Singular Value Decomposition (SVD), Demystified

A comprehensive guide to SVD with Python examples

Dr. Roi Yehoshua
Nov 8, 202320 min read

Singular value decomposition (SVD) is a powerful matrix factorization technique that decomposes a matrix into three other matrices, revealing important structural aspects of the original matrix. It is used in a wide range of applications, including signal processing, image compression, and dimensionality reduction in machine learning. This article provides a step-by-step guide on how to compute the SVD of a matrix, including a detailed numerical example. It then demonstrates how to use SVD for dimensionality reduction using examples in Python. Finally, the article discusses various applications of SVD and some of its limitations.

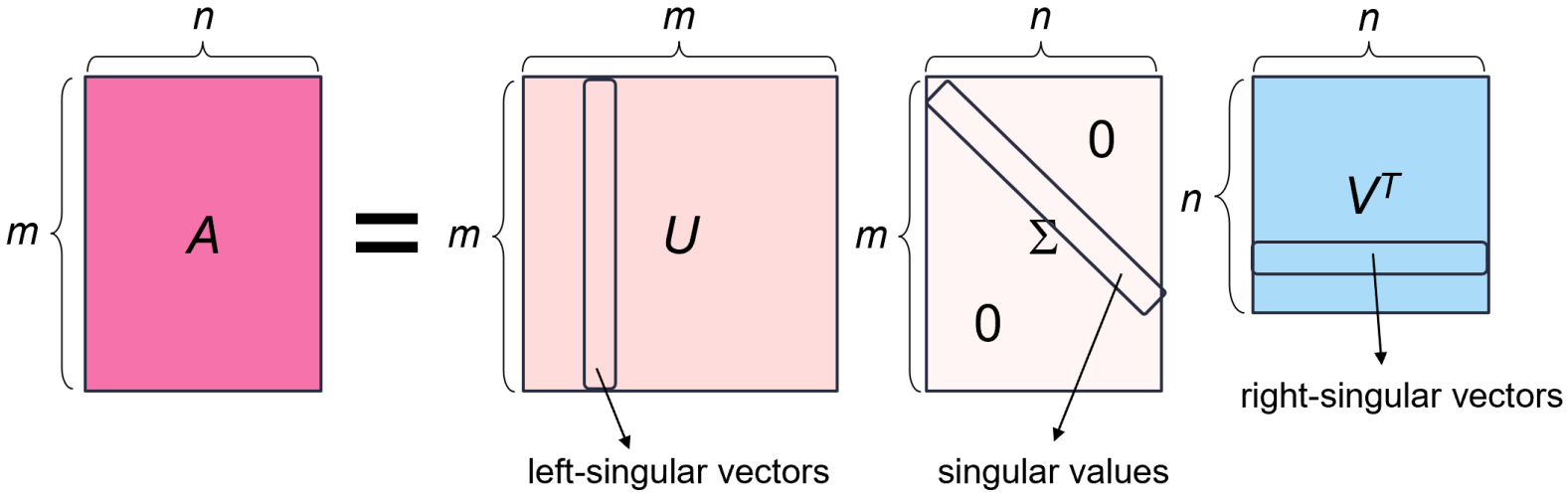
The article assumes the reader has basic knowledge of linear algebra. More specifically, the reader should be familiar with concepts such as vector and matrix norms, rank of a matrix, eigen-decomposition (eigenvectors and eigenvalues), orthonormal vectors, and linear projections.

Image by Peggy und Marco Lachmann-Anke from Pixabay

Mathematical Definition

The singular value decomposition of an $m \times n$ real matrix A is a factorization of the form $A = U\Sigma V^t$, where:

- U is an $m \times m$ **orthogonal matrix** (i.e., its columns and rows are orthonormal vectors). The columns of U are called the **left-singular vectors** of A.
- Σ is an $m \times n$ rectangular diagonal matrix with non-negative real numbers on the diagonal. The diagonal entries $\sigma_i = \Sigma_{ii}$ are known as the **singular values** of A and are typically arranged in descending order, i.e., $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_n \geq 0$. The number of the non-zero singular values is equal to the rank of A.
- V is an $n \times n$ orthogonal matrix. The columns of V are called the **right-singular vectors** of A.



SVD of a matrix

Every matrix has a singular value decomposition __ (a proof of this statement can be found here). This is unlike eigenvalue decomposition, for example, which can be applied only to squared diagonalizable matrices.

Computing the SVD

The singular value decomposition of a matrix A can be computed using the following observations:

1. The left-singular vectors of A are a set of orthonormal eigenvectors of AA^t.
2. The right-singular vectors of A are a set of orthonormal eigenvectors of A^tA.
3. The non-zero singular values of A are the square roots of the non-zero eigenvalues of both A^tA and AA^t.
4. If _U_ΣV^t is the SVD of A, then __ for each singular value σ_i,

$$\mathbf{u}_i = \frac{1}{\sigma_i} A\mathbf{v}_i$$

where $\mathbf{u}_i$ is the i-th column of U and $\mathbf{v}_i$ is the i-th column of V.

Proof:

1. We will first show that the left-singular vectors of A are a set of orthonormal eigenvectors of AA^t.

Let A = _U_ΣV^t be the SVD of A, and let's examine __ the product _AA_^{t}:

$$\begin{aligned} AA^T &= (U\Sigma V^T)(U\Sigma V^T)^T && \text{(definition of SVD)} \\ &= (U\Sigma V^T)[(V^T)^T\Sigma^T U^T] && ((AB)^T = B^T A^T) \\ &= U\Sigma V^T V\Sigma^T U^T && ((V^T)^T = V) \\ &= U\Sigma\Sigma^T U^T && (V^T V = I, \text{ since } V \text{ is orthonormal}) \end{aligned}$$

Since Σ is a diagonal matrix with singular values σ_i on its diagonal, $\Sigma\Sigma^t$ is also a diagonal matrix where each diagonal element is _σ_i_^{2}. Let's denote this matrix by Σ^2. This gives us:

$$AA^T = U\Sigma^2 U^T$$

Since U is orthonormal, $U^tU = I$, and by right multiplying both sides of the equation by U we get:

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

Let's now consider a single column of U, denoted by $\mathbf{u}_i$. Since $AB_i = [AB]_i$ (i.e., matrix A multiplied by column i of matrix B is equal to column i of their product AB), we can write:

$$AA^T\mathbf{u}_i = [U\Sigma^2]_i = \sigma_i^2\mathbf{u}_i$$

Therefore, $\mathbf{u}_i$ is an eigenvector of AA^t corresponding to the eigenvalue λ_i = _σ_i_^{2}. In other words, the columns of U are eigenvectors of AA^t. Because the columns of U are orthonormal, the left-singular vectors of A (the columns of U) are a set of orthonormal vectors of AA^t.

2. In a similar fashion, we can show that the right-singular vectors of A are a set of orthonormal eigenvectors of A^tA.
3. We first notice that AA^t is symmetric and positive semi-definite. Therefore all its eigenvalues are real and non-negative, and it has a full set of orthonormal eigenvectors. Let $\{\mathbf{u}_1, \ldots, \mathbf{u}_n\}$ be the orthonormal eigenvectors of AA^t corresponding to eigenvalues $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n \geq 0$. For any eigenvector $\mathbf{u}_i$ of AA^t corresponding to an eigenvalue λ_i we have:

$$AA^T\mathbf{u}_i = \lambda_i\mathbf{u}_i = \sigma_i^2\mathbf{u}_i$$

Thus, the singular values of A are the square roots of the eigenvalues of AA^t.
Similarly, we can show that the singular values of A are also the square roots of the eigenvalues of A^tA.

4. Left as an exercise to the reader.

Based on the above observations, we can compute the SVD of an $m \times n$ matrix A using the following steps:

1. Construct the matrix A^tA.
2. Compute the eigenvalues and eigenvectors of A^tA. The eigenvalues will be the squares of the singular values of A, and the eigenvectors will form the columns of the matrix V in the SVD.
3. Arrange the singular values of A in descending order. Create an $m \times n$ diagonal matrix Σ with the singular values on the diagonal, padding with zeros if necessary so that the matrix has the same dimensions as A.
4. Normalize the eigenvectors of A^tA to have unit length, and place them as columns of matrix V.
5. For each singular value σ_i, calculate the corresponding left-singular vector $\mathbf{u}_i$ as

$$\mathbf{u}_i = \frac{1}{\sigma_i}A\mathbf{v}_i$$

where $\mathbf{v}_i$ is the i-th column of V. Place these vectors as columns in the matrix U.
If $n < m$ or A is rank-deficient (i.e., $\text{rank}(A) < \min(m, n)$), then there would not be enough non-zero singular values to determine the columns of U. In this case, we need to complete U to an orthogonal matrix by finding additional orthonormal vectors that span the null space (kernel) of A^t.
The **null space** of A^t, denoted $N(A^t)$, is the __ set of vectors **x** such that _A_^{t}**x =** 0, which are also the ** **eigenvectors o_f AA_^{t} corresponding to eigenvalue 0 (sinc_e A_A^{t}x =** $0 \cdot$ **x). To find an orthonormal basis fo*r* _N(_A^t), we first solve the homogenous linear syste*m* A^{t}x** =** 0 to find a basis fo*r* _N(_A^t), then use the Gram-Schmidt process on this set of basis vectors to make them orthogonal, and finally we normalize them into unit vectors.
Another way to find the left-singular vectors of A (the columns of U) is to compute the eigenvectors of AA^t, but this process is usually more time consuming than using the relationship between the left and right singular vectors (observation 4) and computing the null space of A^t (if necessary).
Note that it is also possible to start the SVD computation by finding the left-singular vectors (i.e., the eigenvectors of AA^t), and then use the following relationship to find the right singular vectors:

$$\mathbf{v}_i = \frac{1}{\sigma_i}A^T\mathbf{u}_i$$

The choice of using either AA^t or A^tA depends on which matrix is smaller.

Numerical Example
For example, let's compute the SVD of the following matrix:

$$A = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 2 & 2 \end{pmatrix}$$

Let _U_ΣV^t be the SVD of A. The dimensions of A are 3×2. Therefore, the size of U is 3×3, the size of Σ is 3×2, and the size of V is 2×2.
Since the size of A^tA (2×2) is smaller than the size of AA^t (3×3), it makes sense to start with the right-singular vectors of A.
We first compute A^tA:

$$A^T A = \begin{pmatrix} 1 & 0 & 2 \\ 0 & 1 & 2 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 2 & 2 \end{pmatrix} = \begin{pmatrix} 5 & 4 \\ 4 & 5 \end{pmatrix}$$

We now find the eigenvalues and eigenvectors of A^tA. The characteristic polynomial of the matrix is:

$$A^T A - \lambda I = \begin{pmatrix} 5-\lambda & 4 \\ 4 & 5-\lambda \end{pmatrix}$$

$$|A^T A - \lambda I| = (5-\lambda)(5-\lambda) - 16 = \lambda^2 - 10\lambda + 9$$

The roots of this polynomial are:

$$\lambda^2 - 10\lambda + 9 = 0$$

$$\lambda_{1,2} = \frac{10 \pm \sqrt{10^2 - 36}}{2} = \frac{10 \pm 8}{2}$$

$$\lambda_1 = 9, \lambda_2 = 1$$

The eigenvalues of A^tA in descending order are $\lambda_1 = 9$ and $\lambda_2 = 1$. Therefore, the singular values of A are $\sigma_1 = 3$ and $\sigma_2 = 1$, and the matrix Σ is:

$$\Sigma = \begin{pmatrix} 3 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$$

We now find the right singular vectors (the columns of V) by finding an orthonormal set of eigenvectors of A^tA. The eigenvectors corresponding to $\lambda_1 = 9$ are:

$$A^T A \mathbf{v}_1 = \begin{pmatrix} 5 & 4 \\ 4 & 5 \end{pmatrix} \begin{pmatrix} v_{11} \\ v_{12} \end{pmatrix} = \begin{pmatrix} 9v_{11} \\ 9v_{12} \end{pmatrix}$$

$$5v_{11} + 4v_{12} = 9v_{11} \Rightarrow v_{12} = v_{11}$$

$$4v_{11} + 5v_{12} = 9v_{12} \Rightarrow v_{12} = v_{11}$$

Therefore, the eigenvectors are of the form $\mathbf{v}_1 = (t, t)$. For a unit-length eigenvector we need:

$$2t^2 = 1 \Rightarrow t = \frac{1}{\sqrt{2}}$$

Thus, the unit eigenvector corresponding to $\lambda_1 = 9$ is:

$$\mathbf{v}_1 = \begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{pmatrix}$$

Similarly, the eigenvectors corresponding to $\lambda_2 = 1$ are:

$$A^T A \mathbf{v}_2 = \begin{pmatrix} 5 & 4 \\ 4 & 5 \end{pmatrix} \begin{pmatrix} v_{21} \\ v_{22} \end{pmatrix} = \begin{pmatrix} v_{21} \\ v_{22} \end{pmatrix}$$

$$5v_{21} + 4v_{22} = v_{21} \Rightarrow v_{22} = -v_{21}$$
$$4v_{21} + 5v_{22} = v_{22} \Rightarrow v_{22} = -v_{21}$$

Therefore, the eigenvectors are of the form $\mathbf{v}_2 = (t, t)$. For a unit-length eigenvector we need

$$2t^2 = 1 \Rightarrow t = \frac{1}{\sqrt{2}}$$

Thus, the unit eigenvector corresponding to $\lambda_2 = 1$ is:

$$\mathbf{v}_2 = \begin{pmatrix} 1/\sqrt{2} \\ -1/\sqrt{2} \end{pmatrix}$$

We can now write the matrix V, whose columns are the vectors $\mathbf{v}_1$ and $\mathbf{v}_2$:

$$V = \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{pmatrix}$$

Lastly, we find the left-singular vectors of A. From observation 4, it follows that:

$$\mathbf{u}_1 = \frac{1}{\sigma_1} A \mathbf{v}_1 = \frac{1}{3} \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 2 & 2 \end{pmatrix} \begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{pmatrix} = \frac{1}{3} \begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \\ 4/\sqrt{2} \end{pmatrix} = \begin{pmatrix} 1/\sqrt{18} \\ 1/\sqrt{18} \\ 4/\sqrt{18} \end{pmatrix}$$

$$\mathbf{u}_2 = \frac{1}{\sigma_2} A \mathbf{v}_2 = 1 \cdot \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 2 & 2 \end{pmatrix} \begin{pmatrix} 1/\sqrt{2} \\ -1/\sqrt{2} \end{pmatrix} = \begin{pmatrix} 1/\sqrt{2} \\ -1/\sqrt{2} \\ 0 \end{pmatrix}$$

Since there is only one remaining column vector of U, instead of computing the kernel of A^t, we can simply find a unit vector that is perpendicular to both $\mathbf{u}_1$ and $\mathbf{u}_2$.
Let $\mathbf{u}_3 = (a, b, c)$. To be perpendicular to $\mathbf{u}_2$ we need $a = b$. Then the condition $\mathbf{u}_3^t \mathbf{u}_1 = 0$ becomes

$$\frac{2a}{\sqrt{18}} + \frac{4c}{\sqrt{18}} = 0 \Rightarrow a = -2c$$

Therefore,

$$\mathbf{u}_3 = \begin{pmatrix} a \\ a \\ -a/2 \end{pmatrix}$$

For the vector to be unit-length, we need

$$\frac{9}{4}a^2 = 1 \Rightarrow a = \frac{2}{3}$$

Thus,

$$\mathbf{u}_3 = \begin{pmatrix} 2/3 \\ 2/3 \\ -1/3 \end{pmatrix}$$

And the matrix U is:

$$U = \begin{pmatrix} 1/\sqrt{18} & 1/\sqrt{2} & 2/3 \\ 1/\sqrt{18} & -1/\sqrt{2} & 2/3 \\ 4/\sqrt{18} & 0 & -1/3 \end{pmatrix}$$

The final SVD of A in its full glory is:

$$A = U\Sigma V^T = \begin{pmatrix} 1/\sqrt{18} & 1/\sqrt{2} & 2/3 \\ 1/\sqrt{18} & -1/\sqrt{2} & 2/3 \\ 4/\sqrt{18} & 0 & -1/3 \end{pmatrix} \begin{pmatrix} 3 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{pmatrix}$$

Computing SVD with NumPy

To compute the SVD of a matrix using numpy, you can call the function `[np.linalg.svd](https://numpy.org/doc/stable/reference/generated/numpy.linalg.svd.html)`. Given a matrix *A* with shape (*m*, *n*), the function returns a tuple (*U*, *S*, *V*^{t}), where *U* is a matrix with shape (*m*, *m*) containing the left- singular vectors in its columns, *S* is a vector of size *k* = **min_(_m,** n) containing the singular values in descending order, and*d* V^t is a matrix with shape __ (*n*, n) containing the right singular vectors in its rows.

For example, let's use this function to compute the SVD of the matrix from the previous example:

```
import numpy as np

A = np.array([[1, 0], [0, 1], [2, 2]])
np.linalg.svd(A)
```

The output we get is:

```
(array([[-2.35702260e-01,  7.07106781e-01, -6.66666667e-01],
        [-2.35702260e-01, -7.07106781e-01, -6.66666667e-01],
        [-9.42809042e-01, -1.11022302e-16,  3.33333333e-01]]),
```

```
array([3., 1.]),
array([[-0.70710678, -0.70710678],
       [ 0.70710678, -0.70710678]]))
```

This is the same SVD decomposition we have obtained with our manual computation, up to a sign difference (e.g., the first column of U has flipped direction). This shows that an SVD decomposition of a matrix is not entirely unique. While the singular values themselves are unique, the associated singular vectors (i.e., the columns of U and V) are not strictly unique due to the following reasons:

1. If a singular value is repeated, the corresponding singular vectors can be chosen to be any orthonormal set that spans the associated eigenspace.
2. Even if the singular values are distinct, the corresponding singular vectors can be multiplied by -1 (i.e., their direction can be flipped) and still form a valid SVD.

Compact SVD

The compact singular value decomposition is a reduced form of the full SVD, which retains only the non-zero singular values and their corresponding singular vectors.

Formally, the compact SVD of an $m \times n$ matrix A with rank r ($r \leq \min\{m, n\}$) is a factorization of the form $A = \_U_r\_\Sigma_r V_r^t$, where:

- U_r is an $m \times r$ matrix whose columns are the first r left-singular vectors of A.
- Σ_r is an $r \times r$ diagonal matrix with the r non-zero singular values on the diagonal.
- V_r is an $n \times r$ matrix whose columns are the first r right-singular vectors of A.

For example, the rank of the matrix from our previous example is 2, since it has two non-zero singular values. Therefore, its compact SVD decomposition is:

$$A = U_r \Sigma_r V_r^T = \begin{pmatrix} 1/\sqrt{18} & 1/\sqrt{2} \\ 1/\sqrt{18} & -1/\sqrt{2} \\ 4/\sqrt{18} & 0 \end{pmatrix} \begin{pmatrix} 3 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{pmatrix}$$

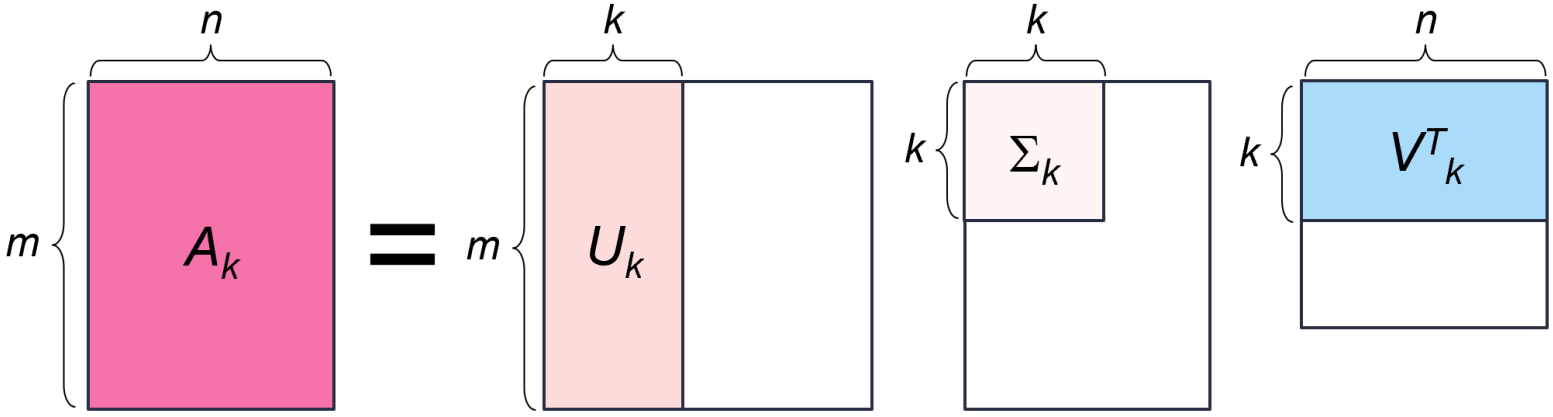
The matrices U_r, Σ_r and V_r contain only the essential information needed to reconstruct the matrix A. The compact SVD can yield significant savings in storage and computation, especially for matrices with many zero singular values (i.e., when $r << \min\{m, n\}$).

Truncated SVD

Truncated (reduced) SVD is a variation of SVD used for approximating the original matrix A with a matrix of a lower rank.

To create a truncated SVD of a matrix A with rank r, we take only the $k < r$ largest singular values and their corresponding singular vectors (k is a parameter). This gives us an approximation of the original matrix $A_k = \_U_k\_\Sigma_k V_k^t$, where:

- U_k is an $m \times k$ matrix whose columns are the first k left-singular vectors of A, corresponding to the k largest singular values.
- Σ_k is an $k \times k$ diagonal matrix with the k largest __ singular values on the diagonal.
- V_k is an $n \times k$ matrix whose columns are the first k right-singular vectors of A, **corresponding to the** k largest singular values.



Truncated SVD

For example, we can truncate the matrix from the previous example to have a rank of $k = 1$ by taking only the largest single value and its corresponding vectors:

$$A_k = U_k \Sigma_k V_k^T = \begin{pmatrix} 1/\sqrt{18} \\ 1/\sqrt{18} \\ 4/\sqrt{18} \end{pmatrix} (3) \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \end{pmatrix} = \begin{pmatrix} 1/2 & 1/2 \\ 1/2 & 1/2 \\ 2 & 2 \end{pmatrix}$$

In NumPy, the truncated SVD can be easily computed using the following code snippet:

```
U, S, Vt = np.linalg.svd(A)

k = 1  # target rank
U_k = U[:, :k]
S_k = np.diag(S[:k])
Vt_k = Vt[:k, :]

A_k = U_k @ S_k @ Vt_k
A_k
```

```
array([[0.5, 0.5],
       [0.5, 0.5],
       [2. , 2. ]])
```

Truncated SVD is particularly effective, since the truncated matrix A_k is the best rank-k approximation of the matrix A in terms of both the Frobenius norm (the least squares difference) and the 2-norm, i.e.,

$$A_k = \underset{\text{rank}(B) \leq k}{\text{argmin}} \; ||A - B||_{F,2}$$

This result is known as the **Eckart-Young-Mirsky theorem** or the matrix approximation lemma, and its proof can be found in this Wikipedia page.
The choice of k controls the tradeoff between approximation accuracy and the compactness of the representation. A smaller k results in a more compact matrix but a rougher approximation. In real-world data matrices, only a very small subset of the singular values are large. In such cases, A_k can be a very good approximation of A by retaining the few singular values that are large.

Dimensionality Reduction with Truncated SVD

Using the truncated SVD, it is also possible to reduce the number of dimensions (features) in the data matrix *A*. To reduce the dimensionality of *A* from *n* to *k*, we project the matrix rows onto the space spanned by the first *k* right-singular vectors. This is done by multiplying the original data matrix *A* by the matrix V_k:

$$A_{\text{reduced}} = AV_k$$

The reduced matrix now has dimensions $n \times k$ and contains the projection of the original data onto the *k*-dimensional subspace. Its columns are the new features in the reduced dimensional space. These features are linear combinations of the original features and are orthogonal to each other.

Using our previous example, we can reduce the number of dimensions of our data matrix *A* from 2 to 1 as follows:

$$A_{\text{reduced}} = AV_k = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 2 & 2 \end{pmatrix} \begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{pmatrix} = \begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \\ 4/\sqrt{2} \end{pmatrix}$$

Another way to compute the reduced matrix is based on the following observation:

$$AV_k = U_k\Sigma_k$$

Proof: The full SVD of *A* is $A = \_U\_\Sigma V^t$, therefore $AV = \_U\_\Sigma$. By comparing the *j*-th columns of each side of the equation we get:

$$A\mathbf{v}_j = \sigma_j \mathbf{u}_j, \quad j = 1, ..., k$$

Therefore, all the columns of AV_k are equal to all the columns of $\_U_k\_\Sigma_k$, so the two matrices must be equal.

Using $\_U_k\_\Sigma_k$ is a more efficient way to compute the reduced matrix, because it requires to multiply matrices of size $m \times k$ and $k \times k$, instead of matrices of size $m \times n$ and $n \times k$ (*k* is typically much smaller than *n*).

In our previous example:

$$A_{\text{reduced}} = U_k\Sigma_k = \begin{pmatrix} 1/\sqrt{18} \\ 1/\sqrt{18} \\ 4/\sqrt{18} \end{pmatrix} (3) = \begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \\ 4/\sqrt{2} \end{pmatrix}$$

Dimensionality reduction using truncated SVD is often used as a data preprocessing step before Machine Learning tasks such as classification or clustering, where it helps dealing with the curse of dimensionality, reduce computational costs and potentially improve the performance of the machine learning algorithm.

To reduce the dimensionality of new data points that arrive after the machine learning model has been trained (e.g., samples in the test set), we simply project them onto the same subspace spanned by the first *k* right-singular vectors of *A*:

$$\mathbf{x}_{\text{reduced}} = (\mathbf{x}^T V_k)^T = V_k^T \mathbf{x}$$

Recall that our convention is to express each vector **x** as a column vector, while data points are stored as rows of *A*, which is why we left-multiply **x** by $V_k{}^t$ instead of right multiplying it by V_k.

Reconstruction Error

A key metric in assessing the effectiveness of dimensionality reduction techniques is called the **reconstruction error**. It provides a quantitative measure of the loss of information resulting from the reduction process.

To measure the reconstruction error of a specific vector, we first project it back onto the original space spanned by the *m* right-singular vectors. This is done by multiplying V_k by the reduced vector:

$$\tilde{\mathbf{x}} = V_k \mathbf{x}_{\text{reduced}}$$

We then measure the reconstruction error as the mean squared error (MSE) between the reconstructed components of the vector and the original components:

$$\text{MSE} = \frac{1}{m}\sum_{i=1}^{m}(x_i - \tilde{x}_i)^2$$

We can also measure the reconstruction error of the entire matrix *A* by projecting the reduced matrix rows back onto the original space spanned by the *m* right-singular vectors. This is done by multiplying the reduced *A* by the transpose of V_k:

$$\tilde{A} = A_{\text{reduced}}V_k^T$$

We can then use either the MSE between the elements of the reconstructed matrix and the original elements, or the Frobenius norm of the difference between the two matrices:

$$||A - \tilde{A}||_F = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{m}(a_{ij} - \tilde{a}_{ij})^2}$$

For example, the reconstructed matrix of the reduced matrix from our previous example is:

$$\tilde{A} = A_{\text{reduced}}V_k^T = \begin{pmatrix}1/\sqrt{2}\\1/\sqrt{2}\\4/\sqrt{2}\end{pmatrix}\begin{pmatrix}1/\sqrt{2} & 1/\sqrt{2}\end{pmatrix} = \begin{pmatrix}1/2 & 1/2\\1/2 & 1/2\\2 & 2\end{pmatrix}$$

And the reconstruction error is:

$$||A - \tilde{A}||_F = \sqrt{(1/2)^2 + (1/2)^2} = 1/\sqrt{2}$$

Truncated SVD in Scikit-Learn

Scikit-Learn provides an efficient implementation of truncated SVD in the class `[sklearn.decomposition.TruncatedSVD](https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.TruncatedSVD.html)`. Its important parameters are:

- `n_components` : Desired number of dimensions for the output data (defaults to 2).
- `algorithm` : The SVD solver to use. Can be one of the following options:
    1. `'arppack'` uses the ARPACK wrapper in SciPy to compute the eigen-decomposition of AA^t or A^tA (whichever is more efficient). ARPACK is an iterative algorithm that efficiently computes a few eigenvalues and eigenvectors of large sparse matrices.
    2. `'randomized'` (the default) uses a fast randomized SVD solver based on an algorithm by Halko et al. [1].
- `n_iter` : The number of iterations for the randomized SVD solver (defaults to 5).

For example, let's demonstrate the usage of this class on our matrix from the previous example:

```
from sklearn.decomposition import TruncatedSVD

svd = TruncatedSVD(n_components=1, random_state=0)
A_reduced = svd.fit_transform(A)
A_reduced
```

The output we get is the reduced matrix:

```
array([[0.70710678],
       [0.70710678],
       [2.82842712]])
```

Example: Image Compression
Singular value decomposition can be used for image compression. Although an image matrix is often of full rank, its lower ranks usually have very small Singular Values. Thus, truncated SVD can lead to a significant reduction in the image size without losing too much information.
For example, we will demonstrate how to use truncated SVD to compress the following image:

Photo taken by the author

We first load the image into a NumPy array using the function `[matplotlib.pyplot.imread]` `(https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.imread.html)` :

```
import matplotlib.pyplot as plt

image = plt.imread('image.jpg')
```

The shape of the image is:

```
image.shape
```

```
(1600, 1200, 3)
```

The height of the image is 1600 pixels, its width is 1200 pixels, and it has 3 color channels (RGB).
Since Svd can only be applied to 2D data, we can either execute it on each color channel separately, or we can reshape the image from a 3D matrix to a 2D matrix by flattening each color channel and stacking them horizontally (or vertically).

For example, the following code snippet reshapes the image into a 2D matrix by stacking the color channels horizontally:

```
height, width, channels = image.shape
flat_image = image.reshape(-1, width * channels)
```

The shape of the flattened image is:

```
flat_image.shape
```

```
(1600, 3600)
```

The rank of the image's matrix is:

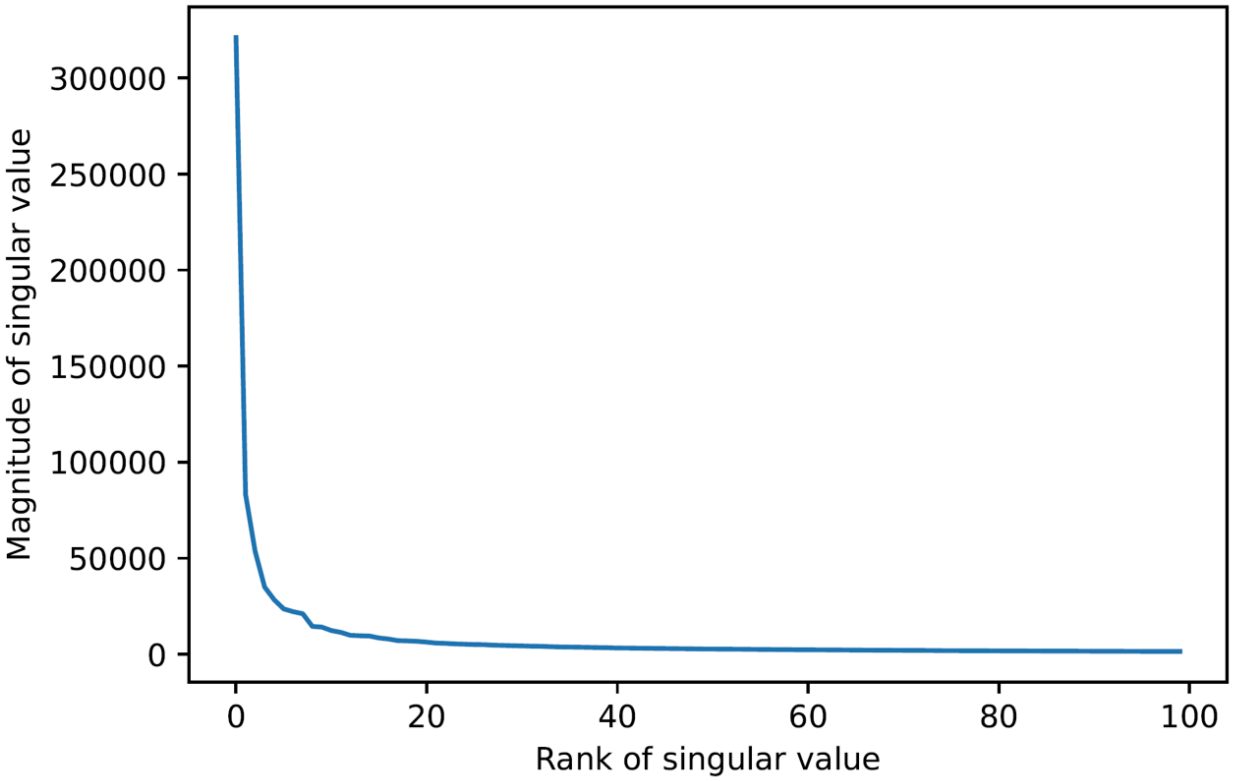
```
np.linalg.matrix_rank(flat_image)
```

```
1600
```

The matrix is of full rank (since min(1600, 3600) = 1600).
Let's plot the first 100 singular values of the matrix:

```
U, S, Vt = np.linalg.svd(flat_image)

k = 100
plt.plot(np.arange(k), S[:k])
plt.xlabel('Rank of singular value')
plt.ylabel('Magnitude of singular value')
```



The first 100 singular values in the image

We can clearly see a rapid decay in the singular values. This decay means that we can effectively truncate the image without a significant loss of accuracy.
For example, let's truncate the image to have a rank of 100 using Truncated SVD:

```
svd = TruncatedSVD(n_components=100)
truncated_image = svd.fit_transform(flat_image)
```

The shape of the truncated image is:

```
truncated_image.shape
```

```
(1600, 100)
```

The size of the truncated image is only 100/3600 = 2.78% of the original image!
To see how much information was lost in the compression we can measure the image's **reconstruction error**. We will measure the reconstruction error as the mean of squared errors (MSE) between the the pixel values of the original image and the reconstructed image.
In Scikit-Learn, the reconstructed image can be obtained by calling the method `inverse_transform` of the `TruncatedSVD` transformer:

```
reconstructed_image = svd.inverse_transform(truncated_image)
```

Therefore, the reconstruction error is:

```
reconstruction_error = np.mean(np.square(reconstructed_image - flat_image))
reconstruction_error
```

```
29.323291415822336
```

Thus, the root mean squared error (RMSE) between the pixel intensities in the original image and the reconstructed image is only about 5.41 (which is small relative to the range of the pixels [0, 255]).
To display the reconstructed image, we first need to reshape it into the original 3D shape and then clip the pixel values to integers in the range [0, 255]:

```
reconstructed_image = reconstructed_image.reshape(height, width, channels)
reconstructed_image = np.clip(reconstructed_image, 0, 255).astype('uint8')
```

We can now display the image using the `plt.imshow` function:

```
plt.imshow(reconstructed_image)
plt.axis('off')
```

The reconstructed image

We can see that the reconstruction at rank 100 loses only a small amount of detail.

Let's place all the above steps into a function that compresses a given 3D image to a specified number of dimensions and then reconstructs it:

```
def compress_image(image, n_components=100):
    # Reshape the 3D image into a 2D array by stacking the color channels horizont
    height, width, channels = image.shape
    flat_image = image.reshape(-1, width * channels)
```

```
    # Truncate the image using SVD
    svd = TruncatedSVD(n_components=n_components)
    truncated_image = svd.fit_transform(flat_image)

    # Recover the image from the reduced representation
    reconstructed_image = svd.inverse_transform(truncated_image)

    # Reshape the image to the original 3D shape
    reconstructed_image = reconstructed_image.reshape(height, width, channels)

    # Clip the output to integers in the range [0, 255]
    reconstructed_image = np.clip(reconstructed_image, 0, 255).astype('uint8')
    return reconstructed_image
```

We can now call this function with different number of components and examine the reconstructions:

```
 fig, axes = plt.subplots(1, 5, figsize=(10, 50))
plt.setp(axes, xticks=[], yticks=[])  # Remove axes from the subplots

for i, k in enumerate([5, 10, 20, 50, 100]):
    output_image = compress_image(image, k)
    axes[i].imshow(output_image)
    axes[i].set_title(f'$k$ = {k}')
```

SVD reconstructions at different ranks

As we can see, using a rank that is too low, such as $k = 10$, can lead to a substantial loss of information, while an SVD of rank 200 is almost indistinguishable from the full-rank image.
In addition to compressing images, SVD can also be used to remove noise from images. This is because discarding the lower-order components of the image tends to remove the granular, noisy elements, while preserving the more significant parts of the image.

Applications of SVD
SVD is employed in many types of applications where it helps to uncover the latent features of the observed data. Examples include:

1. Latent semantic analysis (LSA) is a technique in natural language processing that uncovers the latent relationships between words and text documents by reducing the dimensionality of text data using SVD.

2. In recommendation systems, SVD is used to factorize the user-item interaction matrix, revealing latent features about user preferences and item properties, thus helping the predictive algorithms make more accurate recommendations.
3. SVD can be used to efficiently compute the Moore-Penrose pseudoinverse, which is used in situations where a matrix is not invertible, such as computing the least squares solution to a linear system of equations that lacks a solution.

Limitations of SVD

SVD has several limitations, including:

1. Can be computationally intensive, especially for large matrices. The standard (non-randomized) implementation of SVD has a runtime complexity of $O(mn^2)$ if $m \geq n$, or $O(m^2 n)$ if $m < n$.
2. Requires to store the entire data matrix in memory, which makes it impractical for very large datasets or real-time applications.
3. Assumes that the relationships within the data are linear, which means that SVD may not be able to capture more complex, nonlinear interactions between the variables (features).
4. The latent features obtained from SVD are often not easy to interpret.
5. Standard SVD cannot handle missing data, which means that some form of imputation is needed, potentially introducing biases in the data.
6. There is no simple way to update the SVD incrementally when new data arrives, which is needed in dynamic systems where the data changes frequently (such as real-time recommendation systems).

Final Notes

All the images are by the author unless stated otherwise.
You can find the code examples of this article on my github: https://github.com/roiyeho/medium/tree/main/svd
Thanks for reading!