

# Learning with Kernels III

CS771: Introduction to Machine Learning

Purushottam Kar

# Recap of Last Lecture

2

Learning non-linear models via kernels

*Map data to a high dimensional space using a non-linear map*

*Learn a linear model in that space using our favorite algorithms*

The Kernel Trick: avoid explicitly computing this map

*Several ML algos are kernelizable i.e. they function identically if instead of feat vecs, they are given pairwise dot products between those feat vecs*

*Thus, instead of choosing  $\phi$  and feeding  $\phi(\mathbf{x}^i)$  to the algo, we instead choose  $K(\cdot, \cdot) = \langle \phi(\cdot), \phi(\cdot) \rangle$  and feed  $K(\cdot, \cdot)$  to the algo and algo behaves identically*

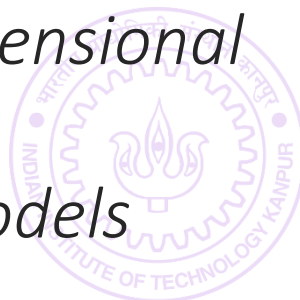
*Saw this to be the case with LwP, kNN, SVM*

*Computing  $K(\cdot, \cdot)$  often takes only  $\mathcal{O}(d)$  time even if map  $\phi$  is  $\infty$ -dimensional*

*Polynomial, Gaussian, Laplacian etc kernels satisfy this*

*However, kernel algos are still more expensive than learning linear models*

*Price to pay for more powerful non-linear models that are more expressive*



# Kernels and Kernels

3



# Kernels and Kernels

3

$$K(\mathbf{x}, \mathbf{y})$$



# Kernels and Kernels

3



Kernels of corn

$$K(\mathbf{x}, \mathbf{y})$$

# Kernels and Kernels

3



Kernels of corn

$$\{\mathbf{x} : A\mathbf{x} = \mathbf{0}\}$$

Kernel of linear transformation

$$K(\mathbf{x}, \mathbf{y})$$

# Kernels and Kernels

3



Kernels of corn

$$\{\mathbf{x} : A\mathbf{x} = \mathbf{0}\}$$

Kernel of linear transformation

$$K(\mathbf{x}, \mathbf{y})$$



Operating System

OS Kernel





# Kernels and Kernels

3



Kernels of corn

$$K(\mathbf{x}, \mathbf{y})$$

$$\{\mathbf{x} : A\mathbf{x} = \mathbf{0}\}$$

Kernel of linear transformation



$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$



Convolution kernels (masks)



Operating System

OS Kernel





# Kernels and Kernels

3



Kernels of corn

$$\{\mathbf{x} : A\mathbf{x} = \mathbf{0}\}$$

Kernel of linear transformation

$$K(\check{\mathbf{x}}, \mathbf{y})$$



$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$



Convolution kernels (masks)



Operating System

OS Kernel

# Kernels and Kernels

3



Kernels of corn

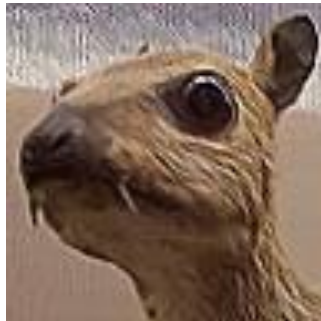
$$\{\mathbf{x} : A\mathbf{x} = \mathbf{0}\}$$

Kernel of linear transformation

$$K(\check{\mathbf{x}}, \mathbf{y})$$



$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$



Convolution kernels (masks)



OS Kernel

# Kernels and Kernels

3



Kernels of corn



$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$



Convolution kernels (masks)

$$\{\mathbf{x} : A\mathbf{x} = \mathbf{0}\}$$

Kernel of linear transformation

$$K(\check{\mathbf{x}}, \mathbf{y})$$



Operating System

OS Kernel

# Kernel kNN

12

**Step 1:** choose a kernel  $K$  with map  $\phi$

*Gaussian kernel is popular but domain specific kernels may do better*

**Step 2:** training

*Receive training points  $(\mathbf{x}^i, y^i), i \in [n], \mathbf{x}^i \in \mathbb{R}^d$*

*Store them*

**Step 3:** prediction

*Receive a test point  $\mathbf{x}^t \in \mathbb{R}^d$*

*Find the nearest neighbour using  $i^t = \arg \min_{i \in [n]} \|\phi(\mathbf{x}^t) - \phi(\mathbf{x}^i)\|_{\mathcal{H}}^2$*

*Too expensive so instead use  $i^t = \arg \min_{i \in [n]} K(\mathbf{x}^i, \mathbf{x}^i) - 2K(\mathbf{x}^i, \mathbf{x}^t)$*

*Predict  $y^{i^t}$*





**Step 1:** choose a kernel  $K$  with map  $\phi$

*Gaussian kernel is popular but domain specific kernels may do better*

**Step 2:** training

*Receive training points  $(\mathbf{x}^i, y^i), i \in [n], \mathbf{x}^i \in \mathbb{R}^d, y^i \in \{-1, 1\}$*

*Store them (cannot compute prototypes explicitly as  $\phi$  may be  $\infty$  dim ☹)*

**Step 3:** prediction

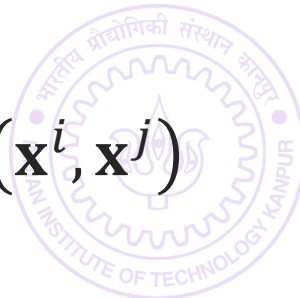
*Receive a test point  $\mathbf{x}^t \in \mathbb{R}^d$*

*Find distance to +ve prototype  $\|\phi(\mathbf{x}^t) - \tilde{\mathbf{p}}^+\|_2^2$  where  $\tilde{\mathbf{p}}^+ = \frac{1}{n_+} \cdot \sum_{y^i=1} \phi(\mathbf{x}^i)$*

*Too expensive so instead use the formula*

$$\|\phi(\mathbf{x}^t) - \tilde{\mathbf{p}}^+\|_2^2 = K(\mathbf{x}^t, \mathbf{x}^t) - \frac{2}{n_+} \cdot \sum_{y^i=1} K(\mathbf{x}^t, \mathbf{x}^i) + \frac{1}{n_+^2} \cdot \sum_{y^i=1} \sum_{y^j=1} K(\mathbf{x}^i, \mathbf{x}^j)$$

*Predict  $\text{sign}(\|\phi(\mathbf{x}^t) - \tilde{\mathbf{p}}^-\|_2^2 - \|\phi(\mathbf{x}^t) - \tilde{\mathbf{p}}^+\|_2^2)$*



**Step 1:** choose a kernel  $K$  with map  $\phi$

*Gaussian kernel is popular but domain specific kernels may do better*

**Step 2:** training

*Receive training points  $(\mathbf{x}^i, y^i), i \in [n], \mathbf{x}^i \in \mathbb{R}^d, y^i \in \{-1, 1\}$*

*Solve the dual problem  $\max_{\alpha_i} \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y^i y^j K(\mathbf{x}^i, \mathbf{x}^j)$  s.t.  $\alpha_i \in [0, C]$*

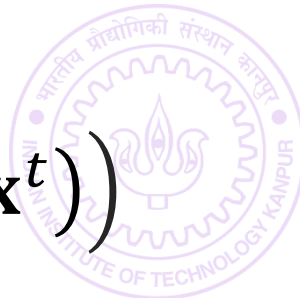
$\mathbf{w} = \sum_{i=1}^n \alpha_i y^i \phi(\mathbf{x}^i) \in \mathcal{H}$  but cannot store it explicitly

*So instead, store  $(\mathbf{x}^i, y^i, \alpha_i)$  for all support vectors i.e where  $\alpha_i \neq 0$*

**Step 3:** prediction

*Receive a test point  $\mathbf{x}^t \in \mathbb{R}^d$*

*Use kernel trick to predict  $\text{sign}(\mathbf{w}^\top \phi(\mathbf{x}^t)) = \text{sign}\left(\sum_{i=1}^n \alpha_i y^i K(\mathbf{x}^i, \mathbf{x}^t)\right)$*



# Kernel Ridge Regression

15

Given data  $(\mathbf{x}^i, y^i)_{i=1}^n, \mathbf{x}^i \in \mathbb{R}^d, y^i \in \mathbb{R}$ , RR solution is simply

$$\hat{\mathbf{w}} = (X^\top X + \lambda \cdot I_d)^{-1} X^\top \mathbf{y}$$

*Is ridge-regression kernelizable? Does not seem so at first*

*In fact, it is – by using the dual problem of ridge regression*

**Deriving the dual for RR:** RR solves  $\min_{\mathbf{w} \in \mathbb{R}^d} \lambda \cdot \|\mathbf{w}\|_2^2 + \|\mathbf{y} - X\mathbf{w}\|_2^2$

*Dual requires constraints – none here so lets deliberately introduce some!*

*New (but equivalent) formulation:  $\min_{\mathbf{w} \in \mathbb{R}^d} \lambda \cdot \|\mathbf{w}\|_2^2 + \|\mathbf{z}\|_2^2$  s.t.  $\mathbf{z} = \mathbf{y} - X\mathbf{w}$*

*Lagrangian becomes  $\lambda \cdot \|\mathbf{w}\|_2^2 + \|\mathbf{z}\|_2^2 + \boldsymbol{\alpha}^\top (\mathbf{y} - X\mathbf{w} - \mathbf{z})$*

*Applying first order optimality gives us  $\mathbf{z} = \boldsymbol{\alpha}/2$  and  $2\lambda \cdot \mathbf{w} = X^\top \boldsymbol{\alpha}$*

*Dual becomes  $\min_{\boldsymbol{\alpha} \in \mathbb{R}^n} \frac{1}{4\lambda} \cdot \sum_{i,j} \alpha_i \alpha_j \langle \mathbf{x}^i, \mathbf{x}^j \rangle + \frac{1}{4} \cdot \|\boldsymbol{\alpha}\|_2^2 - \boldsymbol{\alpha}^\top \mathbf{y}$*

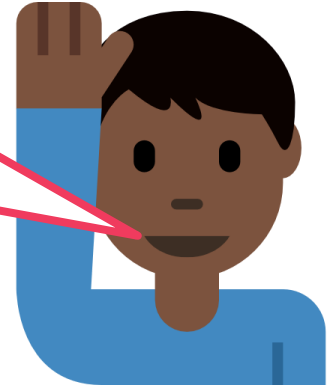




# Kernel Ridge Regression

16

Given To handle equality constraints  $f(x) = c$   
**Method 1:** convert to a pair of inequality constraints  $f(x) \leq c, f(x) \geq c$   
**Method 2:** use a Lagrangian variable that has no constraints ☺



*Is ridge-regression kernelizable? Does not seem so at first*

*In fact, it is – by using the dual problem of ridge regression*

**Deriving the dual for RR:** RR solves  $\min_{\mathbf{w} \in \mathbb{R}^d} \lambda \cdot \|\mathbf{w}\|_2^2 + \|\mathbf{y} - X\mathbf{w}\|_2^2$

*Dual requires constraints – none here so lets deliberately introduce some!*

*New (but equivalent) formulation:  $\min_{\mathbf{w} \in \mathbb{R}^d} \lambda \cdot \|\mathbf{w}\|_2^2 + \|\mathbf{z}\|_2^2$  s.t.  $\mathbf{z} = \mathbf{y} - X\mathbf{w}$*

*Lagrangian becomes  $\lambda \cdot \|\mathbf{w}\|_2^2 + \|\mathbf{z}\|_2^2 + \boldsymbol{\alpha}^\top (\mathbf{y} - X\mathbf{w} - \mathbf{z})$*

*Applying first order optimality gives us  $\mathbf{z} = \boldsymbol{\alpha}/2$  and  $2\lambda \cdot \mathbf{w} = X^\top \boldsymbol{\alpha}$*

*Dual becomes  $\min_{\boldsymbol{\alpha} \in \mathbb{R}^n} \frac{1}{4\lambda} \cdot \sum_{i,j} \alpha_i \alpha_j \langle \mathbf{x}^i, \mathbf{x}^j \rangle + \frac{1}{4} \cdot \|\boldsymbol{\alpha}\|_2^2 - \boldsymbol{\alpha}^\top \mathbf{y}$*



# Kernel Ridge Regression

17

Thus RR does have a dual problem (that makes it kernelizable too)

$$\text{Solve } \hat{\boldsymbol{\alpha}} = \underset{\boldsymbol{\alpha} \in \mathbb{R}^n}{\operatorname{argmin}} \frac{1}{4\lambda} \cdot \sum_{i,j} \alpha_i \alpha_j K(\mathbf{x}^i, \mathbf{x}^j) + \frac{1}{4} \cdot \|\boldsymbol{\alpha}\|_2^2 - \boldsymbol{\alpha}^\top \mathbf{y}$$

Model is  $\hat{\mathbf{w}} = \frac{1}{2\lambda} \cdot \sum_{i=1}^n \hat{\alpha}_i \cdot \phi(\mathbf{x}^i) \in \mathcal{H}$  cannot be stored explicitly

Given a test point  $\mathbf{x}^t$ , predict as  $\langle \hat{\mathbf{w}}, \phi(\mathbf{x}^t) \rangle = \frac{1}{2\lambda} \cdot \sum_{i=1}^n \hat{\alpha}_i \cdot K(\mathbf{x}^i, \mathbf{x}^t)$

Some simplifications

Let  $G = [K(\mathbf{x}^i, \mathbf{x}^j)]_{i,j} \in \mathbb{R}^{n \times n}$  denote the “Gram matrix” of the training points

Dual of kernel RR can be rewritten as  $\min_{\boldsymbol{\alpha} \in \mathbb{R}^n} \frac{1}{4\lambda} \boldsymbol{\alpha}^\top (G + \lambda \cdot I_n) \boldsymbol{\alpha} - \boldsymbol{\alpha}^\top \mathbf{y}$

Solution available in closed form  $\hat{\boldsymbol{\alpha}} = 2\lambda \cdot (G + \lambda \cdot I_n)^{-1} \mathbf{y}$

Requires inverting  $n \times n$  matrix (linear RR required inverting  $d \times d$  matrix)

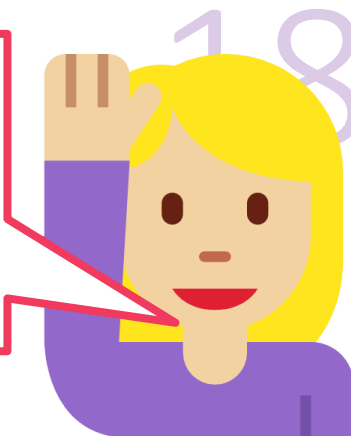
As before, kernel RR requires more train time, test time and larger model size



Key

Thus

Note however, that we can use this dual trick to solve RR even in the linear case when  $d > n$ . Solving linear RR in primal requires  $\mathcal{O}(d^3)$  time (to invert a  $d \times d$  matrix) whereas solving linear RR in dual requires  $\mathcal{O}(n^3)$  time (to invert an  $n \times n$  matrix). Thus, if  $d > n$ , dual solution is cheaper



$$\text{Solve } \boldsymbol{\alpha} = \underset{\boldsymbol{\alpha} \in \mathbb{R}^n}{\operatorname{argmin}} \frac{1}{4\lambda} \cdot \sum_{i,j} \alpha_i \alpha_j K(\mathbf{x}^i, \mathbf{x}^j) + \frac{1}{4} \cdot \|\boldsymbol{\alpha}\|_2^2 - \boldsymbol{\alpha}^\top \mathbf{y}$$

Model is  $\hat{\mathbf{w}} = \frac{1}{2\lambda} \cdot \sum_{i=1}^n \hat{\alpha}_i \cdot \phi(\mathbf{x}^i) \in \mathcal{H}$  cannot be stored explicitly

Given a test point  $\mathbf{x}^t$ , predict as  $\langle \hat{\mathbf{w}}, \phi(\mathbf{x}^t) \rangle = \frac{1}{2\lambda} \cdot \sum_{i=1}^n \hat{\alpha}_i \cdot K(\mathbf{x}^i, \mathbf{x}^t)$

Some simplifications

Let  $G = [K(\mathbf{x}^i, \mathbf{x}^j)]_{i,j} \in \mathbb{R}^{n \times n}$  denote the “Gram matrix” of the training points

Dual of kernel RR can be rewritten as  $\min_{\boldsymbol{\alpha} \in \mathbb{R}^n} \frac{1}{4\lambda} \boldsymbol{\alpha}^\top (G + \lambda \cdot I_n) \boldsymbol{\alpha} - \boldsymbol{\alpha}^\top \mathbf{y}$

Solution available in closed form  $\hat{\boldsymbol{\alpha}} = 2\lambda \cdot (G + \lambda \cdot I_n)^{-1} \mathbf{y}$

Requires inverting  $n \times n$  matrix (linear RR required inverting  $d \times d$  matrix)

As before, kernel RR requires more train time, test time and larger model size



Should be relatively simple given our experience with kernel LwP, kNN

## K-MEANS/LLOYD'S ALGORITHM

1. Initialize centroids  $\{\boldsymbol{\mu}^k\}_{k=1,\dots,K} \in \mathbb{R}^d$
2. For  $i \in [n]$ , do cluster assignment, update  $z^i \in [K]$  using  $\boldsymbol{\mu}^k$ 
  1. Let  $z^i = \arg \min_k \|\mathbf{x}^i - \boldsymbol{\mu}^k\|_2^2$
3. Update  $\boldsymbol{\mu}^k = \frac{1}{n_k} \sum_{i: z^i = k} \mathbf{x}^i$ , where  $n_k = |\{i: z^i = k\}|$
4. Repeat until convergence

All we need to do is kernelize the distance computations and keep track of the cluster centers implicitly since now  $\boldsymbol{\mu}^k \in \mathcal{H}$



# Kernel K-means

20

Note that cluster centers in k-means are always the average of data points that were assigned to that cluster -  $\mathbf{z}^i$  maintains this info

*Need to maintain this information a bit differently for easy processing*

*Let  $\alpha_i^k = 1$  if  $\mathbf{z}^i = k$  and  $\alpha_i^k = 0$  if  $\mathbf{z}^i \neq k$  i.e.  $n_k = \sum_{i=1}^n \alpha_i^k = \mathbf{1}^\top \boldsymbol{\alpha}^k$*

*This lets us write  $\boldsymbol{\mu}^k = \left(\sum_{i=1}^n \alpha_i^k\right)^{-1} \cdot \sum_{i=1}^n \alpha_i^k \cdot \phi(\mathbf{x}^i) \in \mathcal{H}$*

*Let  $G = [K(\mathbf{x}^i, \mathbf{x}^j)]_{i,j} \in \mathbb{R}^{n \times n}$  denote the Gram matrix of training points*

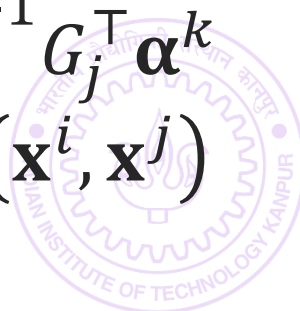
*Using this, we can rewrite distance computations as*

$$\|\phi(\mathbf{x}^j) - \boldsymbol{\mu}^k\|_{\mathcal{H}}^2 = K(\mathbf{x}^j, \mathbf{x}^j) - 2\langle \boldsymbol{\mu}^k, \phi(\mathbf{x}^j) \rangle + \langle \boldsymbol{\mu}^k, \boldsymbol{\mu}^k \rangle$$

*where  $\langle \boldsymbol{\mu}^k, \phi(\mathbf{x}^j) \rangle = \left(\sum_{i=1}^n \alpha_i^k\right)^{-1} \cdot \sum_{i=1}^n \alpha_i^k \cdot K(\mathbf{x}^i, \mathbf{x}^j) = (\mathbf{1}^\top \boldsymbol{\alpha}^k)^{-1} G_j^\top \boldsymbol{\alpha}^k$*

*( $G_j$  is  $j$ -th column of  $G$ ) and  $\langle \boldsymbol{\mu}^k, \boldsymbol{\mu}^k \rangle = \left(\sum_{i=1}^n \alpha_i^k\right)^{-2} \cdot \sum_{i,j=1}^n \alpha_i^k \alpha_j^k K(\mathbf{x}^i, \mathbf{x}^j)$*

*which is nothing but  $(\mathbf{1}^\top \boldsymbol{\alpha}^k)^{-2} (\boldsymbol{\alpha}^k)^\top G \boldsymbol{\alpha}^k$*



points

# KERNEL K-MEANS

Nee

*Let*

*This!*

*Let*

Usir

when

 $(G_j$ 

*which is nothing but*

1. Initialize  $\{\boldsymbol{\alpha}^k\}_{k=1,\dots,K} \in \{0,1\}^n$  randomly
2. For  $i \in [n]$ , do cluster assignment
  1. Let  $n_k = \mathbf{1}^\top \boldsymbol{\alpha}^k$
  2. Let  $z^i = \arg \min_k n_k^{-2} \cdot (\boldsymbol{\alpha}^k)^\top G \boldsymbol{\alpha}^k - 2 n_k^{-1} \cdot G_i^\top \boldsymbol{\alpha}^k$
3. For  $k \in [K]$  update  $\boldsymbol{\alpha}^k$ 
  1. For  $i \in [n]$ , set  $\boldsymbol{\alpha}_i^{z^i} = 1$  and  $\boldsymbol{\alpha}_i^l = 0$  for all  $l \neq z^i$
4. Repeat until convergence

$$j^T \alpha^k$$

# Kernel PCA

22





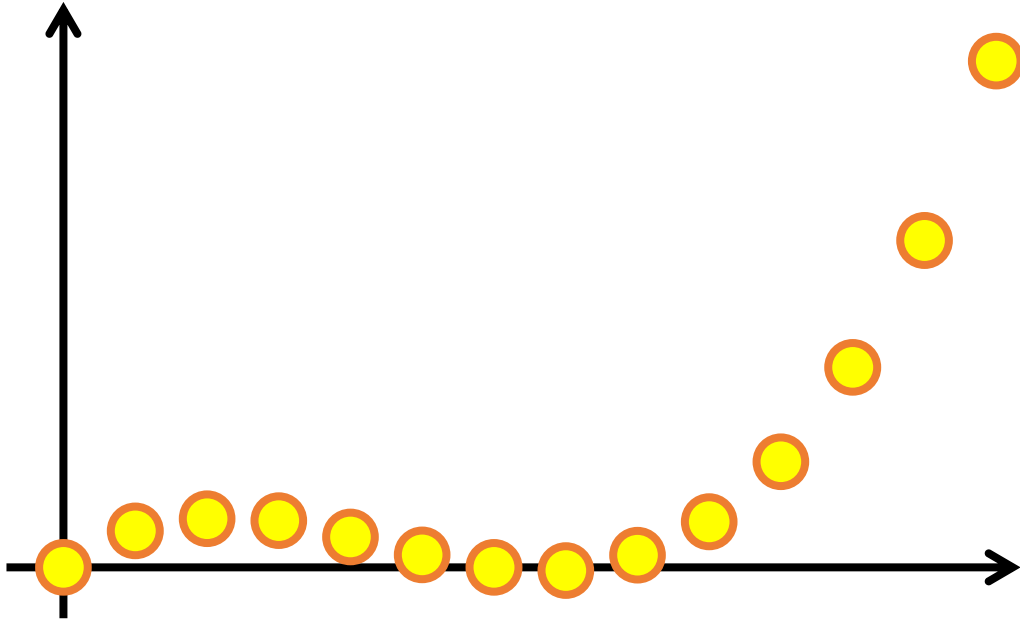
# Kernel PCA

22



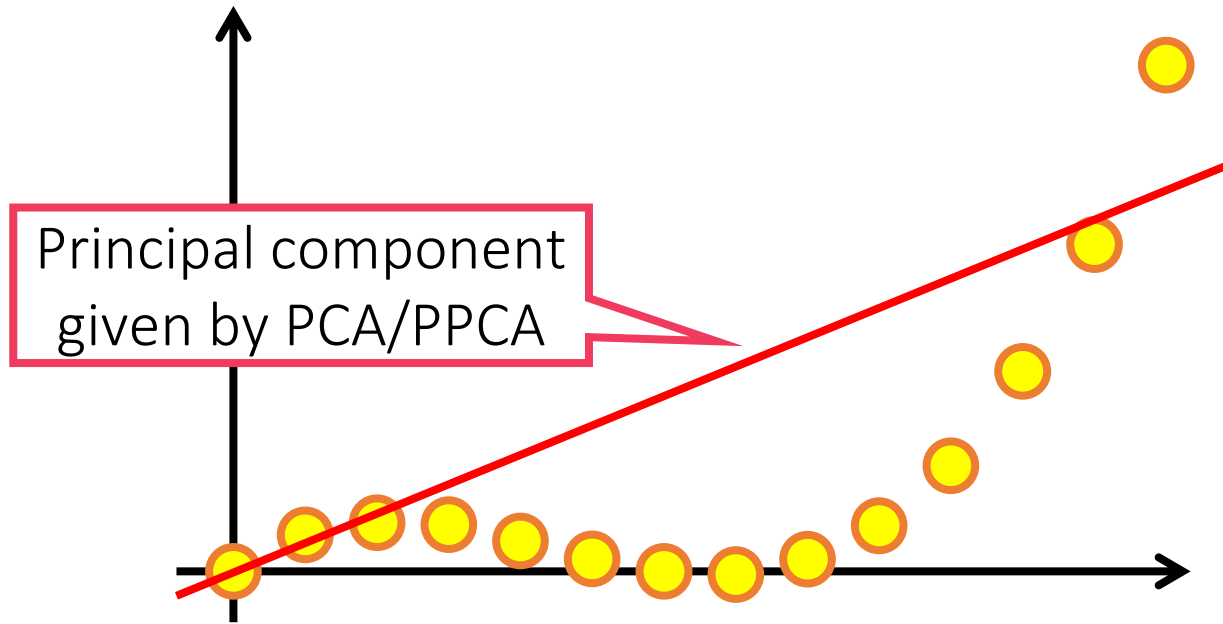
# Kernel PCA

22



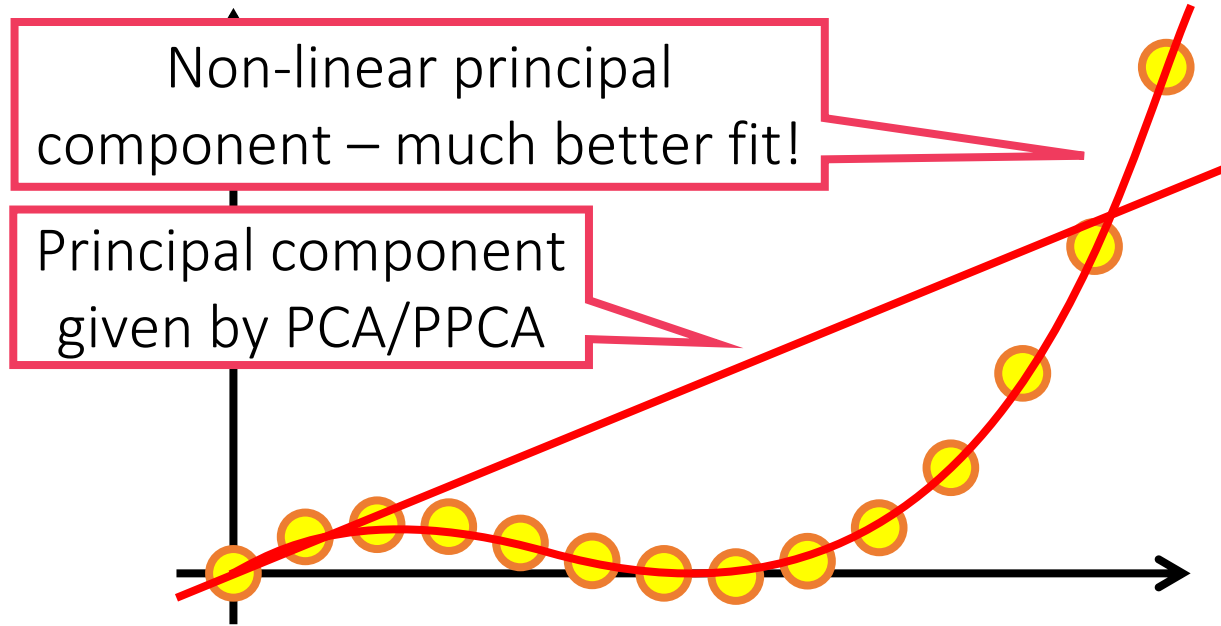
# Kernel PCA

22



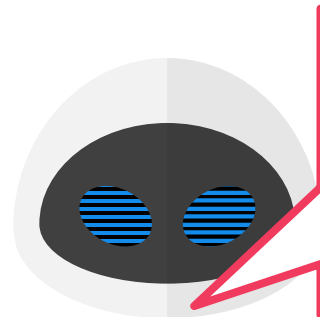
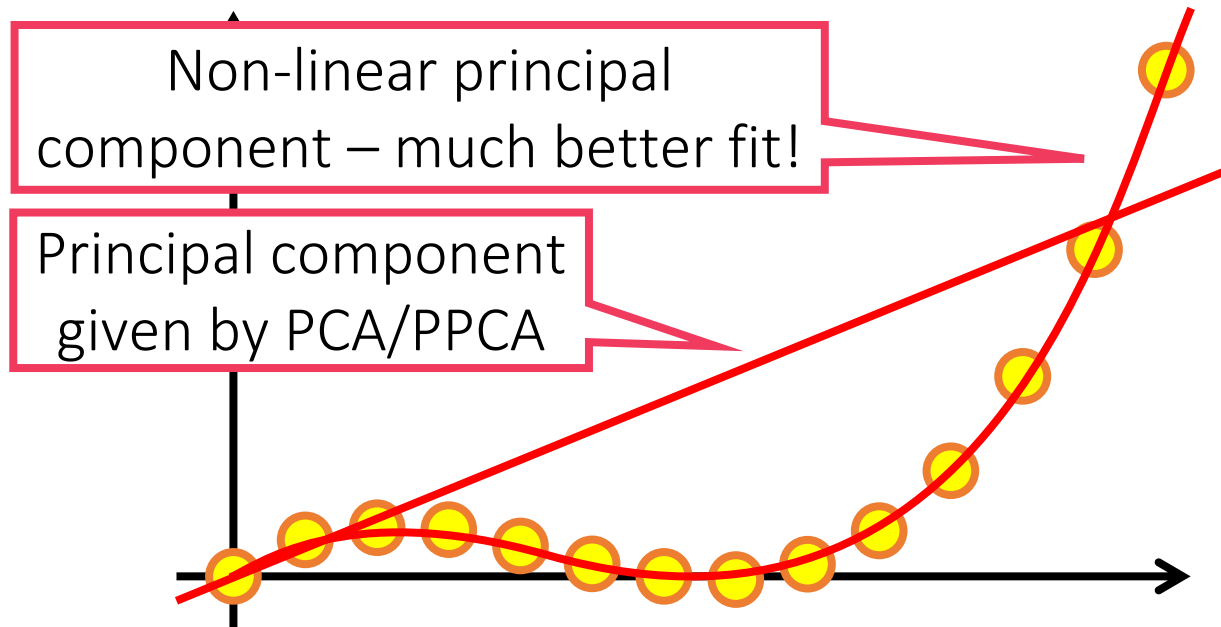
# Kernel PCA

22



# Kernel PCA

22

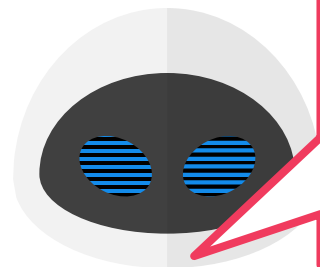
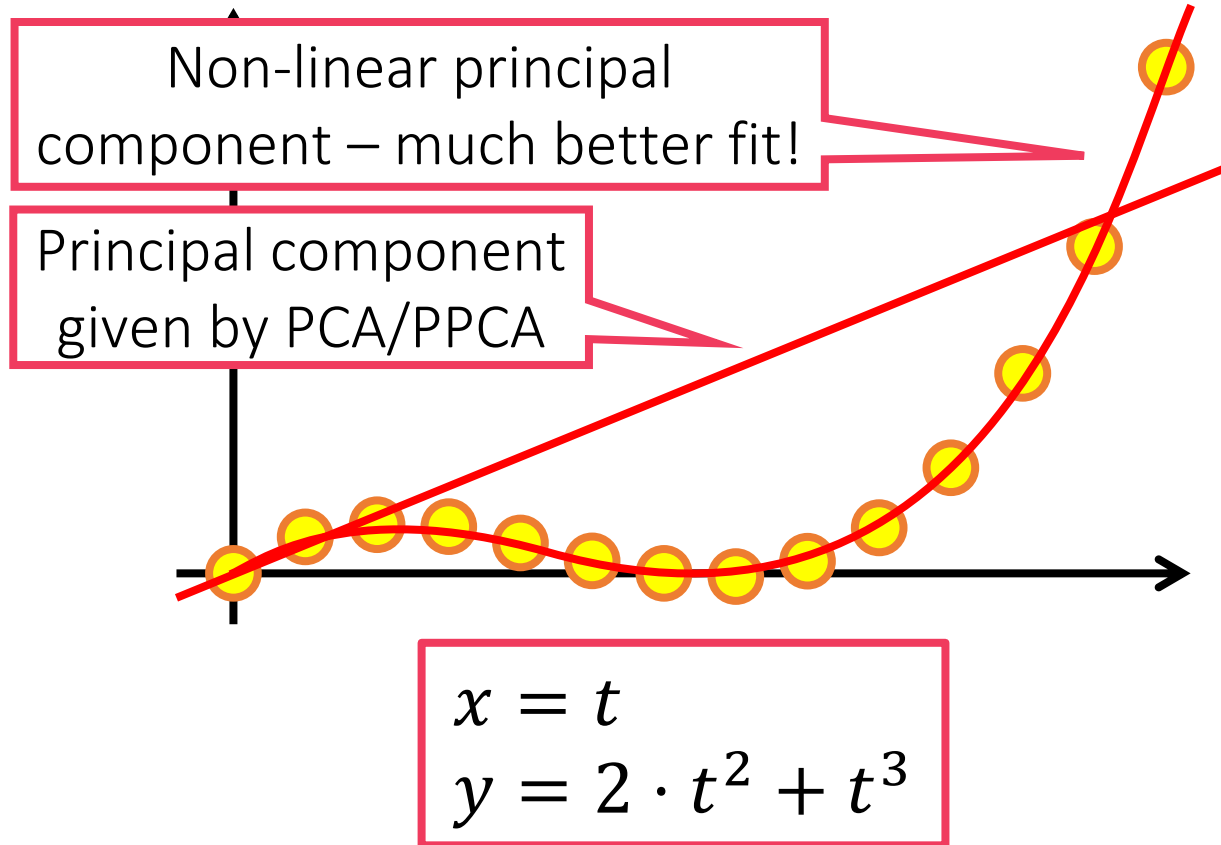


This data can actually be explained using a single variable! Even though the data looks 2D, it is inherently 1D in a nonlinear way. Kernel PCA can recover this hidden structure



# Kernel PCA

22

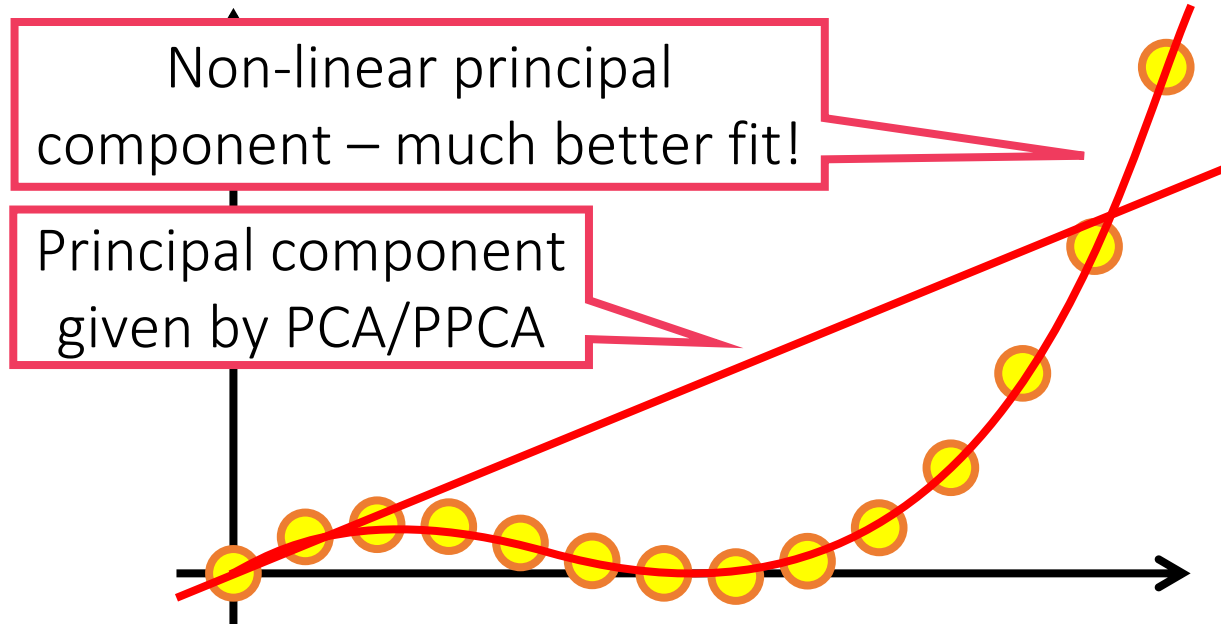


This data can actually be explained using a single variable! Even though the data looks 2D, it is inherently 1D in a nonlinear way. Kernel PCA can recover this hidden structure



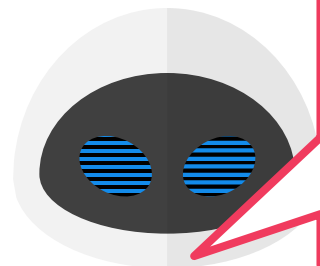
# Kernel PCA

22



$$\begin{aligned}x &= t \\ y &= 2 \cdot t^2 + t^3\end{aligned}$$

$$\mathbb{R}^2 \ni (x, y) \mapsto \phi(x, y) = [y, x^2, x^3] \in \mathbb{R}^3$$



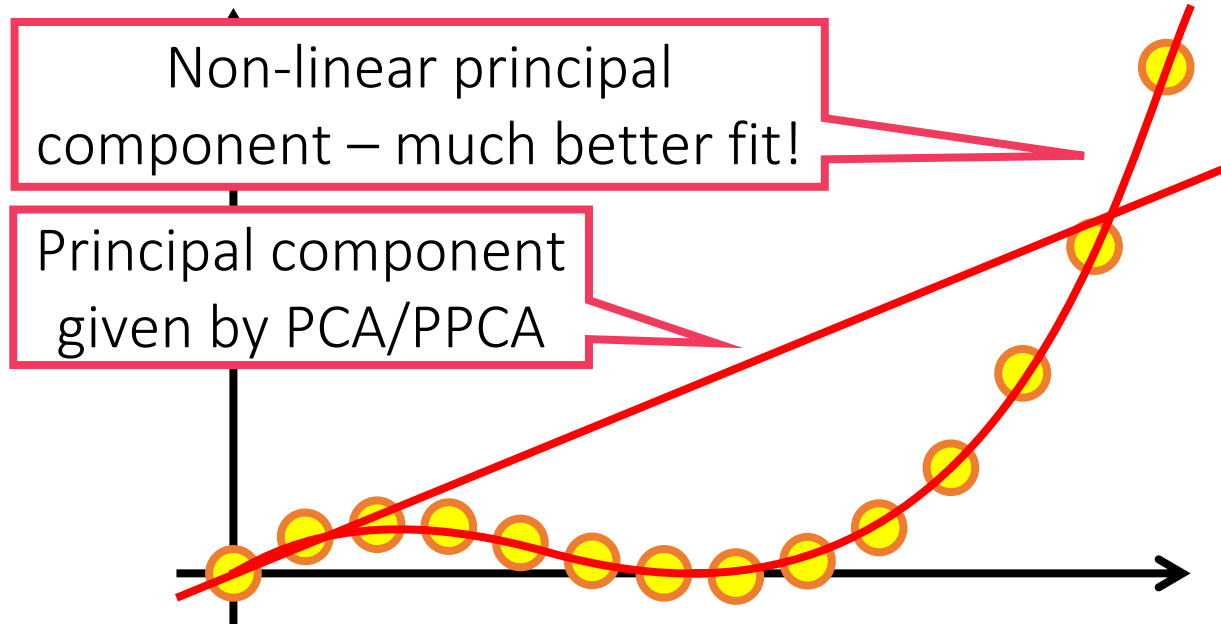
This data can actually be explained using a single variable! Even though the data looks 2D, it is inherently 1D in a nonlinear way. Kernel PCA can recover this hidden structure





# Kernel PCA

22



$$\begin{aligned} x &= t \\ y &= 2 \cdot t^2 + t^3 \end{aligned}$$

$$\mathbb{R}^2 \ni (x, y) \mapsto \phi(x, y) = [y, x^2, x^3] \in \mathbb{R}^3$$

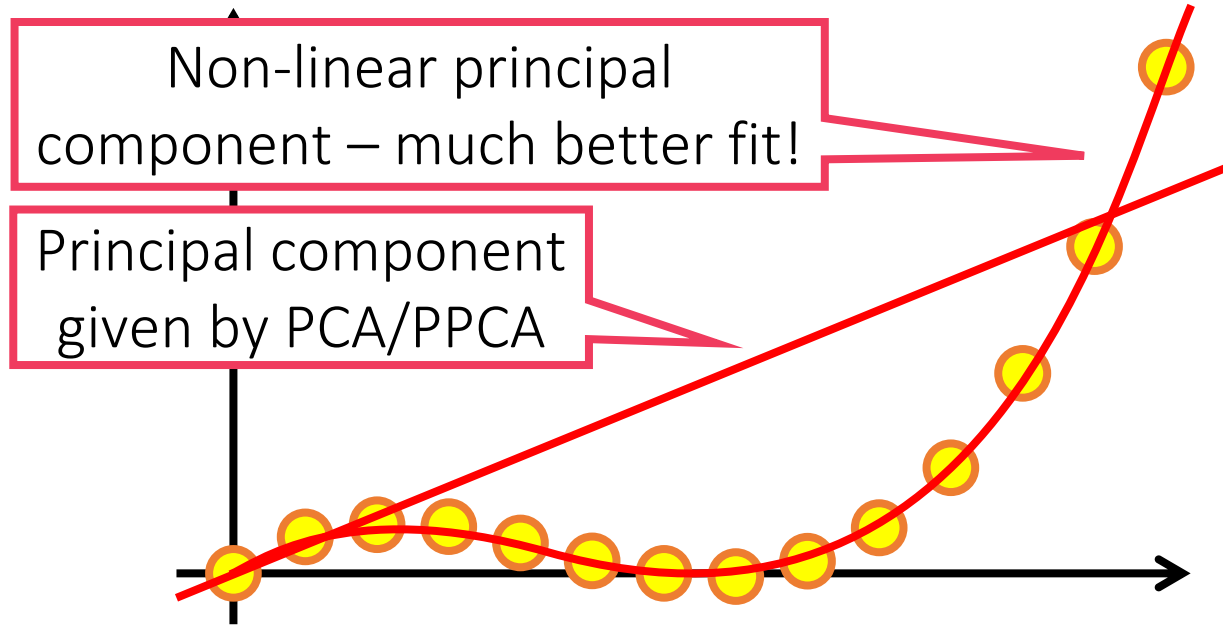
Use  $K_{\text{poly}}$  with  
 $p = 3, c = 1$

This data can actually be explained using a single variable! Even though the data looks 2D, it is inherently 1D in a nonlinear way. Kernel PCA can recover this hidden structure



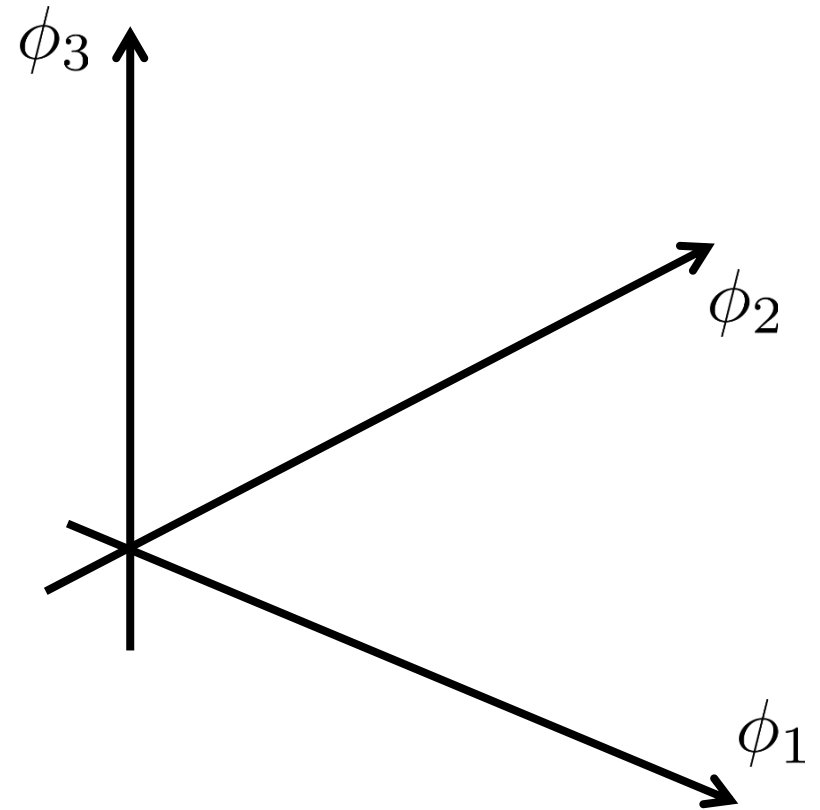
# Kernel PCA

22



$$\begin{aligned} x &= t \\ y &= 2 \cdot t^2 + t^3 \end{aligned}$$

$$\mathbb{R}^2 \ni (x, y) \mapsto \phi(x, y) = [y, x^2, x^3] \in \mathbb{R}^3$$



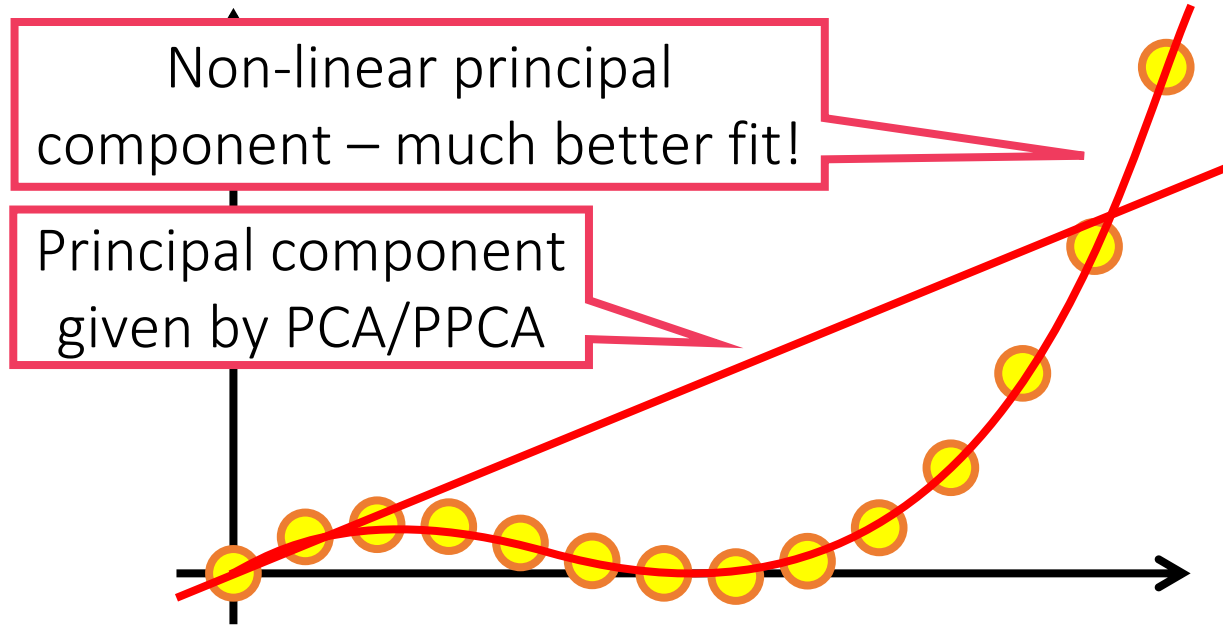
Use  $K_{\text{poly}}$  with  
 $p = 3, c = 1$

This data can actually be explained using a single variable! Even though the data looks 2D, it is inherently 1D in a nonlinear way. Kernel PCA can recover this hidden structure



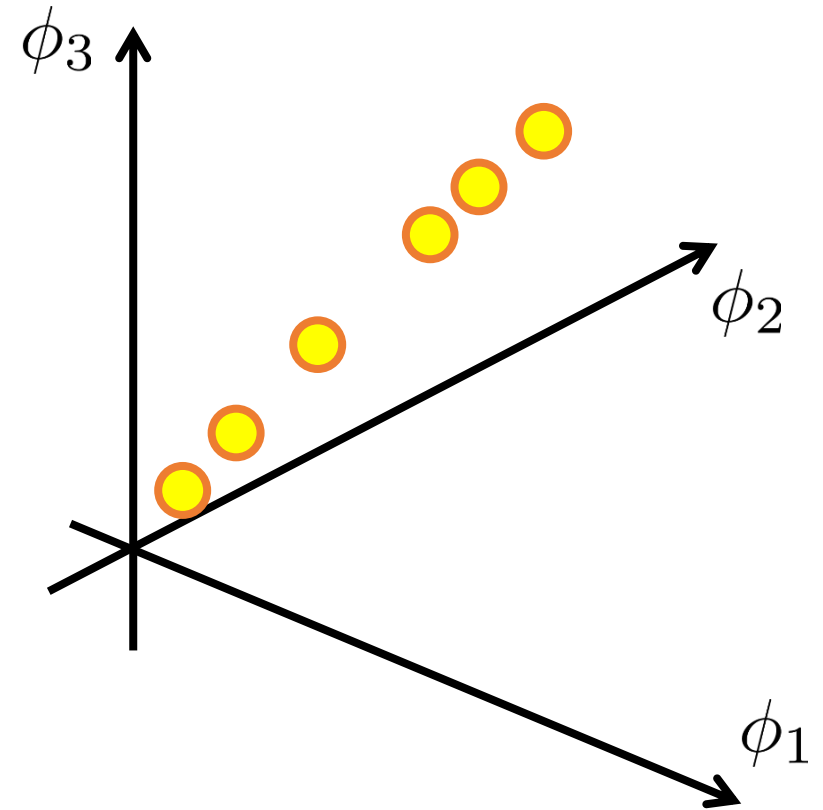
# Kernel PCA

22



$$\begin{aligned} x &= t \\ y &= 2 \cdot t^2 + t^3 \end{aligned}$$

$$\mathbb{R}^2 \ni (x, y) \mapsto \phi(x, y) = [y, x^2, x^3] \in \mathbb{R}^3$$



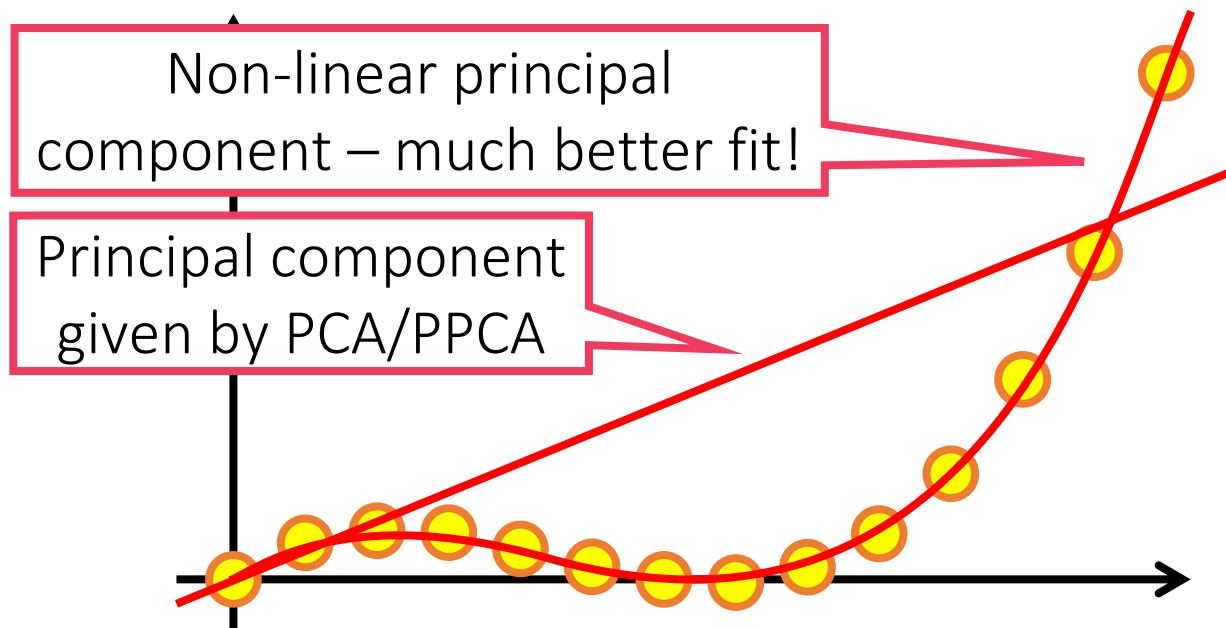
Use  $K_{\text{poly}}$  with  
 $p = 3, c = 1$

This data can actually be explained using a single variable! Even though the data looks 2D, it is inherently 1D in a nonlinear way. Kernel PCA can recover this hidden structure



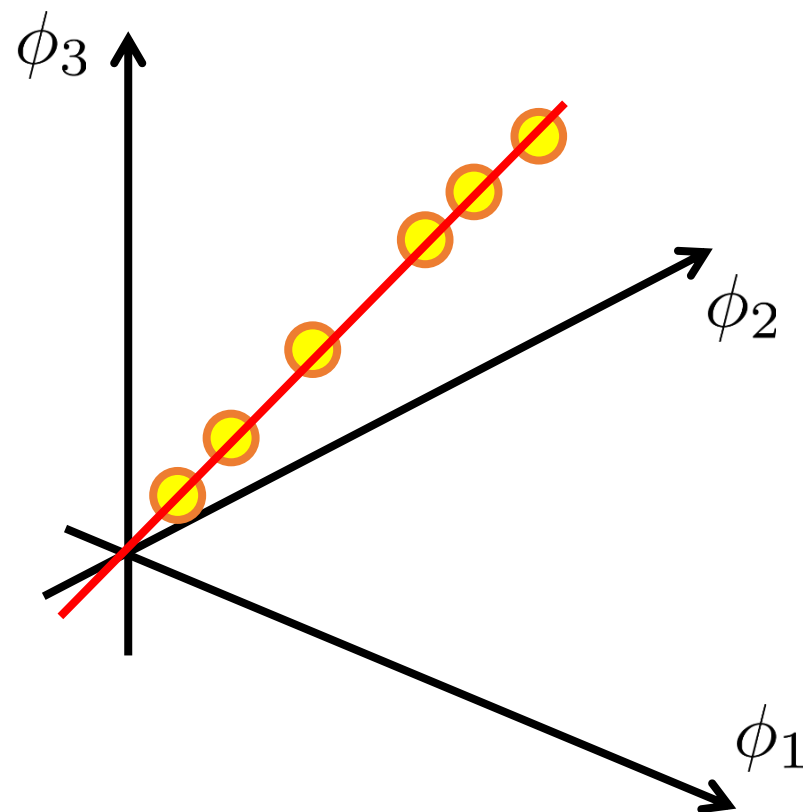
# Kernel PCA

22



$$\begin{aligned}x &= t \\ y &= 2 \cdot t^2 + t^3\end{aligned}$$

$$\mathbb{R}^2 \ni (x, y) \mapsto \phi(x, y) = [y, x^2, x^3] \in \mathbb{R}^3$$

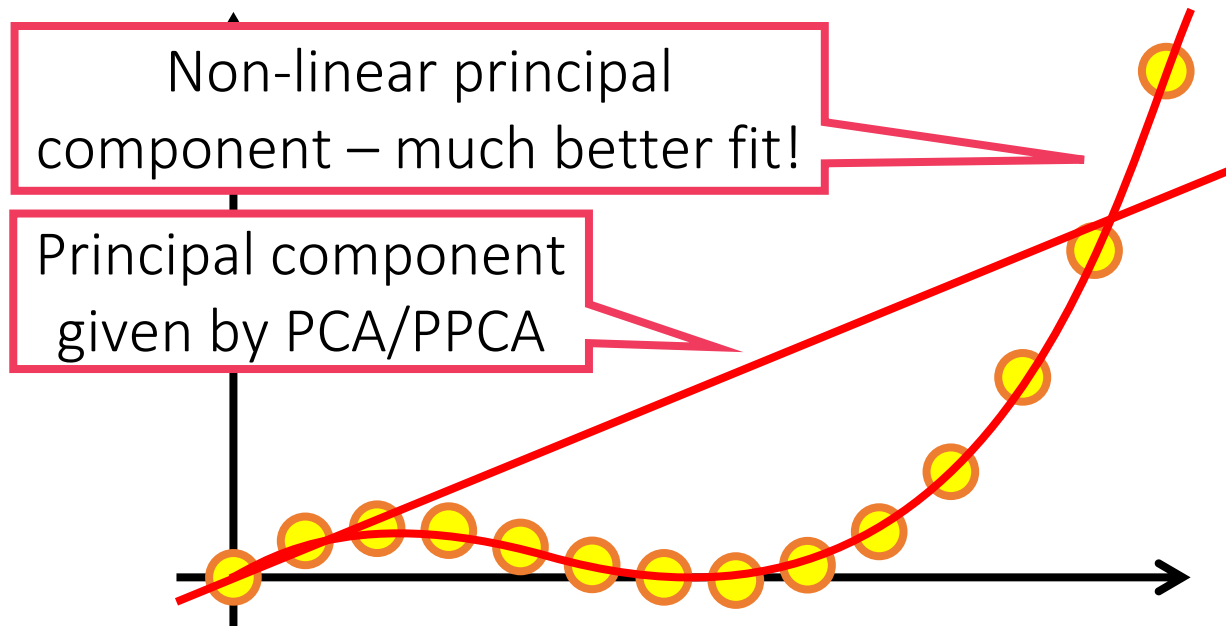


Use  $K_{\text{poly}}$  with  
 $p = 3, c = 1$

This data can actually be explained using a single variable! Even though the data looks 2D, it is inherently 1D in a nonlinear way. Kernel PCA can recover this hidden structure

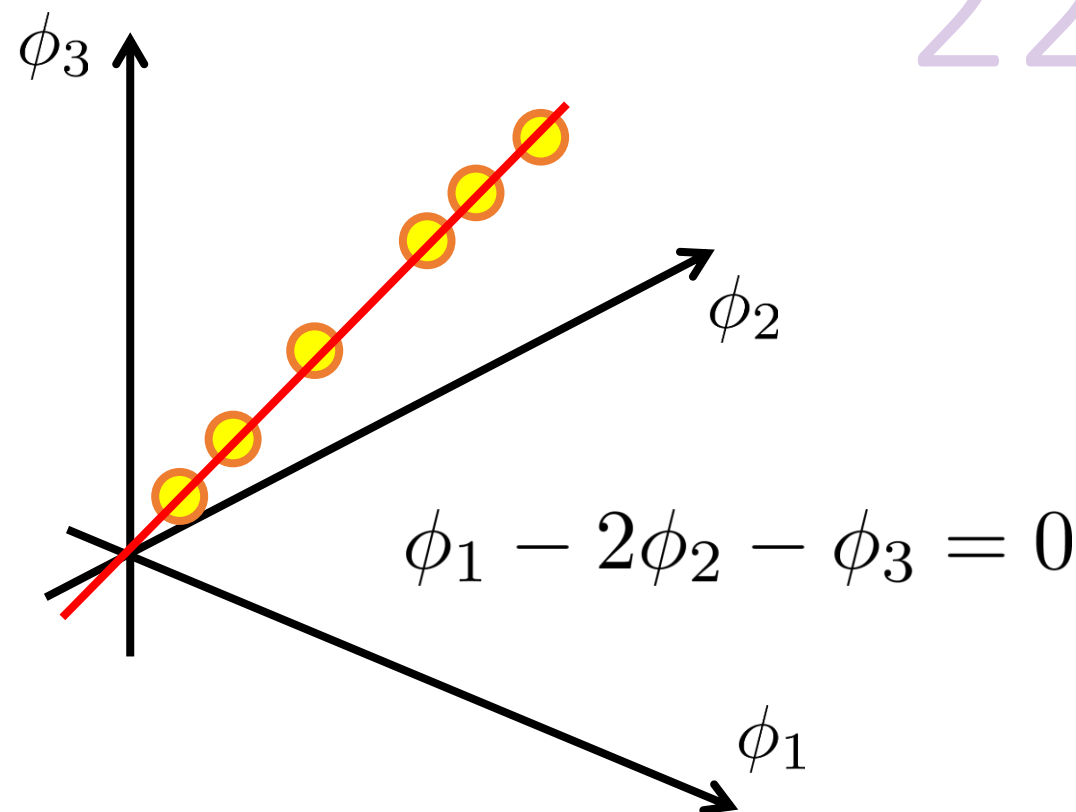
# Kernel PCA

22



$$\begin{aligned} x &= t \\ y &= 2 \cdot t^2 + t^3 \end{aligned}$$

$$\mathbb{R}^2 \ni (x, y) \mapsto \phi(x, y) = [y, x^2, x^3] \in \mathbb{R}^3$$



Use  $K_{\text{poly}}$  with  
 $p = 3, c = 1$

This data can actually be explained using a single variable! Even though the data looks 2D, it is inherently 1D in a nonlinear way. Kernel PCA can recover this hidden structure



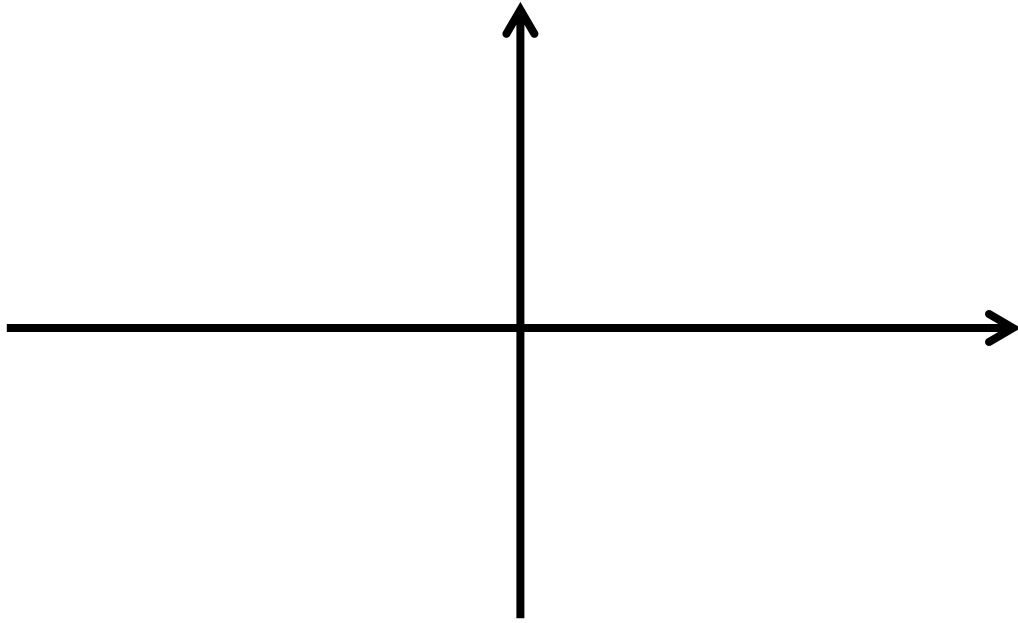
# Kernel PCA

35



# Kernel PCA

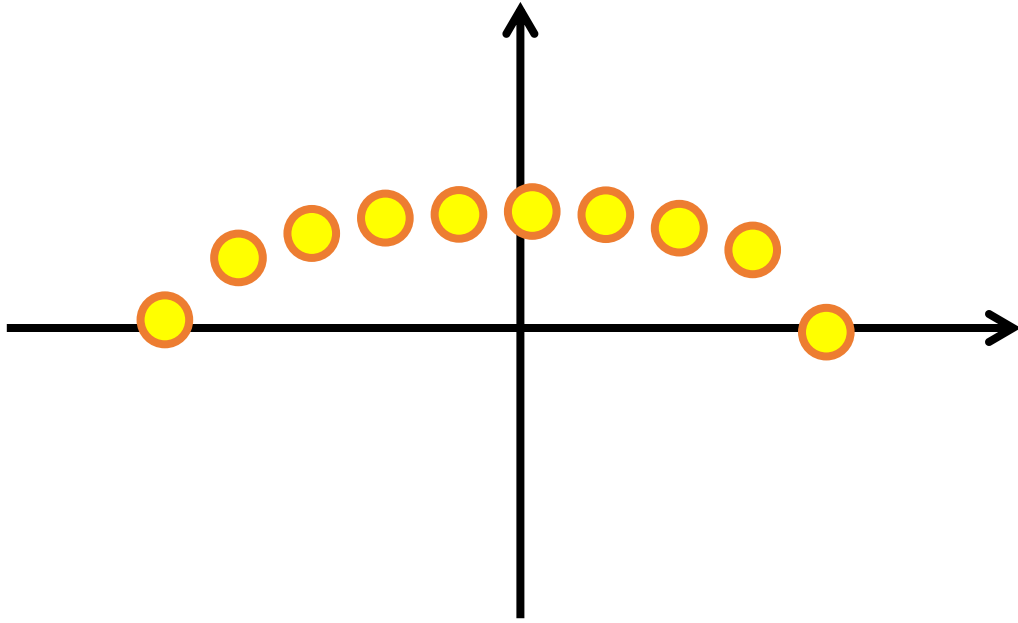
35





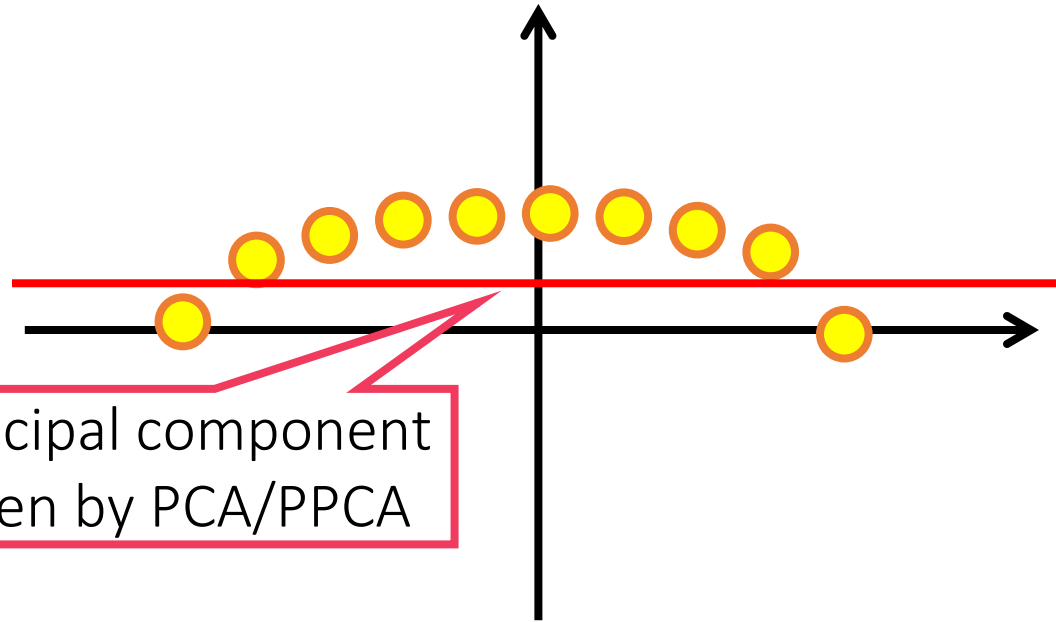
# Kernel PCA

35



# Kernel PCA

35



Principal component  
given by PCA/PPCA

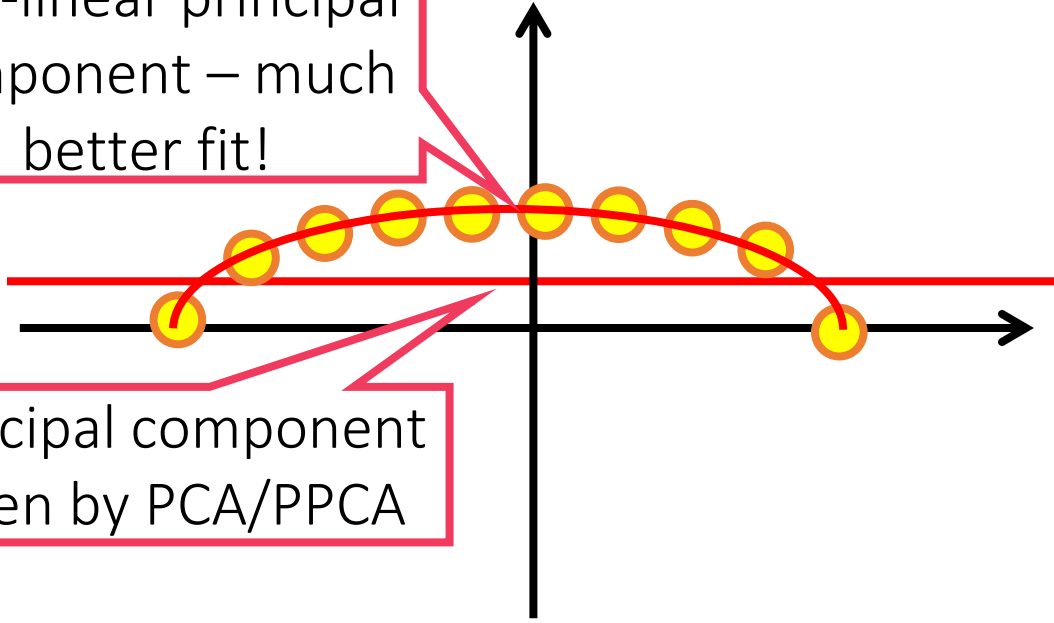


# Kernel PCA

35

Non-linear principal component – much better fit!

Principal component given by PCA/PPCA

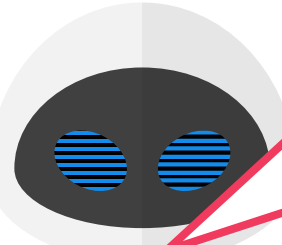
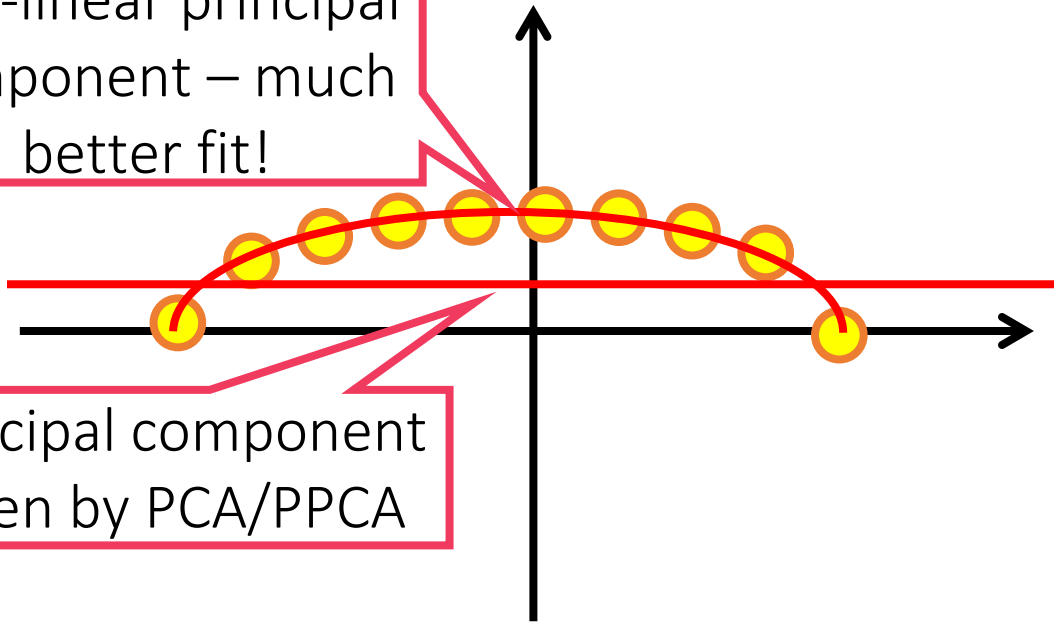


# Kernel PCA

35

Non-linear principal component – much better fit!

Principal component given by PCA/PPCA



Even this data can actually be explained using a single variable! Even though the data looks 2D, it is inherently 1D in a nonlinear way. Kernel PCA can recover this hidden structure

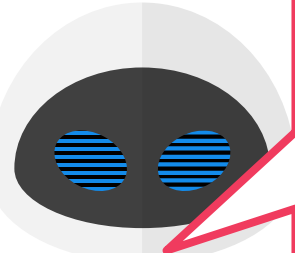


# Kernel PCA

Non-linear principal component – much better fit!

Principal component given by PCA/PPCA

$$x^2 + 4y^2 = 1$$



Even this data can actually be explained using a single variable! Even though the data looks 2D, it is inherently 1D in a nonlinear way. Kernel PCA can recover this hidden structure

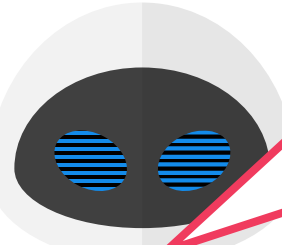
# Kernel PCA

Non-linear principal component – much better fit!

Principal component given by PCA/PPCA

$$x^2 + 4y^2 = 1$$

$$\begin{aligned}x^2 &= t \\ y^2 &= r\end{aligned}$$



Even this data can actually be explained using a single variable! Even though the data looks 2D, it is inherently 1D in a nonlinear way. Kernel PCA can recover this hidden structure

# Kernel PCA

35

Non-linear principal component – much better fit!

Principal component given by PCA/PPCA

$$x^2 + 4y^2 = 1$$

$$\begin{aligned} x^2 &= t \\ y^2 &= r \end{aligned}$$

$$\mathbb{R}^2 \ni (x, y) \mapsto \phi(x, y) = [x^2, y^2] \in \mathbb{R}^2$$

Even this data can actually be explained using a single variable! Even though the data looks 2D, it is inherently 1D in a nonlinear way. Kernel PCA can recover this hidden structure



# Kernel PCA

Non-linear principal component – much better fit!

Principal component given by PCA/PPCA

$$x^2 + 4y^2 = 1$$

$$\begin{aligned} x^2 &= t \\ y^2 &= r \end{aligned}$$

$$\mathbb{R}^2 \ni (x, y) \mapsto \phi(x, y) = [x^2, y^2] \in \mathbb{R}^2$$

Use  $K_{\text{quad}}$

Even this data can actually be explained using a single variable! Even though the data looks 2D, it is inherently 1D in a nonlinear way. Kernel PCA can recover this hidden structure





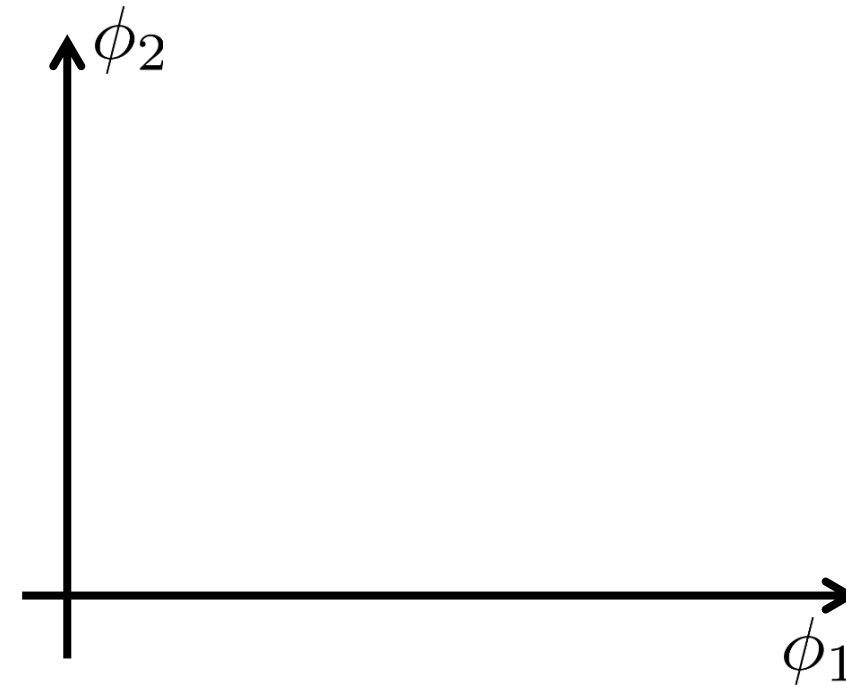
# Kernel PCA

Non-linear principal component – much better fit!

Principal component given by PCA/PPCA

$$x^2 + 4y^2 = 1$$

$$\begin{aligned} x^2 &= t \\ y^2 &= r \end{aligned}$$



$$\mathbb{R}^2 \ni (x, y) \mapsto \phi(x, y) = [x^2, y^2] \in \mathbb{R}^2$$

Use  $K_{\text{quad}}$

Even this data can actually be explained using a single variable! Even though the data looks 2D, it is inherently 1D in a nonlinear way. Kernel PCA can recover this hidden structure



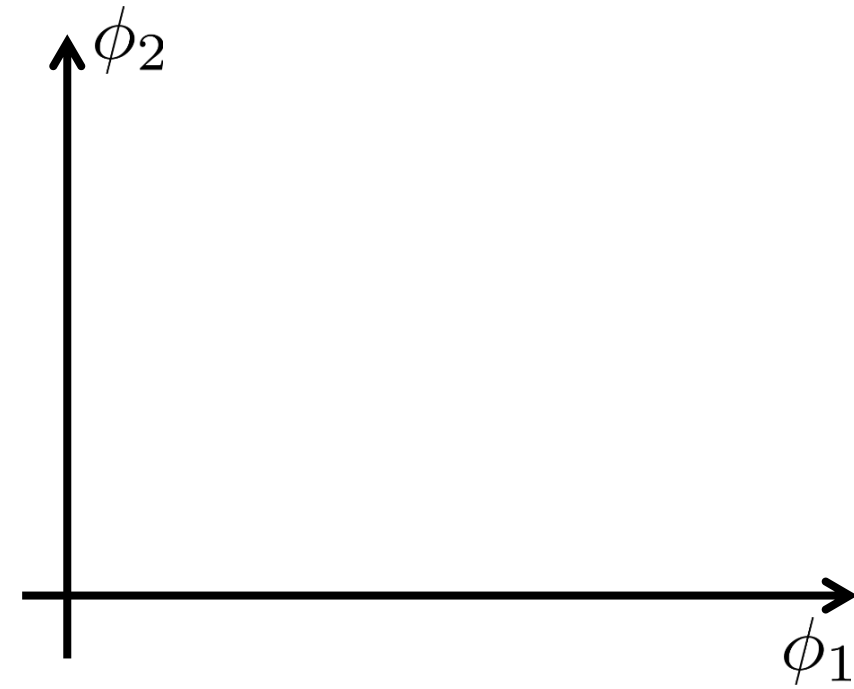
# Kernel PCA

Non-linear principal component – much better fit!

Principal component given by PCA/PPCA

$$x^2 + 4y^2 = 1$$

$$\begin{aligned} x^2 &= t \\ y^2 &= r \end{aligned}$$



$$\mathbb{R}^2 \ni (x, y) \mapsto \phi(x, y) = [x^2, y^2] \in \mathbb{R}^2$$

Use  $K_{\text{quad}}$

Even this data can actually be explained using a single variable! Even though the data looks 2D, it is inherently 1D in a nonlinear way. Kernel PCA can recover this hidden structure



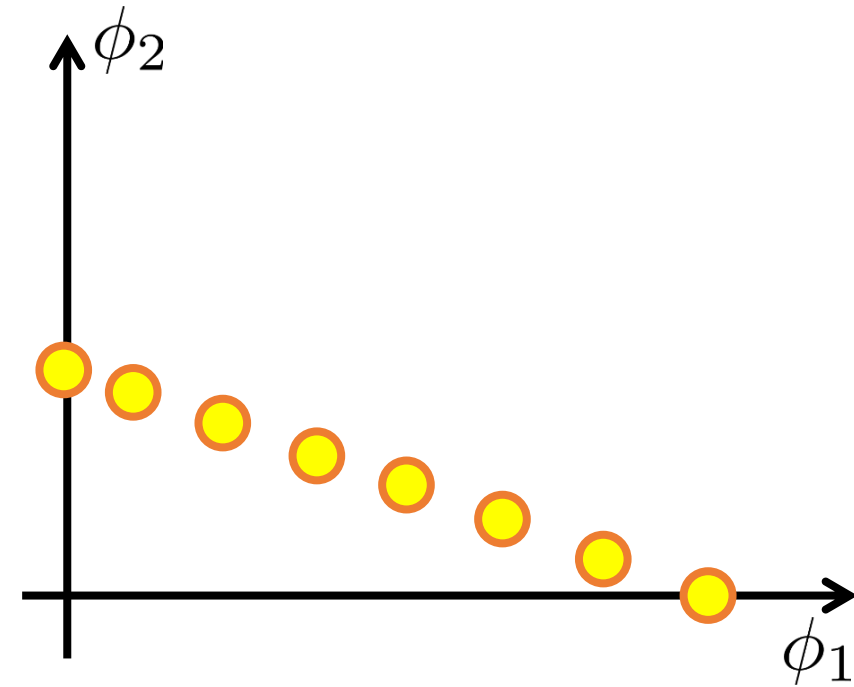
# Kernel PCA

Non-linear principal component – much better fit!

Principal component given by PCA/PPCA

$$x^2 + 4y^2 = 1$$

$$\begin{aligned} x^2 &= t \\ y^2 &= r \end{aligned}$$



$$\mathbb{R}^2 \ni (x, y) \mapsto \phi(x, y) = [x^2, y^2] \in \mathbb{R}^2$$

Use  $K_{\text{quad}}$

Even this data can actually be explained using a single variable! Even though the data looks 2D, it is inherently 1D in a nonlinear way. Kernel PCA can recover this hidden structure



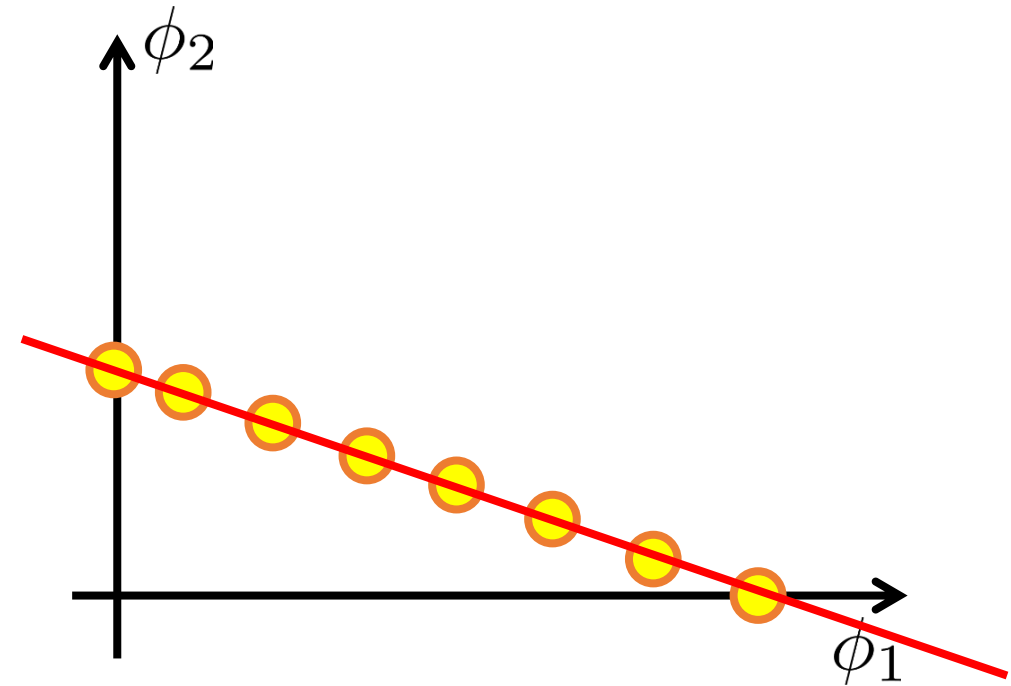
# Kernel PCA

Non-linear principal component – much better fit!

Principal component given by PCA/PPCA

$$x^2 + 4y^2 = 1$$

$$\begin{aligned} x^2 &= t \\ y^2 &= r \end{aligned}$$



$$\mathbb{R}^2 \ni (x, y) \mapsto \phi(x, y) = [x^2, y^2] \in \mathbb{R}^2$$

Use  $K_{\text{quad}}$

Even this data can actually be explained using a single variable! Even though the data looks 2D, it is inherently 1D in a nonlinear way. Kernel PCA can recover this hidden structure



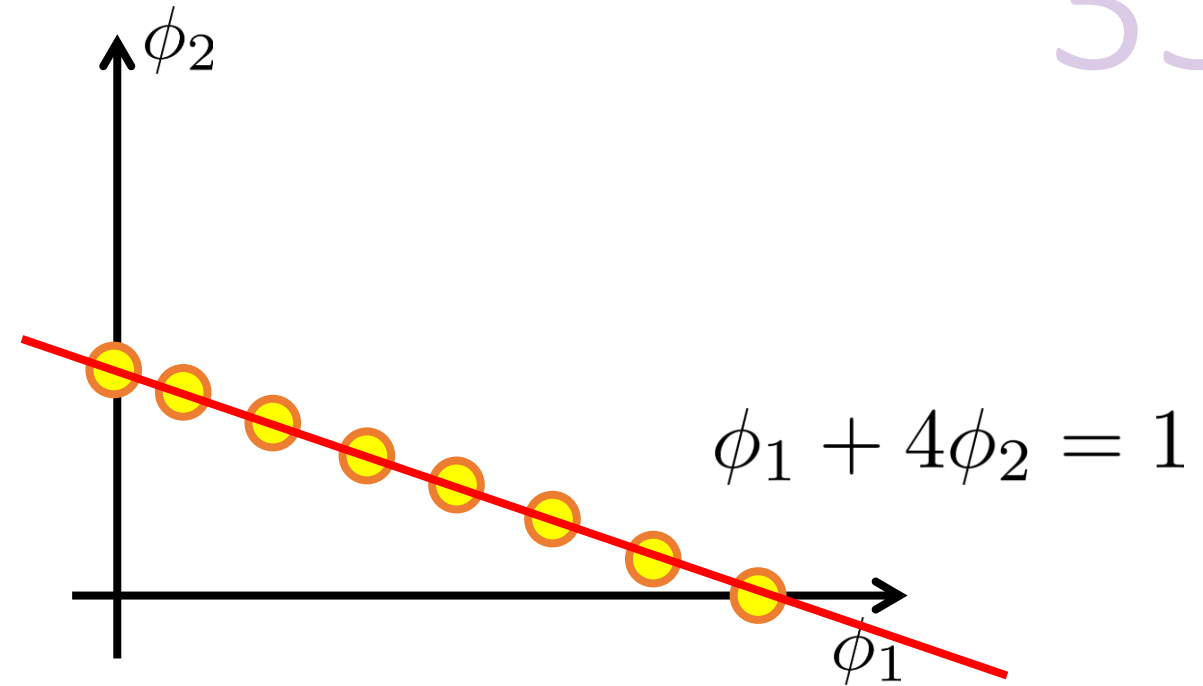
# Kernel PCA

Non-linear principal component – much better fit!

Principal component given by PCA/PPCA

$$x^2 + 4y^2 = 1$$

$$\begin{aligned} x^2 &= t \\ y^2 &= r \end{aligned}$$



$$\mathbb{R}^2 \ni (x, y) \mapsto \phi(x, y) = [x^2, y^2] \in \mathbb{R}^2$$

Use  $K_{\text{quad}}$

Even this data can actually be explained using a single variable! Even though the data looks 2D, it is inherently 1D in a nonlinear way. Kernel PCA can recover this hidden structure



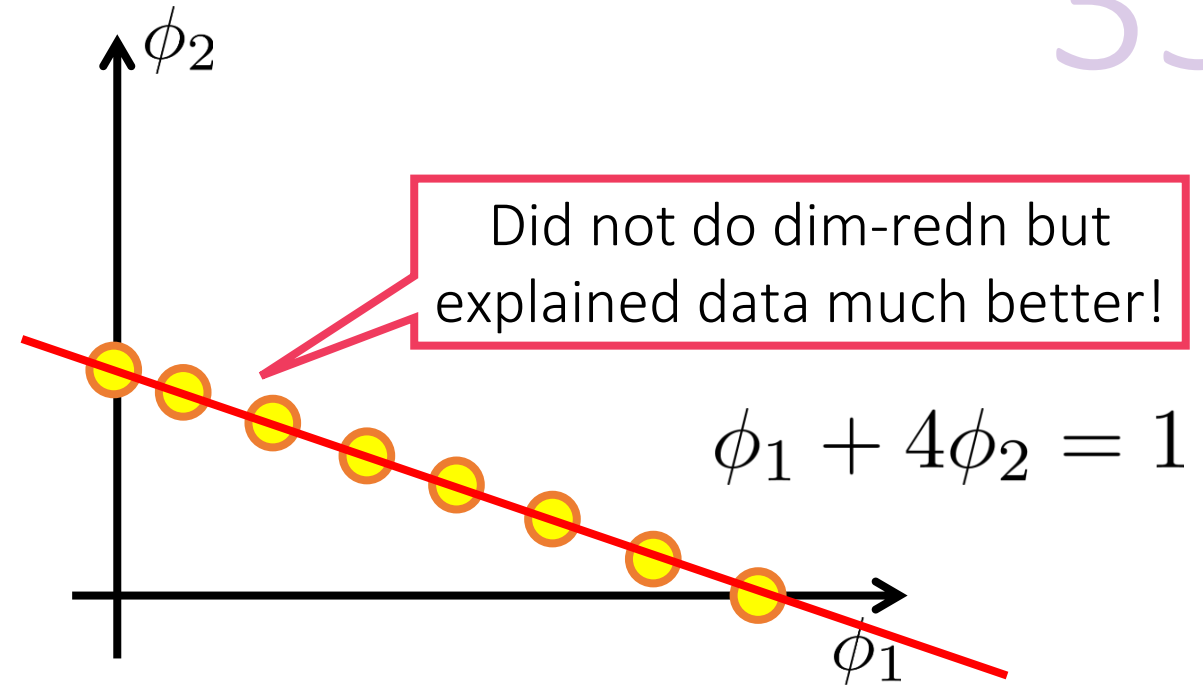
# Kernel PCA

Non-linear principal component – much better fit!

Principal component given by PCA/PPCA

$$x^2 + 4y^2 = 1$$

$$\begin{aligned} x^2 &= t \\ y^2 &= r \end{aligned}$$



$$\mathbb{R}^2 \ni (x, y) \mapsto \phi(x, y) = [x^2, y^2] \in \mathbb{R}^2$$

Use  $K_{\text{quad}}$

Even this data can actually be explained using a single variable! Even though the data looks 2D, it is inherently 1D in a nonlinear way. Kernel PCA can recover this hidden structure



Recall PCA found leading eigenvectors of matrix  $S = \frac{1}{n} \sum_{i=1}^n \mathbf{x}^i (\mathbf{x}^i)^\top$

*Kernel PCA simply does the same for  $M = \frac{1}{n} \sum_{i=1}^n \phi(\mathbf{x}^i) \phi(\mathbf{x}^i)^\top$*

*Notion of matrices does not make sense in  $\infty$ -dimensional settings so we instead talk about operators which are linear maps from  $\mathcal{H} \rightarrow \mathcal{H}$*

Recall that every matrix uniquely corresponds to a linear map so this does make sense

*Thus, instead of thinking about  $M$  as an  $\infty \times \infty$  matrix, think about it as a linear map that takes any vector  $\mathbf{h} \in \mathcal{H}$  and maps it linearly to another vector*

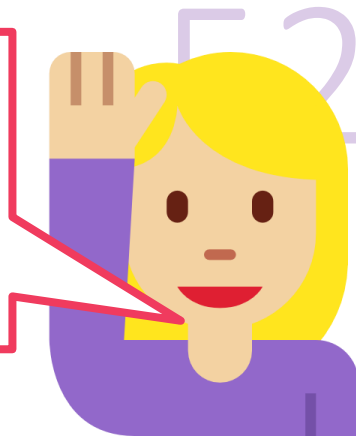
*$M\mathbf{h} \in \mathcal{H}$  as  $M\mathbf{h} = \frac{1}{n} \sum_{i=1}^n \phi(\mathbf{x}^i) \phi(\mathbf{x}^i)^\top \mathbf{h} = \frac{1}{n} \sum_{i=1}^n \left( \phi(\mathbf{x}^i)^\top \mathbf{h} \right) \cdot \phi(\mathbf{x}^i)$*

*Instead of a matrix having eigenvectors and eigenvalues, the operator  $M$  has eigenfunctions and eigenvalues. It may have infinitely many such eigenpairs!*

*However, the operator  $M$  is rank  $n$  where  $n$  is the number of training points*

Try to convince yourself of this in a toy setting where  $\phi(\mathbf{x}) \in \mathbb{R}^D$  where  $D \gg 1$  but finite

**Key** This means  $M$  has only at most  $n$  eigenfunctions with non-zero eigenvalues  
i.e. we should be able to write  $M = \sum_{i=1}^n \lambda_i \cdot \mathbf{v}^i (\mathbf{v}^i)^\top$  where  $\mathbf{v}^i \in \mathcal{H}$   
**Recall** and  $M\mathbf{v}^i = \lambda_i \cdot \mathbf{v}^i$  and  $(\mathbf{v}^i)^\top \mathbf{v}^j = 0$  for  $i \neq j$  and  $\|\mathbf{v}^i\|_{\mathcal{H}} = 1$  for all  $i$



*Kernel PCA simply does the same for  $M = \frac{1}{n} \sum_{i=1}^n \phi(\mathbf{x}^i) \phi(\mathbf{x}^i)^\top$*

*Notion of matrices does not make sense in  $\infty$ -dimensional settings so we instead talk about operators which are linear maps from  $\mathcal{H} \rightarrow \mathcal{H}$*

Recall that every matrix uniquely corresponds to a linear map so this does make sense

*Thus, instead of thinking about  $M$  as an  $\infty \times \infty$  matrix, think about it as a linear map that takes any vector  $\mathbf{h} \in \mathcal{H}$  and maps it linearly to another vector  $M\mathbf{h} \in \mathcal{H}$  as  $M\mathbf{h} = \frac{1}{n} \sum_{i=1}^n \phi(\mathbf{x}^i) \phi(\mathbf{x}^i)^\top \mathbf{h} = \frac{1}{n} \sum_{i=1}^n \left( \phi(\mathbf{x}^i)^\top \mathbf{h} \right) \cdot \phi(\mathbf{x}^i)$*

*Instead of a matrix having eigenvectors and eigenvalues, the operator  $M$  has eigenfunctions and eigenvalues. It may have infinitely many such eigenpairs!*

*However, the operator  $M$  is rank  $n$  where  $n$  is the number of training points*

Try to convince yourself of this in a toy setting where  $\phi(\mathbf{x}) \in \mathbb{R}^D$  where  $D \gg 1$  but finite



# Executing PCA in an RKHS $\mathcal{H}$

53

For sake of gaining intuition, consider a kernel  $K$  whose map  $\phi$  is very large dimensional but still maps to a finite dimensional space  $\mathcal{H} = \mathbb{R}^D$

*If  $\Phi = [\phi(\mathbf{x}^1), \dots, \phi(\mathbf{x}^n)]^\top \in \mathbb{R}^{n \times D}$  then  $M = \frac{1}{n} \Phi^\top \Phi = \frac{1}{n} \sum_{i=1}^n \phi(\mathbf{x}^i) \phi(\mathbf{x}^i)^\top$*

*We wish to kernelize PCA i.e. find an implicit representation for eigenfunctions*

*If  $\mathbf{v} \in \mathcal{H} = \mathbb{R}^D$  is an eigenfunc of  $M$  i.e.  $M\mathbf{v} = \lambda \cdot \mathbf{v}$ , then we can write  $\mathbf{v}$  as  $\mathbf{v} = \frac{1}{\lambda} \cdot M\mathbf{v} = \sum_{i=1}^n \alpha_i \cdot \phi(\mathbf{x}^i)$  where  $\alpha_i = \frac{1}{\lambda n} \phi(\mathbf{x}^i)^\top \mathbf{v}$  for  $i \in [n]$*

*We can rewrite this as  $\mathbf{v} = \Phi^\top \boldsymbol{\alpha}$*

*This is exactly what we wanted – a way to represent  $\mathbf{v}$  as linear comb of  $\phi(\mathbf{x}^i)$*

What kernel PCA does is instead of searching for eigenfuncs  $\mathbf{v} \in \mathcal{H}$ , search for coefficient vectors  $\boldsymbol{\alpha} \in \mathbb{R}^n$

*However, a few more steps still required to do so*



# Executing PCA in an RKHS $\mathcal{H}$

54

**Goal:** find  $\alpha \in \mathbb{R}^n$  such that  $\mathbf{v} = \Phi^\top \alpha$  is an eigenfunc of  $M = \frac{1}{n} \Phi^\top \Phi$

Rewrite  $M\mathbf{v} = \lambda \cdot \mathbf{v}$  as  $\Phi^\top \Phi \Phi^\top \alpha = \lambda n \cdot \Phi^\top \alpha$

*Still not in proper form since  $\Phi^\top \in \mathbb{R}^{D \times n}$  i.e. it is impossibly large dimensional*

*Recall that  $G = \Phi \Phi^\top$  is the Gram matrix of the training points*

*Multiplying both sides by  $\Phi$  gives  $\Phi \Phi^\top \Phi \Phi^\top \alpha = \lambda n \Phi \Phi^\top \alpha$  i.e.  $G^2 \alpha = \lambda n \cdot G \alpha$*

*Easy to see that eigenvectors of  $G$  will always satisfy the above equation*

*Slightly more careful argument required to show that the top eigenvectors of  $G$  will offer the largest values of  $\lambda$  – easy to see when  $G$  is invertible*

*So, in order to find  $\mathbf{v}$ , all we need to do is apply power + peeling method to obtain the top  $k$  eigenpairs of  $G$  in time  $\mathcal{O}(kn^2)$*

*Contrast this with linear PCA which took only  $\mathcal{O}(kd^2)$  time, yet again, the price of non-linearity. Model size for kernel PCA also larger  $\mathcal{O}(kn)$  for  $k$  components*



# Executing PCA in an RKHS $\mathcal{H}$

55

Perform PCA on Gram matrix  $G$  to obtain an eigenpair  $(\tilde{\lambda}, \boldsymbol{\alpha})$

*This (implicitly) gives us the eigenfunctions of  $M$  as  $\Phi^\top \boldsymbol{\alpha}$*

*Need to normalize the eigenfunction though since we may have  $\|\Phi^\top \boldsymbol{\alpha}\|_{\mathcal{H}} \neq 1$*

*Find  $\|\Phi^\top \boldsymbol{\alpha}\|_{\mathcal{H}}^2 = \boldsymbol{\alpha}^\top G \boldsymbol{\alpha} = \tilde{\lambda}$  and use  $\mathbf{v} = \frac{1}{\sqrt{\tilde{\lambda}}} \cdot \Phi^\top \boldsymbol{\alpha}$  as eigenfunction of  $M$*

We can also obtain new representations for data using kernel PCA

*In linear PCA we simply used  $W^\top \mathbf{x}$  as the new (low-dim) representation*

*Kernel PCA also offers a new  $k$ -dim rep if we find  $k$  eigenfuncs  $\mathbf{v}^1, \dots, \mathbf{v}^k \in \mathcal{H}$  using  $k$  eigenvectors  $\boldsymbol{\alpha}^1, \dots, \boldsymbol{\alpha}^k \in \mathbb{R}^n$  of  $G$*

*Given a new point  $\mathbf{x}$  compute  $\mathbf{z} \in \mathbb{R}^k$  where*

$$\mathbf{z}_i = \langle \mathbf{v}^i, \phi(\mathbf{x}) \rangle = \frac{1}{\sqrt{\tilde{\lambda}_i}} \cdot \sum_{j=1}^n \alpha_j^i \cdot K(\mathbf{x}, \mathbf{x}^j)$$



# A note on data centering

56

Recall that PCA performs well when data is mean centered since PCA fits a hyperplane that passes through the origin to the data

*Need to mean center data for kernel PCA as well for good performance*

*As before, all this needs to be done implicitly*

*Mean of data in RKHS is  $\boldsymbol{\mu} = \frac{1}{n} \cdot \sum_{i=1}^n \phi(\mathbf{x}^i)$ . We should perform all operations with mean centered data i.e.  $\phi(\mathbf{x}^i) - \boldsymbol{\mu}$*

*Since all we need to perform kernel PCA is the Gram matrix, need to find out what does the Gram matrix look like for the new centred data i.e.*

$$\tilde{G} = [\tilde{G}_{ij}], \tilde{G}_{ij} = \langle \phi(\mathbf{x}^i) - \boldsymbol{\mu}, \phi(\mathbf{x}^j) - \boldsymbol{\mu} \rangle$$

*Turns out this is available in closed form ( $\mathbf{1} = [1, \dots, 1] \in \mathbb{R}^n$ )*

$$\tilde{G} = \left( I_n - \frac{\mathbf{1}\mathbf{1}^\top}{n} \right) G \left( I_n - \frac{\mathbf{1}\mathbf{1}^\top}{n} \right)$$

*Simply use  $\tilde{G}$  in all kernel PCA operations*



## KERNEL PCA

1. Data points  $\mathbf{x}^1, \dots, \mathbf{x}^n \in \mathcal{X}$ , #components  $k \leq n$
2. Find centered Gram matrix  $\tilde{G} = \left(I_n - \frac{\mathbf{1}\mathbf{1}^\top}{n}\right) G \left(I_n - \frac{\mathbf{1}\mathbf{1}^\top}{n}\right)$
3. Find  $k$  largest eigenvectors/values of  $\tilde{G}$  as  $(\lambda_j, \boldsymbol{\alpha}^j)_{j=1, \dots, k}$
4. Let  $A = [\boldsymbol{\alpha}^1, \dots, \boldsymbol{\alpha}^k] \in \mathbb{R}^{n \times k}$  and  $\Lambda = \text{diag}(\lambda_1, \dots, \lambda_k) \in \mathbb{R}^{k \times k}$
5. For a point  $\mathbf{x} \in \mathcal{X}$ , find  $k$ -dim representation
  1. Let  $\mathbf{k}^{\mathbf{x}} \in \mathbb{R}^n$  such that  $\mathbf{k}_i^{\mathbf{x}} = K(\mathbf{x}^i, \mathbf{x})$
  2. Return  $\mathbf{z} = \sqrt{\Lambda^{-1}} A^\top \mathbf{k}^{\mathbf{x}} \in \mathbb{R}^k$

Whereas PCA could return only  $k \leq d$  components, KPCA can return upto  $k \leq n$  components – more data, more possible components!