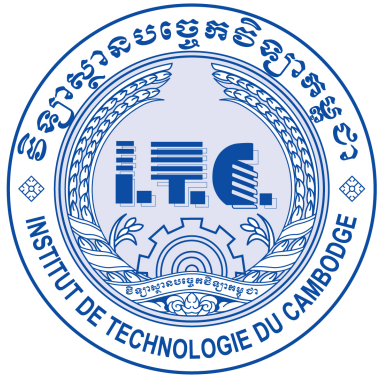

CHAPTER VI
Singular Value Decomposition

PHOK Ponna and NEANG Pheak

Institute of Technology of Cambodia
Department of Applied Mathematics and Statistics (AMS)

2022–2023

Contents

Contents

Introduction

Matrix decomposition, also known as matrix factorization, involves describing a given matrix using its constituent elements. Perhaps the most known and widely used matrix decomposition method is the **Singular-Value Decomposition**, or SVD. All matrices have an SVD, which makes it more stable than other methods, such as the eigendecomposition. As such, it is often used in a wide array of applications including compressing, denoising, and data reduction. In this tutorial, you will discover the Singular-Value Decomposition method for decomposing a matrix into its constituent elements.
The singular value decomposition (SVD) has numerous applications in statistics,machine learning, and computer science.
After completing this tutorial, you will know:

- What Singular-value decomposition is and what is involved.
- How to calculate an SVD and reconstruct a rectangular and square matrix from SVD elements.
- How to calculate the pseudoinverse and perform dimensionality reduction using the SVD.

Contents

The Singular-Value Decomposition, or SVD for short, is a matrix decomposition method for reducing a matrix to its constituent parts in order to make certain subsequent matrix calculationsn simpler. For the case of simplicity we will focus on the SVD for real-valued matrices and ignore the case for complex numbers.

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

where A is the real $m \times n$ matrix that we wish to decompose, U is an $m \times m$ matrix, Σ represented by the uppercase Greek letter sigma) is an $m \times n$ diagonal matrix, and V^T is the transpose of V.
The diagonal values in the Σ matrix are known as the singular values of the original matrix A. The columns of the U matrix are called the left-singular vectors of A, and the columns of V are called the right-singular vectors of A. The SVD is calculated via iterative numericalmethods. The singular value decomposition (SVD) provides another way to factorize a matrix,into singular vectors and singular values. The SVD allows us to discover some ofthe same kind of information as the eigendecomposition. However, the SVD is moregenerally applicable.

Contents

The SVD can be calculated by calling the svd() function. The function takes a matrix and returns the U, Σ and V^T elements. The Σ diagonal matrix is returned as a vector of singularvalues. The V matrix is returned in a transposed form, e.g. V^T. The example below defines a 3×2 matrix and calculates the singular-value decomposition.

```
# eigendecomposition
from numpy import array
from numpy.linalg import eig
# define matrix
A = array([
  [1, 2, 3],
  [4, 5, 6],
   [7, 8, 9]])
print(A)
# factorize
values, vectors = eig(A)
```

```
print(values)
print(vectors)
```

We can confirm that a vector is indeed an eigenvector of a matrix. We do this by multiplying the candidate eigenvector by the value vector and comparing the result with the eigenvalue. First, we will define a matrix, then calculate the eigenvalues and eigenvectors. We will then test whether the first vector and value are in fact an eigenvalue and eigenvector for the matrix.
The eigenvectors are returned as a matrix with the same dimensions as the parent matrix, where each column is an eigenvector, e.g. the first eigenvector is vectors[:, 0]. Eigenvalues are returned as a list, where value indices in the returned array are paired with eigenvectors by column index, e.g. the first eigenvalue at values[0] is paired with the first eigenvector at vectors[:, 0].

```
# confirm eigenvector
from numpy import array
from numpy.linalg import eig
# define matrix
A = array([
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]])
# factorize
values, vectors = eig(A)
# confirm first eigenvector
B = A.dot(vectors[:, 0])
print(B)
C = vectors[:, 0] * values[0]
print(C)
```

The example multiplies the original matrix with the first eigenvector and compares it to the first eigenvector multiplied by the first eigenvalue. Running the example prints the results of these two multiplications that show the same resulting vector, as we would expect.

Contents

We can reverse the process and reconstruct the original matrix given only the eigenvectors and eigenvalues. First, the list of eigenvectors must be taken together as a matrix, where each vector becomes a row. The eigenvalues need to be arranged into a diagonal matrix. The NumPy diag() function can be used for this. Next, we need to calculate the inverse of the eigenvector matrix, which we can achieve with the inv() NumPy function. Finally, these elements need to be multiplied together with the dot() function.

```
# reconstruct matrix
from numpy import diag
from numpy.linalg import inv
from numpy import array
from numpy.linalg import eig
# define matrix
A = array([
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]])
print(A)
# factorize
values, vectors = eig(A)
# create matrix from eigenvectors
Q = vectors
# create inverse of eigenvectors matrix
R = inv(Q)
# create diagonal matrix from eigenvalues
L = diag(values)
# reconstruct the original matrix
B = Q.dot(L).dot(R)
print(B)
```

Contents

We can reverse the process and reconstruct the original matrix given only the eigenvectors and eigenvalues. First, the list of eigenvectors must be taken together as a matrix, where each vector becomes a row. The eigenvalues need to be arranged into a diagonal matrix. The NumPy diag() function can be used for this. Next, we need to calculate the inverse of the eigenvector matrix, which we can achieve with the inv() NumPy function. Finally, these elements need to be multiplied together with the dot() function.

```
# reconstruct matrix
from numpy import diag
from numpy.linalg import inv
from numpy import array
from numpy.linalg import eig
# define matrix
A = array([
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]])
print(A)
# factorize
values, vectors = eig(A)
# create matrix from eigenvectors
Q = vectors
# create inverse of eigenvectors matrix
R = inv(Q)
# create diagonal matrix from eigenvalues
L = diag(values)
# reconstruct the original matrix
B = Q.dot(L).dot(R)
print(B)
```

The above complication with the Σ diagonal only exists with the case where m and n are not equal. The diagonal matrix can be used directly when reconstructing a square matrix, asfollows.

```
# reconstruct square matrix from svd
from numpy import array
from numpy import diag
from scipy.linalg import svd
# define matrix
A = array([
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]])
print(A)
# factorize
U, s, V = svd(A)
# create n x n Sigma matrix
```

```
Sigma = diag(s)
# reconstruct matrix
B = U.dot(Sigma.dot(V))
print(B)
```

Contents

The **pseudoinverse** is the generalization of the matrix inverse for square matrices to rectangular matrices where the number of rows and columns are not equal. It is also called the Moore-PenroseInverse after two independent discoverers of the method or the Generalized Inverse. The pseudoinverse is denoted as A^+, where A is the matrix that is being inverted and $+$ is a superscript. The pseudoinverse is calculated using the singular value decomposition of A:

$$A^+ = V.D^+.U^T$$

Where A^+ is the pseudoinverse, D^+ is the pseudoinverse of the diagonal matrix Σ and U^T is the transpose of U. We can get U and V from the SVD operation.

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

```
# pseudoinverse
from numpy import array
from numpy.linalg import pinv
# define matrix
A = array([
  [0.1, 0.2],
  [0.3, 0.4],
  [0.5, 0.6],
  [0.7, 0.8]])
print(A)
# calculate pseudoinverse
B = pinv(A)
print(B)
```

We can calculate the pseudoinverse manually via the SVD and compare the results to thepinv() function.

```
# pseudoinverse via svd
from numpy import array
from numpy.linalg import svd
from numpy import zeros
from numpy import diag
# define matrix
A = array([
  [0.1, 0.2],
  [0.3, 0.4],
  [0.5, 0.6],
  [0.7, 0.8]])
print(A)
# factorize
U, s, V = svd(A)
# reciprocals of s
d = 1.0 / s
# create m x n D matrix
D = zeros(A.shape)
# populate D with n x n diagonal matrix
D[:A.shape[1], :A.shape[1]] = diag(d)
# calculate pseudoinverse
B = V.T.dot(D.T).dot(U.T)
print(B)
```

Contents

A popular application of SVD is for dimensionality reduction. Data with a large number of features, such as more features (columns) than observations (rows) may be reduced to a smaller subset of features that are most relevant to the prediction problem. The result is a matrix with a lower rank that is said to approximate the original matrix. To do this we can perform an SVD operation on the original data and select the top k largest singular values in Σ. These columnscan be selected from Σ and the rows selected from V^T. An approximate B of the original vector A can then be reconstructed.

$$B = U.\Sigma_k.V_k^T$$

In natural language processing, this approach can be used on matrices of word occurrencesor word frequencies in documents and is called **Latent Semantic Analysis** or Latent Semantic Indexing. In practice, we can retain and work with a descriptive subset of the data called T. This is a dense summary of the matrix or a projection.

$$T = U.\Sigma_k$$

Further, this transform can be calculated and applied to the original matrix A as well as other similar matrices.

$$T = A.V_k^T$$

The example below demonstrates data reduction with the SVD. First a 3×10 matrix is defined, with more columns than rows. The SVD is calculated and only the first two features are selected. The elements are recombined to give an accurate reproduction of the original matrix. Finally the transform is calculated two different ways.

```
# data reduction with svd
from numpy import array
from numpy import diag
from numpy import zeros
from scipy.linalg import svd
# define matrix
A = array([
  [1,2,3,4,5,6,7,8,9,10],
  [11,12,13,14,15,16,17,18,19,20],
  [21,22,23,24,25,26,27,28,29,30]])
print(A)
# factorize
U, s, V = svd(A)
# create m x n Sigma matrix
Sigma = zeros((A.shape[0], A.shape[1]))
# populate Sigma with n x n diagonal matrix
Sigma[:A.shape[0], :A.shape[0]] = diag(s)
# select
n_elements = 2
Sigma = Sigma[:, :n_elements]
V = V[:n_elements, :]
# reconstruct
B = U.dot(Sigma.dot(V))
print(B)
# transform
T = U.dot(Sigma)
print(T)
T = A.dot(V.T)
print(T)
```