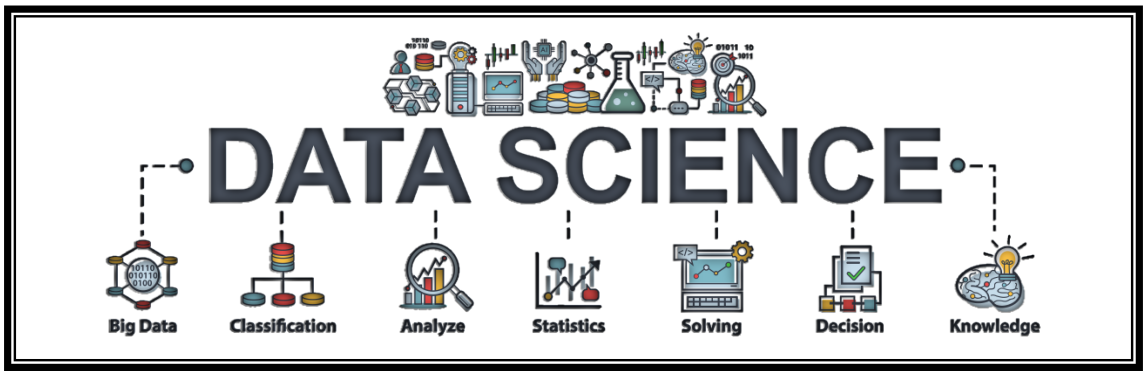
Tikrit University
Computer Science Dept.

Master Degree
Lecture -6-



Assistant Professor

Dr. Eng. Zaidoon.T.AL-Qaysi

College of Computer Science and Mathematics

(2023-2024)

Dimensionality Reduction

The number of input variables or features for a dataset is referred to as its dimensionality. Dimensionality reduction refers to techniques that reduce the number of input variables in a dataset. More input features often make a predictive modeling task more challenging to model, more generally referred to as the curse of dimensionality.

High-dimensionality statistics and dimensionality reduction techniques are often used for data visualization. Nevertheless these techniques can be used in applied machine learning to simplify a classification or regression dataset in order to better fit a predictive model.

- Large numbers of input features can cause poor performance for machine learning algorithms.
- Dimensionality reduction is a general field of study concerned with reducing the number of input features.
- Dimensionality reduction methods include feature selection, linear algebra methods, projection methods, and autoencoders.

Problem With Many Input Variables

The performance of machine learning algorithms can degrade with too many input variables. If your data is represented using rows and columns, such as in a spreadsheet, then the input variables are the columns that are fed as input to a model to predict the target variable. Input variables are also called features. We can consider the columns of data representing dimensions on an n-dimensional feature space and the rows of data as points in that space. This is a useful geometric interpretation of a dataset.

Having a large number of dimensions in the feature space can mean that the volume of that space is very large, and in turn, the points that we have in that space (rows of data) often represent a small and non-representative sample. This can dramatically impact the performance of machine learning algorithms fit on data with many input features, generally referred to as the *curse of dimensionality*. Therefore, it is often desirable to reduce the number of input features. This reduces the number of dimensions of the feature space, hence the name *dimensionality reduction*.

Dimensionality reduction refers to techniques for reducing the number of input variables in training data.

High-dimensionality might mean hundreds, thousands, or even millions of input variables. Fewer input dimensions often mean correspondingly fewer parameters or a simpler structure in the machine learning model, referred to as degrees of freedom. A model with too many degrees of freedom is likely to overfit the training dataset and therefore may not perform well on new data. It is desirable to have simple models that generalize well, and in turn, input data with few input variables. This is particularly true for linear models where the number of inputs and the degrees of freedom of the model are often closely related.

Feature Selection Methods

Perhaps the most common are so-called feature selection techniques that use scoring or statistical methods to select which features to keep and which features to delete.

Two main classes of feature selection techniques include wrapper methods and filter methods. Wrapper methods, as the name suggests, wrap a machine learning model, fitting and evaluating the model with different subsets of input features and selecting the subset the results in the best model performance. RFE is an example of a wrapper feature selection method. Filter methods use scoring methods, like correlation between the feature and the target variable, to select a subset of input features that are most predictive. Examples include Pearson's correlation and Chi-Squared test.

Matrix Factorization

Techniques from linear algebra can be used for dimensionality reduction. Specifically, matrix factorization methods can be used to reduce a dataset matrix into its constituent parts. Examples include the eigendecomposition and singular value decomposition. The parts can then be ranked and a subset of those parts can be selected that best captures the salient structure of the matrix that can be used to represent the dataset. The most common method for ranking the components is principal components analysis, or PCA for short.

Manifold Learning

Techniques from high-dimensionality statistics can also be used for dimensionality reduction.

> In mathematics, a projection is a kind of function or mapping that transforms data in some way.

These techniques are sometimes referred to as *manifold learning* and are used to create a low-dimensional projection of high-dimensional data, often for the purposes of data visualization. The projection is designed to both create a low-dimensional representation of the dataset whilst best preserving the salient structure or relationships in the data. Examples of manifold learning techniques include:

- Kohonen Self-Organizing Map (SOM).
- Sammons Mapping
- Multidimensional Scaling (MDS)
- t-distributed Stochastic Neighbor Embedding (t-SNE).

The features in the projection often have little relationship with the original columns, e.g. they do not have column names, which can be confusing to beginners.

Autoencoder Methods

An auto-encoder is a kind of unsupervised neural network that is used for dimensionality reduction and feature discovery. More precisely, an auto-encoder is a feedforward neural network that is trained to predict the input itself.

After training, the decoder is discarded and the output from the bottleneck is used directly as the reduced dimensionality of the input. Inputs transformed by this encoder can then be fed into another model, not necessarily a neural network model.

Deep autoencoders are an effective framework for nonlinear dimensionality reduction. Once such a network has been built, the top-most layer of the encoder, the code layer hc, can be input to a supervised classification procedure.

Matrix Decomposition

A matrix decomposition is a way of reducing a matrix into its constituent parts. It is an approach that can simplify more complex matrix operations that can be performed on the decomposed matrix rather than on the original matrix itself. A common analogy for matrix decomposition is the factoring of numbers, such as the factoring of 10 into 2 × 5. For this reason, matrix decomposition is also called matrix factorization. Like factoring real values, there are many ways to decompose a matrix, hence there are a range of different matrix decomposition techniques. Two simple and widely used matrix decomposition methods are the LU matrix decomposition and the QR matrix decomposition.

The LU decomposition is for square matrices and decomposes a matrix into L and U components.

$$A = L \cdot U$$

Or, without the dot notation.

$$A = LU$$

Where A is the square matrix that we wish to decompose, L is the lower triangle matrix and U is the upper triangle matrix.

> The factors L and U are triangular matrices. The factorization that comes from elimination is $A = LU$.

The LU decomposition is found using an iterative numerical process and can fail for those matrices that cannot be decomposed or decomposed easily. A variation of this decomposition that is numerically more stable to solve in practice is called the LUP decomposition, or the LU decomposition with partial pivoting.

$$A = L \cdot U \cdot P$$

The rows of the parent matrix are re-ordered to simplify the decomposition process and the additional P matrix specifies a way to permute the result or return the result to the original order. There are also other variations of the LU. The LU decomposition is often used to simplify the solving of systems of linear equations, such as finding the coefficients in a linear regression, as well as in calculating the determinant and inverse of a matrix.

The LU decomposition can be implemented in Python with the `lu()` function. More specifically, this function calculates an LPU decomposition. The example below first defines a 3×3 square matrix. The LU decomposition is calculated, then the original matrix is reconstructed from the components.

```
# LU decomposition
from numpy import array
from scipy.linalg import lu
# define a square matrix
A = array([
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]])
print(A)
# factorize
P, L, U = lu(A)
print(P)
print(L)
print(U)
# reconstruct
B = P.dot(L).dot(U)
print(B)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]

[[ 0.  1.  0.]
 [ 0.  0.  1.]
 [ 1.  0.  0.]]

[[ 1.         0.         0.        ]
 [ 0.14285714 1.         0.        ]
 [ 0.57142857 0.5        1.        ]]

[[ 7.00000000e+00  8.00000000e+00  9.00000000e+00]
 [ 0.00000000e+00  8.57142857e-01  1.71428571e+00]
 [ 0.00000000e+00  0.00000000e+00 -1.58603289e-16]]

[[ 1.  2.  3.]
 [ 4.  5.  6.]
 [ 7.  8.  9.]]
```

Example1: Sample output from calculating LU decomposition

QR Decomposition

The QR decomposition is for $n \times m$ matrices (not limited to square matrices) and decomposes a matrix into Q and R components.

$$A = Q \cdot R$$

Or, without the dot notation.

$$A = QR$$

Where A is the matrix that we wish to decompose, Q a matrix with the size $m \times m$, and R is an upper triangle matrix with the size $m \times n$. The QR decomposition is found using an iterative numerical method that can fail for those matrices that cannot be decomposed, or decomposed easily. Like the LU decomposition, the QR decomposition is often used to solve systems of linear equations, although is not limited to square matrices.

The QR decomposition can be implemented in NumPy using the `qr()` function. By default, the function returns the Q and R matrices with smaller or *reduced* dimensions that is more economical. We can change this to return the expected sizes of $m \times m$ for Q and $m \times n$ for R by specifying the mode argument as `'complete'`, although this is not required for most applications. The example below defines a 3×2 matrix, calculates the QR decomposition, then reconstructs the original matrix from the decomposed elements.

```
# QR decomposition
from numpy import array
from numpy.linalg import qr
# define rectangular matrix
A = array([
  [1, 2],
  [3, 4],
  [5, 6]])
print(A)
# factorize
Q, R = qr(A, 'complete')
print(Q)
print(R)
# reconstruct
B = Q.dot(R)
print(B)
```

```
[[1 2]
 [3 4]
 [5 6]]

[[-0.16903085 0.89708523 0.40824829]
 [-0.50709255 0.27602622 -0.81649658]
 [-0.84515425 -0.34503278 0.40824829]]

[[-5.91607978 -7.43735744]
 [ 0.         0.82807867]
 [ 0.         0.        ]]

[[ 1.  2.]
 [ 3.  4.]
 [ 5.  6.]]
```

Example 2: Sample output from calculating QR decomposition

Cholesky Decomposition

The Cholesky decomposition is for square symmetric matrices where all values are greater than zero, so-called positive definite matrices. For our interests in machine learning, we will focus on the Cholesky decomposition for real-valued matrices and ignore the cases when working with complex numbers. The decomposition is defined as follows:

$$A = L \cdot L^T$$

Or without the dot notation:

$$A = LL^T$$

Where A is the matrix being decomposed, L is the lower triangular matrix and L^T is the transpose of L. The decompose can also be written as the product of the upper triangular matrix, for example:

$$A = U^T \cdot U$$

Where U is the upper triangular matrix. The Cholesky decomposition is used for solving linear least squares for linear regression, as well as simulation and optimization methods. When decomposing symmetric matrices, the Cholesky decomposition is nearly twice as efficient as the LU decomposition and should be preferred in these cases.

The Cholesky decomposition can be implemented in NumPy by calling the `cholesky()` function. The function only returns L as we can easily access the L transpose as needed. The example below defines a 3×3 symmetric and positive definite matrix and calculates the Cholesky decomposition, then the original matrix is reconstructed.

```
# Cholesky decomposition
from numpy import array
from numpy.linalg import cholesky
# define symmetrical matrix
A = array([
  [2, 1, 1],
  [1, 2, 1],
  [1, 1, 2]])
print(A)
# factorize
L = cholesky(A)
print(L)
# reconstruct
B = L.dot(L.T)
print(B)
```

```
[[2 1 1]
 [1 2 1]
 [1 1 2]]

[[ 1.41421356 0.         0.        ]
 [ 0.70710678 1.22474487 0.        ]
 [ 0.70710678 0.40824829 1.15470054]]

[[ 2.  1.  1.]
 [ 1.  2.  1.]
 [ 1.  1.  2.]]
```

Example 3: Sample output from calculating Cholesky decomposition.

Eigendecomposition of a Matrix

Eigendecomposition of a matrix is a type of decomposition that involves decomposing a square matrix into a set of eigenvectors and eigenvalues.

> One of the most widely used kinds of matrix decomposition is called eigendecomposition, in which we decompose a matrix into a set of eigenvectors and eigenvalues.

A vector is an eigenvector of a matrix if it satisfies the following equation.

$$A \cdot v = \lambda \cdot v$$

This is called the eigenvalue equation, where A is the parent square matrix that we are decomposing, v is the eigenvector of the matrix, and λ is the lowercase Greek letter lambda and represents the eigenvalue scalar. Or without the dot notation.

$$Av = \lambda v$$

A matrix could have one eigenvector and eigenvalue for each dimension of the parent matrix. Not all square matrices can be decomposed into eigenvectors and eigenvalues, and some can only be decomposed in a way that requires complex numbers. The parent matrix can be shown to be a product of the eigenvectors and eigenvalues.

$$A = Q \cdot \Lambda \cdot Q^{-1}$$

Or, without the dot notation.

$$A = Q\Lambda Q^{-1}$$

Where Q is a matrix comprised of the eigenvectors, Λ is the uppercase Greek letter lambda and is the diagonal matrix comprised of the eigenvalues, and Q^{-1} is the inverse of the matrix comprised of the eigenvectors.

> However, we often want to decompose matrices into their eigenvalues and eigenvectors. Doing so can help us to analyze certain properties of the matrix, much as decomposing an integer into its prime factors can help us understand the behavior of that integer.

Eigen is not a name, e.g. the method is not named after "Eigen"; eigen (pronounced eye-gan) is a German word that means *own* or *innate*, as in belonging to the parent matrix. A decomposition operation does not result in a compression of the matrix; instead, it breaks it down into constituent parts to make certain operations on the matrix easier to perform. Like other matrix decomposition methods, Eigendecomposition is used as an element to simplify the calculation of other more complex matrix operations.

> Almost all vectors change direction, when they are multiplied by A. Certain exceptional vectors x are in the same direction as Ax. Those are the "eigenvectors". Multiply an eigenvector by A, and the vector Ax is the number λ times the original x. [...] The eigenvalue λ tells whether the special vector x is stretched or shrunk or reversed or left unchanged — when it is multiplied by A.

Eigendecomposition can also be used to calculate the principal components of a matrix in the Principal Component Analysis method or PCA that can be used to reduce the dimensionality of data in machine learning.

Calculation of Eigendecomposition

An eigendecomposition is calculated on a square matrix using an efficient iterative algorithm, of which we will not go into the details. Often an eigenvalue is found first, then an eigenvector is found to solve the equation as a set of coefficients. The eigendecomposition can be calculated in NumPy using the `eig()` function. The example below first defines a 3×3 square matrix. The eigendecomposition is calculated on the matrix returning the eigenvalues and eigenvectors.

```
# 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)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]

[  1.61168440e+01 -1.11684397e+00 -9.75918483e-16]

[[-0.23197069 -0.78583024  0.40824829]
 [-0.52532209 -0.08675134 -0.81649658]
 [-0.8186735   0.61232756  0.40824829]]
```

Example 4: Sample output from calculating an eigendecomposition.

```
# 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)
```

```
[ -3.73863537 -8.46653421 -13.19443305]

[ -3.73863537 -8.46653421 -13.19443305]
```

Example 5: Sample output from calculating a confirmation of an eigendecomposition.

```
# 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)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]

[[ 1.  2.  3.]
 [ 4.  5.  6.]
 [ 7.  8.  9.]]
```

Example 6: Sample output from reconstructing a matrix from an eigendecomposition

Singular-Value Decomposition

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 calculations simpler. For the case of simplicity we will focus on the SVD for real-valued matrices and ignore the case for complex numbers.

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

Where A is the real $n \times m$ 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 V transpose of an $n \times n$ matrix where T is a superscript.

The Singular Value Decomposition is a highlight of linear algebra.

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 numerical methods.

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 singular values. 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.

```
# singular-value decomposition
from numpy import array
from scipy.linalg import svd
# define a matrix
A = array([
  [1, 2],
  [3, 4],
  [5, 6]])
print(A)
# factorize
U, s, V = svd(A)
print(U)
print(s)
print(V)
```

Running the example first prints the defined 3×2 matrix, then the 3×3 U matrix, 2 element Σ vector, and 2×2 V^T matrix elements calculated from the decomposition.

```
[[1 2]
 [3 4]
 [5 6]]

[[-0.2298477  0.88346102 0.40824829]
 [-0.52474482 0.24078249 -0.81649658]
 [-0.81964194 -0.40189603 0.40824829]]

[ 9.52551809 0.51430058]

[[-0.61962948 -0.78489445]
 [-0.78489445 0.61962948]]
```

Example 7: Sample output from calculating singular-value decomposition

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 columns can be selected from Σ and the rows selected from V^T. An approximate B of the original vector A can then be reconstructed.

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

In natural language processing, this approach can be used on matrices of word occurrences or 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.

$$T = U \cdot \Sigma_k$$

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

$$T = A \cdot V_k^T$$

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)
```

Example 8: Example of calculating data reduction with the SVD

```
[[ 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]]

[[ 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.]]

[[-18.52157747  6.47697214]
 [-49.81310011  1.91182038]
 [-81.10462276 -2.65333138]]

[[-18.52157747  6.47697214]
 [-49.81310011  1.91182038]
 [-81.10462276 -2.65333138]]
```

Example 9: Sample output from calculating data reduction with the SVD

The scikit-learn provides a `TruncatedSVD` class that implements this capability directly. The `TruncatedSVD` class can be created in which you must specify the number of desirable features or components to select, e.g. 2. Once created, you can fit the transform (e.g. calculate V_k^T) by calling the `fit()` function, then apply it to the original matrix by calling the `transform()` function. The result is the transform of A called T above. The example below demonstrates the `TruncatedSVD` class.

```
# svd data reduction in scikit-learn
from numpy import array
from sklearn.decomposition import TruncatedSVD
# 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)
# create transform
svd = TruncatedSVD(n_components=2)
# fit transform
svd.fit(A)
# apply transform
result = svd.transform(A)
print(result)
```

```
[[ 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]]

[[ 18.52157747  6.47697214]
 [ 49.81310011  1.91182038]
 [ 81.10462276 -2.65333138]]
```

Example 10: Sample output from calculating data reduction with the SVD in scikit-learn

Expected Value and Mean

In probability, the average value of some random variable X is called the expected value or the expectation. The expected value uses the notation E with square brackets around the name of the variable; for example:

$$E[X]$$

It is calculated as the probability weighted sum of values that can be drawn.

$$E[X] = \sum x_1 \times p_1, x_2 \times p_2, x_3 \times p_3, \cdots, x_n \times p_n$$

In simple cases, such as the flipping of a coin or rolling a dice, the probability of each event is just as likely. Therefore, the expected value can be calculated as the sum of all values multiplied by the reciprocal of the number of values.

$$E[X] = \frac{1}{n} \times \sum x_1, x_2, x_3, \cdots, x_n$$

In statistics, the mean, or more technically the arithmetic mean or sample mean, can be estimated from a sample of examples drawn from the domain. It is confusing because mean, average, and expected value are used interchangeably. In the abstract, the mean is denoted by the lower case Greek letter mu μ and is calculated from the sample of observations, rather than all possible values.

$$\mu = \frac{1}{n} \times \sum x_1, x_2, x_3, \cdots, x_n$$

Or, written more compactly:

$$\mu = P(x) \times \sum x$$

Where x is the vector of observations and $P(x)$ is the calculated probability for each value. When calculated for a specific variable, such as x, the mean is denoted as a lower case variable name with a line above, called x-bar e.g. $\bar{x}$.

$$\bar{x} = \frac{1}{n} \times \sum_{i=1}^{n} x_i$$

The arithmetic mean can be calculated for a vector or matrix in NumPy by using the `mean()` function. The example below defines a 6-element vector and calculates the mean.

```
# vector mean
from numpy import array
from numpy import mean
# define vector
v = array([1,2,3,4,5,6])
print(v)
# calculate mean
result = mean(v)
print(result)
```

```
[1 2 3 4 5 6]

3.5
```

Example 11: Sample output from calculating a vector mean

The mean function can calculate the row or column means of a matrix by specifying the axis argument and the value 0 or 1 respectively. The example below defines a 2×6 matrix and calculates both column and row means.

```
# matrix means
from numpy import array
from numpy import mean
# define matrix
M = array([
  [1,2,3,4,5,6],
  [1,2,3,4,5,6]])
print(M)
# column means
col_mean = mean(M, axis=0)
print(col_mean)
# row means
row_mean = mean(M, axis=1)
print(row_mean)
```

```
[[1 2 3 4 5 6]
 [1 2 3 4 5 6]]

[ 1.  2.  3.  4.  5.  6.]

[ 3.5 3.5]
```

Example 12: Sample output from calculating matrix means

Variance and Standard Deviation

In probability, the variance of some random variable X is a measure of how much values in the distribution vary on average with respect to the mean. The variance is denoted as the function $Var()$ on the variable.

$$Var[X]$$

Variance is calculated as the average squared difference of each value in the distribution from the expected value. Or the expected squared difference from the expected value.

$$Var[X] = E[(X - E[X])^2]$$

Assuming the expected value of the variable has been calculated ($E[X]$), the variance of the random variable can be calculated as the sum of the squared difference of each example from the expected value multiplied by the probability of that value.

$$Var[X] = \sum p(x_1) \times (x_1 - E[X])^2, p(x_2) \times (x_2 - E[X])^2, \cdots, p(x_n) \times (x_n - E[X])^2$$

If the probability of each example in the distribution is equal, variance calculation can drop the individual probabilities and multiply the sum of squared differences by the reciprocal of the number of examples in the distribution.

$$Var[X] = \frac{1}{n} \times \sum (x_1 - E[X])^2, (x_2 - E[X])^2, \cdots, (x_n - E[X])^2$$

In statistics, the variance can be estimated from a sample of examples drawn from the domain. In the abstract, the sample variance is denoted by the lower case sigma with a 2 superscript indicating the units are squared (e.g. σ^2), not that you must square the final value.

In NumPy, the variance can be calculated for a vector or a matrix using the `var()` function. By default, the `var()` function calculates the population variance. To calculate the sample variance, you must set the `ddof` argument to the value 1. The example below defines a 6-element vector and calculates the sample variance.

```
# vector variance
from numpy import array
from numpy import var
# define vector
v = array([1,2,3,4,5,6])
print(v)
# calculate variance
result = var(v, ddof=1)
print(result)
```

```
[1 2 3 4 5 6]

3.5
```

Example 13: Example of calculating a vector variance.

```
# matrix variances
from numpy import array
from numpy import var
# define matrix
M = array([
  [1,2,3,4,5,6],
  [1,2,3,4,5,6]])
print(M)
# column variances
col_var = var(M, ddof=1, axis=0)
print(col_var)
# row variances
row_var = var(M, ddof=1, axis=1)
print(row_var)
```

```
[[1 2 3 4 5 6]
 [1 2 3 4 5 6]]

[ 0.  0.  0.  0.  0.  0.]

[ 3.5  3.5]
```

Example 14: Sample output from calculating matrix variances.

The standard deviation is calculated as the square root of the variance and is denoted as lowercase s.

$$s = \sqrt{\sigma^2}$$

To keep with this notation, sometimes the variance is indicated as s^2, with 2 as a superscript, again showing that the units are squared. NumPy also provides a function for calculating the standard deviation directly via the `std()` function. As with the `var()` function, the `ddof` argument must be set to 1 to calculate the unbiased sample standard deviation and column and row standard deviations can be calculated by setting the axis argument to 0 and 1 respectively. The example below demonstrates how to calculate the sample standard deviation for the rows and columns of a matrix.

```
# matrix standard deviation
from numpy import array
from numpy import std
# define matrix
M = array([
  [1,2,3,4,5,6],
  [1,2,3,4,5,6]])
print(M)
# column standard deviations
col_std = std(M, ddof=1, axis=0)
print(col_std)
# row standard deviations
row_std = std(M, ddof=1, axis=1)
print(row_std)
```

```
[[1 2 3 4 5 6]
 [1 2 3 4 5 6]]

[ 0.  0.  0.  0.  0.  0.]

[ 1.87082869 1.87082869]
```

Example 15: Sample output from calculating matrix standard deviations

Covariance and Correlation

In probability, covariance is the measure of the joint probability for two random variables. It describes how the two variables change together. It is denoted as the function $cov(X, Y)$, where X and Y are the two random variables being considered.

$$cov(X, Y)$$

Covariance is calculated as expected value or average of the product of the differences of each random variable from their expected values, where $E[X]$ is the expected value for X and $E[Y]$ is the expected value of y.

$$cov(X, Y) = E[(X - E[X] \times (Y - E[Y])]$$

Assuming the expected values for X and Y have been calculated, the covariance can be calculated as the sum of the difference of x values from their expected value multiplied by the difference of the y values from their expected values multiplied by the reciprocal of the number of examples in the population.

$$cov(X, Y) = \frac{1}{n} \times \sum(x - E[X]) \times (y - E[Y])$$

In statistics, the sample covariance can be calculated in the same way, although with a bias correction, the same as with the variance.

$$cov(X, Y) = \frac{1}{n-1} \times \sum(x - E[X]) \times (y - E[Y])$$

The sign of the covariance can be interpreted as whether the two variables increase together (positive) or decrease together (negative). The magnitude of the covariance is not easily interpreted. A covariance value of zero indicates that both variables are completely independent. NumPy does not have a function to calculate the covariance between two variables directly. Instead, it has a function for calculating a covariance matrix called `cov()` that we can use to retrieve the covariance. By default, the `cov()`function will calculate the unbiased or sample covariance between the provided random variables.

The example below defines two vectors of equal length with one increasing and one decreasing. We would expect the covariance between these variables to be negative. We access just the covariance for the two variables as the $[0, 1]$ element of the square covariance matrix returned.

```
# vector correlation
from numpy import array
from numpy import corrcoef
# define first vector
x = array([1,2,3,4,5,6,7,8,9])
print(x)
# define second vector
y = array([9,8,7,6,5,4,3,2,1])
print(y)
# calculate correlation
corr = corrcoef(x,y)[0,1]
print(corr)
```

```
[1 2 3 4 5 6 7 8 9]
[9 8 7 6 5 4 3 2 1]

-1.0
```

Example16: Sample output from calculating vector correlation

Covariance Matrix

The covariance matrix is a square and symmetric matrix that describes the covariance between two or more random variables. The diagonal of the covariance matrix are the variances of each of the random variables, as such it is often called the variance-covariance matrix. A covariance matrix is a generalization of the covariance of two variables and captures the way in which all variables in the dataset may change together. The covariance matrix is denoted as the uppercase Greek letter Sigma, e.g. Σ. The covariance for each pair of random variables is calculated as above.

$$\Sigma = E[(X - E[X] \times (Y - E[Y])]$$

Where:

$$\Sigma_{i,j} = cov(X_i, X_j)$$

And X is a matrix where each column represents a random variable. The covariance matrix provides a useful tool for separating the structured relationships in a matrix of random variables. This can be used to decorrelate variables or applied as a transform to other variables. It is a key element used in the Principal Component Analysis data reduction method, or PCA for short.

The covariance matrix can be calculated in NumPy using the `cov()` function. By default, this function will calculate the sample covariance matrix. The `cov()` function can be called with a single 2D array where each sub-array contains a feature (e.g. column). If this function is called with your data defined in a normal matrix format (rows then columns), then a transpose of the matrix will need to be provided to the function in order to correctly calculate the covariance of the columns. Below is an example that defines a dataset with 5 observations across 3 features and calculates the covariance matrix.

```
# covariance matrix
from numpy import array
from numpy import cov
# define matrix of observations
X = array([
  [1, 5, 8],
  [3, 5, 11],
  [2, 4, 9],
  [3, 6, 10],
  [1, 5, 10]])
print(X)
# calculate covariance matrix
Sigma = cov(X.T)
print(Sigma)
```

```
[[ 1  5  8]
 [ 3  5 11]
 [ 2  4  9]
 [ 3  6 10]
 [ 1  5 10]]

[[ 1.   0.25 0.75]
 [ 0.25 0.5  0.25]
 [ 0.75 0.25 1.3 ]]
```

Example 17: Sample output from calculating a covariance matrix

Principal Component Analysis

Principal Component Analysis, or PCA for short, is a method for reducing the dimensionality of data. It can be thought of as a projection method where data with m-columns (features) is projected into a subspace with m or fewer columns, whilst retaining the essence of the original data. The PCA method can be described and implemented using the tools of linear algebra.

PCA is an operation applied to a dataset, represented by an $n \times m$ matrix A that results in a projection of A which we will call B. Let's walk through the steps of this operation.

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \\ a_{3,1} & a_{3,2} \end{pmatrix}$$

$$B = PCA(A)$$

The first step is to calculate the mean values of each column.

$$M = mean(A)$$

Next, we need to center the values in each column by subtracting the mean column value.

$$C = A - M$$

The next step is to calculate the covariance matrix of the centered matrix C. Correlation is a normalized measure of the amount and direction (positive or negative) that two columns change together. Covariance is a generalized and unnormalized version of correlation across multiple columns. A covariance matrix is a calculation of covariance of a given matrix with covariance scores for every column with every other column, including itself.

$$V = cov(C)$$

Finally, we calculate the eigendecomposition of the covariance matrix V. This results in a list of eigenvalues and a list of eigenvectors.

$$values, vectors = eig(V)$$

The eigenvectors represent the directions or components for the reduced subspace of B, whereas the eigenvalues represent the magnitudes for the directions. The eigenvectors can be sorted by the eigenvalues in descending order to provide a ranking of the components or axes of the new subspace for A. If all eigenvalues have a similar value, then we know that the existing representation may already be reasonably compressed or dense and that the projection may offer little. If there are eigenvalues close to zero, they represent components or axes of B that may be discarded. A total of m or less components must be selected to comprise the chosen subspace. Ideally, we would select k eigenvectors, called principal components, that have the k largest eigenvalues.

$$B = select(values, vectors)$$

Other matrix decomposition methods can be used such as Singular-Value Decomposition, or SVD. As such, generally the values are referred to as singular values and the vectors of the subspace are referred to as principal components. Once chosen, data can be projected into the subspace via matrix multiplication.

$$P = B^T \cdot A$$

Where A is the original data that we wish to project, B^T is the transpose of the chosen principal components and P is the projection of A. This is called the covariance method for calculating the PCA, although there are alternative ways to calculate it.

Calculate Principal Component Analysis

There is no `pca()` function in NumPy, but we can easily calculate the Principal Component Analysis step-by-step using NumPy functions. The example below defines a small 3×2 matrix, centers the data in the matrix, calculates the covariance matrix of the centered data, and then the eigendecomposition of the covariance matrix. The eigenvectors and eigenvalues are taken as the principal components and singular values and used to project the original data.

Running the example first prints the original matrix, then the eigenvectors and eigenvalues of the centered covariance matrix, followed finally by the projection of the original matrix. Interestingly, we can see that only the first eigenvector is required, suggesting that we could project our 3×2 matrix onto a 3×1 matrix with little loss.

```
# principal component analysis
from numpy import array
from numpy import mean
from numpy import cov
from numpy.linalg import eig
# define matrix
A = array([
  [1, 2],
  [3, 4],
  [5, 6]])
print(A)
# column means
M = mean(A.T, axis=1)
# center columns by subtracting column means
C = A - M
# calculate covariance matrix of centered matrix
V = cov(C.T)
# factorize covariance matrix
values, vectors = eig(V)
print(vectors)
print(values)
# project data
P = vectors.T.dot(C.T)
print(P.T)
```

```
[[1 2]
 [3 4]
 [5 6]]

[[ 0.70710678 -0.70710678]
 [ 0.70710678 0.70710678]]

[8. 0.]

[[-2.82842712 0.        ]
 [ 0.         0.        ]
 [ 2.82842712 0.        ]]
```

Example 18: Sample output from calculating a PCA manually

Principal Component Analysis in scikit-learn

We can calculate a Principal Component Analysis on a dataset using the `PCA()` class in the scikit-learn library. The benefit of this approach is that once the projection is calculated, it can be applied to new data again and again quite easily. When creating the class, the number of components can be specified as a parameter. The class is first fit on a dataset by calling the `fit()` function, and then the original dataset or other data can be projected into a subspace with the chosen number of dimensions by calling the `transform()` function. Once fit, the singular values and principal components can be accessed on the PCA class via the `explained_variance_` and `components_` attributes. The example below demonstrates using this class by first creating an instance, fitting it on a 3×2 matrix, accessing the values and vectors of the projection, and transforming the original data.

```
# principal component analysis with scikit-learn
from numpy import array
from sklearn.decomposition import PCA
# define matrix
A = array([
  [1, 2],
  [3, 4],
  [5, 6]])
print(A)
# create the transform
pca = PCA(2)
# fit transform
pca.fit(A)
# access values and vectors
print(pca.components_)
print(pca.explained_variance_)
# transform data
B = pca.transform(A)
print(B)
```

```
[[1 2]
 [3 4]
 [5 6]]

[[ 0.70710678 0.70710678]
 [ 0.70710678 -0.70710678]]

[  8.00000000e+00  2.25080839e-33]

[[ -2.82842712e+00 2.22044605e-16]
 [  0.00000000e+00  0.00000000e+00]
 [  2.82842712e+00 -2.22044605e-16]]
```

Example19: Sample output from calculating a PCA with scikit-learn

```
# evaluate pca with logistic regression algorithm for classification
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.pipeline import Pipeline
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# define the pipeline
steps = [('pca', PCA(n_components=10)), ('m', LogisticRegression())]
model = Pipeline(steps=steps)
# evaluate model
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
n_scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report performance
print('Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

```
Accuracy: 0.816 (0.034)
```

Example 20: output from evaluating a model on data prepared with PCA transform

```
# compare pca number of components with logistic regression algorithm for classification
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.pipeline import Pipeline
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=7)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  for i in range(1,21):
    steps = [('pca', PCA(n_components=i)), ('m', LogisticRegression())]
    models[str(i)] = Pipeline(steps=steps)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  scores = evaluate_model(model, X, y)
  results.append(scores)
  names.append(name)
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.xticks(rotation=45)
pyplot.show()
```

```
>1 0.542 (0.048)
>2 0.713 (0.048)
>3 0.720 (0.053)
>4 0.723 (0.051)
>5 0.725 (0.052)
>6 0.730 (0.046)
>7 0.805 (0.036)
>8 0.800 (0.037)
>9 0.814 (0.036)
>10 0.816 (0.034)
>11 0.819 (0.035)
>12 0.819 (0.038)
>13 0.819 (0.035)
>14 0.853 (0.029)
>15 0.865 (0.027)
>16 0.865 (0.027)
>17 0.865 (0.027)
>18 0.865 (0.027)
>19 0.865 (0.027)
>20 0.865 (0.027)
```

Example 21: Performance with the number of selected components in the PCA transforms