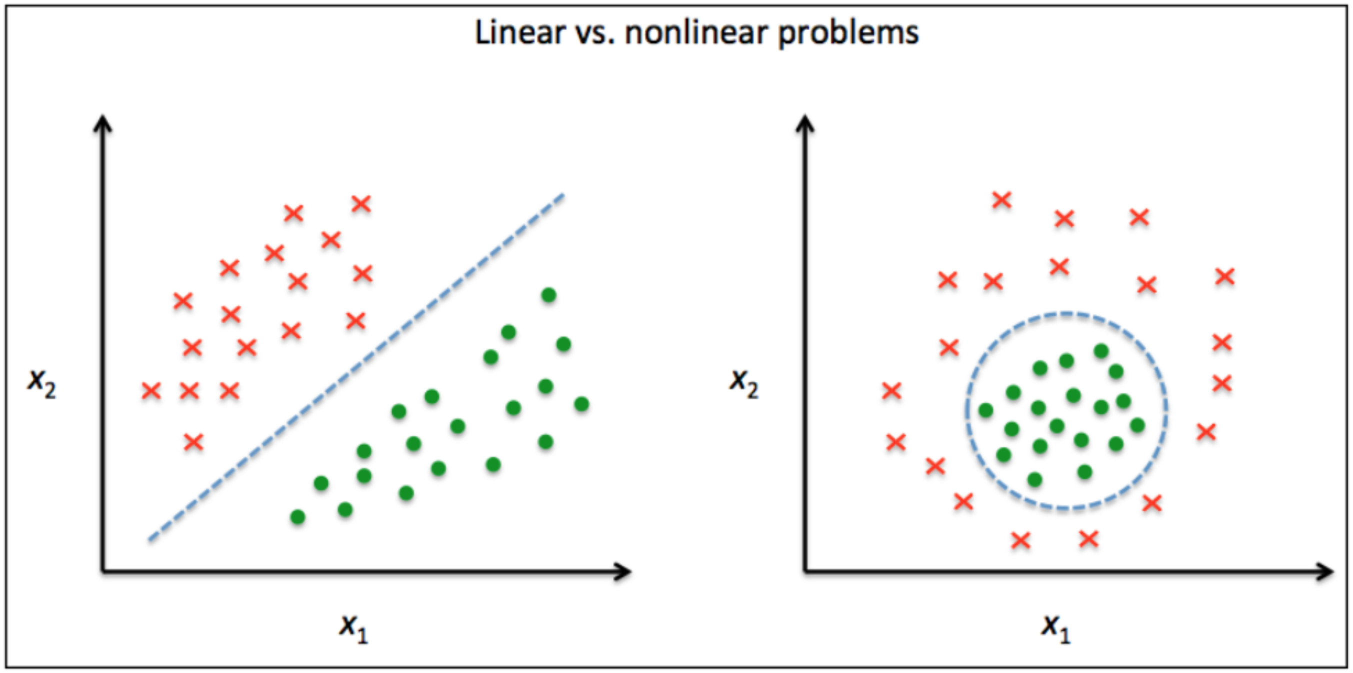
Nonlinear model

via feature transform & kernel trick

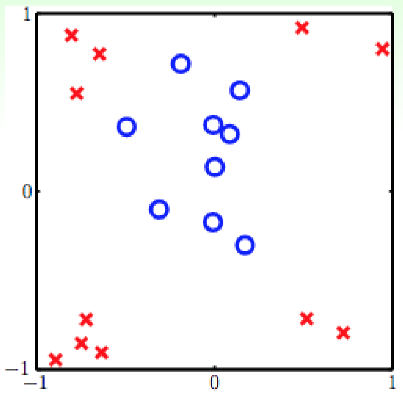
DeepNK Machine Learning Webinar
(https://hackmd.io/CwBgTAxgbARgZgQwLQEYDMYCsTRqkgTgFM584DoB2IlMADgBMI6g)

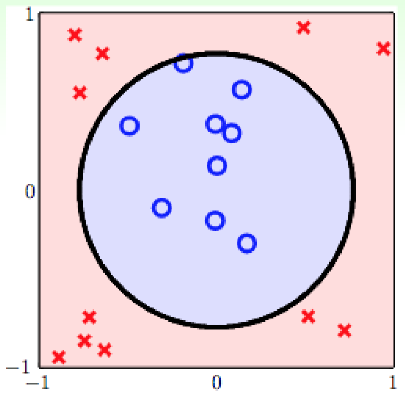
Linear model is limited …

- On some dataset, the linear model just performed poorly during training phase
- Non-trivally linear separaable data
- Essentially training data are non-linear



Circular Separable

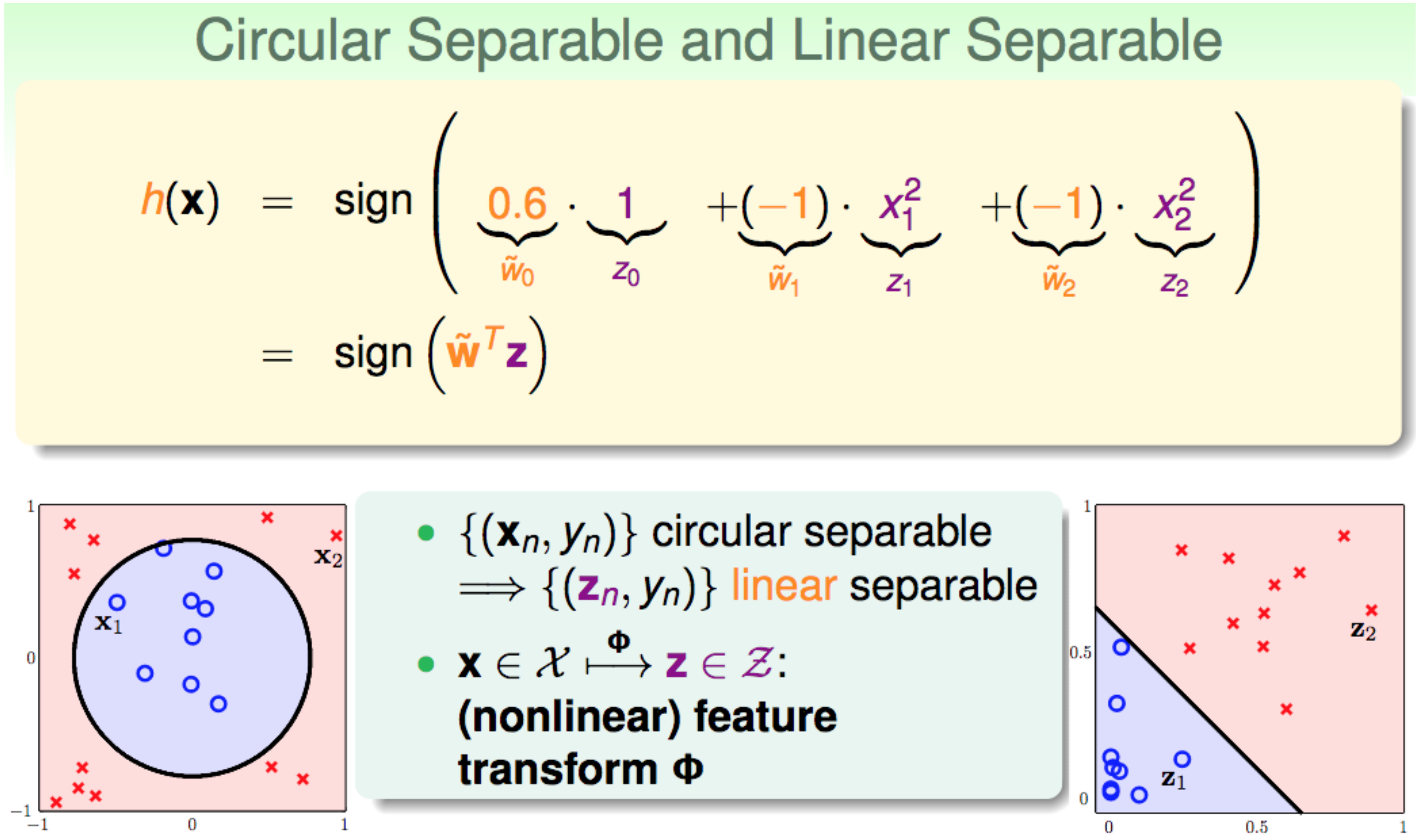




- $\mathcal{D}$ not linear separable
- but **circular separable** by a circle of radius $\sqrt{0.6}$ centered at origin:

$$h_{\text{SEP}}(\mathbf{x}) = \operatorname{sign}\left(-x_1^2 - x_2^2 + 0.6\right)$$

Often we want to **capture nonlinear patterns** …



Circular Separable and Linear Separable

$$\begin{aligned} h(\mathbf{x}) &= \text{sign}\left(\underbrace{0.6}_{\tilde{w}_0} \cdot \underbrace{1}_{z_0} + \underbrace{(-1)}_{\tilde{w}_1} \cdot \underbrace{x_1^2}_{z_1} + \underbrace{(-1)}_{\tilde{w}_2} \cdot \underbrace{x_2^2}_{z_2} \right) \\ &= \text{sign}\left(\tilde{\mathbf{w}}^T \mathbf{z}\right) \end{aligned}$$

- $\{(\mathbf{x}_n, y_n)\}$ circular separable $\Longrightarrow \{(\mathbf{z}_n, y_n)\}$ linear separable
- $\mathbf{x} \in \mathcal{X} \overset{\Phi}{\longmapsto} \mathbf{z} \in \mathcal{Z}$: (nonlinear) feature transform Φ

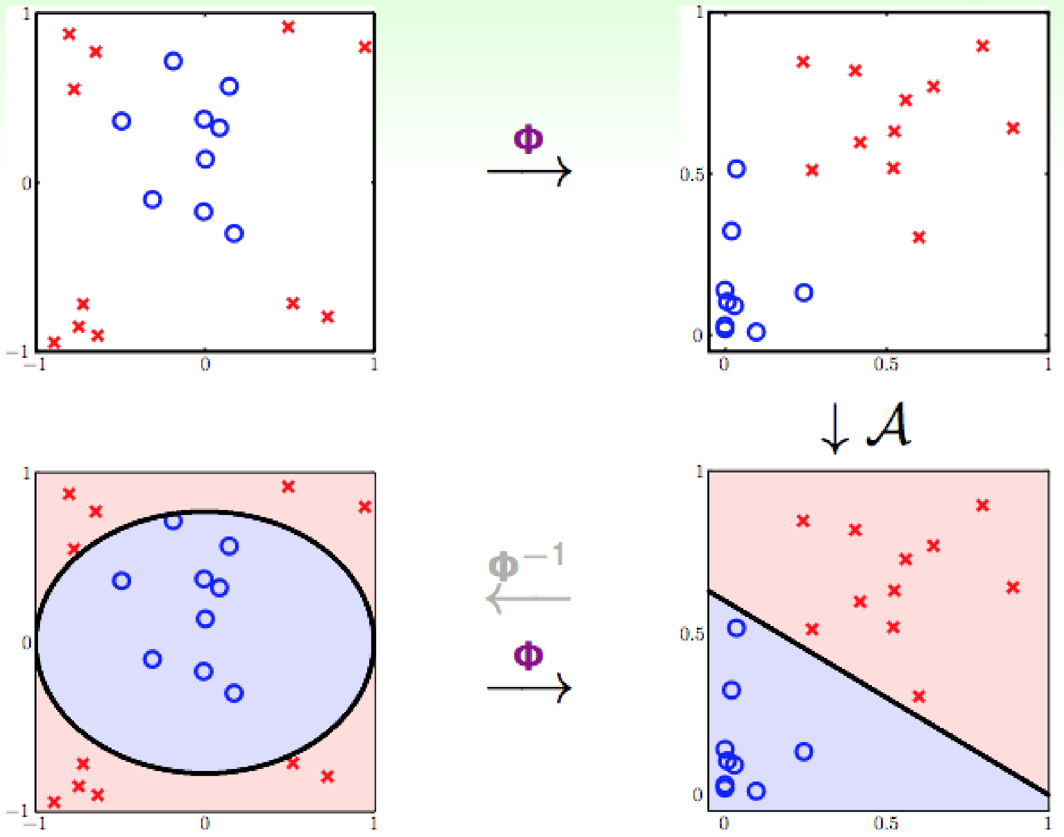
From: Hsuan-Tien Lin's ML Foundation (https://www.youtube.com/watch?v=8pQ06pku1xA)

Idea of nonlinear transform

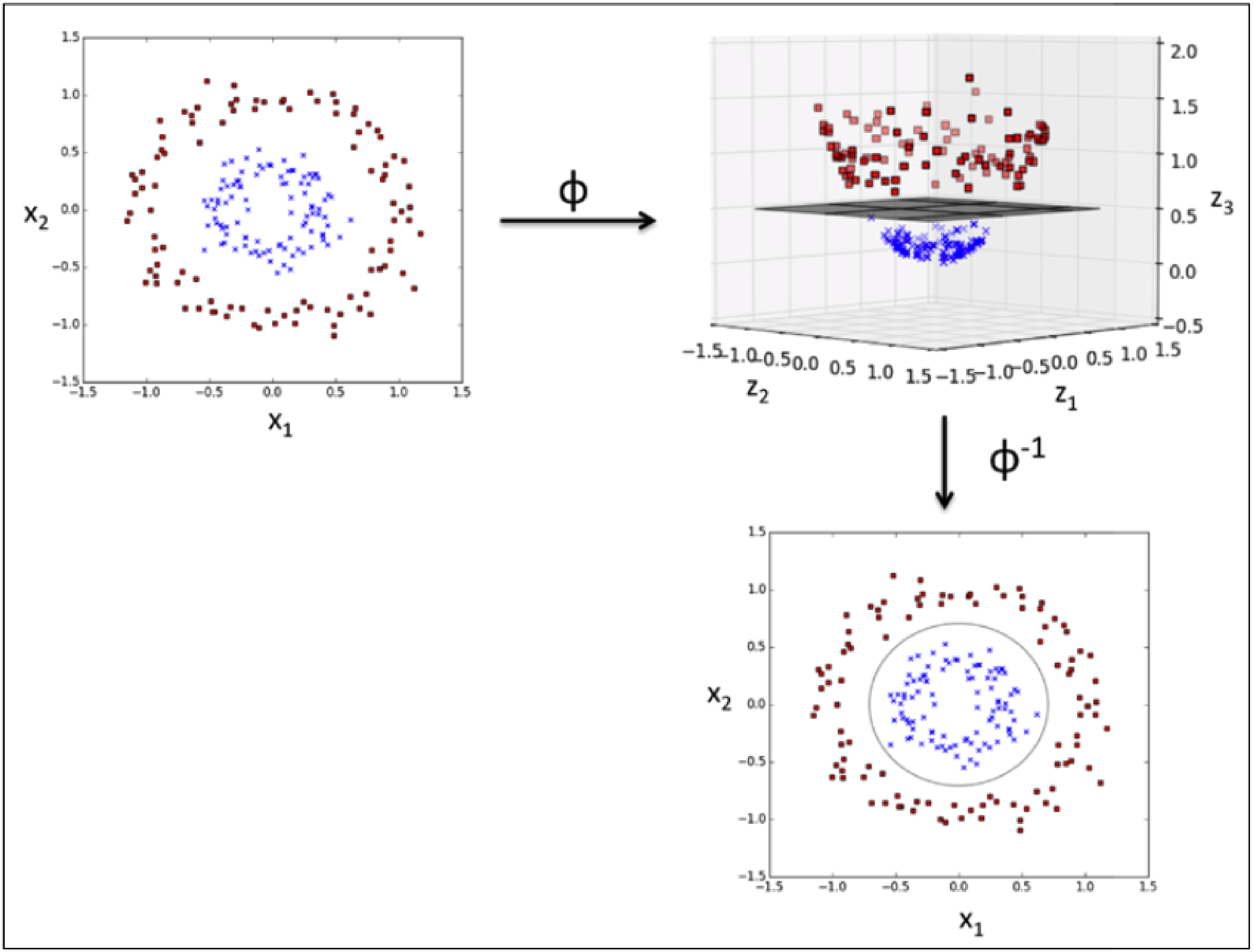
Instead of direct linear weighting, we can projectmap features by non-linear feature transfrom $\Phi : \mathcal{X} \to \mathcal{F}$ and then apply linear weighting to these mapped features.

- So we can apply good-old linear classification algorithms (e.g. Perceptron, Logistic regression, etc …)
- An inverse mapping is not required, but we human usually define bijection for good.

The Nonlinear Transform Steps



1. transform original data $\{(\mathbf{x}_n, y_n)\}$ to $\{(\mathbf{z}_n = \mathbf{\Phi}(\mathbf{x}_n), y_n)\}$ by $\mathbf{\Phi}$
2. get a good perceptron $\tilde{\mathbf{w}}$ using $\{(\mathbf{z}_n, y_n)\}$ and your favorite linear classification algorithm $\mathcal{A}$
3. return $g(\mathbf{x}) = \text{sign}\left(\tilde{\mathbf{w}}^T \mathbf{\Phi}_2(\mathbf{x})\right)$



x_2
x_1
ϕ
z_3
z_2
z_1
ϕ^{-1}
x_2
x_1

Nonlinear and higher dimensionality

The transform does not need to increase the dimension of feature vector space (As written in the book). But one may argue that:

A circle in 2D is one kind of conic section in 3D. So it is in a sense that a nonlinear transform projects feature vectors into a higher dimensional space.

And usually, a linear separator in higher dimension implies nonlinear separator in the original space.

But, No free lunch …

As we know that nonlinear model increase the expressive power to describe the data, but it comes with price …

- The price of **computation/storage**
  - But **kernel tricks** will help, see later !!
- The price of **model complexity**
  - The buzzword: **overfitting**

Computation/Storage Price

Q-th order polynomial transform: $\mathbf{\Phi}_Q(\mathbf{x}) = \Big(1,$
$x_1, x_2, \ldots, x_d,$
$x_1^2, x_1 x_2, \ldots, x_d^2,$
$\ldots,$
$x_1^Q, x_1^{Q-1} x_2, \ldots, x_d^Q \Big)$

$\underbrace{1}_{\tilde{w}_0} + \underbrace{\tilde{d}}_{\text{others}}$ dimensions

= # ways of $\leq Q$-combination from d kinds with repetitions

= $\binom{Q+d}{Q}$ = $\binom{Q+d}{d}$ = $O(Q^d)$

= efforts needed for computing/storing $\mathbf{z} = \mathbf{\Phi}_Q(\mathbf{x})$ and $\tilde{\mathbf{w}}$

Q large $\Longrightarrow$ **difficult to compute/store**

Idea of Kernel method (tricks)

We know from abve that such a **nonlinear transform** is usually costly to compute and store, especially in much higher dimensions.

In some learning algorithms (https://en.wikipedia.org/wiki/Instance-based_learning), they may computes the **dot product** or **similarity measure** between data samples.

Is there any way to do **nonlinear transform** and **similarity measure "at the same time"**?

Def. of kernel function

Let $\boldsymbol{x} = [x_1, x_2, \ldots x_m]^T$ be the sample vector in *m*-dimension input space. And the nonlinear transform Φ : $\mathcal{X} \to \mathcal{F}$ maps $\boldsymbol{x} \in \mathcal{X}$ into $\phi(\boldsymbol{x}) \in \mathcal{F}$

The kernel function κ : $\mathcal{X} \times \mathcal{X} \to \mathbb{R}$

$$k(\boldsymbol{x}, \boldsymbol{x}') = \phi(\boldsymbol{x})^T \phi(\boldsymbol{x}')$$

computes **dot product similarity** in the transformed feature space $\mathcal{Z}$ given two sample vectors $\boldsymbol{x}$ and $\boldsymbol{x}'$ in orignal input space

Properties of kernel function

- $k(\boldsymbol{x}, \boldsymbol{x}')$ must be a proper inner product in feature space $\mathcal{Z}$. So $\mathcal{Z}$ is an inner product space (https://en.wikipedia.org/wiki/Inner_product_space).
    - If $\mathcal{Z}$ is a complete vector space then it is called Hilbert space (https://en.wikipedia.org/wiki/Hilbert_space)
- Can just *any* function be used as a kernel function ?
    - No, It must satisfy **Mercer’s Condition (https://en.wikipedia.org/wiki/Mercer%27s_condition)**
- The kernel will do **implicit** feature transform

Mercer's Condition

- For k to be a kernel function
  - There must exist a Hilbert Space $\mathcal{F}$ for which k defines a dot product
  - The above is true if K is a positive definite function

$$\int d\mathbf{x} \int d\mathbf{z} f(\mathbf{x}) k(\mathbf{x}, \mathbf{z}) f(\mathbf{z}) > 0 \quad (\forall f \in L_2)$$

  - This is Mercer's Condition
- Let k_1, k_2 be two kernel functions then the following are as well:
  - $k(\mathbf{x}, \mathbf{z}) = k_1(\mathbf{x}, \mathbf{z}) + k_2(\mathbf{x}, \mathbf{z})$: direct sum
  - $k(\mathbf{x}, \mathbf{z}) = \alpha k_1(\mathbf{x}, \mathbf{z})$: scalar product
  - $k(\mathbf{x}, \mathbf{z}) = k_1(\mathbf{x}, \mathbf{z}) k_2(\mathbf{x}, \mathbf{z})$: direct product
  - Kernels can also be constructed by composing these rules

Kernels as High Dimensional Feature Mapping

- Consider two examples $\mathbf{x} = \{x_1, x_2\}$ and $\mathbf{z} = \{z_1, z_2\}$
- Let's assume we are given a function k (kernel) that takes as inputs $\mathbf{x}$ and $\mathbf{z}$

$$\begin{aligned} k(\mathbf{x}, \mathbf{z}) &= (\mathbf{x}^\top \mathbf{z})^2 \\ &= (x_1 z_1 + x_2 z_2)^2 \\ &= x_1^2 z_1^2 + x_2^2 z_2^2 + 2x_1 x_2 z_1 z_2 \\ &= (x_1^2, \sqrt{2}x_1 x_2, x_2^2)^\top (z_1^2, \sqrt{2}z_1 z_2, z_2^2) \\ &= \phi(\mathbf{x})^\top \phi(\mathbf{z}) \end{aligned}$$

- The above k implicitly defines a mapping ϕ to a higher dimensional space

$$\phi(\mathbf{x}) = \{x_1^2, \sqrt{2}x_1 x_2, x_2^2\}$$

- Note that we didn't have to define/compute this mapping
- Simply defining the kernel a certain way gives a higher dim. mapping ϕ
- Moreover the kernel $k(\mathbf{x}, \mathbf{z})$ also computes the dot product $\phi(\mathbf{x})^\top \phi(\mathbf{z})$
  - $\phi(\mathbf{x})^\top \phi(\mathbf{z})$ would otherwise be much more expensive to compute explicitly
- All kernel functions have these properties

Kernel PCA

Simply put: Apply PCA on the projected features $\phi(\boldsymbol{x})$, but use **kernel tricks** to avoid the explicit Φ transfrom. Before we add the kernel tricks, let’s recap *standard* PCA:

- Standardize the dataset
- Compute the covariance matrix Σ on the dataset
- Find the eigenvalues and eigenvectors of Σ
- Select m-th principal components
- Project the dataset by these PCs

Projected covariance matrix

After standardization, each element σ_{jk} of *m* x *m* covariance matrix Σ is

$$\sigma_{jk} = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{x}_j^{(i)} \boldsymbol{x}_k^{(i)}$$

Where $\boldsymbol{x}^{(i)} = [x_1^{(i)}, x_2^{(i)}, \ldots x_m^{(i)}]^T$ represents a mean-centered sample vector with *m* features. Reader may quickly find that Σ is also the sum of *n* *m*x*m* matrices of $\boldsymbol{x}^{(i)}\boldsymbol{x}^{(i)T}$ (outer product of $\boldsymbol{x}^{(i)}$ to itself)

So we can write the covariance matrix in this form:

$$\Sigma = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{x}^{(i)}\boldsymbol{x}^{(i)T} = \frac{1}{n}[\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(n)}]\begin{bmatrix} \boldsymbol{x}^{(1)T} \\ \vdots \\ \boldsymbol{x}^{(n)T} \end{bmatrix}$$

$$= \frac{1}{n}X^TX \ , \ x_j^{(i)} \in X_{n\times m} \ , \ i \le n \ , \ j \le m$$

Replace each $\boldsymbol{x}^{(i)}$ into projected $\phi(\boldsymbol{x}^{(i)})$ for us to apply PCA on the projected space:

$$\Sigma_\Phi = \frac{1}{n}\sum_{i=1}^{n} \phi(\boldsymbol{x}^{(i)})\phi(\boldsymbol{x}^{(i)})^T$$

$$= \frac{1}{n}[\phi(\boldsymbol{x}^{(1)}), \dots, \phi(\boldsymbol{x}^{(n)})]\begin{bmatrix} \phi(\boldsymbol{x}^{(1)})^T \\ \vdots \\ \phi(\boldsymbol{x}^{(n)})^T \end{bmatrix}$$

$$= \frac{1}{n}\phi(X)^T\phi(X)$$

The kernel (Gram) matrix

We know that directly compute the projected covariance matrix $\Sigma_\Phi = \frac{1}{n}\phi(X)^T\phi(X)$ is costly or sometimes impossible. Time for the **kernel tricks**!

We can use the kernel function $k(\boldsymbol{x}, \boldsymbol{x}') = \phi(\boldsymbol{x})^T\phi(\boldsymbol{x}')$ to construct the *n*x*n* **kernel matrix** K such that each element $K_{ij} = \phi(\boldsymbol{x}^{(i)})^T\phi(\boldsymbol{x}^{(j)})$.

How we relate K to the projected PCA:
$\Sigma_\Phi \boldsymbol{v} = \lambda \boldsymbol{v}$?

We can rewrite the kernel matrix K as:

$$K = \begin{bmatrix} \phi(\boldsymbol{x}^{(1)})^T \\ \vdots \\ \phi(\boldsymbol{x}^{(n)})^T \end{bmatrix} [\phi(\boldsymbol{x}^{(1)}), \ldots, \phi(\boldsymbol{x}^{(n)})] = \phi(X)\phi(X)^T$$

- $K = \phi(X)\phi(X)^T$ (*n*x*n* matrix)
  $\neq \phi(X)^T\phi(X)$ (*?*x*?* matrix)
- K is a positive semi-definite matrix by the symmetry property of dot product.
  And we can solve the eigenproblem of it: $K\boldsymbol{u} = \lambda\boldsymbol{u}$.

Multiply $\phi(X)^T$ to the both side of $K\boldsymbol{u} = \lambda\boldsymbol{u}$:

$$\phi(X)^T K\boldsymbol{u} = \phi(X)^T[\phi(X)\phi(X)^T]\boldsymbol{u} = \lambda\phi(X)^T\boldsymbol{u}$$
$$\implies [\phi(X)^T\phi(X)]\phi(X)^T\boldsymbol{u} = \lambda\phi(X)^T\boldsymbol{u}$$

- This means $\phi(X)^T\boldsymbol{u}$ is the eigenvector of $\phi(X)^T\phi(X) = n\Sigma_\Phi$.

Voila! We just relate K to the projected PCA

- However, the norm of $\phi(X)^T\boldsymbol{u}$ may not be 1 (not ortho"normal")

Projecting test samples

The eigenvector $\boldsymbol{v}$ of $\phi(X)^T\phi(X)$ can be computed by the eigenvalue λ of K:

$$\boldsymbol{v} = \frac{1}{\|\phi(X)^T\boldsymbol{u}\|}\phi(X)^T\boldsymbol{u} = \frac{1}{\sqrt{\boldsymbol{u}^T\phi(X)\phi(X)^T\boldsymbol{u}}}\phi(X)^T\boldsymbol{u}$$
$$= \frac{1}{\sqrt{\boldsymbol{u}^T(\lambda\boldsymbol{u})}}\phi(X)^T\boldsymbol{u} = \frac{1}{\sqrt{\lambda}}\phi(X)^T\boldsymbol{u}$$

Uh… but we don't know $\phi(X)^T$… Should we compute $\boldsymbol{v}$ for transforming test sample $\boldsymbol{x}'$ to the principal components of feature space?

No, we use **kernel tricks** again!

The principal component projection (dot product) of the test sample $\phi(\boldsymbol{x}')$ in feature space is:

$$\boldsymbol{v}^T\phi(\boldsymbol{x}') = [\frac{1}{\sqrt{\lambda}}\phi(X)^T\boldsymbol{u}]^T\phi(\boldsymbol{x}') = \frac{1}{\sqrt{\lambda}}\boldsymbol{u}^T\phi(X)\phi(\boldsymbol{x}')$$

$$= \frac{1}{\sqrt{\lambda}}\boldsymbol{u}^T\begin{bmatrix}\phi(\boldsymbol{x}^{(1)})^T\\ \vdots \\ \phi(\boldsymbol{x}^{(n)})^T\end{bmatrix}\phi(\boldsymbol{x}') = \frac{1}{\sqrt{\lambda}}\boldsymbol{u}^T\begin{bmatrix}k(\boldsymbol{x}^{(1)},\boldsymbol{x}')\\ \vdots \\ k(\boldsymbol{x}^{(n)},\boldsymbol{x}')\end{bmatrix}$$

It needs original training dataset to project new samples, a kind of instance-based learning (https://en.wikipedia.org/wiki/Instance-based_learning).

The complete algorithm

- Choose or invent a kernel function $k(\boldsymbol{x},\boldsymbol{x}') = \phi(\boldsymbol{x})^T\phi(\boldsymbol{x}')$ for the *n*x*n* kernel (pairwise-similarity) matrix K
- Compute K for solving $K\boldsymbol{u} = \lambda\boldsymbol{u}$
- Center the kernel matrix by

$$K_{centered} = K - 1_nK - K1_n + 1_nK1_n$$

Where 1_n is an *n*x*n* matrix with all $\frac{1}{n}$

- Collect top-*k* eigenvectors $\boldsymbol{u}$
  The eigenvectors are samples already projected onto principal axes of feature space

Why center the kernel matrix ?

It is similar to the standardization process in *standard* PCA.

But when doing kernel PCA, we do not know whether the ϕ-transformed data is zero-mean, hence the step becomes necessary.

How to center the kernel matrix ?
For each element K_{c-ij} of the centered kernel matrix K_c

TODO (binomial expansion to the projected means)

ref-1 (http://www.ics.uci.edu/~welling/classnotes/papers_class/Kernel-PCA.pdf), ref-2 (http://www.cs.haifa.ac.il/~rita/uml_course/lectures/KPCA.pdf)

Why $\boldsymbol{u}$ is projected samples ?

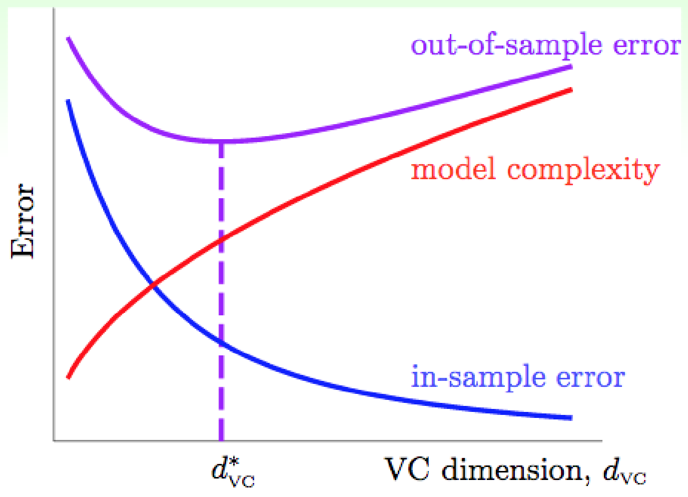
The projected sample is:

$$\begin{aligned} \boldsymbol{v}^T\phi(X)^T = \frac{1}{\sqrt{\lambda}}\boldsymbol{u}^T\phi(X)\phi(X)^T = \frac{1}{\sqrt{\lambda}}\boldsymbol{u}^T K \\ \implies \phi(X)\boldsymbol{v} = \frac{1}{\sqrt{\lambda}}K\boldsymbol{u} = \sqrt{\lambda}\boldsymbol{u} \end{aligned}$$

Apply transpose to both side since K is symmetric.

Linear model first (https://stats.stackexchange.com/questions/73032/)

Linear Model First



- tempting sin: use $\mathcal{H}_{1126}$, low $E_{\text{in}}(g_{1126})$ to fool your boss —**really? :-(a dangerous path of no return**
- safe route: $\mathcal{H}_1$ first
  - if $E_{\text{in}}(g_1)$ good enough, **live happily thereafter :-)**
  - otherwise, move right of the curve **with nothing lost except 'wasted' computation**

linear model first:
simple, efficient, **safe**, **and workable!**

The secret of Radial Basis Function (RBF) kernel

- https://stats.stackexchange.com/questions/80398 (https://stats.stackexchange.com/questions/80398)
- https://stats.stackexchange.com/questions/131138 (https://stats.stackexchange.com/questions/131138)
- https://stats.stackexchange.com/questions/172554 (https://stats.stackexchange.com/questions/172554)
- https://math.stackexchange.com/questions/276707 (https://math.stackexchange.com/questions/276707)

Reference

- https://www.cs.utah.edu/~piyush/teaching/15-9-slides.pdf (https://www.cs.utah.edu/~piyush/teaching/15-9-slides.pdf)
- 李政軒 PCA & kPCA (https://www.youtube.com/watch?v=G2NRnh7W4NQ)
- 林軒田 ML Foundation, Nonlinear Transform (https://www.youtube.com/watch?v=8pQ06pku1xA)