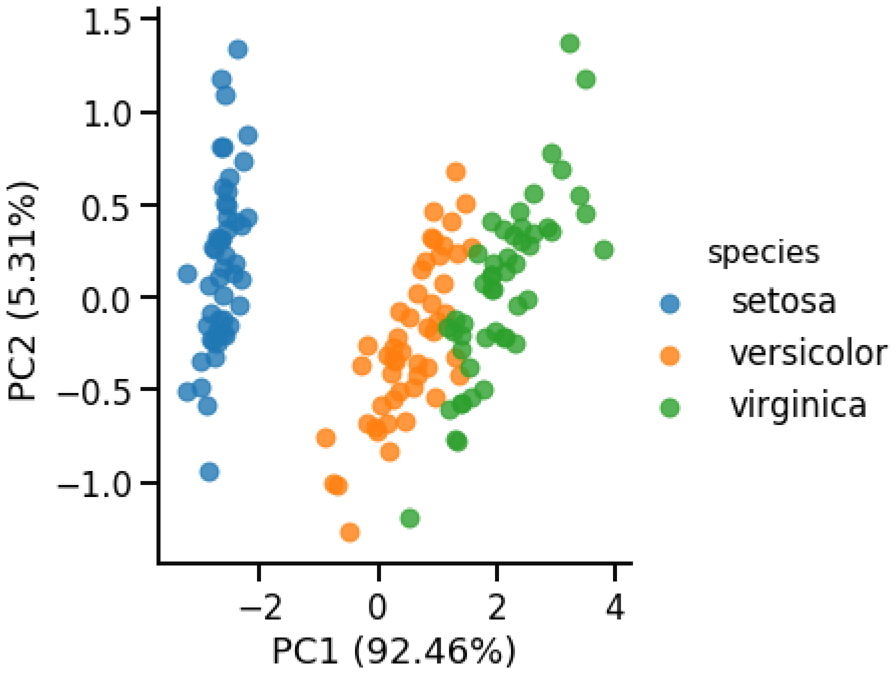
PCA and SVD explained with numpy

Zichen Wang
Mar 16, 2019 · 6 min read ★

How exactly are principal component analysis and singular value decomposition related and how to implement using numpy.



Principal component analysis (PCA) and singular value decomposition (SVD) are commonly used dimensionality reduction approaches in exploratory data analysis (EDA) and Machine Learning. They are both classical linear dimensionality reduction methods that attempt to find linear combinations of features in the original high dimensional data matrix to construct meaningful representation of the dataset. They are preferred by different fields when it comes to reducing the dimensionality: PCA are often used by biologists to analyze and visualize the source variances in datasets from population genetics, transcriptomics, proteomics and microbiome. Meanwhile, SVD, particularly its reduced version truncated SVD, is more popular in the field of natural language processing to achieve a representation of the gigantic while sparse word frequency matrices.

One may find the resultant representations from PCA and SVD are similar in some data. In fact, PCA and SVD are closely related. In this post, I will use some linear algebra and a few lines of numpy code to illustrate their relationship.

0. Linear algebra refresher

Let's first quickly review some basics from linear algebra since both PCA and SVD involves some matrix decomposition.

- **Matrix transpose**: reindex a 2-D matrix ***A*** to switch the row and column indices, effectively replacing all of its elements *a_{ij}* with *a_{ji}*. The notation for transpose is a superscripted ⊤ or *'* on the matrix. In numpy, you can call the .T or .transpose() method of the np.ndarray object to transpose a matrix.

- **Dot product and matrix multiplication**: the product ***C***=***AB*** of two matrices ***A*** *(n×m)* and ***B*** *(m×p)* should have a shape of *n×p*. Two matrices can be multiplied only when the second dimension of the former matches the first dimension of the latter. The element *c_{ij}* in the resultant matrix ***C*** is computed as:

$$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj} = \mathbf{a}_{i,*} \cdot \mathbf{b}_{*,j}$$

Elements in the product matrix of two matrices are the dot products of the corresponding row vectors and column vectors

You may realize that the element in the product matrix ***C*** is the dot product of the corresponding row vector and column vector in matrices ***A*** and ***B***, respectively.

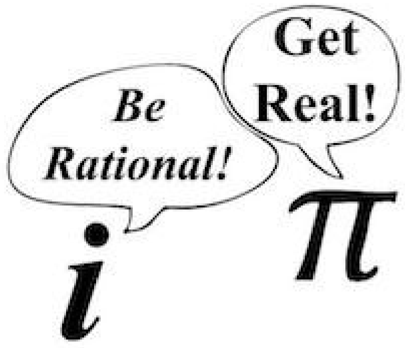
- **Matrix inverse**: only square matrices can be inverted, the product of a matrix ***A*** *(n×n)* with its inverse ***A****^(-1)* is an identity matrix ***I***, where elements on the diagonal are 1's everywhere else are 0's. In numpy, a matrix can be inverted by np.linalg.inv function.

- **Conjugate transpose**: defined as the transpose of a conjugate matrix. Typically denoted with a * or *H* (Hermitian) as superscript. A conjugate matrix is a matrix obtained from taking the complex conjugate of all the elements in the original matrix:

$$\mathbf{A}^* \equiv (\overline{\mathbf{A}})^\top = \overline{\mathbf{A}^\top}$$

Conjugate transpose

Recall complex numbers, where a number is composed of a real part and an imaginary part. For instance, $a + i\,b$ is a complex number where i is the imaginary unit which equals to the square root of -1. The complex conjugate of $a + i\,b$ is a - $i\,b$. Since most datasets we deal with are matrices of real numbers, the conjugate transpose of the matrix is equivalent to a ordinary transpose.



- **Unitary matrix**: defined as a square matrix whose conjugate transpose is also its inverse. For a unitary matrix, we have its transpose equals its inverse:

$$\mathbf{UU}^* = \mathbf{UU}^{-1} = \mathbf{I}$$
$$\mathbf{U}^* = \mathbf{U}^\top = \mathbf{U}^{-1}\text{, if } \mathbf{U} \text{ is real matrix}$$

Unitary matrix, where the conjugate transpose equates the matrix inverse

- **Covariance matrix**: covariance quantifies the joint variability between two random variables X and Y and is calculated as:

$$cov(X, Y) = \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])]$$

Covariance

A covariance matrix $\boldsymbol{C}$ is a square matrix of pairwise covariances of features from the data matrix $\boldsymbol{X}$ *(n samples* $\times$ *m features)*. Observe from the definition of covariance, if two random variables are both centered at 0, the expectations of the random variables become 0's, and the covariance can be calculated as the dot product of the two feature vectors x and y. Therefore, the covariance matrix of a data matrix with all features centered can be computed as:

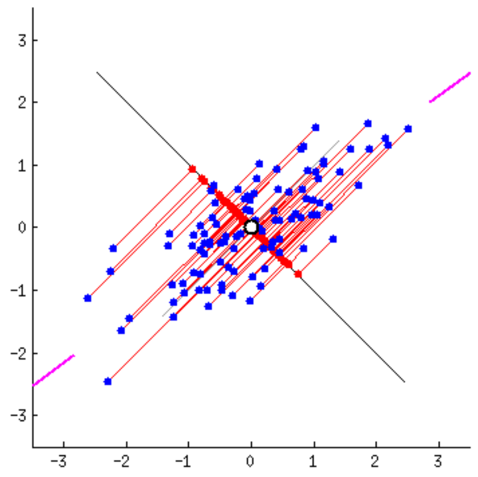
$$\mathbf{C} = \frac{\mathbf{X}^\top \mathbf{X}}{n-1}$$

Covariance matrix of a 0-centered matrix **X**

Okay that brings some nice memories from college linear algebra. Now let's get to know the protagonists of this post.

1. PCA

PCA aims to find linearly uncorrelated orthogonal axes, which are also known as principal components (PCs) in the m dimensional space to project the data points onto those PCs. The first PC captures the largest variance in the data. Let's intuitively understand PCA by fitting it on a 2-D data matrix, which can be conveniently represented by a 2-D scatter plot:



Making sense of PCA by fitting on a 2-D dataset (source)

Since all the PCs are orthogonal to each other, we can use a pair of perpendicular lines in the 2-D space as the two PCs. To make the first PC capture the largest variance, we rotate our pair of PCs to make one of them optimally align with the spread of the data points. Next, all the data points can be projected onto the PCs, and their projections (red dots on PC1) are essentially the resultant dimensionality-reduced representation of the dataset. Viola, we just reduced the matrix from 2-D to 1-D while retaining the largest variance!

The PCs can be determined via eigen decomposition of the covariance matrix ***C***. After all, the geometrical meaning of eigen decomposition is to find a new coordinate system of the eigenvectors for ***C*** through rotations.

$$\mathbf{C} = \mathbf{W}\boldsymbol{\Lambda}\mathbf{W}^{-1}$$

Eigendecomposition of the covariance matrix **C**

In the equation above, the covariance matrix $\boldsymbol{C}(m\times m)$ is decomposed to a matrix of eigenvectors $\boldsymbol{W}(m\times m)$ and a diagonal matrix of m eigenvalues $\boldsymbol{\Lambda}$. The eigenvectors, which are the column vectors in ***W***, are in fact the PCs we are seeking. We can then use matrix multiplication to project the data onto the PC space. For the purpose of dimensionality reduction, we can project the data points onto the first *k* PCs as the representation of the data:

$$\mathbf{X}_k = \mathbf{X}\mathbf{W}_k$$

Project data onto the first k PCs

PCA can be very easily implemented with numpy as the key function performing eigen decomposition (np.linalg.eig) is already built-in:

```python
def pca(X):
  # Data matrix X, assumes 0-centered
  n, m = X.shape
  assert np.allclose(X.mean(axis=0), np.zeros(m))
  # Compute covariance matrix
  C = np.dot(X.T, X) / (n-1)
  # Eigen decomposition
  eigen_vals, eigen_vecs = np.linalg.eig(C)
  # Project X onto PC space
  X_pca = np.dot(X, eigen_vecs)
  return X_pca
```

pca_eigen_docomposition_np.py hosted with ❤ by **GitHub** **view raw**

2. SVD

SVD is another decomposition method for both real and complex matrices. It decomposes a matrix into the product of two unitary matrices ($\boldsymbol{U}$, $\boldsymbol{V}^*$) and a rectangular diagonal matrix of singular values ($\boldsymbol{\Sigma}$):

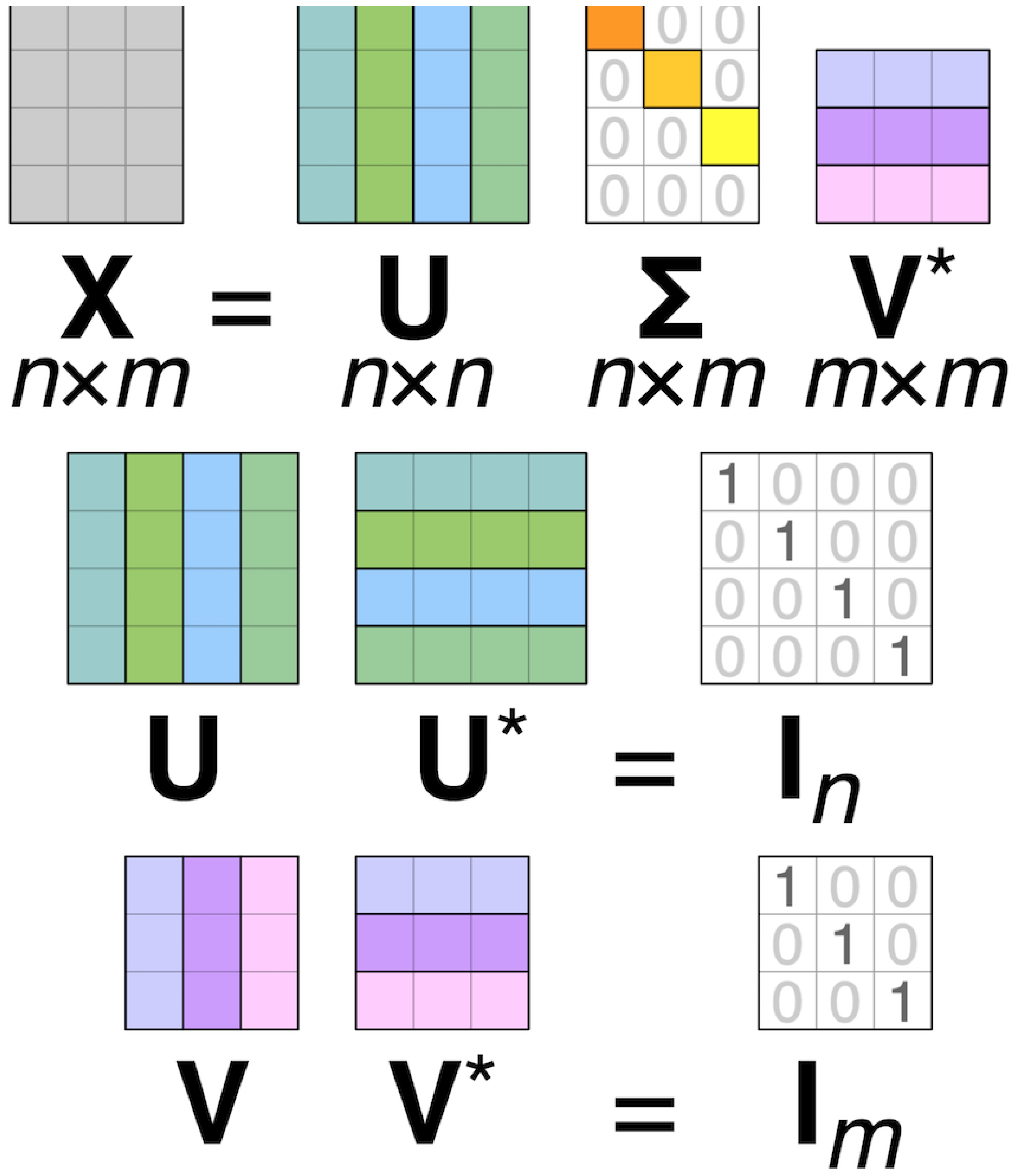


Illustration of SVD, modified from source.

In most cases, we work with real matrix ***X***, and the resultant unitary matrices ***U*** and ***V*** will also be real matrices. Hence, the conjugate transpose of the ***U*** is simply the regular transpose.

SVD has also already been implemented in numpy as np.linalg.svd. To use SVD to transform your data:

```
def svd(X):
```

```python
    # Data matrix X, X doesn't need to be 0-centered
    n, m = X.shape
    # Compute full SVD
    U, Sigma, Vh = np.linalg.svd(X,
        full_matrices=False, # It's not necessary to compute the full matrix of U or V
        compute_uv=True)
    # Transform X with SVD components
    X_svd = np.dot(U, np.diag(Sigma))
    return X_svd
```

svd_np.py hosted with ❤ by GitHub view raw

3. Relationship between PCA and SVD

PCA and SVD are closely related approaches and can be both applied to decompose any rectangular matrices. We can look into their relationship by performing SVD on the covariance matrix ***C***:

$$\begin{aligned}\mathbf{C} &= \frac{\mathbf{X}^\top \mathbf{X}}{n-1} \\ &= \frac{\mathbf{V}\mathbf{\Sigma}\mathbf{U}^\top\mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top}{n-1} \\ &= \mathbf{V}\frac{\mathbf{\Sigma}^2}{n-1}\mathbf{V}^\top \\ &= \mathbf{V}\frac{\mathbf{\Sigma}^2}{n-1}\mathbf{V}^{-1} \text{ (because } \mathbf{V} \text{ is unitary)}\end{aligned}$$

From the above derivation, we notice that the result is in the same form with eigen decomposition of ***C***, we can easily see the relationship between singular values (***Σ***) and eigenvalues (***Λ***):

$$\mathbf{\Lambda} = \frac{\mathbf{\Sigma}^2}{n-1}$$

Relationship between eigenvalue and singular values

To confirm that with numpy:

```python
# Compute covariance matrix
C = np.dot(X.T, X) / (n-1)
# Eigen decomposition
eigen_vals, eigen_vecs = np.linalg.eig(C)
# SVD
```

```python
U, Sigma, Vh = np.linalg.svd(X,
    full_matrices=False,
    compute_uv=True)
# Relationship between singular values and eigen values:
print(np.allclose(np.square(Sigma) / (n - 1), eigen_vals)) # True
```

pca_svd_relationship_np.py hosted with ❤ by GitHub view raw

So what does this imply? It suggests that we can actually perform PCA using SVD, or vice versa. In fact, most implementations of PCA actually use performs SVD under the hood rather than doing eigen decomposition on the covariance matrix because SVD can be much more efficient and is able to handle sparse matrices. In addition, there are reduced forms of SVD which are even more economic to compute.

In the next post, I will delve into the complexities of different SVD solvers and implementations, including numpy, scipy, and the newly developed autograd library Google JAX.

References:

- Wolfram MathWorld
- Making sense of principal component analysis, eigenvectors & eigenvalues
- Relationship between SVD and PCA

Data Science | Machine Learning | Dimensionality Reduction | Numpy | Exploratory Data Analysis