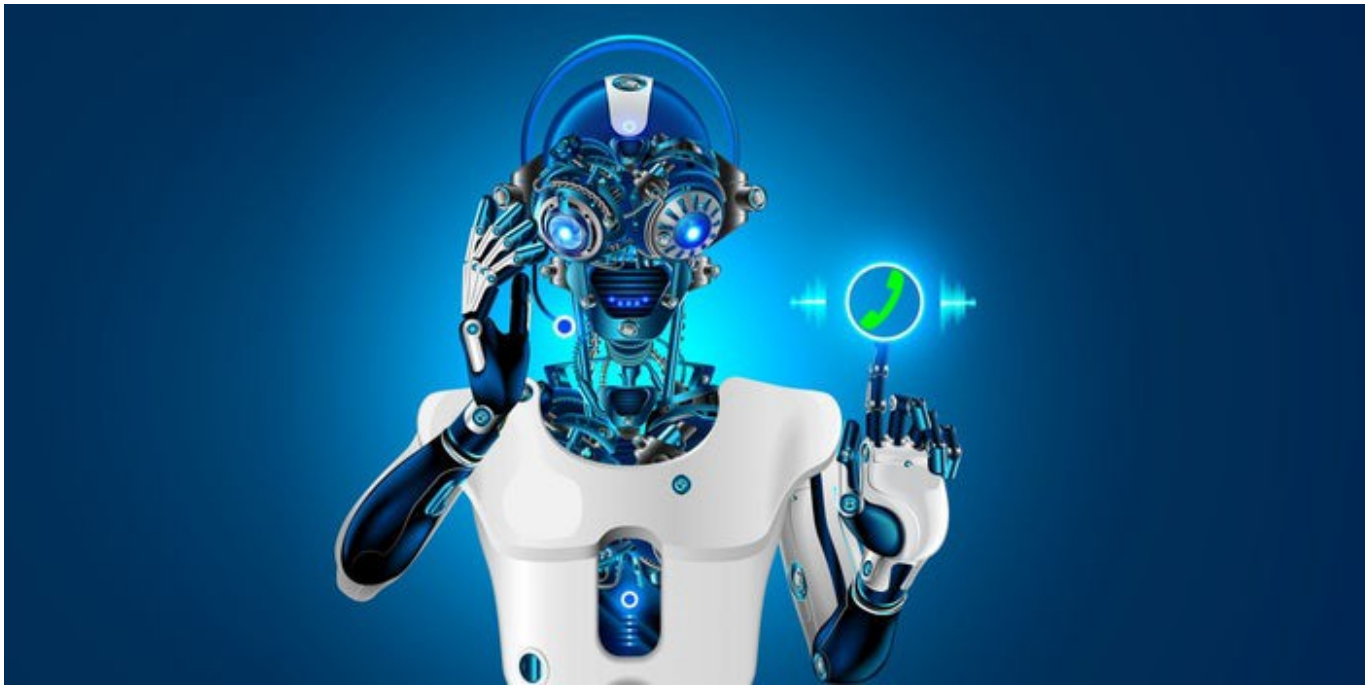


Singular Value Decomposition Example In Python

Cory Maklin Aug 5, 2019 · 7 min read ★

Singular Value Decomposition, or SVD, has a wide array of applications. These include dimensionality reduction, image compression, and denoising data. In essence, SVD states that a matrix can be represented as the product of three other matrices. In mathematical terms, SVD can be written as follows:

$$\mathbf{A}_{nxp} = \mathbf{U}_{nxn}\,\mathbf{S}_{nxp}\,\mathbf{V}^{\mathbf{T}}{}_{pxp}$$

where $\boldsymbol{n}$ is the number of rows (i.e. samples) and $\boldsymbol{p}$ represents the number of dimensions.

Suppose we had a matrix $\boldsymbol{A}$.

$$A = \begin{bmatrix} 2 & 4 \\ 1 & 3 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

In order to determine the associated $\boldsymbol{U}$ matrix, we must first find the eigenvectors of the matrix $\boldsymbol{A}$ multiplied by the transpose of the matrix $\boldsymbol{A}$.

$$AA^T = \begin{bmatrix} 2 & 4 \\ 1 & 3 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 2 & 4 & 0 & 0 \\ 1 & 3 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 20 & 14 & 0 & 0 \\ 14 & 10 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} = W$$

since

$$\begin{bmatrix} 2 \times 2 + 4 \times 4 & 2 \times 1 + 4 \times 3 & 0 & 0 \\ 1 \times 2 + 3 \times 4 & 1 \times 1 + 3 \times 3 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Recall how the definition for eigenvectors and eigenvalues is:

$$Wx = \lambda x$$

The latter can also be expressed as:

$$Wx = (W - \lambda I)x$$

Thus, going back to our example:

$$\begin{bmatrix} 20-\lambda & 14 & 0 & 0 \\ 14 & 10-\lambda & 0 & 0 \\ 0 & 0 & -\lambda & 0 \\ 0 & 0 & 0 & -\lambda \end{bmatrix} \mathbf{x} = (W - \lambda I)\mathbf{x} = 0$$

We end up with a fourth degree polynomial.

$$= \lambda^4 - 30 \times \lambda^3 + 4 \times \lambda^2 = \lambda \times \lambda \times (\lambda^2 - 30 \times \lambda + 4) = \lambda \times \lambda \times (\lambda + (\sqrt{221} - 15)) \times (\lambda - (\sqrt{221} + 15))$$

1. $\lambda_1 = 0$

2. $\lambda_2 = -\sqrt{221} + 15$
3. $\lambda_3 = \sqrt{221} + 15$

$$\sqrt{221} + 15 = 29.833$$

$$-\sqrt{221} + 15 = 0.117$$

After solving, we replace lambda with one of the eigenvalues.

$$\begin{bmatrix} 20 - 0.117 & 14 & 0 & 0 \\ 14 & 10 - 0.117 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 19.833 & 14 & 0 & 0 \\ 14 & 9.833 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

After multiplying the matrix by the vector $\boldsymbol{x}$, we obtain the following:

$$\begin{aligned} &19.883\ x1 + 14\ x2 = 0 \\ &14\ x1 + 9.883\ x2 = 0 \\ &x3 = 0 \\ &x4 = 0 \end{aligned}$$

In solving the equations, we get:

$$\begin{aligned} &x1 = 0.82 \\ &x2 = -0.58 \\ &x3 = 0 \\ &x4 = 0 \end{aligned}$$

We plug the eigenvectors into the columns of $\boldsymbol{U}$.

$$U = \begin{bmatrix} 0.82 & -0.58 & 0 & 0 \\ 0.58 & 0.82 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Then, we repeat the same process for the transpose of the matrix $\boldsymbol{A}$ multiplied by the matrix $\boldsymbol{A}$.

$$A^T . A = \begin{bmatrix} 2 & 4 & 0 & 0 \\ 1 & 3 & 0 & 0 \end{bmatrix} \begin{bmatrix} 2 & 4 \\ 1 & 3 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

After solving, we obtain the expression for $\boldsymbol{V}$.

$$V = \begin{bmatrix} 0.40 & -0.91 \\ 0.91 & 0.40 \end{bmatrix}$$

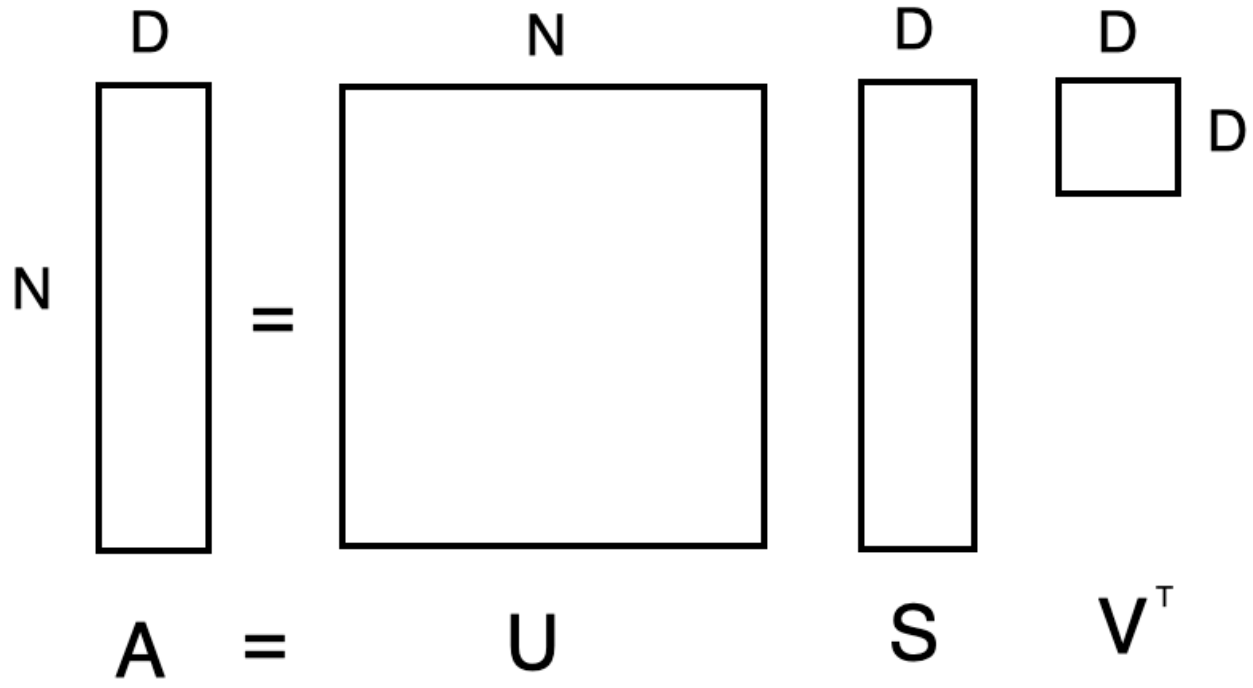
Finally, $\boldsymbol{S}$ is a diagonal matrix whose values are the square root of the eigenvalues from either

$$AA^T \text{ or } A^TA.$$

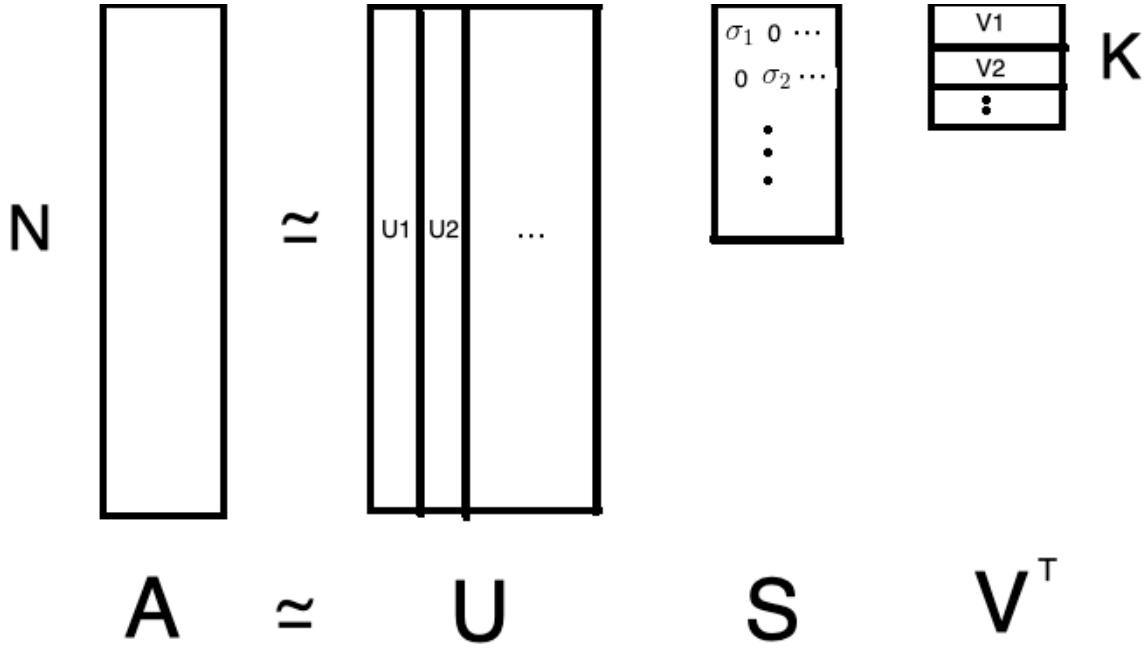
$$\begin{bmatrix} \sqrt{29.883} & 0 \\ 0 & \sqrt{0.117} \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

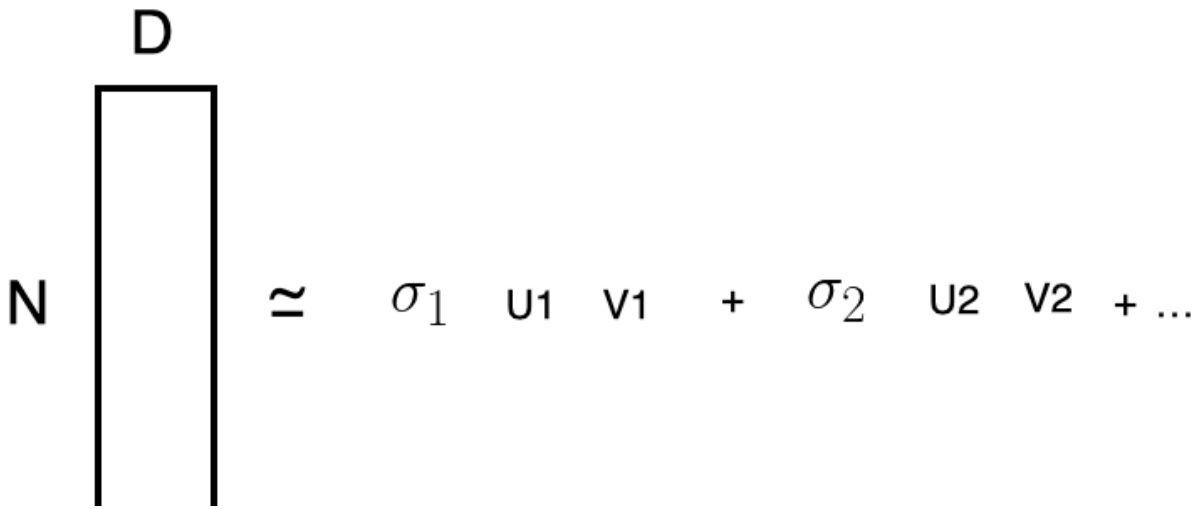
$$S = \begin{bmatrix} 5.47 & 0 \\ 0 & 0.37 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Here's where things get interesting. We know that the product of all three matrices is equivalent to the matrix on the lefthand side.



We can exclude features and still maintain an approximation of the original matrix. Say that the matrix A is dataset of columns and rows or the pixels that make up an image, we can in theory train models using the newly formed matrix and achieve comparable if not better (due to the curse of dimensionality) accuracy.


N
≃
U1
U2
...
σ_1 0 ⋯
0 σ_2 ⋯
V1
V2
K
A ≃ U S V^T


D
N
≃
σ_1 U1 V1 + σ_2 U2 V2 + ...

Code

Let's take a look at how we could go about applying Singular Value Decomposition in Python. To begin, import the following libraries.

```
import numpy as np
from sklearn.datasets import load_digits
from matplotlib import pyplot as plt
from sklearn.decomposition import TruncatedSVD
float_formatter = lambda x: "%.2f" % x
np.set_printoptions(formatter={'float_kind':float_formatter})
from sklearn.ensemble import RandomForestClassifier
```

In the proceeding tutorial, we'll attempt to classify handwritten digits. Fortunately, the `scikit-learn` library provides a wrapper function for importing the dataset into our program.

```
X, y = load_digits(return_X_y=True)
```

The dataset contains 1797 8x8 images. If you specify `return_X_y=True` the function will return the pixels as a one dimensional array.

```
X.shape
```

```
(1797, 64)
```

y contains the labels for every digit.

```
y
```

```
array([0, 1, 2, ..., 8, 9, 8])
```

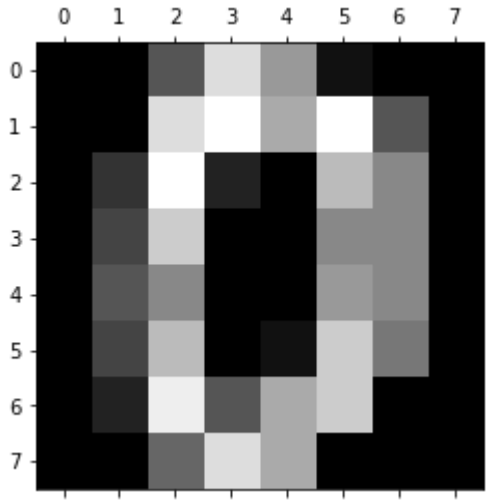
Let's take a look at the first digit. As we can see, it's simply an array of length 64 containing the pixel intensities.

```
image = X[0]
```

```
array([0.00, 0.00, 5.00, 13.00, 9.00, 1.00, 0.00, 0.00, 0.00, 0.00, 13.00,
       15.00, 10.00, 15.00, 5.00, 0.00, 0.00, 3.00, 15.00, 2.00, 0.00,
       11.00, 8.00, 0.00, 0.00, 4.00, 12.00, 0.00, 0.00, 8.00, 8.00, 0.00,
       0.00, 5.00, 8.00, 0.00, 0.00, 9.00, 8.00, 0.00, 0.00, 4.00, 11.00,
       0.00, 1.00, 12.00, 7.00, 0.00, 0.00, 2.00, 14.00, 5.00, 10.00,
       12.00, 0.00, 0.00, 0.00, 0.00, 6.00, 13.00, 10.00, 0.00, 0.00,
       0.00])
```

If we want to view the image using `matplotlib`, we must first reshape the array.

```
image = image.reshape((8, 8))

plt.matshow(image, cmap = 'gray')
```

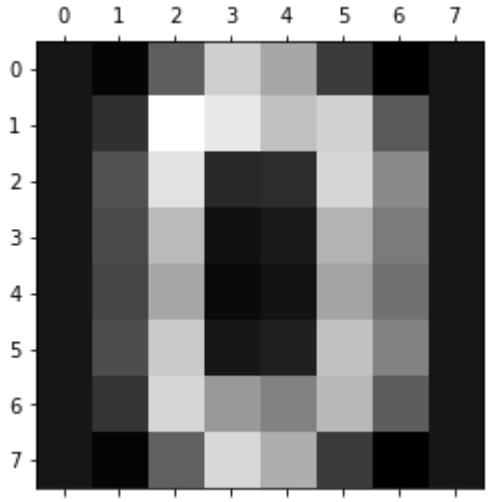


Next, we'll use Singular Value Decomposition to see whether we are able to reconstruct the image using only 2 features for each row. The **s** matrix returned by the function must be converted into a diagonal matrix using the `diag` method. By default, `diag` will create a matrix that is *n x n*, relative to the original matrix. This causes a problem as the size of the matrices no longer follow the rule of matrix multiplication where the number of columns in a matrix must match the number of rows in the other matrix. Therefore, we create a new *m x n* matrix and populate the first *n x n* part of it with the diagonal matrix.

```
U, s, V = np.linalg.svd(image)

S = np.zeros((image.shape[0], image.shape[1]))

S[:image.shape[0], :image.shape[0]] = np.diag(s)

n_component = 2

S = S[:, :n_component]
VT = VT[:n_component, :]

A = U.dot(Sigma.dot(VT))

print(A)
```

```
[[0.00 -1.10 4.74 12.16 9.52 2.44 -1.33 0.00]
 [0.00 1.81 15.31 13.79 11.23 12.27 4.48 0.00]
 [0.00 3.86 13.36 1.19 1.49 12.59 7.65 0.00]
 [0.00 3.39 10.76 -0.35 0.20 10.36 6.61 0.00]
 [0.00 3.11 9.52 -0.85 -0.24 9.26 6.04 0.00]
 [0.00 3.61 11.79 0.12 0.60 11.27 7.07 0.00]
 [0.00 2.08 12.63 8.58 7.13 10.60 4.66 0.00]
 [0.00 -1.18 4.88 12.71 9.96 2.47 -1.44 0.00]]
```

```
plt.matshow(A, cmap = 'gray')
```



We can get the reduced feature space by taking the dot product of the **U** and **S** matrices.

```
U.dot(S)
```

```
array([[10.97, -12.23],
       [26.04, -6.82],
       [18.73, 7.95],
       [14.60, 8.08],
       [12.72, 7.84],
       [16.19, 8.21],
       [20.45, -2.05],
       [11.36, -12.85]])
```

Original vs Reduced Feature Space

Let's compare the accuracy of a Random Forest model when it's trained using the original handwritten digits and when it's trained using the reduced feature space obtained from Singular Value Decomposition.

We can gauge the accuracy of the model by taking a look at the Out-Of-Bag score. If you're unfamiliar with the concept of OOB, I encourage you checkout *this* post on Random Forest.

```
rf_original = RandomForestClassifier(oob_score=True)

rf_original.fit(X, y)

rf_original.oob_score_
```

```
0.8603227601558152
```

Next, we create and fit an instance of the `TruncatedSVD` class with 2 components. It's worth mentioning that unlike the previous example, we're using 2/64 features.

```
svd = TruncatedSVD(n_components=2)

X_reduced = svd.fit_transform(X)
```

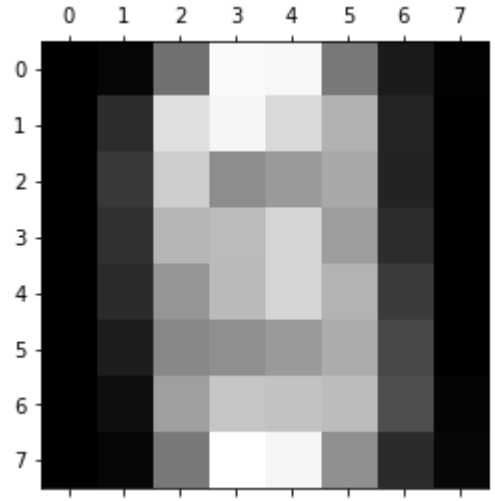
Every image (i.e. row) in the reduced dataset contains 2 features.

```
X_reduced[0]
```

```
array([45.86, -1.19])
```

Taking a look at the image, it's difficult to distinguish what digit the image consists of, it could very well be a 5 and not a 0.

```
image_reduced = svd.inverse_transform(X_reduced[0].reshape(1,-1))

image_reduced = image_reduced.reshape((8,8))

plt.matshow(image_reduced, cmap = 'gray')
```



After training a Random Forest Classifier on the reduced dataset, we obtain a meager accuracy of 36.7%

```
rf_reduced = RandomForestClassifier(oob_score=True)

rf_reduced.fit(X_reduced, y)

rf_reduced.oob_score_
```

```
0.36672231496939345
```

We can get the total variance explained by taking the sum of the `explained_variance_ratio_` property. We generally want to aim for 80 to 90 percent.

```
svd.explained_variance_ratio_.sum()
```

```
0.17760900862040058
```

Let's try again, only, this time, we use 16 components. We check to see the amount of information contained in the 16 features.

```
svd = TruncatedSVD(n_components=16)

X_reduced = svd.fit_transform(X)

svd.explained_variance_ratio_.sum()
```

```
0.8479578256047824
```

We obtain an accuracy comparable to the model trained using the original images and we used 16/64=0.25 the amount of data.

```
rf_reduced = RandomForestClassifier(oob_score=True)

rf_reduced.fit(X_reduced, y)

rf_reduced.oob_score_
```

```
0.8647746243739566
```

Sign up for The Variable