COMP7630 – Web Intelligence and its Applications

Linear Algebra
+
Principal Component Analysis

Valentino Santucci

(valentino.santucci@unistrapg.it)

Outline

- Basic Linear Algebra
- Principal Component Analysis

Why we need Linear Algebra?

- After vectorization of texts, they become numerical vectors. Moreover, we will work with a series of texts and, by stacking up their feature vectors, we have a data-table of numerical features. This is a matrix!

- In Machine Learning there is often the need to reduce the dimensionality of a dataset (because of visualization and/or effectiveness and/or efficiency and/or noisy data). Dimensionality reduction techniques are based on linear algebra ideas.

- When we will talk about Social Network Analysis, we will describe networks as graphs, so we can work with adjacency or incidence matrices of graphs.

Scalars and Vectors

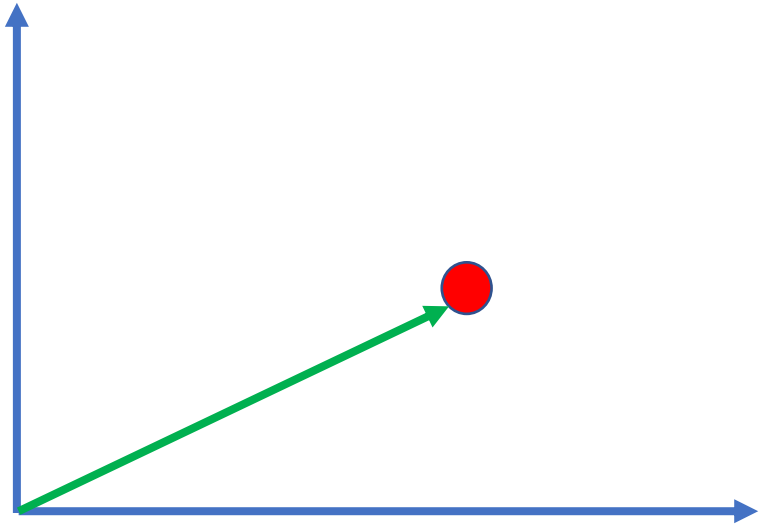
- A scalar is just a single number
  - Real-valued scalar $s \in \mathbb{R}$
  - Natural number scalar $n \in \mathbb{N}$

- A vector is an array/sequence of numbers (not a set)
  - A (column) vector with n real-valued elements $x \in \mathbb{R}^n$

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix}$$

Vectors are both points and "arrows"

- Vectors are:
  - Points in space (1D, 2D, 3D, …n-D) with each element giving the coordinate of the dimension
  - Arrows from the origin to the points with each element giving the displacements of the point from the origin



Matrices (and tensors)

- A matrix is a 2-D array of numbers.
  - A matrix with m rows and n columns $A \in \mathbb{R}^{m \times n}$ (also called m by n matrix)

$$\boldsymbol{A} = \begin{bmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,n} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m,1} & A_{m,2} & \cdots & A_{m,n} \end{bmatrix}$$

- A tensor is an array of numbers with more than two axes.

Vectors and Matrices in Python

- You need to install the NumPy module: `pip install numpy`

```
In [32]: import numpy as np

In [33]: x = np.array([1.1, 2.2, 3.3, 4.4])

In [34]: x[0]
Out[34]: 1.1

In [35]: x[-1]
Out[35]: 4.4

In [36]: x[1:3]
Out[36]: array([2.2, 3.3])

In [37]: x[ x>2.0 ]
Out[37]: array([2.2, 3.3, 4.4])

In [38]: A = np.array([ [1,2,3],
    ...:                [4,5,6],
    ...:                [7,8,9] ])

In [39]: A.shape
Out[39]: (3, 3)

In [40]: x.shape
Out[40]: (4,)

In [41]: A[1][2]
Out[41]: 6

In [42]: A[1,2]
Out[42]: 6

In [43]: A[1,:]
Out[43]: array([4, 5, 6])

In [44]: A[:,2]
Out[44]: array([3, 6, 9])
```

Simple operations on matrices

- Transpose of a matrix: $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ to $\boldsymbol{A}^{\boldsymbol{T}} \in \mathbb{R}^{n \times m}$

$$\boldsymbol{A} = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \\ A_{3,1} & A_{3,2} \end{bmatrix} \quad \boldsymbol{A}^{\boldsymbol{T}} = \begin{bmatrix} A_{1,1} & A_{2,1} & A_{3,1} \\ A_{1,2} & A_{2,2} & A_{3,2} \end{bmatrix}$$

- To save space, people may put column vector as:

$$\boldsymbol{x} = [x_1 \quad x_2 \quad \cdots \quad x_n]^T$$

- Add two matrices: $\boldsymbol{C} = \boldsymbol{A} + \boldsymbol{B} \quad C_{i,j} = A_{i,j} + B_{i,j}$
- Add a scalar to a matrix or multiply by a scalar:

$$\boldsymbol{D} = a \cdot \boldsymbol{B} + c \quad D_{i,j} = a \cdot B_{i,j} + c$$

Matrices simple operations in Python

```
In [50]: import numpy as np

In [51]: T = np.zeros( (3,4) ) + 2

In [52]: T
Out[52]:
array([[2., 2., 2., 2.],
       [2., 2., 2., 2.],
       [2., 2., 2., 2.]])

In [53]: H = np.ones( (3,4) ) + 0.4

In [54]: H
Out[54]:
array([[1.4, 1.4, 1.4, 1.4],
       [1.4, 1.4, 1.4, 1.4],
       [1.4, 1.4, 1.4, 1.4]])

In [55]: S = 3*T + H + 100

In [56]: S
Out[56]:
array([[107.4, 107.4, 107.4, 107.4],
       [107.4, 107.4, 107.4, 107.4],
       [107.4, 107.4, 107.4, 107.4]])

In [57]: S.transpose()
Out[57]:
array([[107.4, 107.4, 107.4],
       [107.4, 107.4, 107.4],
       [107.4, 107.4, 107.4],
       [107.4, 107.4, 107.4]])
```

Product of Matrices

- Product of Matrices $\boldsymbol{A} \in \mathbb{R}^{m \times k}$ and $\boldsymbol{B} \in \mathbb{R}^{k \times n}$

$$\boldsymbol{C} = \boldsymbol{AB} \in \mathbb{R}^{m \times n} \quad C_{i,j} = \sum_k A_{i,k} B_{k,j}$$

E.g. $$\boldsymbol{A} = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix} \quad \boldsymbol{B} = \begin{bmatrix} 11 & 14 & 17 \\ 12 & 15 & 18 \\ 13 & 16 & 19 \end{bmatrix}$$

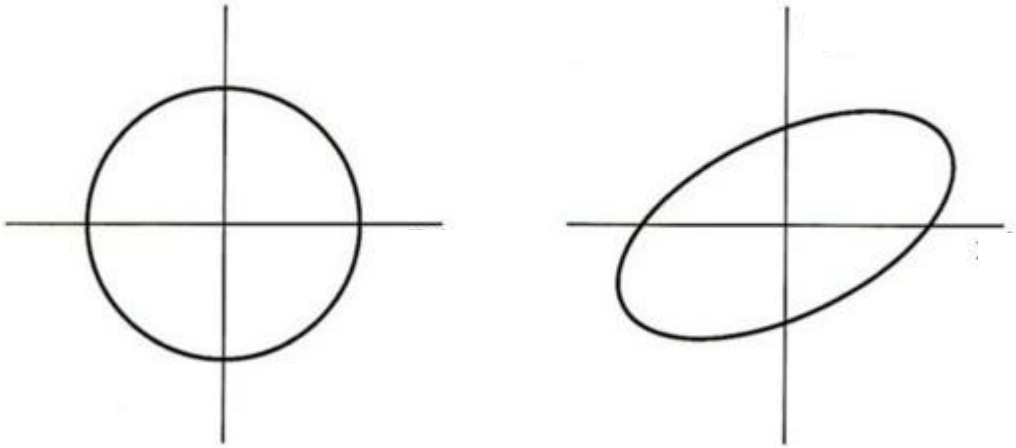
$$\boldsymbol{C} = \begin{bmatrix} 1\times11 + 4\times12 + 7\times13 & 1\times14 + 4\times15 + 7\times16 & 1\times17 + 4\times18 + 7\times19 \\ 2\times11 + 5\times12 + 8\times13 & 2\times14 + 5\times15 + 8\times16 & 2\times17 + 5\times18 + 8\times19 \\ 3\times11 + 6\times12 + 9\times13 & 3\times14 + 6\times15 + 9\times16 & 3\times17 + 6\times18 + 9\times19 \end{bmatrix}$$

Matrix multiplication in Python

```
In [65]: import numpy as np

In [66]: A1 = np.array( [ [1,4,5],
    ...:                  [-5,8,9] ] )

In [67]: B1 = np.array( [ [1,1,1],
    ...:                  [2,1,1] ] )

In [68]: A1.shape
Out[68]: (2, 3)

In [69]: B1.transpose().shape
Out[69]: (3, 2)

In [70]: A1.dot(B1.transpose())
Out[70]:
array([[10, 11],
       [12,  7]])
```

Geometric interpretation of matrix-vector product

- Given a square matrix $A \in \mathbb{R}^{n \times n}$ and a vector $x \in \mathbb{R}^n$
- The vector $y = Ax$ is a *n*-dimensional vector like x, i.e. $y \in \mathbb{R}^n$
- The matrix A is a linear application ("affine application" to be precise) which "moves" the points of a vector space, thus "distorting figures" in the vector space



… and if the matrix is rectangular?

- Given a rectangular matrix $A \in \mathbb{R}^{m \times n}$ and a vector $x \in \mathbb{R}^{n}$, with *m*<*n*
- The vector $y = Ax$ is a *m*-dimensional vector
- Hence, *y* is the projection of *x* in a lower dimensional space
  - *x* is a point of the *n*-dimensional space
  - *y* is a point of the *m*-dimensional space (*m*<n)
- Hence, the matrix *A* acts on *x* as a projection

Other important products

- Element-wise Product of $\boldsymbol{A}$ & $\boldsymbol{B}$ $(\in \mathbb{R}^{m\times n})$ $\quad \boldsymbol{C} = \boldsymbol{A} \odot \boldsymbol{B}$
- Dot product of vectors $\boldsymbol{x} \in \mathbb{R}^m$ & $\boldsymbol{y} \in \mathbb{R}^m$ $\quad \boldsymbol{x}^T\boldsymbol{y} \in \mathbb{R}$

E.g. $\boldsymbol{x}^T = [1 \quad 2 \quad 3] \quad \boldsymbol{y} = \begin{bmatrix} 5 \\ 6 \\ 7 \end{bmatrix} \quad \boldsymbol{x}^T\boldsymbol{y} = 1\times5 + 2\times6 + 3\times7$

- Outer product of vectors $\boldsymbol{x} \in \mathbb{R}^m$ & $\boldsymbol{y} \in \mathbb{R}^n$ $\boldsymbol{x}\boldsymbol{y}^T \in \mathbb{R}^{m\times n}$

E.g. $\boldsymbol{x} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \quad \boldsymbol{y}^T = [5 \quad 6 \quad 7] \quad \boldsymbol{x}\boldsymbol{y}^T = \begin{bmatrix} 1\times5 & 1\times6 & 1\times7 \\ 2\times5 & 2\times6 & 2\times7 \\ 3\times5 & 3\times6 & 3\times7 \end{bmatrix}$

Element-wise product in Python

```
In [1]: import numpy as np

In [2]: A1 = np.array([[ 1, 4, 5],
   ...:                [-5, 8, 9]])

In [3]: B1 = np.array([[ 1, 1, 1],
   ...:                [ 2, 1, 1]])

In [4]: Pe = np.multiply(A1,B1)

In [5]: Pe
Out[5]:
array([[  1,   4,   5],
       [-10,   8,   9]])
```

Vector dot and outer products in Python

```
In [5]: import numpy as np

In [6]: A = np.array([ [1,4,5], [-5,8,9] ])

In [7]: A
Out[7]:
array([[ 1,  4,  5],
       [-5,  8,  9]])

In [8]: #inner product of the vectors A[0] and A[1]

In [9]: np.dot(A[0],A[1])
Out[9]: 72

In [10]: #outer product of the vectors A[0] and A[1]

In [11]: np.outer(A[0],A[1])
Out[11]:
array([[ -5,   8,   9],
       [-20,  32,  36],
       [-25,  40,  45]])
```

Inverse of a (Square) Matrix

- Identity Matrix (all entries along the diagonal are 1)

$$\boldsymbol{I} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

- Matrix Inverse $\boldsymbol{A^{-1}}$

$$\boldsymbol{A^{-1}A = I} \quad (\boldsymbol{AA^{-1} = I})$$

- Why powerful?

$$\boldsymbol{Ax = b} \quad \boldsymbol{A^{-1}Ax = A^{-1}b} \quad \boldsymbol{Ix = A^{-1}b} \quad \boldsymbol{x = A^{-1}b}$$

Matrix Inversion in Python

```
In [22]: import numpy as np

In [23]: A = np.array( [ [1,4,5], [-5,8,9], [1,2,3] ] )

In [24]: A
Out[24]:
array([[ 1,  4,  5],
       [-5,  8,  9],
       [ 1,  2,  3]])

In [25]: A_inv = np.linalg.inv(A)

In [26]: A_inv
Out[26]:
array([[0.50, -0.17, -0.33],
       [2.00, -0.17, -2.83],
       [-1.50, 0.17, 2.33]])

In [27]: A.dot(A_inv)
Out[27]:
array([[1.00, -0.00, -0.00],
       [-0.00, 1.00, 0.00],
       [-0.00, -0.00, 1.00]])

In [28]: A_inv.dot(A)
Out[28]:
array([[1.00, -0.00, -0.00],
       [0.00, 1.00, -0.00],
       [0.00, -0.00, 1.00]])

In [29]: np.eye(3)
Out[29]:
array([[1.00, 0.00, 0.00],
       [0.00, 1.00, 0.00],
       [0.00, 0.00, 1.00]])
```

Vectorial norms and a dot-product property

- L^p norm - a function to measure magnitude of a vector

$$\|\boldsymbol{x}\|_p = \left(\sum_i |x_i|^p\right)^{\frac{1}{p}}$$

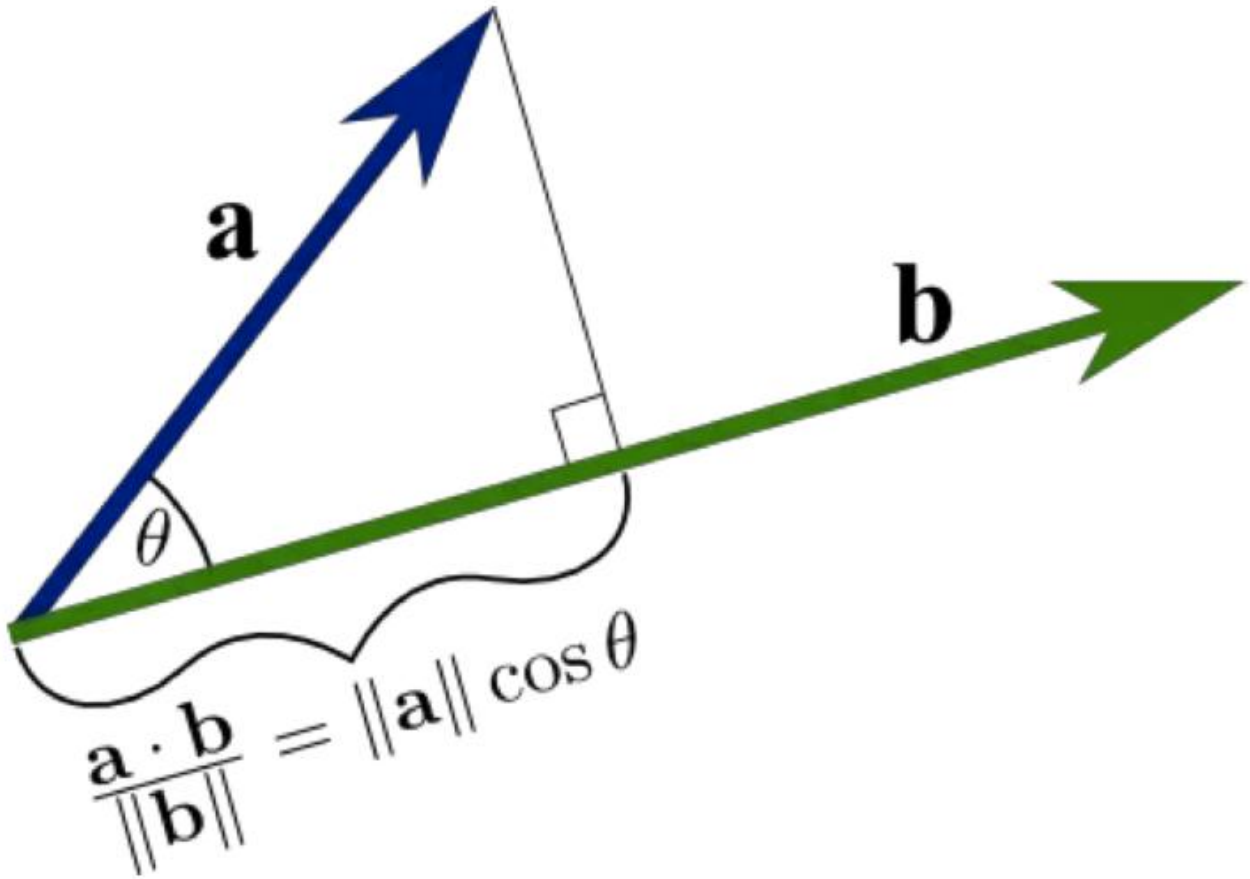
- Most common one L^2 norm (called Euclidean norm)

$$\|\boldsymbol{x}\|_2 = \left(\sum_i |x_i|^2\right)^{\frac{1}{2}} = \boldsymbol{x}^T\boldsymbol{x}$$

- Dot product of $\boldsymbol{x}$ and $\boldsymbol{y}$ (θ is the angle between them)

$$\boldsymbol{x}^T\boldsymbol{y} = \|\boldsymbol{x}\|_2\|\boldsymbol{y}\|_2 \cos\theta$$

Dot Product and Projection



Dot product is what allows to consider vectors as coordinates of a point in a space

- Given an orthonormal set of vectors (called basis of the vector space)
  - For instance, with *n=3* we have the following basis:

$$e_1 = (1,0,0); e_2 = (0,1,0); e_3 = (0,0,1)$$

- Suppose *n=3,* a generic vector like $x = (x_1, x_2, x_3)$ may be understood as

$$x = \sum_{i=1}^{3} x_i e_i$$

Cosine Similarity

- The dot-product measures the "correlation" between two vectors
  - The dot product is high (positive) when vectors have similar directions, indicating a positive correlation.
  - The dot product is low (near zero or negative) when vectors have dissimilar directions, indicating a lower correlation or negative correlation.
  - The dot product is 0 when the two vectors are orthogonal.

- The cosine similarity between two vectors is their dot-product normalized by their Euclidean norms

$$CosineSimilarity(x, y) = \frac{x \cdot y}{\|x\|_2 \|y\|_2}$$

- The cosine similarity is in [-1,+1]
  - … but if *x* and *y* have all non-negative entries, then their cosine similarity is in [0,1]

- Some libraries (like Scipy) have a function called *CosineDistance* which returns 1 – *CosineSimilirity*(*x*,*y*)
  - It is easy to see that its codomain is in [0,2].

Matrix Norm and Trace

- Frobenius norm – a function to measure the magnitude of a matrix

$$\|\boldsymbol{A}\|_F = \sqrt{\sum_{i,j} \boldsymbol{A}_{i,j}^2}$$

- Frobenius norm can be expressed also using the trace operator

$$\|A\|_F = \sqrt{\mathrm{Tr}(A^\top A)}$$

where the trace of A is the sum of its diagonal elements

$$\mathrm{Tr}(A) = \sum_{i=1}^{n} a_{ii}$$

Norms in Python

```
In [31]: import numpy as np

In [32]: a = np.arange(9) - 4

In [33]: a
Out[33]: array([-4, -3, -2, -1,  0,  1,  2,  3,  4])

In [34]: B = a.reshape( (3,3) )

In [35]: B
Out[35]:
array([[-4, -3, -2],
       [-1,  0,  1],
       [ 2,  3,  4]])

In [36]: #L2 norm of the vector a

In [37]: np.linalg.norm(a)
Out[37]: 7.745966692414834

In [38]: #L2 norm of the matrix B

In [39]: np.linalg.norm(B)
Out[39]: 7.745966692414834

In [40]: #L-infinity norm of the vector a

In [41]: np.linalg.norm(a, np.inf)
Out[41]: 4.0
```

Some special matrices

- Diagonal matrix (only non-zero entries along the main diagonal) $D_{i,j} = 0$ for all $i \neq j$

$$diag(\boldsymbol{v}) = \begin{bmatrix} v_1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & v_n \end{bmatrix} \quad diag(\boldsymbol{v})^{-1} = \begin{bmatrix} 1/v_1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & 1/v_n \end{bmatrix}$$

- Symmetric matrix: $\boldsymbol{A} = \boldsymbol{A}^T \quad A_{i,j} = A_{j,i}$
- Unit vector: $\|\boldsymbol{x}\|_2 = 1$
- Orthogonal (orthonormal): $\boldsymbol{x}^T\boldsymbol{y} = 0$ (also $\|\boldsymbol{x}\|_2 = \|\boldsymbol{y}\|_2 = 1$)
- Orthogonal matrix: $\boldsymbol{A}^T\boldsymbol{A} = \boldsymbol{I}$; that is $\boldsymbol{A}^{-1} = \boldsymbol{A}^T$

Rank of a matrix

- The rank of a matrix is the dimension of the vector space spanned by its columns …
- … or, equivalently, is the maximal number of linearly independent columns in the matrix
- … or, equivalently, is the maximal number of linearly independent rows in the matrix
- … or, equivalently, is the number of non-zero eigenvalues or singular-values of the matrix

- A *n*-by-*n* matrix is invertible iff its rank is full, i.e., it is equal to *n*
- A matrix without full rank is a kind of "projection operator" because it projects vectors to a (usually lower) dimensional space
  - So, "it loses information" and this intuitively explains why such matrices are not invertible

- The outer product of two vectors create a matrix with rank 1

Eigendecomposition

- Decompose a square matrix -> a set of *eigenvectors* and *eigenvalues*
- Def: An eigenvector of a square matrix $\boldsymbol{A}$:
  - A nonzero vector $\boldsymbol{v}$ where multiplication by $\boldsymbol{A}$ alters only the scale of $\boldsymbol{v}$

$$\boldsymbol{A}\boldsymbol{v} = \lambda\boldsymbol{v} \qquad (\boldsymbol{v}^{\mathrm{T}}\boldsymbol{A} = \lambda\boldsymbol{v}^{\mathrm{T}})$$

- The scalar λ is known as the eigenvalue corresponding to the eigenvector.
- We usually look for unit eigenvectors only. (Why?)

Eigendecomposition

- Suppose a matrix $\boldsymbol{A}$ has n eigenvectors $\{\boldsymbol{v}^{(1)}, \cdots, \boldsymbol{v}^{(n)}\}$ with corresponding eigenvalues $\{\lambda_1, \cdots, \lambda_n\}$.
- Define: $\boldsymbol{V} = \left[\boldsymbol{v}^{(1)}, \cdots, \boldsymbol{v}^{(n)}\right]$ $\boldsymbol{\lambda} = [\lambda_1, \cdots, \lambda_n]^{\mathrm{T}}$
- Eigendecomposition of $\boldsymbol{A}$

$$\boldsymbol{A}\boldsymbol{v}^{(1)} = \lambda_1 \boldsymbol{v}^{(1)} \quad \ldots \quad \boldsymbol{A}\boldsymbol{v}^{(n)} = \lambda_n \boldsymbol{v}^{(n)}$$

$$\boldsymbol{A}\boldsymbol{V} = \boldsymbol{V} diag(\boldsymbol{\lambda})$$

$$\boldsymbol{A} = \boldsymbol{V}\, diag(\boldsymbol{\lambda})\, \boldsymbol{V}^{-1}$$

Eigendecomposition of a symmetric matrix

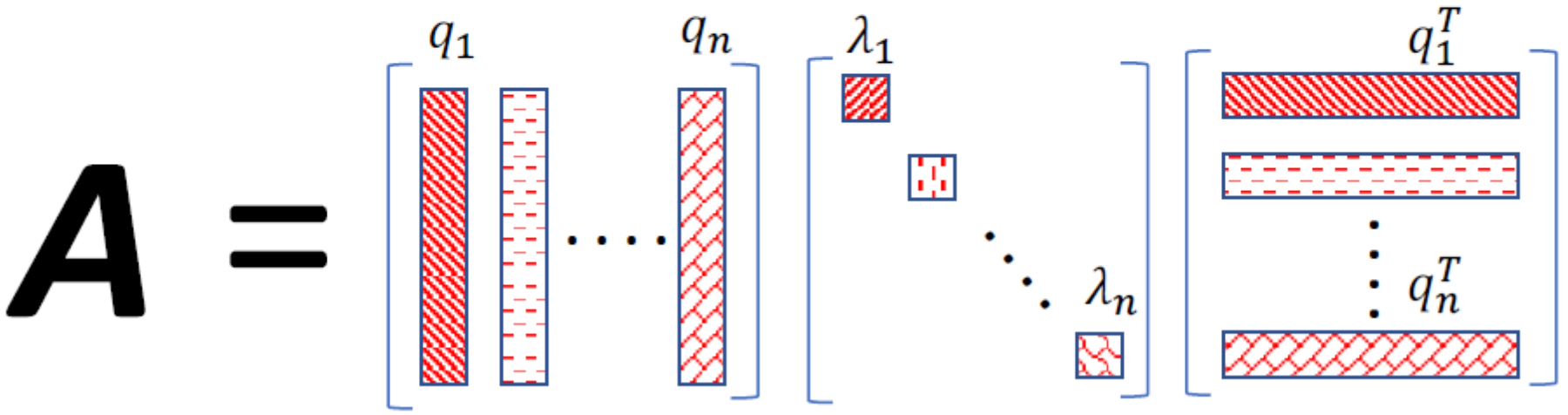
- For real symmetric matrix $\boldsymbol{A}$, the eigenvectors and eigenvalues will be *real-valued*, and the eigenvectors will be *orthogonal*:

$$\boldsymbol{A} = \boldsymbol{Q}\, diag(\boldsymbol{\lambda})\, \boldsymbol{Q}^{\mathrm{T}}$$

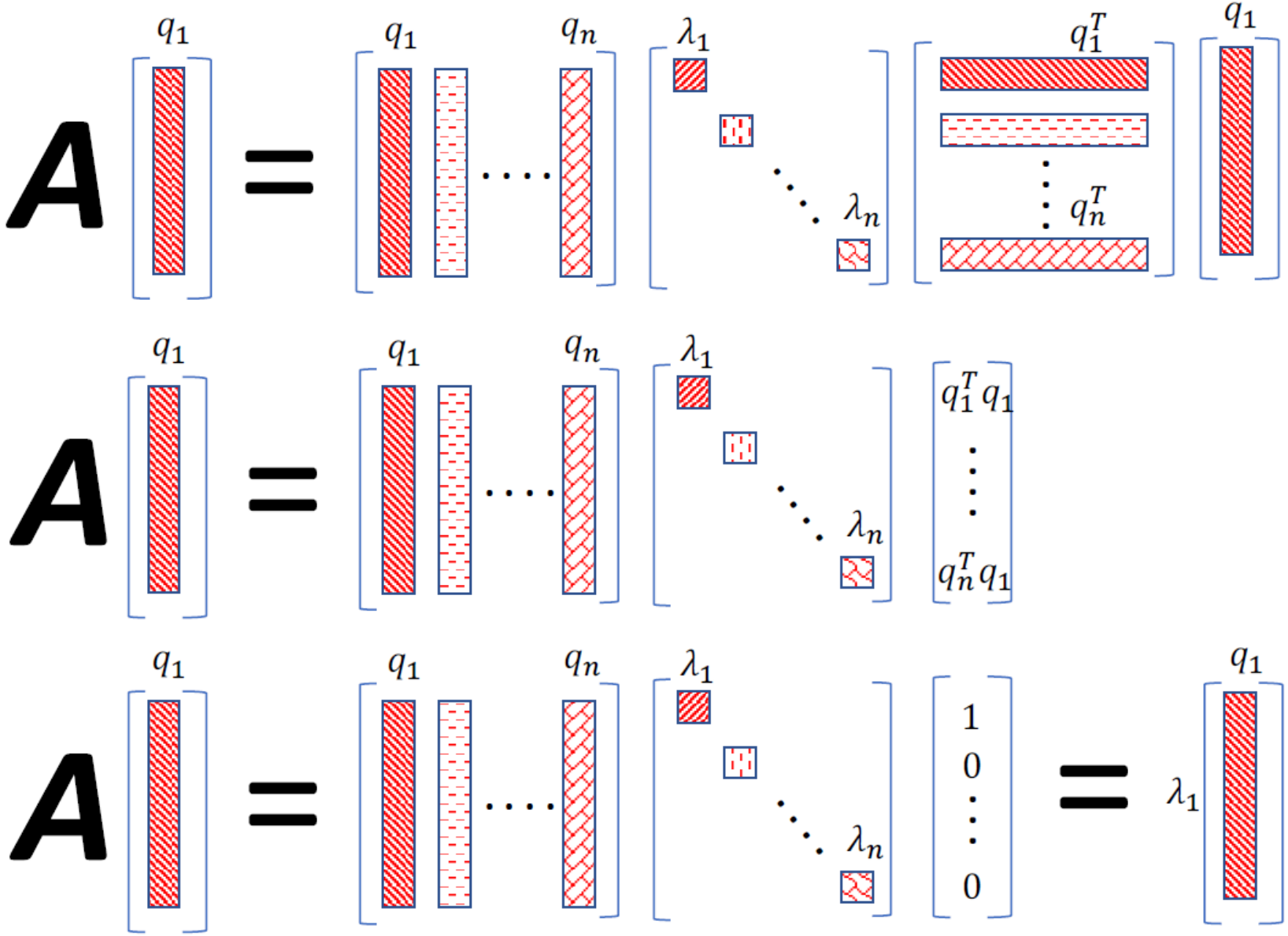
- Important for *principle component analysis*!

Eigendecomposition of a symmetric matrix

$$\boldsymbol{A} = \boldsymbol{Q}\, diag(\boldsymbol{\lambda})\, \boldsymbol{Q}^{\mathrm{T}}$$



Multiplying a matrix by one of its eigenvectors



Eigendecomposition in Python

```
In [27]: import numpy as np

In [28]: A = np.array([[1,4,5],[-5,8,9],[1,2,3]])

In [29]: A
Out[29]:
array([[ 1,  4,  5],
       [-5,  8,  9],
       [ 1,  2,  3]])

In [30]: val,vec = np.linalg.eig(A)

In [31]: print(f'Eigenvalue #0: {val[0]}')
Eigenvalue #0: 0.44720089256937 89

In [32]: print(f'Eigenvector #0: {vec[:,0]}')
Eigenvector #0: [ 0.04575924  0.77757688 -0.62712064]

In [33]: print(f'Eigenvalue #1: {val[1]}')
Eigenvalue #1: 3.2203843196683564

In [34]: print(f'Eigenvector #1: {vec[:,1]}')
Eigenvector #1: [ 0.75089651 -0.31126489  0.58246768]

In [35]: print(f'Eigenvalue #2: {val[2]}')
Eigenvalue #2: 8.332414787762268

In [36]: print(f'Eigenvector #2: {vec[:,2]}')
Eigenvector #2: [0.6275103  0.68298774 0.37384297]
```

Multiplying a matrix by its eigenvectors

```
In [38]: A.dot(vec[:,0])
Out[38]: array([ 0.02046358,  0.34773307, -0.28044891])

In [39]: val[0] * vec[:,0]
Out[39]: array([ 0.02046358,  0.34773307, -0.28044891])

In [40]: A.dot(vec[:,1])
Out[40]: array([ 2.41817536, -1.00239256,  1.87576978])

In [41]: val[1] * vec[:,1]
Out[41]: array([ 2.41817536, -1.00239256,  1.87576978])

In [42]: val[2] * vec[:,2]
Out[42]: array([5.22867612, 5.69093715, 3.1150147 ])

In [43]: val[2] * vec[:,2]
Out[43]: array([5.22867612, 5.69093715, 3.1150147 ])
```

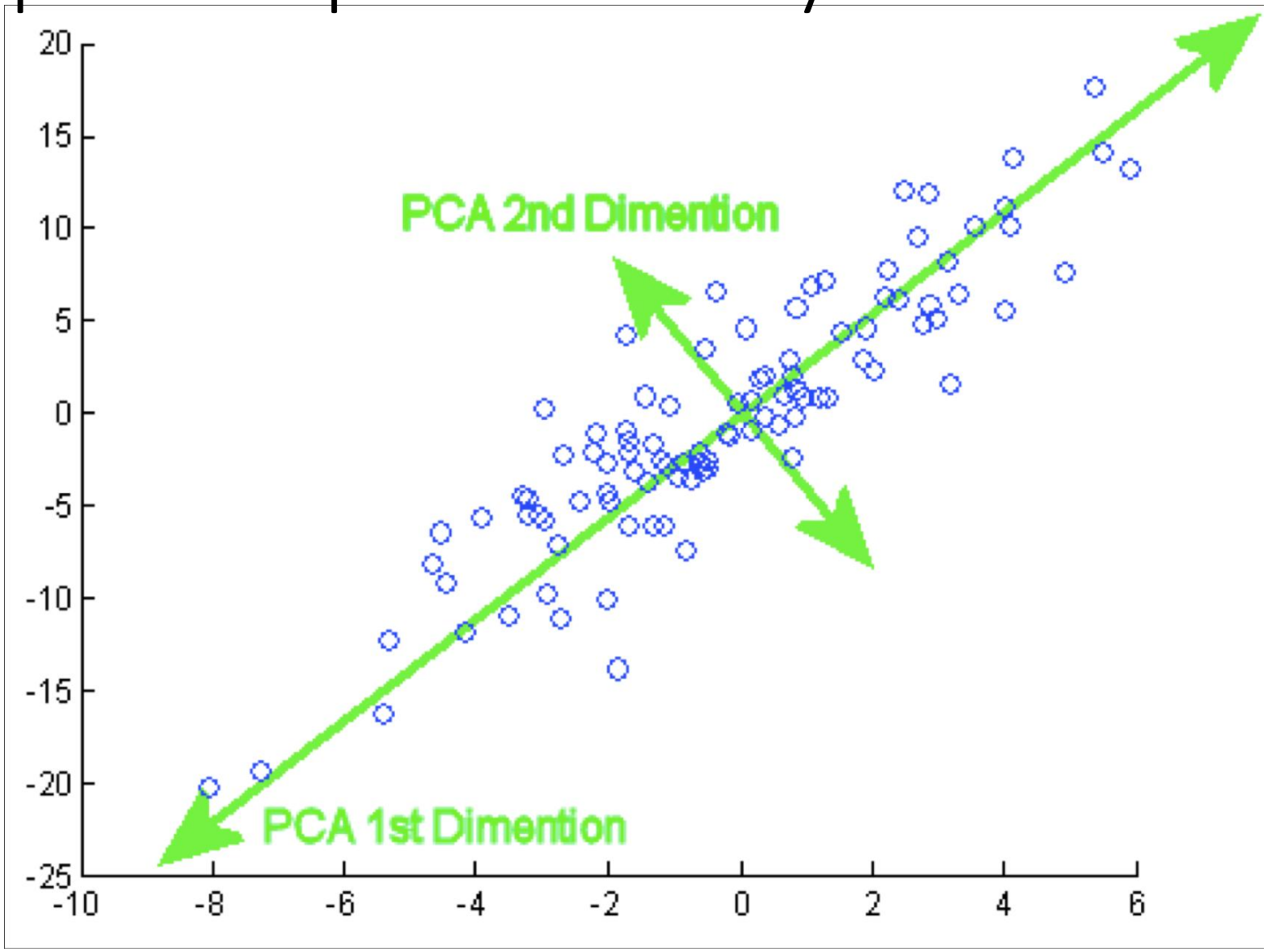
Outline

- Basic Linear Algebra
- Principal Component Analysis

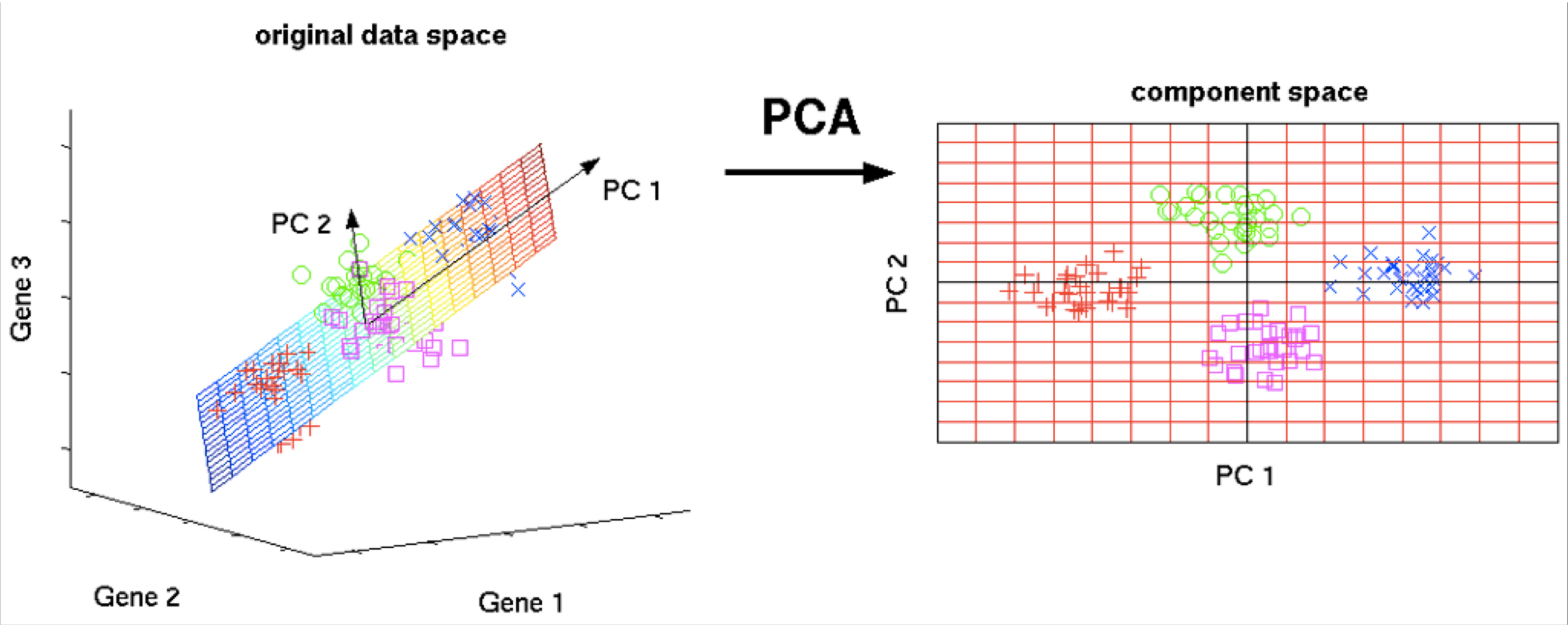
Principal Component Analysis (PCA)

- Reduce the number of (and transform the) features in a dataset
- Identify the (linear) transformation which reduces the features
- Transform a large set of variables into a smaller one that still contains most of the information in the larger set
- Reducing the number of variables of a dataset naturally comes at the expense of accuracy, but the trick in dimensionality reduction is to trade a little accuracy for simplicity.
- Why?
  - To make further computations more efficient
  - To visualize data in 2D or 3D
  - To remove noise from the data

Principal Component Analysis



Principal Component Analysis



Variance and Covariance

- Variance:
  - For a set of m observations $\{x^{(1)}, \cdots, x^{(m)}\}$ where $x^{(j)} \in \mathbb{R}$ & $\bar{x} = {}^{1}/_{m} \sum_{j=1}^{m} x^{(j)}$,

$$var(x) = \frac{\sum_{j=1}^{m} \left(x^{(j)} - \bar{x}\right)^2}{m}$$

- Covariance:
  - For two sets of points $\{x^{(1)}, \cdots, x^{(m)}\}$ & $\{y^{(1)}, \cdots, y^{(m)}\}$ where $x^{(j)} \in \mathbb{R}$ is corresponding to $y^{(j)} \in \mathbb{R}$,

$$cov(x, y) = \frac{\sum_{j=1}^{m} \left(x^{(j)} - \bar{x}\right)\left(y^{(j)} - \bar{y}\right)}{m}$$

  - If $cov(x, y) = 0$, the variables x and y are *independent*.

Variance and Covariance

- Covariance:

$$cov(x, y) = \frac{\sum_{j=1}^{m} \left(x^{(j)} - \bar{x}\right)\left(y^{(j)} - \bar{y}\right)}{m}$$

m observations of x and $y \in \mathbb{R}$

$$\begin{bmatrix} x^{(1)} \\ \vdots \\ x^{(j)} \\ \vdots \\ x^{(m)} \end{bmatrix} \quad \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(j)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

m observations of $\boldsymbol{x}^{(j)} \in \mathbb{R}^n$

$$\begin{bmatrix} x_1^{(1)} \\ \vdots \\ x_1^{(j)} \\ \vdots \\ x_1^{(m)} \end{bmatrix} \quad \begin{bmatrix} x_i^{(1)} \\ \vdots \\ x_i^{(j)} \\ \vdots \\ x_i^{(m)} \end{bmatrix} \quad \begin{bmatrix} x_n^{(1)} \\ \vdots \\ x_n^{(j)} \\ \vdots \\ x_n^{(m)} \end{bmatrix}$$

Covariance Matrix

- For m n-dimensional points $\{\boldsymbol{x}^{(1)}, \cdots, \boldsymbol{x}^{(m)}\}$ where $\boldsymbol{x}^{(j)} \in \mathbb{R}^n$ and $\bar{\boldsymbol{x}} = {}^1/_m \sum_{j=1}^{m} \boldsymbol{x}^{(j)} \in \mathbb{R}^n$
- Let $\boldsymbol{X} = \left[\boldsymbol{x}^{(1)}, \cdots, \boldsymbol{x}^{(m)}\right]^{\mathrm{T}} \in \mathbb{R}^{m \times n}$ & $\bar{\boldsymbol{X}} = [\bar{\boldsymbol{x}}, \cdots, \bar{\boldsymbol{x}}]^{\mathrm{T}} \in \mathbb{R}^{m \times n}$

$$Cov(\boldsymbol{X}) = \frac{1}{m}(\boldsymbol{X} - \bar{\boldsymbol{X}})^T (\boldsymbol{X} - \bar{\boldsymbol{X}})$$

- If we can assume $\bar{\boldsymbol{x}} = \boldsymbol{0}$. Then,

$$Cov(\boldsymbol{X}) = \frac{1}{m}\boldsymbol{X}^T\boldsymbol{X}$$

- If the n dimensions are uncorrelated, $Cov(\boldsymbol{X})$ will be a diagonal matrix.

Covariance Matrix

$$\boldsymbol{X^T X} = \begin{matrix} x_1 \\ x_2 \\ x_3 \end{matrix} \overbrace{\begin{bmatrix} a & b & c & d & \cdots \end{bmatrix}}^{m} \begin{bmatrix} a^T \\ b^T \\ c^T \\ d^T \\ \vdots \end{bmatrix}$$

$$= \begin{bmatrix} var(x_1) & cov(x_1, x_2) & cov(x_1, x_3) \\ cov(x_1, x_2) & var(x_2) & cov(x_2, x_3) \\ cov(x_1, x_3) & cov(x_2, x_3) & var(x_3) \end{bmatrix}$$

Covariance Matrix in Python

```
In [43]: import numpy as np

In [44]: X = np.array([ [1,2,1,2],
    ...:                [0,1,0,1],
    ...:                [1,0,1,0],
    ...:                [1,2,3,4] ])

In [45]: np.mean(X, axis=0)
Out[45]: array([0.75, 1.25, 1.25, 1.75])

In [46]: np.mean(X, axis=1)
Out[46]: array([1.50, 0.50, 0.50, 2.50])

In [47]: np.cov(X)
Out[47]:
array([[0.33, 0.33, -0.33, 0.33],
       [0.33, 0.33, -0.33, 0.33],
       [-0.33, -0.33, 0.33, -0.33],
       [0.33, 0.33, -0.33, 1.67]])
```

Correlation = "normalized covariance" (useful in the future)

- Correlation between two random variable is their covariance normalized by their standard deviations

$$corr(x, y) = \frac{cov(x, y)}{std(x)\ std(y)}$$

- Correlation values are in [-1,+1]

- It is also called "Pearson Correlation"

- The relation between covariance and correlation is analogous to the relation between dot-product and cosine similarity

Principal Component Analysis (PCA)

- Objective: Transforming (also called projecting) the original coordinate system (or space) to another one ($\boldsymbol{X} \rightarrow \boldsymbol{Y}$) so that the different dimensions in the new coordinate system are linearly uncorrelated, i.e., $Cov(\boldsymbol{Y})$ is a diagonal matrix.
- For the new coordinate system, the new set of dimensions should be organized so that the one with the largest variance should be the first one (first principal component), followed by the second largest one (second principal component), and so on.
- This can be achieved by eigendecomposition!

Principal Component Analysis (PCA)

- Apply eigendecomposition to $Cov(\boldsymbol{X})$ (absorb $1/m$ into $\boldsymbol{X}^T\boldsymbol{X}$) and $Cov(\boldsymbol{X})$ is symmetric.

$$\boldsymbol{X}^T\boldsymbol{X} = \boldsymbol{Q}\, diag(\boldsymbol{\lambda})\boldsymbol{Q}^T$$

$$\boldsymbol{Q}^T\boldsymbol{X}^T\boldsymbol{X}\,\boldsymbol{Q} = \boldsymbol{Q}^T\boldsymbol{Q}\, diag(\boldsymbol{\lambda})\boldsymbol{Q}^T\boldsymbol{Q}$$
$$(\boldsymbol{X}\boldsymbol{Q})^T(\boldsymbol{X}\boldsymbol{Q}) = diag(\boldsymbol{\lambda})$$

- The transformation becomes: $\boldsymbol{Y} = \boldsymbol{X}\,\boldsymbol{Q}$
- Columns of $\boldsymbol{Q}$: Principle component vectors (eigenvectors)
- $diag(\boldsymbol{\lambda})$: Variances (eigenvalues) of $\boldsymbol{Y}$ in new coordinate system.

Principal Component Analysis

$$\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{Q} = \begin{bmatrix} a^T \\ b^T \\ c^T \\ d^T \\ \vdots \end{bmatrix} \begin{bmatrix} q_1 & q_2 & \cdots \end{bmatrix}$$

Dimensionality Reduction

- Keep only the first k principle component vectors

$$\boldsymbol{X}^T\boldsymbol{X} \approx \boldsymbol{Q}_k \, diag(\boldsymbol{\lambda}_k) \, \boldsymbol{Q}_k^T$$

- The transformation becomes: $\boldsymbol{Y} = \boldsymbol{X}\,\boldsymbol{Q}_k \in \mathbb{R}^{\boldsymbol{m}\times\boldsymbol{k}}$
- Columns of $\boldsymbol{Q}_k$: first k principle component vectors (eigenvectors)
- $diag(\boldsymbol{\lambda}_k)$: first k eigenvalues

PCA in practice

- You usually have a data-matrix A such that:
  - its rows are records
  - its columns are (numerical) features, i.e. distinct variables of the records
  - so any record has multiple features (its columns)
- Can we reduce the number of features useful for a particular task?
  - What task?
    - Visualization (reduce to 2D), Learning (classification or regression), etc…
  - How to reduce?
    - Center any column (by subtracting columns means), say the resulting matrix is X
    - Compute $X^{\mathsf{T}}X$ … this is the covariance matrix of the features (its clearly symmetric)
    - Compute the eigendecomposition $X^{\mathsf{T}}X = Q\,\mathrm{diag}(\lambda)\,Q^{\mathsf{T}}$ with the eigenvalues in $\mathrm{diag}(\lambda)$ in decreasing order
    - Decide the new dimensionality k on the basis of your need
    - Select the first k columns of Q and call it Q_k
    - … thus you also selected the first k rows of Q^{T} and the upper-left k-by-k block of $\mathrm{diag}(\lambda)$
    - The new reduced data-matrix is $B = A\,Q_k$

Explained Variance Ratio

- The original data-matrix *A* has *n* features/columns
- The reduced data-matrix *B* has *k* features/columns
- *B* was obtained by selecting the largest *k* eigenvalues, among a total of *n* eigenvalues
- Each eigenvalue amounts for the variance along the *k* axis of the eigenbase (the corresponding eigenvector)
- Explained Variance Ratio (of the reduction) is given by

$$\sum_{i=1}^{k} \frac{\lambda_i}{\sum_{j=1}^{n} \lambda_j}$$

Each single term of the outer sum is the explained variance of *i*-th principal component (or eigenvector)

- Intuitively, the explained variance ratio is the percentage of preserved information!

How to decide the number of components in PCA?

- In general, it is a trade-off between retaining as much information as possible while reducing the complexity of the data.
- However, it depends on the specific problem you're trying to solve. Some possible use cases as follows.
  - If the goal is visualization (e.g. draw a scatter plot of the records), choose 2 or 3 components (otherwise you cannot plot the records).
  - If the goal is to compress the data because the next elaboration steps (e.g. classification or clustering) are computationally expensive, then the number of components should be chosen based on the desired level of tolerance.
  - If the goal is to improve the effectiveness/accuracy of the next processing step (e.g. classification or clustering), and computational time is not an issue, then try different values and choose the best for underlying task (if classification or regression use the well-known cross-validation methodology).
- Two commonly used heuristic approaches that work in many cases are as follows.
  - Keep components that explain a significant portion of the total variance in the data (e.g., 90%, 95% or 99% of the variance are reasonable choices).
  - "Elbow method": run PCA with a number of components equal to the entire dimensionality; draw a "scree plot" (a simple plot with the principal components in the x-axis, and the explained variance in the y-axis); visually inspect the plot and select the point at which the proportion of variance explained by each subsequent principal component drops off; rerun PCA with the selected number of components.

Standardization of the data-matrix

- In practical situations, the preprocessing is not limited to subtract means
- Often, standardization of all the variables is performed
- Standardization:
  - Any column of a data-matrix represent the values of a variable
  - Subtract the mean of the column
  - Divide by the standard deviation of the column
  - The colum have now mean=0 and stdev=1
  - The new values are said *z*-scores, which indicates how many standard-deviations away is a value with resect to the mean

PCA in Python

See a demo of PCA on the Iris dataset in the Python file `pca_example.py`

In order to run it, you need to install Python modules as follows:

```
pip install numpy sklearn matplotlib
```

PCA interpretability

- Principal components are very useful but they may miss interpretability
  - though we know that any features obtained through PCA is a linear combination of existing features
  - the coefficients in the linear combination are called "loadings" and their absolute values measure how important a feature is for a P.C.

- What to do if you require the new features need to be more interpretable?
  - Use features selection methods!
  - They select a subset of features, thus number of features is reduced, but the selected ones retain their original interpretation
  - The explained variance is worse than PCA
  - Which features selection method?
    - Python's Scikit Learn library has implemented Recursive Features Elimination (`sklearn.feature_selection.RFE`)
    - Use an Evolutionary Algorithm with a binary representation and a purposely defined objective function for the task at hand