Project Presentations Schedule

Feel free to use Piazza if you want to switch days with another group (just send me an email if you do so with cc to all students involved).

Tuesday, 3/1

1. Group 1 (**Hussain**, Tanvir; **Lewis**, Cameron; **Villamar**, Sandra)
2. Group 2 (**Dong**, Meng; **Long**, Jianzhi; **Wen**, Bo; **Zhang**, Haochen)
3. Group 3 (**Chen**, Yuzhao; **Li**, Zonghuan; **Song**, Yuze; **Yan**, Ge)
4. Group 4 (**Li**, Jiayuan; **Xiao**, Nan; **Yu**, Nancy; **Zhou**, Pei)
5. Group 5 (**Li**, Zheng; **Tao**, Jianyu; **Yang**, Fengqi)
6. Group 6 (**Bian**, Xintong; **Jiang**, Yufan; **Wu**, Qiyao)
7. Group 7 (**Chen**, Yongxing; **Yao**, Yanzhi; **Zhang**, Canwei)
8. Group 8 (**Nukala**, Kishore; **Pulleti**, Sai; **Vaidyula**, Srikar)

Thursday, 3/3

1. Group 9 (**Baluja**, Michael; **Cao**, Fangning; **Huff**, Mikael; **Shen**, Xuyang)
2. Group 10 (**Arun**, Aditya; **Long**, Heyang; **Peng**, Haonan)
3. Group 11 (**Cowin**, Samuel; **Liao**, Albert; **Mandadi**, Sumega)
4. Group 12 (**Jia**, Yichen; **Jiang**, Zhiyun; **Li**, Zhuofan)
5. Group 13 (**Dandu**, Murali; **Daru**, Srinivas; **Pamidi**, Sri)
6. Group 14 (**He**, Bolin; **Huang**, Yen-Ting; **Wang**, Shi; **Wang**, Tzu-Kao)
7. Group 15 (**Chen**, Luobin; **Feng**, Ruining; **Wu**, Ximei; **Xu**, Haoran)

Tuesday, 3/8

1. Group 16 (**Chen**, Rex; **Liang**, Youwei; **Zheng**, Xinran)
2. Group 17 (**Aguilar**, Matthew; **Millhiser**, Jacob; **O'Boyle**, John; **Sharpless**, Will)
3. Group 18 (**Wang**, Haoyu; **Wang**, Jiawei; **Zhang**, Yuwei)
4. Group 19 (**Chen**, Yinbo; **Di**, Zonglin; **Mu**, Jiteng)
5. Group 20 (**Chowdhury**, Debalina; **He**, Scott; **Ye**, Yiheng)
6. Group 21 (**Lin**, Wei-Ru; **Ru**, Liyang; **Zhang**, Shaohua)
7. Group 22 (**Bhavsar**, Shivad; **Blazej**, Christopher; **Bu**, Yinyan; **Liu**, Haozhe)

Thursday, 3/10

1. Group 23 (**Chen**, Claire; **Hsieh**, Chia-Wei; **Lin**, Jui-Yu; **Tsai**, Ya-Chen)
2. Group 24 (**Cheng**, Yu; **Yu**, Zhaowei; **Zaidi**, Ali)
3. Group 25 (**Assadi**, Parsa; **Brugere**, Tristan; **Pathak**, Nikhil; **Zou**, Yuxin)
4. Group 28 (**Candassamy**, Gokulakrishnan; **Dixit**, Rajeev; **Huang**, Joyce)
5. Group 27 (**Kok**, Hong; **Wang**, Jacky; **Yan**, Yijia; **Yuan**, Zhouyuan)
6. Group 28 (**Luan**, Zeting; **Yang**, Zheng)
7. Group 29 (**Cuawenberghs**, Kalyani; **Mojtahed**, Hamed)

Project Presentations

- You are HIGHLY encouraged to attend all days of the presentations (there will be **no recording**) – this is a chance to see different approaches and areas of application of what was covered in class in a setting similar to a conference.

- Each presentation will be allocated

  9 minutes (pts will be deducted if you go over 9 minutes)

- The presentation slides of ALL GROUPS (saved as pdf) are due by

  Do not include PIDs on the slides!

  Monday, 2/28 @ 11:59 pm

  The reason why **all groups** need to submit the slides is to make sure that they show the status of project at the start of all presentations (the slides that you submit are the ones that are going to be presented and evaluated). This is for fairness to all students, given the time differential between the first and last day of presentations.

  The presentation should discuss the **problem that you are trying to solve**, the **data that you are using**, the **proposed solution**(s), and the **results that you have so far** (they can later be UPDATED IN THE PROJECT PAPER).

- Email me the file (mvasconcelos@eng.ucsd.edu) and

  name the file GroupX.pdf, where X is your group number (see previous slide).

  Use Group X Presentation as the subject of your email and cc to all members.

ECE 271B – Winter 2022

Kernels

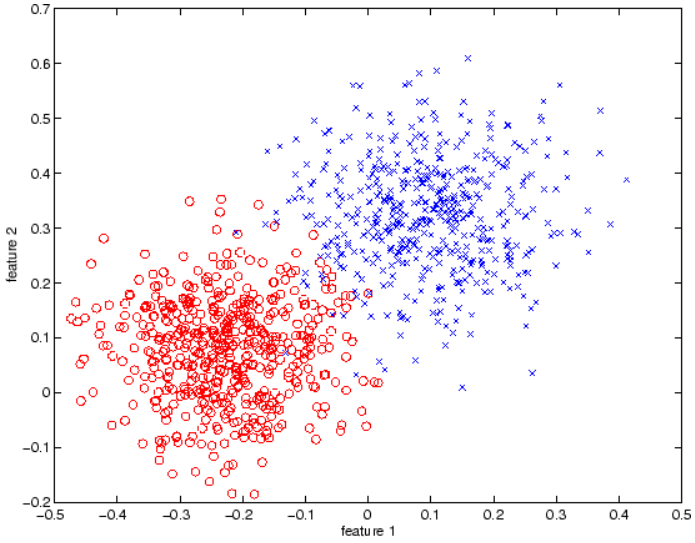
Disclaimer:
This class will be recorded
and made available to students asynchronously.

Manuela Vasconcelos

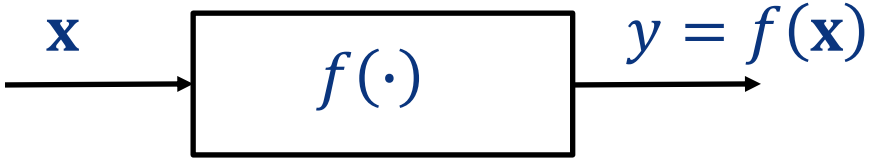
ECE Department, UCSD

Classification

- a **classification problem** has two types of variables
  - $\mathbf{x}$ – vector of observations (**features**) in the world
  - y – state (**class**) of the world
- e.g.
  - $\mathbf{x} \in \mathcal{X} \in \mathbb{R}^2 = (\text{fever}, \text{blood pressure})$
  - $\mathrm{y} \in \mathcal{Y} = \{\text{disease}, \text{no disease}\}$



- $\mathbf{x}$, y related by (unknown) function



- **goal**: design a **classifier** h: $\mathcal{X} \longrightarrow \mathcal{Y}$ such that $h(\mathbf{x}) = f(\mathbf{x}), \forall \mathbf{x}$

Perceptron

- **classifier** that implements the linear decision rule

$$h(\mathbf{x}) = \text{sgn}[g(\mathbf{x})] \quad \text{with} \quad g(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$$

- learning is formulated as an optimization problem

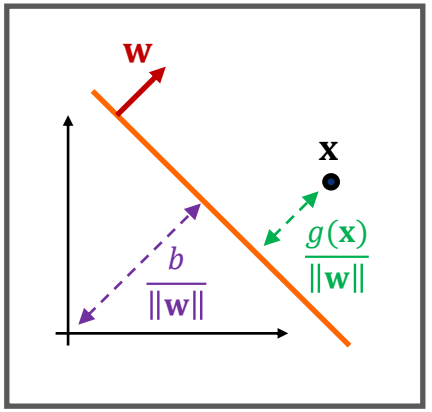
  - define set of errors

$$E = \{\mathbf{x}_i \mid y_i(\mathbf{w}^T\mathbf{x}_i + b) \leq 0\}$$

  - define the **cost**

$$J_P(\mathbf{w}, b) = -\sum_{i|\mathbf{x}_i \in E} y_i(\mathbf{w}^T\mathbf{x}_i + b)$$

  and **minimize**



Perceptron Learning

- is simply stochastic gradient descent on this cost:

set $k = 0, \mathbf{w}_k = 0, b_k = 0$

set $R = \max_i \|\mathbf{x}_i\|$

do {

  for $i = 1{:}n$ {

    if $y_i(\mathbf{w}_k{}^T \mathbf{x}_i + b_k) \leq 0$ then {

      - $\mathbf{w}_{k+1} = \mathbf{w}_k + \eta\, y_i\, \mathbf{x}_i$
      - $b_{k+1} = b_k + \eta\, y_i\, R^2$
      - $k = k + 1$

    }

  }

} until $y_i(\mathbf{w}^T \mathbf{x}_i + b_k) > 0, \forall i$ (no errors)

Perceptron Learning

- the interesting part is that this is guarantee to converge in **finite time**

- **Theorem:** Let $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ and

$$R = \max_i \|\mathbf{x}_i\|.$$

If there is $(\mathbf{w}^*, b^*)$ such that $\|\mathbf{w}^*\| = 1$ and

$$y_i\left(\mathbf{w}^{*T}\mathbf{x}_i + b^*\right) > \gamma, \forall i,$$

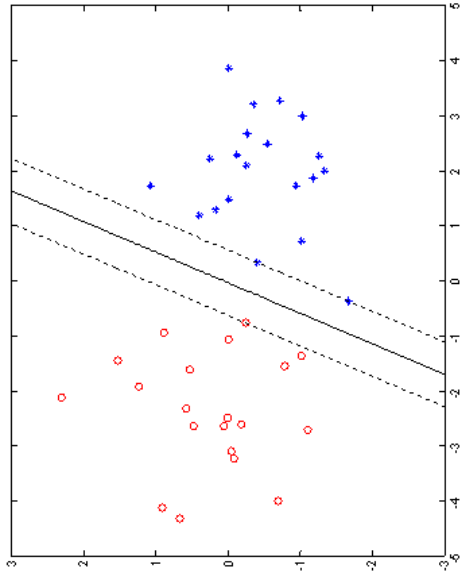
then the Perceptron will find an error free hyper−plane in at most

$$\left(\frac{2R}{\gamma}\right)^2 \text{ iterations}$$

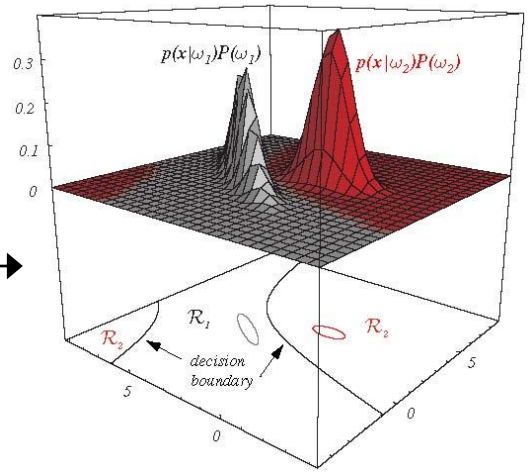
- the **main problem** is that it only implements a **linear discriminant**

Linear Discriminant

- Q: when is this a good decision function?
- clearly works if data is linearly separable
  - there is a plane which has
    - all -1's on one side
    - all 1's on the other



- 271A: it was also showed that it is optimal for
  - two Gaussian classes
  - equal class probability and covariance
- but, clearly, will **not** work even for only slightly more general Gaussian cases



- Q: what are possible **solutions** to this problem?

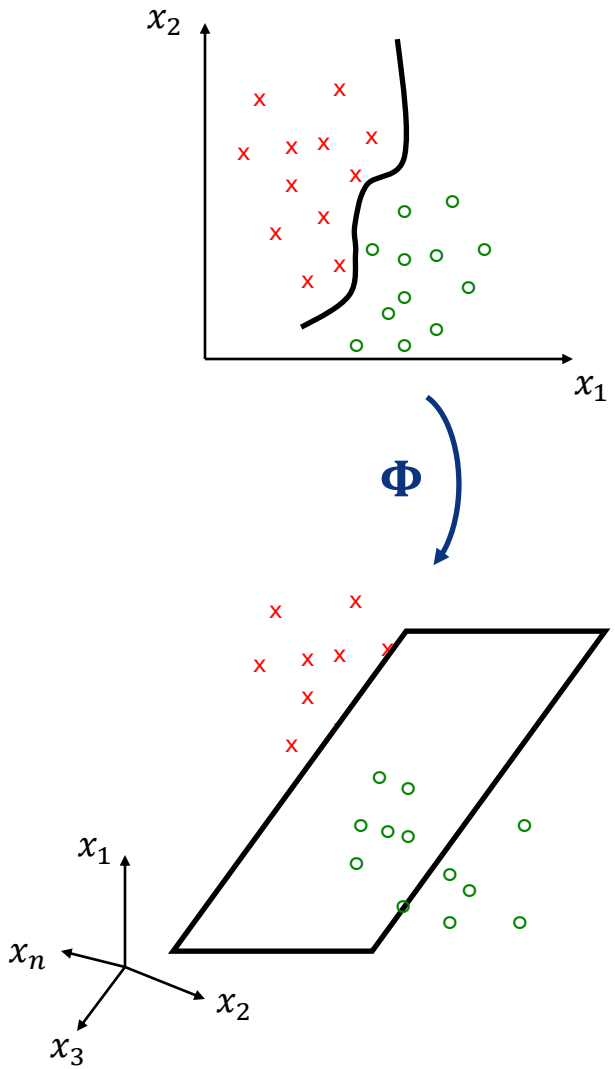
Alternatives

- 1) more complex classifier
  - let's try to avoid this
- 2) transform the space
  - introduce a mapping

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

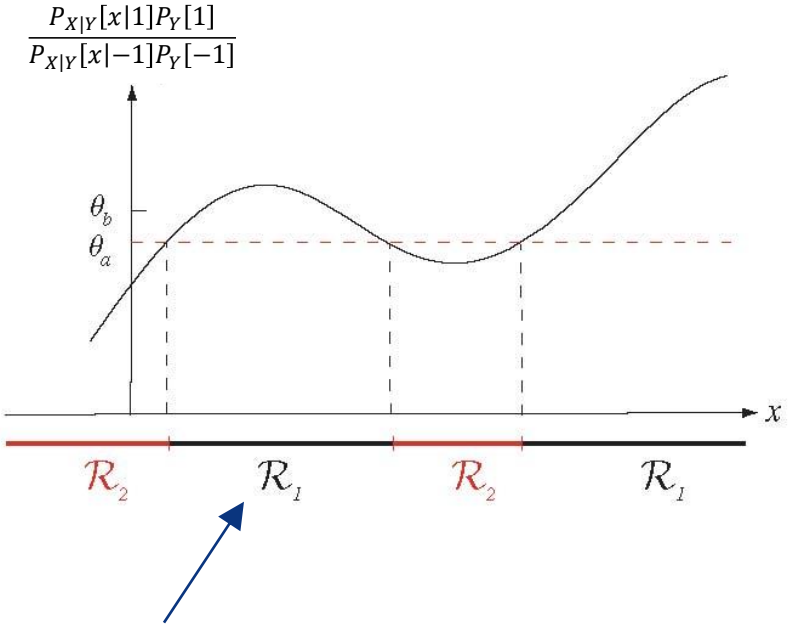
such that $\dim(\mathcal{Z}) > \dim(\mathcal{X})$

  - learning a **linear** boundary in $\mathcal{Z}$ is equivalent to learning a **non–linear** boundary in $\mathcal{X}$
  - **how do we do this?**
    - we already mentioned **three** possibilities



Solution One

- because the BDR is
  - pick $h(\mathbf{x}) = 1$ if
    $$\frac{P_{\mathbf{X}|Y}[\mathbf{x}|1]P_Y[1]}{P_{\mathbf{X}|Y}[\mathbf{x}|-1]P_Y[-1]} > 1$$
  - and $h(\mathbf{x}) = -1$, otherwise



- the **mapping**

$$\Phi_{BDR}: \mathbb{R}^d \to \mathbb{R}^{d+1} \text{ with } \Phi_{BDR}(\mathbf{x}) = \left(\mathbf{x}, \frac{P_{\mathbf{X}|Y}[\mathbf{x}|1]P_Y[1]}{P_{\mathbf{X}|Y}[\mathbf{x}|-1]P_Y[-1]}\right)$$

always works, since the hyperplane

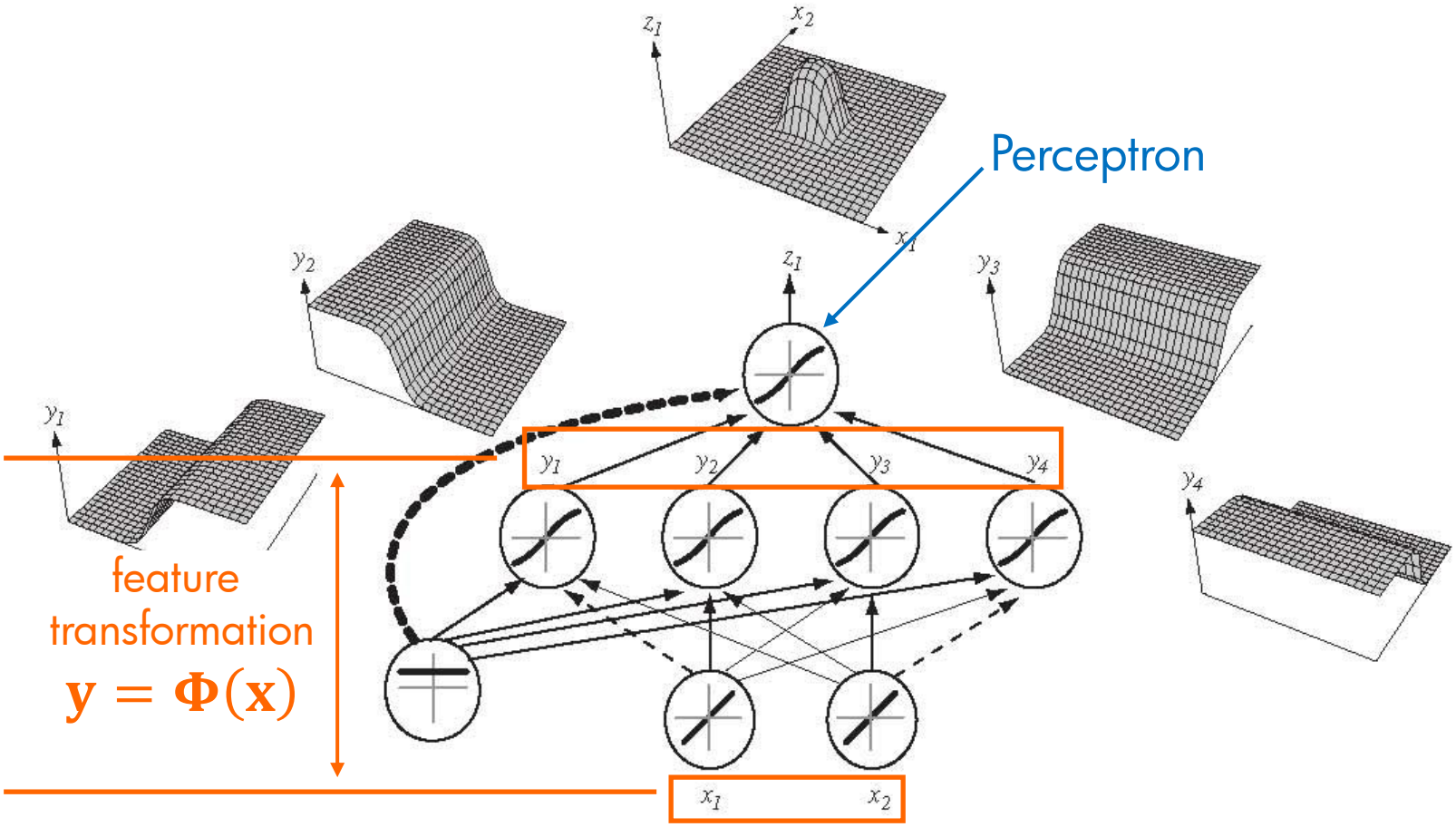
$$\mathbf{w}^T \Phi_{BDR}(\mathbf{x}) + b \quad \text{with} \quad \mathbf{w} = (0, 0, \cdots, 1)^T \text{ and } b = -1$$

optimally separates the classes

Solution Two

- add Perceptron layers:

MLP : non–linear feature transformation + linear discriminant

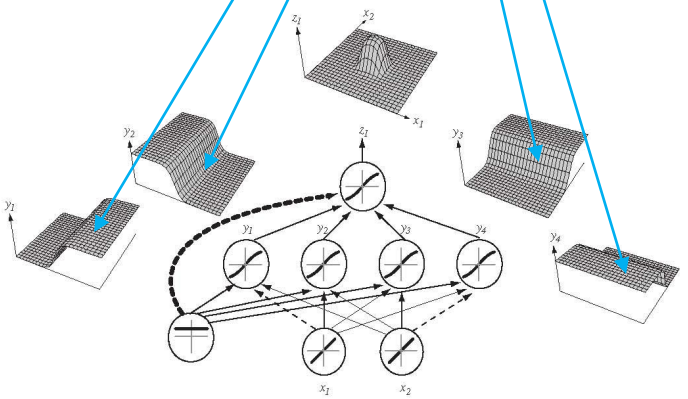


Solution Three

- use an **ensemble learner**

$$h(\mathbf{x}) = \text{sgn}[g(\mathbf{x})]$$

$$g(\mathbf{x}) = \sum_i w_i \alpha_i(\mathbf{x})$$



- the functions $\alpha_i(\mathbf{x})$ are called weak learners
- but very **similar** classifier to the MLP
- in the sense that the weak learners play a **similar** role to the MLP hidden units

Today's Alternative

$\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$

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

$\dim[\mathcal{X}] = d$
$\dim[\boldsymbol{\Phi}(\mathcal{X})] = k$

- so far, we have been trying to do this "**on the cheap**": from $\mathbb{R}^d$ to $\mathbb{R}^k$, where $k > d$ but **not** by much!
- let's **aim big** and make $k = \dim[\boldsymbol{\Phi}(\mathcal{X})]$ **really large**
- **intuitively**, for **larger** k,
  - it will be easier to separate the classes linearly, i.e. the set of mappings that achieves **linear separation** grows
- here is a **bold plan**:
  - let's pick $\boldsymbol{\Phi}(\mathcal{X})$ randomly
  - as $k \rightarrow \infty$, the probability that this make the data **linearly separable** increases
- where do we **stop?**
  - well, we can go **all** the way to $k = \infty$
  - i.e., we map **each point** into a **function**

$$\boldsymbol{\Phi}(\mathbf{x}) = \big(\phi_1(\mathbf{x}), \ldots, \phi_k(\mathbf{x})\big) \xrightarrow[k \rightarrow \infty]{} \phi(\mathbf{x}; t)$$

$\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$

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

$\dim[\mathcal{X}] = d$
$\dim[\boldsymbol{\Phi}(\mathcal{X})] = k$

Implementation

- at first, this looks like a **bad** idea

set $k = 0, \mathbf{w}_k = 0, b_k = 0$

set $R = \max_i \|\mathbf{x}_i\|$

do {

  for $i = 1{:}n$ {

    if $y_i\left(\mathbf{w}_k^T \boldsymbol{\Phi}(\mathbf{x}_i) + b_k\right) \leq 0$ then

- $\mathbf{w}_{k+1} = \mathbf{w}_k + \eta\, y_i\, \mathbf{x}_i$
- $b_{k+1} = b_k + \eta\, y_i\, R^2$
- $k = k + 1$

    }

  }

} until $y_i(\mathbf{w}_k{}^T \boldsymbol{\Phi}(\mathbf{x}_i) + b_k) > 0, \forall i$ (no errors)

how do we

- compute $\mathbf{w}_k^T \boldsymbol{\Phi}(\mathbf{x}_i)$
- store $\boldsymbol{\Phi}(\mathbf{x}_i)$

as $k \rightarrow \infty$????

The Dot–Product Implementation

- this turns out to be **possible** when the learning algorithm can be written in "dot–product" form

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

$$\mathcal{D} = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)\},$$

it only depends on the points $\mathbf{x}_i$ through their **dot–products**

$$\mathbf{x}_i^T \mathbf{x}_j .$$

- we will see that, luckily, this is a natural representation for many optimization procedures

- the **Perceptron** learning algorithm can be written in this form quite easily (Quiz #2 – Prob. 4)

$\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$

$\boldsymbol{\Phi}: \mathcal{X} \to \mathcal{Z}$

$\dim[\mathcal{X}] = d$
$\dim[\boldsymbol{\Phi}(\mathcal{X})] = k$

Perceptron Learning

- in dot–product form: (Quiz #2 – Prob. 4)

set $\alpha_i = 0,\ b = 0$

set $R = \max_i \|\mathbf{x}_i\|$

do {

  for $i = 1{:}n$ {

    if $y_i\left(\sum_{j=1}^{n} \alpha_j y_j \mathbf{x}_j^T \mathbf{x}_i + b\right) \leq 0$ then {

      - $\alpha_i = \alpha_i + 1$
      - $b = b + \eta\, y_i\, R^2$

    }

  }

} until no errors

Notes:

- in original form, we update $\mathbf{w} \in \mathbb{R}^d$
- in dot–product form, we update $\boldsymbol{\alpha} \in \mathbb{R}^n$
- note that n is the training set size and, in general, $n \gg d$
- at first look, does **not** appear very productive
- the benefits only become visible when we introduce $\boldsymbol{\Phi}$

$\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$

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

$\dim[\mathcal{X}] = d$
$\dim[\boldsymbol{\Phi}(\mathcal{X})] = k$

Perceptron Learning

- in range space:

set $\alpha_i = 0,\ b = 0$

set $R = \max_i \|\mathbf{x}_i\|$

do {

  for $i = 1{:}n$ {

    if $y_i \left(\sum_{j=1}^{n} \alpha_j y_j \boldsymbol{\Phi}(\mathbf{x}_j)^T \boldsymbol{\Phi}(\mathbf{x}_i) + b \right) \leq 0$ then {

- $\alpha_i = \alpha_i + 1$
- $b = b + \eta\, y_i\, R^2$

    }

  }

} until no errors

Notes:

- only requires dot–products $\boldsymbol{\Phi}(\mathbf{x}_j)^T \boldsymbol{\Phi}(\mathbf{x}_i)$
- no longer a need to store the $\boldsymbol{\Phi}(\mathbf{x}_j)$: only the n^2 dot–product matrix
- when $\dim[\boldsymbol{\Phi}(\mathbf{x}_j)]$ is infinite, this is significant

- what about the computation of the dot–products $\boldsymbol{\Phi}(\mathbf{x}_j)^T \boldsymbol{\Phi}(\mathbf{x}_i)$?

when $\boldsymbol{\Phi}(\mathbf{x}_j)$ is infinite-dimensional, the computation looks impossible…

The "Kernel Trick"

- "instead of defining $\boldsymbol{\Phi}(\mathcal{X})$, computing $\boldsymbol{\Phi}(\mathbf{x}_i)$ for each i and $\boldsymbol{\Phi}(\mathbf{x}_i)^T\boldsymbol{\Phi}(\mathbf{x}_j)$ for each pair (i,j), simply **define** the function

$$K(\mathbf{x},\mathbf{z}) = \boldsymbol{\Phi}(\mathbf{x})^T\boldsymbol{\Phi}(\mathbf{z})$$

and work with it **directly**"

$K(\mathbf{x},\mathbf{z})$ is called a **dot-product kernel**

- in fact, since we **only** use the kernel, **why define $\boldsymbol{\Phi}(\mathcal{X})$?**
  - just define the kernel $K(\mathbf{x},\mathbf{z})$ **directly**!
  - in this way, we never have to deal with the complexity of $\boldsymbol{\Phi}(\mathcal{X})$...
  - this is usually called the **"kernel trick"**

Questions

- I am confused!
- how do I know that if I pick a function $K(\mathbf{x}, \mathbf{z})$, it is equivalent to $\boldsymbol{\Phi}(\mathbf{x})^T \boldsymbol{\Phi}(\mathbf{z})$?
  - in general, it is **not** (we will talk about this later)
- if it is, how do I know what $\boldsymbol{\Phi}(\mathbf{x})$ is?
  - you may **never** know

e.g., the Gaussian kernel

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

is very popular, but it is not obvious what $\boldsymbol{\Phi}(\mathbf{x})$ is…

  - on the **positive** side, we didn't know how to choose $\boldsymbol{\Phi}(\mathbf{x})$ → choosing instead $K(\mathbf{x}, \mathbf{z})$ makes **no** difference
- why is it that using $K(\mathbf{x}, \mathbf{z})$ is easier/better?
  - **complexity**
  - let's look at an example

Example: Polynomial Kernels

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

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

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

$$= x_1x_1z_1z_1 + x_1x_2z_1z_2 + \cdots + x_1x_dz_1z_d +$$

$$+ x_2x_1z_2z_1 + x_2x_2z_2z_2 + \cdots + x_2x_dz_2z_d +$$

$$\vdots$$

$$+ x_dx_1z_dz_1 + x_dx_2z_dz_2 + \cdots + x_dx_dz_dz_d$$

Example: Polynomial Kernels

- can be written as

$$(\mathbf{x}^T\mathbf{z})^2 = \underbrace{[x_1x_1, x_1x_2, \cdots, x_1x_d, \cdots, x_dx_1, x_dx_2, \cdots, x_dx_d]}_{\boldsymbol{\Phi}(\mathbf{x})^T} \begin{bmatrix} z_1z_1 \\ z_1z_2 \\ \vdots \\ z_1z_d \\ \vdots \\ z_dz_1 \\ z_dz_2 \\ \vdots \\ z_dz_d \end{bmatrix} \Bigg\} \boldsymbol{\Phi}(\mathbf{z})$$

- hence, we have

$$K(\mathbf{x}, \mathbf{z}) = (\mathbf{x}^T\mathbf{z})^2 = \boldsymbol{\Phi}(\mathbf{x})^T \, \boldsymbol{\Phi}(\mathbf{z})$$

with $\boldsymbol{\Phi}$: $\mathbb{R}^d \longrightarrow \mathbb{R}^{d^2}$

$$\begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix} \longrightarrow [x_1x_1, x_1x_2, \cdots, x_1x_d, \cdots, x_dx_1, x_dx_2, \cdots, x_dx_d]^T$$

Example: Polynomial Kernels

- **the point** is that
  - while $\boldsymbol{\Phi}(\mathbf{x})^T\boldsymbol{\Phi}(\mathbf{z})$ has complexity $O(d^2)$
  - **direct** computation of $K(\mathbf{x},\mathbf{z}) = (\mathbf{x}^T\mathbf{z})^2$ has complexity $O(d)$
- direct evaluation is **more** efficient by a factor of d
- as $d \to \infty$, 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

$$K(\mathbf{x},\mathbf{z}) = (1 + \mathbf{x}^T\mathbf{z})^k, k \in \{1,2,\dots\}$$

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

Kernel Summary

1. training set $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ not linearly separable in $\mathcal{X}$, apply feature transformation $\boldsymbol{\Phi}: \mathcal{X} \to \mathcal{Z}$, such that $\dim(\mathcal{Z}) \gg \dim(\mathcal{X})$

2. computing $\boldsymbol{\Phi}(\mathcal{X})$ is too expensive:
   - write our learning algorithm in dot–product form
   - instead of $\boldsymbol{\Phi}(\mathbf{x}_i)$, we only need $\boldsymbol{\Phi}(\mathbf{x}_i)^T \boldsymbol{\Phi}(\mathbf{x}_j), \forall_{ij}$

3. instead of computing $\boldsymbol{\Phi}(\mathbf{x}_i)^T \boldsymbol{\Phi}(\mathbf{x}_j), \forall_{ij}$, define the "dot–product kernel"

$$K(\mathbf{x}, \mathbf{z}) = \boldsymbol{\Phi}(\mathbf{x})^T \boldsymbol{\Phi}(\mathbf{z})$$

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

note: the matrix

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

is called the "kernel" or Gram matrix

4. forget about $\boldsymbol{\Phi}(\mathcal{X})$ and use $K(\mathbf{x}, \mathbf{z})$ from the start!

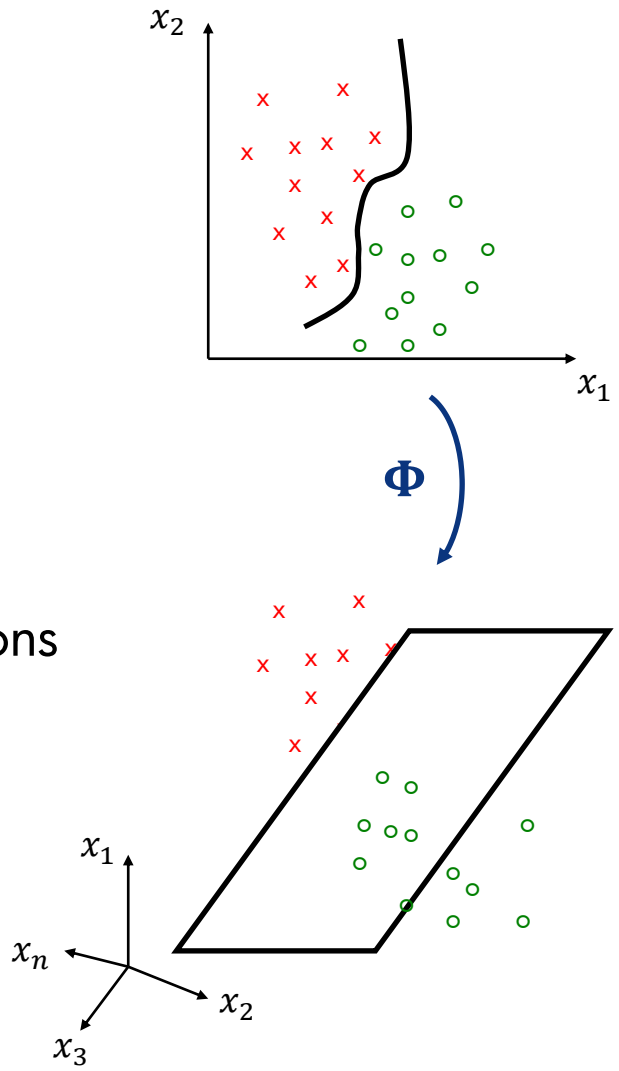
Dot–Product Kernels

- have various **nice properties**
- **Perceptron example**
  - using the "**kernelized**" Perceptron in domain space $\mathcal{X}$

    is **equivalent** to

    using original Perceptron in range space $\mathcal{Z} = \Phi(\mathcal{X})$
  - we have shown that Perceptron learning will converge in a **finite** number of iterations
  - the proof did **not** make any assumptions about the space
  - hence, this **holds** in **range space** and shows that

    Perceptron can learn **non–linear bounds** in **finite time**!



Kernelized Perceptron Learning

- just for completeness, we recall that this is:

set $\alpha_i = 0,\ b = 0$

set $R = \max_i \|\mathbf{x}_i\|$

do {

  for $i = 1{:}n$ {

    if $y_i \left(\sum_{j=1}^{n} \alpha_j y_j \boldsymbol{\Phi}(\mathbf{x}_j)^T \boldsymbol{\Phi}(\mathbf{x}_i) + b \right) \leq 0$ then {

      - $\alpha_i = \alpha_i + 1$
      - $b = b + \eta\, y_i\, R^2$

    }

  }

} until no errors

Perceptron Learning

- **Theorem:** Let $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ and

$$R = \max_i \|\mathbf{\Phi}(\mathbf{x}_i)\|.$$

If there is $(\mathbf{w}^*, b^*)$ such that $\|\mathbf{w}^*\| = 1$ and

$$y_i\left(\mathbf{w}^{*T}\mathbf{\Phi}(\mathbf{x}_i) + b^*\right) > \gamma, \forall i,$$

then the **kernelized** Perceptron will find an error free hyper−plane in at most

$$\left(\frac{2R}{\gamma}\right)^2 \text{ iterations}$$

- note that the margin γ is <u>now</u> in range space $\mathcal{Z}$
- this implies that the choice of kernel must matter!

Choice of Kernel

- what is a good dot–product kernel?
  - the above result suggests that a **good kernel** is one that **maximizes the margin γ in range space**
  - however, nobody knows how to do this

- in practice:
  - pick a kernel from a library of known kernels
  - we have already met
    - the linear kernel: $K(\mathbf{x}, \mathbf{z}) = \mathbf{x}^T\mathbf{z}$
    - the Gaussian family: $K(\mathbf{x}, \mathbf{z}) = e^{-\frac{\|\mathbf{x}-\mathbf{z}\|^2}{\sigma}}$
    - the polynomial family: $K(\mathbf{x}, \mathbf{z}) = (1 + \mathbf{x}^T\mathbf{z})^k, k \in \{1,2, \dots\}$

Dot–Product Kernels

- this may **not** be a bad idea

  - we rip the **benefits** of moving to a high–dimensional space
  - **without** paying a price in complexity

  - the kernel simply adds a **few** parameters (e.g., σ, k)
  - learning it would imply introducing **many** parameters (up to n^2)

- recall that
  - the learning algorithm is **still** maximizing γ in $\mathcal{Z}$
  - it is just that the maximum may not be as **large** as if the kernel were chosen optimally
  - but the difference may **not** justify the risk of overfitting
  - anyway, this is an **open research question**

Question

- does the kernel have to be a dot–product kernel?
- not necessarily
  - e.g., neural networks can be seen as implementing kernels that are not of this type
- however,
  - you lose the **parallelism**: what you know about the learning machine may **no** longer hold **after** you kernelize
  - most of what we are going to do in coming lectures no longer holds if we don't have a dot–product kernel
  - dot–product kernels usually lead to **convex learning problems**: usually you lose this guarantee for non dot–product kernels
- i.e., **you have to be careful!**

Kernelization

- so far, we have seen how to kernelize the Perceptron
- note that **kernelization** is a "trick" that can be applied to **many** learning algorithms
- it **suffices** that you can work the algorithm in terms of dot–products $\mathbf{x}_i^T \mathbf{x}_j$
- however, this form is **not always easy to see**
  - a lot of times you have to **manipulate** the algorithm to write it like this
- we will use **PCA** as an **example**
- recall: PCA is a special case of the **Rayleigh quotient** problem

Recall: Rayleigh Quotient

- Rayleigh quotient

$$\max_{\mathbf{w}} \frac{\mathbf{w}^T \mathbf{S}_B \mathbf{w}}{\mathbf{w}^T \mathbf{S}_W \mathbf{w}}$$

between class scatter
measures separation between class means

$\mathbf{S}_B, \mathbf{S}_W$ symmetric
positive−semidefinite

within class scatter
measures variability inside the classes

- in the PCA case,

$$\mathbf{S}_B = \boldsymbol{\Sigma} \qquad \mathbf{S}_W = \mathbf{I}$$

$$\max_{\mathbf{w}} \frac{\mathbf{w}^T \boldsymbol{\Sigma}\, \mathbf{w}}{\mathbf{w}^T \mathbf{w}}$$

and

$\mathbf{S}_W^{-1}\mathbf{S}_B$

- $\mathbf{w}^*$ is the eigenvector of largest eigenvalue of the covariance $\boldsymbol{\Sigma}$
- the maximum value of the Rayleigh quotient is the largest eigenvalue λ

Kernelization of PCA

- the **problem** is that the covariance $\boldsymbol{\Sigma}$ is **not** a function of the dot–products $\mathbf{x}_i^T \mathbf{x}_j$

- in fact, it is the opposite, since

$$\boldsymbol{\Sigma} = \frac{1}{n}\sum_i (\mathbf{x}_i - \boldsymbol{\mu})(\mathbf{x}_i - \boldsymbol{\mu})^T = \frac{1}{n}\sum_i \mathbf{x}_i^c (\mathbf{x}_i^c)^T$$

depends on the **outer–products**, or in matrix notation

$$\boldsymbol{\Sigma} = \frac{1}{n}\mathbf{X}_c \mathbf{X}_c^T \qquad \mathbf{X}_c = \begin{bmatrix} | & & | \\ \mathbf{x}_1^c & \cdots & \mathbf{x}_n^c \\ | & & | \end{bmatrix}$$

- we have seen, however, that there is an **alternative** form, which the dual of the Rayleigh quotient

The Rayleigh Quotient Dual: PCA

$$\max_{\mathbf{w}} \frac{\mathbf{w}^T \mathbf{S}_B \mathbf{w}}{\mathbf{w}^T \mathbf{S}_W \mathbf{w}}$$

$$\max_{\mathbf{w}} \frac{\mathbf{w}^T \boldsymbol{\Sigma}\, \mathbf{w}}{\mathbf{w}^T \mathbf{w}}$$

- let's **assume**, for a moment, that the PCA solution is of the form

$$\mathbf{w} = \mathbf{X}_c \boldsymbol{\alpha} \qquad \mathbf{X}_c = \begin{bmatrix} | & & | \\ \mathbf{x}_1^c & \cdots & \mathbf{x}_n^c \\ | & & | \end{bmatrix} \quad \boldsymbol{\alpha} \in \mathbb{R}^n$$

  i.e., a **linear** combination of the centered datapoints

- hence, the problem is equivalent to

$$\max_{\boldsymbol{\alpha}} \frac{\boldsymbol{\alpha}^T \mathbf{X}_c^T \boldsymbol{\Sigma}\, \mathbf{X}_c \boldsymbol{\alpha}}{\boldsymbol{\alpha}^T \mathbf{X}_c^T \mathbf{X}_c \boldsymbol{\alpha}}$$

- this does **not** change the form of the Rayleigh quotient, so the solution is
  - $\boldsymbol{\alpha}^*$ is the eigenvector of **largest eigenvalue** of $(\mathbf{X}_c^T \mathbf{X}_c)^{-1} \mathbf{X}_c^T \boldsymbol{\Sigma} \mathbf{X}_c$ ← $\mathbf{S}_W^{-1} \mathbf{S}_B$
  - the **maximum value** of the Rayleigh quotient is the largest eigenvalue λ

The Rayleigh Quotient Dual: PCA

- since

$$\mathbf{S}_B = \boldsymbol{\Sigma} = \frac{1}{n}\mathbf{X}_c\mathbf{X}_c^T \qquad \mathbf{S}_W = \mathbf{I}$$

the solution satisfies

$$\mathbf{S}_B\mathbf{w} = \lambda\mathbf{S}_W^{-1}\mathbf{w} \iff \frac{1}{n}\mathbf{X}_c\mathbf{X}_c^T\mathbf{w} = \lambda\mathbf{w} \iff \mathbf{w} = \mathbf{X}_c\underbrace{\frac{1}{n\lambda}\mathbf{X}_c^T\mathbf{w}}_{\boldsymbol{\alpha}}$$

- two things to **note** here
  - this confirms our assumption that $\mathbf{w} = \mathbf{X}_c\boldsymbol{\alpha}$
  - the solution is now
    - $\mathbf{w}^*$ – the eigenvector of $\boldsymbol{\Sigma} = \mathbf{X}_c\mathbf{X}_c^T$
    - $\boldsymbol{\alpha}^*$ – the eigenvector of $(\mathbf{X}_c^T\mathbf{X}_c)^{-1}\mathbf{X}_c^T\boldsymbol{\Sigma}\,\mathbf{X}_c = (\mathbf{X}_c^T\mathbf{X}_c)^{-1}\mathbf{X}_c^T\mathbf{X}_c\mathbf{X}_c^T\mathbf{X}_c = \mathbf{X}_c^T\mathbf{X}_c$

$$\max_{\mathbf{w}} \frac{\mathbf{w}^T\boldsymbol{\Sigma}\,\mathbf{w}}{\mathbf{w}^T\mathbf{w}} \qquad \max_{\boldsymbol{\alpha}} \frac{\boldsymbol{\alpha}^T\mathbf{X}_c^T\boldsymbol{\Sigma}\,\mathbf{X}_c\boldsymbol{\alpha}}{\boldsymbol{\alpha}^T\mathbf{X}_c^T\mathbf{X}_c\boldsymbol{\alpha}}$$

- hence, we have two alternative manners in which to compute PCA

The Rayleigh Quotient Dual: PCA

primal

- assemble matrix

$$\boldsymbol{\Sigma} = \mathbf{X}_c \mathbf{X}_c^T$$

- compute eigenvectors $\boldsymbol{\phi}_i$
- these are the principal components

dual

- assemble matrix

$$\mathbf{K} = \mathbf{X}_c^T \mathbf{X}_c$$

- compute eigenvectors $\boldsymbol{\alpha}_i$
- the principal components are $\boldsymbol{\phi}_i = \mathbf{X}_c \boldsymbol{\alpha}_i$

- in both cases, we have an **eigenvalue** problem
  - **primal** on the **sum of the outer–products** $\boldsymbol{\Sigma} = \sum_i \mathbf{x}_i^c \left(\mathbf{x}_j^c\right)^T$
  - **dual** on the **matrix of the dot–products** $K_{ij} = (\mathbf{x}_i^c)^T \mathbf{x}_j^c$

Kernelization of PCA

- the dual form is trivial to kernelize, since

$$\mathbf{K} = \mathbf{X}_c^T \mathbf{X}_c \qquad \mathbf{X}_c = \begin{bmatrix} | & & | \\ \mathbf{x}_1^c & \cdots & \mathbf{x}_n^c \\ | & & | \end{bmatrix}$$

  has elements $K_{ij} = (\mathbf{x}_i^c)^T \mathbf{x}_j^c$, which are the dot–products of the centered examples

- to kernelize the algorithm it suffices to replace the dot–product by the kernel

$$K'_{ij} = \mathbf{\Phi}(\mathbf{x}_i^c)^T \mathbf{\Phi}(\mathbf{x}_j^c) = K(\mathbf{x}_i^c, \mathbf{x}_j^c)$$

- the kernelized version of the algorithm is obtained by replacing $\mathbf{K}$ with $\mathbf{K}'$

Kernel PCA

- this is known as **Kernel PCA (KPCA)**

  - compute the **kernel matrix $\mathbf{K}'$** such that

    $$K'_{ij} = K\left(\mathbf{x}_i^c, \mathbf{x}_j^c\right)$$

  - compute its eigenvectors $\boldsymbol{\alpha}_i$

- there is still a **problem**
  - the **principal components** are now

$$\mathbf{w}_i = \boldsymbol{\Phi}(\mathbf{X}_c)\boldsymbol{\alpha}_i = \sum_j \boldsymbol{\Phi}\left(\mathbf{x}_j^c\right)(\boldsymbol{\alpha}_i)_j$$

and **cannot** be computed explicitly since we **cannot** compute $\boldsymbol{\Phi}(\cdot)$
(a lot of times we don't even know what it is)

$$\mathbf{w}_i = \boldsymbol{\Phi}(\mathbf{X}_c)\boldsymbol{\alpha}_i = \sum_j \boldsymbol{\Phi}\left(\mathbf{x}_j^c\right)(\boldsymbol{\alpha}_i)_j$$

Kernel PCA

- however, we can **still** compute the **projections** of new points

$$\boldsymbol{\Phi}(\mathbf{x})^T\mathbf{w}_i = \sum_j \boldsymbol{\Phi}(\mathbf{x})^T\boldsymbol{\Phi}\left(\mathbf{x}_j^c\right)(\boldsymbol{\alpha}_i)_j = \sum_j K\left(\mathbf{x}, \mathbf{x}_j^c\right)(\boldsymbol{\alpha}_i)_j$$

- this is a **very common feature** of **kernelized** algorithms

  - we **no longer know everything** about the solution
    - here, we do **not know** what the PCs are
    - for a classifier, you may **not know** what the boundary is
  - but we **still know enough** to evaluate **everything** we need
    - here, we **just need** the KPCA projections of a new point $\mathbf{x}$
    - for a classifier, we **only need** the scores of the decision rule

Kernel PCA

- the advantage of KPCA is that we are now computing the projection on a
  - linear subspace after the feature transformation
  - non–linear subspace in the original space
- this allows us to apply the dimensionality reduction idea to very non–linear problems