# Kernels

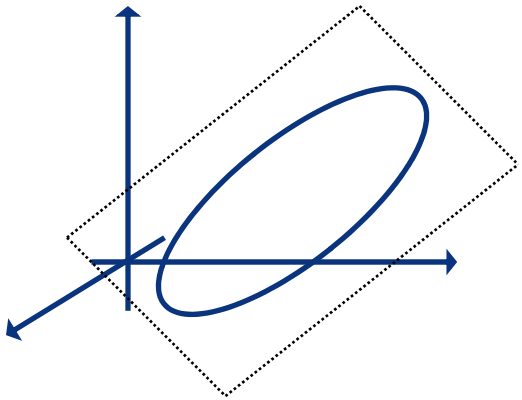
Nuno Vasconcelos

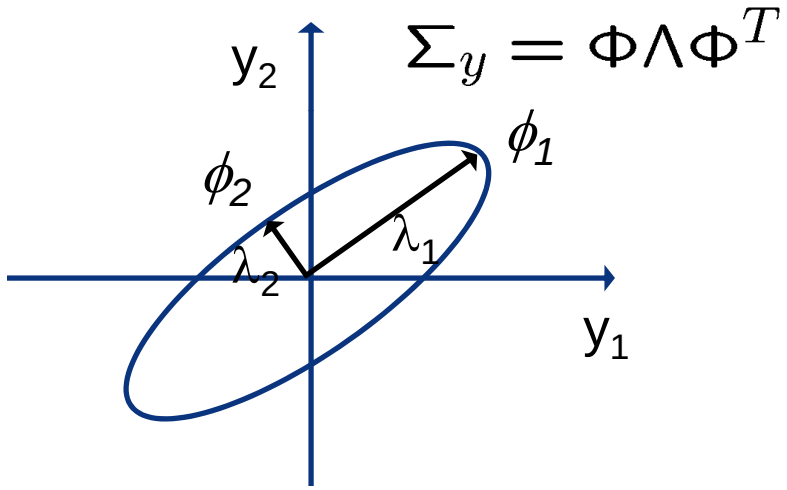
*ECE Department, UCSD*

# Principal component analysis

- Dimensionality reduction:
  - Last time, we saw that when the data lives in a subspace, it is best to design our learning algorithms in this subspace

2D subspace in 3D






- this can be done by computing the principal components of the data
  - principal components $\phi_i$ are the eigenvectors of $\Sigma$
  - principal lengths $\lambda_i$ are the eigenvalues of $\Sigma$

# Principal component analysis (learning)

- Given sample $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, $x_i \in \mathcal{R}^d$
  - compute sample mean: $\hat{\mu} = \frac{1}{n}\sum_i(\mathbf{x}_i)$
  - compute sample covariance: $\hat{\Sigma} = \frac{1}{n}\sum_i(\mathbf{x}_i - \hat{\mu})(\mathbf{x}_i - \hat{\mu})^T$
  - compute eigenvalues and eigenvectors of $\hat{\Sigma}$

$$\hat{\Sigma} = \Phi\Lambda\Phi^T, \;\; \Lambda = diag(\sigma_1^2, \ldots, \sigma_n^2) \;\; \Phi^T\Phi = I$$

  - order eigenvalues $\sigma_1^2 > ... > \sigma_n^2$
  - if, for a certain $k$, $\sigma_k << \sigma_1$ eliminate the eigenvalues and eigenvectors above $k$.

# Principal component analysis

- Given principal compoenents $\phi_i, i \in 1, \ldots, k$ and a test sample $\mathcal{T} = \{\mathbf{t}_1, \ldots, \mathbf{t}_n\}, \;\; t_i \in \mathcal{R}^d$

  - subtract mean to each point $\mathbf{t}'_i = \mathbf{t}_i - \widehat{\mu}$

  - project onto eigenvector space $\mathbf{y}_i = \mathbf{A}\mathbf{t}'_i$ where

$$\mathbf{A} = \begin{bmatrix} \phi_1^T \\ \vdots \\ \phi_k^T \end{bmatrix}$$

  - use $\mathcal{T}' = \{\mathbf{y}_1, \ldots \mathbf{y}_n\}$ to estimate class conditional densities and do all further processing on $\mathbf{y}$.

# PCA by SVD

- we next saw that PCA can be computed by the SVD of the data matrix directly
- given X with one example per column
  - 1) create the centered data-matrix

$$X_c^T = \left( I - \frac{1}{n} 1 1^T \right) X^T$$

  - 2) compute its SVD

$$X_c^T = \mathrm{M} \Pi \mathrm{N}^T$$

  - 3) principal components are columns of N, eigenvalues are

$$\lambda_i = n \sqrt{\pi_i}$$

# Extensions

- Today we will talk about kernels
  - turns out that any algorithm which depends on the data through dot-products only, i.e. the matrix of elements

$$x_i^T x_j$$

    can be kernelized
  - this is usually beneficial, we will see why later
  - for now we look at the question of whether PCA can be written in the form above
- recall the data matrix is

$$X = \begin{bmatrix} | & & | \\ x_1 & \dots & x_n \\ | & & | \end{bmatrix}$$

# Extensions

- we saw that the centered-data matrix and the covariance can be written as

$$X_c = X\left(I - \frac{1}{n}11^T\right) \qquad \Sigma = \frac{1}{n}X_c X_c^T$$

- the eigenvector $\phi_i$ of eigenvalue $\lambda_i$ is

$$\phi_i = \frac{1}{n\lambda_i}X_c X_c{}^T \phi_i = \frac{1}{n\lambda_i}X_c \alpha_i, \qquad \alpha_i = X_c{}^T \phi_i$$

- hence, the eigenvector matrix is

$$\Phi = X_c \Gamma, \qquad \Gamma = \begin{bmatrix} | & & | \\ \alpha_1/n\lambda_d & \cdots & \alpha_d/n\lambda_d \\ | & & | \end{bmatrix}$$

# Extensions

- we next note that, from the eigenvector decomposition

$$\Sigma = \Phi \Lambda \Phi^T \Leftrightarrow \Lambda = \Phi^T \Sigma \Phi$$

- and

$$\Lambda = \Gamma^T X_c{}^T \left( \frac{1}{n} X_c X_c{}^T \right) X_c \Gamma$$

$$= \frac{1}{n} \Gamma^T \left( X_c{}^T X_c \right) \left( X_c{}^T X_c \right) \Gamma$$

- i.e.

$$\frac{1}{n} \left( X_c{}^T X_c \right) \left( X_c{}^T X_c \right) = \Gamma \Lambda \Gamma^T$$

# Extensions

- in summary, we have

$$\Sigma = \Phi \Lambda \Phi^T \qquad \Phi = X_c \Gamma$$

$$\frac{1}{n}\left(X_c{}^T X_c\right)\left(X_c{}^T X_c\right) = \Gamma \Lambda \Gamma^T$$

- this means that we can obtain PCA by
  - 1) assembling $n^{-1}(X_c{}^T X_c)(X_c{}^T X_c)$
  - 2) computing its eigen-decomposition $(\Lambda, \Gamma)$
- PCA
  - the principal components are then given by $X_c \Gamma$
  - the eigenvalues are given by $\Lambda$

# Extensions

- what is interesting here is that we only need the matrix

$$K_c = X_c^T X_c = \begin{bmatrix} - & x_1^c & - \\ & \vdots & \\ - & x_n^c & - \end{bmatrix} \begin{bmatrix} | & & | \\ x_1^c & \cdots & x_n^c \\ | & & | \end{bmatrix}$$

$$= \begin{bmatrix} & \vdots & \\ \cdots & \left(x_n^c\right)^T x_n^c & \cdots \\ & \vdots & \end{bmatrix}$$

- this is the matrix of dot-products of the centered data-points
- notice that you don't need the points themselves, only their dot-products (similarities)

# Extensions

- to compute PCA, we use the fact that

$$\frac{1}{n}\left(X_c{}^T X_c\right)\left(X_c{}^T X_c\right)=\frac{1}{n}K_c K_c{}^T$$

- but if $K_c$ has eigendecomposition $(\Lambda,\Gamma)$

$$\frac{1}{n}K_c K_c^T=\frac{1}{n}\Gamma\Lambda\Gamma^T\Gamma\Lambda\Gamma^T=\frac{1}{n}\Gamma\Lambda^2\Gamma^T$$

- then, $n^{-1}(X_c{}^T X_c)(X_c{}^T X_c)$ has eigendecomposition $(\Lambda^2,\Gamma)$

# Extensions

- in summary, to get PCA
  - 1) compute the dot-product matrix K
  - 2) compute its eigen-decomposition ($\Lambda$,$\Gamma$)
- PCA
  - the principal components are then given by $\Phi = X_c\Gamma$
  - the eigenvalues are given by $\Lambda^2$
  - the projection of the data-points on the principal components is given by

$$X_c^T \Phi = X_c^T X_c \Gamma = K \Gamma$$

- this allows the computation of the eigenvalues and PCA coefficients when we only have access to the dot-product matrix K

# The dot product form

- This turns out to be the case for many learning algorithms
- If you manipulate a little bit, you can write them in “dot product form”

- **Definition:** a learning algorithm is in dot product form if, given a training set

$$D = \{(x_1,y_1), \ldots, (x_n,y_n)\},$$

it only depends on the points $X_i$ through their dot products

$$X_i^T X_j.$$

- for example, let’s look at k-means

# Clustering

- We saw that it iterates bewteen
  - 1) classification:

$$i^*(x) = \arg\min_i \left\| x - \mu_i \right\|^2$$

  - 2) re-estimation:

$$\mu_i^{new} = \frac{1}{n} \sum_j x_j^{(i)}$$

- note that

$$\begin{aligned} \left\| x - \mu_i \right\|^2 &= \left( x - \mu_i \right)^T \left( x - \mu_i \right) \\ &= x^T x - 2 x^T \mu_i + \mu_i^T \mu_i \end{aligned}$$

# Clustering

- and

$$\mu_i = \frac{1}{n}\sum_j x_j^{(i)}$$

- combining the two, we can write the top equation as a function of the dot products $x_i^T x_j$

$$\left\| x_k - \mu_i \right\|^2 = x_k{}^T x_k - \frac{2}{n}\sum_j x_k{}^T x_j^{(i)} + \frac{1}{n^2}\sum_{jl} x_j^{(i)T} x_l^{(i)}$$

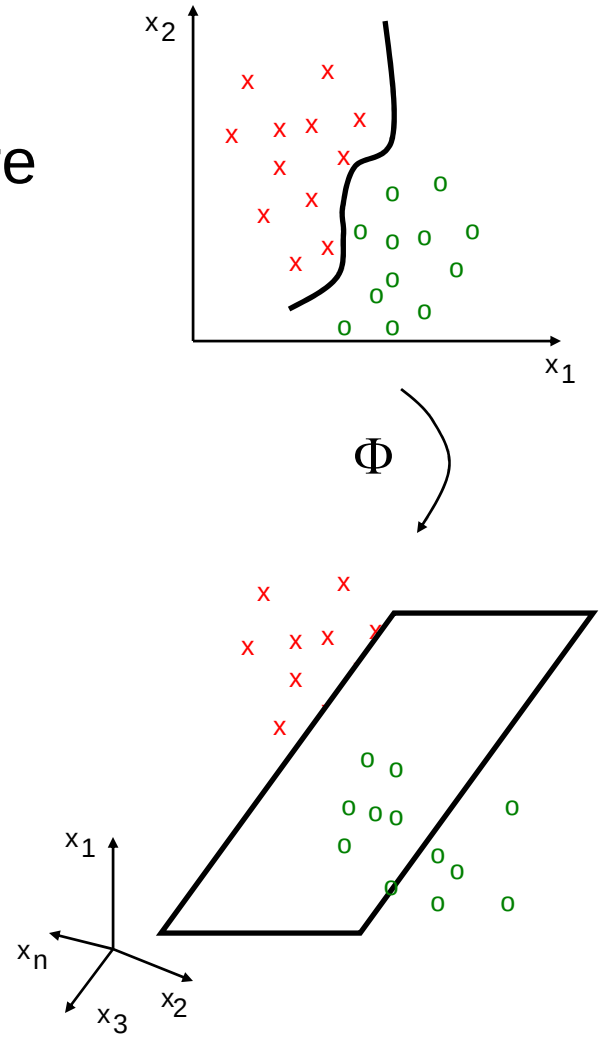
# The kernel trick

- why is this interesting?
- consider transformation of the feature space:
  - introduce a mapping

$$\Phi: \mathcal{X} \rightarrow \mathcal{Z}$$

such that *dim*($\mathcal{Z}$) > *dim*($\mathcal{X}$)

- if the algorithm only depends on the data through dot-products $x_i^T x_j$
- then, in the transformed space, it only depends on $\phi^T(x_i)\phi(x_j)$



# The dot product implementation

- in the transformed space, the learning algorithms only requires dot-products

$$\Phi(x_j)^T \Phi(x_i)$$

- note that we no-longer need to store the $\Phi(x_j)$
- only the $n^2$ dot-product matrix
- interestingly, this holds even when $\Phi(x)$ is infinite dimensional
- we get a reduction from infinity to $n^2$!
- there is, however, still one problem:
  - when $dim[\Phi(x_j)]$ is infinite the computation of the dot products looks impossible

# The “kernel trick”

- “instead of defining defining $\Phi(x)$, computing $\Phi(x_i)$ for each $i$ and $\Phi(x_i)^T \Phi(x_j)$ for each pair $(i,j)$, simply define the function

$$K(x,z) = \Phi(x)^T \Phi(z)$$

  and work with it directly.”
- $K(x,z)$ is called a dot-product kernel
- in fact, since we only use the kernel, why define $\Phi(x)$?
- just define the kernel $K(x,z)$ directly!
- in this way we never have to deal with the complexity of $\Phi(x)$...
- this is usually called the “kernel trick”

# Questions

- I am confused!
- how do I know that if I pick a function $K(x,z)$, it is equivalent to $\Phi(x)^T\Phi(z)$?
  - in general, it is not. We will talk about this later.
- if it is, how do I know what $\Phi(x)$ is?
  - you may never know. E.g. the Gaussian kernel

$$K(x,z) = e^{-\frac{\|x-z\|^2}{\sigma}}$$

    is very popular. It is not obvious what $\Phi(x)$ is...
  - on the positive side, we did not know how to choose $\Phi(x)$. Choosing instead $K(x,z)$ makes no difference.
- why is it that using $K(x,z)$ is easier/better?
  - complexity. let's look at an example.

# Polynomial kernels

- still in $\mathcal{R}^d$, consider the square of the dot product between two vectors

$$\left(x^T z\right)^2 = \left(\sum_{i=1}^{d} x_i z_i\right)^2 = \left(\sum_{i=1}^{d} x_i z_i\right)\left(\sum_{j=1}^{d} x_j z_j\right) =$$

$$= \sum_{i=1}^{d}\sum_{j=1}^{d} x_i x_j z_i z_j$$

$$= x_1 x_1 z_1 z_1 + x_1 x_2 z_1 z_2 + \ldots + x_1 x_d z_1 z_d +$$

$$+ x_2 x_1 z_2 z_1 + x_2 x_2 z_2 z_2 + \ldots + x_2 x_d z_2 z_d +$$

$$\vdots$$

$$+ x_d x_1 z_d z_1 + x_d x_2 z_d z_2 + \ldots + x_d x_d z_d z_d$$

# Polynomial kernels

- can be written as

$$\left(x^T z\right)^2 = \underbrace{\left[x_1 x_1, x_1 x_2, \ldots, x_1 x_d, \ldots, x_d x_1, x_d x_2, \ldots, x_d x_d, \ldots\right]}_{\Phi(x)^T} \begin{bmatrix} z_1 z_1 \\ z_1 z_2 \\ \vdots \\ z_1 z_d \\ \vdots \\ z_d z_1 \\ z_d z_2 \\ \vdots \\ z_d z_d \end{bmatrix} \Phi(z)$$

- hence, we have

$$K(x,z) = \left(x^T z\right)^2 = \Phi(x)^T \Phi(z)$$

with $\Phi: \quad \Re^d \rightarrow \Re^{d^2}$

$$\begin{pmatrix} x_1 \\ \vdots \\ x_d \end{pmatrix} \rightarrow \left(x_1 x_1, x_1 x_2, \cdots, x_1 x_d, \cdots, x_d x_1, x_d x_2, \cdots, x_d x_d\right)^T$$

# Polynomial kernels

- the point is that
  - while $\Phi(x)^T \Phi(z)$ has complexity $O(d^2)$
  - direct computation of $K(x,z) = (x^T z)^2$ has complexity $O(d)$
- direct evaluation is more efficient by a factor of $d$
- as $d$ goes to infinity this makes the idea feasible
- BTW, you just met another kernel family
  - this implements polynomials of second order
  - in general, the family of polynomial kernels is defined as

$$K(x,z) = \left(1 + x^T z\right)^k, \quad k \in \{1,2,\cdots\}$$

  - I don't even want to think about writing down $\Phi(x)$ !

# Kernel summary

1. *D* not easy to deal with in $\mathcal{X}$, apply feature transformation $\Phi: \mathcal{X} \to \mathcal{Z}$, such that *dim*($\mathcal{Z}$) >> *dim*($\mathcal{X}$)
2. computing $\Phi(x)$ too expensive:
   - write your learning algorithm in dot-product form
   - instead of $\Phi(x_i)$, we only need $\Phi(x_i)^T \Phi(x_j) \ \forall_{ij}$
3. instead of computing $\Phi(x_i)^T \Phi(x_j) \ \forall_{ij}$, define the "dot-product kernel"

$$K(x, z) = \Phi(x)^T \Phi(z)$$

   and compute $K(x_i, x_j) \ \forall_{ij}$ directly

   - note: the matrix

$$K = \begin{bmatrix} \vdots \\ \cdots K(x_i, z_j) \cdots \\ \vdots \end{bmatrix}$$

     is called the "kernel" or Gram matrix

4. forget about $\Phi(x)$ and use $K(x,z)$ from the start!

# Question

- what is a good dot-product kernel?
  - this is a difficult question (see Prof. Lenckriet's work)
- in practice, the usual recipe is:
  - pick a kernel from a library of known kernels
  - we have already met
    - the linear kernel $K(x,z) = x^T z$
    - the Gaussian family

$$K(x,z) = e^{-\frac{\|x-z\|^2}{\sigma}}$$

    - the polynomial family

$$K(x,z) = \left(1 + x^T z\right)^k, \quad k \in \{1,2,\cdots\}$$

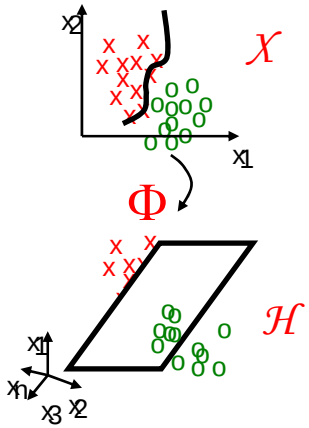
# Dot-product kernels

- this may not be a bad idea
  - we rip the benefits of a high-dimensional space without a price in complexity
  - the kernel simply adds a few parameters $(\sigma, k)$ learning it would imply introducing many parameters (up to $n^2$)
- what if I need to check whether $K(x,z)$ is a kernel?
- **Definition:** a mapping

$$k: \mathcal{X} \times \mathcal{X} \rightarrow \mathcal{R}$$
$$(x,y) \rightarrow k(x,y)$$

is a dot-product kernel if and only if

$$k(x,y) = \langle \Phi(x), \Phi(y) \rangle$$




where $\Phi: \mathcal{X} \rightarrow \mathcal{H}$, $\mathcal{H}$ is a vector space and <.,.> a dot-product in $\mathcal{H}$

# Positive definite matrices

- recall that (e.g. Linear Algebra and Applications, Strang)
- **Definition:** each of the following is a necessary and sufficient condition for a real symmetric matrix $A$ to be (semi) positive definite:
  - i) $x^T A x \geq 0, \ \forall x \neq 0$
  - ii) all eigenvalues of $A$ satisfy $\lambda_i \geq 0$
  - iii) all upper-left submatrices $A_k$ have non-negative determinant
  - iv) there is a matrix $R$ with independent rows such that $A = R^T R$
- upper left submatrices:

$$A_1 = a_{1,1} \qquad A_2 = \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix} \qquad A_3 = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \qquad \cdots$$

# Positive definite matrices

- property iv) is particularly interesting
  - in $\mathcal{R}^d$, $\langle x,x \rangle = x^T A x$ is a dot-product kernel if and only if A is positive definite
  - from iv) this holds if and only if there is $R$ such that $A = R^T R$
  - hence

$$\langle x,y \rangle = x^T A y = (xR)^T (Ry) = \Phi(x)^T \Phi(y)$$

*with*

$$\Phi: \mathcal{R}^d \rightarrow \mathcal{R}^d$$
$$x \rightarrow Rx$$

- i.e. the dot-product kernel

$$k(x,z) = x^T A z, \quad (A \text{ positive definite})$$

- is the standard dot-product in the range space of the mapping $\Phi(x) = Rx$

# Positive definite kernels

- how do I extend this notion of positive definiteness to functions?
- Definition: a function $k(x,y)$ is a positive definite kernel on $\mathcal{X}x\mathcal{X}$ if $\forall l$ and $\forall \{x_1, ..., x_l\}$, $x_i \in \mathcal{X}$, the Gram matrix

$$K = \begin{bmatrix} \vdots \\ \cdots k(x_i, x_j) \cdots \\ \vdots \end{bmatrix}$$

  is positive definite.
- like in in $\mathcal{R}^d$, this allows us to check that we have a positive definite kernel

# Dot product kernels

- **Theorem:** $k(x,y)$, $x,y \in \mathcal{X}$, is a dot-product kernel if and only if it is a positive definite kernel
- in summary, to check whether a kernel is a dot product:
  - check if the Gram matrix is positive definite
  - for all possible sequences $\{x_1, ..., x_l\}$, $x_i \in \mathcal{X}$
- does the kernel have to be a dot-product kernel?
- not necessarily. For example, neural networks can be seen as implementing kernels that are not of this type
- however:
  - you loose the parallelism. what you know about the learning machine may no longer hold after you kernelize
  - dot-product kernels usually lead to convex learning problems. Usually you loose this guarantee for non dot-product

# Clustering

- so far, this is mostly theoretical
- how does it affect my algorithms?
- consider, for example, the k-means algorithm
  - 1) classification:

$$i^*(x) = \arg\min_i \left\| x - \mu_i \right\|^2$$

  - 2) re-estimation:

$$\mu_i^{new} = \frac{1}{n} \sum_j x_j^{(i)}$$

- can we kernelize the classification step?

# Clustering

- well, we saw that

$$\left\| x_k - \mu_i \right\|^2 = x_k{}^T x_k - \frac{2}{n} \sum_j x_k{}^T x_j^{(i)} + \frac{1}{n^2} \sum_{jl} x_j^{(i)T} x_l^{(i)}$$

- this can then be kernelized into

$$\left\| x_k - \mu_i \right\|^2 = \Phi(x_k)^T \Phi(x_k) - \frac{2}{n} \sum_j \Phi(x_k)^T \Phi\left(x_j^{(i)}\right) + \frac{1}{n^2} \sum_{jl} \Phi\left(x_j^{(i)}\right)^T \Phi\left(x_l^{(i)}\right)$$

# Clustering

- furthermore, this can be done with relative efficiency

$$\|x_k - \mu_i\|^2 = \Phi(x_k)^T \Phi(x_k) - \frac{2}{n}\sum_j \Phi(x_k)^T \Phi(x_j^{(i)}) + \frac{1}{n^2}\sum_{jl} \Phi(x_j^{(i)})^T \Phi(x_l^{(i)})$$

$k^{th}$ diagonal entry of Gram matrix

computed once per cluster when all points are assigned

- the assignment of the point only requires computing for each cluster

$$\frac{2}{n}\sum_j \Phi(x_k)^T \Phi(x_j^{(i)})$$

- this is a sum of entries of Gram matrix

# Clustering

- note, however, that we cannot explicitly compute

$$\Phi\left(\mu_i\right)=\frac{1}{n}\sum_j \Phi\left(x_j^{(i)}\right)$$

- this is probably infinite dimensional...
- in any case, if we define
  - a Gram matrix $K^{(i)}$ for each cluster (dot products between points in cluster)
  - and $S^{(i)}$ the scaled sum of the entries in this matrix

$$S^{(i)}=\frac{1}{n^2}\sum_{jl}\Phi\left(x_j^{(i)}\right)^T\Phi\left(x_l^{(i)}\right)$$

# Clustering

- we obtain the kernel k-means algorithm
  - 1) classification:

$$i^*(x_l) = \arg\min_i \left[ K_{l,l} + s^{(i)} - \frac{2}{n} \sum_j \Phi(x_l)^T \Phi\left(x_j^{(i)}\right) \right]$$

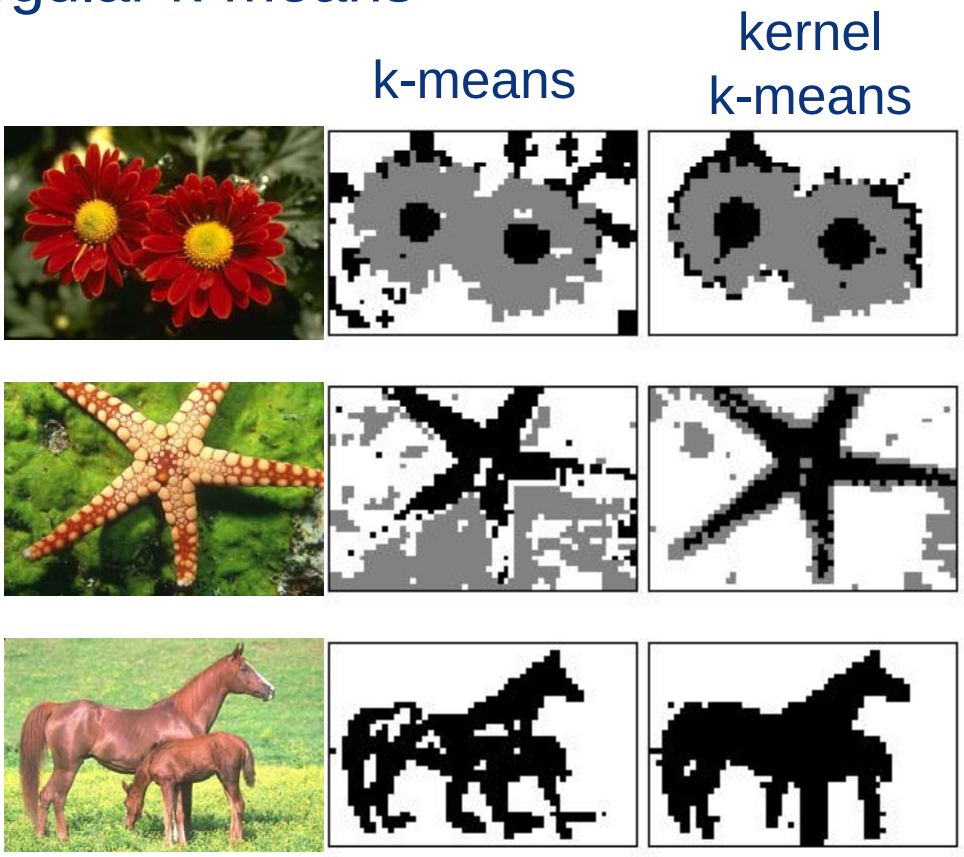
  - 2) re-estimation: update

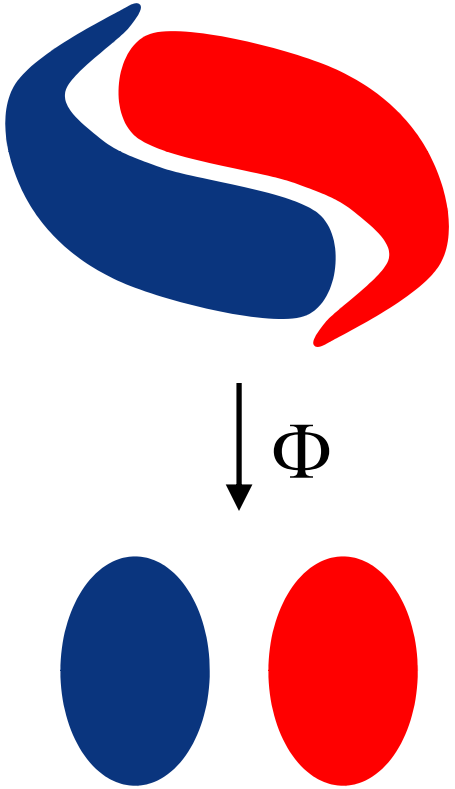
$$s^{(i)} = \frac{1}{n^2} \sum_{jl} \Phi\left(x_j^{(i)}\right)^T \Phi\left(x_l^{(i)}\right)$$

- but we no longer have access to the prototype for each cluster

# Clustering

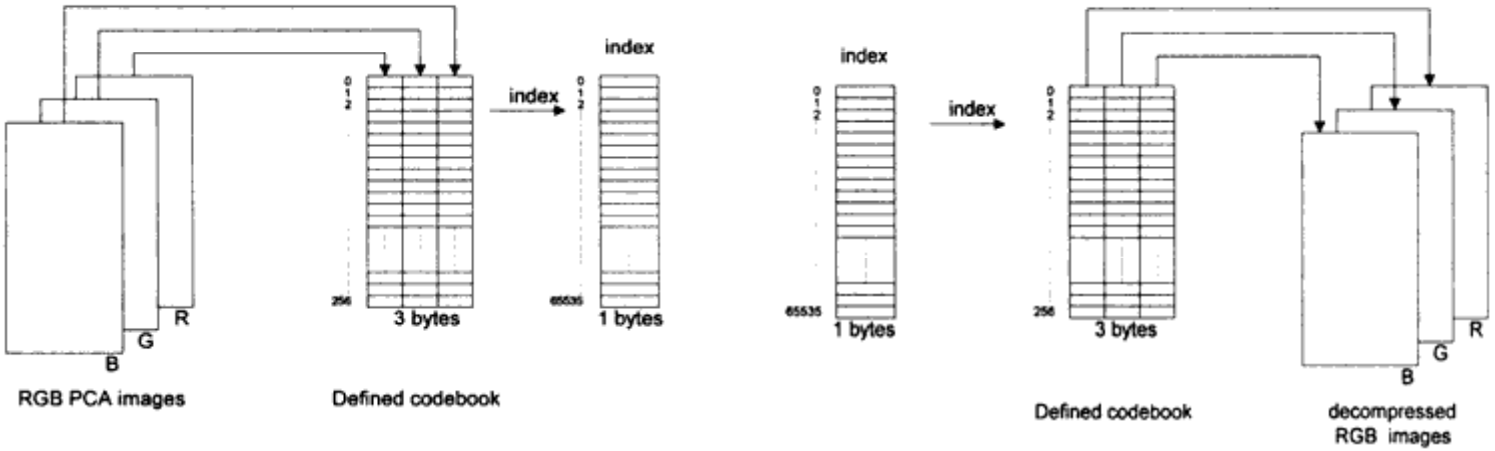
- with the right kernel this can work significantly better than regular k-means



# Clustering

- but for other applications, where the prototypes are important, this may be useless
- e.g. compression




- we can try replacing the prototype by the closest vector, but this is not necessarily optimal

# PCA

- we saw that, to get PCA
  - 1) compute the dot-product matrix K
  - 2) compute its eigen-decomposition $(\Lambda,\Gamma)$
- PCA
  - the principal components are then given by $\Phi = X_c\Gamma$
  - the eigenvalues are given by $\Lambda^2$
  - the projection of the data-points on the principal components is given by

$$X_c^T \Phi = K\,\Gamma$$

- note that most of this holds when we kernelize, we only have to change the matrix $K$ from $x_i^T x_j$ to $\phi(x_i)^T\phi(x_j)$
  - the only thing we can no longer access are the PCs $\Phi = X_c\Gamma$

# Kernel methods

- most learning algorithms can be kernelized
  - kernel PCA
  - kernel LDA
  - kernel ICA,
  - etc.
- as in k-means, sometimes we loose some of the features of the original algorithm
- but the performance is frequently better
- next week we will look at the canonical application, the support vector machine


Any Questions?