u[0,:], u[1,:], color=['b'], width=0.012, angles='xy', scale_units='xy', sca
ax2.set_xlabel('x', fontsize=14)
ax2.set_ylabel('y', fontsize=14)
ax2.set_xlim([-4,4])
ax2.set_ylim([-4,4])
ax2.set_aspect('equal')
ax2.grid(True)
ax2.set_title("New vectors after transformation")
ax2.axhline(y=0, color='k')
ax2.axvline(x=0, color='k')
ax2.text(1, 0.3, "$\mathbf{u_1}$", fontsize=14)
ax2.text(-1.6, 0.5, "$\mathbf{u_2}$", fontsize=14)
ax2.text(2, 2.3, "$\mathbf{Ax}$", color='b', fontsize=14)

plt.show()
```

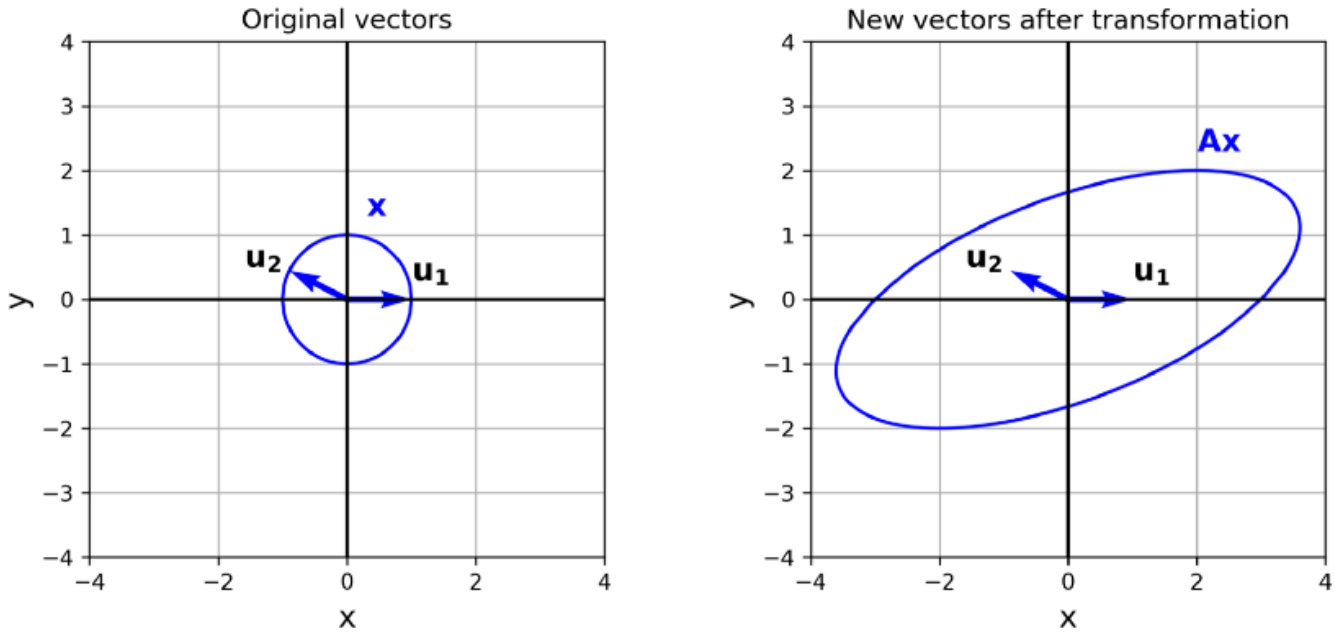


Figure 3

There is nothing special about these eigenvectors in Figure 3. Now let me try another matrix:

$$\mathbf{B} = \begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix}$$

Here we have two eigenvectors:

$$\mathbf{u_1} = \begin{bmatrix} 0.8507 \\ 0.5257 \end{bmatrix} \qquad \mathbf{u_2} = \begin{bmatrix} -0.5257 \\ 0.8507 \end{bmatrix}$$

and the corresponding eigenvalues are:

$$\lambda_1 = 3.618 \qquad \lambda_2 = 1.382$$

Now we can plot the eigenvectors on top of the transformed vectors by replacing this new matrix in Listing 5. The result is shown in Figure 4.

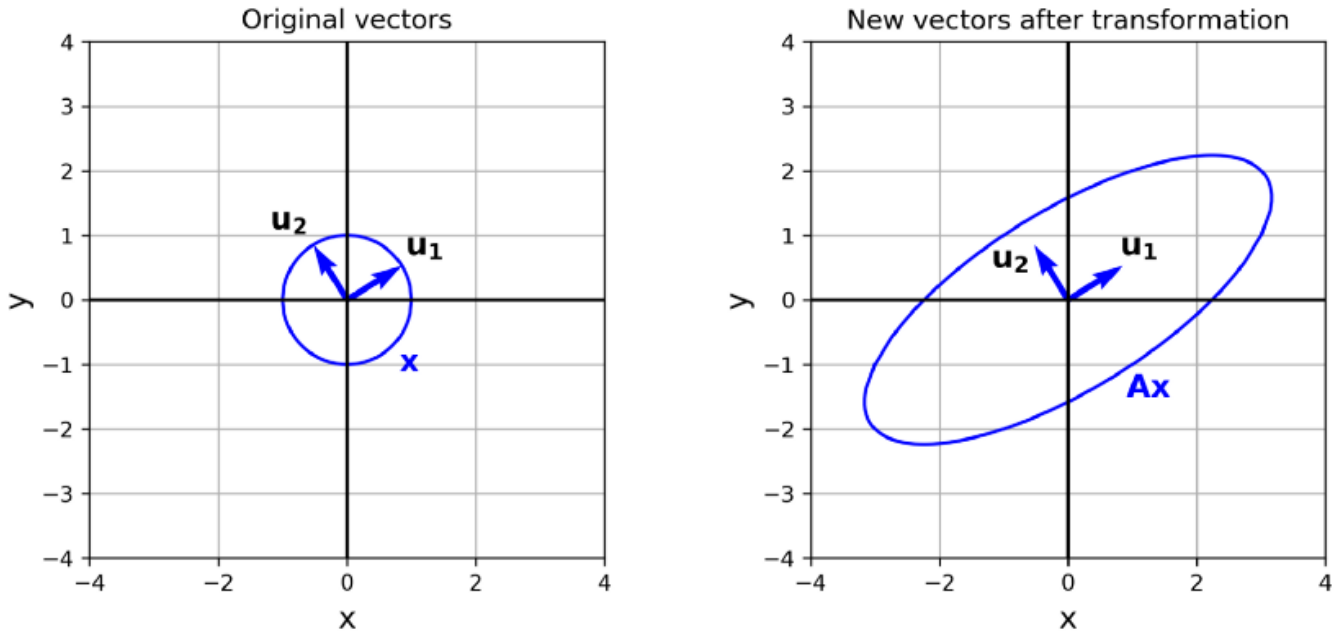
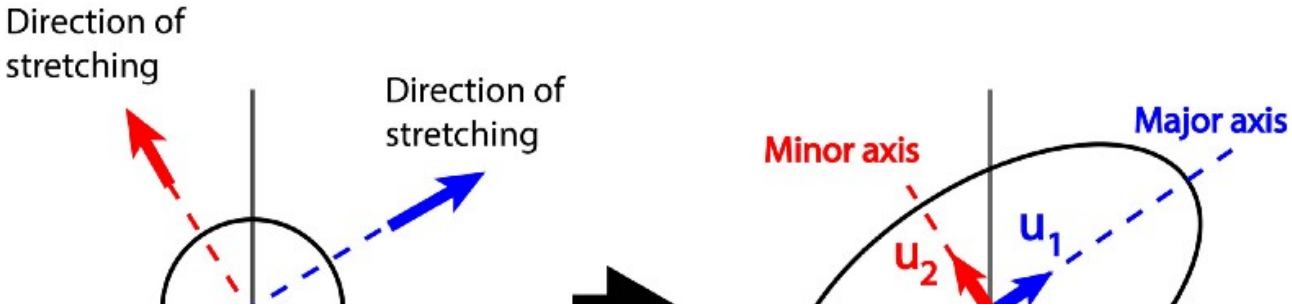


Figure 4

This time the eigenvectors have an interesting property. We see that the eigenvectors are along the major and minor axes of the ellipse (principal axes). An ellipse can be thought of as a circle stretched or shrunk along its principal axes as shown in Figure 5, and matrix **B** transforms the initial circle by stretching it along **u1** and **u2**, the eigenvectors of **B**.



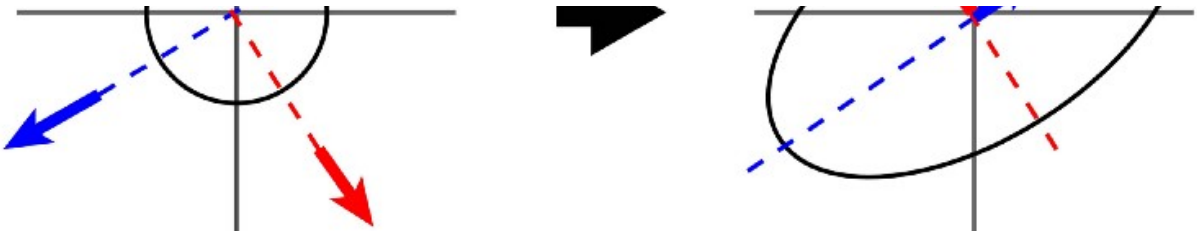


Figure 5

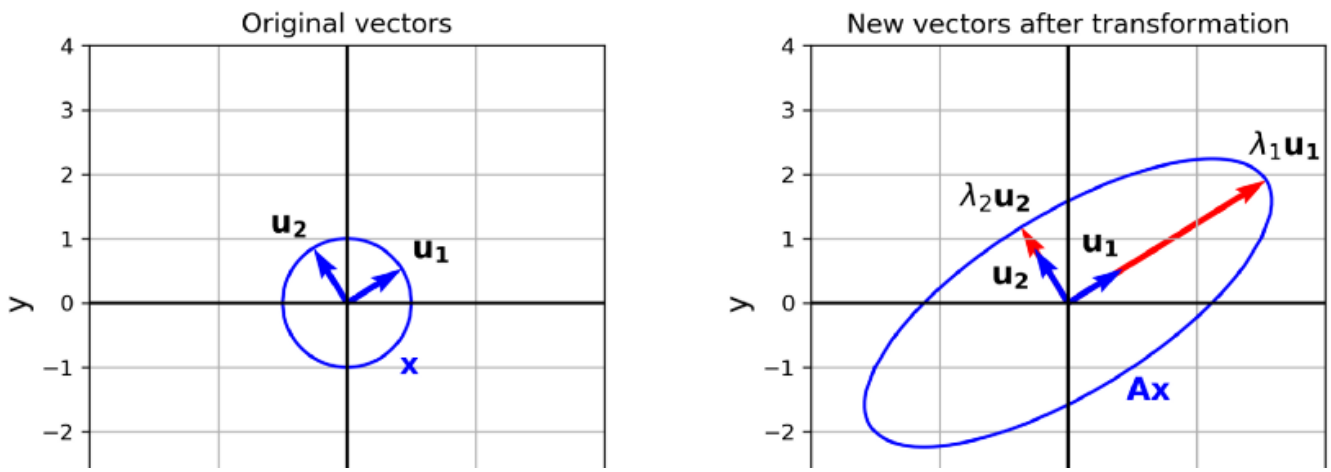
But why the eigenvectors of **A** did not have this property? That is because **B** is a symmetric matrix. A symmetric matrix is a matrix that is equal to its transpose. So the elements on the main diagonal are arbitrary but for the other elements, each element on row i and column j is equal to the element on row j and column i ($aij = aji$). Here is an example of a symmetric matrix:

$$\begin{bmatrix} 5 & 0 & 4 & 3 \\ 0 & 7 & 9 & 2 \\ 4 & 9 & 6 & 1 \\ 3 & 2 & 1 & 3 \end{bmatrix}$$

A symmetric matrix is always a square matrix ($n \times n$). You can now easily see that **A** was not symmetric. A symmetric matrix transforms a vector by stretching or shrinking it along its eigenvectors. In addition, we know that all the matrices transform an eigenvector by multiplying its length (or magnitude) by the corresponding eigenvalue. We know that the initial vectors in the circle have a length of 1 and both **u1** and **u2** are normalized, so they are part of the initial vectors **x**. Now their transformed vectors are:

$$\mathbf{Bu_1} = \lambda_1 \mathbf{u_1} \qquad \mathbf{Bu_2} = \lambda_2 \mathbf{u_2}$$

So the amount of stretching or shrinking along each eigenvector is proportional to the corresponding eigenvalue as shown in Figure 6.



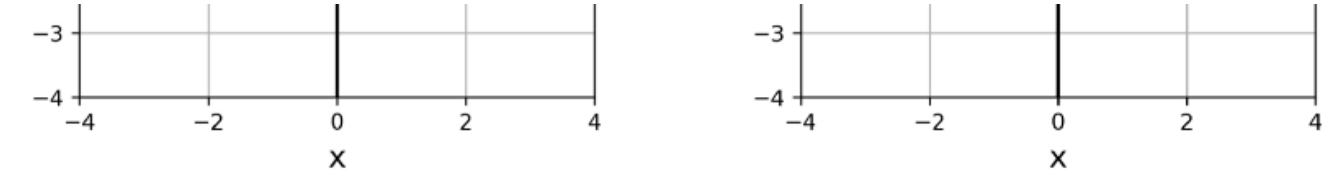


Figure 6

So when you have more stretching in the direction of an eigenvector, the eigenvalue corresponding to that eigenvector will be greater. In fact, if the absolute value of an eigenvalue is greater than 1, the circle **x** stretches along it, and if the absolute value is less than 1, it shrinks along it. Let me try this matrix:

$$\mathbf{C} = \begin{bmatrix} 3 & 1 \\ 1 & 0.8 \end{bmatrix}$$

The eigenvectors and corresponding eigenvalues are:

$$\mathbf{u_1} = \begin{bmatrix} 0.9327 \\ 0.3606 \end{bmatrix} \quad \mathbf{u_2} = \begin{bmatrix} -0.3606 \\ 0.9327 \end{bmatrix}$$
$$\lambda_1 = 3.3866 \quad \lambda_2 = 0.4134$$

Now if we plot the transformed vectors we get:

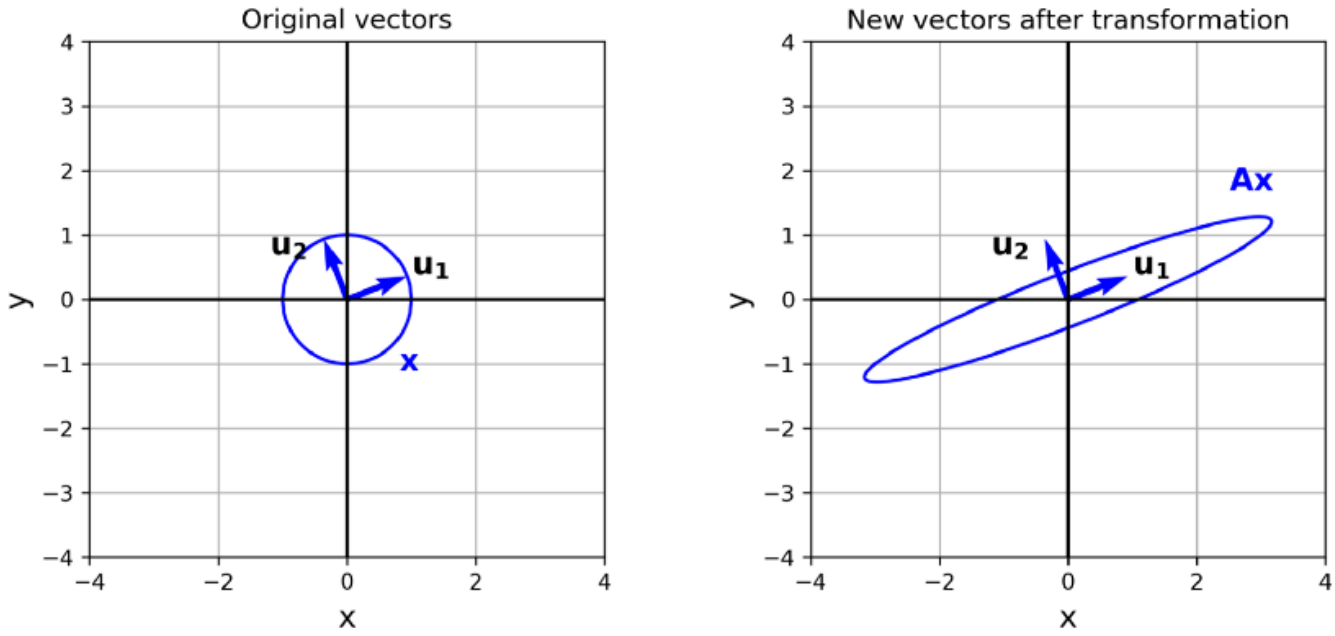


Figure 7

As you see now we have stretching along **u1** and shrinking along **u2**. The other important thing about these eigenvectors is that they can form a *basis* for a vector space.

Basis

A set of vectors {**v1**, **v2**, **v3** …, **vn}** form a basis for a vector space **V**, if they are linearly independent and span **V**. A vector space is a set of vectors that can be added together or multiplied by scalars. This is a closed set, so when the vectors are added or multiplied by a scalar, the result still belongs to the set. The operations of vector addition and scalar multiplication must satisfy certain requirements which are not discussed here. Euclidean space $\mathbf{R}^2$ (in which we are plotting our vectors) is an example of a vector space.

When a set of vectors is linearly independent, it means that no vector in the set can be written as a linear combination of the other vectors. So it is not possible to write

$$\mathbf{v_i} = a_1\mathbf{v_1} + a_2\mathbf{v_2} + ... + a_{i-1}\mathbf{v_{i-1}} + a_{i+1}\mathbf{v_{i+1}} + ... + a_n\mathbf{v_n}$$

when some of *a1*, *a2*, .., *an* are not zero. In other words, none of the **vi** vectors in this set can be expressed in terms of the other vectors. A set of vectors spans a space if every other vector in the space can be written as a linear combination of the spanning set. So every vector **s** in **V** can be written as:

$$\mathbf{s} = a_1\mathbf{v_1} + a_2\mathbf{v_2} + ... + a_n\mathbf{v_n}$$

A vector space **V** can have many different vector bases, but each basis always has the same number of basis vectors. The number of basis vectors of vector space **V** is called the *dimension* of **V**. In Euclidean space $\mathbf{R}^2$, the vectors:

$$\mathbf{v_1} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad \mathbf{v_2} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

is the simplest example of a basis since they are linearly independent and every vector in $\mathbf{R}^2$ can be expressed as a linear combination of them. They are called the *standard basis* for $\mathbf{R}^2$. As a result, the dimension of $\mathbf{R}^2$ is 2. It can have other bases, but all of them have two vectors that are linearly independent and span it. For example, vectors:

$$\mathbf{v_1} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad \mathbf{v_2} = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

can also form a basis for $\mathbf{R}^2$. An important reason to find a basis for a vector space is to have a coordinate system on that. If the set of vectors $B = \{\mathbf{v1}, \mathbf{v2}, \mathbf{v3} \ldots, \mathbf{vn}\}$ form a basis for a vector space, then every vector $\mathbf{x}$ in that space can be uniquely specified using those basis vectors :

$$\mathbf{x} = a_1\mathbf{v_1} + a_2\mathbf{v_2} + \ldots + a_n\mathbf{v_n}$$

Now the coordinate of $\mathbf{x}$ relative to this basis B is:

$$[\mathbf{x}]_B = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}$$

In fact, when we are writing a vector in $\mathbf{R}^2$, we are already expressing its coordinate relative to the standard basis. That is because any vector

$$\mathbf{x} = \begin{bmatrix} a \\ b \end{bmatrix}$$

can be written as

$$\mathbf{x} = a\begin{bmatrix} 1 \\ 0 \end{bmatrix} + b\begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Now a question comes up. If we know the coordinate of a vector relative to the standard basis, how can we find its coordinate relative to a new basis?

The equation:

$$\mathbf{x} = a_1\mathbf{v_1} + a_2\mathbf{v_2} + \ldots + a_n\mathbf{v_n}$$

can be also written as:

$$\mathbf{x} = \begin{bmatrix} \mathbf{v_1} & \mathbf{v_2} & \ldots & \mathbf{v_n} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \end{bmatrix} = \begin{bmatrix} \mathbf{v_1} & \mathbf{v_2} & \ldots & \mathbf{v_n} \end{bmatrix} [\mathbf{x}]_B = \mathbf{P}_B[\mathbf{x}]_B$$

$$\lfloor a_n \rfloor$$

The matrix:

$$\mathbf{P_B} = \begin{bmatrix} \mathbf{v_1} & \mathbf{v_2} & \dots & \mathbf{v_n} \end{bmatrix}$$

is called the change-of-coordinate matrix. The columns of this matrix are the vectors in basis *B*. The equation

$$\mathbf{x} = \mathbf{P}_B[\mathbf{x}]_B$$

gives the coordinate of **x** in $\mathbf{R}^2$ if we know its coordinate in basis *B*. If we need the opposite we can multiply both sides of this equation by the inverse of the change-of-coordinate matrix to get:

$$[\mathbf{x}]_B = \mathbf{P}_B^{-1}\mathbf{x}$$

Now if we know the coordinate of **x** in $\mathbf{R}^2$ (which is simply **x** itself), we can multiply it by the inverse of the change-of-coordinate matrix to get its coordinate relative to basis *B*. For example, suppose that our basis set *B* is formed by the vectors:

$$\mathbf{v_1} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad \mathbf{v_2} = \begin{bmatrix} -1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}$$

and we have a vector:

$$\mathbf{x} = \begin{bmatrix} 2 \\ 2 \end{bmatrix}$$

To calculate the coordinate of **x** in *B*, first, we form the change-of-coordinate matrix:

$$\mathbf{P_B} = \begin{bmatrix} 1 & -1/\sqrt{2} \\ 0 & 1/\sqrt{2} \end{bmatrix}$$

Now the coordinate of **x** relative to *B* is:

$$[\quad]_{\mathbf{B}} = \mathbf{P}^{-1} \qquad \begin{bmatrix} 1 & -1/\sqrt{2} \end{bmatrix}^{-1} \begin{bmatrix} 2 \end{bmatrix}$$

$$[\mathbf{x}]_B = P_B^{-1}\mathbf{x} = \begin{bmatrix} 0 & 1/\sqrt{2} \\ \end{bmatrix} \quad \begin{bmatrix} 2 \end{bmatrix}$$

Listing 6 shows how this can be calculated in NumPy. To calculate the inverse of a matrix, the function `np.linalg.inv()` can be used.

```
# Listing 6

# The Basis
v_1 = np.array([[1],[0]])
v_2 = np.array([[-1/mt.sqrt(2)],[1/mt.sqrt(2)]])

# Change of coordinate matrix
p =np.concatenate([v_1, v_2], axis=1)
p_inv =  np.linalg.inv(p)

# Coordinate of x in R^2
x=np.array([[2], [2]])

# New coordinate relative to basis B
x_B = p_inv @ x

print("x_B=", np.round(x_B, 2))
```

SVD_Listing6.py hosted with ♡ by GitHub view raw

The output shows the coordinate of x in B:

```
x_B= [[4.  ]
      [2.83]]
```

Figure 8 shows the effect of changing the basis.

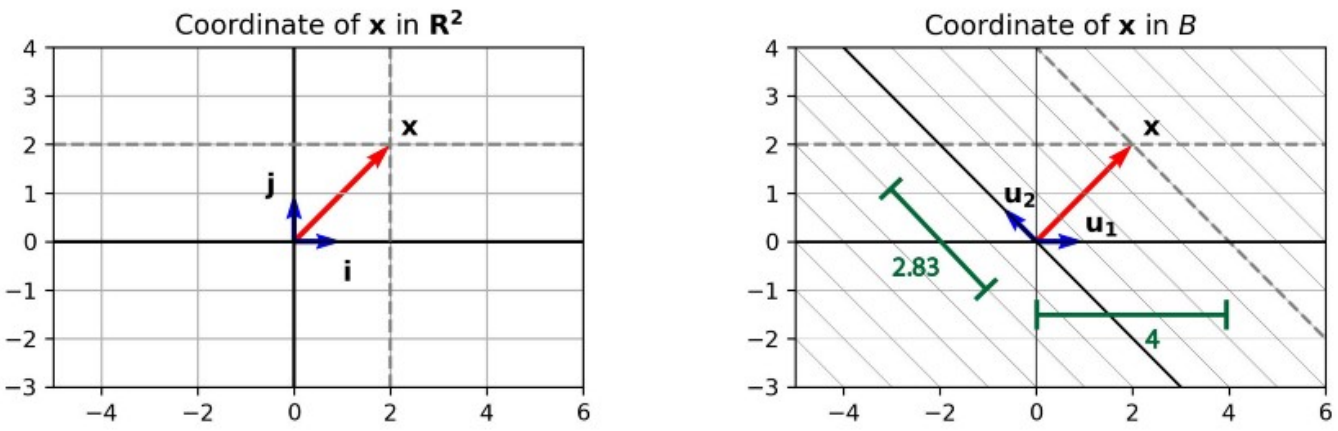


Figure 8

To find the **u1**-coordinate of **x** in basis *B*, we can draw a line passing from **x** and parallel to **u2** and see where it intersects the **u1** axis. **u2**-coordinate can be found similarly as shown in Figure 8. In an *n*-dimensional space, to find the coordinate of **ui**, we need to draw a hyper-plane passing from **x** and parallel to all other eigenvectors except **ui** and see where it intersects the **ui** axis. As Figure 8 (left) shows when the eigenvectors are orthogonal (like **i** and **j** in $\mathbf{R}^2$), we just need to draw a line that passes through point **x** and is perpendicular to the axis that we want to find its coordinate.

Properties of symmetric matrices

As figures 5 to 7 show the eigenvectors of the symmetric matrices **B** and **C** are perpendicular to each other and form orthogonal vectors. This is not a coincidence and is a property of symmetric matrices.

An important property of the symmetric matrices is that an $n \times n$ symmetric matrix has n linearly independent and orthogonal eigenvectors, and it has n real eigenvalues corresponding to those eigenvectors. It is important to note that these eigenvalues are not necessarily different from each other and some of them can be equal. Another important property of symmetric matrices is that they are orthogonally diagonalizable.

Eigendecomposition

A symmetric matrix is orthogonally diagonalizable. It means that if we have an $n \times n$ symmetric matrix **A**, we can decompose it as

$$\mathbf{A} = \mathbf{PDP}^T$$

where **D** is an $n \times n$ diagonal matrix comprised of the n eigenvalues of **A**. **P** is also an $n \times n$ matrix, and the columns of **P** are the n linearly independent eigenvectors of **A** that correspond to those eigenvalues in **D** respectively. In other words, if **u1**, **u2**, **u3** …, **un** are the eigenvectors of **A**, and $\lambda 1, \lambda 2, \ldots, \lambda n$ are their corresponding eigenvalues respectively, then **A** can be written as

$$\mathbf{A} = \begin{bmatrix} \mathbf{u_1} & \mathbf{u_2} & \dots & \mathbf{u_n} \end{bmatrix} \begin{bmatrix} \lambda_1 & 0 & \dots & 0 \\ 0 & \lambda_2 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & \lambda_n \end{bmatrix} \begin{bmatrix} \mathbf{u_1} & \mathbf{u_2} & \dots & \mathbf{u_n} \end{bmatrix}^T$$

This can also be written as

$$\mathbf{A} = \begin{bmatrix} \mathbf{u_1} & \mathbf{u_2} & \dots & \mathbf{u_n} \end{bmatrix} \begin{bmatrix} \lambda_1 & 0 & \dots & 0 \\ 0 & \lambda_2 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & \lambda_n \end{bmatrix} \begin{bmatrix} \mathbf{u_1}^T \\ \mathbf{u_2}^T \\ \vdots \\ \mathbf{u_n}^T \end{bmatrix}$$

You should notice that each **ui** is considered a column vector and its transpose is a row vector. So the transpose of **P** has been written in terms of the transpose of the columns of **P**. This factorization of **A** is called the eigendecomposition of **A**.

Let me clarify it by an example. Suppose that

$$\mathbf{A} = \begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix}$$

It has two eigenvectors:

$$\mathbf{u_1} = \begin{bmatrix} 0.8507 \\ 0.5257 \end{bmatrix} \quad \mathbf{u_2} = \begin{bmatrix} -0.5257 \\ 0.8507 \end{bmatrix}$$

and the corresponding eigenvalues were:

$$\lambda_1 = 3.618 \qquad \lambda_2 = 1.382$$

So **D** can be defined as

$$\mathbf{D} = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} = \begin{bmatrix} 3.618 & 0 \\ 0 & 1.382 \end{bmatrix}$$

Now the columns of **P** are the eigenvectors of **A** that correspond to those eigenvalues in **D** respectively. So

$$\mathbf{P} = \begin{bmatrix} \mathbf{u_1} & \mathbf{u_2} \end{bmatrix} = \begin{bmatrix} 0.8507 & -0.5257 \\ 0.5257 & 0.8507 \end{bmatrix}$$

The transpose of **P** is

$$\mathbf{P}^T = \begin{bmatrix} \mathbf{u_1} & \mathbf{u_2} \end{bmatrix}^T = \begin{bmatrix} \mathbf{u_1}^T \\ \mathbf{u_2}^T \end{bmatrix} = \begin{bmatrix} 0.8507 & 0.5257 \\ -0.5257 & 0.8507 \end{bmatrix}$$

So **A** can be written as

$$\mathbf{A} = \begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix} = \begin{bmatrix} 0.8507 & -0.5257 \\ 0.5257 & 0.8507 \end{bmatrix} \begin{bmatrix} 3.618 & 0 \\ 0 & 1.382 \end{bmatrix} \begin{bmatrix} 0.8507 & 0.5257 \\ -0.5257 & 0.8507 \end{bmatrix}$$

It is important to note that if you do the multiplications on the right side of the above equation, you will not get **A** exactly. That is because we have the rounding errors in NumPy to calculate the irrational numbers that usually show up in the eigenvalues and eigenvectors, and we have also rounded the values of the eigenvalues and eigenvectors here, however, in theory, both sides should be equal. But what does it mean? To understand the eigendecomposition better, we can take a look at its geometrical interpretation.

Geometrical interpretation of eigendecomposition

To better understand the eigendecomposition equation, we need to first simplify it. If we assume that each eigenvector **ui** is an $n \times 1$ column vector

$$\mathbf{u_i} = \begin{bmatrix} u_{i1} \\ u_{i2} \\ \vdots \\ u_{in} \end{bmatrix}$$

then the transpose of **ui** is a $1 \times n$ row vector

$$\mathbf{u_i^T} = \begin{bmatrix} u_{i1} & u_{i2} & \dots & u_{in} \end{bmatrix}$$

and their multiplication

$$\mathbf{u_i u_i^T} = \begin{bmatrix} u_{i1} \\ u_{i2} \\ \vdots \end{bmatrix} \begin{bmatrix} u_{i1} & u_{i2} & \dots & u_{in} \end{bmatrix} = \begin{bmatrix} u_{i1}u_{i1} & u_{i1}u_{i2} & \dots & u_{i1}u_{in} \\ u_{i2}u_{i1} & u_{i2}u_{i2} & \dots & u_{i2}u_{in} \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

$$\begin{bmatrix} u_{in} \end{bmatrix} \qquad \begin{bmatrix} u_{in}u_{i1} & u_{in}u_{i2} & \dots & u_{in}u_{in} \end{bmatrix}$$

becomes an $n \times n$ matrix. First, we calculate **DP^T** to simplify the eigendecomposition equation:

$$\begin{bmatrix} \lambda_1 & 0 & \dots & 0 \\ 0 & \lambda_2 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & \lambda_n \end{bmatrix} \begin{bmatrix} \mathbf{u_1}^T \\ \mathbf{u_2}^T \\ \vdots \\ \mathbf{u_n}^T \end{bmatrix} =$$

$$\begin{bmatrix} \lambda_1 & 0 & \dots & 0 \\ 0 & \lambda_2 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & \lambda_n \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ u_{21} & u_{22} & \dots & u_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ u_{n1} & u_{n2} & \dots & u_{nn} \end{bmatrix} =$$

$$\begin{bmatrix} \lambda_1 u_{11} & \lambda_1 u_{12} & \dots & \lambda_1 u_{1n} \\ \lambda_2 u_{21} & \lambda_2 u_{22} & \dots & \lambda_2 u_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ \lambda_n u_{n1} & \lambda_n u_{n2} & \dots & \lambda_n u_{nn} \end{bmatrix} = \begin{bmatrix} \lambda_1 \mathbf{u_1}^T \\ \lambda_2 \mathbf{u_2}^T \\ \vdots \\ \lambda_n \mathbf{u_n}^T \end{bmatrix}$$

Now the eigendecomposition equation becomes:

$$\mathbf{A} = \begin{bmatrix} \mathbf{u_1} & \mathbf{u_2} & \dots & \mathbf{u_n} \end{bmatrix} \begin{bmatrix} \lambda_1 \mathbf{u_1}^T \\ \lambda_2 \mathbf{u_2}^T \\ \vdots \\ \lambda_n \mathbf{u_n}^T \end{bmatrix} =$$

$$\lambda_1 \mathbf{u_1} \mathbf{u_1}^T + \lambda_2 \mathbf{u_2} \mathbf{u_2}^T + \dots + \lambda_n \mathbf{u_n} \mathbf{u_n}^T$$

So the $n \times n$ matrix **A** can be broken into n matrices with the same shape ($n \times n$), and each of these matrices has a multiplier which is equal to the corresponding eigenvalue λi. Each of the matrices

$$\mathbf{u_i} \mathbf{u_i}^T$$

is called a *projection matrix*. Imagine that we have a vector **x** and a unit vector **v**. The inner product of **v** and **x** which is equal to **v.x=v^T x** gives the scalar projection of **x** onto **v** (which is the length of the vector projection of **x** into **v**), and if we multiply it by **v**

again, it gives a vector which is called the orthogonal projection of **x** onto **v**. This is shown in Figure 9.

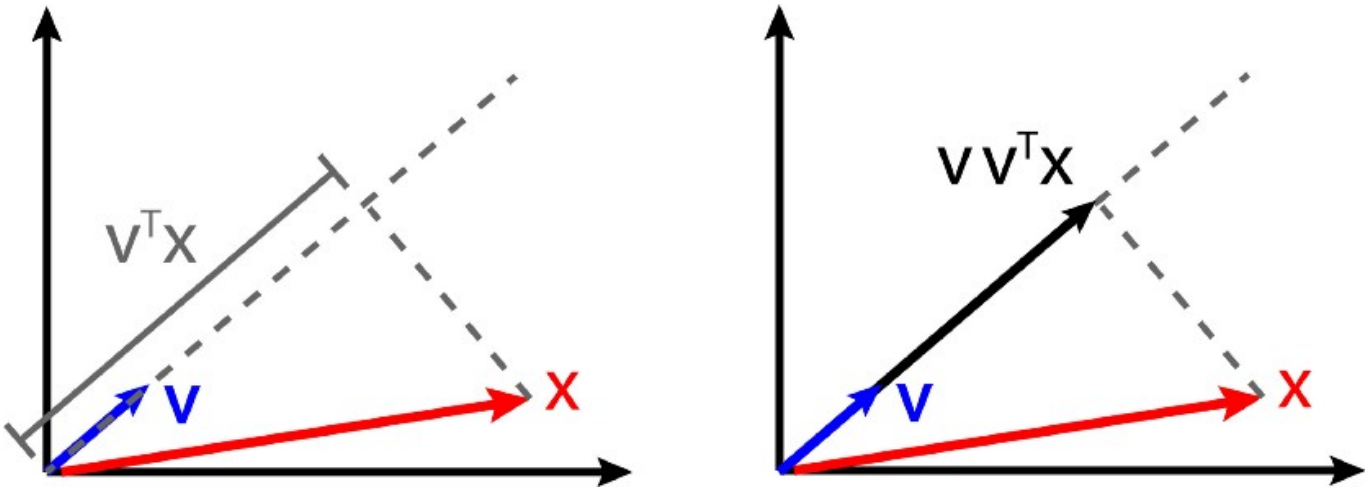


Figure 9

So when **v** is a unit vector, multiplying

$$\mathbf{v}\mathbf{v}^T$$

by **x**, will give the orthogonal projection of **x** onto **v**, and that is why it is called the projection matrix. So multiplying **ui ui**^T by **x**, we get the orthogonal projection of **x** onto **ui**.

Now let me calculate the projection matrices of matrix **A** mentioned before.

$$\mathbf{A} = \begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix}$$

We already had calculated the eigenvalues and eigenvectors of **A**.

Using the output of Listing 7, we get the first term in the eigendecomposition equation (we call it **A1** here):

```
# Listing 7
A = np.array([[3, 1],
              [1, 2]])

lam, u = LA.eig(A)
u1= u[:,0].reshape(2,1)
lam1 = lam[0]
# A_1 = lambda_1 * u1 * u1^T
```

```
A_1 = lam1 * (u1 @ u1.T)
u2= u[:,1].reshape(2,1)
lam2 = lam[1]
# A_2 = lambda_2 * u2 * u2^T
A_2 = lam2 * (u2 @ u2.T)
print("A_1=", np.round(A_1, 4))
```

SVD_Listing7.py hosted with ♡ by GitHub view raw

$$\mathbf{A_1} = \lambda_1 \mathbf{u_1} \mathbf{u_1}^T = 3.618 \begin{bmatrix} 0.8507 \\ 0.5257 \end{bmatrix} \begin{bmatrix} 0.8507 & 0.5257 \end{bmatrix} = \begin{bmatrix} 2.618 & 1.618 \\ 1.618 & 1 \end{bmatrix}$$

As you see it is also a symmetric matrix. In fact, all the projection matrices in the eigendecomposition equation are symmetric. That is because the element in row *m* and column *n* of each matrix

$$\lambda_i \mathbf{u_i} \mathbf{u_i}^T$$

is equal to

$$\lambda_i u_{im} u_{in}$$

and the element at row *n* and column *m* has the same value which makes it a symmetric matrix. This projection matrix has some interesting properties. First, we can calculate its eigenvalues and eigenvectors:

```
# Listing 8
lam_A_1, u_A_1 = LA.eig(A_1)
print("lam=", np.round(lam_A_1, 4))
print("u=", np.round(u_A_1, 4))
```

SVD_Listing8.py hosted with ♡ by GitHub view raw

```
lam= [ 3.618  0.   ]
u= [[ 0.8507 -0.5257]
    [ 0.5257  0.8507]]
```

As you see, it has two eigenvalues (since it is a 2×2 symmetric matrix). One of them is zero and the other is equal to *λ1* of the original matrix **A**. In addition, the eigenvectors

are exactly the same eigenvectors of **A**. This is not a coincidence. Suppose we get the *i*-th term in the eigendecomposition equation and multiply it by **ui**.

$$(\lambda_i \mathbf{u_i} \mathbf{u_i^T}) \mathbf{u_i} = \lambda_i \mathbf{u_i} (\mathbf{u_i^T} \mathbf{u_i})$$

We know that **ui** is an eigenvector and it is normalized, so its length and its inner product with itself are both equal to 1. So:

$$(\lambda_i \mathbf{u_i} \mathbf{u_i^T}) \mathbf{u_i} = \lambda_i \mathbf{u_i}$$

Now if you look at the definition of the eigenvectors, this equation means that one of the eigenvalues of the matrix

$$\lambda_i \mathbf{u_i} \mathbf{u_i}^T$$

is *λi* and the corresponding eigenvector is **ui**. But this matrix is an *n*×*n symmetric* matrix and should have *n* eigenvalues and eigenvectors. Now we can multiply it by any of the remaining *(n-1)* eigenvalues of **A** to get:

$$(\lambda_i \mathbf{u_i} \mathbf{u_i}^T) \mathbf{u_j} = \lambda_i \mathbf{u_i} (\mathbf{u_i}^T \mathbf{u_j})$$

where i ≠ j. We know that the eigenvalues of **A** are orthogonal which means each pair of them are perpendicular. The inner product of two perpendicular vectors is zero (since the scalar projection of one onto the other should be zero). So the inner product of **ui** and **uj** is zero, and we get

$$(\lambda_i \mathbf{u_i} \mathbf{u_i}^T) \mathbf{u_j} = \lambda_i \mathbf{u_i} (\mathbf{u_i}^T \mathbf{u_j}) = 0 = 0 \times \mathbf{u_j}$$

which means that **uj** is also an eigenvector and its corresponding eigenvalue is zero. So we conclude that each matrix

$$\lambda_i \mathbf{u_i} \mathbf{u_i}^T$$

in the eigendecomposition equation is a symmetric *n*×*n* matrix with *n* eigenvectors. The eigenvectors are the same as the original matrix **A** which are **u1, u2, … un**. The corresponding eigenvalue of **ui** is *λi* (which is the same as **A**), but all the other

eigenvalues are zero. Now, remember how a symmetric matrix transforms a vector. It will stretch or shrink the vector along its eigenvectors, and the amount of stretching or shrinking is proportional to the corresponding eigenvalue. So this matrix will stretch a vector along **ui**. But since the other eigenvalues are zero, it will shrink it to zero in those directions. Let me go back to matrix **A** and plot the transformation effect of **A1** using Listing 9.

```
# Listing 9

# Creating the vectors for a circle and storing them in x
xi1 = np.linspace(-1.0, 1.0, 100)
xi2 = np.linspace(1.0, -1.0, 100)
yi1 = np.sqrt(1 - xi1**2)
yi2 = -np.sqrt(1 - xi2**2)

xi = np.concatenate((xi1, xi2),axis=0)
yi = np.concatenate((yi1, yi2),axis=0)
x = np.vstack((xi, yi))

t = A_1 @ x  # Vectors in t are the transformed vectors of x

# getting the transformed sample of x from t
t_sample = t[:, 100]

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10,15))

plt.subplots_adjust(wspace=0.4)

# Plotting s
ax1.plot(x[0,:], x[1,:], color='b')
ax1.set_xlabel('x', fontsize=14)
ax1.set_ylabel('y', fontsize=14)
ax1.set_xlim([-4,4])
ax1.set_ylim([-4,4])
ax1.set_aspect('equal')
ax1.grid(True)
ax1.set_title("Original vectors")
ax1.axhline(y=0, color='k')
ax1.axvline(x=0, color='k')
ax1.text(0.8, 0.8, "$\mathbf{x}$", fontsize=14)

# Plotting t
ax2.plot(t[0, :], t[1, :], color='r')
ax2.quiver(*origin, u[0,:], u[1,:], color=['b'], width=0.012, angles='xy', scale_units='xy', sca
ax2.set_xlabel('x', fontsize=14)
```

```
ax2.set_ylabel('y', fontsize=14)
ax2.set_xlim([-4,4])
ax2.set_ylim([-4,4])
ax2.set_aspect('equal')
ax2.grid(True)
ax2.set_title("New vectors after transformation")
ax2.axhline(y=0, color='k')
ax2.axvline(x=0, color='k')
ax2.text(0.4, 0.8, "$\mathbf{u_1}$", fontsize=14)
ax2.text(-1.2, 0.5, "$\mathbf{u_2}$", fontsize=14)
ax2.text(1.6, 2.3, "$\lambda_1\mathbf{u_1}\mathbf{u_1}^T\mathbf{x}$", fontsize=14)
#plt.savefig('8.png', dpi=300, bbox_inches='tight')

plt.show()
```

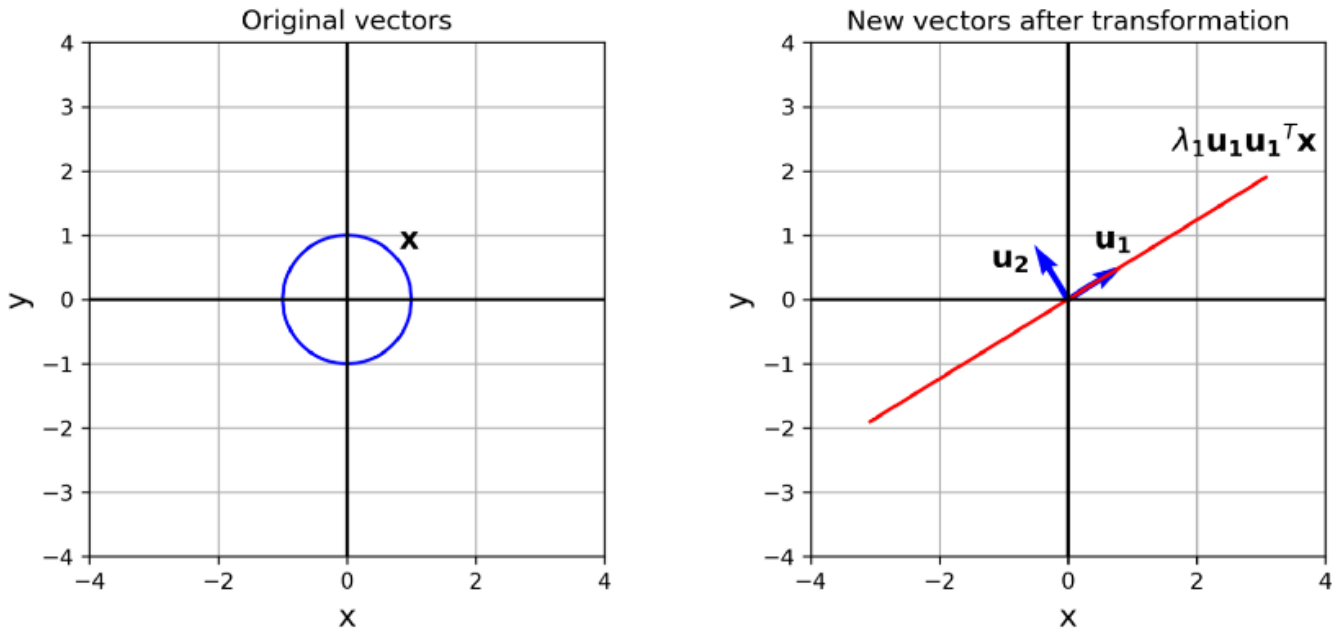


Figure 10

As you see, the initial circle is stretched along **u1** and shrunk to zero along **u2.** So the result of this transformation is a straight line, not an ellipse. This is consistent with the fact that **A1** is a projection matrix and should project everything onto **u1**, so the result should be a straight line along **u1.**

Rank

Figure 10 shows an interesting example in which the 2×2 matrix **A1** is multiplied by a 2-d vector **x**, but the transformed vector **Ax** is a straight line. Here is another example. Suppose that we have a matrix:

$$\mathbf{F} = \begin{bmatrix} 3 & 1 \\ 6 & 2 \end{bmatrix}$$

Figure 11 shows how it transforms the unit vectors **x**.

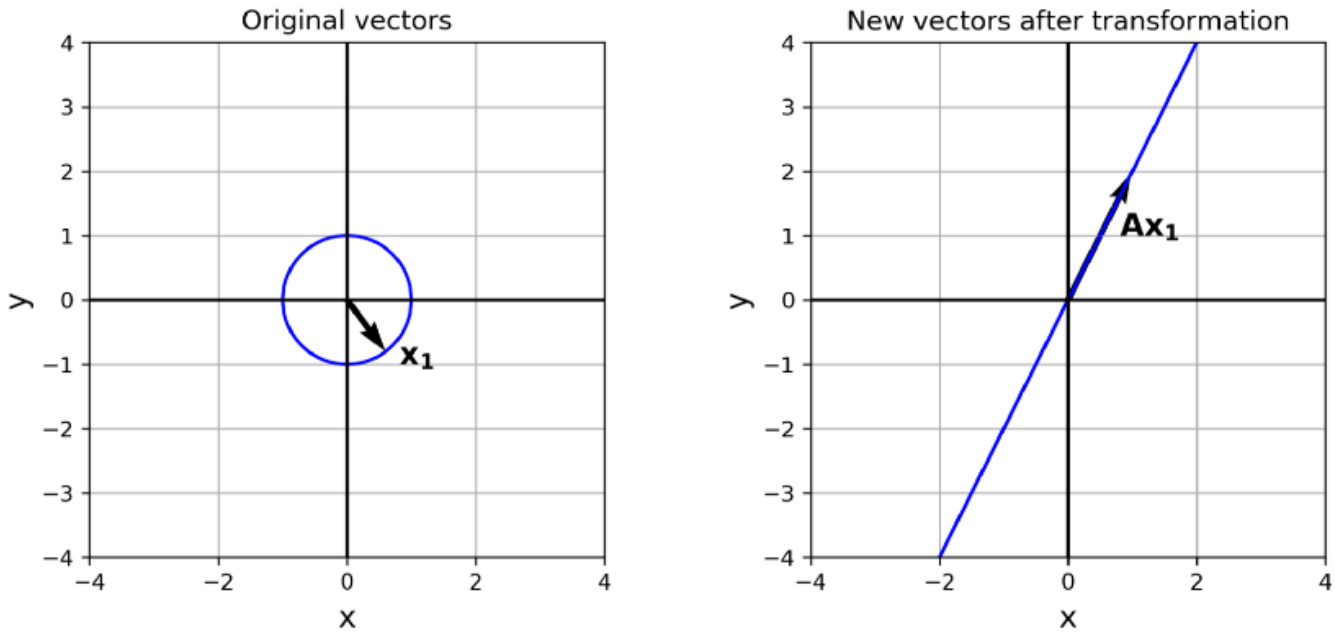


Figure 11

So it acts as a projection matrix and projects all the vectors in **x** on the line $y=2x$. That is because the columns of F are not linear independent. In fact, if the columns of **F** are called **f1** and **f2** respectively, then we have **f1**$=$**2f2**. Remember that we write the multiplication of a matrix and a vector as:

$$\mathbf{Fx} = \begin{bmatrix} \mathbf{f}_1 & \mathbf{f}_2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = x\mathbf{f}_1 + y\mathbf{f}_2 = 2x\mathbf{f}_2 + y\mathbf{f}_2 = (2x + y)\mathbf{f}_2 = c\mathbf{f}_2$$

So unlike the vectors in **x** which need two coordinates, **Fx** only needs one coordinate and exists in a 1-d space. In general, an $m \times n$ matrix does not necessarily transform an n-dimensional vector into anther m-dimensional vector. The dimension of the transformed vector can be lower if the columns of that matrix are not linearly independent.

The *column space* of matrix **A** written as *Col* **A** is defined as the set of all linear combinations of the columns of **A**, and since **Ax** is also a linear combination of the columns of **A**, *Col* **A** is the set of all vectors in **Ax**. The number of basis vectors of *Col* **A** or the dimension of *Col* **A** is called the rank of **A**. So the rank of **A** is the dimension of **Ax**.

The rank of **A** is also the maximum number of linearly independent columns of **A**. That is because we can write all the dependent columns as a linear combination of these linearly independent columns, and **Ax** which is a linear combination of all the columns can be written as a linear combination of these linearly independent columns. So they span **Ax** and form a basis for *col* **A**, and the number of these vectors becomes the dimension of *col* of **A** or rank of **A**.

In the previous example, the rank of **F** is 1. In addition, in the eigendecomposition equation, the rank of each matrix

$$\lambda_i \mathbf{u_i} \mathbf{u_i}^T$$

is 1. Remember that they only have one non-zero eigenvalue and that is not a coincidence. It can be shown that the rank of a symmetric matrix is equal to the number of its non-zero eigenvalues.

Now we go back to the eigendecomposition equation again. Suppose that we apply our symmetric matrix **A** to an arbitrary vector **x**. Now the eigendecomposition equation becomes:

$$\mathbf{Ax} = \lambda_1 \mathbf{u_1} \mathbf{u_1}^T \mathbf{x} + \lambda_2 \mathbf{u_2} \mathbf{u_2}^T \mathbf{x} + ... + \lambda_n \mathbf{u_n} \mathbf{u_n}^T \mathbf{x}$$

Each of the eigenvectors **ui** is normalized, so they are unit vectors. Now in each term of the eigendecomposition equation

$$\mathbf{u_i} \mathbf{u_i}^T \mathbf{x}$$

gives a new vector which is the orthogonal projection of **x** onto **ui**. Then this vector is multiplied by *λi*. Since *λi* is a scalar, multiplying it by a vector, only changes the magnitude of that vector, not its direction. So *λi* only changes the magnitude of

$$\mathbf{u_i} \mathbf{u_i}^T \mathbf{x}$$

Finally all the *n* vectors

$$\lambda_i \mathbf{u_i} \mathbf{u_i}^T \mathbf{x}$$

are summed together to give **Ax**. This process is shown in Figure 12.

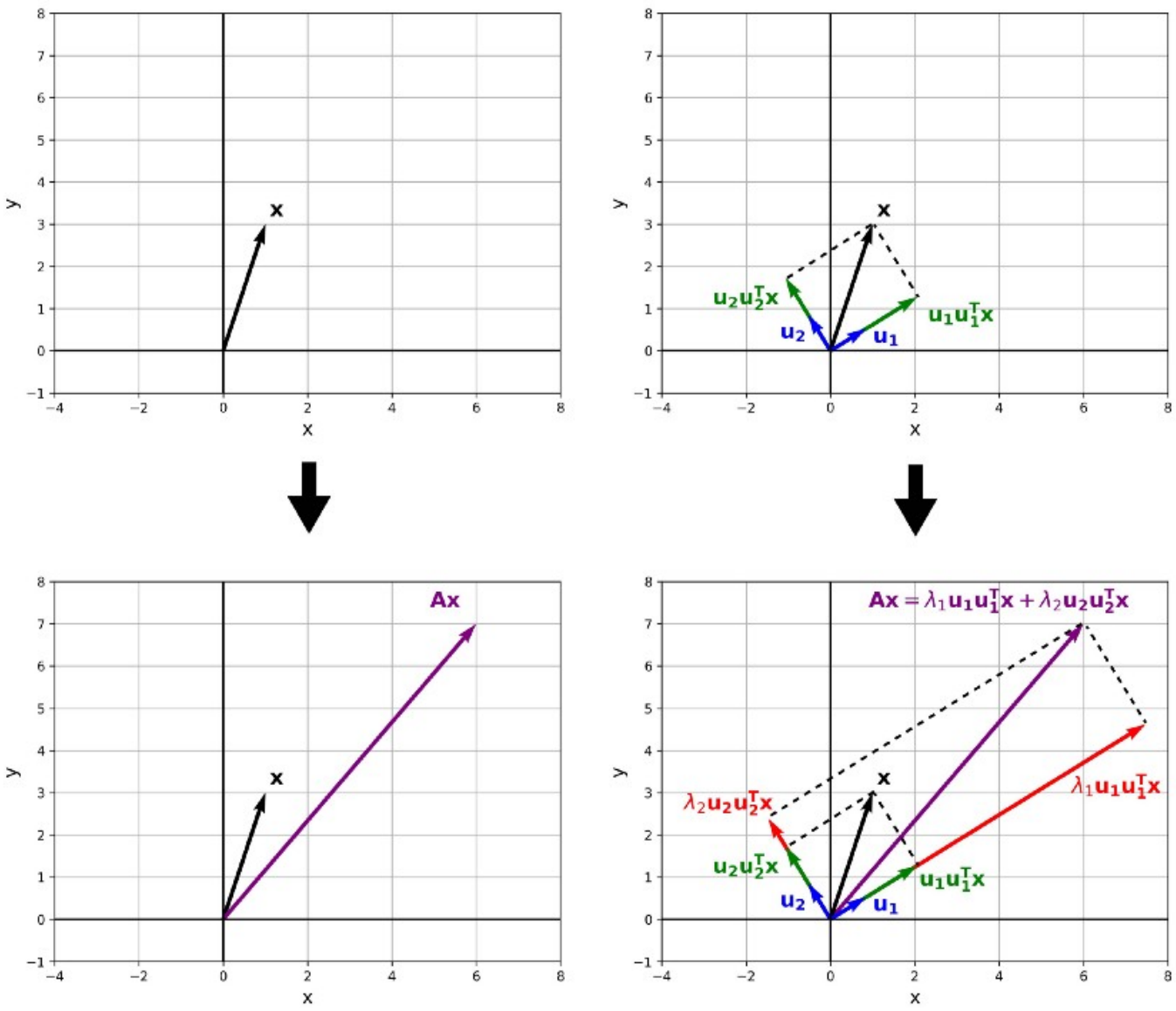


Figure 12

So the eigendecomposition mathematically explains an important property of the symmetric matrices that we saw in the plots before. A symmetric matrix transforms a vector by stretching or shrinking it along its eigenvectors, and the amount of stretching or shrinking along each eigenvector is proportional to the corresponding eigenvalue.

In addition, the eigendecomposition can break an $n \times n$ symmetric matrix into n matrices with the same shape ($n \times n$) multiplied by one of the eigenvalues. The eigenvalues play an important role here since they can be thought of as a multiplier. The projection matrix only projects **x** onto each **ui**, but the eigenvalue scales the length of the vector projection (**ui ui^Tx**). The bigger the eigenvalue, the bigger the length of the resulting vector (λi**ui ui^Tx**) is, and the more weight is given to its corresponding matrix (**ui ui^T**). So we can approximate our original symmetric matrix **A** by summing the terms

which have the highest eigenvalues. For example, if we assume the eigenvalues *λi* have been sorted in descending order,

$$\mathbf{A} = \lambda_1 \mathbf{u_1} \mathbf{u_1}^T + \lambda_2 \mathbf{u_2} \mathbf{u_2}^T + \ldots + \lambda_n \mathbf{u_n} \mathbf{u_n}^T$$
$$\lambda 1 \geq \lambda 2 \geq \cdots \geq \lambda_k \geq \cdots \geq \lambda_n$$

then we can only take the first *k* terms in the eigendecomposition equation to have a good approximation for the original matrix:

$$\mathbf{A} \approx \mathbf{A}_k = \lambda_1 \mathbf{u_1} \mathbf{u_1}^T + \lambda_2 \mathbf{u_2} \mathbf{u_2}^T + \ldots + \lambda_k \mathbf{u_k} \mathbf{u_k}^T$$

where **Ak** is the approximation of **A** with the first *k* terms. If we only include the first *k* eigenvalues and eigenvectors in the original eigendecomposition equation, we get the same result:

$$\mathbf{A} \approx \mathbf{A_k} = \mathbf{P_k D_k P_k^T} = \begin{bmatrix} \mathbf{u_1} & \mathbf{u_2} & \ldots & \mathbf{u_k} \end{bmatrix} \begin{bmatrix} \lambda_1 & 0 & \ldots & 0 \\ 0 & \lambda_2 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \ldots & \lambda_k \end{bmatrix} \begin{bmatrix} \mathbf{u_1}^T \\ \mathbf{u_2}^T \\ \vdots \\ \mathbf{u_k}^T \end{bmatrix}$$

$$= \lambda_1 \mathbf{u_1} \mathbf{u_1}^T + \lambda_2 \mathbf{u_2} \mathbf{u_2}^T + \ldots + \lambda_k \mathbf{u_k} \mathbf{u_k}^T$$

Now **Dk** is a *k*×*k* diagonal matrix comprised of the first *k* eigenvalues of **A, Pk** is an *n*×*k* matrix comprised of the first *k* eigenvectors of **A**, and its transpose becomes a *k*×*n* matrix. So their multiplication still gives an *n*×*n* matrix which is the same approximation of **A**.

If in the original matrix **A**, the other *(n-k)* eigenvalues that we leave out are very small and close to zero, then the approximated matrix is very similar to the original matrix, and we have a good approximation. Matrix

$$\mathbf{C} = \begin{bmatrix} 5 & 1 \\ 1 & 0.35 \end{bmatrix}$$

with

$$\lambda_1 = 5.2059 \quad \lambda_2 = 0.1441$$
$$\begin{bmatrix} 0.9794 \end{bmatrix} \qquad \begin{bmatrix} -0.2017 \end{bmatrix}$$

$$\mathbf{u_1} = \begin{bmatrix} 0.2017 \end{bmatrix} \qquad \mathbf{u_2} = \begin{bmatrix} 0.9794 \end{bmatrix}$$

is an example. Here *λ2* is rather small. *W*e call the vectors in the unit circle **x**, and plot the transformation of them by the original matrix (**Cx**). Then we approximate matrix **C** with the first term in its eigendecomposition equation which is:

$$\lambda_1 \mathbf{u_1} \mathbf{u_1}^T$$

and plot the transformation of **s** by that. As you see in Figure 13, the result of the approximated matrix which is a straight line is very close to the original matrix.

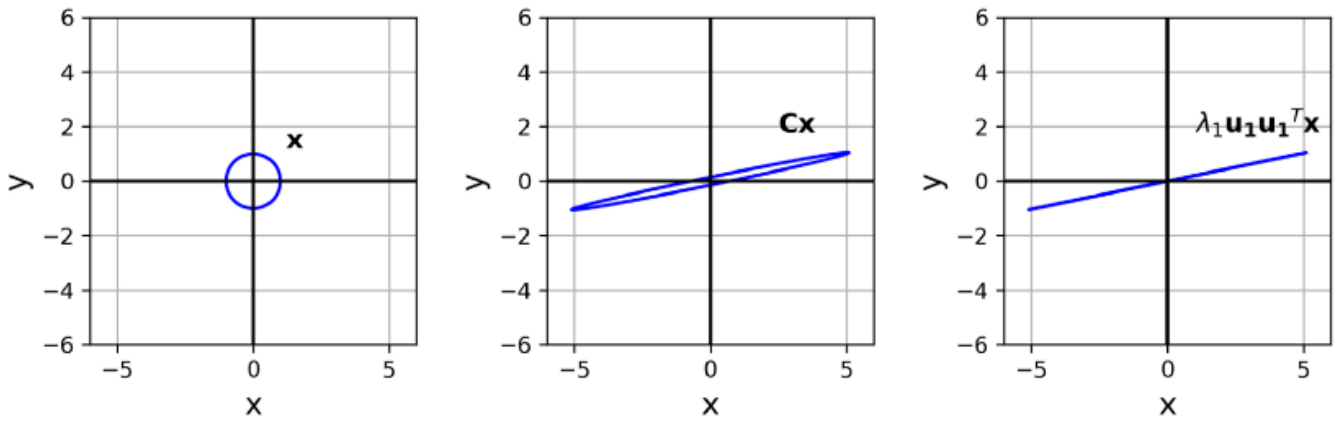


Figure 13

Why the eigendecomposition equation is valid and why it needs a symmetric matrix? Remember the important property of symmetric matrices. Suppose that **x** is an $n \times 1$ column vector. If **A** is an $n \times n$ symmetric matrix, then it has n linearly independent and orthogonal eigenvectors which can be used as a new basis. So we can now write the coordinate of **x** relative to this new basis:

$$[\mathbf{x}]_B = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}$$

and based on the definition of basis, any vector **x** can be uniquely written as a linear combination of the eigenvectors of **A**.

$$\mathbf{x} = a_1 \mathbf{u_1} + a_2 \mathbf{u_2} + \ldots + a_n \mathbf{u_n}$$

But the eigenvectors of a symmetric matrix are orthogonal too. So to find each coordinate *ai*, we just need to draw a line perpendicular to an axis of **ui** through point **x** and see where it intersects it (refer to Figure 8). As mentioned before this can be also done using the projection matrix. So each term *ai* is equal to the dot product of **x** and **ui** (refer to Figure 9), and **x** can be written as

$$\mathbf{x} = (\mathbf{u_1^T x})\mathbf{u_1} + (\mathbf{u_2^T x})\mathbf{u_2} + \cdots + (\mathbf{u_n^T x})\mathbf{u_n}$$

So we need a symmetric matrix to express **x** as a linear combination of the eigenvectors in the above equation. Now if we multiply **A** by **x**, we can factor out the *ai* terms since they are scalar quantities. So we get:

$$\mathbf{Ax} = (\mathbf{u_1^T x})\mathbf{Au_1} + (\mathbf{u_2^T x})\mathbf{Au_2} + \cdots + (\mathbf{u_n^T x})\mathbf{Au_n}$$

and since the **ui** vectors are the eigenvectors of **A**, we finally get:

$$\begin{aligned}\mathbf{Ax} &= (\mathbf{u_1^T x})\lambda_1\mathbf{u_1} + (\mathbf{u_2^T x})\lambda_2\mathbf{u_2} + \cdots + (\mathbf{u_n^T x})\lambda_n\mathbf{u_n} \\ &= \lambda_1\mathbf{u_1}\mathbf{u_1^T x} + \lambda_2\mathbf{u_2}\mathbf{u_2^T x} + \cdots + \lambda_n\mathbf{u_n}\mathbf{u_n^T x}\end{aligned}$$

which is the eigendecomposition equation. Whatever happens after the multiplication by **A** is true for all matrices, and does not need a symmetric matrix. We need an $n \times n$ symmetric matrix since it has n real eigenvalues plus n linear independent and orthogonal eigenvectors that can be used as a new basis for **x**. When you have a non-symmetric matrix you do not have such a combination. For example, suppose that you have a non-symmetric matrix:

$$\begin{bmatrix} 3 & -1 \\ 1 & 2 \end{bmatrix}$$

If you calculate the eigenvalues and eigenvectors of this matrix, you get:

```
lam= [2.5+0.866j 2.5-0.866j]
u= [[0.7071+0.j     0.7071-0.j    ]
    [0.3536-0.6124j 0.3536+0.6124j]]
```

which means you have no real eigenvalues to do the decomposition. Another example is:

$$\begin{bmatrix} 2 & 1 \\ 0 & 2 \end{bmatrix}$$

and you get:

```
lam= [2. 2.]
u= [[ 1. -1.]
    [ 0.  0.]]
```

Here the eigenvectors are not linearly independent. In fact **u1** = -**u2**. So you cannot reconstruct **A** like Figure 11 using only one eigenvector. In addition, it does not show a direction of stretching for this matrix as shown in Figure 14.

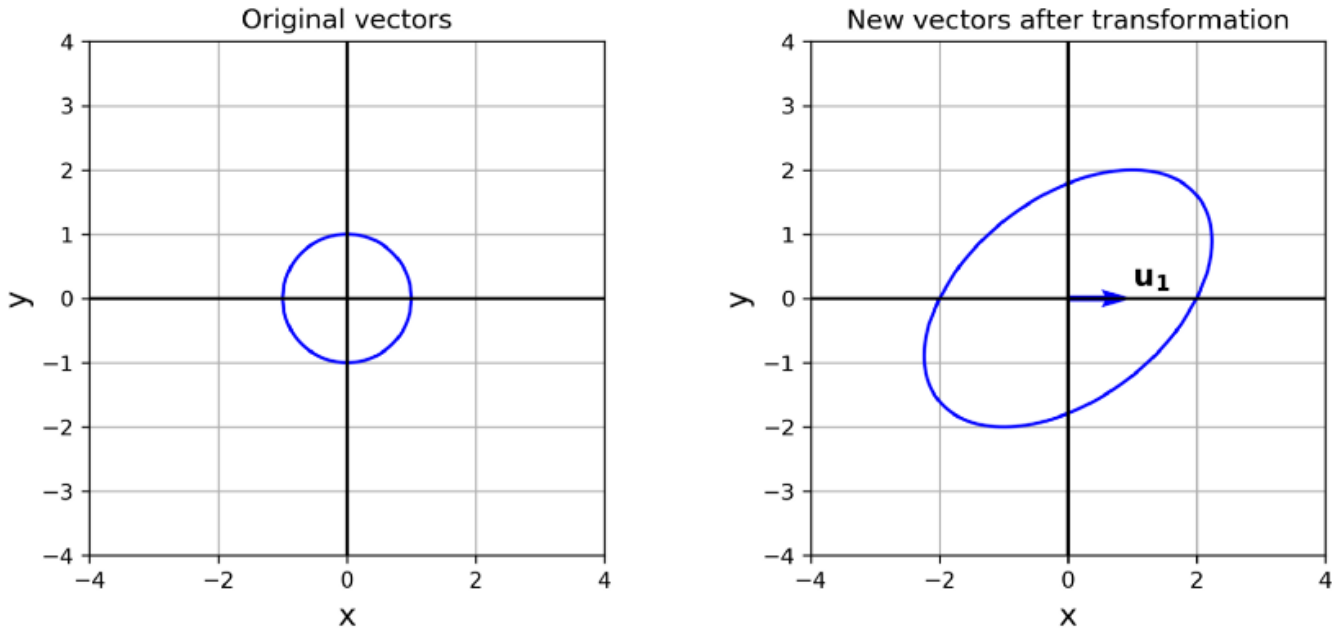


Figure 14

Finally, remember that for

$$\mathbf{A} = \begin{bmatrix} 3 & 2 \\ 0 & 2 \end{bmatrix}$$

we had:

```
lam= [ 7.8151 -2.8151]
u= [[ 0.639  -0.5667]
    [ 0.7692  0.8239]]
```

Here the eigenvectors are linearly independent, but they are not orthogonal (refer to Figure 3), and they do not show the correct direction of stretching for this matrix after transformation.

The eigendecomposition method is very useful, but only works for a symmetric matrix. A symmetric matrix is always a square matrix, so if you have a matrix that is not square, or a square but non-symmetric matrix, then you cannot use the eigendecomposition method to approximate it with other matrices. SVD can overcome this problem.

Singular Values

Before talking about SVD, we should find a way to calculate the stretching directions for a non-symmetric matrix. Suppose that **A** is an $m \times n$ matrix which is not necessarily symmetric. Then it can be shown that

$$\mathbf{A^T A}$$

is an $n \times n$ symmetric matrix. Remember that the transpose of a product is the product of the transposes in the reverse order. So

$$(\mathbf{A}^T\mathbf{A})^T = \mathbf{A}^T(\mathbf{A}^T)^T = \mathbf{A}^T\mathbf{A}$$

So **A^T A** is equal to its transpose, and it is a symmetric matrix. we want to calculate the stretching directions for a non-symmetric matrix., but how can we define the stretching directions mathematically?

So far, we only focused on the vectors in a 2-d space, but we can use the same concepts in an n-d space. Here I focus on a 3-d space to be able to visualize the concepts. Now the column vectors have 3 elements. Initially, we have a sphere that contains all the vectors that are one unit away from the origin as shown in Figure 15. If we call these vectors **x** then $||\mathbf{x}||=1$. Now if we multiply them by a 3×3 symmetric matrix, **Ax** becomes a 3-d oval. The first direction of stretching can be defined as the direction of the vector which has the greatest length in this oval (**Av1** in Figure 15). In fact, **Av1** is the maximum of $||\mathbf{Ax}||$ over all unit vectors **x**. This vector is the transformation of the vector **v1** by **A**.

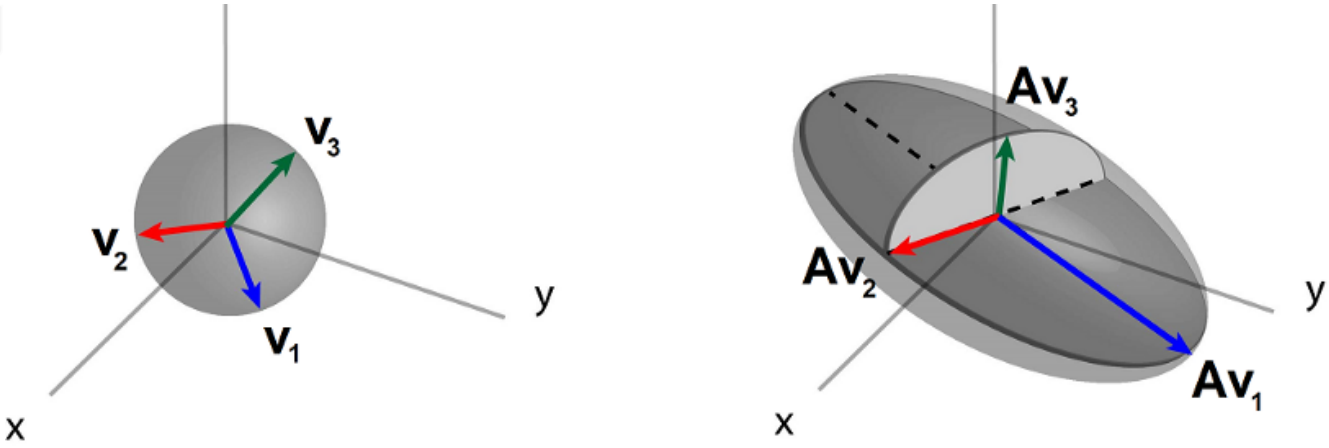


Figure 15

The second direction of stretching is along the vector **Av2**. **Av2** is the maximum of ||**Ax**|| over all vectors in **x** which are perpendicular to **v1**. So among all the vectors in **x,** we maximize ||**Ax**|| with this constraint that **x** is perpendicular to **v1**. Finally, **v3** is the vector that is perpendicular to both **v1** and **v2** and gives the greatest length of **Ax** with these constraints. The direction of **Av3** determines the third direction of stretching. So generally in an *n*-dimensional space, the *i*-th direction of stretching is the direction of the vector **Avi** which has the greatest length and is perpendicular to the previous (*i*-1) directions of stretching.

Now let **A** be an *m*×*n* matrix. We showed that **A^T A** is a symmetric matrix, so it has *n* real eigenvalues and *n* linear independent and orthogonal eigenvectors which can form a basis for the *n*-element vectors that it can transform (in **R^n** space). We call these eigenvectors **v1**, **v2**, … **vn** and we assume they are normalized. For each of these eigenvectors we can use the definition of length and the rule for the product of transposed matrices to have:

$$||\mathbf{Av_i}||^2 = (\mathbf{Av_i})^T \mathbf{Av_i} = \mathbf{v_i}^T \mathbf{A}^T \mathbf{Av_i}$$

Now we assume that the corresponding eigenvalue of **vi** is λi

$$\mathbf{v_i}^T \mathbf{A}^T \mathbf{Av_i} = \mathbf{v_i}^T \lambda_i \mathbf{v_i} = \lambda_i \mathbf{v_i}^T \mathbf{v_i}$$

But vi is normalized, so

$$||\mathbf{v_i}||^2 = \mathbf{v_i}^T \mathbf{v_i} = 1$$

As a result:

$$||\mathbf{Av_i}||^2 = \lambda_i \mathbf{v_i}^T \mathbf{v_i} = \lambda_i$$

This result shows that all the eigenvalues are positive. Now assume that we label them in decreasing order, so:

$$\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_n \geq 0$$

Now we define the singular value of **A** as the square root of *λi* (the eigenvalue of **A^T A**), and we denote it with *σi*.

$$\sigma_i = \sqrt{\lambda_i} = ||\mathbf{Av_i}||$$
$$\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_n \geq 0$$

So the singular values of **A** are the length of vectors **Avi**. Now we can summarize an important result which forms the backbone of the SVD method. It can be shown that the maximum value of ||**Ax**|| subject to the constraints

$$||\mathbf{x}|| = 1 \qquad \mathbf{x}^T\mathbf{v_1} = 0 \qquad \mathbf{x}^T\mathbf{v_2} = 0 \quad ... \quad \mathbf{x}^T\mathbf{v_{k-1}} = 0$$

is *σk*, and this maximum is attained at **vk**. For the constraints, we used the fact that when **x** is perpendicular to **vi**, their dot product is zero.

So if **vi** is the eigenvector of **A^T A** (ordered based on its corresponding singular value), and assuming that ||**x**||=1, then **Avi** is showing a direction of stretching for **Ax**, and the corresponding singular value *σi* gives the length of **Avi**.

The singular values can also determine the rank of **A**. Suppose that the number of non-zero singular values is *r*. Since they are positive and labeled in decreasing order, we can write them as

$$\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_r > 0$$

which correspond to

$$\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_r > 0$$

and each λi is the corresponding eigenvalue of **vi**. Then it can be shown that rank **A** which is the number of vectors that form the basis of **Ax** is r. It can be also shown that the set {**Av1**, **Av2**, …, **Avr**} is an orthogonal basis for **Ax** (the *Col* **A**). So the vectors **Avi** are perpendicular to each other as shown in Figure 15.

Now we go back to the non-symmetric matrix

$$\mathbf{A} = \begin{bmatrix} 3 & 2 \\ 0 & 2 \end{bmatrix}$$

We plotted the eigenvectors of **A** in Figure 3, and it was mentioned that they do not show the directions of stretching for **Ax**. In Figure 16 the eigenvectors of **A^T A** have been plotted on the left side (**v1** and **v2**). Since **A^T A** is a symmetric matrix, these vectors show the directions of stretching for it. On the right side, the vectors **Av1** and **Av2** have been plotted, and it is clear that these vectors show the directions of stretching for **Ax**.

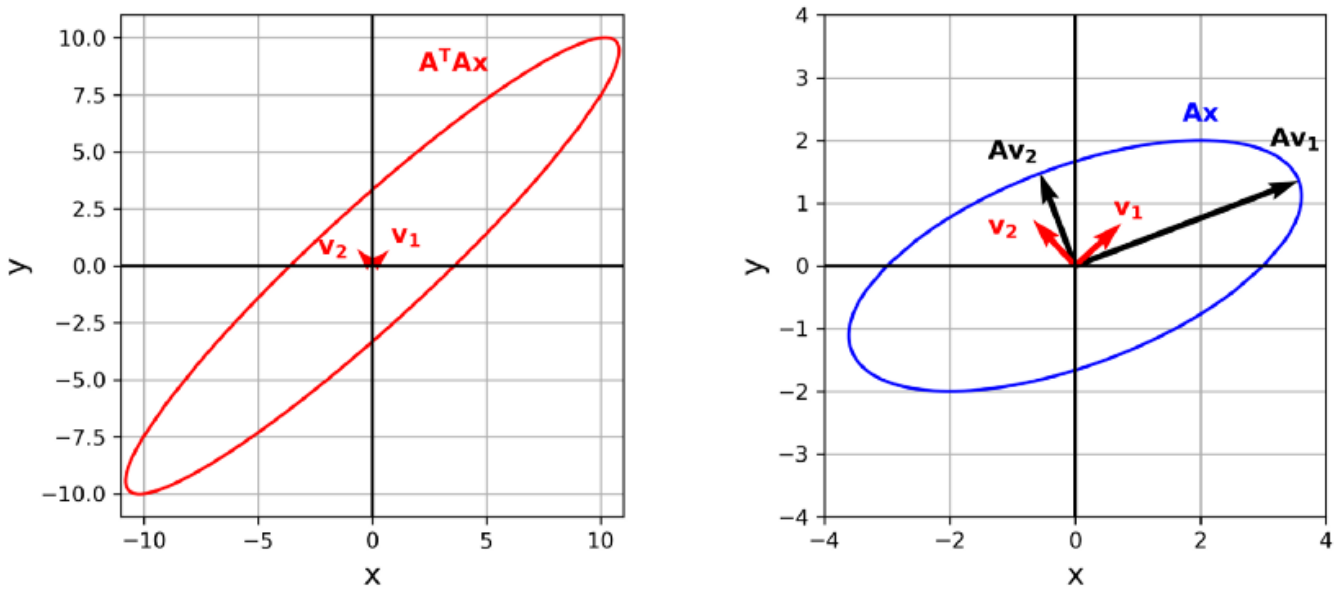


Figure 16

So **Avi** shows the direction of stretching of **A** no matter **A** is symmetric or not.

Now imagine that matrix **A** is symmetric and is equal to its transpose. In addition, suppose that its i-th eigenvector is **ui** and the corresponding eigenvalue is λi. If we multiply **A^T A** by **ui** we get:

$$(\mathbf{A}^\mathbf{T}\mathbf{A})\mathbf{u_i} = \mathbf{A}(\mathbf{A}\mathbf{u_i}) = \mathbf{A}\lambda_i\mathbf{u_i} = \lambda_i\mathbf{A}\mathbf{u_i} = \lambda_i^2\mathbf{u_i}$$

which means that **ui** is also an eigenvector of **A^T A**, but its corresponding eigenvalue is λi^2. So when **A** is symmetric, instead of calculating **Avi** (where **vi** is the eigenvector of **A^T A)** we can simply use **ui** (the eigenvector of **A**) to have the directions of stretching, and this is exactly what we did for the eigendecomposition process. Now that we know how to calculate the directions of stretching for a non-symmetric matrix, we are ready to see the SVD equation.

Singular Value Decomposition (SVD)

Let **A** be an $m \times n$ matrix and rank **A** $= r$. So the number of non-zero singular values of **A** is r. Since they are positive and labeled in decreasing order, we can write them as

$$\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_n$$
$$\mathbf{v_1} \quad \mathbf{v_2} \quad \cdots \quad \mathbf{v_n}$$

where

$$\sigma_{r+1} = \sigma_{r+2} = ... = \sigma_n = 0$$

We know that each singular value σi is the square root of the λi (eigenvalue of **A^TA**), and corresponds to an eigenvector **vi** with the same order. Now we can write the *singular value decomposition* of **A** as:

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\mathbf{T}$$

where **V** is an $n \times n$ matrix that its columns are **vi**. So:

$$\mathbf{V} = \begin{bmatrix} \mathbf{v_1} & \mathbf{v_2} & \dots & \mathbf{v_n} \end{bmatrix}$$

We call a set of orthogonal and normalized vectors an *orthonormal* set. So the set {**vi**} is an orthonormal set. A matrix whose columns are an orthonormal set is called an *orthogonal matrix*, and **V** is an orthogonal matrix.

Σ is an $m \times n$ diagonal matrix of the form:

$$\begin{bmatrix} \sigma_1 & 0 & \dots & 0 & 0 & \dots & 0 \\ 0 & \sigma_2 & \dots & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \dots & \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

$$\mathbf{\Sigma} = \underbrace{\left.\begin{bmatrix} 0 & 0 & \dots & \sigma_r & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & 0 & \dots & 0 \end{bmatrix}\right\} \text{m rows}}_{\text{n columns}}$$

So we first make an $r \times r$ diagonal matrix with diagonal entries of *σ1, σ2, …, σr*. Then we pad it with zero to make it an $m \times n$ matrix.

We also know that the set {**Av1**, **Av2**, …, **Avr**} is an orthogonal basis for *Col* **A,** and $\sigma i = ||\mathbf{Avi}||$. So we can normalize the **Avi** vectors by dividing them by their length:

$$\mathbf{u_i} = \frac{\mathbf{Av_i}}{||\mathbf{Av_i}||} = \frac{\mathbf{Av_i}}{\sigma_i} \qquad 1 < i < r$$

Now we have a set {**u1**, **u2**, …, **ur**} which is an orthonormal basis for **Ax** which is *r*-dimensional. We know that **A** is an $m \times n$ matrix, and the rank of **A** can be m at most (when all the columns of **A** are linearly independent). Since we need an $m \times m$ matrix for **U**, we add *(m-r)* vectors to the set of **ui** to make it a normalized basis for an m-dimensional space **R**^m (There are several methods that can be used for this purpose. For example we can use the *Gram-Schmidt Process.* However, explaining it is beyond the scope of this article). So now we have an orthonormal basis {**u1**, **u2**, … ,**um**}. These vectors will be the columns of **U** which is an orthogonal $m \times m$ matrix

$$\mathbf{U} = \begin{bmatrix} \mathbf{u_1} & \mathbf{u_2} & \dots & \mathbf{u_m} \end{bmatrix}$$

So in the end, we can decompose **A** as

$$\mathbf{A} = \begin{bmatrix} \mathbf{u_1} & \mathbf{u_2} & \dots & \mathbf{u_m} \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 & \dots & 0 & 0 & \dots & 0 \\ 0 & \sigma_2 & \dots & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \dots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & \sigma_r & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & 0 & \dots & 0 \end{bmatrix} \begin{bmatrix} \mathbf{v_1}^T \\ \mathbf{v_2}^T \\ \vdots \\ \mathbf{v_n}^T \end{bmatrix}$$

To better understand this equation, we need to simplify it:

$$\mathbf{A} = \begin{bmatrix} \mathbf{u_1} & \mathbf{u_2} & \dots & \mathbf{u_m} \end{bmatrix} \begin{bmatrix} \sigma_1 \mathbf{v_1}^T \\ \sigma_2 \mathbf{v_2}^T \\ \vdots \\ \sigma_r \mathbf{v_r}^T \\ 0 \\ \vdots \\ 0 \end{bmatrix} \} \text{ m rows}$$

$$= \sigma_1 \mathbf{u_1} \mathbf{v_1}^T + \sigma_2 \mathbf{u_2} \mathbf{v_2}^T + \ldots + \sigma_r \mathbf{u_r} \mathbf{v_r}^T$$

We know that *σi* is a scalar; **ui** is an m-dimensional column vector, and **vi** is an n-dimensional column vector. So each *σi***ui vi**^T is an $m \times n$ matrix, and the SVD equation decomposes the matrix **A** into *r* matrices with the same shape ($m \times n$).

First, let me show why this equation is valid. If we multiply both sides of the SVD equation by **x** we get:

$$\mathbf{Ax} = \sigma_1 \mathbf{u_1} \mathbf{v_1}^T \mathbf{x} + \sigma_2 \mathbf{u_2} \mathbf{v_2}^T \mathbf{x} + \ldots + \sigma_r \mathbf{u_r} \mathbf{v_r}^T \mathbf{x}$$

We know that the set {**u1**, **u2**, …, **ur**} is an orthonormal basis for **Ax**. So the vector **Ax** can be written as a linear combination of them.

$$\mathbf{Ax} = a_1 \mathbf{u_1} + a_2 \mathbf{u_2} + \ldots + a_r \mathbf{u_r}$$

and since **ui** vectors are orthogonal, each term *ai* is equal to the dot product of **Ax** and **ui** (scalar projection of **Ax** onto **ui**):

$$a_i = \mathbf{Ax}.\mathbf{u_i} = (\mathbf{Ax})^T \mathbf{u_i} = \mathbf{x}^T \mathbf{A}^T \mathbf{u_i}$$

but we also know that

$$\mathbf{u_i} = \frac{\mathbf{A v_i}}{\sigma_i}$$

So by replacing that into the previous equation, we have:

$$a_i = \mathbf{x}^T \mathbf{A}^T \frac{\mathbf{A}\mathbf{v_i}}{\sigma_i} = \frac{1}{\sigma_i} \mathbf{x}^T \mathbf{A}^T \mathbf{A}\mathbf{v_i}$$

We also know that **vi** is the eigenvector of **A**^T **A** and its corresponding eigenvalue *λi* is the square of the singular value *σi*

$$a_i = \frac{1}{\sigma_i} \mathbf{x}^T \mathbf{A}^T \mathbf{A}\mathbf{v_i} = \frac{1}{\sigma_i} \mathbf{x}^T \sigma_i^2 \mathbf{v_i} = \sigma_i \mathbf{x}^T \mathbf{v_i}$$

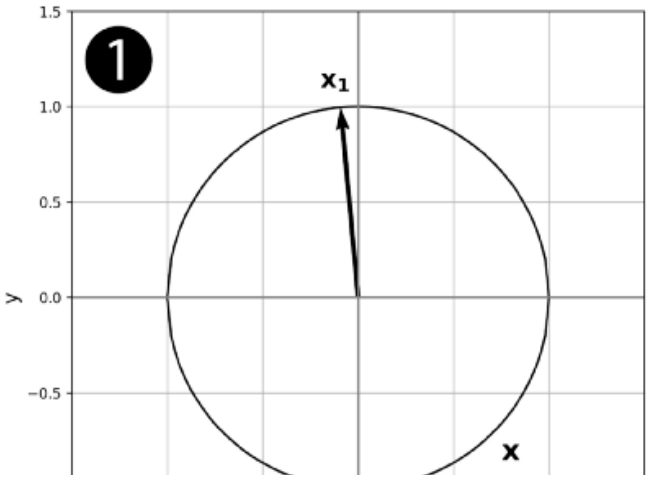
But dot product is commutative, so

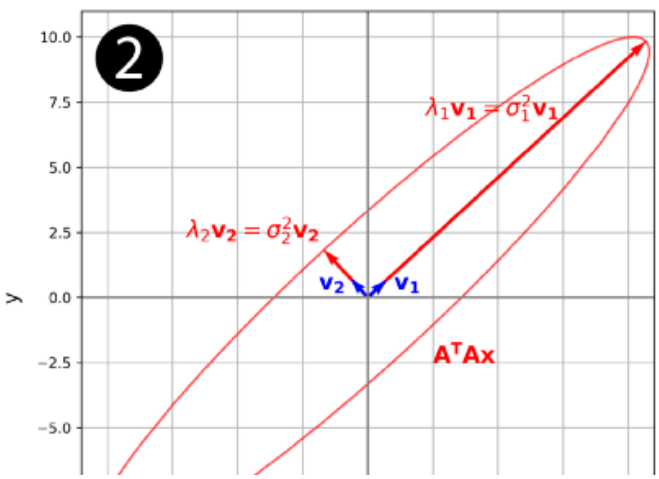
$$a_i = \sigma_i \mathbf{x}^T \mathbf{v_i} = \sigma_i \mathbf{v_i}^T \mathbf{x}$$

Notice that **vi**^**Tx** gives the scalar projection of **x** onto **vi**, and the length is scaled by the singular value. Now if we replace the *ai* value into the equation for **Ax**, we get the SVD equation:

$$\mathbf{A}\mathbf{x} = \sigma_1 \mathbf{u_1} \mathbf{v_1}^T \mathbf{x} + \sigma_2 \mathbf{u_2} \mathbf{v_2}^T \mathbf{x} + \ldots + \sigma_r \mathbf{u_r} \mathbf{v_r}^T \mathbf{x}$$

So each $ai = \sigma i \mathbf{vi}^T \mathbf{x}$ is the scalar projection of **Ax** onto **ui**, and if it is multiplied by **ui**, the result is a vector which is the orthogonal projection of **Ax** onto **ui**. The singular value *σi* scales the length of this vector along **ui**. Remember that in the eigendecomposition equation, each **ui ui**^**T** was a projection matrix that would give the orthogonal projection of **x** onto **ui**. Here *σi***vi** ^**T** can be thought as a projection matrix that takes **x**, but projects **Ax** onto **ui**. Since it projects all the vectors on **ui**, its rank is 1. Figure 17 summarizes all the steps required for SVD. We start by picking a random 2-d vector **x1** from all the vectors that have a length of 1 in **x** (Figure 17–1). Then we try to calculate **Ax1** using the SVD method.





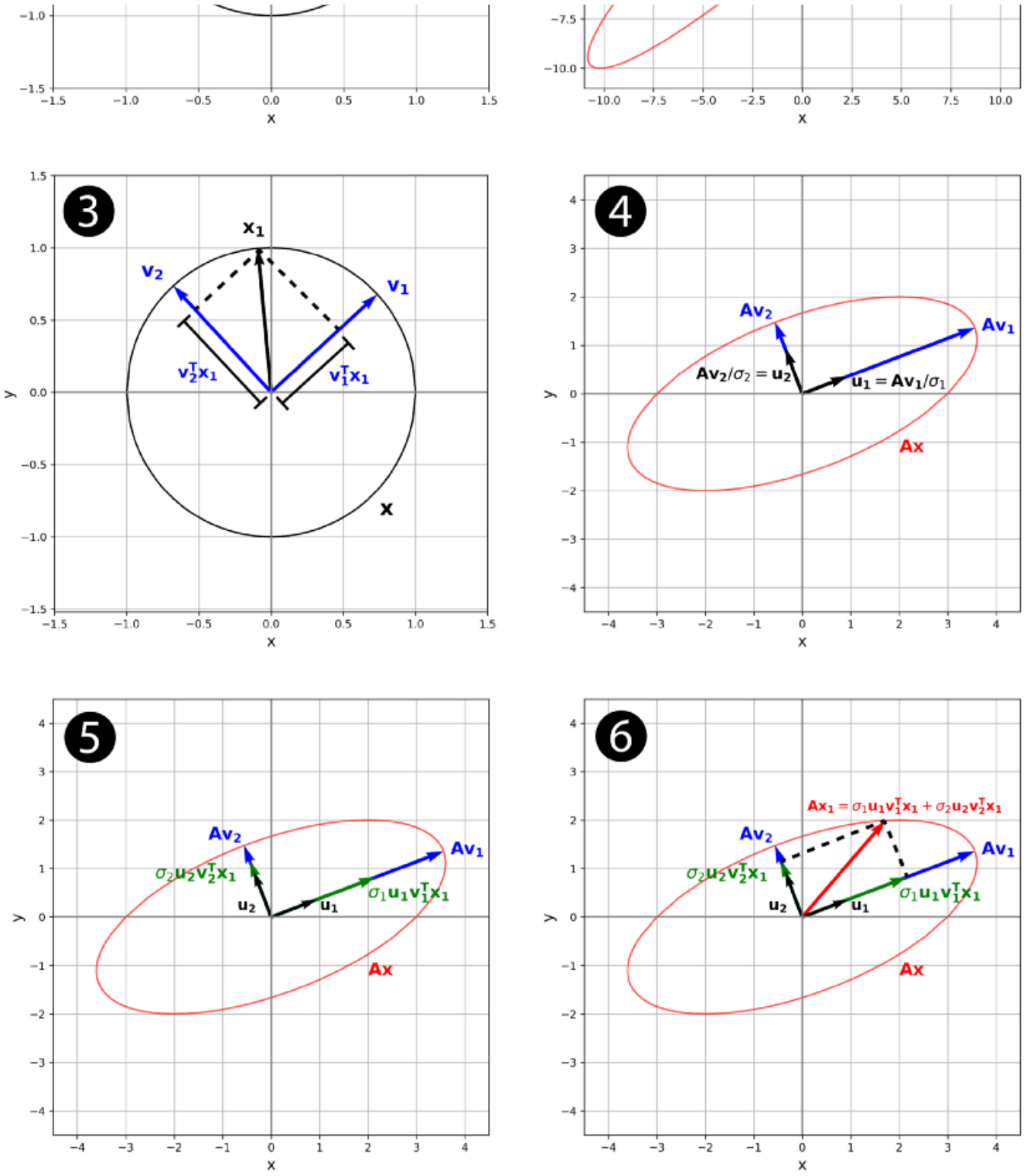


Figure 17

First, we calculate the eigenvalues ($\lambda 1, \lambda 2$) and eigenvectors (**v1**, **v2**) of **A^TA**. We know that the singular values are the square root of the eigenvalues ($\sigma i^2 = \lambda i$) as shown in (Figure 17–2). **Av1** and **Av2** show the directions of stretching of **Ax**, and **u1** and **u2** are the unit vectors of **Av1** and **Av2** (Figure 17–4). The orthogonal projection of **Ax1** onto **u1** and **u2** are

$$\sigma_1 \mathbf{u_1} \mathbf{v_1^T} \mathbf{x_1} \quad , \quad \sigma_2 \mathbf{u_2} \mathbf{v_2^T} \mathbf{x_1}$$

respectively (Figure 17–5), and by simply adding them together we get **Ax1**

$$\mathbf{Ax_1} = \sigma_1 \mathbf{u_1 v_1^T x_1} + \sigma_2 \mathbf{u_2 v_2^T x_1}$$

as shown in (Figure 17–6).

Here is an example showing how to calculate the SVD of a matrix in Python. We want to find the SVD of

$$\mathbf{A} = \begin{bmatrix} 4 & 1 & 3 \\ 8 & 3 & -2 \end{bmatrix}$$

This is a 2×3 matrix. So **x** is a 3-d column vector, but **Ax** is a not 3-dimensional vector, and **x** and **Ax** exist in different vector spaces. First, we calculate the eigenvalues and eigenvectors of **A^T A**.

```
# Listing 10

A = np.array([[4, 1, 3],
              [8, 3, -2]])
lam, v = LA.eig(A.T @ A)
print("lam=", np.round(lam, 4))
print("v=", np.round(v, 4))
```

SVD_Listing10.py hosted with ♡ by GitHub view raw

The output is:

```
lam= [90.1167  0.      12.8833]
v= [[ 0.9415  0.3228  0.0969]
    [ 0.3314 -0.9391 -0.0906]
    [-0.0617 -0.1174  0.9912]]
```

As you see the 2nd eigenvalue is zero. Since **A^T A** is a symmetric matrix and has two non-zero eigenvalues, its rank is 2. Figure 18 shows two plots of **A^T Ax** from different angles. Since the rank of **A^TA** is 2, all the vectors **A^TAx** lie on a plane.

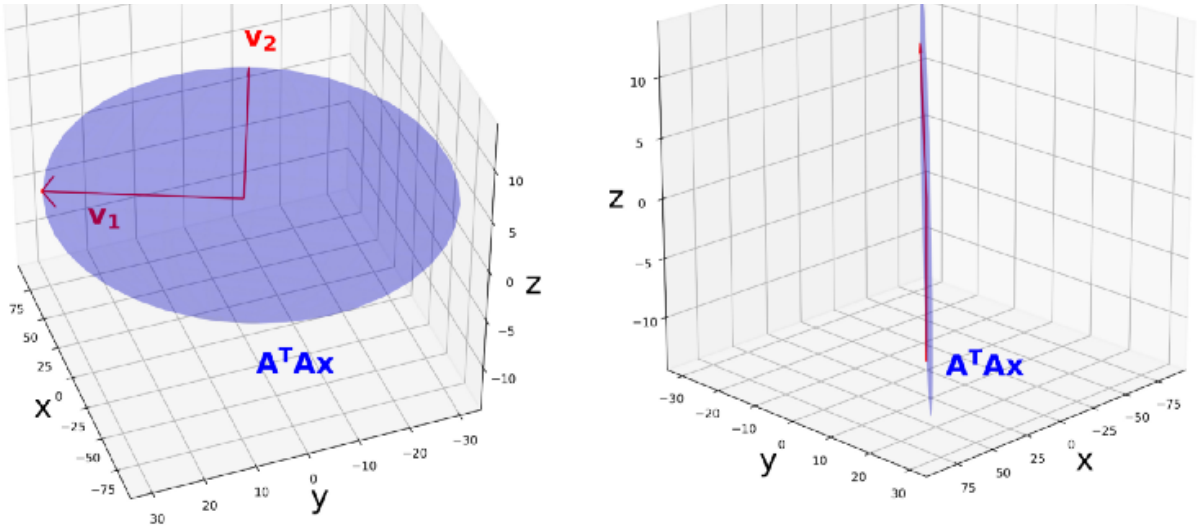


Figure 18

Listing 11 shows how to construct the matrices $\mathbf{\Sigma}$ and $\mathbf{V}$. We first sort the eigenvalues in descending order. The columns of $\mathbf{V}$ are the corresponding eigenvectors in the same order.

```python
# Listing 11

V = v[:, lam.argsort()[::-1]]

lam_sorted = np.sort(lam)[::-1]
lam_sorted = lam_sorted[lam_sorted > 1e-8]
sigma = np.sqrt(lam_sorted)
Sigma = np.zeros((A.shape[0], A.shape[1]))
Sigma[:min(A.shape[0],A.shape[1]), :min(A.shape[0],A.shape[1])] = np.diag(sigma)

print("Sigma=", np.round(Sigma, 4))
print("V=", np.round(V, 4))
```

SVD_Listing11.py hosted with ♡ by GitHub view raw

Then we filter the non-zero eigenvalues and take the square root of them to get the non-zero singular values. We know that $\mathbf{\Sigma}$ should be a 3×3 matrix. So we place the two non-zero singular values in a 2×2 diagonal matrix and pad it with zero to have a 3×3 matrix. The output is:

```
Sigma= [[9.493  0.     0.    ]
        [0.     3.5893 0.    ]]
V= [[ 0.9415  0.0969  0.3228]
    [ 0.3314 -0.0906 -0.9391]
    [-0.0617  0.9912 -0.1174]]
```

To construct **V**, we take the **vi** vectors corresponding to the r non-zero singular values of **A** and divide them by their corresponding singular values. Since **A** is a 2×3 matrix, **U** should be a 2×2 matrix. We have 2 non-zero singular values, so the rank of **A** is 2 and $r=2$. As a result, we already have enough **vi** vectors to form **U**.

```
# Listing 12

r = len(sigma)
U = A @ V[:,:r] / sigma
print("U=", np.round(U, 4))
```

SVD_Listing12.py hosted with ♡ by GitHub view raw

The output is:

```
U= [[ 0.4121  0.9111]
    [ 0.9111 -0.4121]]
```

Finally, we get the decomposition of **A**:

$$\mathbf{A} = \mathbf{U\Sigma V^T} =$$

$$\begin{bmatrix} 0.4121 & 0.9111 \\ 0.9111 & -0.4121 \end{bmatrix} \begin{bmatrix} 9.493 & 0 & 0 \\ 0 & 3.5893 & 0 \end{bmatrix} \begin{bmatrix} 0.9415 & 0.0969 & 0.3228 \\ 0.3314 & -0.0906 & -0.9391 \\ -0.0617 & 0.9912 & -0.1174 \end{bmatrix}^T$$

We really did not need to follow all these steps. NumPy has a function called `svd()` which can do the same thing for us. Listing 13 shows how we can use this function to calculate the SVD of matrix **A** easily.

```
# Listing 13

U, s, VT = LA.svd(A)
print("U=", np.round(U, 4))
print("s=", np.round(s, 4))
print("V", np.round(VT.T, 4))
```

SVD_Listing13.py hosted with ♡ by GitHub view raw

The output is:

```
U= [[-0.4121 -0.9111]
    [-0.9111  0.4121]]
s= [9.493  3.5893]
V [[-0.9415 -0.0969 -0.3228]
   [-0.3314  0.0906  0.9391]
   [ 0.0617 -0.9912  0.1174]]
```

You should notice a few things in the output. First, This function returns an array of non-zero singular values, not the matrix **Σ**. In addition, it returns **V^T,** not **V**, so I have printed the transpose of the array **VT** that it returns. Finally, the **ui** and **vi** vectors reported by `svd()` have the opposite sign of the **ui** and **vi** vectors that were calculated in Listing 10-12. Remember that if **vi** is an eigenvector for an eigenvalue, then (-1)**vi** is also an eigenvector for the same eigenvalue, and its length is also the same. So if **vi** is normalized, (-1)**vi** is normalized too. In fact, in Listing 10 we calculated **vi** with a different method and `svd()` is just reporting (-1)**vi** which is still correct. Since **ui**=**Avi**/σi, the set of **ui** reported by `svd()` will have the opposite sign too.

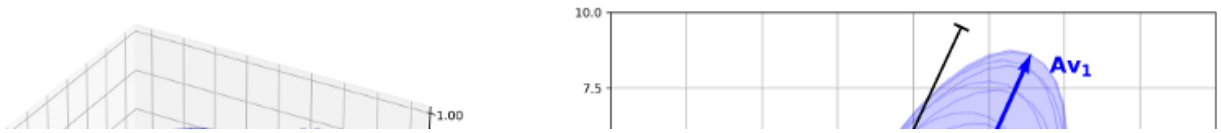
You can easily construct the matrix **Σ** and check that multiplying these matrices gives **A.**

```
# Listing 14

Sigma = np.zeros((A.shape[0], A.shape[1]))
Sigma[:min(A.shape[0],A.shape[1]), :min(A.shape[0],A.shape[1])] = np.diag(s)
A_reconstructed = U @ Sigma @ VT
print("Reconstructed A=", A_reconstructed)
```

SVD_Listing14.py hosted with ♡ by GitHub view raw

```
Reconstructed A= [[ 4.  1.  3.]
                  [ 8.  3. -2.]]
```

In Figure 19, you see a plot of **x** which is the vectors in a unit sphere and **Ax** which is the set of 2-d vectors produced by **A**. The vectors **u1** and **u2** show the directions of stretching. The ellipse produced by **Ax** is not hollow like the ones that we saw before (for example in Figure 6), and the transformed vectors fill it completely.



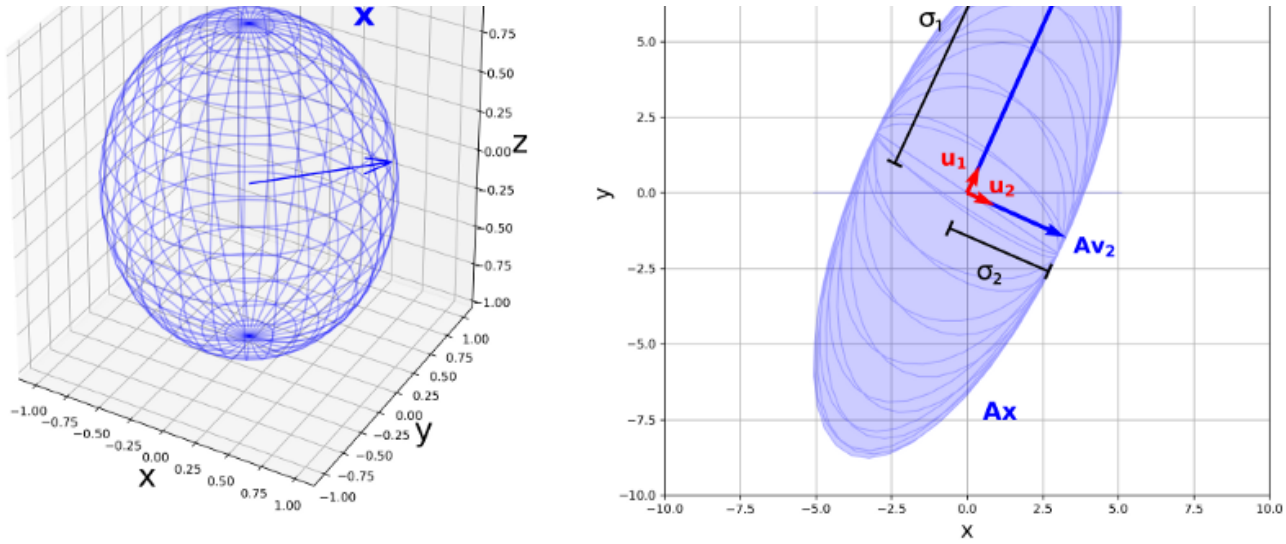


Figure 19

Similar to the eigendecomposition method, we can approximate our original symmetric matrix **A** by summing the terms which have the highest singular values. So we can use the first *k* terms in the SVD equation, using the *k* highest singular values which means we only include the first *k* vectors in **U** and **V** matrices in the decomposition equation:

$$\mathbf{A_k} = \begin{bmatrix} \mathbf{u_1} & \mathbf{u_2} & \dots & \mathbf{u_k} \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 & \dots & 0 \\ 0 & \sigma_2 & \dots & 0 \\ \vdots & \vdots & \dots & \vdots \\ 0 & 0 & \dots & \sigma_k \end{bmatrix} \begin{bmatrix} \mathbf{v_1}^T \\ \mathbf{v_2}^T \\ \vdots \\ \mathbf{v_k}^T \end{bmatrix}$$

$$= \sigma_1 \mathbf{u_1} \mathbf{v_1}^T + \sigma_2 \mathbf{u_2} \mathbf{v_2}^T + \dots + \sigma_k \mathbf{u_k} \mathbf{v_k}^T$$

We know that the set {**u1**, **u2**, …, **ur**} forms a basis for **Ax**. So when we pick *k* vectors from this set, **Ak x** is written as a linear combination of **u1, u2, … uk**. So they span **Ak x** and since they are linearly independent they form a basis for **Ak x** (or *col* **A**). So the rank of **Ak** is *k*, and by picking the first *k* singular values, we approximate **A** with a rank-*k* matrix.

As an example, suppose that we want to calculate the SVD of matrix

$$\mathbf{A} = \begin{bmatrix} 4 & 1 & 3 \\ 8 & 7 & -2 \\ 1 & 5 & 4 \end{bmatrix}$$

Again **x** is the vectors in a unit sphere (Figure 19 left). The singular values are $\sigma 1 = 11.97$, $\sigma 2 = 5.57$, $\sigma 3 = 3.25$, and the rank of **A** is 3. So **Ax** is an ellipsoid in 3-d space

as shown in Figure 20 (left). If we approximate it using the first singular value, the rank of **Ak** will be one and **Ak** multiplied by **x** will be a line (Figure 20 right). If we only use the first two singular values, the rank of **Ak** will be 2 and **Ak** multiplied by **x** will be a plane (Figure 20 middle).

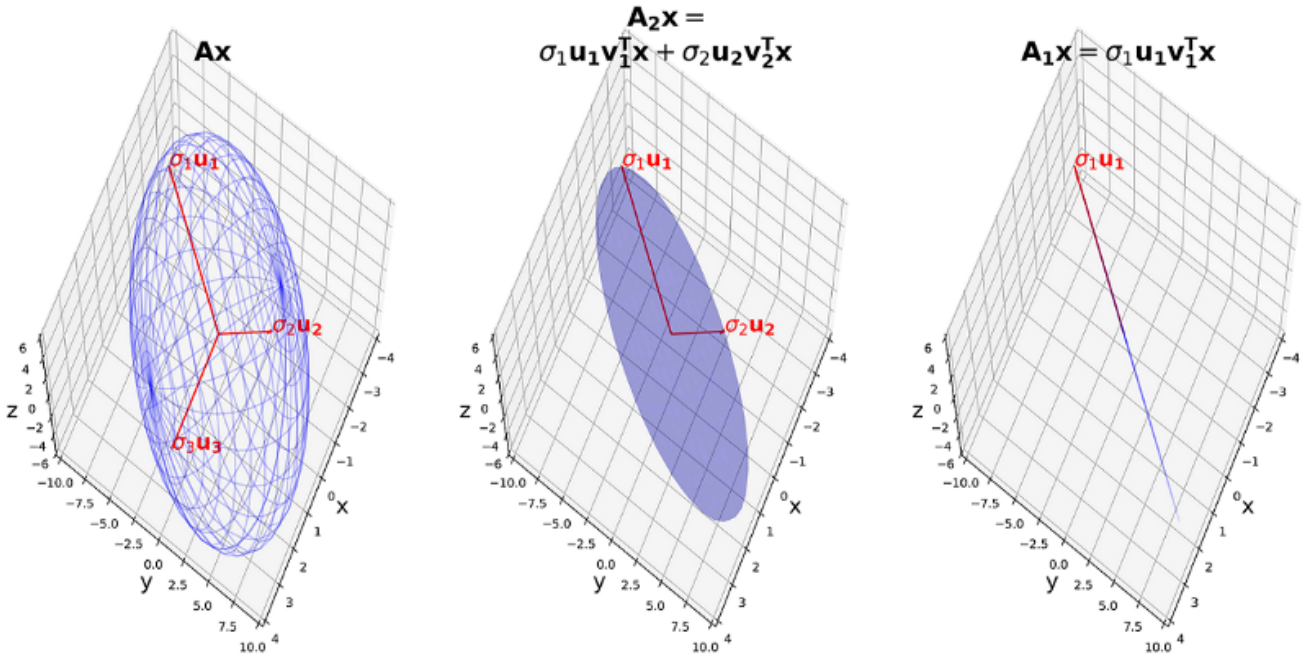


Figure 20

It is important to note that if we have a symmetric matrix, the SVD equation is simplified into the eigendecomposition equation. Suppose that the symmetric matrix **A** has eigenvectors **vi** with the corresponding eigenvalues λi. So we

$$\mathbf{A}\mathbf{v_i} = \lambda_i \mathbf{v_i}$$

We already showed that for a symmetric matrix, **vi** is also an eigenvector of **A^TA** with the corresponding eigenvalue of λi^2. So the singular values of **A** are the square root of λi^2 and $\sigma i = \lambda i$. now we can calculate **ui**:

$$\mathbf{u_i} = \frac{\mathbf{A}\mathbf{v_i}}{\sigma_i} = \frac{\lambda_i \mathbf{v_i}}{\sigma_i} = \frac{\lambda_i \mathbf{v_i}}{\lambda_i} = \mathbf{v_i}$$

So **ui** is the eigenvector of **A** corresponding to λi (and σi). Now we can simplify the SVD equation to get the eigendecomposition equation:

$$\mathbf{A} = \sum_i \sigma_i \mathbf{u_i} \mathbf{v_i}^T = \sum_i \lambda_i \mathbf{u_i} \mathbf{u_i}^T$$

Finally, it can be shown that SVD is the best way to approximate **A** with a rank-*k* matrix. The *Frobenius norm* of an $m \times n$ matrix **A** is defined as the square root of the sum of the absolute squares of its elements:

$$\|\mathbf{A}\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2}$$

So this is like the generalization of the vector length for a matrix. Now if the $m \times n$ matrix **Ak** is the approximated rank-*k* matrix by SVD, we can think of

$$\|\mathbf{A} - \mathbf{A_k}\|_F$$

as the distance between **A** and **Ak**. The smaller this distance, the better **Ak** approximates **A**. Now if **B** is any $m \times n$ rank-*k* matrix, it can be shown that

$$\|\mathbf{A} - \mathbf{A_k}\|_F \leq \|\mathbf{A} - \mathbf{B}\|_F$$

In other words, the difference between **A** and its rank-*k* approximation generated by SVD has the minimum Frobenius norm, and no other rank-*k* matrix can give a better approximation for **A** (with a closer distance in terms of the Frobenius norm).

Now that we are familiar with SVD, we can see some of its applications in data science.

Applications

Dimensionality reduction

We can store an image in a matrix. Every image consists of a set of pixels which are the building blocks of that image. Each pixel represents the color or the intensity of light in a specific location in the image. In a grayscale image with PNG format, each pixel has a value between 0 and 1, where zero corresponds to black and 1 corresponds to white. So a grayscale image with $m \times n$ pixels can be stored in an $m \times n$ matrix or NumPy array. Here we use the `imread()` function to load a grayscale image of Einstein which has 480 $\times$ 423 pixels into a 2-d array. Then we use SVD to decompose the matrix and reconstruct it using the first 30 singular values.

```
# Listing 15

```

```python
# Reading the image
mat = plt.imread("Picture.png")

# SVD
U, s, VT = LA.svd(mat)

Sigma = np.zeros((mat.shape[0], mat.shape[1]))
Sigma[:min(mat.shape[0], mat.shape[1]), :min(mat.shape[0], mat.shape[1])] = np.diag(s)

# Reconstruction of the matrix using the first 30 singular values
k = 30
mat_approx = U[:, :k] @ Sigma[:k, :k] @ VT[:k, :]

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10,8))
plt.subplots_adjust(wspace=0.3, hspace=0.2)

ax1.imshow(mat, cmap='gray')
ax1.set_title("Original image")

ax2.imshow(mat_approx, cmap='gray')
ax2.set_title("Reconstructed image using the \n first {} singular values".format(k))
plt.show()
```

SVD_Listing15.py hosted with ♡ by GitHub view raw

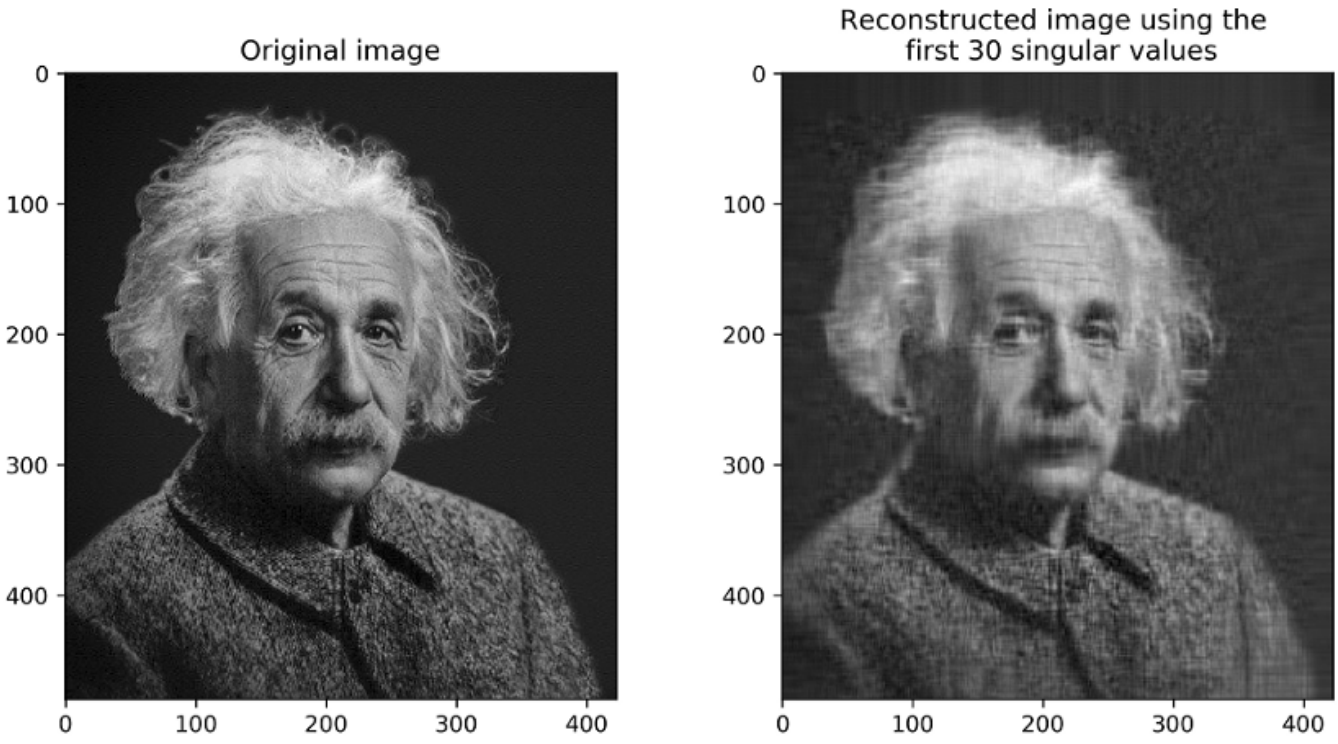


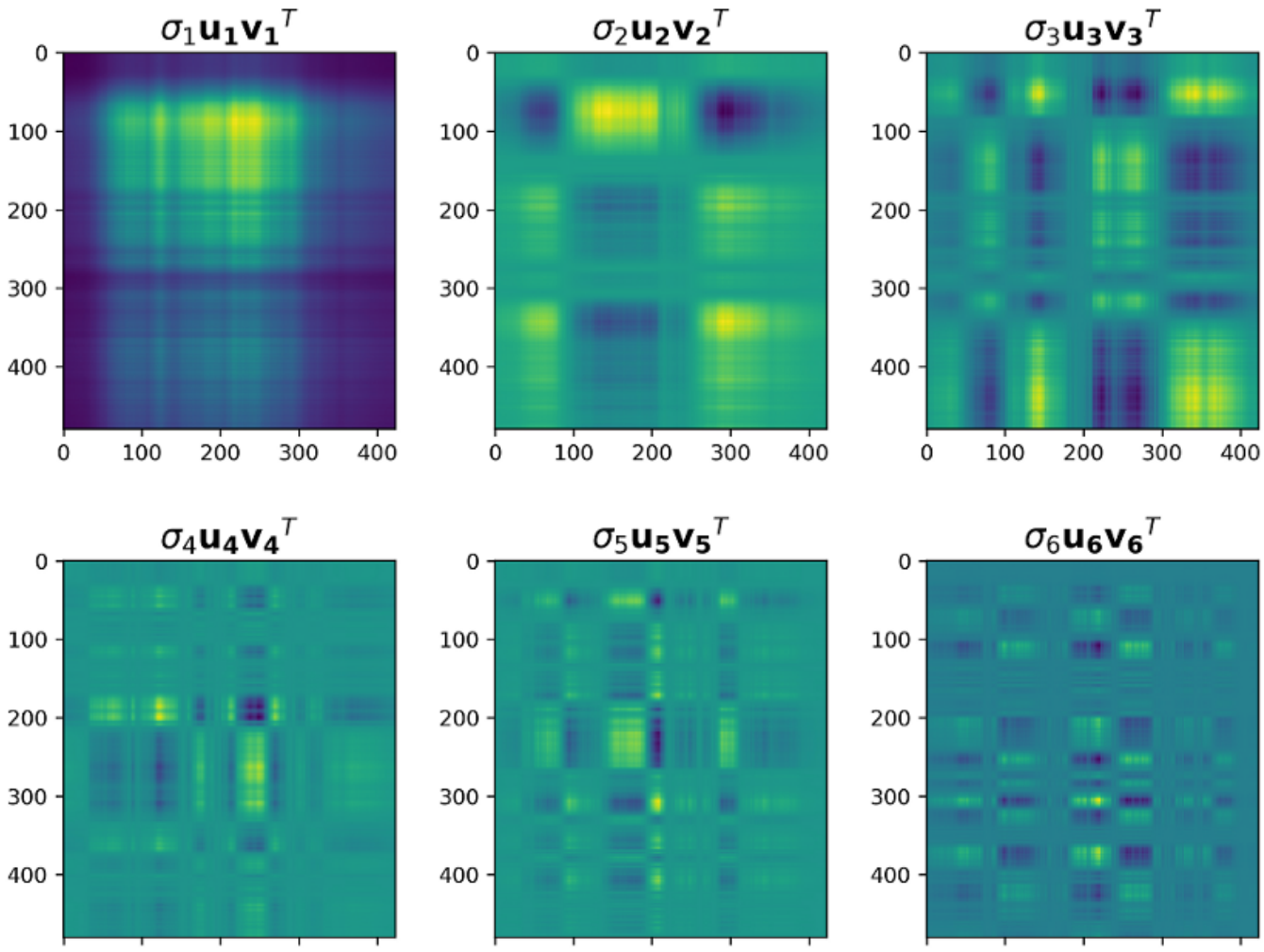
Figure 21. Image source

The original matrix is 480×423. So we need to store 480×423=203040 values. After SVD each **ui** has 480 elements and each **vi** has 423 elements. To be able to reconstruct

the image using the first 30 singular values we only need to keep the first 30 σi, **ui**, and **vi** which means storing 30×(1+480+423)=27120 values. This is roughly 13% of the number of values required for the original image. So using SVD we can have a good approximation of the original image and save a lot of memory. Listing 16 and calculates the matrices corresponding to the first 6 singular values. Each matrix *σi***ui vi ^T** has a rank of 1 and has the same number of rows and columns as the original matrix. Figure 22 shows the result.

```python
# Listing 16
fig, axes = plt.subplots(2, 3, figsize=(10,8))
plt.subplots_adjust(wspace=0.3, hspace=0.2)

for i in range(0, 6):
    mat_i = s[i] * U[:,i].reshape(-1,1) @ VT[i,:].reshape(1,-1)
    axes[i // 3, i % 3].imshow(mat_i)
    axes[i // 3, i % 3].set_title("$\sigma_{0}\mathbf{{u_{0}}}\mathbf{{v_{0}}}^T$".format(i+1),

plt.show()
```

SVD_Listing16.py hosted with ♡ by GitHub view raw



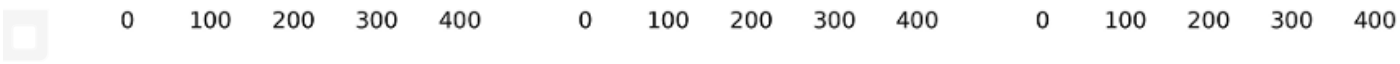


Figure 22

Please note that unlike the original grayscale image, the value of the elements of these rank-1 matrices can be greater than 1 or less than zero, and they should not be interpreted as a grayscale image. So I did not use `cmap='gray'` and did not display them as grayscale images. When plotting them we do not care about the absolute value of the pixels. Instead, we care about their values relative to each other.

To understand how the image information is stored in each of these matrices, we can study a much simpler image. In Listing 17, we read a binary image with five simple shapes: a rectangle and 4 circles. The result is shown in Figure 23.

```
# listing 17
# Reading the image
mat = plt.imread("shapes.png")

# SVD
U, s, VT = LA.svd(mat)

Sigma = np.zeros((mat.shape[0], mat.shape[1]))
Sigma[:min(mat.shape[0], mat.shape[1]), :min(mat.shape[0], mat.shape[1])] = np.diag(s)

fig, axes = plt.subplots(2, 2, figsize=(10,8))
plt.subplots_adjust(wspace=0.3, hspace=0.2)

axes[0, 0].imshow(mat, cmap='gray')
axes[0, 0].set_title("Original image")

for i in range(1, 4):
    k = i * 2
    # Reconstruction of the matrix using the first k singular values
    mat_approx = U[:, :k] @ Sigma[:k, :k] @ VT[:k, :]

    axes[i // 2, i % 2].imshow(mat_approx, cmap='gray')
    axes[i // 2, i % 2].set_title("Reconstructed image using the \n first {} singular values".fo

plt.show()
```

SVD_Listing17.py hosted with ♡ by GitHub view raw

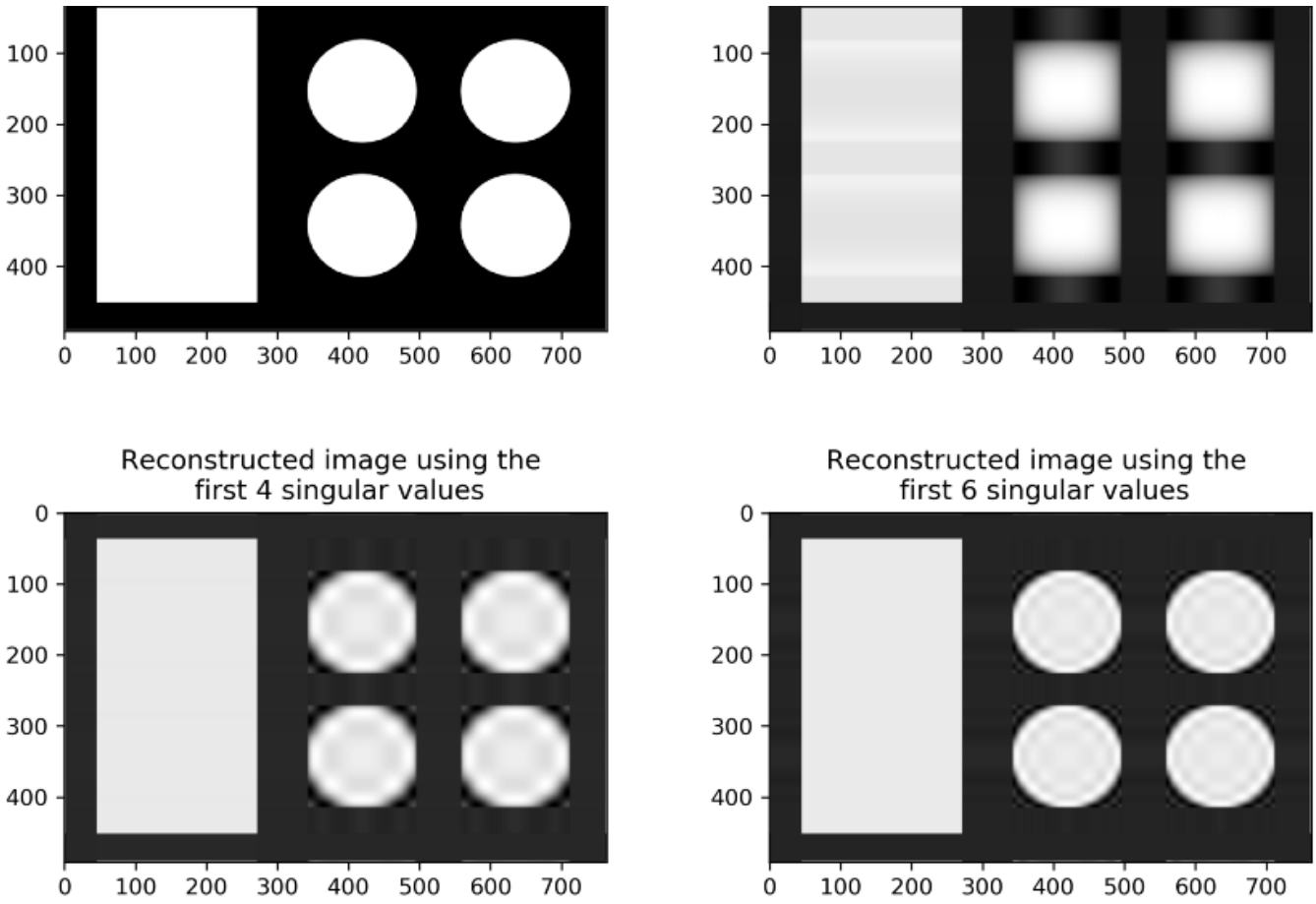
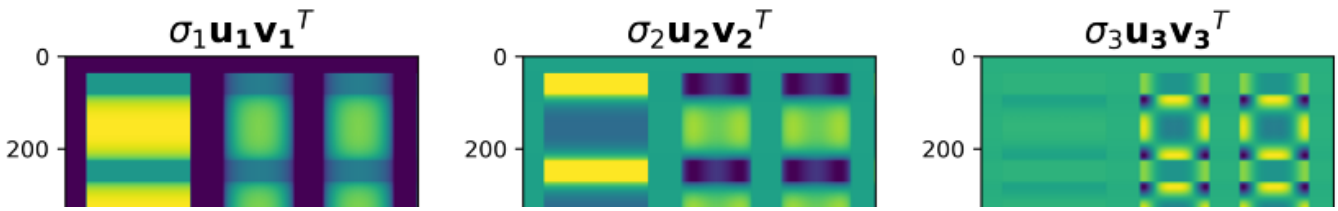


Figure 23

The image has been reconstructed using the first 2, 4, and 6 singular values. Now we plot the matrices corresponding to the first 6 singular values:

```python
# Listing 18
fig, axes = plt.subplots(2, 3, figsize=(10,6))
plt.subplots_adjust(wspace=0.3, hspace=0.05)

for i in range(0, 6):
    mat_i = s[i] * U[:,i].reshape(-1,1) @ VT[i,:].reshape(1,-1)
    #mat_i[mat_i < 1e-8] = 0
    axes[i // 3, i % 3].imshow(mat_i)
    axes[i // 3, i % 3].set_title("$\sigma_{0}\mathbf{{u_{0}}}\mathbf{{v_{0}}}^T$".format(i+1),

plt.show()
```

SVD_Listing18.py hosted with ♡ by GitHub view raw



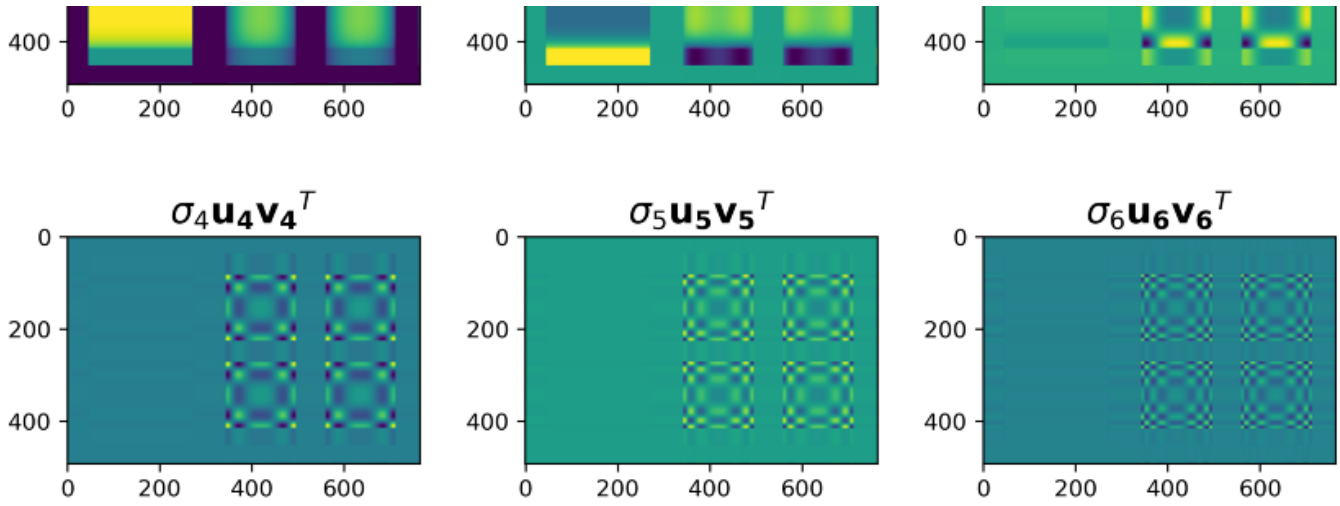


Figure 24

Each matrix (*σi* **ui vi ^T**) has a rank of 1 which means it only has one independent column and all the other columns are a scalar multiplication of that one. So if call the independent column **c1** (or it can be any of the other column), the columns have the general form of:

$$\sigma_i \mathbf{u_i v_i^T} = \mathbf{u_i} \begin{bmatrix} \sigma_i v_{i1} & \sigma_i v_{i2} & \dots & \sigma_i v_{in} \end{bmatrix} = \begin{bmatrix} \sigma_i v_{i1} \mathbf{u_i} & \sigma_i v_{i2} \mathbf{u_i} & \dots & \sigma_i v_{in} \mathbf{u_i} \end{bmatrix}$$
$$= \begin{bmatrix} \mathbf{c_1} & a_2 \mathbf{c_2} & \dots & a_n \mathbf{c_n} \end{bmatrix}$$

where *ai* is a scalar multiplier. In addition, this matrix projects all the vectors on **ui**, so every column is also a scalar multiplication of **ui**. This can be seen in Figure 25. Two columns of the matrix *σ2***u2 v2^T** are shown versus **u2**. Both columns have the same pattern of **u2** with different values (*ai* for column #300 has a negative value).

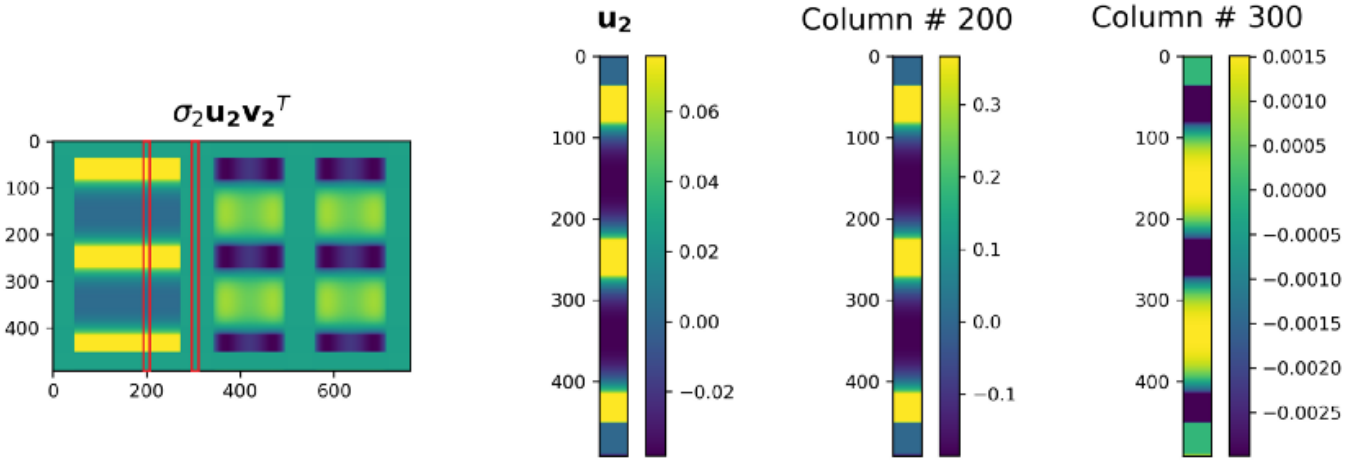


Figure 25

So using the values of **c1** and *ai* (or **u2** and its multipliers), each matrix captures some details of the original image. In figure 24, the first 2 matrices can capture almost all the information about the left rectangle in the original image. The 4 circles are roughly captured as four rectangles in the first 2 matrices in Figure 24, and more details on them are added in the last 4 matrices. This can be also seen in Figure 23 where the circles in the reconstructed image become rounder as we add more singular values. These rank-1 matrices may look simple, but they are able to capture some information about the repeating patterns in the image. For example in Figure 26, we have the image of the national monument of Scotland which has 6 pillars (in the image), and the matrix corresponding to the first singular value can capture the number of pillars in the original image.

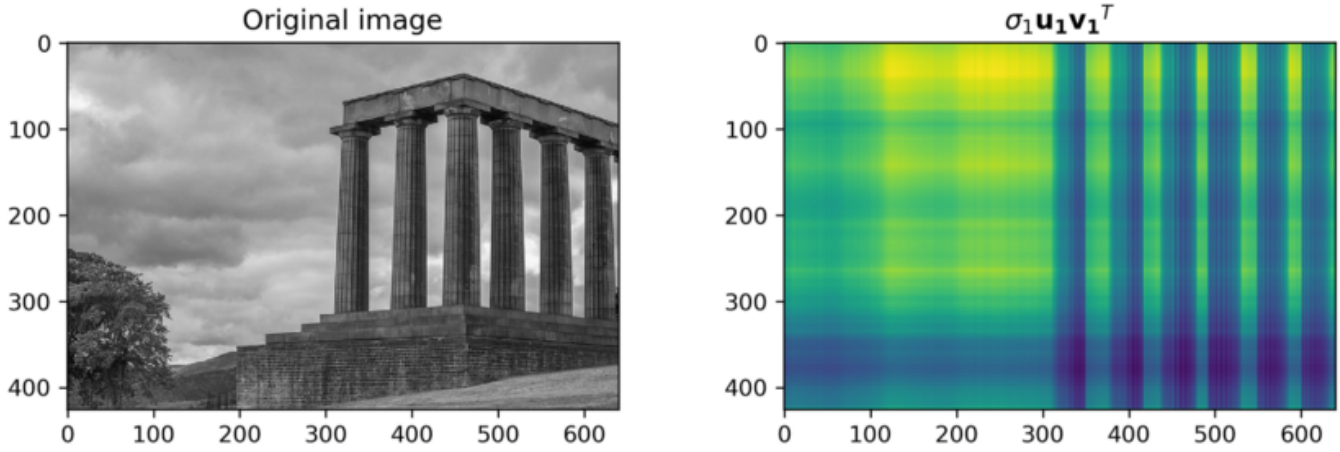


Figure 26. Image source

Eigenfaces

In this example, we are going to use the Olivetti faces dataset in the Scikit-learn library. This data set contains 400 images. The images were taken between April 1992 and April 1994 at AT&T Laboratories Cambridge. The images show the face of 40 distinct subjects. For some subjects, the images were taken at different times, varying the lighting, facial expressions, and facial details. These images are grayscale and each image has 64×64 pixels. The intensity of each pixel is a number on the interval [0, 1]. First, we load the dataset:

```
# Listing 19
data = fetch_olivetti_faces()
imgs = data.images
print(imgs.shape)
```

SVD_Listing19.py hosted with ♡ by GitHub view raw

The `fetch_olivetti_faces()` function has been already imported in Listing 1. We call it to read the data and stores the images in the `imgs` array. This is a (400, 64, 64) array which contains 400 grayscale 64×64 images. We can show some of them as an example here:

```
# Listing 20
fig, axes = plt.subplots(1, 5, figsize=(14, 8))
plt.subplots_adjust(wspace=0.4)

for i in range(0, 5):
    axes[i].imshow(imgs[i*40], cmap='gray')

plt.show()
```

SVD_Listing20.py hosted with ♡ by GitHub view raw

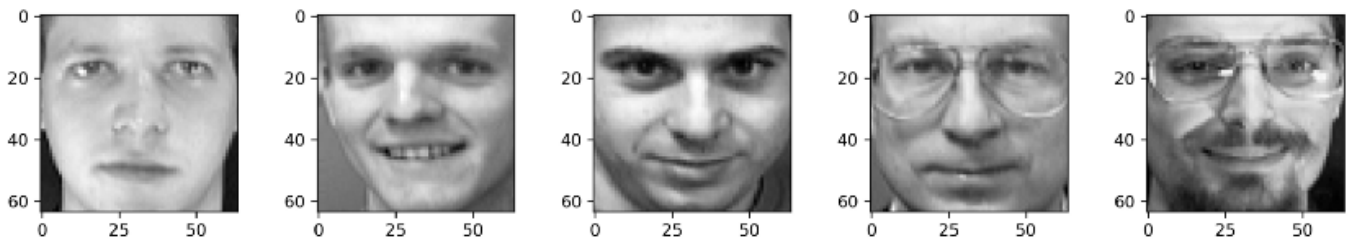


Figure 27

In the previous example, we stored our original image in a matrix and then used SVD to decompose it. Here we take another approach. We know that we have 400 images, so we give each image a label from 1 to 400. Now we use one-hot encoding to represent these labels by a vector. We use a column vector with 400 elements. For each label k, all the elements are zero except the k-th element. So label k will be represented by the vector:

$$\mathbf{i_k} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \qquad \mathbf{i_k}[k] = 1$$

Now we store each image in a column vector. Each image has $64 \times 64 = 4096$ pixels. So we can flatten each image and place the pixel values into a column vector $\mathbf{f}$ with 4096

elements as shown in Figure 28:

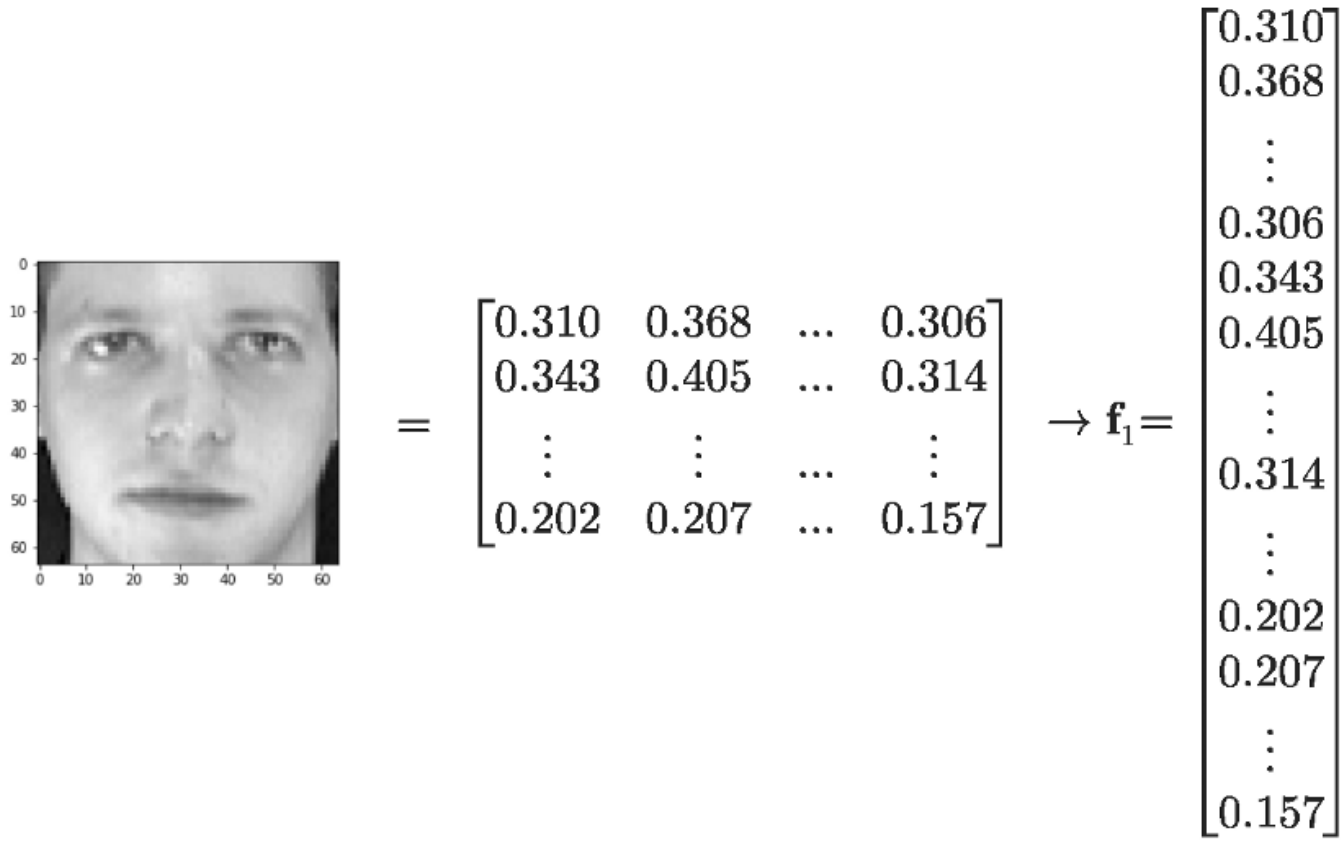
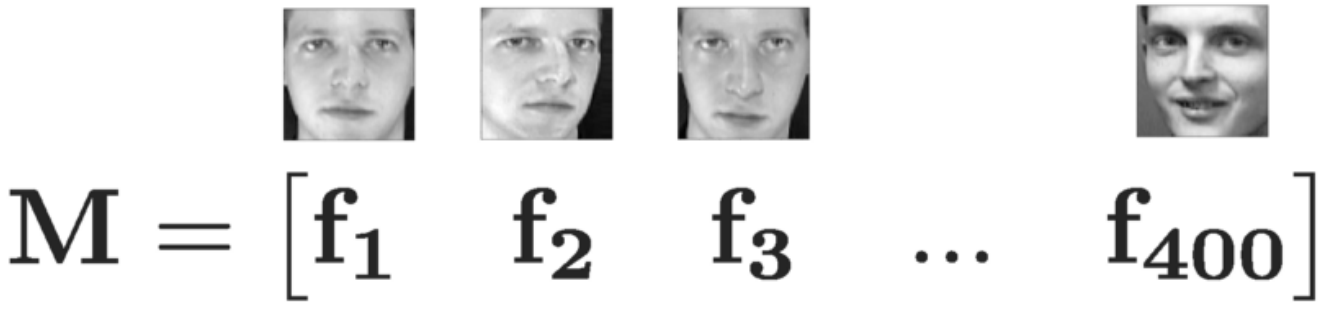


Figure 28

So each image with label *k* will be stored in the vector **fk**, and we need 400 **fk** vectors to keep all the images. Now we define a transformation matrix **M** which transforms the label vector **ik** to its corresponding image vector **fk**. The vectors **fk** will be the columns of matrix **M**:



This matrix has 4096 rows and 400 columns. We can simply use **y=Mx** to find the corresponding image of each label (**x** can be any vectors **ik**, and **y** will be the corresponding **fk**). For example for the third image of this dataset, the label is 3, and all the elements of **i3** are zero except the third element which is 1. Now, remember the

multiplication of partitioned matrices. When we multiply $\mathbf{M}$ by $\mathbf{i3}$, all the columns of $\mathbf{M}$ are multiplied by zero except the third column $\mathbf{f3}$, so:

$$\mathbf{y} = \mathbf{Mx} = \mathbf{Mi_3} = \mathbf{f_3}$$

$$\mathbf{y} = \begin{bmatrix} \mathbf{f_1} & \mathbf{f_2} & \mathbf{f_3} & \dots & \mathbf{f_{400}} \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix} = \mathbf{f_3}$$

Listing 21 shows how we can construct $\mathbf{M}$ and use it to show a certain image from the dataset.

```
# Listing 21
M = imgs.reshape((-1, imgs.shape[1]*imgs.shape[2])).T

x= np.zeros((400, 1))
x[50, 0] = 1
y = M @ x
plt.imshow(y.reshape((64,64)), cmap='gray')
plt.show()
```

SVD_Listing21.py hosted with ♡ by GitHub view raw

The length of each label vector $\mathbf{ik}$ is one and these label vectors form a standard basis for a 400-dimensional space. In this space, each axis corresponds to one of the labels with the restriction that its value can be either zero or one. The vectors $\mathbf{fk}$ live in a 4096-dimensional space in which each axis corresponds to one pixel of the image, and matrix $\mathbf{M}$ maps $\mathbf{ik}$ to $\mathbf{fk}$. Now we can use SVD to decompose $\mathbf{M}$. Remember that when we decompose $\mathbf{M}$ (with rank r) to

$$\mathbf{M} = \mathbf{U\Sigma V^T}$$

the set {**u1**, **u2**, …, **ur**} which are the first *r* columns of **U** will be a basis for **Mx**. Each vector **ui** will have 4096 elements. Since **y**=**Mx** is the space in which our image vectors live, the vectors **ui** form a basis for the image vectors as shown in Figure 29. In this figure, I have tried to visualize an *n*-dimensional vector space. This is, of course, impossible when $n \geq 3$, but this is just a fictitious illustration to help you understand this method.

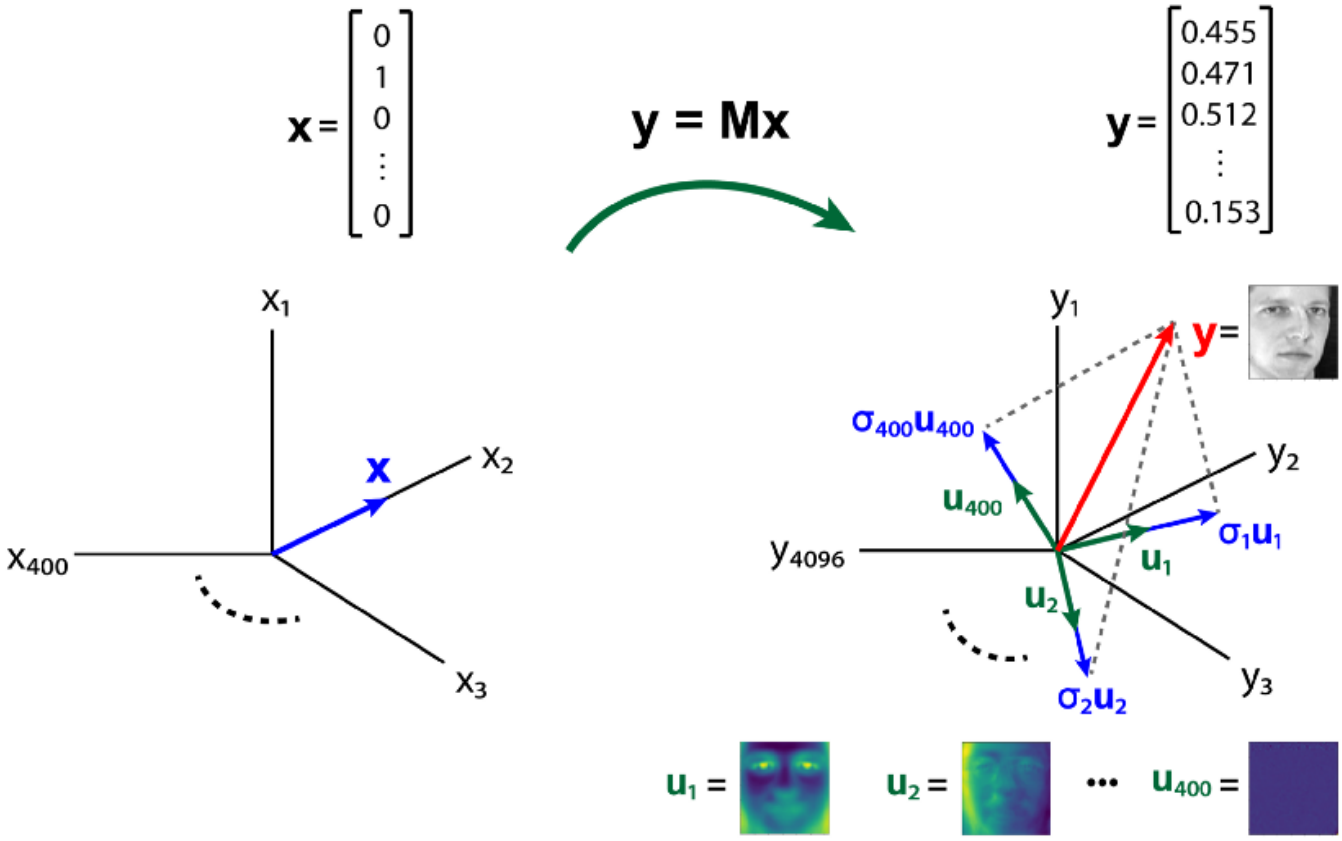


Figure 29

So we can reshape **ui** into a 64 ×64 pixel array and try to plot it like an image. The value of the elements of these vectors can be greater than 1 or less than zero, and when reshaped they should not be interpreted as a grayscale image. So I did not use `cmap='gray'` when displaying them.

```
# Listing 22
U, s, VT = LA.svd(M)

fig, axes = plt.subplots(2, 4, figsize=(10,6))
plt.subplots_adjust(wspace=0.3, hspace=0.1)

for i in range(0, 8):
    axes[i // 4, i % 4].imshow(U[:, i].reshape((64,64)))
    axes[i // 4, i % 4].set_title("$\mathbf{{u_{0}}}$".format(i+1), fontsize=16)

```

```
plt.show()
```

SVD_Listing22.py hosted with ♡ by GitHub view raw

The output is:

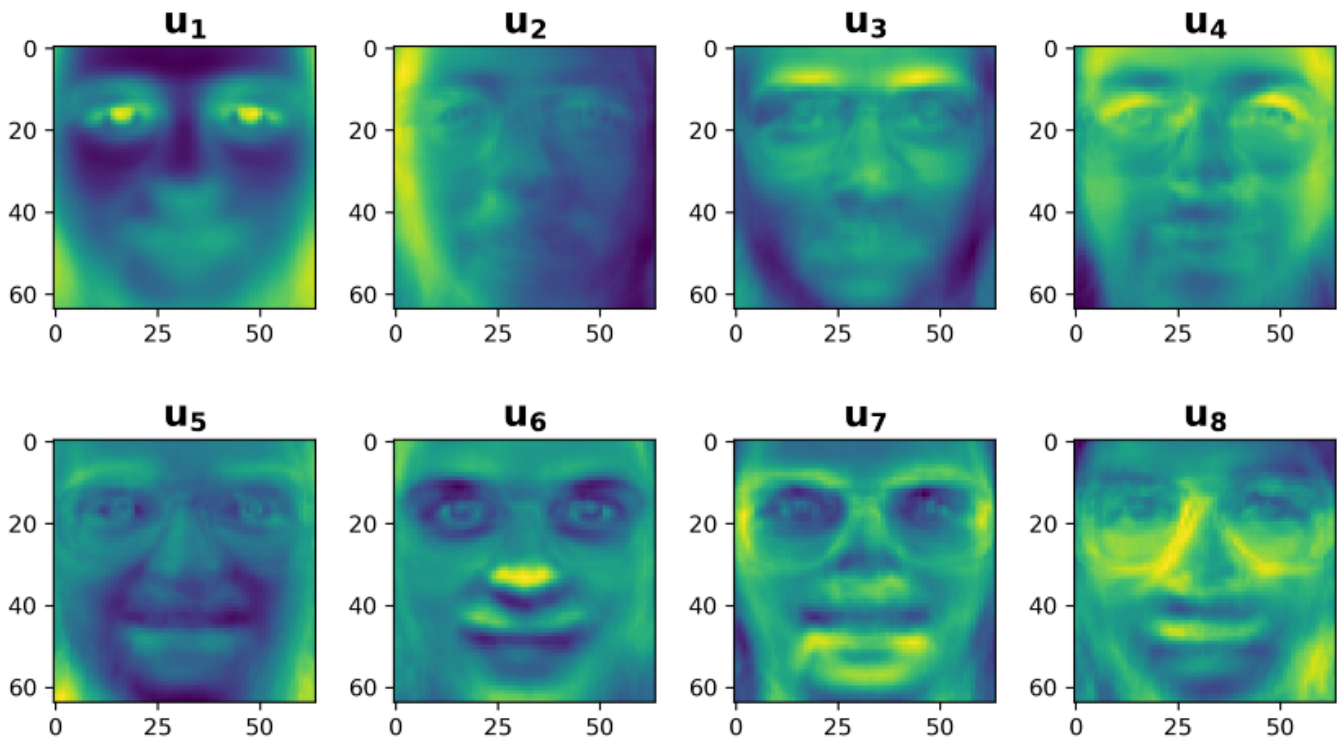


Figure 30

You can check that the array `s` in Listing 22 has 400 elements, so we have 400 non-zero singular values and the rank of the matrix is 400. As a result, we need the first 400 vectors of **U** to reconstruct the matrix completely. We can easily reconstruct one of the images using the basis vectors:

```
# Listing 23
x= np.zeros((400, 1))
x[160, 0] = 1
Sigma = np.zeros((M.shape[0], M.shape[1]))
Sigma[:min(M.shape[0], M.shape[1]), :min(M.shape[0], M.shape[1])] = np.diag(s)

fig, axes = plt.subplots(2, 4, figsize=(14, 8))
fig.suptitle("Reconstructed image using the first k singular values", fontsize=16)
plt.subplots_adjust(wspace=0.3, hspace=0.1)

axes[0, 0].imshow(imgs[160], cmap='gray')
axes[0, 0].set_title("Original image")

k_list = [1, 6, 10, 15, 20, 35, 80]
```

```python
for i in range(1, 8):
    # Reconstruction of the matrix using the first k singular values
    k = k_list[i-1]
    mat_approx = U[:, :k] @ Sigma[:k, :k] @ VT[:k, :] @ x

    axes[i // 4, i % 4].imshow(mat_approx.reshape((64,64)), cmap='gray')
    axes[i // 4, i % 4].set_title("k = {}".format(k))

plt.show()
```

SVD_Listing23.py hosted with ♡ by GitHub view raw

Here we take image #160 and reconstruct it using different numbers of singular values:

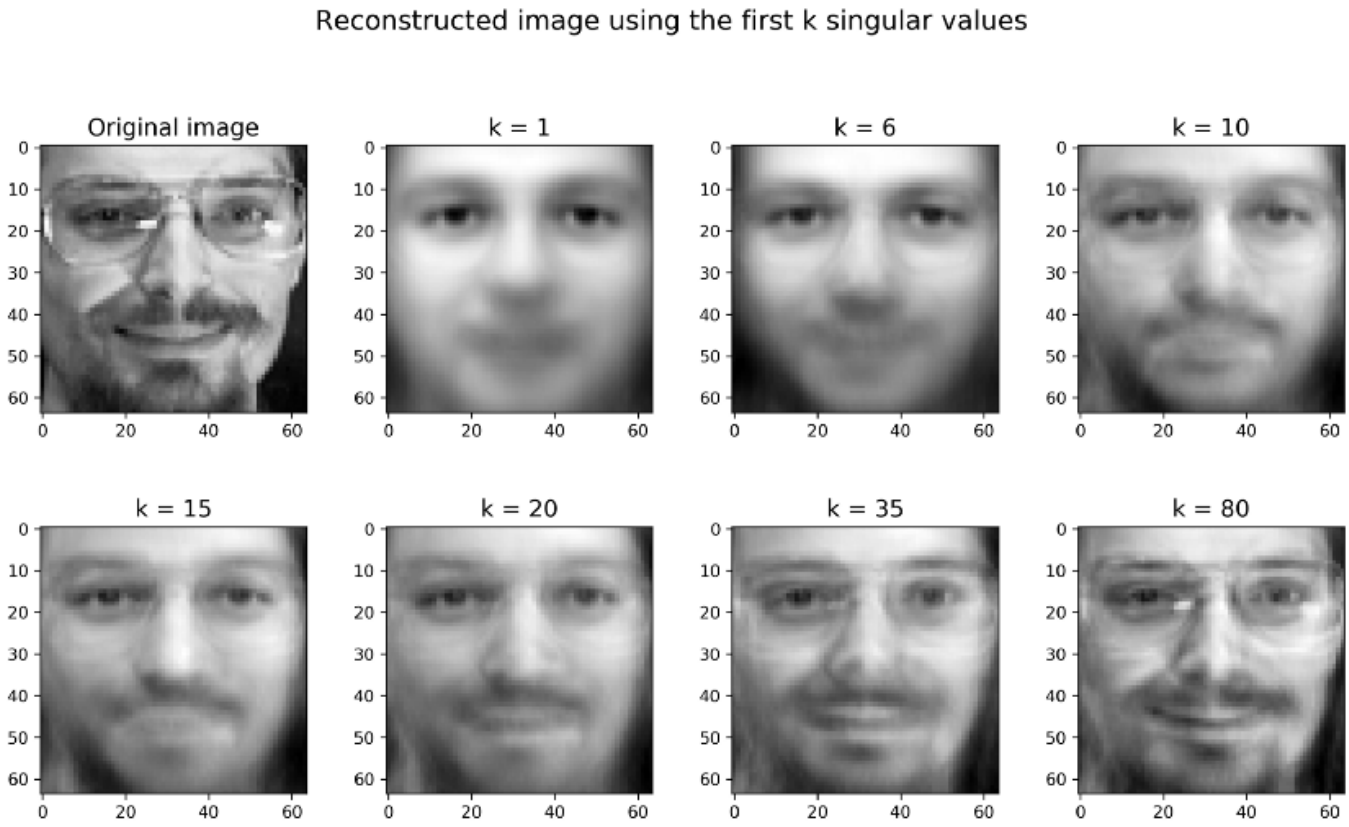


Figure 31

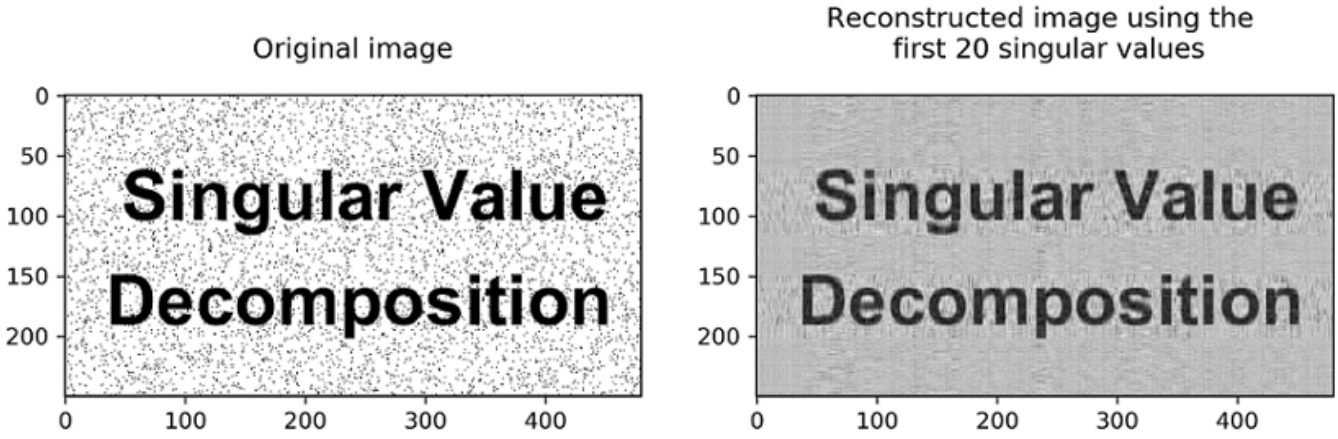
The vectors **ui** are called the *eigenfaces* and can be used for face recognition. As you see in Figure 30, each eigenface captures some information of the image vectors. For example, **u1** is mostly about the eyes, or **u6** captures part of the nose. When reconstructing the image in Figure 31, the first singular value adds the eyes, but the rest of the face is vague. By increasing *k,* nose, eyebrows, beard, and glasses are added to the face. Some people believe that the eyes are the most important feature of your face. It seems that SVD agrees with them since the first eigenface which has the highest singular value captures the eyes.

Reducing noise

SVD can be used to reduce the noise in the images. Listing 24 shows an example:

```python
# Listing 24
# Reading the image
mat = plt.imread("text.png")

# Adding noise
noise = np.random.rand(mat.shape[0], mat.shape[1])
mat[noise > 0.95] = 0

# SVD
U, s, VT = LA.svd(mat)

Sigma = np.zeros((mat.shape[0], mat.shape[1]))
Sigma[:min(mat.shape[0], mat.shape[1]), :min(mat.shape[0], mat.shape[1])] = np.diag(s)

fig, axes = plt.subplots(2, 2, figsize=(10,8))
plt.subplots_adjust(wspace=0.2, hspace=0.1)

axes[0, 0].imshow(mat, cmap='gray')
axes[0, 0].set_title("Original image", y=1.08)

k_list = [20, 55, 200]
for i in range(1, 4):
    k = k_list[i-1]
    mat_rank_k = U[:, :k] @ Sigma[:k, :k] @ VT[:k, :]
    axes[i // 2, i % 2].imshow(mat_rank_k, cmap='gray')
    axes[i // 2, i % 2].set_title("Reconstructed image using the \n first {} singular values".fo

plt.show()
```

SVD_Listing24.py hosted with ♡ by GitHub view raw



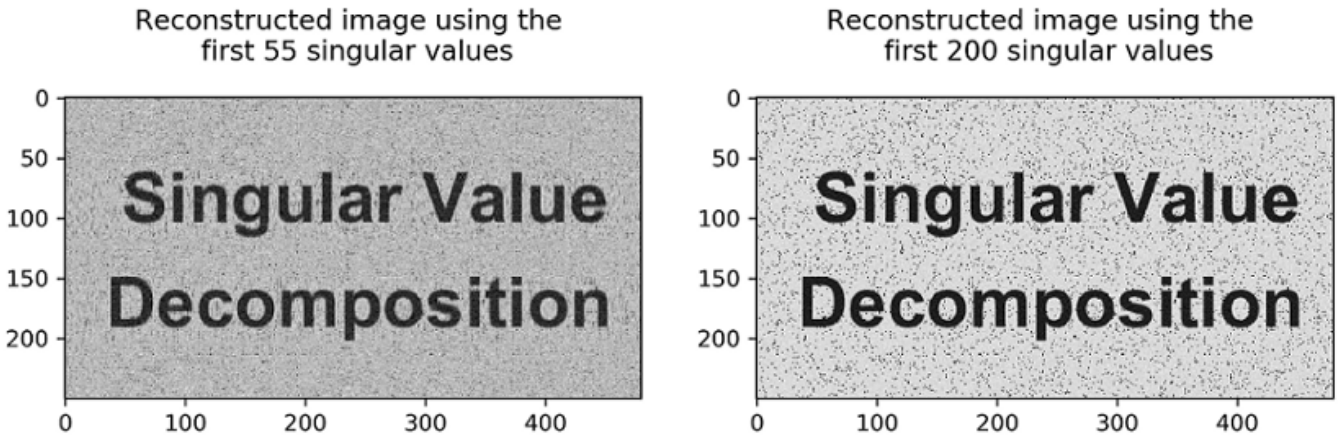


Figure 32

Here we first load the image and add some noise to it. Then we reconstruct the image using the first 20, 55 and 200 singular values. As you see in Figure 32, the amount of noise increases as we increase the rank of the reconstructed matrix. So if we use a lower rank like 20 we can significantly reduce the noise in the image. It is important to understand why it works much better at lower ranks.

Here is a simple example to show how SVD reduces the noise. Imagine that we have 3×15 matrix defined in Listing 25:

```
# Listing 25
# Defining the matrix
mat = np.array([[1.5, 1.65, 1.4, 1.5, 1.4, 0,   0,   0,    0,   0,    0,    0,     0.1,   0,
                [0,   0,    0,   0,   0   , 1.4, 1.5, 1.45, 1.5, 1.4,  1.45, 1.35,  1.55, 1.5,
                [0,   0,    0,   0,   0   , 1.3, 1.4, 1.45, 1.5, 1.5,  1.45, 1.45,  0,   1.5,  1

fig, ax = plt.subplots(1, 1, figsize=(12,3))
pos = ax.imshow(mat)
fig.colorbar(pos, ax=ax)
plt.show()
```

SVD_Listing25.py hosted with ♡ by GitHub view raw

A color map of this matrix is shown below:

Figure 33

The matrix columns can be divided into two categories. In the first 5 columns, only the first element is not zero, and in the last 10 columns, only the first element is zero. We also have a noisy column (column #12) which should belong to the second category, bit its first and last element do not have the right values. We can assume that these two elements contain some noise. Now we decompose this matrix using SVD. The rank of the matrix is 3, and it only has 3 non-zero singular values. Now we reconstruct it using the first 2 and 3 singular values.

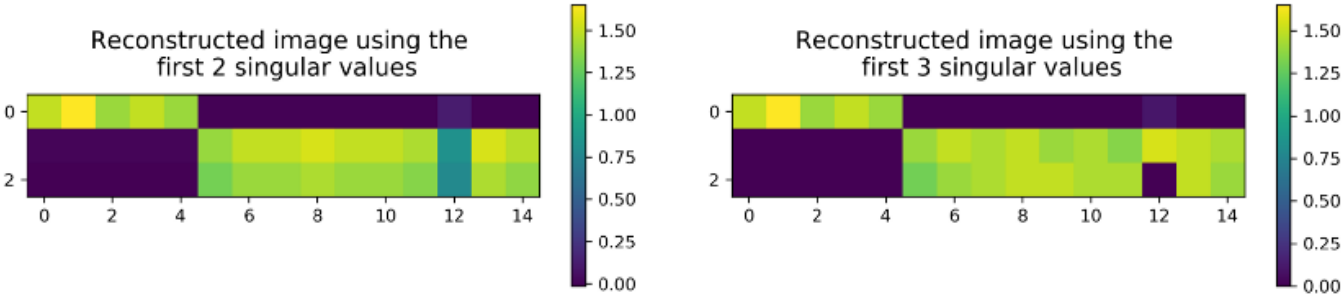
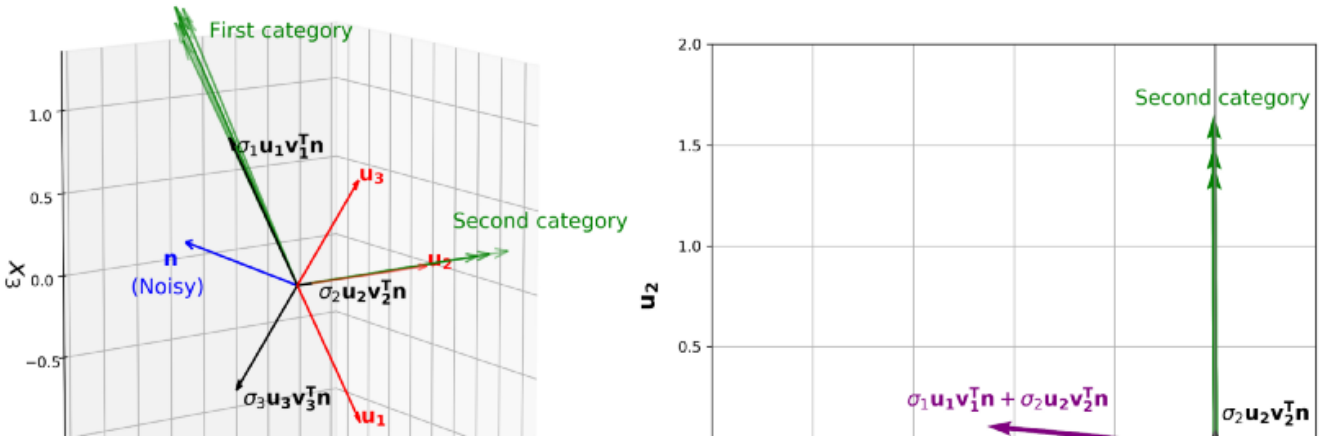


Figure 34

As Figure 34 shows, by using the first 2 singular values column #12 changes and follows the same pattern of the columns in the second category. However, the actual values of its elements are a little lower now. If we use all the 3 singular values, we get back the original noisy column. Figure 35 shows a plot of these columns in 3-d space.



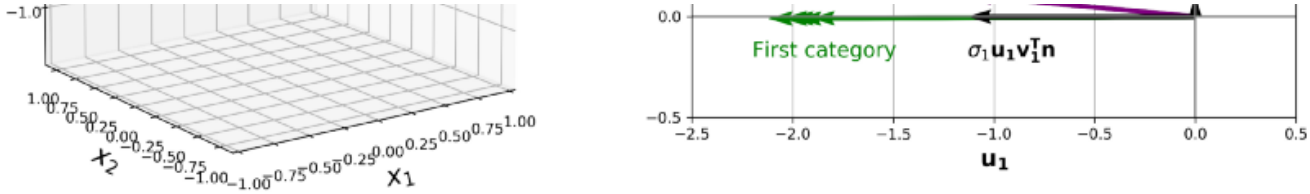


Figure 35

First look at the **ui** vectors generated by SVD. **u1** shows the average direction of the column vectors in the first category. Of course, it has the opposite direction, but it does not matter (Remember that if **vi** is an eigenvector for an eigenvalue, then (-1)**vi** is also an eigenvector for the same eigenvalue, and since **ui**=**Avi**/σi, then its sign depends on **vi**). What is important is the stretching direction not the sign of the vector. Similarly, **u2** shows the average direction for the second category.

The noisy column is shown by the vector **n**. It is not along **u1** and **u2**. Now if we use **ui** as a basis, we can decompose **n** and find its orthogonal projection onto **ui**. As you see it has a component along **u3** (in the opposite direction) which is the noise direction. This direction represents the noise present in the third element of **n**. It has the lowest singular value which means it is not considered an important feature by SVD. When we reconstruct **n** using the first two singular values, we ignore this direction and the noise present in the third element is eliminated. Now we only have the vector projections along **u1** and **u2**. But the scalar projection along **u1** has a much higher value. That is because vector **n** is more similar to the first category.

So the projection of **n** in the **u1-u2** plane is almost along **u1**, and the reconstruction of **n** using the first two singular values gives a vector which is more similar to the first category. It is important to note that the noise in the first element which is represented by **u2** is not eliminated. In addition, though the direction of the reconstructed **n** is almost correct, its magnitude is smaller compared to the vectors in the first category. In fact, in the reconstructed vector, the second element (which did not contain noise) has now a lower value compared to the original vector (Figure 36).

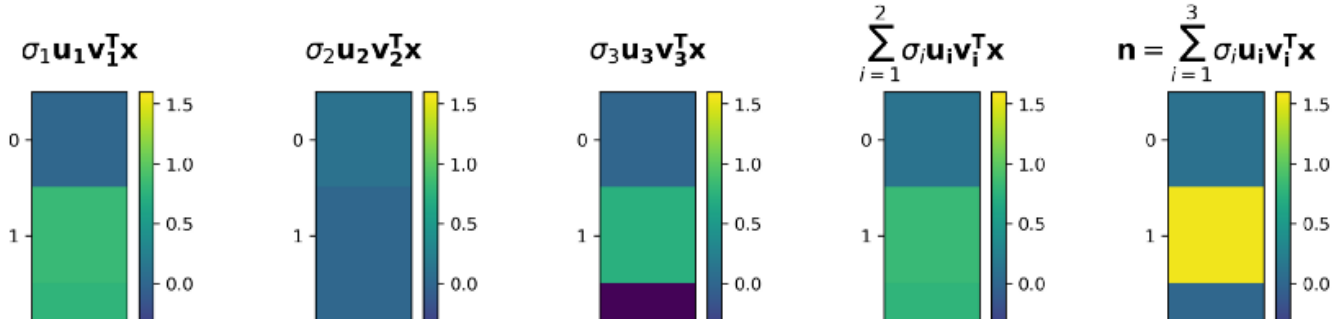


Figure 36

So SVD assigns most of the noise (but not all of that) to the vectors represented by the lower singular values. If we reconstruct a low-rank matrix (ignoring the lower singular values), the noise will be reduced, however, the correct part of the matrix changes too. The result is a matrix that is only an approximation of the noiseless matrix that we are looking for. This can be seen in Figure 32. The image background is white and the noisy pixels are black. When we reconstruct the low-rank image, the background is much more uniform but it is gray now. In fact, what we get is a less noisy approximation of the white background that we expect to have if there is no noise in the image.

I hope that you enjoyed reading this article. Please let me know if you have any questions or suggestions. All the Code Listings in this article are available for download as a Jupyter notebook from GitHub at: https://github.com/reza-bagheri/SVD_article

Further reading:

Eigendecomposition and SVD can be also used for the Principal Component Analysis (PCA). PCA is very useful for dimensionality reduction. To learn more about the application of eigendecomposition and SVD in PCA, you can read these articles:

https://reza-bagheri79.medium.com/understanding-principal-component-analysis-and-its-application-in-data-science-part-1-54481cd0ad01

https://reza-bagheri79.medium.com/understanding-principal-component-analysis-and-its-application-in-data-science-part-2-e16b1b225620