Learning with Kernels

CS771: Introduction to Machine Learning

Purushottam Kar

Announcements

Quiz: October 16 (Wednesday), 6PM, **L20 – same as before**

Assigned seating – don't be late (will waste time finding your seat)

Syllabus is till whatever we covered till October 04 (before midsem recess)

*Bring your **institute ID card** with you – will lose time if you forget*

*Bring a **pencil, pen, eraser, sharpener** with you – we wont provide!*

*Answers to be written on question paper itself. If you write with pen and make a mistake, no extra paper. Final answer **must be in pen***

***Auditors cannot appear** for quiz – please come to L20 at ~ 6:40PM*

Doubt clearing session: Oct 15 (Tue), 6PM **KD101**

Non-linear Learning

These are learning techniques that involve non-linear models

- *Classifiers with non-linear decision boundaries*
- *Regressors that predicts the label using non-linear functions*
- *Dim-redn that reveal if data was lying on a low-dim curved surface*

We have actually seen a few non-linear learning techniques already

- *kNN, learning with multiple prototypes are all capable of learning non-linear decision boundaries, non-linear functions etc*
- *Also saw generalized linear models in the last class*
- *These are nice and valuable but much more powerful methods exist*
- *Will study two such methods – kernel learning and deep learning*

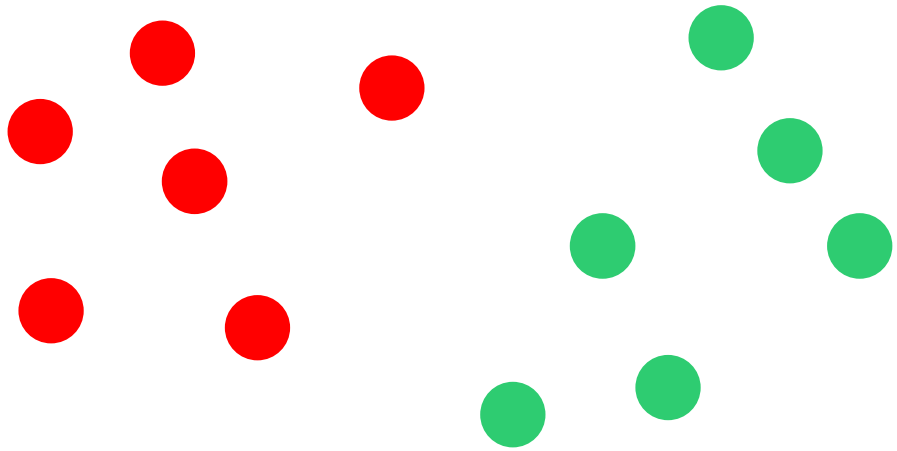
Start with kernel learning

When should I use kernels?

Kernel methods are a good option to try whenever linear models do a poor job on our data

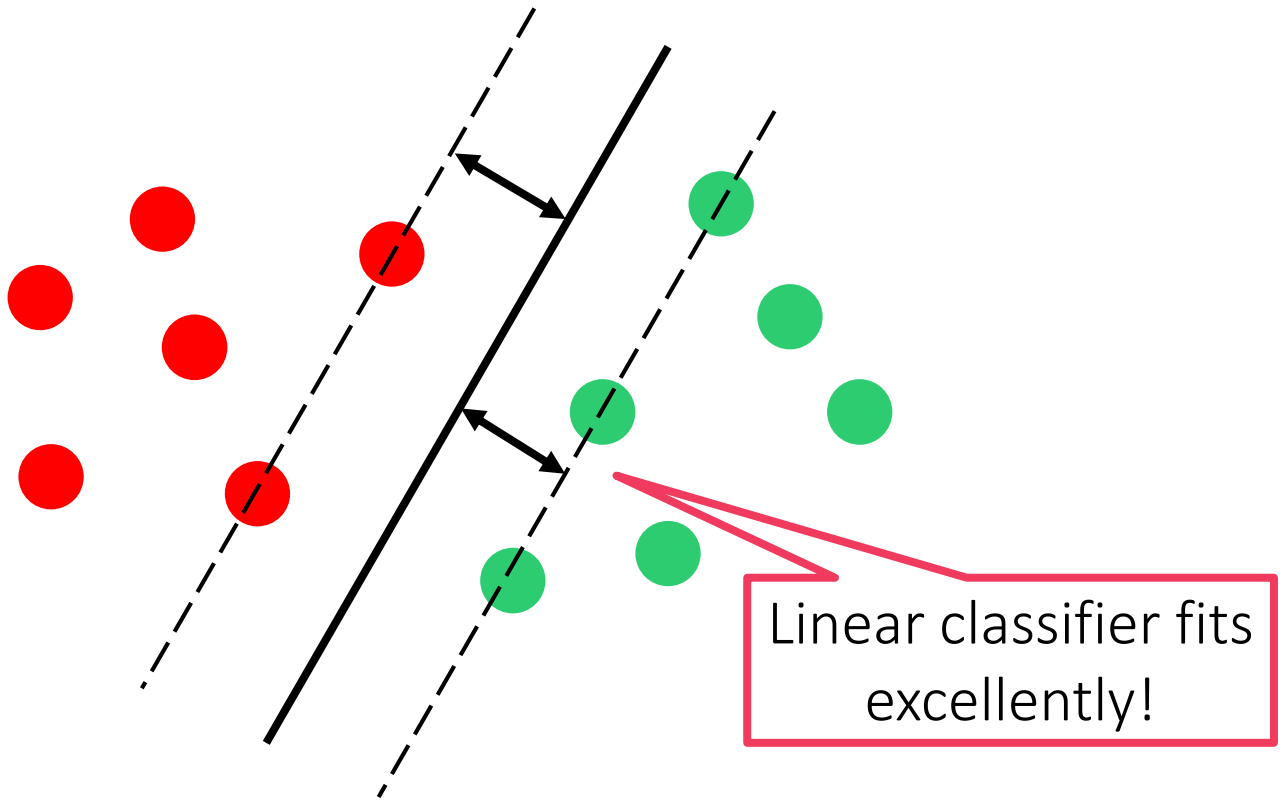
When should I use kernels?

Kernel methods are a good option to try whenever linear models do a poor job on our data



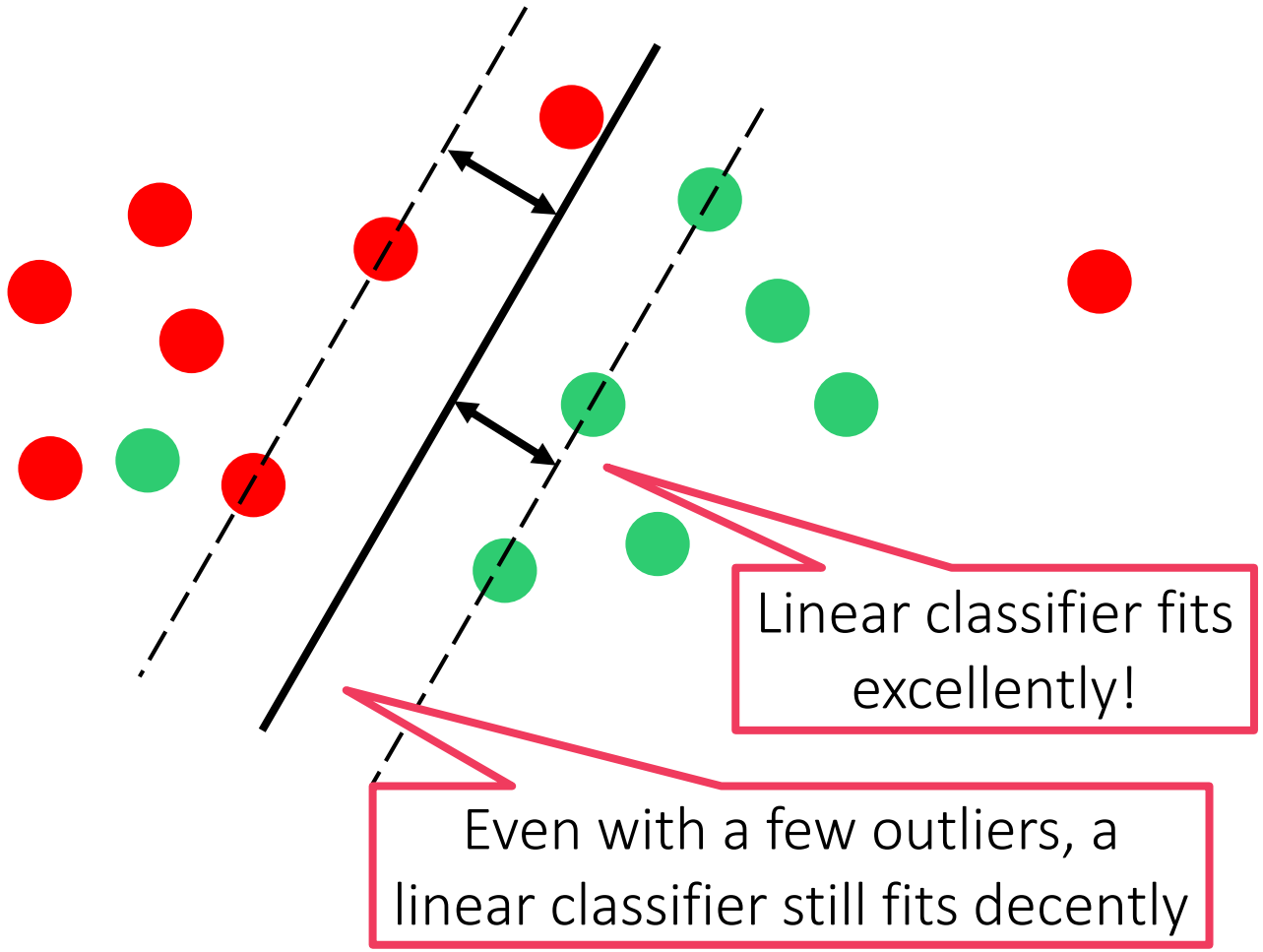
When should I use kernels?

Kernel methods are a good option to try whenever linear models do a poor job on our data



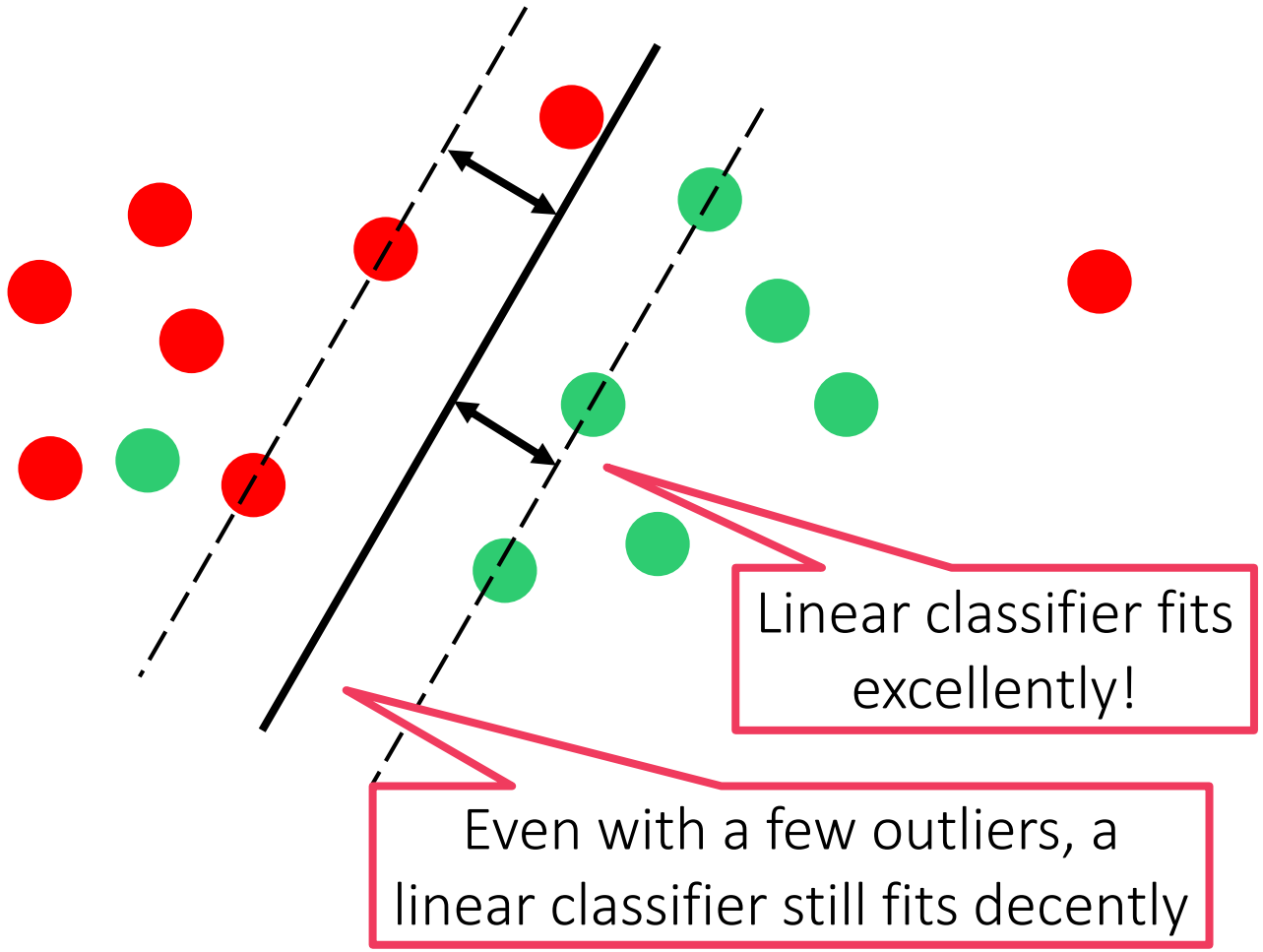
When should I use kernels?

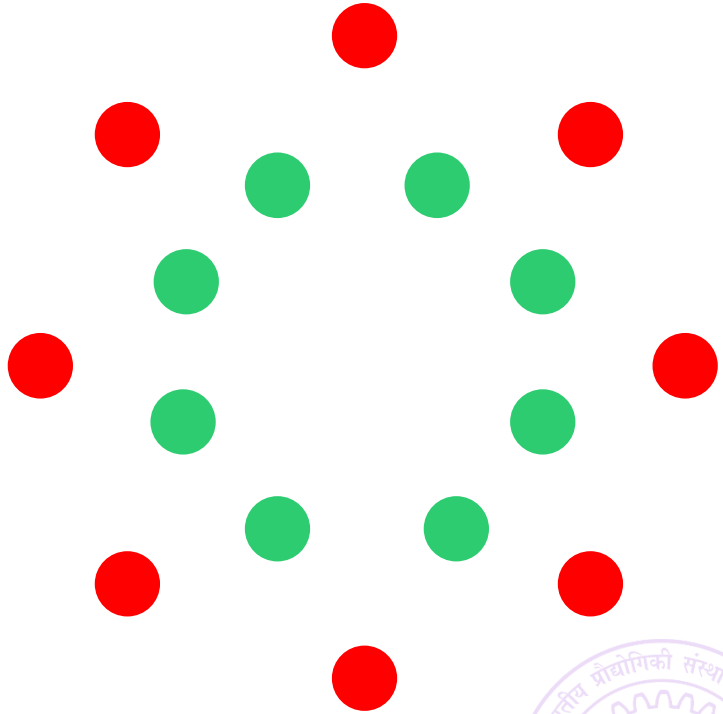
Kernel methods are a good option to try whenever linear models do a poor job on our data



When should I use kernels?

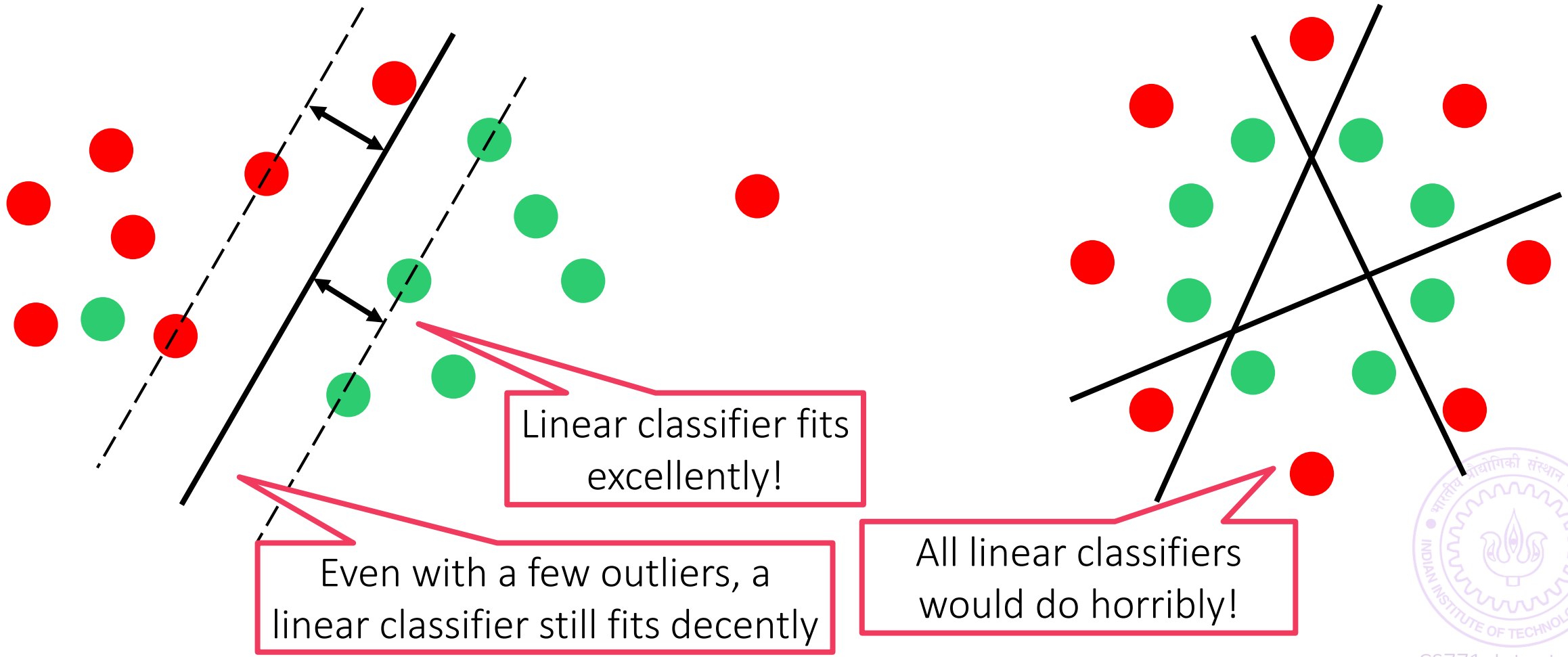
Kernel methods are a good option to try whenever linear models do a poor job on our data





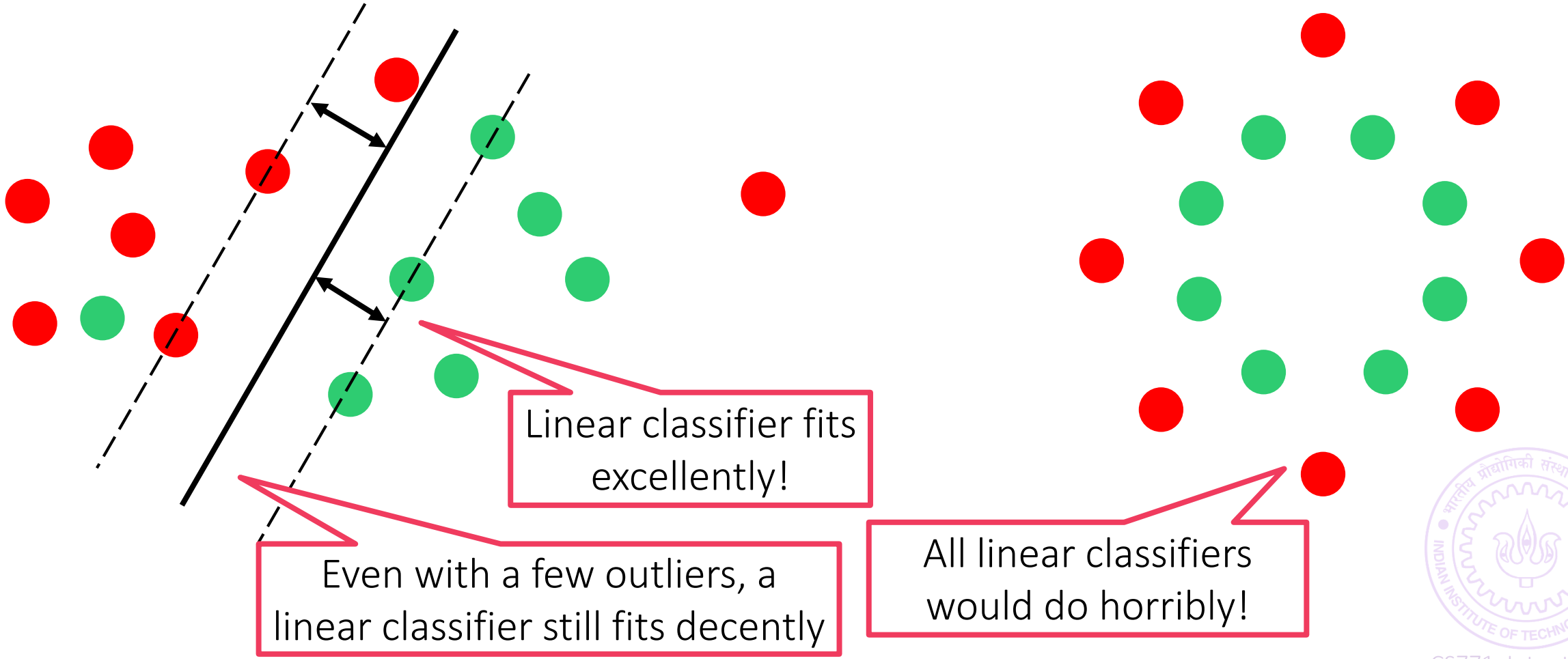
When should I use kernels?

Kernel methods are a good option to try whenever linear models do a poor job on our data



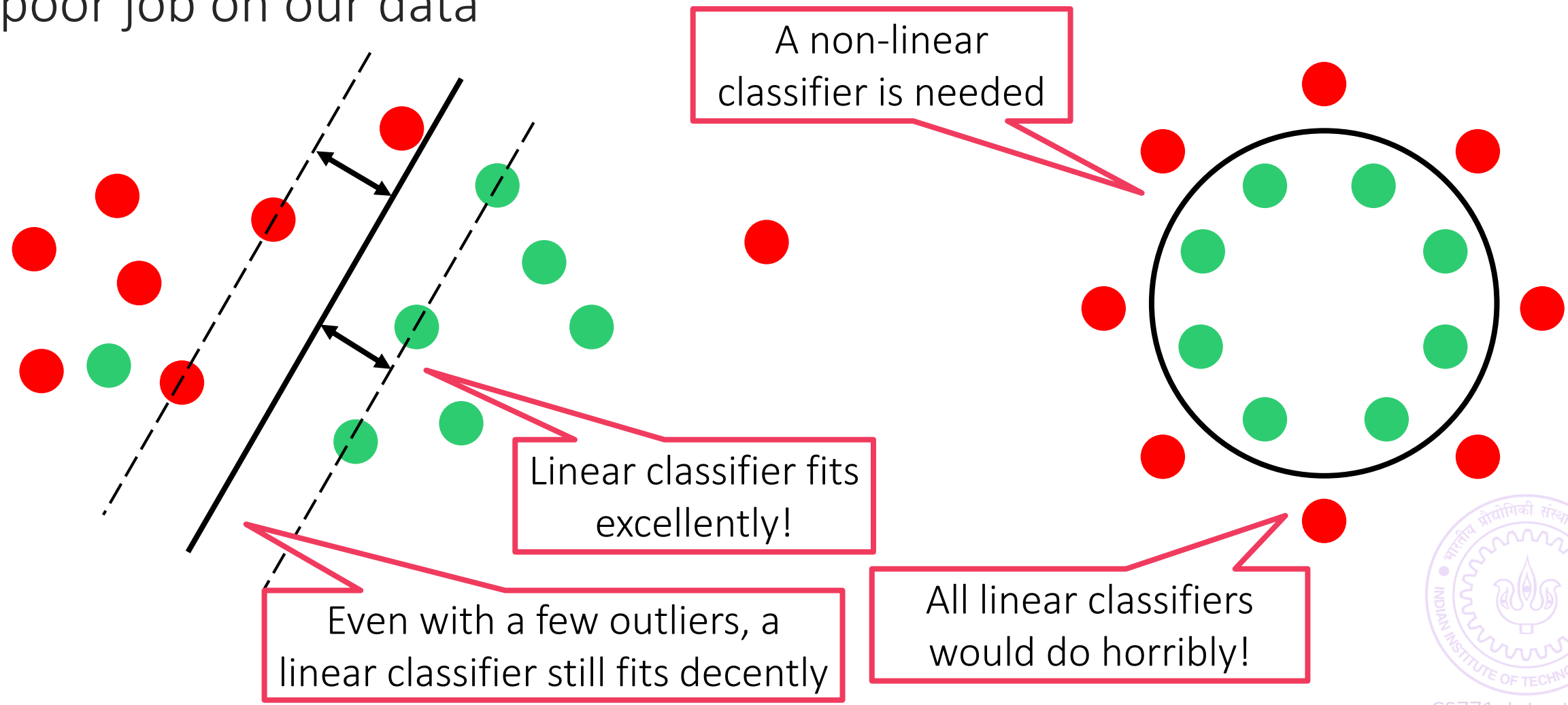
When should I use kernels?

Kernel methods are a good option to try whenever linear models do a poor job on our data



When should I use kernels?

Kernel methods are a good option to try whenever linear models do a poor job on our data



When should I use kernels?

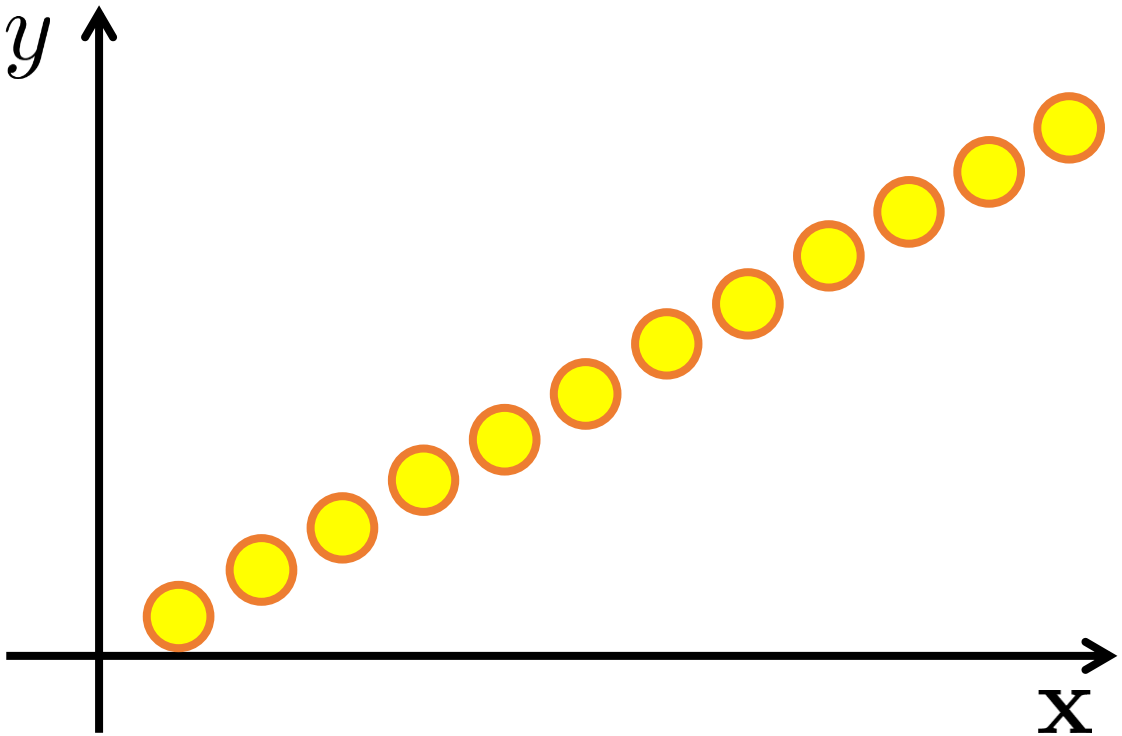
Kernel methods are a good option to try whenever linear models do a poor job on our data

When should I use kernels?

Kernel methods are a good option to try whenever linear models do a poor job on our data

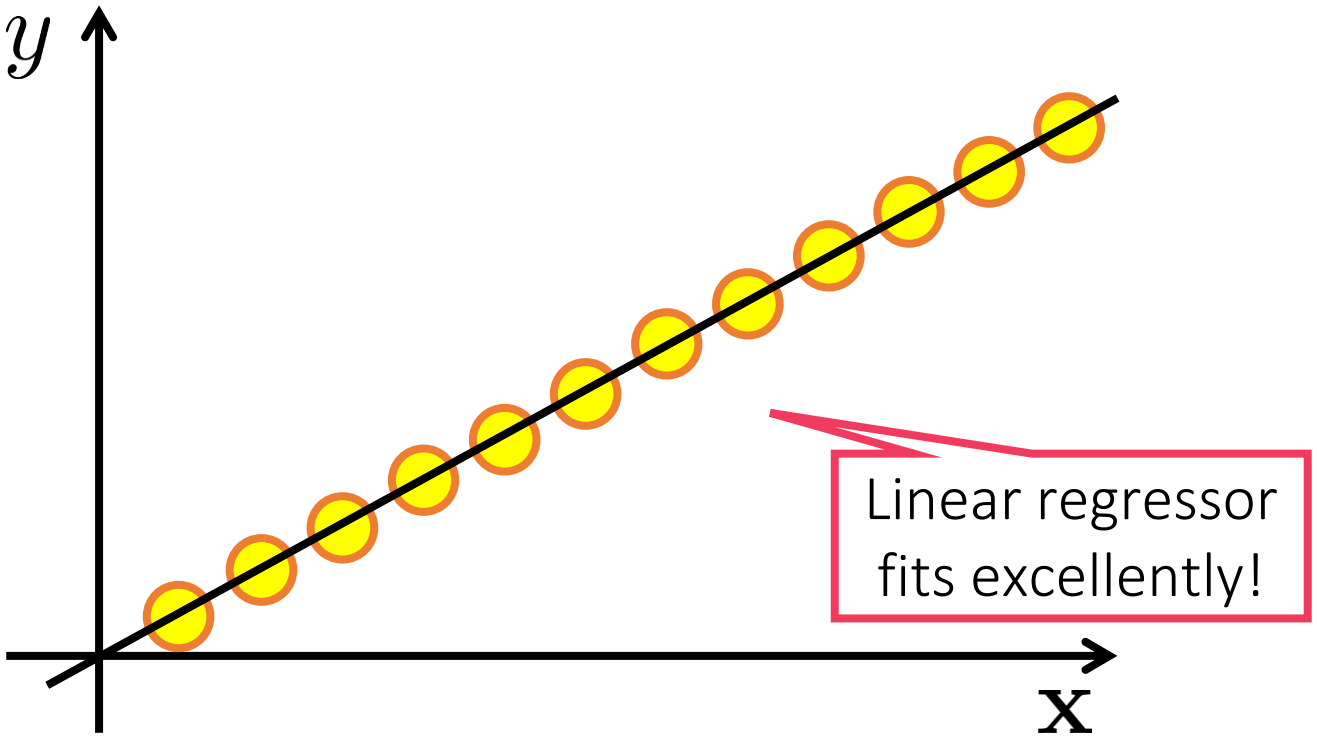
When should I use kernels?

Kernel methods are a good option to try whenever linear models do a poor job on our data



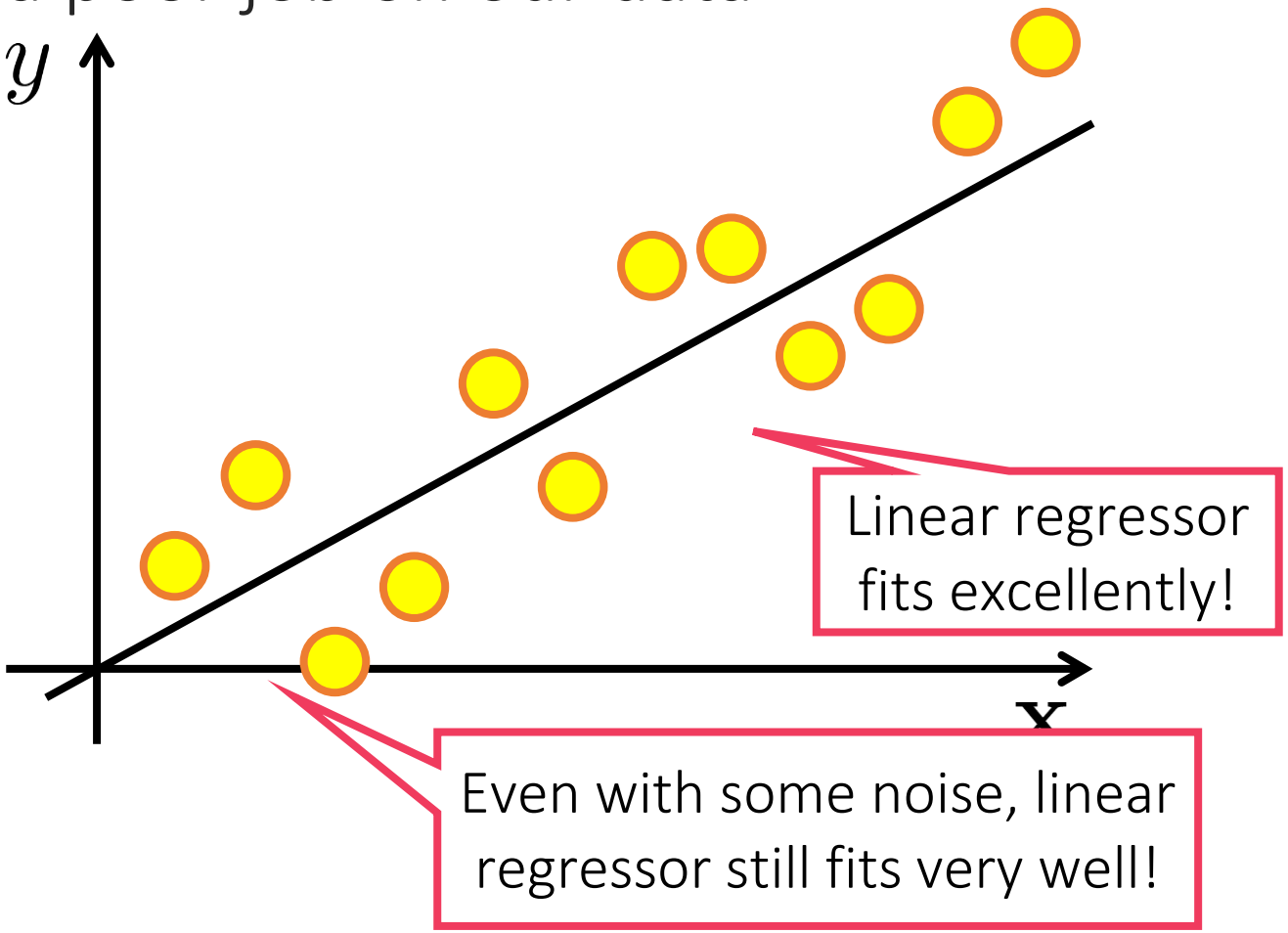
When should I use kernels?

Kernel methods are a good option to try whenever linear models do a poor job on our data



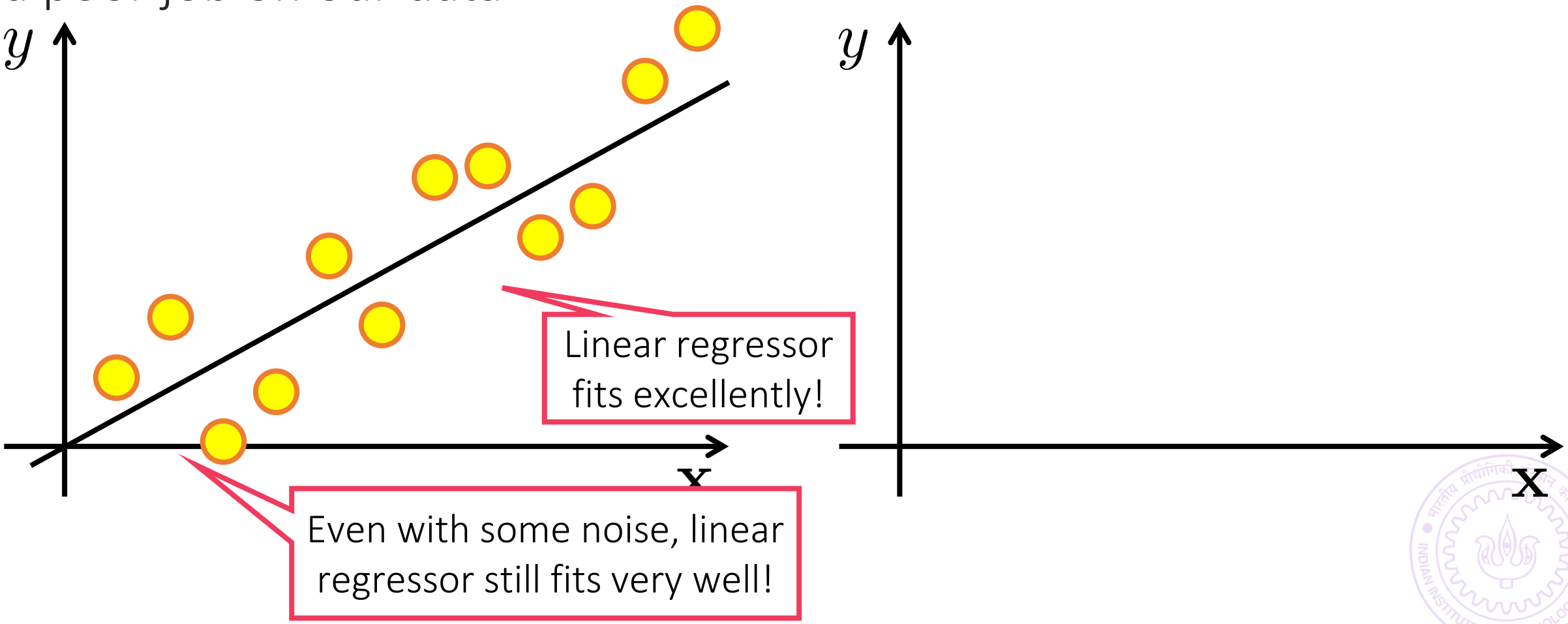
When should I use kernels?

Kernel methods are a good option to try whenever linear models do a poor job on our data



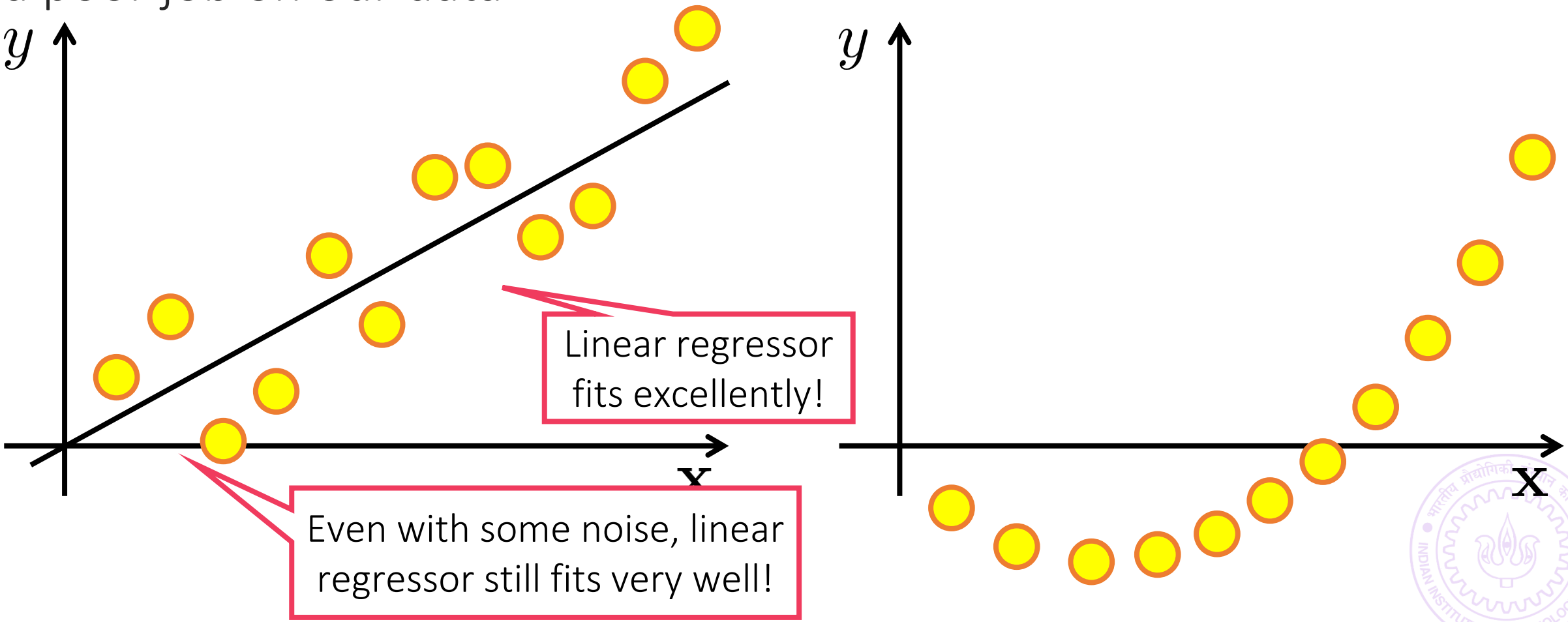
When should I use kernels?

Kernel methods are a good option to try whenever linear models do a poor job on our data



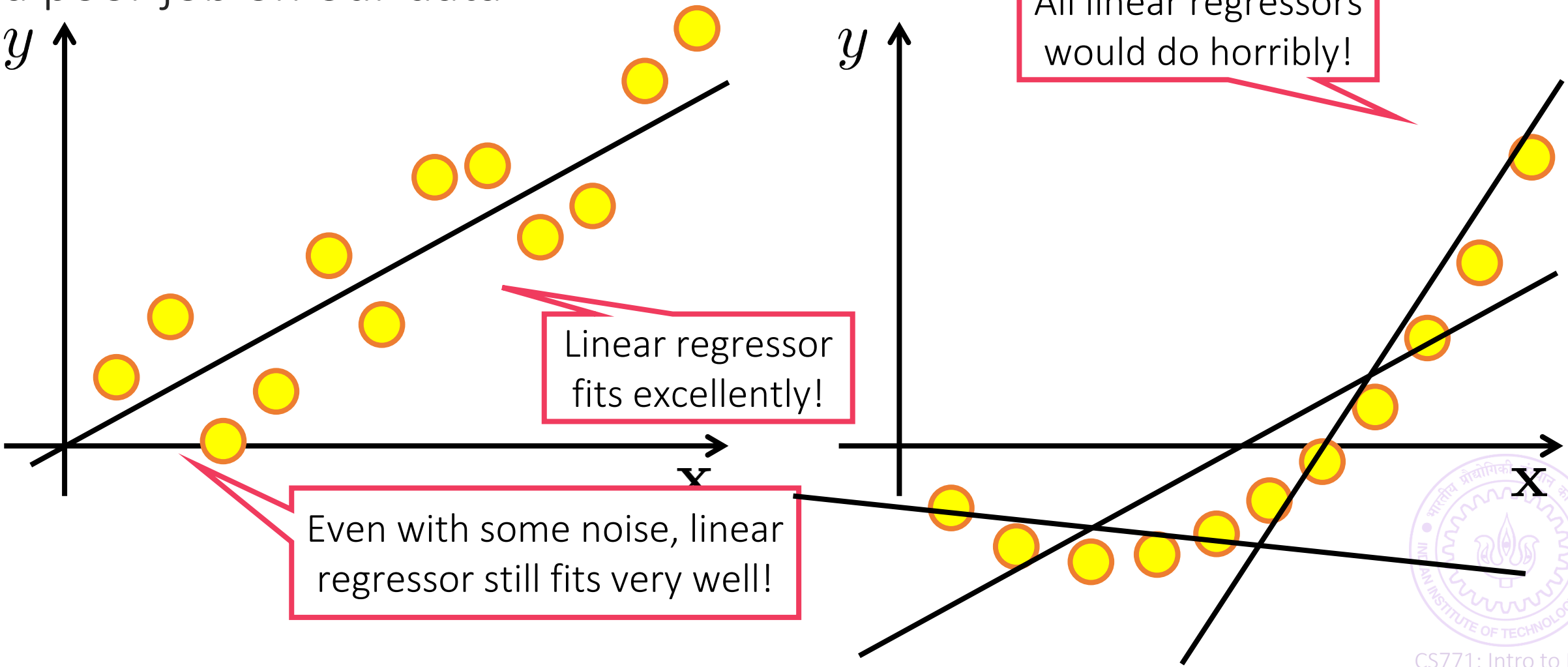
When should I use kernels?

Kernel methods are a good option to try whenever linear models do a poor job on our data



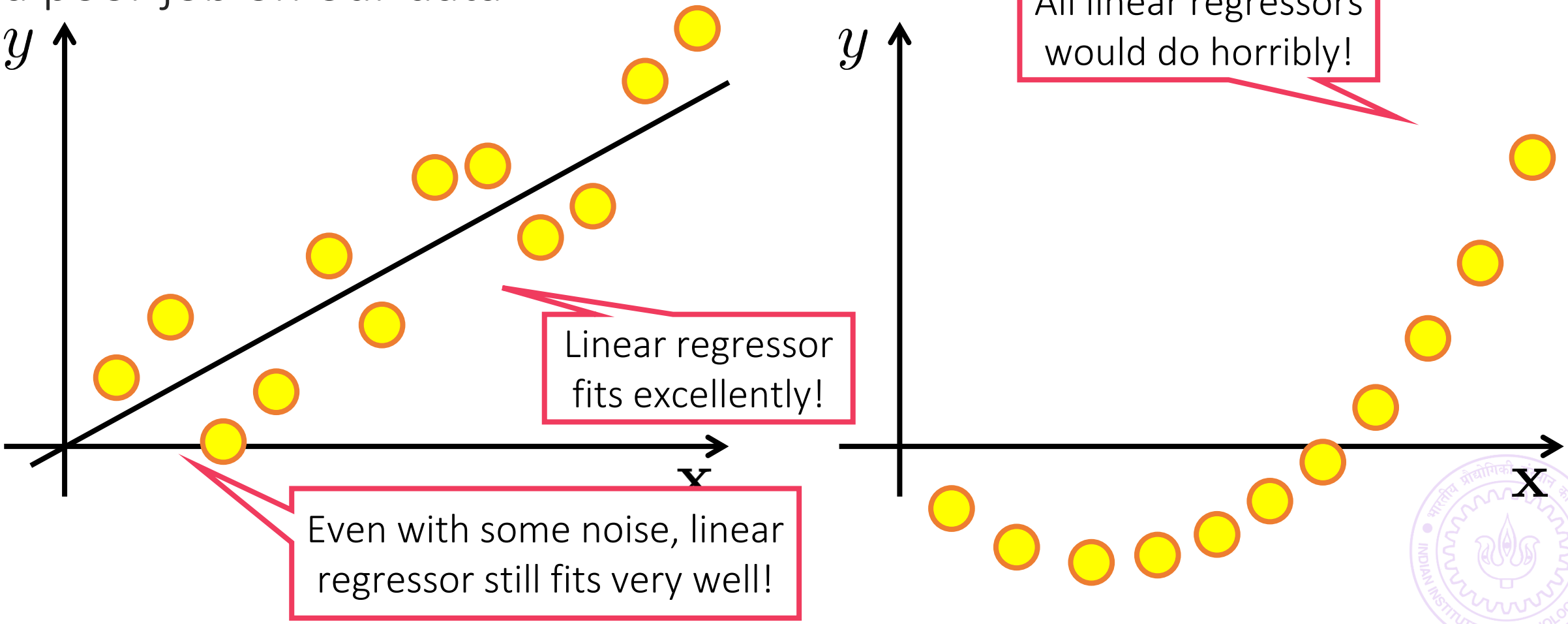
When should I use kernels?

Kernel methods are a good option to try whenever linear models do a poor job on our data



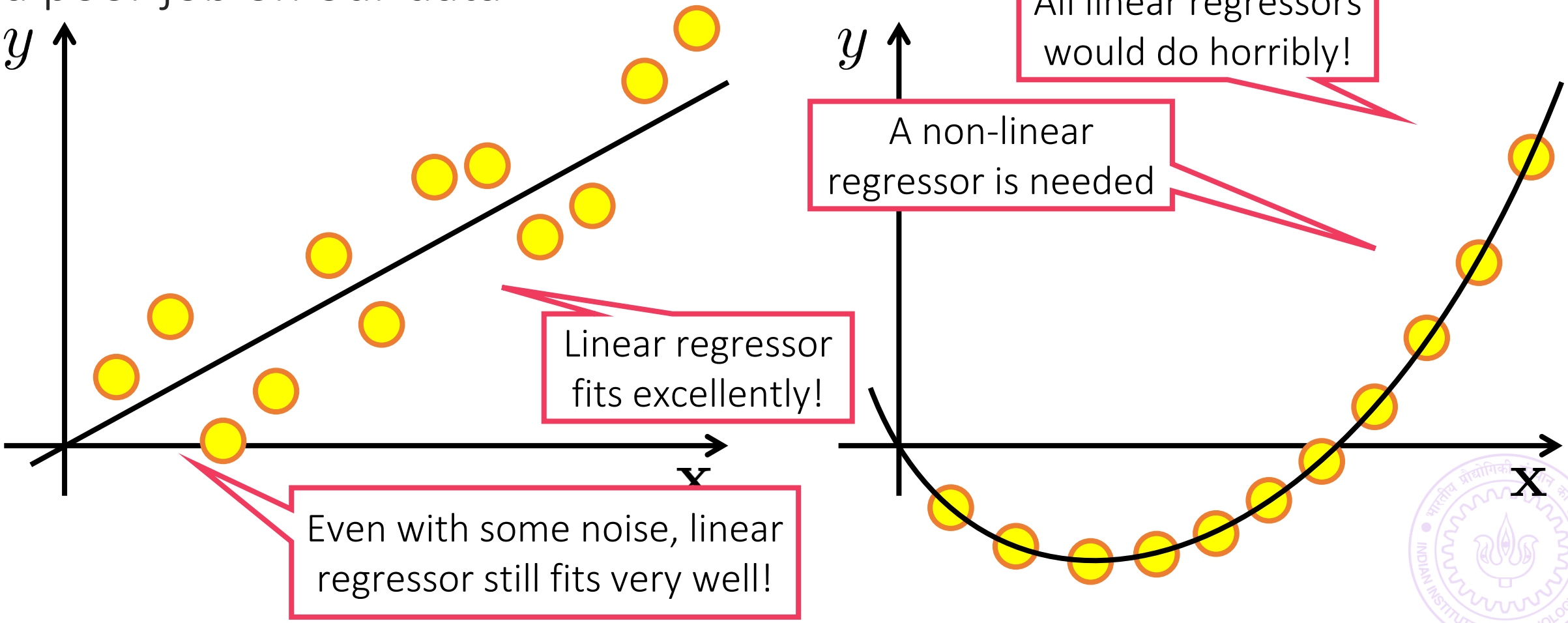
When should I use kernels?

Kernel methods are a good option to try whenever linear models do a poor job on our data



When should I use kernels?

Kernel methods are a good option to try whenever linear models do a poor job on our data

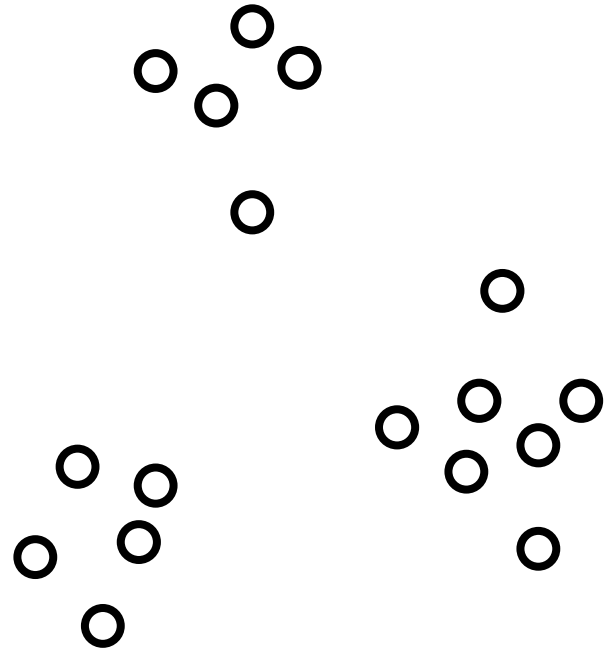


When should I use kernels?

Kernel methods are a good option to try whenever linear models do a poor job on our data

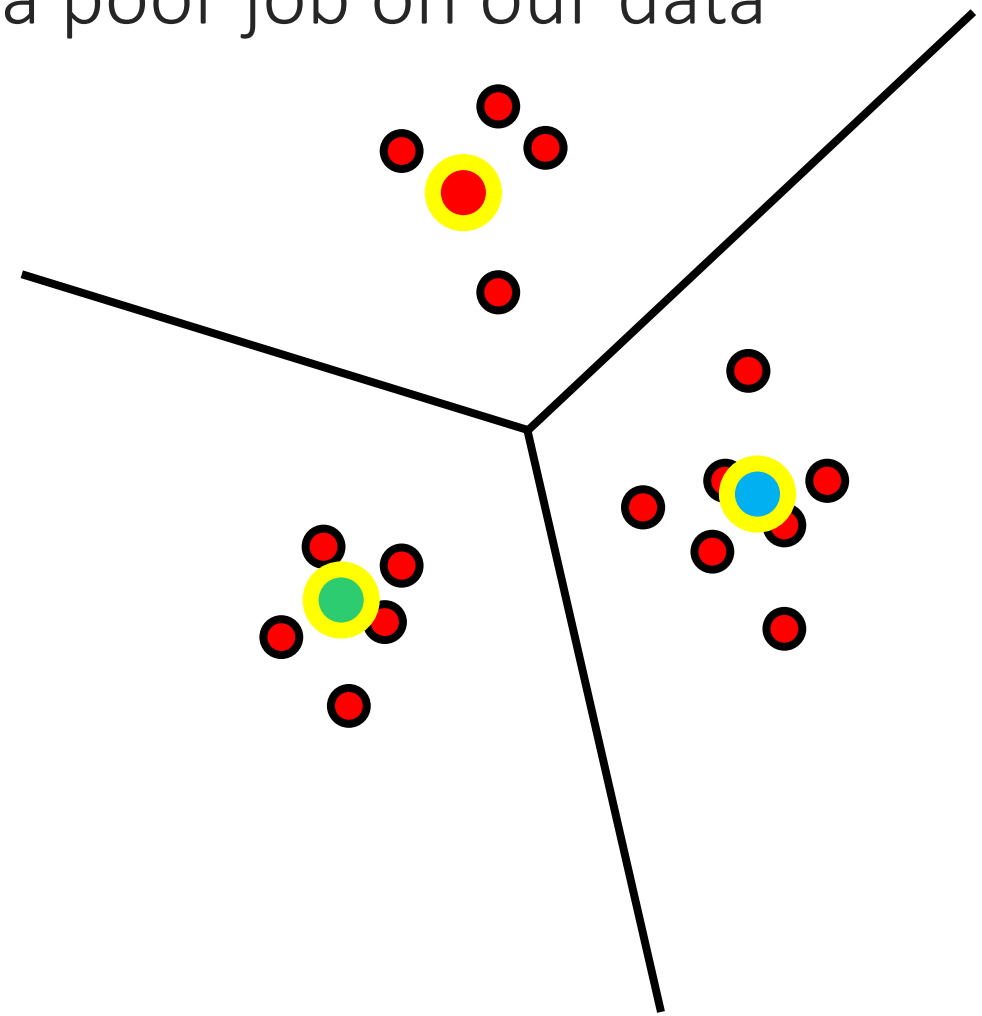
When should I use kernels?

Kernel methods are a good option to try whenever linear models do a poor job on our data



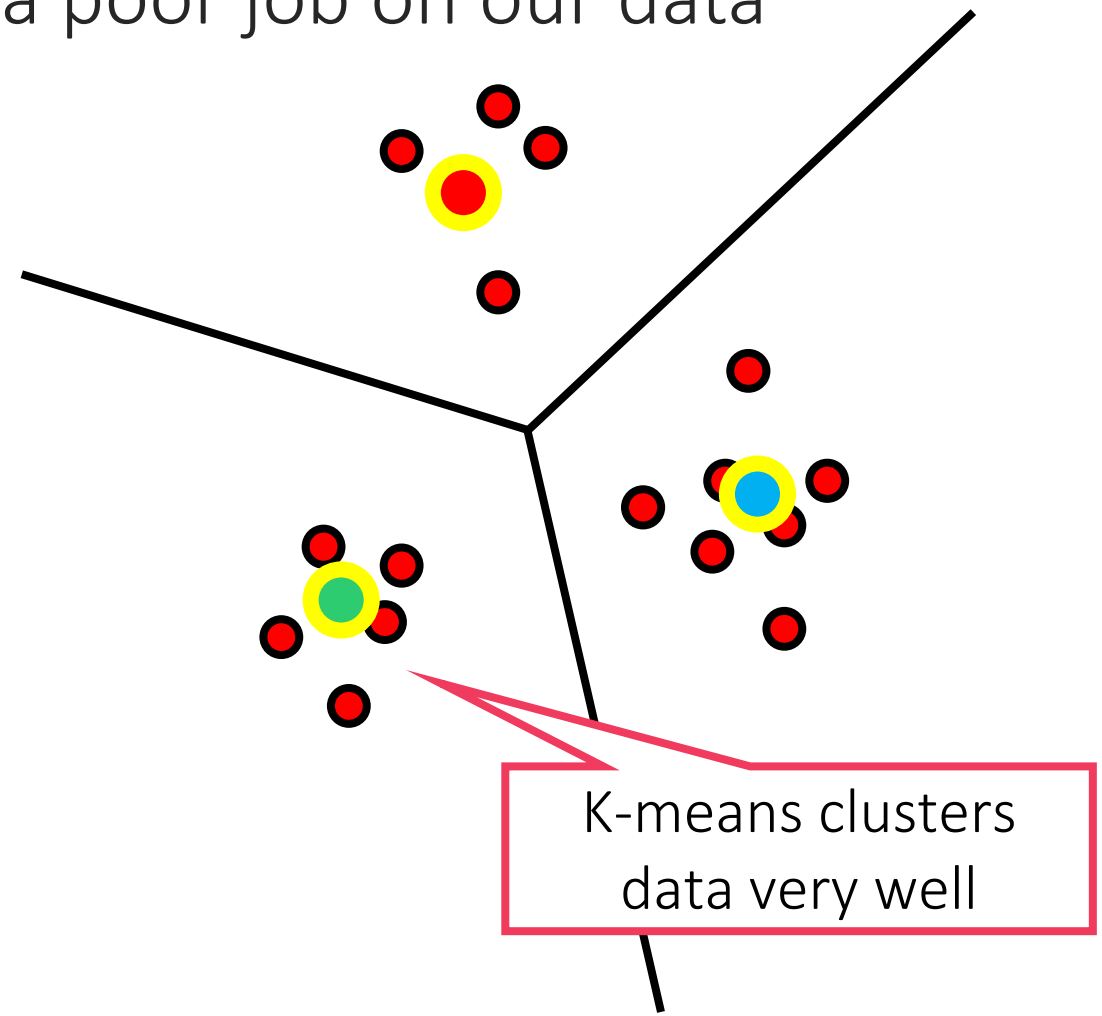
When should I use kernels?

Kernel methods are a good option to try whenever linear models do a poor job on our data



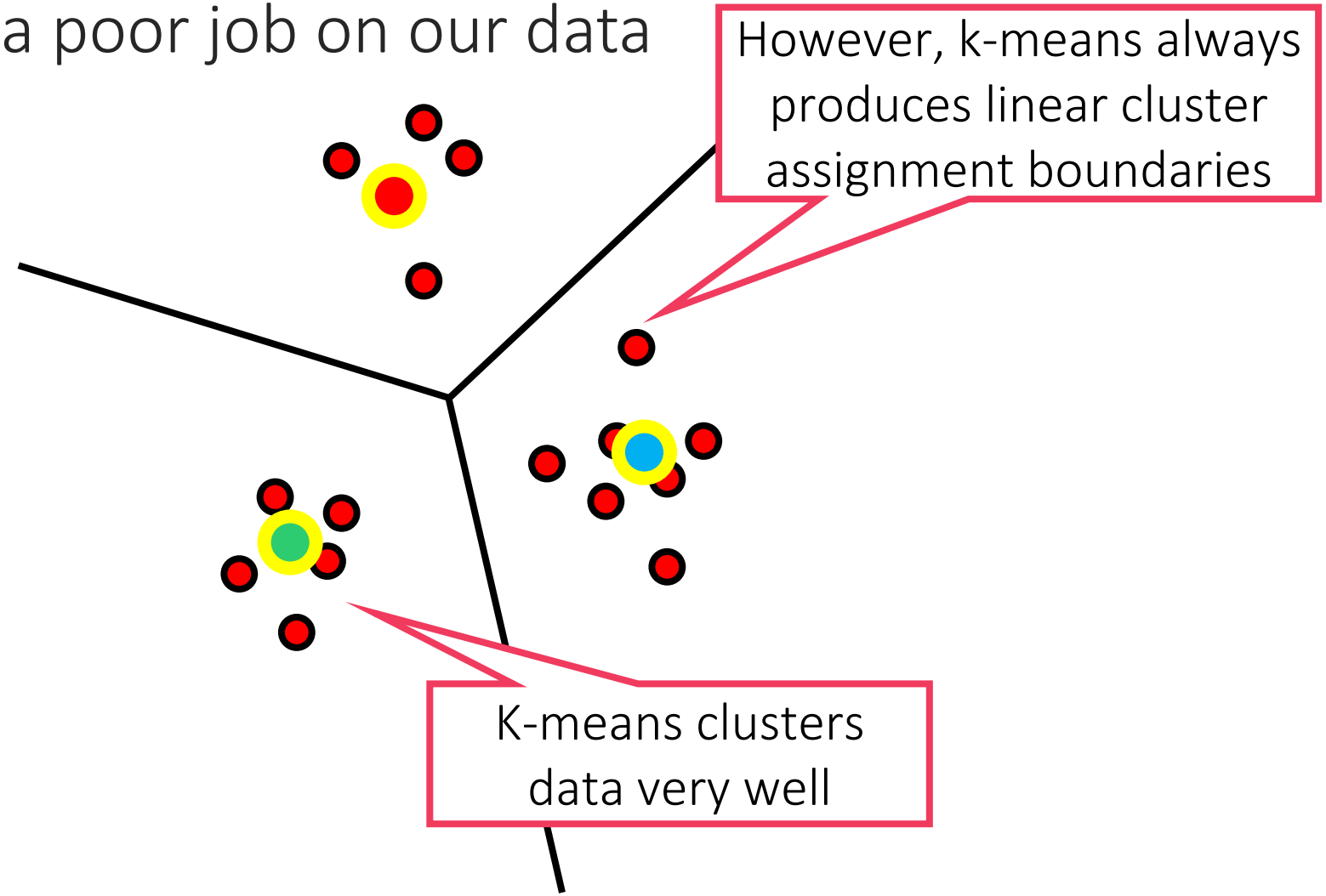
When should I use kernels?

Kernel methods are a good option to try whenever linear models do a poor job on our data



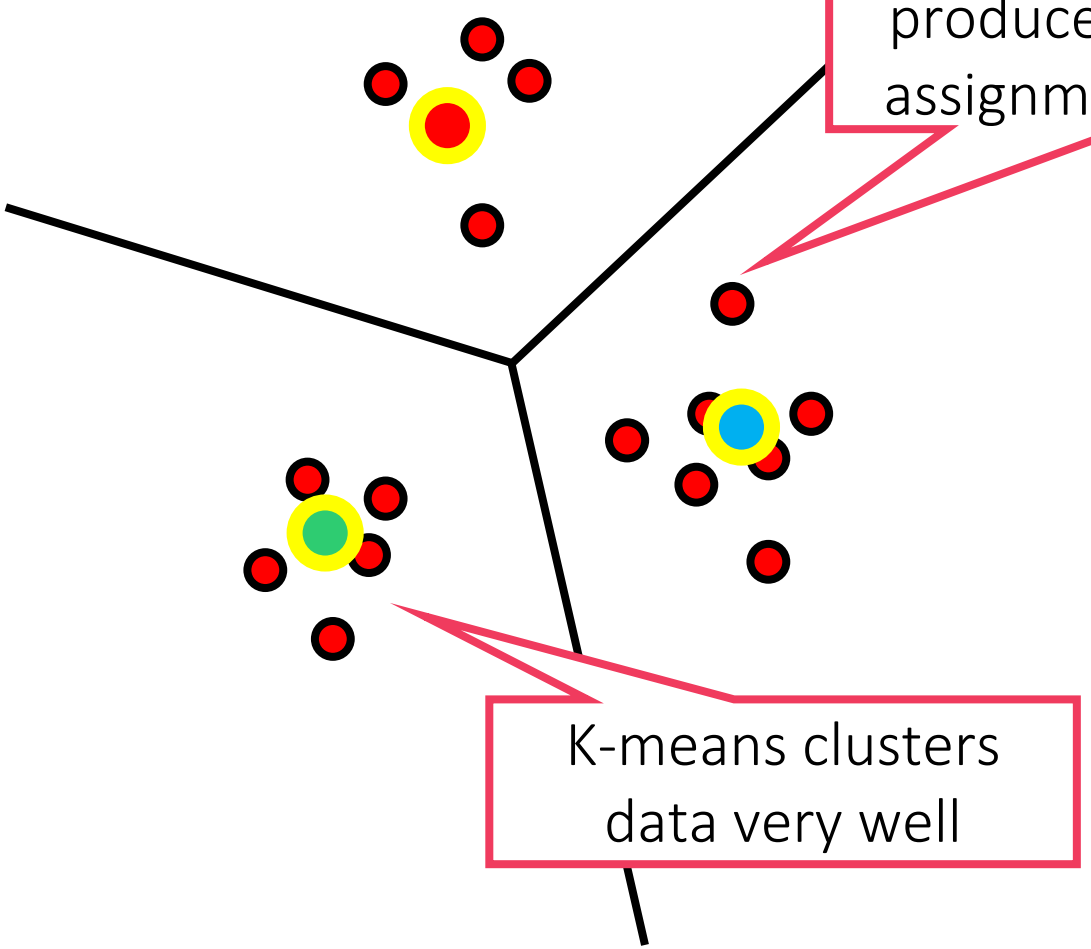
When should I use kernels?

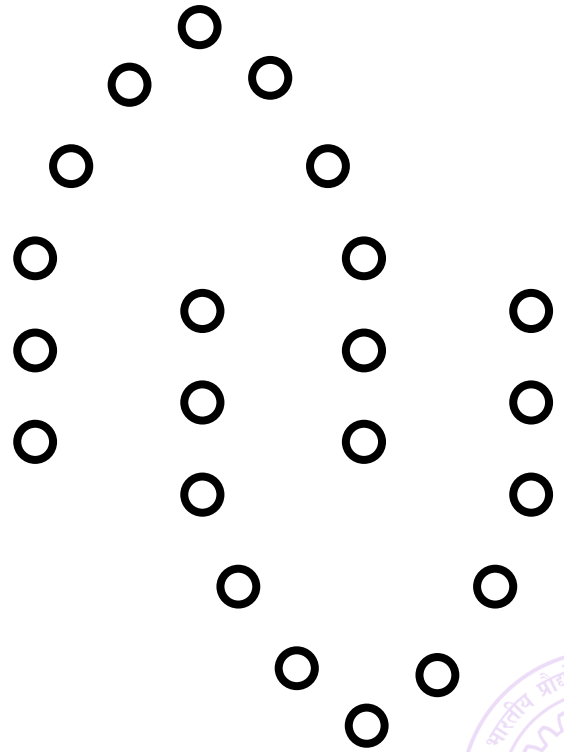
Kernel methods are a good option to try whenever linear models do a poor job on our data



When should I use kernels?

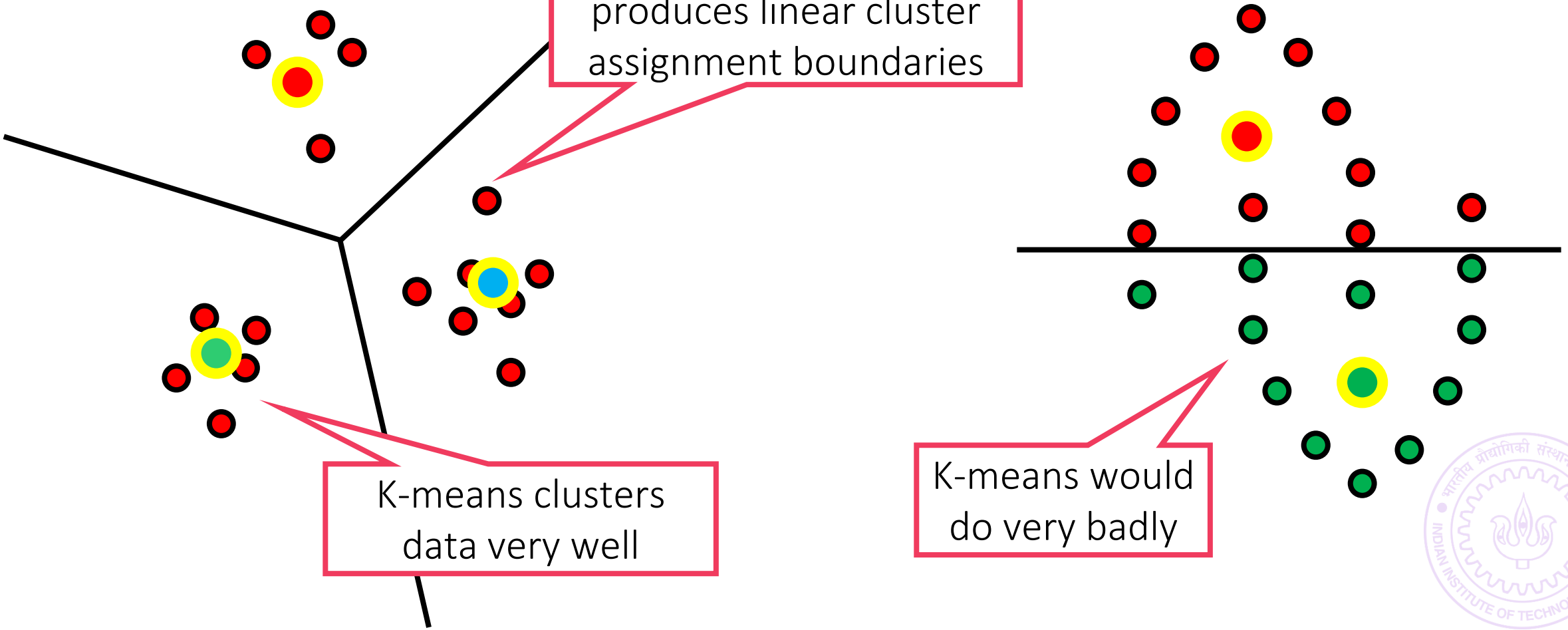
Kernel methods are a good option to try whenever linear models do a poor job on our data





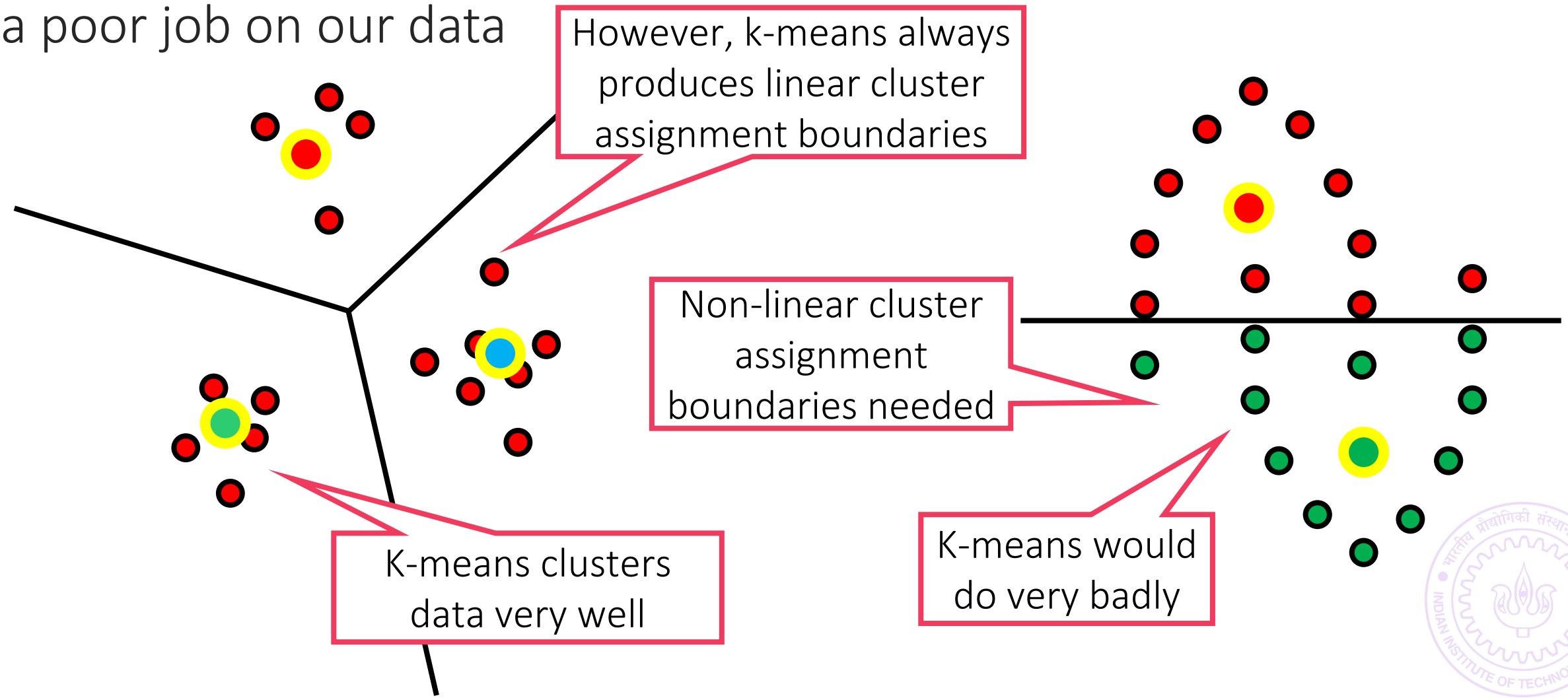
When should I use kernels?

Kernel methods are a good option to try whenever linear models do a poor job on our data



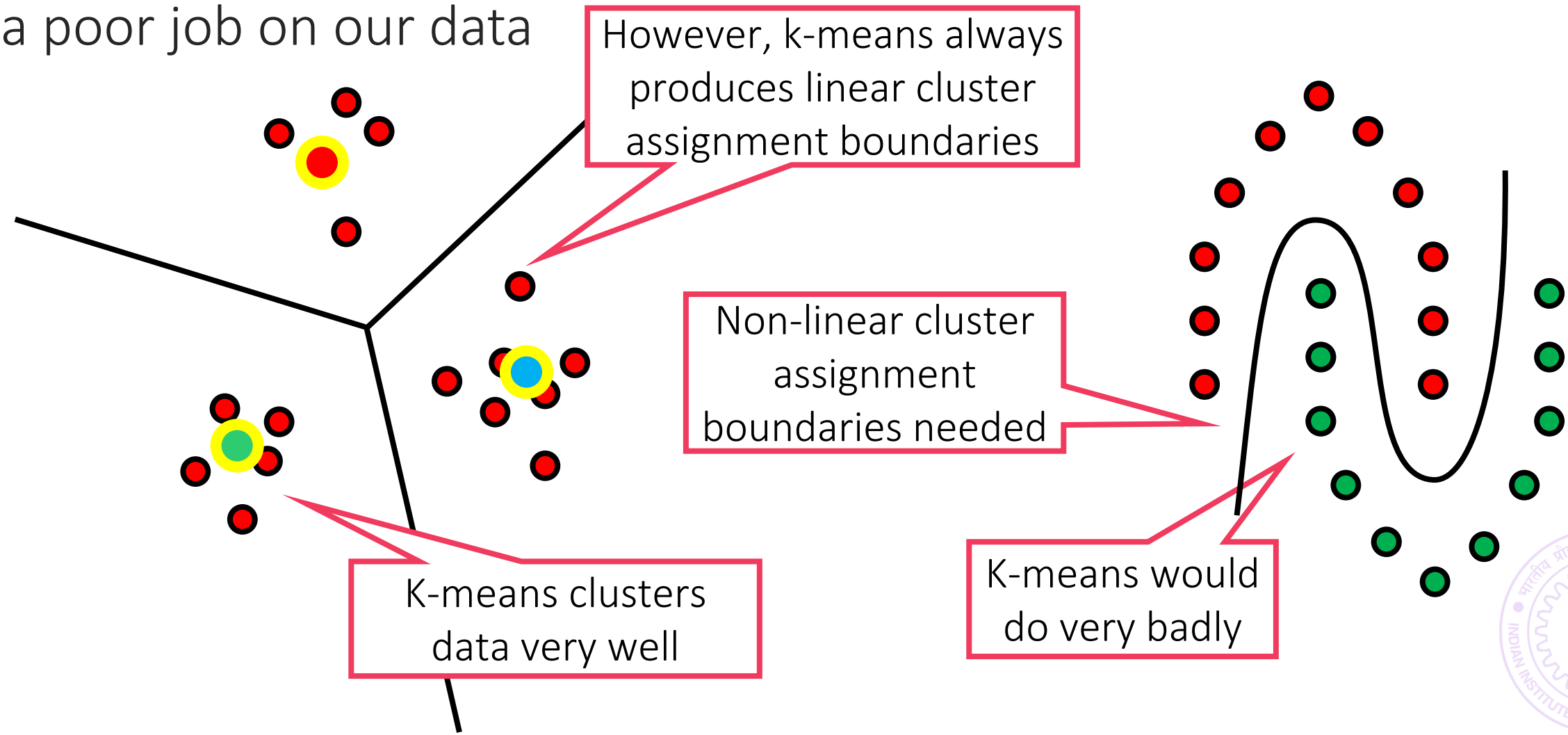
When should I use kernels?

Kernel methods are a good option to try whenever linear models do a poor job on our data



When should I use kernels?

Kernel methods are a good option to try whenever linear models do a poor job on our data

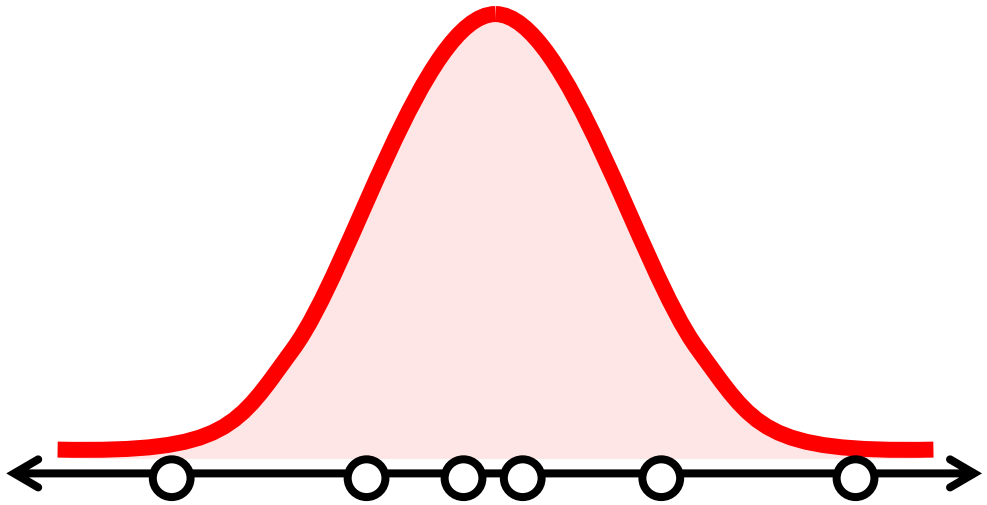


When should I use kernels?

Kernel methods are a good option to try whenever linear models do a poor job on our data

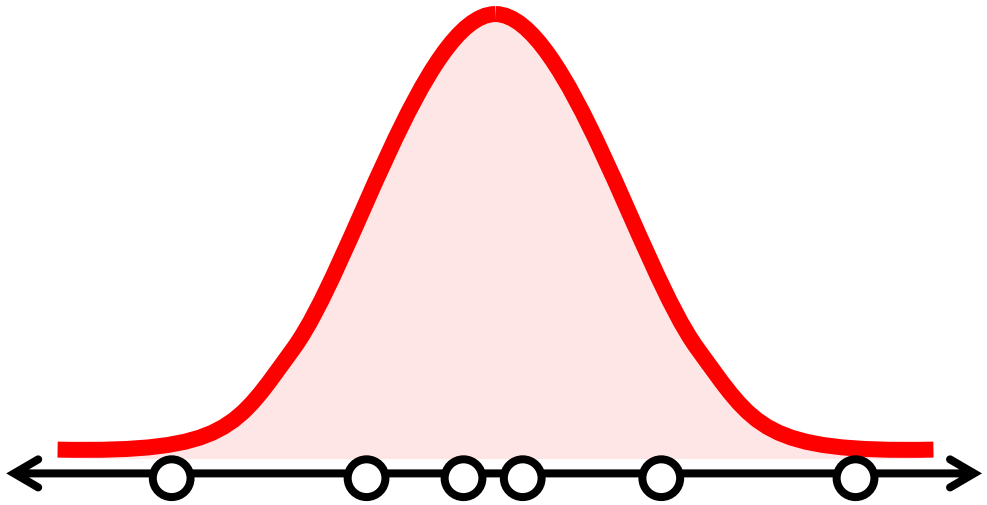
When should I use kernels?

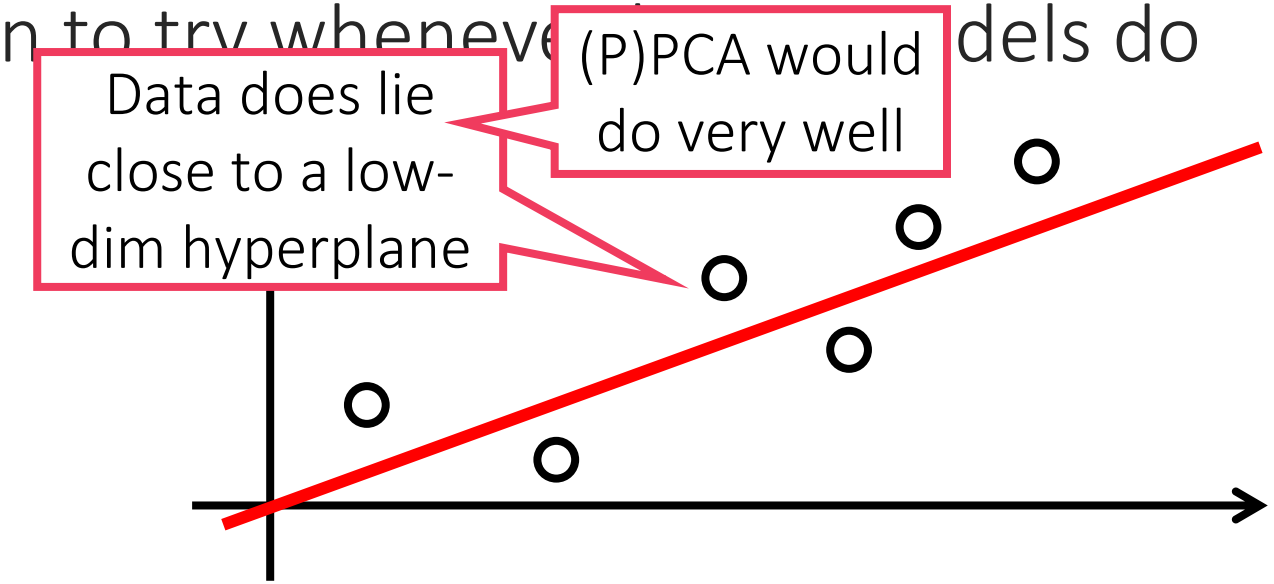
Kernel methods are a good option to try whenever linear models do a poor job on our data



When should I use kernels?

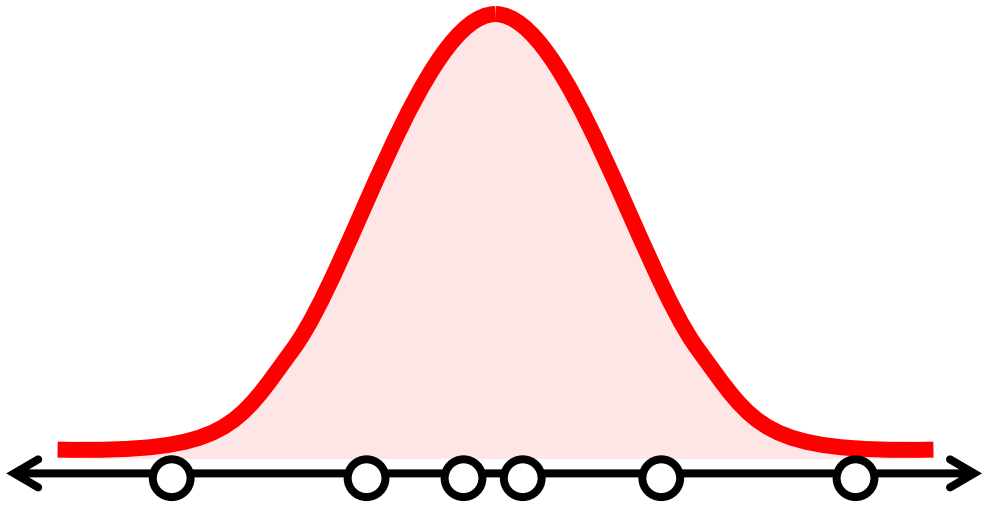
Kernel methods are a good option to try whenever linear models do a poor job on our data

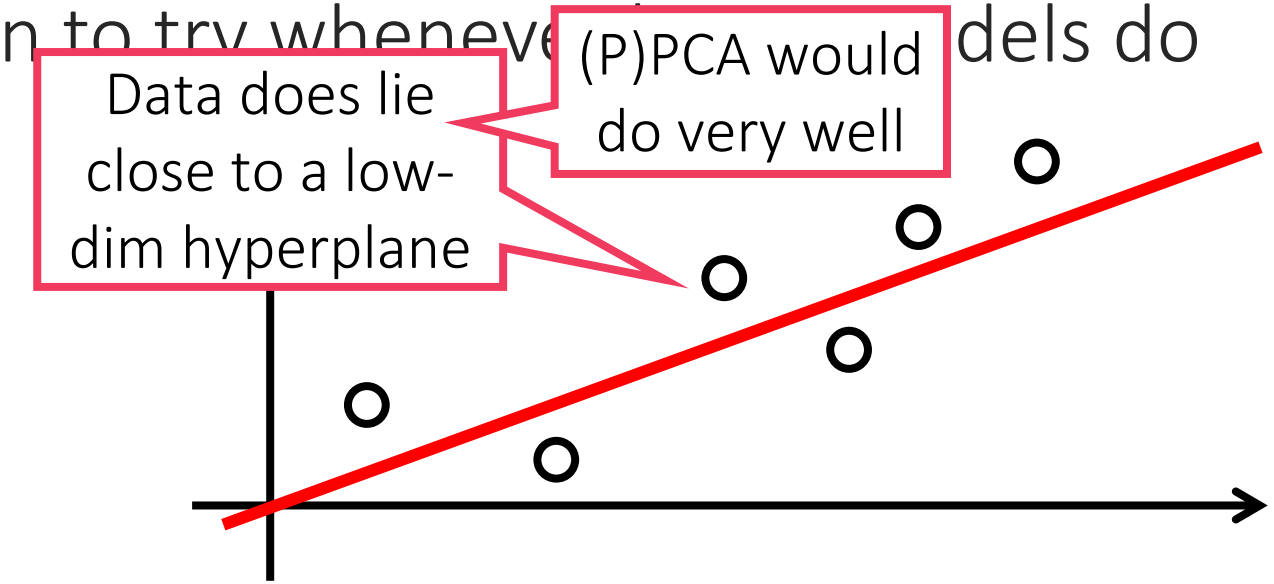


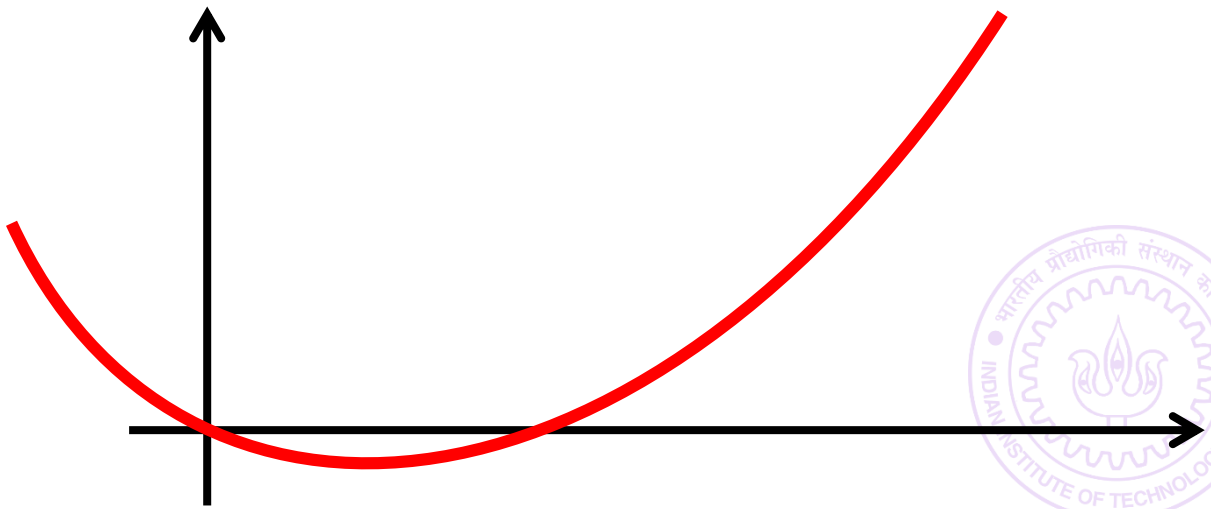


When should I use kernels?

Kernel methods are a good option to try whenever models do a poor job on our data

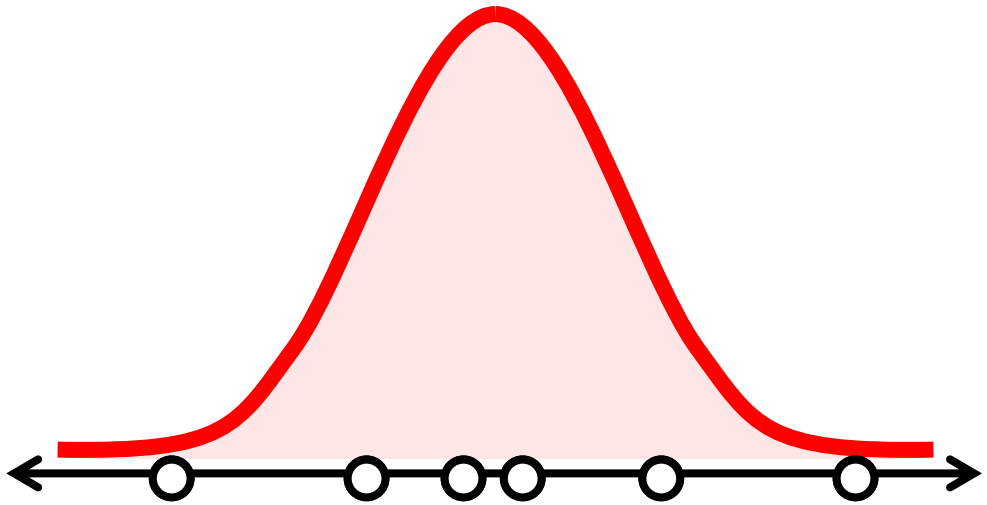


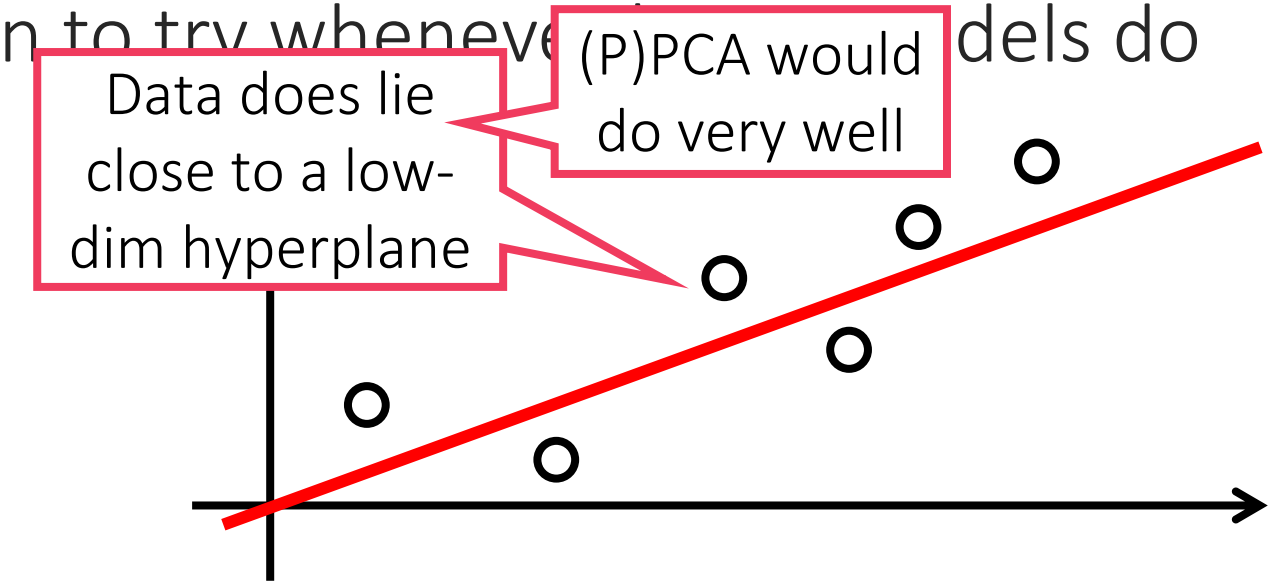


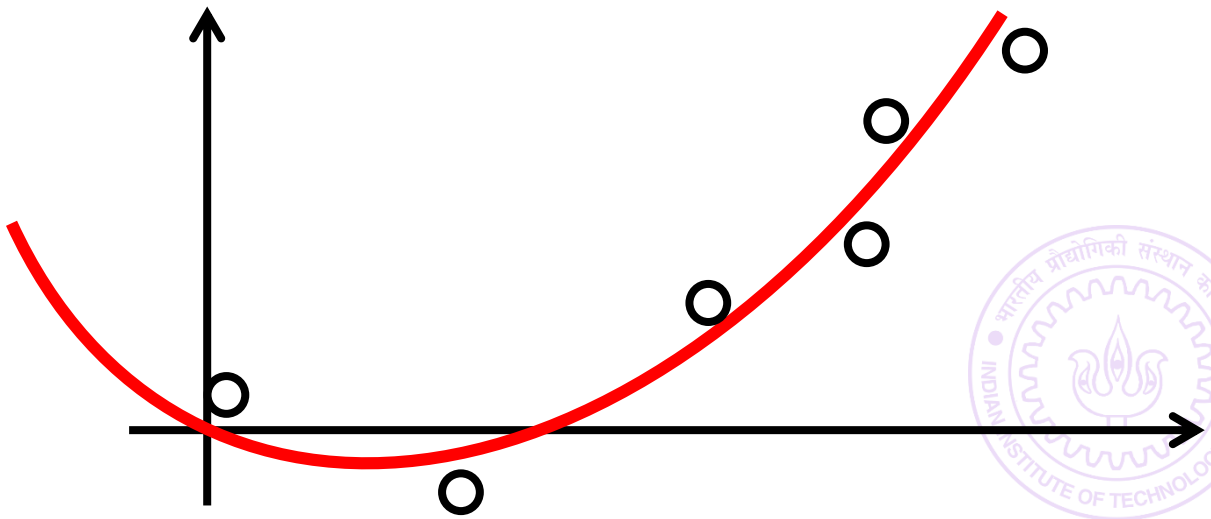


When should I use kernels?

Kernel methods are a good option to try whenever models do a poor job on our data

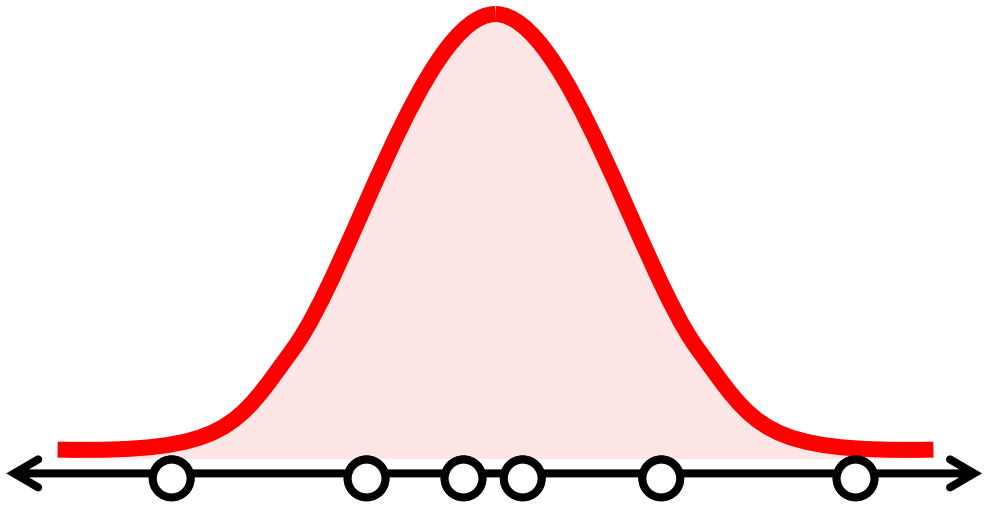


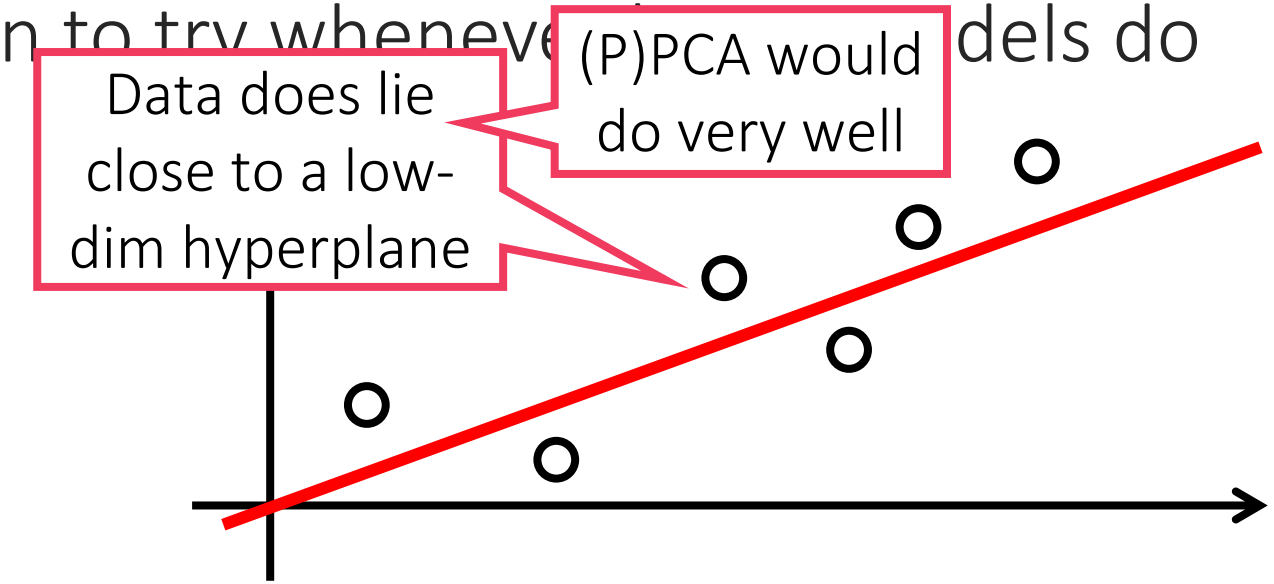




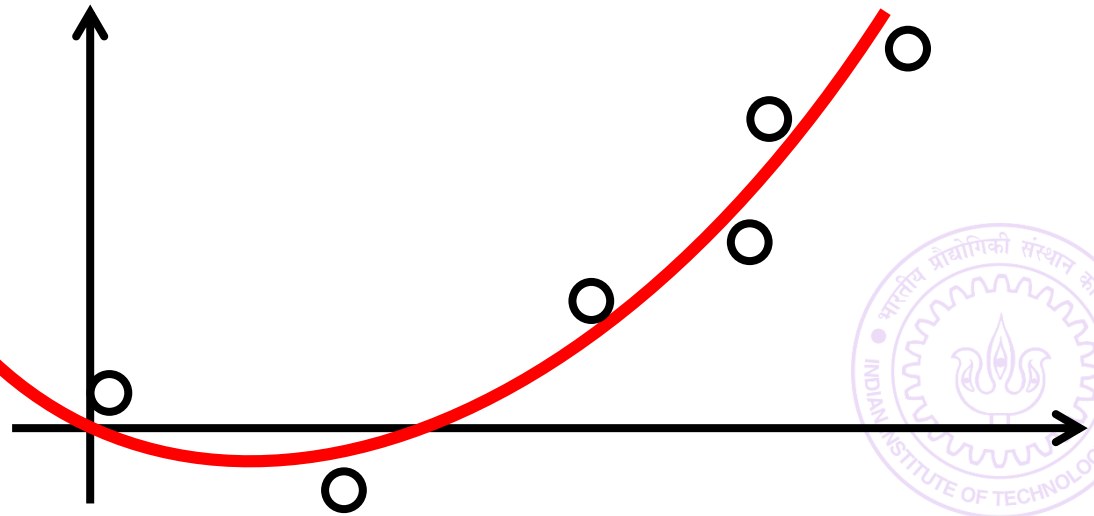
When should I use kernels?

Kernel methods are a good option to try whenever linear models do a poor job on our data



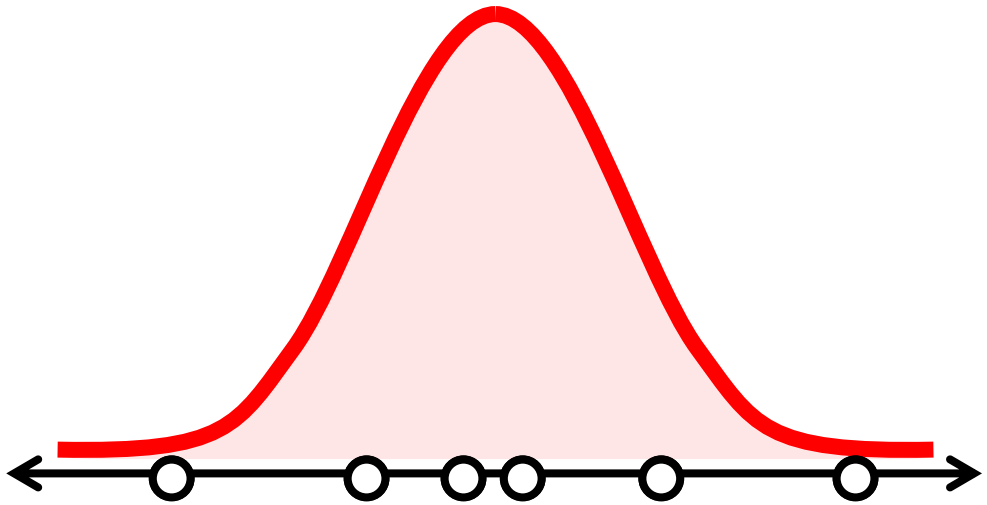


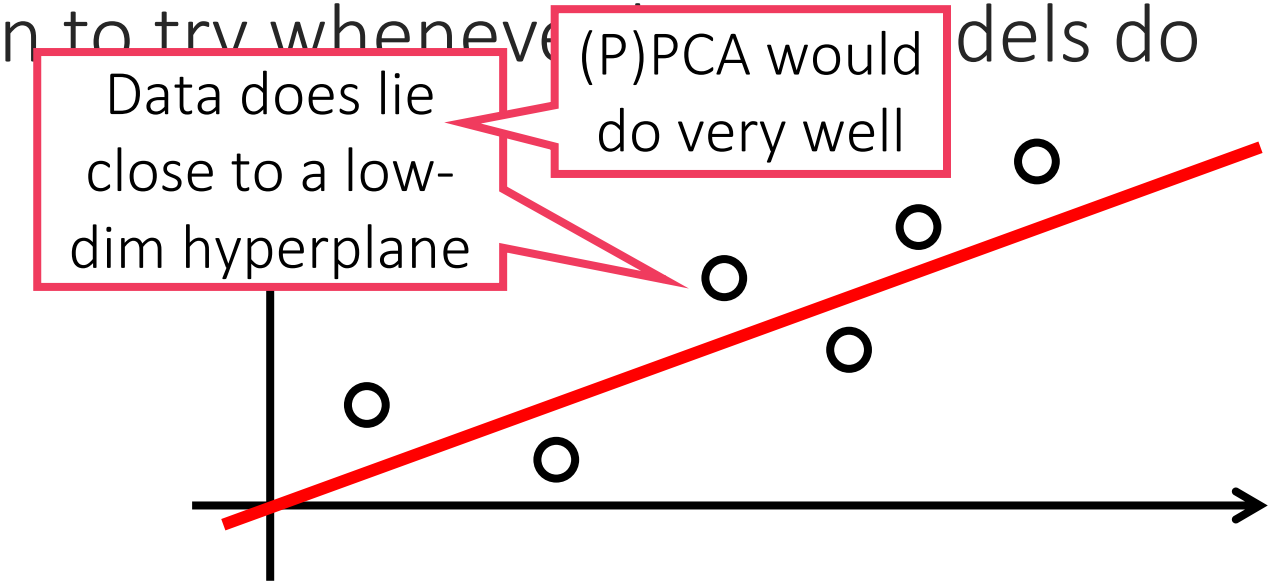
No low-dim hyperplane approximates data well

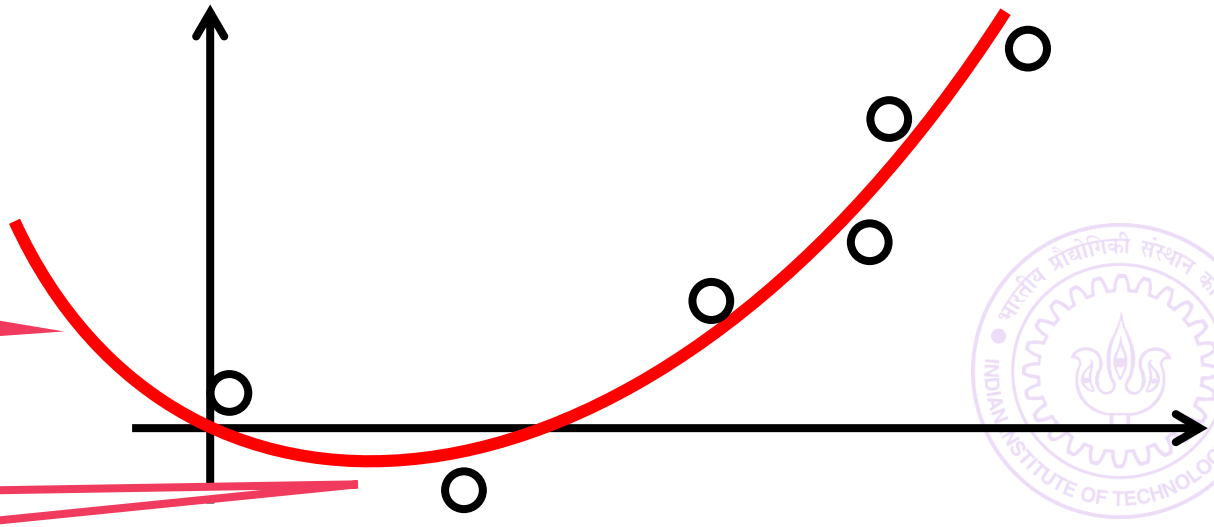


When should I use kernels?

Kernel methods are a good option to try whenever linear models do a poor job on our data

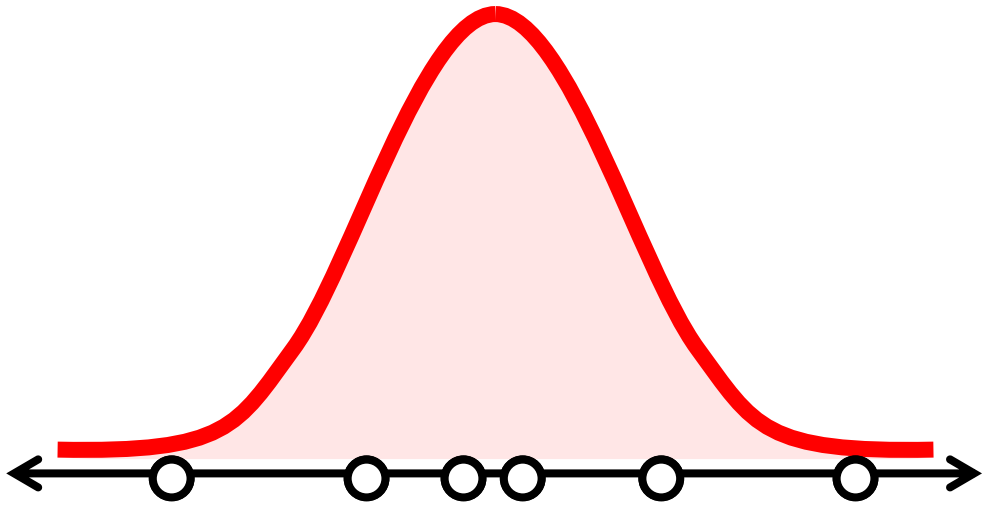






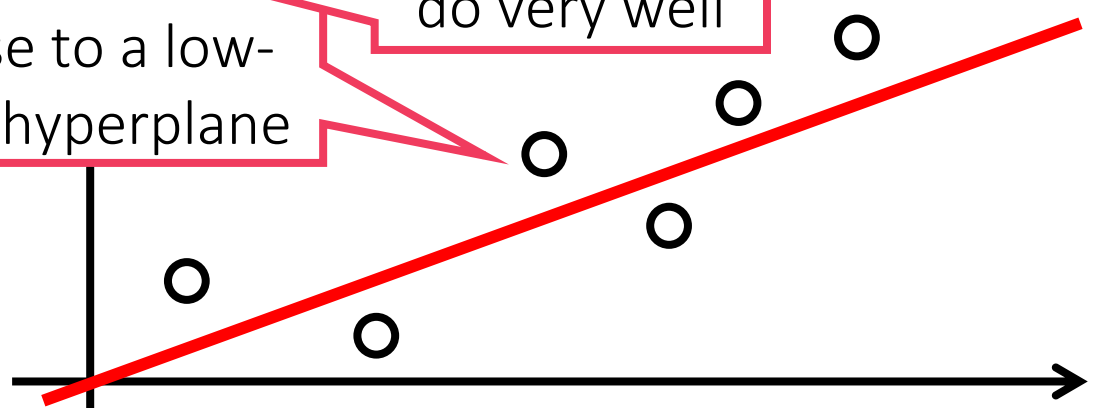
When should I use kernels?

Kernel methods are a good option to try whenever models do a poor job on our data

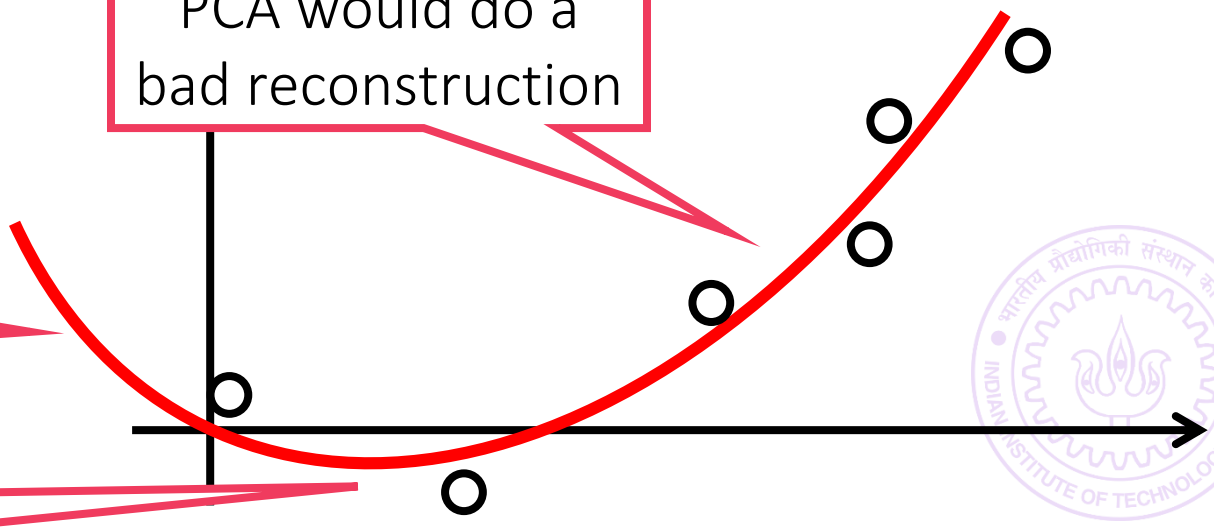


Data does lie close to a low-dim hyperplane

(P)PCA would do very well



PCA would do a bad reconstruction



No low-dim hyperplane approximates data well

Data actually lies close to a smooth low-dim (but non-linear) surface

The Blessing of Dimensionality

High dimensionality is often criticized for the hardships it brings

Algorithms become expensive, kNN behaves weirdly, much more training data is needed to learn good models ☹

Collectively called the "curse of dimensionality"

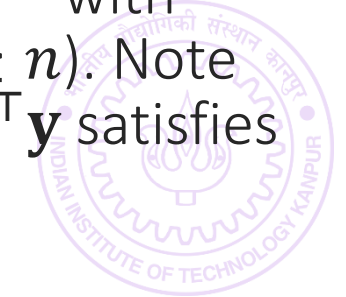
However, large dimensionality also allows powerful and flexible models

As an extreme case, consider n feature vectors in $d \geq n$ dimensions (not a typo) with all n feature vectors linearly independent i.e. $\text{rank}(X) = n$

Easy to see that perfect classification, regression is possible for this data

Proof: let $\mathbf{y}$ be label vector $\mathbf{y} \in \{-1,1\}^n$ for classfn, $\mathbf{y} \in \mathbb{R}^n$ for regression. Let $X = U\Sigma V^\top$ be the (thin) SVD of the feature matrix X with $U \in \mathbb{R}^{n\times n}, \Sigma \in \mathbb{R}^{n\times n}, V \in \mathbb{R}^{d\times n}$ with $V^\top V = I_d = UU^\top = U^\top U$ (be careful that we may have $VV^\top \neq I_n$ since $d \geq n$). Note that Σ is invertible since we have the thin SVD. Then, the model $\mathbf{w} = V\Sigma^{-1}U^\top\mathbf{y}$ satisfies $X\mathbf{w} = \mathbf{y}$ i.e. perfect learning. To see why, note that

$$X\mathbf{w} = U\Sigma V^\top V\Sigma^{-1}U^\top\mathbf{y} = U\Sigma\Sigma^{-1}U^\top\mathbf{y} = UU^\top\mathbf{y} = \mathbf{y}$$



Non-linear Classification

Original data non-separable by any linear classifier but using a nice non-linear map makes them separable with a margin!

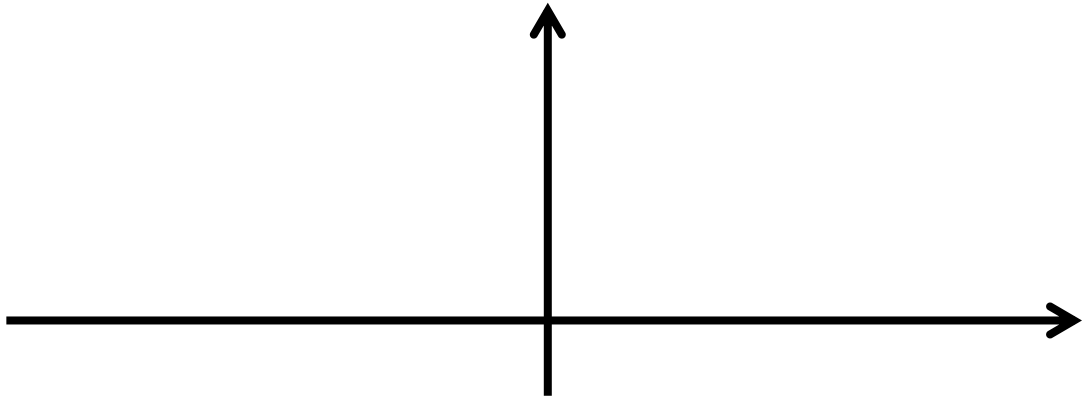
Linear maps could have never accomplished this. In fact, not all non-linear maps guaranteed to do so either, only nice ones would do this

A good decision boundary is $\phi_2(x) = c$ i.e. $|x| = c$ since $\phi_2(x) = |x|$

Decision boundary is non-lin in orig space $\mathbb{R}$ i.e lin model in $\mathbb{R}^2$ imposed non-lin boundary in $\mathbb{R}$

Non-linearity came from map ϕ. There exists no $W \in \mathbb{R}^{2\times 1}$ so that $\phi(x) = Wx$ for all $x \in \mathbb{R}$

Non-linear Classification



Original data non-separable by any linear classifier but using a nice non-linear map makes them separable with a margin!

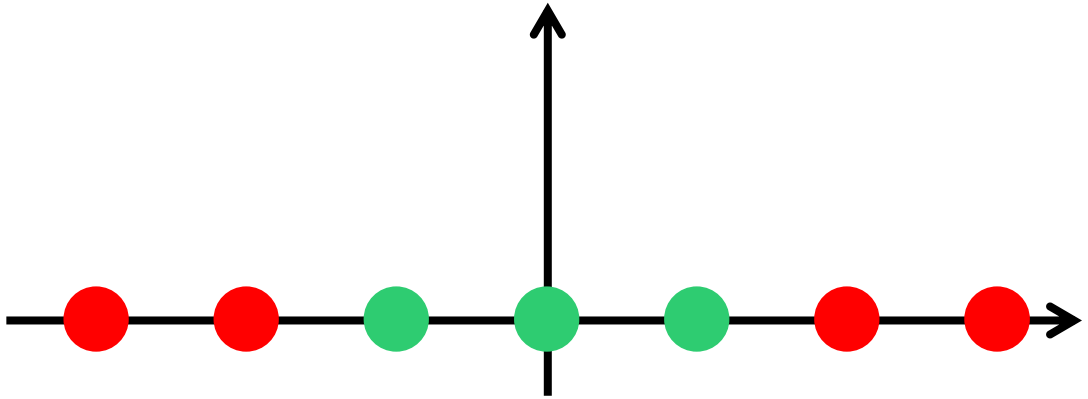
Linear maps could have never accomplished this. In fact, not all non-linear maps guaranteed to do so either, only nice ones would do this

A good decision boundary is $\phi_2(x) = c$ i.e. $|x| = c$ since $\phi_2(x) = |x|$

Decision boundary is non-lin in orig space $\mathbb{R}$ i.e lin model in $\mathbb{R}^2$ imposed non-lin boundary in $\mathbb{R}$

Non-linearity came from map ϕ. There exists no $W \in \mathbb{R}^{2\times 1}$ so that $\phi(x) = Wx$ for all $x \in \mathbb{R}$

Non-linear Classification



Original data non-separable by any linear classifier but using a nice non-linear map makes them separable with a margin!

Linear maps could have never accomplished this. In fact, not all non-linear maps guaranteed to do so either, only nice ones would do this

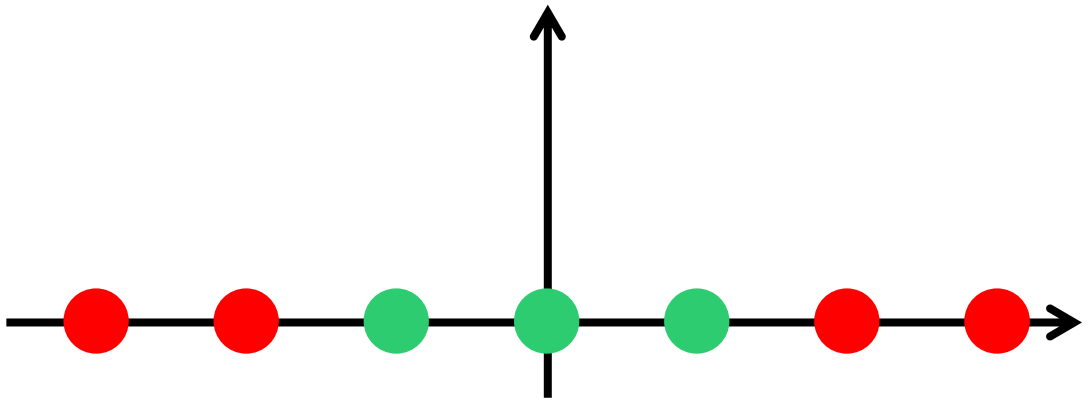
A good decision boundary is $\phi_2(x) = c$ i.e. $|x| = c$ since $\phi_2(x) = |x|$

Decision boundary is non-lin in orig space $\mathbb{R}$ i.e lin model in $\mathbb{R}^2$ imposed non-lin boundary in $\mathbb{R}$

Non-linearity came from map ϕ. There exists no $W \in \mathbb{R}^{2\times 1}$ so that $\phi(x) = Wx$ for all $x \in \mathbb{R}$

Non-linear Classification

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



Original data non-separable by any linear classifier but using a nice non-linear map makes them separable with a margin!

Linear maps could have never accomplished this. In fact, not all non-linear maps guaranteed to do so either, only nice ones would do this

A good decision boundary is $\phi_2(x) = c$ i.e. $|x| = c$ since $\phi_2(x) = |x|$

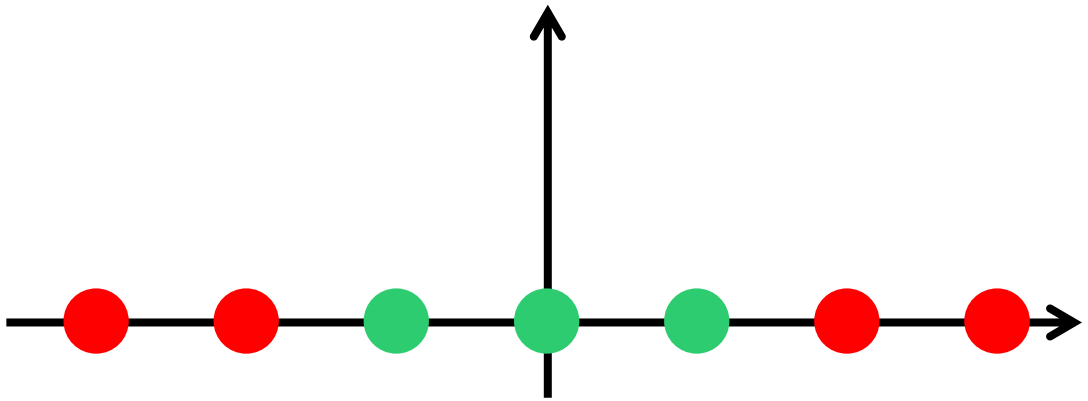
Decision boundary is non-lin in orig space $\mathbb{R}$ i.e lin model in $\mathbb{R}^2$ imposed non-lin boundary in $\mathbb{R}$

Non-linearity came from map ϕ. There exists no $W \in \mathbb{R}^{2\times 1}$ so that $\phi(x) = Wx$ for all $x \in \mathbb{R}$

Non-linear Classification

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

$$\phi(x) = [\phi_1(x), \phi_2(x)], \phi_1(x) = x, \phi_2(x) = |x|$$



Original data non-separable by any linear classifier but using a nice non-linear map makes them separable with a margin!

Linear maps could have never accomplished this. In fact, not all non-linear maps guaranteed to do so either, only nice ones would do this

A good decision boundary is $\phi_2(x) = c$ i.e. $|x| = c$ since $\phi_2(x) = |x|$

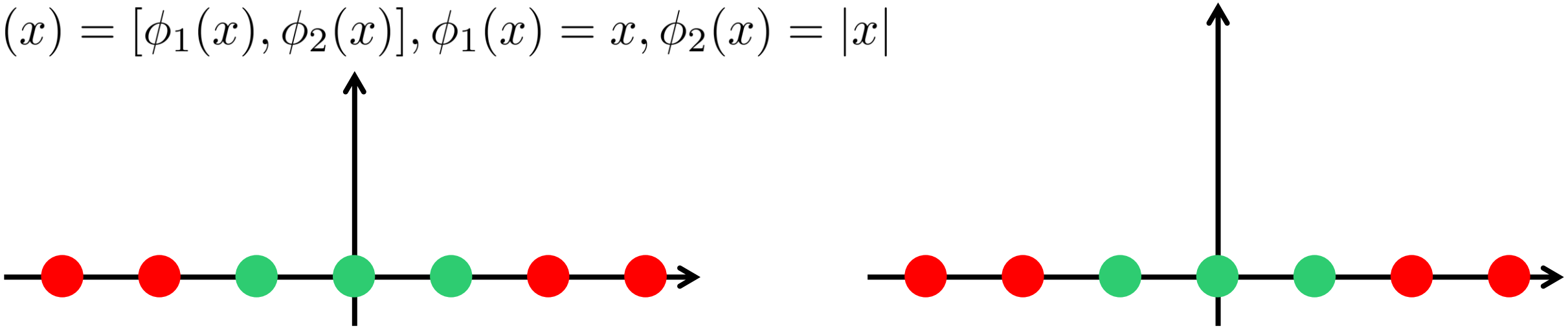
Decision boundary is non-lin in orig space $\mathbb{R}$ i.e lin model in $\mathbb{R}^2$ imposed non-lin boundary in $\mathbb{R}$

Non-linearity came from map ϕ. There exists no $W \in \mathbb{R}^{2\times1}$ so that $\phi(x) = Wx$ for all $x \in \mathbb{R}$

Non-linear Classification

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

$$\phi(x) = [\phi_1(x), \phi_2(x)], \phi_1(x) = x, \phi_2(x) = |x|$$



Original data non-separable by any linear classifier but using a nice non-linear map makes them separable with a margin!

Linear maps could have never accomplished this. In fact, not all non-linear maps guaranteed to do so either, only nice ones would do this

A good decision boundary is $\phi_2(x) = c$ i.e. $|x| = c$ since $\phi_2(x) = |x|$

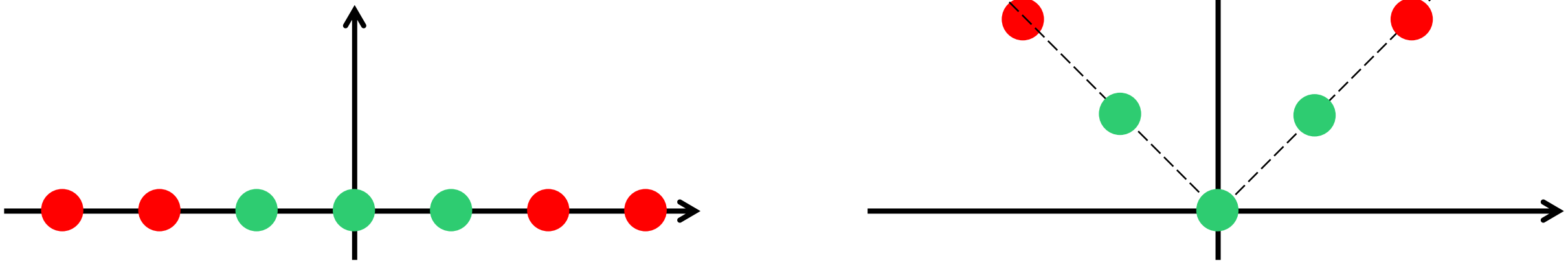
Decision boundary is non-lin in orig space $\mathbb{R}$ i.e lin model in $\mathbb{R}^2$ imposed non-lin boundary in $\mathbb{R}$

Non-linearity came from map ϕ. There exists no $W \in \mathbb{R}^{2\times1}$ so that $\phi(x) = Wx$ for all $x \in \mathbb{R}$

Non-linear Classification

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

$$\phi(x) = [\phi_1(x), \phi_2(x)], \phi_1(x) = x, \phi_2(x) = |x|$$



Original data non-separable by any linear classifier but using a nice non-linear map makes them separable with a margin!

Linear maps could have never accomplished this. In fact, not all non-linear maps guaranteed to do so either, only nice ones would do this

A good decision boundary is $\phi_2(x) = c$ i.e. $|x| = c$ since $\phi_2(x) = |x|$

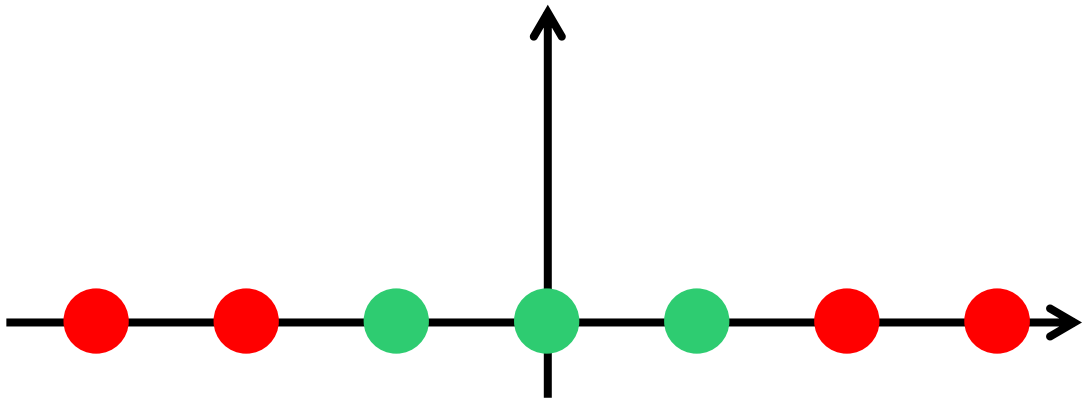
Decision boundary is non-lin in orig space $\mathbb{R}$ i.e lin model in $\mathbb{R}^2$ imposed non-lin boundary in $\mathbb{R}$

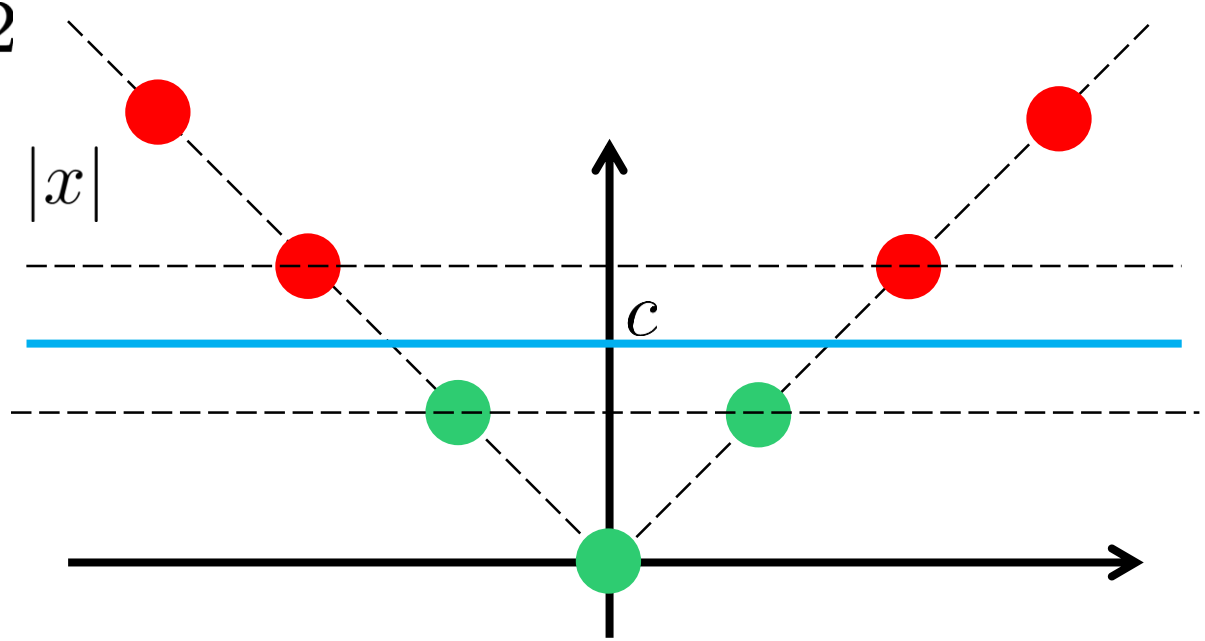
Non-linearity came from map ϕ. There exists no $W \in \mathbb{R}^{2\times1}$ so that $\phi(x) = Wx$ for all $x \in \mathbb{R}$

Non-linear Classification

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

$$\phi(x) = [\phi_1(x), \phi_2(x)], \phi_1(x) = x, \phi_2(x) = |x|$$





Original data non-separable by any linear classifier but using a nice non-linear map makes them separable with a margin!

Linear maps could have never accomplished this. In fact, not all non-linear maps guaranteed to do so either, only nice ones would do this

A good decision boundary is $\phi_2(x) = c$ i.e. $|x| = c$ since $\phi_2(x) = |x|$

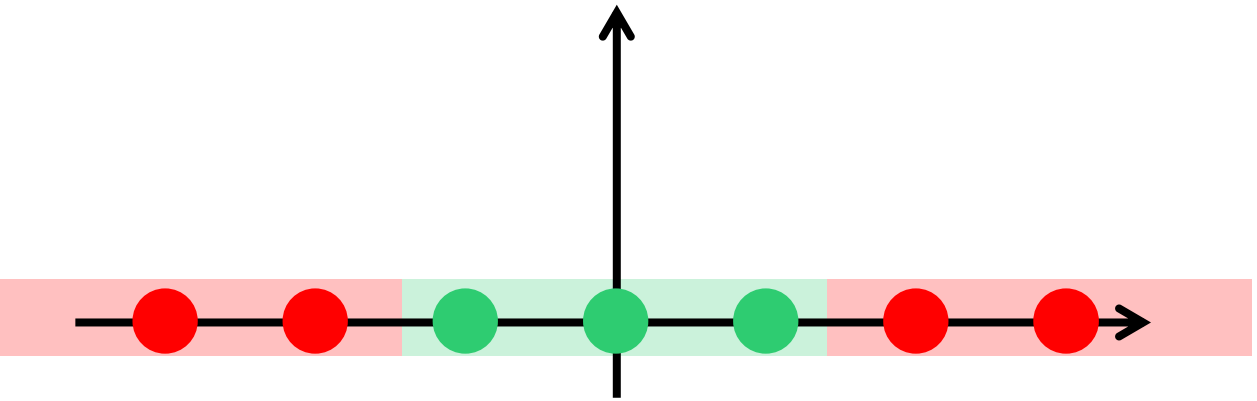
Decision boundary is non-lin in orig space $\mathbb{R}$ i.e lin model in $\mathbb{R}^2$ imposed non-lin boundary in $\mathbb{R}$

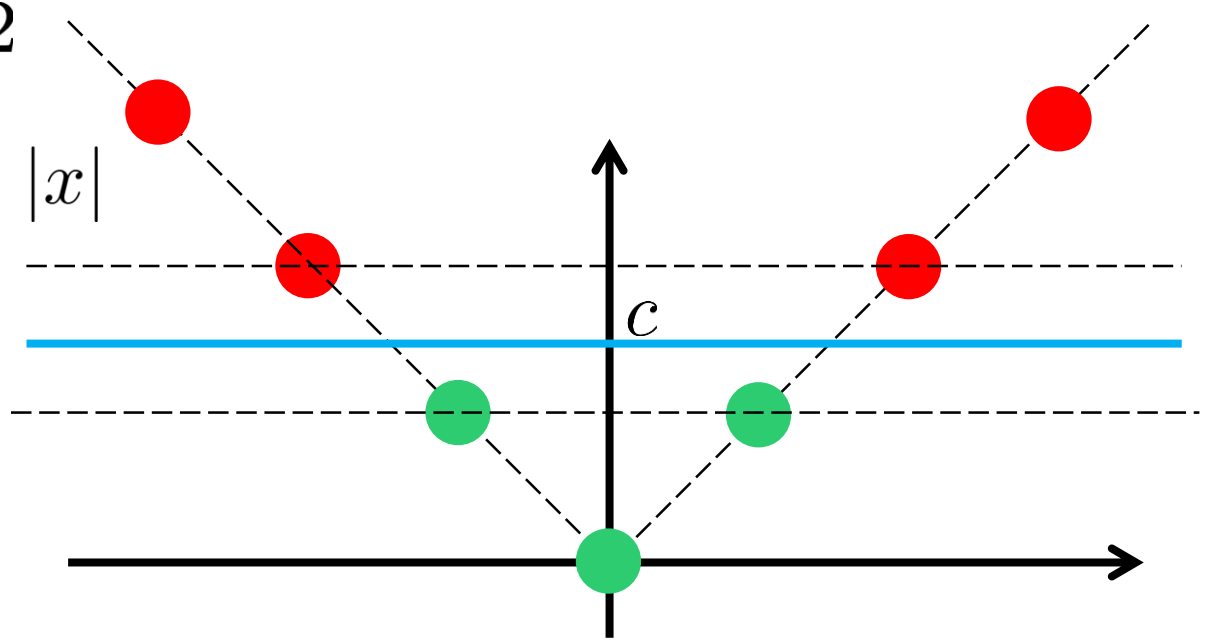
Non-linearity came from map ϕ. There exists no $W \in \mathbb{R}^{2\times1}$ so that $\phi(x) = Wx$ for all $x \in \mathbb{R}$

Non-linear Classification

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

$$\phi(x) = [\phi_1(x), \phi_2(x)], \phi_1(x) = x, \phi_2(x) = |x|$$





Original data non-separable by any linear classifier but using a nice non-linear map makes them separable with a margin!

Linear maps could have never accomplished this. In fact, not all non-linear maps guaranteed to do so either, only nice ones would do this

A good decision boundary is $\phi_2(x) = c$ i.e. $|x| = c$ since $\phi_2(x) = |x|$

Decision boundary is non-lin in orig space $\mathbb{R}$ i.e lin model in $\mathbb{R}^2$ imposed non-lin boundary in $\mathbb{R}$

Non-linearity came from map ϕ. There exists no $W \in \mathbb{R}^{2\times1}$ so that $\phi(x) = Wx$ for all $x \in \mathbb{R}$

Non-linear Classification

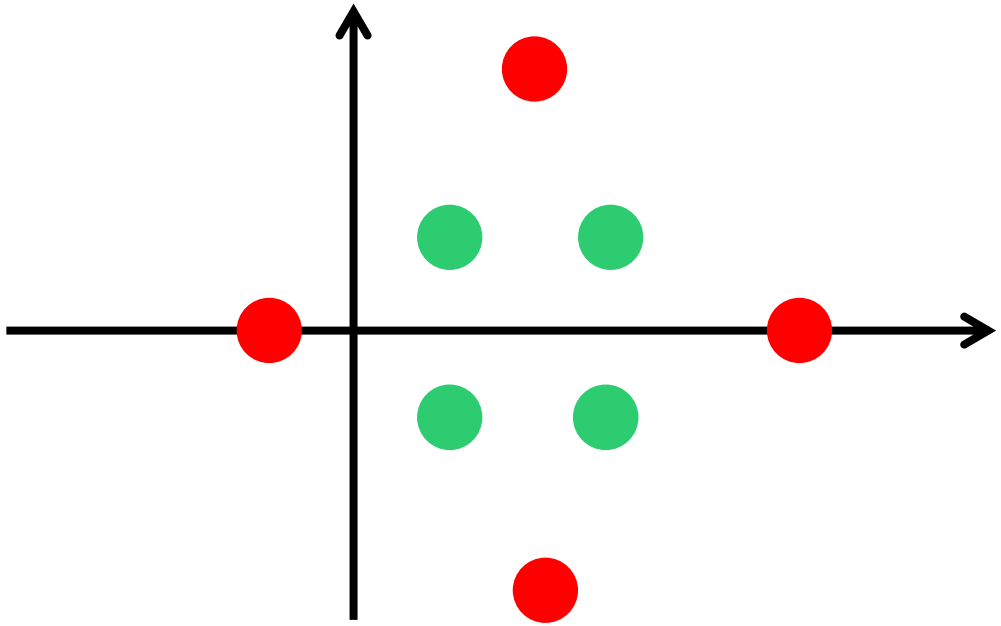
A linear function on the mapped feature vecs looks like

$$\langle \mathbf{w}, \phi(x, y) \rangle = w_1 \cdot x + w_2 \cdot y^2 + w_3 \cdot x^2$$

Consider vector $\mathbf{w} = (-2,1,1)$. Corresponds to the decision boundary

$$-2x + y^2 + x^2 = 0 \equiv (x-1)^2 + y^2 = 1$$

Non-linear Classification



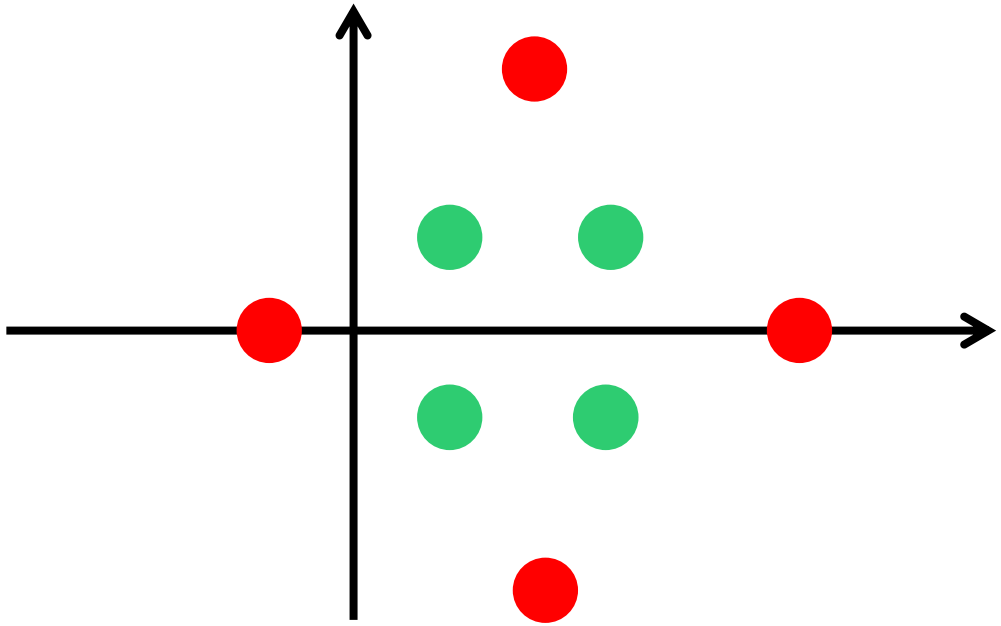
A linear function on the mapped feature vecs looks like

$$\langle \mathbf{w}, \phi(x, y) \rangle = w_1 \cdot x + w_2 \cdot y^2 + w_3 \cdot x^2$$

Consider vector $\mathbf{w} = (-2,1,1)$. Corresponds to the decision boundary

$$-2x + y^2 + x^2 = 0 \equiv (x-1)^2 + y^2 = 1$$

Non-linear Classification



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

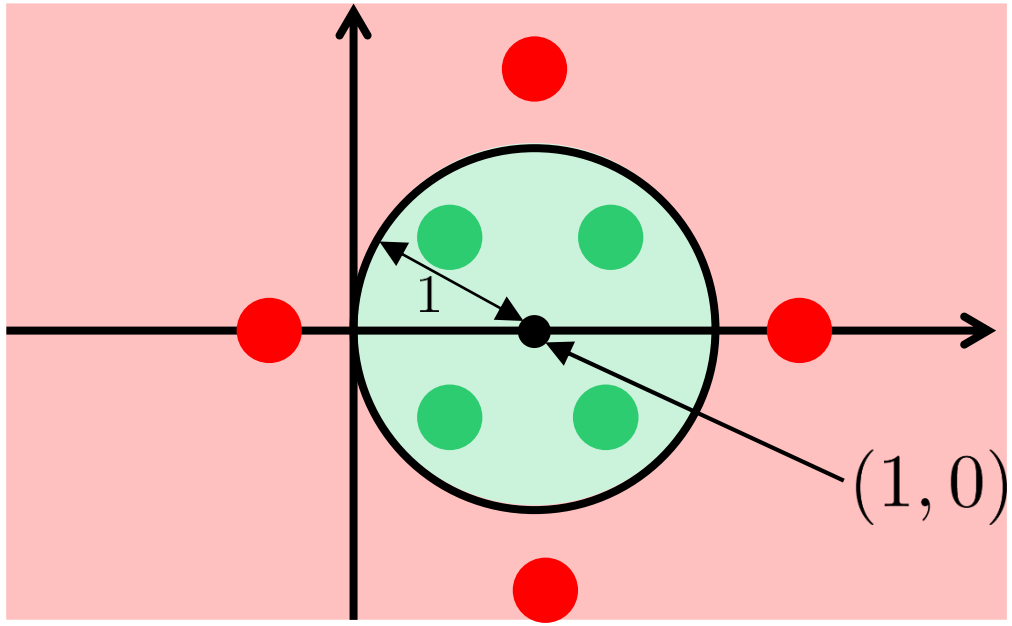
A linear function on the mapped feature vecs looks like

$$\langle \mathbf{w}, \phi(x, y) \rangle = w_1 \cdot x + w_2 \cdot y^2 + w_3 \cdot x^2$$

Consider vector $\mathbf{w} = (-2,1,1)$. Corresponds to the decision boundary

$$-2x + y^2 + x^2 = 0 \equiv (x - 1)^2 + y^2 = 1$$

Non-linear Classification



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

A linear function on the mapped feature vecs looks like

$$\langle \mathbf{w}, \phi(x, y) \rangle = w_1 \cdot x + w_2 \cdot y^2 + w_3 \cdot x^2$$

Consider vector $\mathbf{w} = (-2,1,1)$. Corresponds to the decision boundary

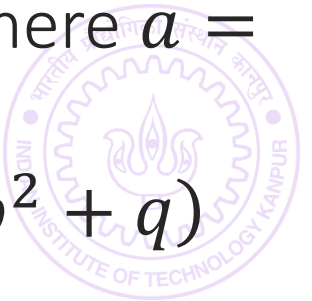
$$-2x + y^2 + x^2 = 0 \equiv (x - 1)^2 + y^2 = 1$$

Non-linear Regression

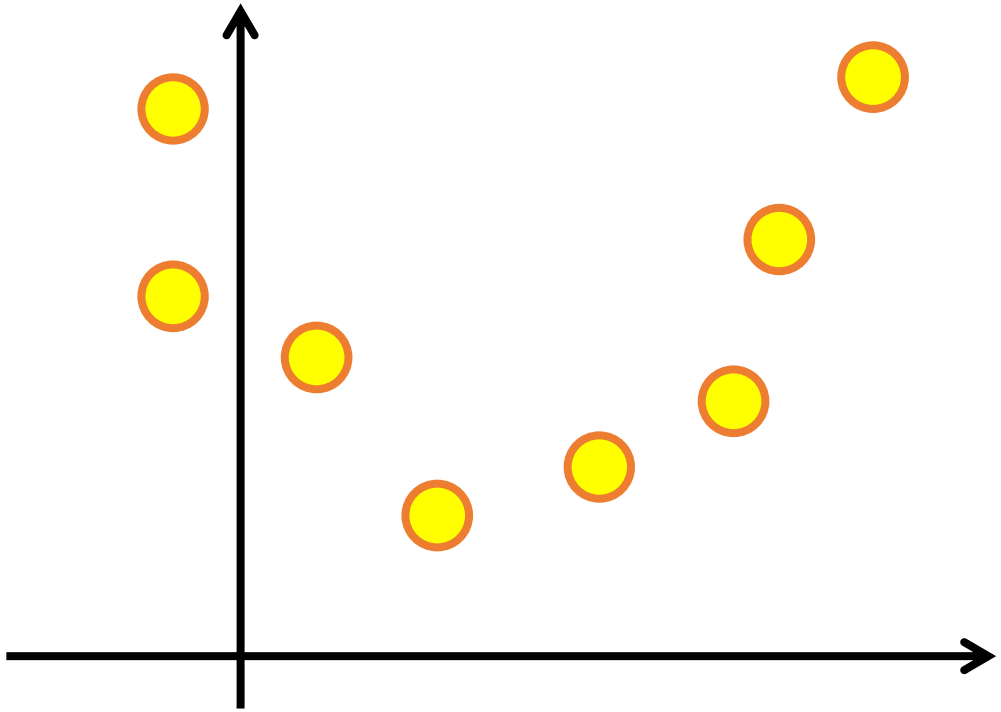
The function $y = (x-2)^2 + 1$ closely fits the above data

A function of the form $(x-p)^2 + q$ can be written as $ax^2 + bx + c$, where $a = 1, b = 2 \cdot p, c = p^2 + q$

Given the map $\phi(x)$, we can easily write this as $\langle \mathbf{w}, \phi(x) \rangle, \mathbf{w} = (1, 2p, p^2 + q)$



Non-linear Regression

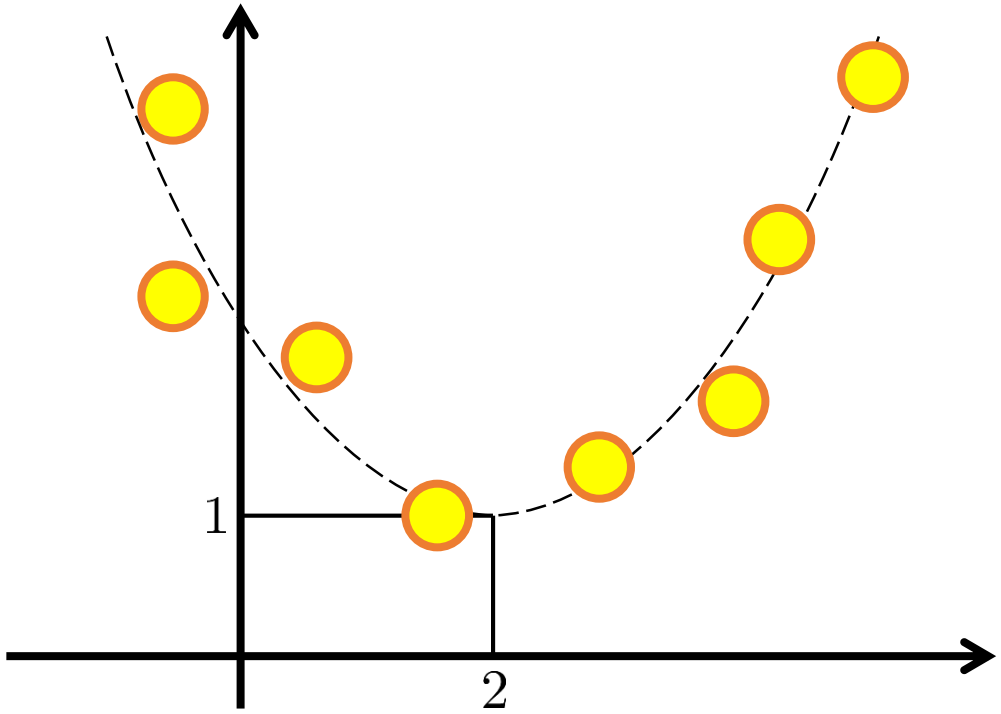


The function $y = (x-2)^2 + 1$ closely fits the above data

A function of the form $(x-p)^2 + q$ can be written as $ax^2 + bx + c$, where $a = 1, b = 2 \cdot p, c = p^2 + q$

Given the map $\phi(x)$, we can easily write this as $\langle \mathbf{w}, \phi(x) \rangle, \mathbf{w} = (1, 2p, p^2 + q)$

Non-linear Regression

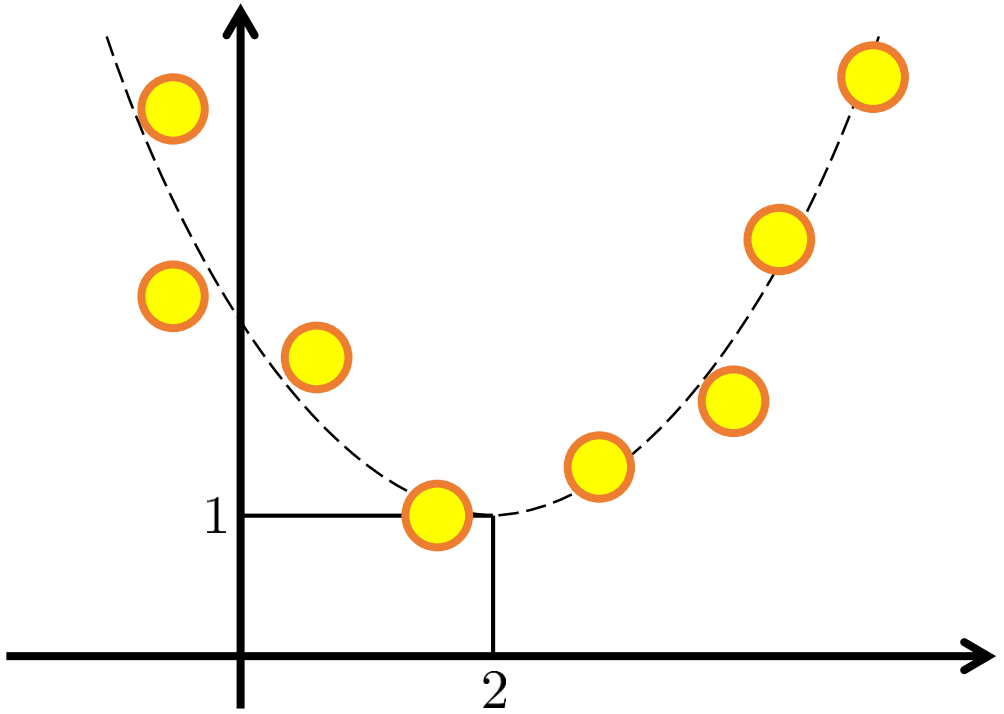


The function $y = (x-2)^2 + 1$ closely fits the above data

A function of the form $(x-p)^2 + q$ can be written as $ax^2 + bx + c$, where $a = 1, b = 2 \cdot p, c = p^2 + q$

Given the map $\phi(x)$, we can easily write this as $\langle \mathbf{w}, \phi(x) \rangle, \mathbf{w} = (1, 2p, p^2 + q)$

Non-linear Regression



$$\mathbb{R} \ni x \mapsto \phi(x) = [1, x, x^2] \in \mathbb{R}^3$$

The function $y = (x-2)^2 + 1$ closely fits the above data

A function of the form $(x-p)^2 + q$ can be written as $ax^2 + bx + c$, where $a = 1, b = 2 \cdot p, c = p^2 + q$

Given the map $\phi(x)$, we can easily write this as $\langle \mathbf{w}, \phi(x) \rangle, \mathbf{w} = (1, 2p, p^2 + q)$

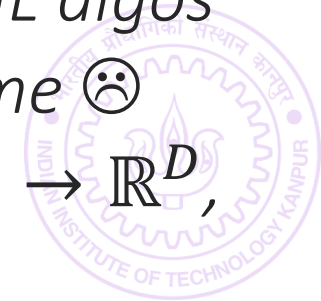
The Kernel Trick

Take original feature vecs (say in d dims) and map them to $D \gg d$ dims

- *Use a non-lin map $\phi: \mathbb{R}^d \rightarrow \mathbb{R}^D$ so that we can hope that vecs in D dims are mostly lin indep (using a linear map is futile – see course notes Example D.4)*
- *Use linear models (SVM, linear regression, k-means, PCA etc) on these new, D dimensional feature vectors – hopefully we will get very good performance*
- *Since ϕ is a non-lin map, our final classifier/regressor $\mathbf{w}^\top \phi(\mathbf{x}), \mathbf{w} \in \mathbb{R}^D$ will look non-lin in the original feat vecs $\mathbf{x}$ even though we used linear ML algos*

Only catch with above scheme is running time

- *Most ML algorithms take time $\Omega(dn)$ so the above scheme may take $\mathcal{O}(dDn)$ time to prepare the new vecs and then $\mathcal{O}(Dn)$ time to execute the ML algos*
- *Since our earlier argument works for $D \geq n$, the above is $\mathcal{O}(dn^2)$ time ☹*
- *"Kernel trick" allows us to, for some very special non-linear maps $\mathbb{R}^d \rightarrow \mathbb{R}^D$, run ML algos on the D-dim vecs without ever computing ϕ explicitly*



The Kernel Trick

Take original feature vecs (say in d dims) and map them to $D \gg d$ dims

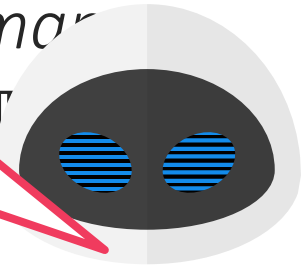
Use a non-lin map $\phi: \mathbb{R}^d \to \mathbb{R}^D$ so that we can hope that vecs in D dims are mostly lin indep (using a linear map is futile – see course notes Example D.4)

Use linear models (SVM, linear regression, k-means, PCA etc) on these new, D dimensional

Since ϕ is a

look non-lin

Indeed, even if ϕ had been a simple linear map, it would have been represented using a $D \times d$ matrix A and simply computing $XA^\top$ for the feature matrix $X \in \mathbb{R}^{n \times d}$ would take $\mathcal{O}(dDn)$ time



Only catch with above scheme is running time

Most ML algorithms take time $\Omega(dn)$ so the above scheme may take $\mathcal{O}(dDn)$ time to prepare the new vecs and then $\mathcal{O}(Dn)$ time to execute the ML algos

Since our earlier argument works for $D \geq n$, the above is $\mathcal{O}(dn^2)$ time ☹

"Kernel trick" allows us to, for some very special non-linear maps $\mathbb{R}^d \to \mathbb{R}^D$, run ML algos on the D-dim vecs without ever computing ϕ explicitly

Kernels vs Distance Measures

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

KERNELS

Give measures of similarity

High value ⇒ Similar points

Example: Gaussian, polynomial

Nice similarity functions satisfy the *Mercer's theorem*

Supports kNN, LwP and can be used to implement SVMs, least squares regression and much more

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

DISTANCE MEASURES

Give measures of dissimilarity

High value ⇒ Different points

Example: Euclidean, Mahalanobis

Nice distance functions satisfy metric or norm properties

Can be used for (multi-label) classfn, regression via kNN or LwP. Can also be used for clustering via k-means

Mercer Kernels

Suppose $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$ are any two unit vectors

The dot product is a natural notion of similarity between these vectors

It is highest when the vectors are the same i.e. $\mathbf{x} = \mathbf{y}$ when we have $\mathbf{x}^\top \mathbf{y} = 1$

It is lowest when the vectors are diametrically opposite i.e. $\mathbf{x} = -\mathbf{y}$ and $\mathbf{x}^\top \mathbf{y} = -1$

Mercer kernels are notions of similarity that extend such nice behaviour

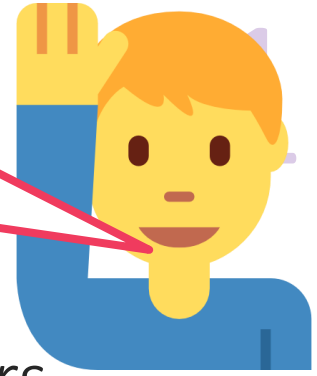
Given a set of objects $\mathcal{X}$ (images, video, strings, genome sequences), a similarity function $K: \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ is called a Mercer kernel if there exists a map $\phi: \mathcal{X} \rightarrow \mathcal{H}$ s.t. for all $x, y \in \mathcal{X}$, $K(x, y) = \langle \phi(x), \phi(y) \rangle$

ϕ often called feature map or feature embedding, $\mathcal{H}$ can be $\mathbb{R}^D$ for some large/moderate D. $\mathcal{H}$ can even be infinite dimensional

Thus, when asked to give similarity between two objects, all that a Mercer kernel does is first map those objects to two (high-dim) vectors and return the dot/inner product between those two vectors

Me

In general, $\mathcal{H}$ has to be a *Hilbert space* which, technicalities aside, is very much like a real vector space such as $\mathbb{R}^D$ but is possibly infinite dimensional. It is always possible to define inner/dot products on Hilbert spaces



Suppose $\mathbf{x}, \mathbf{y} \in \mathbb{R}^D$ are any two unit vectors

The dot product is a natural notion of similarity between these vectors

It is highest when the vectors are the same i.e. $\mathbf{x} = \mathbf{y}$ when we have $\mathbf{x}^\top \mathbf{y} = 1$

It is lowest when the vectors are diametrically opposite i.e. $\mathbf{x} = -\mathbf{y}$ and $\mathbf{x}^\top \mathbf{y} = -1$

Mercer kernels are notions of similarity that extend such nice behaviour

Given a set of objects $\mathcal{X}$ (images, video, strings, genome sequences), a similarity function $K: \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ is called a Mercer kernel if there exists a map $\phi: \mathcal{X} \rightarrow \mathcal{H}$ s.t. for all $x, y \in \mathcal{X}$, $K(x, y) = \langle \phi(x), \phi(y) \rangle$

ϕ often called feature map or feature embedding, $\mathcal{H}$ can be $\mathbb{R}^D$ for some large/moderate D. $\mathcal{H}$ can even be infinite dimensional

Thus, when asked to give similarity between two objects, all that a Mercer kernel does is first map those objects to two (high-dim) vectors and return the dot/inner product between those two vectors

Examples of Kernels

When $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$ are vectors

- Linear kernel $K_{\text{lin}}(\mathbf{x}, \mathbf{y}) = \langle \mathbf{x}, \mathbf{y} \rangle$
- Quadratic kernel $K_{\text{quad}}(\mathbf{x}, \mathbf{y}) = (\langle \mathbf{x}, \mathbf{y} \rangle + 1)^2$
- Polynomial kernel $K_{\text{poly}}(\mathbf{x}, \mathbf{y}) = (\langle \mathbf{x}, \mathbf{y} \rangle + c)^p, c \geq 0, p \in \mathbb{N}$
- Gaussian kernel $K_{\text{gauss}}(\mathbf{x}, \mathbf{y}) = \exp(-\gamma \cdot \|\mathbf{x} - \mathbf{y}\|_2^2)$
- Laplacian kernel $K_{\text{lap}}(\mathbf{x}, \mathbf{y}) = \exp(-\gamma \cdot \|\mathbf{x} - \mathbf{y}\|_1)$

All of the above are Mercer kernels

There indeed exist feature maps ϕ for each of them (proving so a bit tedious)

p, γ need to be tuned. Large p, γ can cause overfitting

Notice all are the above are indeed notions of similarity

Take two unit vectors $\|\mathbf{x}\|_2 = 1 = \|\mathbf{y}\|_2$ (unit for sake of normalization). Easy to verify that $K(\mathbf{x}, \mathbf{y})$ is largest when $\mathbf{x} = \mathbf{y}$ and smallest when $\mathbf{x} = -\mathbf{y}$

Mercer Kernel Feature Maps

Linear kernel $K_{\text{lin}}(\mathbf{x}, \mathbf{y}) = \langle \mathbf{x}, \mathbf{y} \rangle$

Use $\phi(\mathbf{x}) = \mathbf{x}$. Called "linear" for a reason: any linear function over $\phi(\mathbf{x})$ is just a linear function over the original features $\mathbf{x}$

Quadratic kernel $K_{\text{quad}}(\mathbf{x}, \mathbf{y}) = (\langle \mathbf{x}, \mathbf{y} \rangle + 1)^2$

We have $(\langle \mathbf{x}, \mathbf{y} \rangle + 1)^2 = 1 + 2 \cdot \langle \mathbf{x}, \mathbf{y} \rangle + \sum_{i,j}^{d} \mathbf{x}_i \mathbf{x}_j \mathbf{y}_i \mathbf{y}_j = \langle \phi(\mathbf{x}), \phi(\mathbf{y}) \rangle$, when $\phi(\mathbf{x}) = [\phi_0(\mathbf{x}), \phi_1(\mathbf{x}), \phi_2(\mathbf{x})] \in \mathbb{R}^{d^2+d+1}$ where $\phi_0(\mathbf{x}) = [1] \in \mathbb{R}^1, \phi_1(\mathbf{x}) = \sqrt{2} \cdot \mathbf{x} \in \mathbb{R}^d, \phi_2(\mathbf{x}) = [\mathbf{x}_1\mathbf{x}_1, \mathbf{x}_1\mathbf{x}_2, \mathbf{x}_1\mathbf{x}_3, \dots, \mathbf{x}_1\mathbf{x}_d, \mathbf{x}_2\mathbf{x}_1, \dots, \mathbf{x}_d\mathbf{x}_d] \in \mathbb{R}^{d^2}$

Similar constructions (more tedious to write) for polynomial kernel

Called "quadratic" for a reason: any linear function over $\phi(\mathbf{x})$ is a quadratic function over the original features $\mathbf{x}$. Verify for the simple case $d = 1$ yourself

If we use a linear ML algo over $\phi(\mathbf{x}) \in \mathbb{R}^{d^2+d+1}$, can learn any quadratic function over original features $\mathbf{x} \in \mathbb{R}^d$. Polynomial kernel of degree p similarly allows learning of degree p polynomial functions over original data

Mercer Kernel Feature Maps

Warning: may exist more than one map ϕ for the same kernel K

Example: *we used* $\phi(\mathbf{x}) = [\phi_0(\mathbf{x}), \phi_1(\mathbf{x}), \phi_2(\mathbf{x})]$ *for quadratic. However* $\tilde{\phi}(\mathbf{x}) = \left[\phi_0(\mathbf{x}), \tilde{\phi}_{11}(\mathbf{x}), \tilde{\phi}_{12}(\mathbf{x}), \phi_2(\mathbf{x})\right]$ *with* $\tilde{\phi}_{11}(\mathbf{x}) = \mathbf{x} = \tilde{\phi}_{12}$ *gives same* K

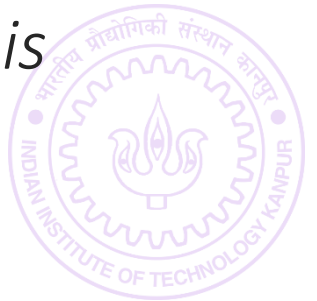
Gaussian/Laplacian Kernels correspond to infinite dimensional maps

$$K_{\text{gauss}}(\mathbf{x}, \mathbf{y}) = \exp(-\gamma \cdot \|\mathbf{x} - \mathbf{y}\|_2^2)$$
$$= \exp(-\gamma \cdot \|\mathbf{x}\|_2^2) \exp(-\gamma \cdot \|\mathbf{y}\|_2^2) \exp(2\gamma \cdot \langle \mathbf{x}, \mathbf{y} \rangle)$$
$$= \exp(-\gamma \cdot \|\mathbf{x}\|_2^2) \exp(-\gamma \cdot \|\mathbf{y}\|_2^2) \sum_{i=0}^{\infty} \frac{(2\gamma)^i}{i!} \cdot \langle \mathbf{x}, \mathbf{y} \rangle^i$$

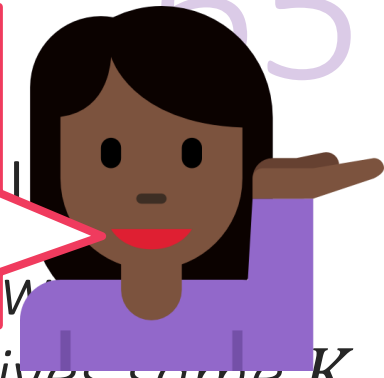
Gaussian kernel is an infinite linear combination of poly kernels of all orders

Let ϕ_i^{poly} *be a map for the poly kernel* $\langle \mathbf{x}, \mathbf{y} \rangle^i$. *Then a map for* K_{gauss} *is*

$$\phi_{\text{gauss}}(\mathbf{x}) = \exp(-\gamma \cdot \|\mathbf{x}\|_2^2) \cdot \left[\frac{\sqrt{(2\gamma)^i}}{\sqrt{i!}} \cdot \phi_i^{\text{poly}}(\mathbf{x}) \right]_{i=1,2,\ldots,\infty}$$



Learning a linear function over the features $\phi_{\text{gauss}}(\mathbf{x})$ amounts to learning an infinite-degree polynomial over $\mathbf{x}$. Gauss/Lap are very powerful kernels, often called *universal kernels*. By using these kernels, theoretically speaking, one can learn *any* function over data (details beyond scope of CS771)



$\tilde{\phi}(\mathbf{x}) = [\phi_0(\mathbf{x}), \tilde{\phi}_{11}(\mathbf{x}), \tilde{\phi}_{12}(\mathbf{x}), \phi_2(\mathbf{x})]$ *with* $\tilde{\phi}_{11}(\mathbf{x}) = \mathbf{x} = \tilde{\phi}_{12}$ *gives same* K

Gaussian/Laplacian Kernels correspond to infinite dimensional maps

$$K_{\text{gauss}}(\mathbf{x}, \mathbf{y}) = \exp(-\gamma \cdot \|\mathbf{x} - \mathbf{y}\|_2^2)$$
$$= \exp(-\gamma \cdot \|\mathbf{x}\|_2^2) \exp(-\gamma \cdot \|\mathbf{y}\|_2^2) \exp(2\gamma \cdot \langle \mathbf{x}, \mathbf{y} \rangle)$$
$$= \exp(-\gamma \cdot \|\mathbf{x}\|_2^2) \exp(-\gamma \cdot \|\mathbf{y}\|_2^2) \sum_{i=0}^{\infty} \frac{(2\gamma)^i}{i!} \cdot \langle \mathbf{x}, \mathbf{y} \rangle^i$$

Gaussian kernel is an infinite linear combination of poly kernels of all orders

Let ϕ_i^{poly} *be a map for the poly kernel* $\langle \mathbf{x}, \mathbf{y} \rangle^i$. *Then a map for* K_{gauss} *is*

$$\phi_{\text{gauss}}(\mathbf{x}) = \exp(-\gamma \cdot \|\mathbf{x}\|_2^2) \cdot \left[\frac{\sqrt{(2\gamma)^i}}{\sqrt{i!}} \cdot \phi_i^{\text{poly}}(\mathbf{x}) \right]_{i=1,2,\ldots,\infty}$$

Some Domain Specific Kernels

Over the years people have designed innovative and powerful Mercer kernels specifically for NLP, vision and other domains

When $\mathbf{x}, \mathbf{y}$ are bag of words features for strings/documents

- Let dictionary $\Sigma = \{w_1, w_2, \ldots, w_d\}$ have d words in it
- Let $c_i(x)$ be the count of word i in string $\mathbf{x}$
- Intersection kernel $K_{\mathbf{int}}(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{d} \mathbf{min}\{c_i(\mathbf{x}), c_i(\mathbf{y})\}$ (Mercer kernel)
- Normalize intersection kernel $K_{\mathbf{int-n}}(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{d} \frac{\mathbf{min}\{c_i(\mathbf{x}), c_i(\mathbf{y})\}}{\sqrt{c_i(\mathbf{x}) \cdot c_i(\mathbf{y})}}$ (define $\frac{0}{0} = 0$)

More generally, when $X, Y \subseteq \mathcal{U}$ are sets

- Intersection kernel $K_{\mathbf{int}}(X, Y) = |X \cap Y|$
- Norm. Int. kernel $K_{\mathbf{int-n}}(X, Y) = \frac{|X \cap Y|}{\sqrt{|X| \cdot |Y|}}$ (notice that $K_{\mathbf{int-n}} \in (0,1)$)
- The above are just the linear kernel in disguise and hence clearly Mercer

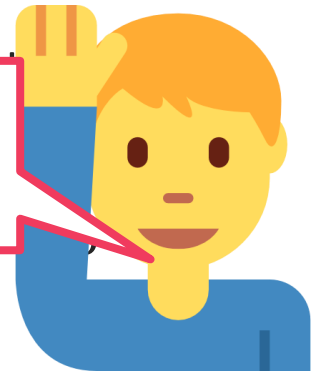
Some Domain Specific Kernels

Over the years people have designed innovative and powerful Mercer kernels specifically for NLP, vision and other domains

When $\mathbf{x}, \mathbf{y}$ are bag of words features for strings/documents

- Let dictionary $\Sigma = \{w_1, w_2, \ldots, w_d\}$ have d words in it
- Let $c_i(x)$ be the count of word i in string $\mathbf{x}$
- Intersecti
- Normalize

Simply represent a set X using an *indicator vector* $\mathbb{I}_X \in \{0,1\}^{|\mathcal{U}|}$ with $\mathbb{I}_X[i] = 1$ if $i \in X$ else $\mathbb{I}_X[i] = 0$. In this case, $|X \cap Y| = \langle \mathbb{I}_X, \mathbb{I}_Y \rangle$



More generally, when $X, Y \subseteq \mathcal{U}$ are sets

- Intersection kernel $K_{\text{int}}(X, Y) = |X \cap Y|$
- Norm. Int. kernel $K_{\text{int-n}}(X, Y) = \frac{|X \cap Y|}{\sqrt{|X| \cdot |Y|}}$ (notice that $K_{\text{int-n}} \in (0,1)$)
- The above are just the linear kernel in disguise and hence clearly Mercer

Some Domain Specific Kernels

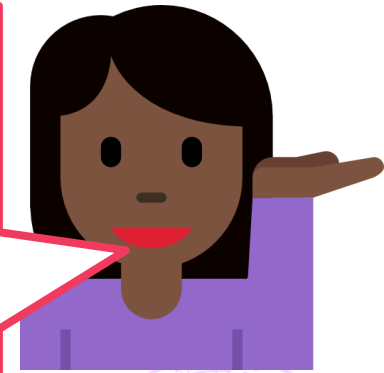
N-gram, substring, Fisher kernels: other kernels between two strings

Random walk kernels between two graphs

Subtree, convolutional kernels between two trees

Pyramid kernel used in vision … combination of intersection kernels

In practice, we often use a linear method first e.g. SVM/ridge regression. If that gives unsatisfactory performance, often we jump directly to Gaussian kernel ☺ although we should not neglect polynomial/other domain specific kernels. There exist "kernel learning" methods that can learn the most appropriate kernel for us or else tune the kernel parameters e.g. p, c, γ for us automatically



Creating New Kernels

Method 1: combine old kernels. If K_1, K_2 are existing Mercer kernels

$K_3 = c_1 \cdot K_1 + c_2 \cdot K_2$ is also a Mercer kernel if $c_1, c_2 \geq 0$

$$K_3(\mathbf{x}, \mathbf{y}) = c_1 \cdot K_1(\mathbf{x}, \mathbf{y}) + c_2 \cdot K_2(\mathbf{x}, \mathbf{y})$$

$K_4 = K_1 \cdot K_2$ is also a nice kernel

$$K_4(\mathbf{x}, \mathbf{y}) = K_1(\mathbf{x}, \mathbf{y}) \cdot K_2(\mathbf{x}, \mathbf{y})$$

$K_5(\mathbf{x}, \mathbf{y}) = K_1(\mathbf{x}, \mathbf{y}) / \sqrt{K_1(\mathbf{x}, \mathbf{x}) \cdot K_1(\mathbf{y}, \mathbf{y})}$ gives a normalized kernel

Method 2: find a new feature rep. for data $\phi_{\mathbf{new}}(x) \in \mathbb{R}^d$ and use

$$K_{\mathbf{new}}(x, y) = \langle \phi_{\mathbf{new}}(x), \phi_{\mathbf{new}}(y) \rangle$$

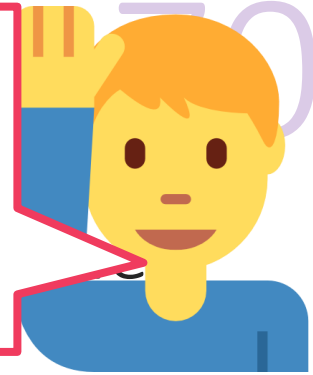
Method 3: mix and match. Take new data rep. $\phi_{\mathbf{new}}(x) \in \mathbb{R}^d$ and an old kernel $K_{\mathbf{old}}$

$$K_{\mathbf{newer}}(x, y) = K_{\mathbf{old}}\big(\phi_{\mathbf{new}}(x), \phi_{\mathbf{new}}(y)\big)$$

Creati

The normalized kernel actually normalizes the feature map as well. Verify that if ϕ is a map for K_1 then a map for K_5 is $\tilde{\phi}$ where

$$\tilde{\phi}: \mathbf{x} \mapsto \frac{\phi(\mathbf{x})}{\|\phi(\mathbf{x})\|_{\mathcal{H}}}$$

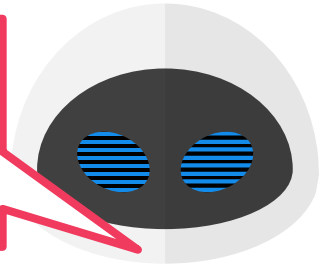


Method 1

$K_3 = c_1 \cdot K_1 + c_2 \cdot K_2$ is also a Mercer kernel if $c_1, c_2 \geq 0$

$K_4 =$

If K_1 gives very large values (in magnitude), some algorithms may suffer. The normalized version K_5 will always give values between $[-1,1]$



$K_5(\mathbf{x}, \mathbf{y}) = K_1(\mathbf{x}, \mathbf{y}) / \sqrt{K_1(\mathbf{x}, \mathbf{x}) \cdot K_1(\mathbf{y}, \mathbf{y})}$ gives a normalized kernel

Method 2: find a new feature rep. for data $\phi_{\text{new}}(x) \in \mathbb{R}^d$ and use

$$K_{\text{new}}(x, y) = \langle \phi_{\text{new}}(x), \phi_{\text{new}}(y) \rangle$$

Method 3: mix and match. Take new data rep. $\phi_{\text{new}}(x) \in \mathbb{R}^d$ and an old kernel K_{old}

$$K_{\text{newer}}(x, y) = K_{\text{old}}\big(\phi_{\text{new}}(x), \phi_{\text{new}}(y)\big)$$

kNN with Kernels

All that is needed to execute kNN is compute Euclidean distances

If working with kernel K with map ϕ, need $\|\phi(\mathbf{x}) - \phi(\mathbf{y})\|_{\mathcal{H}}^2$

$\|\cdot\|_{\mathcal{H}}$ is just fancy notation for $\|\cdot\|_2$ in a Hilbert space $\mathcal{H}$

$$\|\phi(\mathbf{x}) - \phi(\mathbf{y})\|_{\mathcal{H}}^2 = \|\phi(\mathbf{x})\|_2^2 - 2\langle\phi(\mathbf{x}), \phi(\mathbf{y})\rangle + \|\phi(\mathbf{y})\|_2^2$$
$$= \langle\phi(\mathbf{x}), \phi(\mathbf{x})\rangle - 2\langle\phi(\mathbf{x}), \phi(\mathbf{y})\rangle + \langle\phi(\mathbf{y}), \phi(\mathbf{y})\rangle$$
$$= K(\mathbf{x}, \mathbf{x}) - 2K(\mathbf{x}, \mathbf{y}) + K(\mathbf{y}, \mathbf{y})$$

Thus, distances in $\mathcal{H}$ can be computed without computing ϕ first ☺

1NN: Given training points $\left(\mathbf{x}^i, y^i\right), i = 1, \ldots n$ and a test point $\mathbf{x}^t$

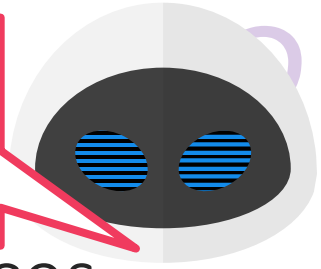
Find closest neighbor in $\mathcal{H}$ i.e. $i^t = \arg\min_{i \in [n]} \left\|\phi(\mathbf{x}^t) - \phi\left(\mathbf{x}^i\right)\right\|_{\mathcal{H}}^2$ which is the same as $\arg\min_{i \in [n]} K\left(\mathbf{x}^i, \mathbf{x}^i\right) - 2K\left(\mathbf{x}^i, \mathbf{x}^t\right)$ and predict y^{i^t} as the label

***Note**: if $K(\mathbf{x}, \mathbf{x}) \equiv 1$ then this finds most "similar" point i.e. $\operatorname{argmax}_{i \in [n]} K\left(\mathbf{x}^i, \mathbf{x}^t\right)$*

Similarly we can execute kNN for $k > 1$ as well

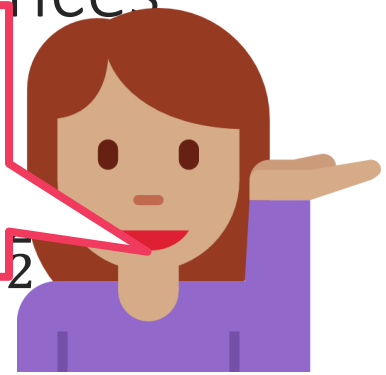
kNN

This is a recurring theme in kernel learning. **Never ever compute $\phi(\cdot)$.** Instead, express all operations in the ML algo in terms of inner product computations which are then expressible as kernel computations



All that is needed to execute kNN is compute Euclidean distances

Indeed, computing $K(\mathbf{x}, \mathbf{y})$ usually takes $\mathcal{O}(d)$ time if $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$ but computing $\phi(\mathbf{x})$ may take much longer e.g. for Gaussian kernel it will take ∞ time



If working with kerne

$\|\cdot\|_{\mathcal{H}}$ *is just*

$$\|\phi(\mathbf{x}) - \phi(\mathbf{y}$$

$$= \langle\phi(\mathbf{x}), \phi(\mathbf{x})\rangle - 2\langle\phi(\mathbf{x}), \phi(\mathbf{y})\rangle + \langle\phi(\mathbf{y}), \phi(\mathbf{y})\rangle$$

$$= K(\mathbf{x}, \mathbf{x}) - 2K(\mathbf{x}, \mathbf{y}) + K(\mathbf{y}, \mathbf{y})$$

Thus, distances in $\mathcal{H}$ can be computed without computing ϕ first ☺

1NN: Given training points $(\mathbf{x}^i, y^i), i = 1, \ldots n$ and a test point $\mathbf{x}^t$

Find closest neighbor in $\mathcal{H}$ i.e. $i^t = \arg\min_{i \in [n]} \left\|\phi(\mathbf{x}^t) - \phi(\mathbf{x}^i)\right\|_{\mathcal{H}}^2$ which is the same as $\arg\min_{i \in [n]} K(\mathbf{x}^i, \mathbf{x}^i) - 2K(\mathbf{x}^i, \mathbf{x}^t)$ and predict y^{i^t} as the label

***Note**: if $K(\mathbf{x}, \mathbf{x}) \equiv 1$ then this finds most "similar" point i.e. $\operatorname{argmax}_{i \in [n]} K(\mathbf{x}^i, \mathbf{x}^t)$*

Similarly we can execute kNN for $k > 1$ as well

LwP with Kernels

Given train data $(\mathbf{x}^i, y^i), y^i \in \{-1,1\}$, we earlier found prototypes $\mathbf{p}^+ = \frac{1}{n_+} \cdot \sum_{y^i=1} \mathbf{x}^i$ and $\mathbf{p}^- = \frac{1}{n_-} \cdot \sum_{y^i=-1} \mathbf{x}^i$ and used them to predict on test point $\mathbf{x}^t$ as $\mathbf{sign}(\|\mathbf{x}^t - \mathbf{p}^-\|_2^2 - \|\mathbf{x}^t - \mathbf{p}^+\|_2^2)$

If using a kernel K with map ϕ, we should now compute new prototypes as $\tilde{\mathbf{p}}^+ = \frac{1}{n_+} \cdot \sum_{y^i=1} \phi(\mathbf{x}^i)$ *and* $\tilde{\mathbf{p}}^- = \frac{1}{n_-} \cdot \sum_{y^i=-1} \phi(\mathbf{x}^i)$ *and predict using*

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

Need to be careful now – cannot compute these new prototypes explicitly

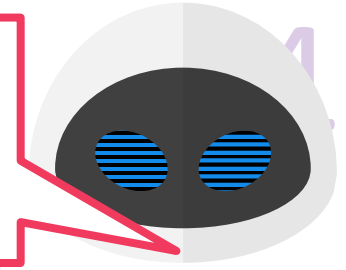
Instead, as before, we reduce the above to kernel computations instead

$$\|\phi(\mathbf{x}^t) - \tilde{\mathbf{p}}^+\|_2^2 = \langle \phi(\mathbf{x}^t), \phi(\mathbf{x}^t) \rangle - 2\langle \phi(\mathbf{x}^t), \tilde{\mathbf{p}}^+ \rangle + \langle \tilde{\mathbf{p}}^+, \tilde{\mathbf{p}}^+ \rangle$$

The first term is simply $K(\mathbf{x}^t, \mathbf{x}^t)$*, second term is* $\frac{2}{n_+} \cdot \sum_{y^i=1} K(\mathbf{x}^t, \mathbf{x}^i)$*, third term is* $\frac{1}{n_+^2} \cdot \sum_{y^i=1} \sum_{y^j=1} K(\mathbf{x}^i, \mathbf{x}^j)$ *which can be pre-calculated at train time*

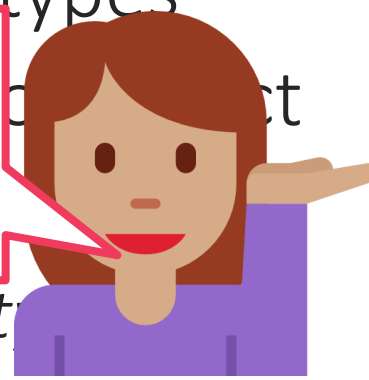
LwP

Observe that in LwP with kernels, we now have to store entire training data whereas earlier we just had to store two prototypes. This is common in kernel learning – larger model sizes and longer prediction times



Given train data $(\mathbf{x}^i, y^i), y^i \in \{-1, +1\}$, we earlier found prototypes $\mathbf{p}^+ = \frac{1}{n_+} \cdot \sum_{y^i=1} \mathbf{x}^i$ and $\mathbf{p}^- =$ [illegible] ct on test point $\mathbf{x}^t$ as $\mathbf{sign}(\|\mathbf{x}^t -$ [illegible]

There are ways in which kernel methods can be sped up and model sizes reduced. Will see those techniques later



If using a kernel K with map ϕ, we should now compute new prototypes $\tilde{\mathbf{p}}^+ = \frac{1}{n_+} \cdot \sum_{y^i=1} \phi(\mathbf{x}^i)$ *and* $\tilde{\mathbf{p}}^- = \frac{1}{n_-} \cdot \sum_{y^i=-1} \phi(\mathbf{x}^i)$ *and predict using*

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

Need to be careful now – cannot compute these new prototypes explicitly

Instead, as before, we reduce the above to kernel computations instead

$$\|\phi(\mathbf{x}^t) - \tilde{\mathbf{p}}^+\|_2^2 = \langle \phi(\mathbf{x}^t), \phi(\mathbf{x}^t) \rangle - 2\langle \phi(\mathbf{x}^t), \tilde{\mathbf{p}}^+ \rangle + \langle \tilde{\mathbf{p}}^+, \tilde{\mathbf{p}}^+ \rangle$$

The first term is simply $K(\mathbf{x}^t, \mathbf{x}^t)$*, second term is* $\frac{2}{n_+} \cdot \sum_{y^i=1} K(\mathbf{x}^t, \mathbf{x}^i)$*, third term is* $\frac{1}{n_+^2} \cdot \sum_{y^i=1} \sum_{y^j=1} K(\mathbf{x}^i, \mathbf{x}^j)$ *which can be pre-calculated at train time*