Principal Component Analysis

Daifeng Wang

`daifeng.wang@wisc.edu`

University of Wisconsin, Madison

Based on slides from Xiaojin Zhu, Yingyu Liang and
UCL Linear Algebra & Matrices, MfD 2009, modified
by Daifeng Wang

Outline

- Basic review on linear algebra
- Introduction to dimensionality reduction
- Principal component analysis: formulation and computation
- Applications

REVIEW ON
LINEAR ALGEBRA

Vector

- Not a physics vector (magnitude, direction)
- Column of numbers e.g. intensity of same voxel at different time points

$$\begin{bmatrix} x1 \\ x2 \\ x3 \end{bmatrix}$$

Matrices

- Rectangular display of vectors in rows and columns
- Can inform about the same vector intensity at different times or different voxels at the same time
- Vector is just a n x 1 matrix

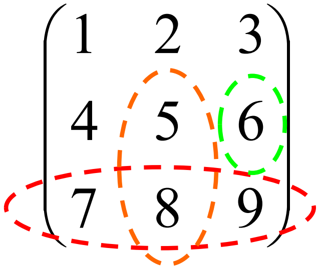
$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 5 & 4 & 1 \\ 6 & 7 & 4 \end{bmatrix} \qquad \mathbf{C} = \begin{bmatrix} 1 & 4 \\ 2 & 7 \\ 3 & 8 \end{bmatrix} \qquad \mathbf{D} = \begin{bmatrix} d_{11} & d_{12} & d_{13} \\ d_{21} & d_{22} & d_{23} \\ d_{31} & d_{32} & d_{33} \end{bmatrix}$$

Square (3 x 3) Rectangular (3 x 2) d_{ij} : i^{th} row, j^{th} column

Defined as rows x columns (R x C)

Matrices in Python

- X=[[1,2,3],[4,5,6],[7,8,9]]
- Index from 0 to nrow/ncol-1
- :=all row or column



$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Subscripting – each element of a matrix can be addressed with a pair of numbers; [row first. column second]

e.g. **X[1, 2]** = 6

X[2, :] = $\begin{pmatrix} \mathbf{7} & \mathbf{8} & \mathbf{9} \end{pmatrix}$

X[1:2, 1] = $\begin{pmatrix} \mathbf{5} \\ \mathbf{8} \end{pmatrix}$

Transposition

$$\mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix} \qquad \mathbf{b}^T = \begin{bmatrix} 1 & 1 & 2 \end{bmatrix} \qquad \mathbf{d} = \begin{bmatrix} 3 & 4 & 9 \end{bmatrix} \qquad \mathbf{d}^T = \begin{bmatrix} 3 \\ 4 \\ 9 \end{bmatrix}$$

column → row row → column

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 5 & 4 & 1 \\ 6 & 7 & 4 \end{bmatrix} \qquad \mathbf{A}^T = \begin{bmatrix} 1 & 5 & 6 \\ 2 & 4 & 7 \\ 3 & 1 & 4 \end{bmatrix}$$

Scalar multiplication

- Scalar x matrix = scalar multiplication

$$\lambda \begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix} = \begin{pmatrix} \lambda a & \lambda b & \lambda c \\ \lambda d & \lambda e & \lambda f \end{pmatrix}$$

Matrix Calculations

Addition

- Commutative: **A+B=B+A**
- Associative: **(A+B)+C=A+(B+C)**

$$\mathbf{A}+\mathbf{B}=\begin{bmatrix}2 & 4\\ 2 & 5\end{bmatrix}+\begin{bmatrix}1 & 0\\ 3 & 1\end{bmatrix}=\begin{bmatrix}2+1 & 4+0\\ 2+3 & 5+1\end{bmatrix}=\begin{bmatrix}3 & 4\\ 5 & 6\end{bmatrix}$$

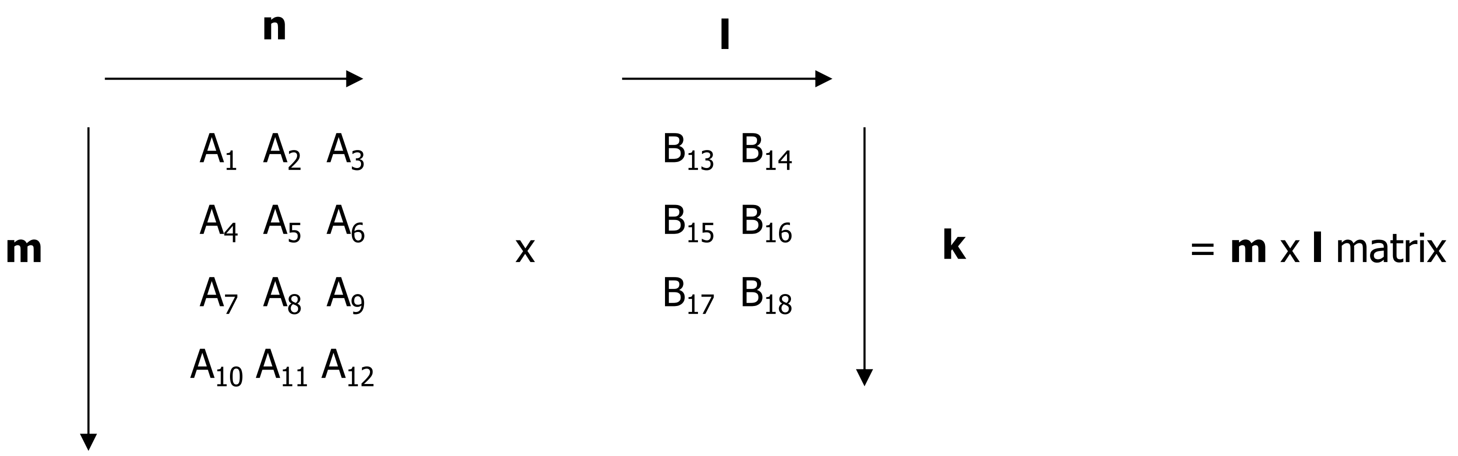
Subtraction

- By adding a negative matrix

$$\mathbf{A}-\mathbf{B}=\begin{bmatrix}2 & 4\\ 5 & 3\end{bmatrix}-\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}=\begin{bmatrix}2 & 4\\ 5 & 3\end{bmatrix}+\begin{bmatrix}-1 & -2\\ -3 & -4\end{bmatrix}=\begin{bmatrix}1 & 2\\ 2 & -1\end{bmatrix}$$

Matrix Multiplication

*"When **A** is a mxn matrix & **B** is a kxl matrix, **AB** is only possible if n=k. The result will be an mxl matrix"*



$$\mathbf{m}\ \overset{\mathbf{n}}{\begin{matrix} A_1 & A_2 & A_3 \\ A_4 & A_5 & A_6 \\ A_7 & A_8 & A_9 \\ A_{10} & A_{11} & A_{12} \end{matrix}} \quad \times \quad \overset{\mathbf{l}}{\begin{matrix} B_{13} & B_{14} \\ B_{15} & B_{16} \\ B_{17} & B_{18} \end{matrix}}\ \mathbf{k} \qquad = \mathbf{m} \times \mathbf{l} \text{ matrix}$$

Number of columns in A = Number of rows in B

Matrix multiplication

- Multiplication method:

Sum over product of respective rows and columns

$$\underset{\mathbf{A}}{\begin{pmatrix}1 & 0\\ 2 & 3\end{pmatrix}} \times \underset{\mathbf{B}}{\begin{pmatrix}2 & 1\\ 3 & 1\end{pmatrix}} = \begin{pmatrix}\mathbf{c}_{11} & \mathbf{c}_{12}\\ \mathbf{c}_{21} & \mathbf{c}_{22}\end{pmatrix}$$

Define output matrix

$$= \begin{bmatrix}(1\times 2)+(0\times 3) & (1\times 1)+(0\times 1)\\ (2\times 2)+(3\times 3) & (2\times 1)+(3\times 1)\end{bmatrix}$$

$$= \begin{pmatrix}\mathbf{2} & \mathbf{1}\\ \mathbf{13} & \mathbf{5}\end{pmatrix}$$

$$\boxed{C_{i,j} = \sum_{k=1}^{n} A_{i,k} B_{k,j}}$$

Matrix multiplication

- Matrix multiplication is NOT commutative
- $AB \neq BA$
- Matrix multiplication IS associative
- $A(BC)=(AB)C$
- Matrix multiplication IS distributive
- $A(B+C)=AB+AC$
- $(A+B)C=AC+BC$

Identity matrix

Is there a matrix which plays a similar role as the number 1 in number multiplication?

Consider the nxn matrix:

$$I_n = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \cdot & & & & \cdot \\ \cdot & & & & \cdot \\ \cdot & & & & \cdot \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}.$$

For any nxn matrix $\boldsymbol{A}$, we have $\boldsymbol{A}\,\boldsymbol{I}_n = \boldsymbol{I}_n\,\boldsymbol{A} = \boldsymbol{A}$

For any nxm matrix $\boldsymbol{A}$, we have $\boldsymbol{I}_n\,\boldsymbol{A} = \boldsymbol{A}$, and $\boldsymbol{A}\,\boldsymbol{I}_m = \boldsymbol{A}$ (so 2 possible matrices)

Matrix inverse

- **Definition.** A matrix *A* is called **nonsingular** or **invertible** if there exists a matrix *B* such that:

$$A\,B = B\,A = I_n$$

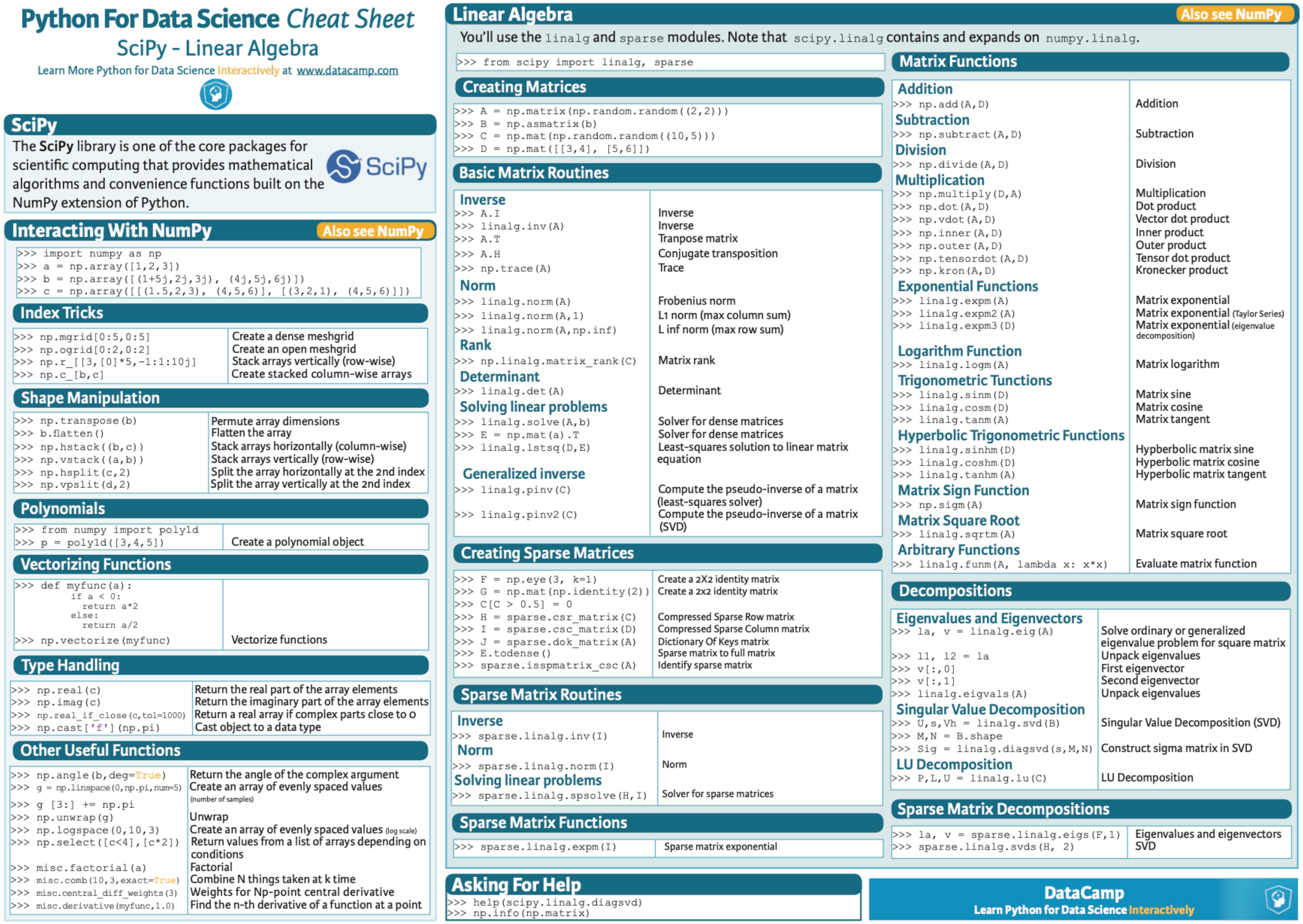
$$\begin{bmatrix} 1 & 1 \\ -1 & 2 \end{bmatrix} \text{X} \begin{bmatrix} \frac{2}{3} & \frac{-1}{3} \\ \frac{1}{3} & \frac{1}{3} \end{bmatrix} = \begin{bmatrix} \frac{2}{3}+\frac{1}{3} & \frac{-1}{3}+\frac{1}{3} \\ \frac{-2}{3}+\frac{2}{3} & \frac{1}{3}+\frac{2}{3} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

- **Notation.** A common notation for the inverse of a matrix *A* is A^{-1}. So:

$$A\,A^{-1} = A^{-1}\,A = I_n\,.$$

- The inverse matrix is unique when it exists. So if ***A*** is invertible, then $\boldsymbol{A}^{-1}$ is also invertible and then $(A^T)^{-1} = (A^{-1})^T$

SciPy - Linear Algebra



Python For Data Science *Cheat Sheet*

SciPy - Linear Algebra

Learn More Python for Data Science Interactively at www.datacamp.com

SciPy

The **SciPy** library is one of the core packages for scientific computing that provides mathematical algorithms and convenience functions built on the NumPy extension of Python.

Interacting With NumPy

Also see NumPy

```
>>> import numpy as np
>>> a = np.array([1,2,3])
>>> b = np.array([(1+5j,2j,3j), (4j,5j,6j)])
>>> c = np.array([[(1.5,2,3), (4,5,6)], [(3,2,1), (4,5,6)]])
```

Index Tricks

`>>> np.mgrid[0:5,0:5]`	Create a dense meshgrid
`>>> np.ogrid[0:2,0:2]`	Create an open meshgrid
`>>> np.r_[[3,[0]*5,-1:1:10j]`	Stack arrays vertically (row-wise)
`>>> np.c_[b,c]`	Create stacked column-wise arrays

Shape Manipulation

`>>> np.transpose(b)`	Permute array dimensions
`>>> b.flatten()`	Flatten the array
`>>> np.hstack((b,c))`	Stack arrays horizontally (column-wise)
`>>> np.vstack((a,b))`	Stack arrays vertically (row-wise)
`>>> np.hsplit(c,2)`	Split the array horizontally at the 2nd index
`>>> np.vpslit(d,2)`	Split the array vertically at the 2nd index

Polynomials

`>>> from numpy import poly1d`	
`>>> p = poly1d([3,4,5])`	Create a polynomial object

Vectorizing Functions

```
>>> def myfunc(a):
        if a < 0:
          return a*2
        else:
          return a/2
```

`>>> np.vectorize(myfunc)`	Vectorize functions

Type Handling

`>>> np.real(c)`	Return the real part of the array elements
`>>> np.imag(c)`	Return the imaginary part of the array elements
`>>> np.real_if_close(c,tol=1000)`	Return a real array if complex parts close to 0
`>>> np.cast['f'](np.pi)`	Cast object to a data type

Other Useful Functions

`>>> np.angle(b,deg=True)`	Return the angle of the complex argument
`>>> g = np.linspace(0,np.pi,num=5)`	Create an array of evenly spaced values (number of samples)
`>>> g [3:] += np.pi`	
`>>> np.unwrap(g)`	Unwrap
`>>> np.logspace(0,10,3)`	Create an array of evenly spaced values (log scale)
`>>> np.select([c<4],[c*2])`	Return values from a list of arrays depending on conditions
`>>> misc.factorial(a)`	Factorial
`>>> misc.comb(10,3,exact=True)`	Combine N things taken at k time
`>>> misc.central_diff_weights(3)`	Weights for Np-point central derivative
`>>> misc.derivative(myfunc,1.0)`	Find the n-th derivative of a function at a point

Linear Algebra

Also see NumPy

You'll use the `linalg` and `sparse` modules. Note that `scipy.linalg` contains and expands on `numpy.linalg`.

```
>>> from scipy import linalg, sparse
```

Creating Matrices

```
>>> A = np.matrix(np.random.random((2,2)))
>>> B = np.asmatrix(b)
>>> C = np.mat(np.random.random((10,5)))
>>> D = np.mat([[3,4], [5,6]])
```

Basic Matrix Routines

Inverse	
`>>> A.I`	Inverse
`>>> linalg.inv(A)`	Inverse
`>>> A.T`	Tranpose matrix
`>>> A.H`	Conjugate transposition
`>>> np.trace(A)`	Trace
Norm	
`>>> linalg.norm(A)`	Frobenius norm
`>>> linalg.norm(A,1)`	L1 norm (max column sum)
`>>> linalg.norm(A,np.inf)`	L inf norm (max row sum)
Rank	
`>>> np.linalg.matrix_rank(C)`	Matrix rank
Determinant	
`>>> linalg.det(A)`	Determinant
Solving linear problems	
`>>> linalg.solve(A,b)`	Solver for dense matrices
`>>> E = np.mat(a).T`	Solver for dense matrices
`>>> linalg.lstsq(D,E)`	Least-squares solution to linear matrix equation
Generalized inverse	
`>>> linalg.pinv(C)`	Compute the pseudo-inverse of a matrix (least-squares solver)
`>>> linalg.pinv2(C)`	Compute the pseudo-inverse of a matrix (SVD)

Creating Sparse Matrices

`>>> F = np.eye(3, k=1)`	Create a 2X2 identity matrix
`>>> G = np.mat(np.identity(2))`	Create a 2x2 identity matrix
`>>> C[C > 0.5] = 0`	
`>>> H = sparse.csr_matrix(C)`	Compressed Sparse Row matrix
`>>> I = sparse.csc_matrix(D)`	Compressed Sparse Column matrix
`>>> J = sparse.dok_matrix(A)`	Dictionary Of Keys matrix
`>>> E.todense()`	Sparse matrix to full matrix
`>>> sparse.isspmatrix_csc(A)`	Identify sparse matrix

Sparse Matrix Routines

Inverse	
`>>> sparse.linalg.inv(I)`	Inverse
Norm	
`>>> sparse.linalg.norm(I)`	Norm
Solving linear problems	
`>>> sparse.linalg.spsolve(H,I)`	Solver for sparse matrices

Sparse Matrix Functions

`>>> sparse.linalg.expm(I)`	Sparse matrix exponential

Matrix Functions

Addition	
`>>> np.add(A,D)`	Addition
Subtraction	
`>>> np.subtract(A,D)`	Subtraction
Division	
`>>> np.divide(A,D)`	Division
Multiplication	
`>>> np.multiply(D,A)`	Multiplication
`>>> np.dot(A,D)`	Dot product
`>>> np.vdot(A,D)`	Vector dot product
`>>> np.inner(A,D)`	Inner product
`>>> np.outer(A,D)`	Outer product
`>>> np.tensordot(A,D)`	Tensor dot product
`>>> np.kron(A,D)`	Kronecker product
Exponential Functions	
`>>> linalg.expm(A)`	Matrix exponential
`>>> linalg.expm2(A)`	Matrix exponential (Taylor Series)
`>>> linalg.expm3(D)`	Matrix exponential (eigenvalue decomposition)
Logarithm Function	
`>>> linalg.logm(A)`	Matrix logarithm
Trigonometric Tunctions	
`>>> linalg.sinm(D)`	Matrix sine
`>>> linalg.cosm(D)`	Matrix cosine
`>>> linalg.tanm(A)`	Matrix tangent
Hyperbolic Trigonometric Functions	
`>>> linalg.sinhm(D)`	Hypberbolic matrix sine
`>>> linalg.coshm(D)`	Hyperbolic matrix cosine
`>>> linalg.tanhm(A)`	Hyperbolic matrix tangent
Matrix Sign Function	
`>>> np.sigm(A)`	Matrix sign function
Matrix Square Root	
`>>> linalg.sqrtm(A)`	Matrix square root
Arbitrary Functions	
`>>> linalg.funm(A, lambda x: x*x)`	Evaluate matrix function

Decompositions

Eigenvalues and Eigenvectors	
`>>> la, v = linalg.eig(A)`	Solve ordinary or generalized eigenvalue problem for square matrix
`>>> l1, l2 = la`	Unpack eigenvalues
`>>> v[:,0]`	First eigenvector
`>>> v[:,1]`	Second eigenvector
`>>> linalg.eigvals(A)`	Unpack eigenvalues
Singular Value Decomposition	
`>>> U,s,Vh = linalg.svd(B)`	Singular Value Decomposition (SVD)
`>>> M,N = B.shape`	
`>>> Sig = linalg.diagsvd(s,M,N)`	Construct sigma matrix in SVD
LU Decomposition	
`>>> P,L,U = linalg.lu(C)`	LU Decomposition

Sparse Matrix Decompositions

`>>> la, v = sparse.linalg.eigs(F,1)`	Eigenvalues and eigenvectors
`>>> sparse.linalg.svds(H, 2)`	SVD

Asking For Help

```
>>> help(scipy.linalg.diagsvd)
>>> np.info(np.matrix)
```

DataCamp
Learn Python for Data Science Interactively

INTRODUCTION TO DIMENSIONALITY REDUCTION

Big & High-Dimensional Data

- High-Dimensions = Lot of Features

Document classification

Features per document =

thousands of words/unigrams

millions of bigrams, contextual information

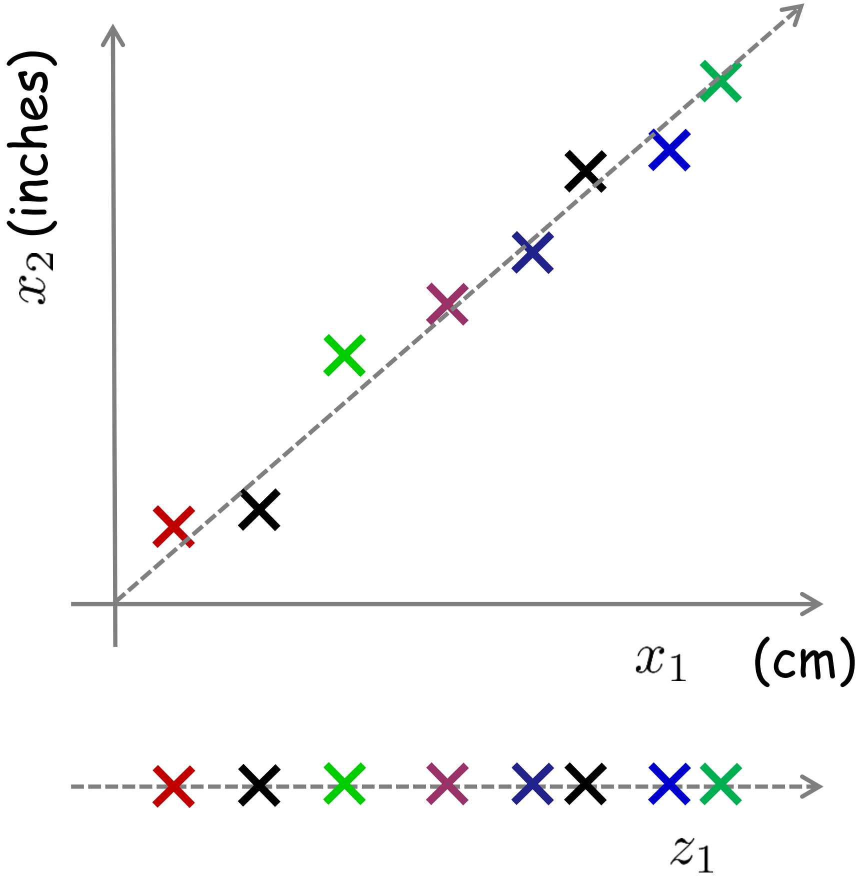
Surveys - Netflix

480189 users x 17770 movies

	movie 1	movie 2	movie 3	movie 4	movie 5	movie 6
Tom	5	?	?	1	3	?
George	?	?	3	1	2	5
Susan	4	3	1	?	5	1
Beth	4	3	?	2	4	2

- Big & High-Dimensional Data.
- Useful to learn lower dimensional representations of the data.
  - Given data points in D dimensions
  - Convert them to data points in d<D dimensions
  - With minimal loss of information

Data Compression



Reduce data from 2D to 1D

(x_1=cm, x_2=inch) -> z_1

$$x^{(1)} \rightarrow z^{(1)}$$

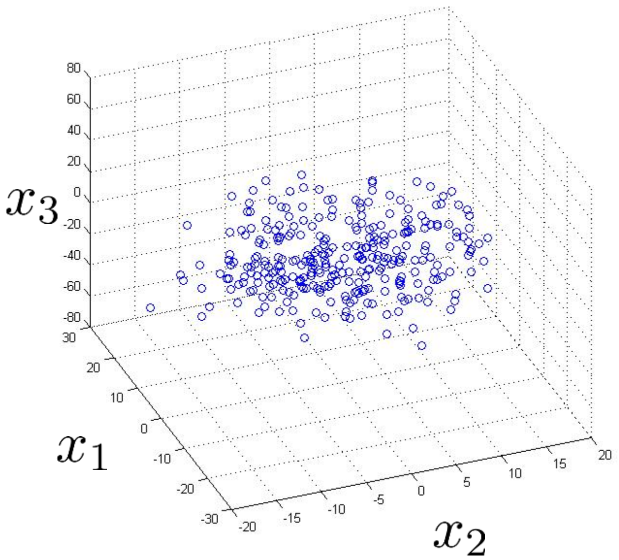
$$x^{(2)} \rightarrow z^{(2)}$$

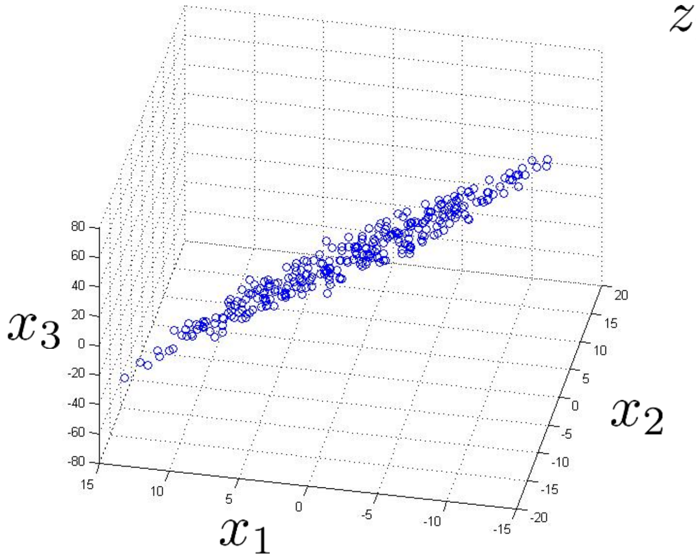
$$\vdots$$

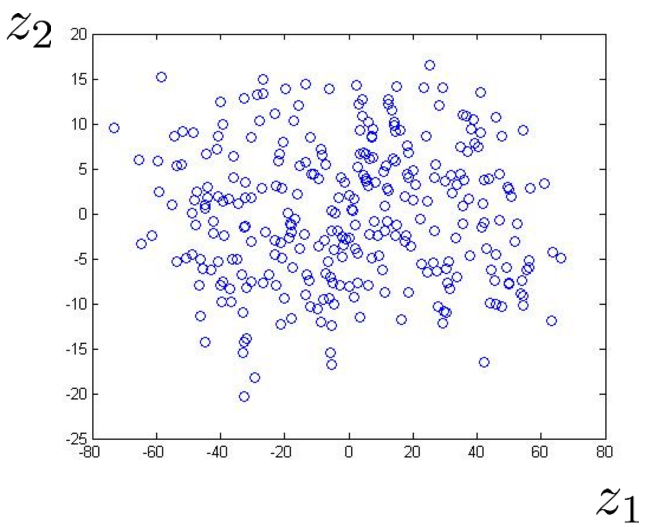
$$x^{(m)} \rightarrow z^{(m)}$$

Data Compression

Reduce data from 3D to 2D

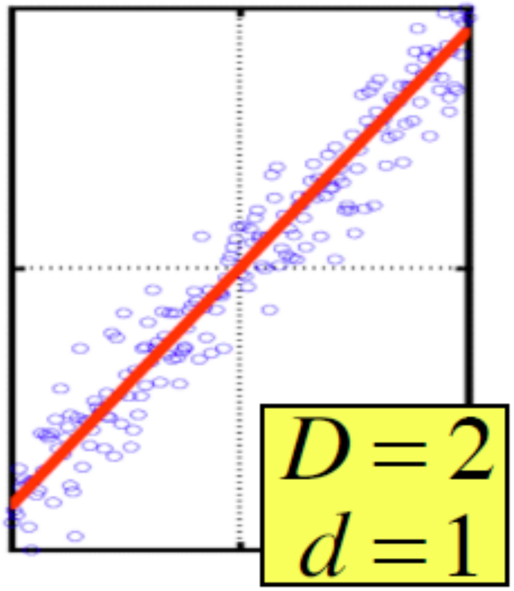


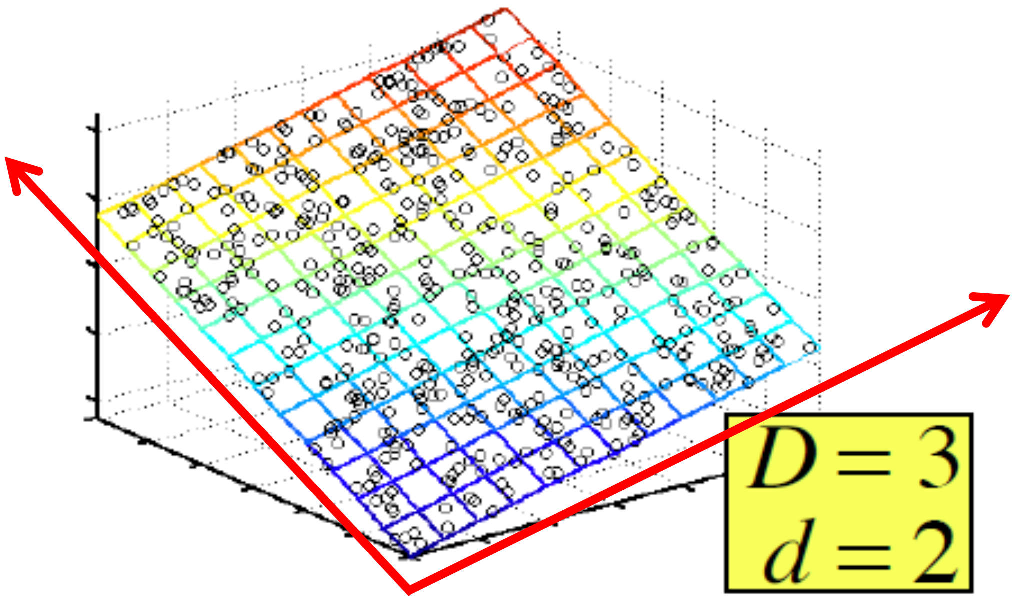




PRINCIPAL COMPONENT ANALYSIS (PCA)

Principal Component Analysis (PCA)

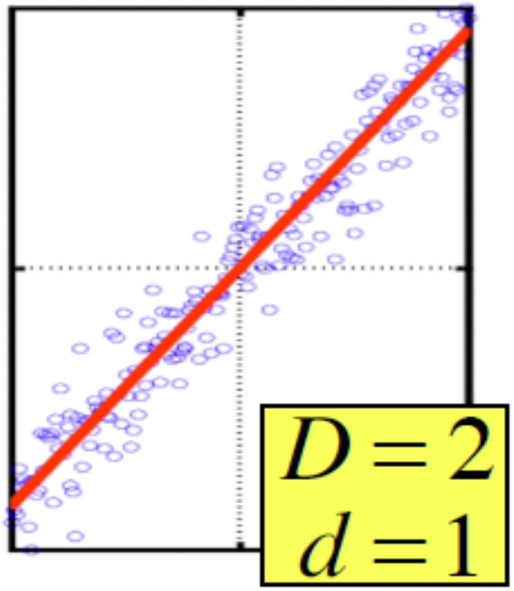


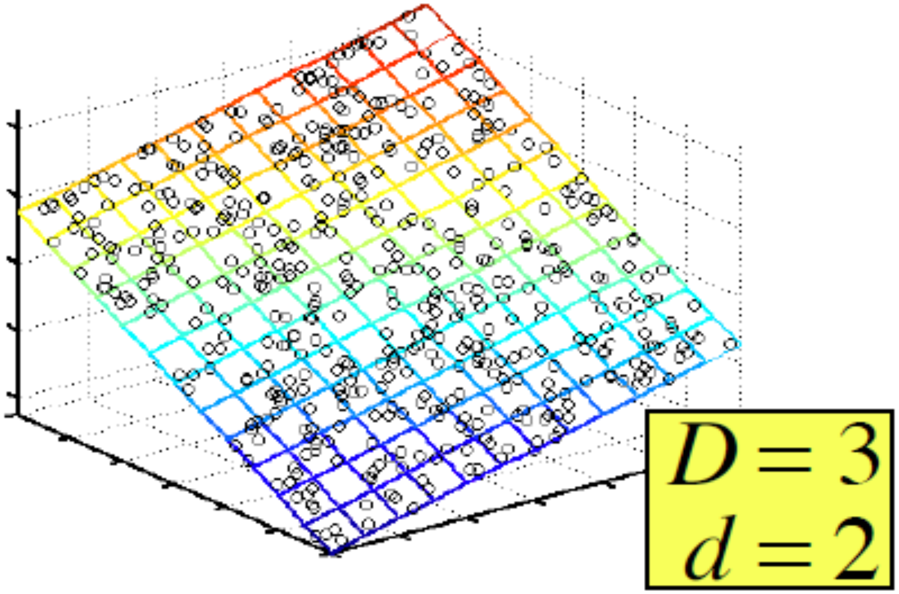


In case where data lies on or near a low d-dimensional linear subspace, axes of this subspace are an effective representation of the data.

Identifying the axes is known as Principal Components Analysis, and can be obtained by using classic matrix computation tools (Eigen or Singular Value Decomposition).

PCA formulation





- Reduce from 2-dimension to 1-dimension: Find a direction (a red vector $v_1 \in R^D$) onto which to project the data so as to minimize the projection error.

- Reduce from D-dimension to d-dimension: Find d vectors $v_i \in R^D, i = 1,2,..,d$ onto which to project the data, so as to minimize the projection error.
- v_i is called a principal component (PC)

Principal Component Analysis

Input: *N* data points (D-dim vectors)

$$\mathbf{x} \in \mathbb{R}^D: \ \mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$$

Output:

- *d* principal components (PCs)

$$v_j \in R^D, j = 1,2,..,d$$

$$\text{s.t., Euclidean norm } ||v_j||_2 = \left(\sum_{k=1}^{D} v_j^2[k]\right)^{\frac{1}{2}} = 1$$

- For each x_i, it's project coordinates on $\{v_j\}$:

$$w_{i,j} = v_j^T * x_i, j = 1,2,\ldots,d$$

- Now x_i, a D-dim vector can be represented by a d-dim vector (d<D)

⇨ $[w_{i,1}, w_{i,2}, \ldots, w_{i,d}]$

Learning Representations

PCA, Kernel PCA, ICA, CCA: Powerful unsupervised learning techniques for extracting hidden (potentially lower dimensional) structure from high dimensional datasets.

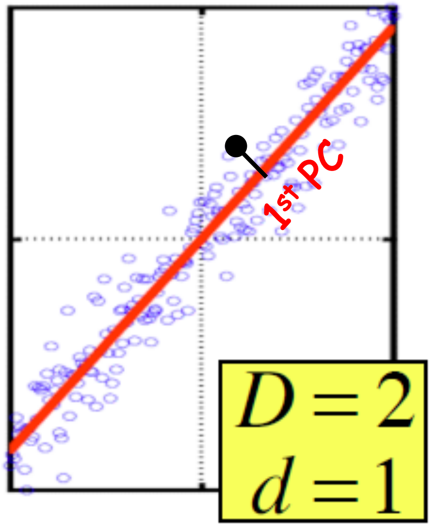
Useful for:

- Visualization
- More efficient use of resources
  (e.g., time, memory, communication)
- Statistical: fewer dimensions → better generalization
- Noise removal (improving data quality)
- Further processing by machine learning algorithms

PCA COMPUTATION

Principal Component Analysis (PCA)

Principal Components (PC) are orthogonal directions that capture most of the variance in the data.

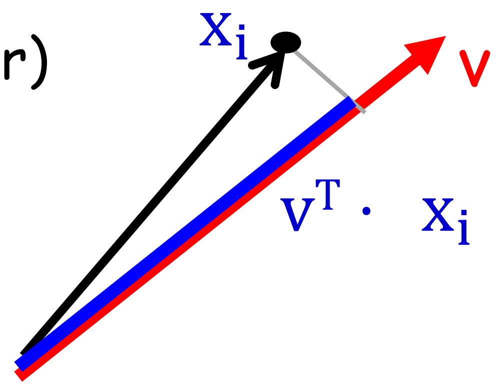


- First PC – direction of greatest variability in data.
- Projection of data points along first PC discriminates data most along any one direction (pts are the most spread out when we project the data on that direction compared to any other directions).

Quick reminder:

$||v||=1$, Point x_i (D-dimensional vector)

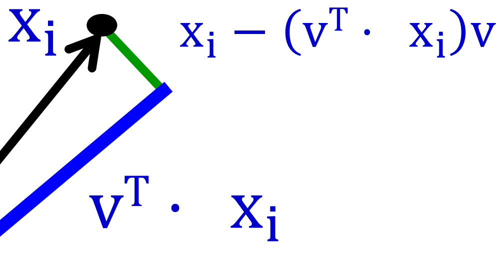
Projection of x_i onto v is $v^T \cdot x_i$

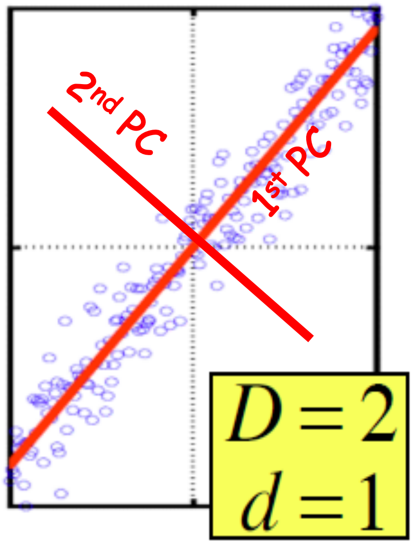


Principal Component Analysis (PCA)

Principal Components (PC) are orthogonal directions that capture most of the variance in the data.

- 1st PC – direction of greatest variability in data.





- 2nd PC – Next orthogonal (uncorrelated) direction of greatest variability

  (remove all variability in first direction, then find next direction of greatest variability)

- And so on …

Eigenvector and Eigenvalue

$$\mathbf{Ax} = \lambda \mathbf{x}$$

A: Square Matrix

x: Eigenvector

λ : Eigenvalue

Example

$$\text{Show } x = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \text{ is an eigenvector for } A = \begin{bmatrix} 2 & -4 \\ 3 & -6 \end{bmatrix}$$

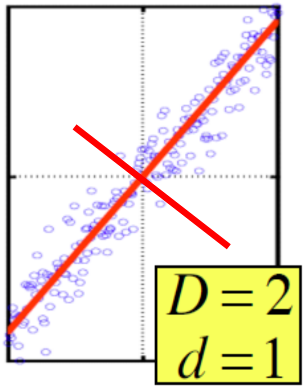
$$\text{Solution}: Ax = \begin{bmatrix} 2 & -4 \\ 3 & -6 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\text{But for } \lambda = 0, \quad \lambda x = 0 \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\text{Thus, } x \text{ is an eigenvector of } A, \text{ and } \lambda = 0 \text{ is an eigenvalue.}$$

- ***The zero vector can not be an eigenvector***
- ***The value zero can be eigenvalue***

Principal Component Analysis (PCA)



Let $v_1, v_2, \dots, v_d$ denote the d principal components.

$v^T_i \cdot v_j = 0, i \neq j$ and $v^T_i \cdot v_i = 1, \; i = j$

Assume data is centered (we extracted the sample mean).

Let $X = [x_1, x_2, \dots, x_n]$ (columns are the datapoints)

Find vector that maximizes sample variance of projected data

$$\sum_{i=1}^{n} (\mathbf{v}^T \mathbf{x}_i)^2 = \mathbf{v}^T \mathbf{X}\mathbf{X}^T \mathbf{v}$$

$$\max_{\mathbf{v}} \; \mathbf{v}^T \mathbf{X}\mathbf{X}^T \mathbf{v} \quad \text{s.t.} \quad \mathbf{v}^T \mathbf{v} = 1$$

Lagrangian: $\max_{\mathbf{v}} \mathbf{v}^T \mathbf{X}\mathbf{X}^T \mathbf{v} - \lambda \mathbf{v}^T \mathbf{v}$

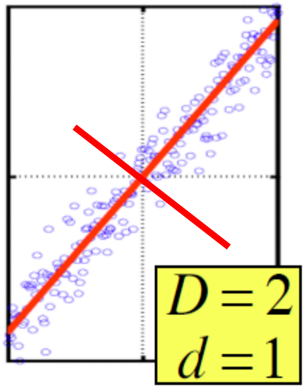
Wrap constraints into the objective function

$$\partial / \partial \mathbf{v} = 0 \qquad (\mathbf{X}\mathbf{X}^T - \lambda \mathbf{I})\mathbf{v} = 0 \qquad \Rightarrow \boxed{(\mathbf{X}\mathbf{X}^T)\mathbf{v} = \lambda \mathbf{v}}$$

Principal Component Analysis (PCA)

$(X X^T)v = \lambda v$, so v (the first PC) is the eigenvector of sample correlation/covariance matrix $X X^T$

Sample variance of projection $v^T X X^T v = \lambda v^T v = \lambda$



Thus, the eigenvalue λ denotes the amount of variability captured along that dimension (aka amount of energy along that dimension).

Eigenvalues $\lambda_1 \geq \lambda_2 \geq \lambda_3 \geq \cdots$

- The 1st PC v_1 is the eigenvector of the sample covariance matrix $X X^T$ associated with the largest eigenvalue
- The 2nd PC v_2 is the eigenvector of the sample covariance matrix $X X^T$ associated with the second largest eigenvalue
- And so on …

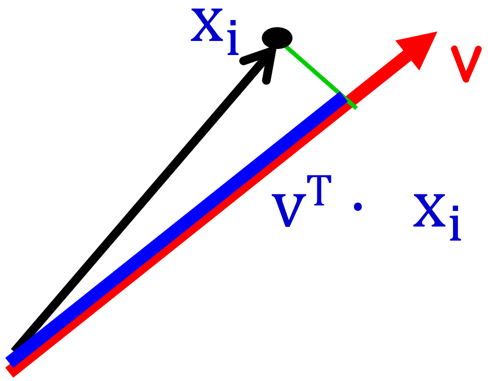
Two Interpretations

So far: Maximum Variance Subspace. PCA finds vectors v such that projections on to the vectors capture maximum variance in the data

$$\sum_{i=1}^{n} (\mathbf{v}^T \mathbf{x}_i)^2 = \mathbf{v}^T \mathbf{X} \mathbf{X}^T \mathbf{v}$$

Alternative viewpoint: Minimum Reconstruction Error. PCA finds vectors v such that projection on to the vectors yields minimum mean squared error (MSE) reconstruction

$$\sum_{i=1}^{n} \|\mathbf{x}_i - (\mathbf{v}^T \mathbf{x}_i)\mathbf{v}\|^2$$

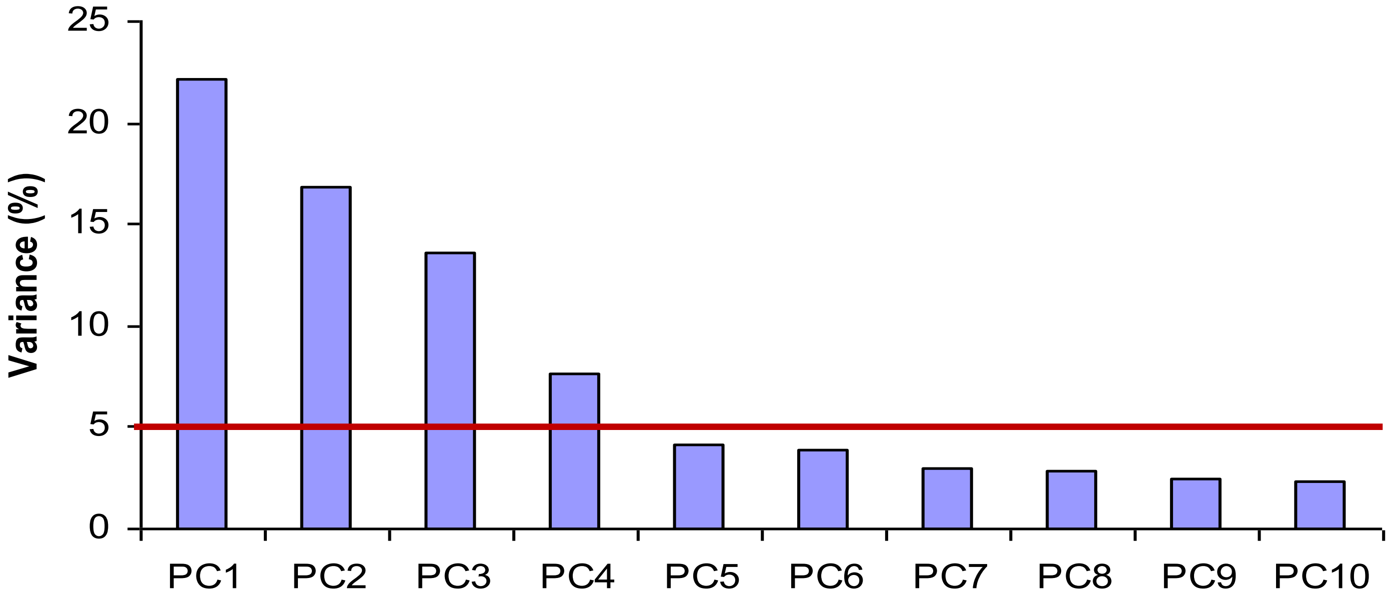


Dimensionality Reduction using PCA

In high-dimensional problems, data sometimes lies near a linear subspace, as noise introduces small variability

Only keep data projections onto principal components with large eigenvalues

Can *ignore* the components of smaller significance.



Might lose some info, but if eigenvalues are small, do not lose much

APPLICATION EXAMPLES

The space of all face images

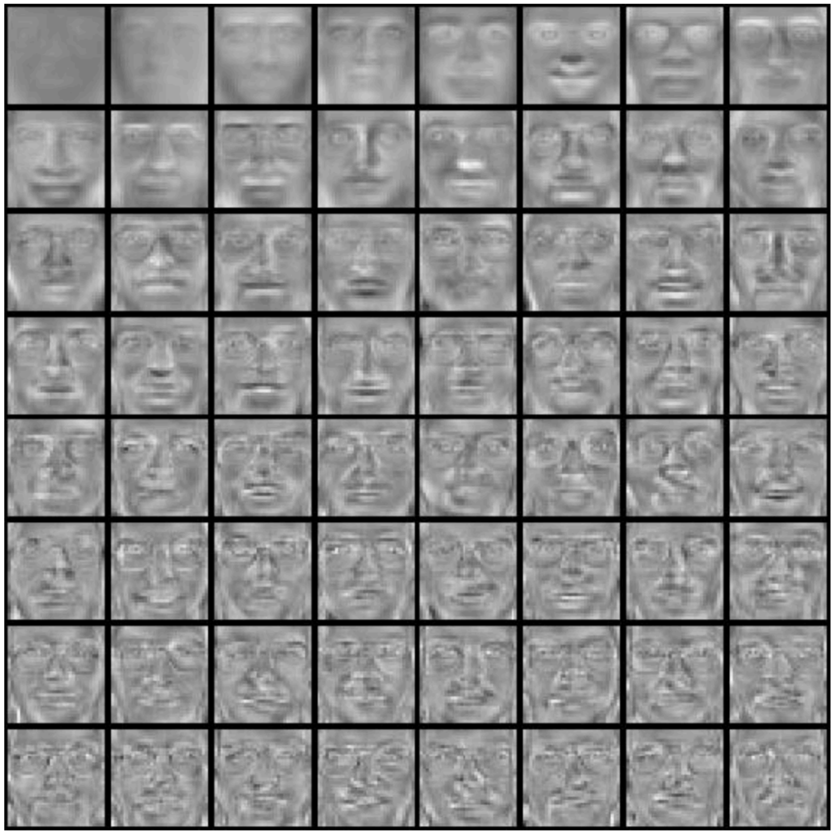
- When viewed as vectors of pixel values, face images are extremely high-dimensional
  - 100x100 image = 10,000 dimensions
  - Slow and lots of storage
- But very few 10,000-dimensional vectors are valid face images
- We want to effectively model the subspace of face images

slide by Derek Hoiem

Eigenfaces example

Mean: μ

Top eigenvectors: $u_1, \ldots u_k$



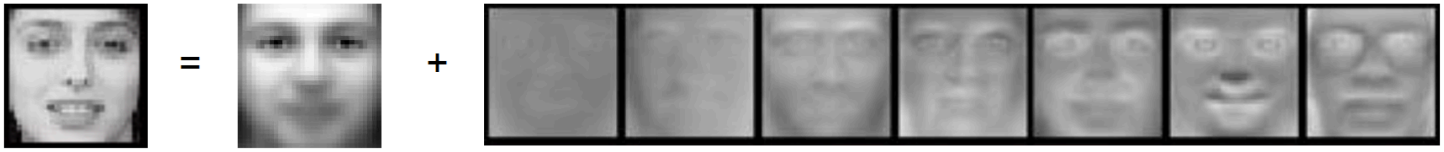
slide by Derek Hoiem

Representation and reconstruction

- Face **x** in "face space" coordinates:

$$\mathbf{x} \rightarrow [\mathbf{u}_1^{\mathrm{T}}(\mathbf{x}-\mu), \ldots, \mathbf{u}_k^{\mathrm{T}}(\mathbf{x}-\mu)]$$
$$= \quad w_1, \ldots, w_k$$

- Reconstruction:



$$\hat{x} = \mu + w_1u_1+w_2u_2+w_3u_3+w_4u_4+ \ldots$$

slide by Derek Hoiem

Reconstruction

After computing eigenfaces using 400 face images from ORL face database

slide by Derek Hoiem