Making Linear Models Nonlinear via Kernel Methods

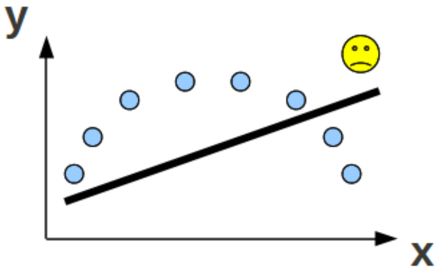
Piyush Rai

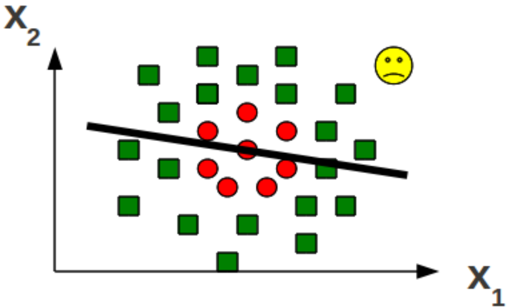
Introduction to Machine Learning (CS771A)

September 6, 2018

Linear Models

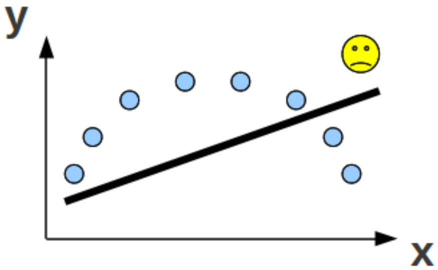
- Nice and interpretable but can't learn "difficult" nonlinear patterns

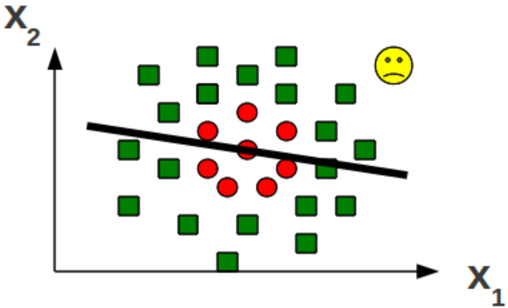




Linear Models

- Nice and interpretable but can't learn "difficult" nonlinear patterns





- So, are linear models useless for such problems?

Linear Models for Nonlinear Problems!

- Consider the following one-dimensional inputs from two classes

Linear Models for Nonlinear Problems!

- Consider the following one-dimensional inputs from two classes

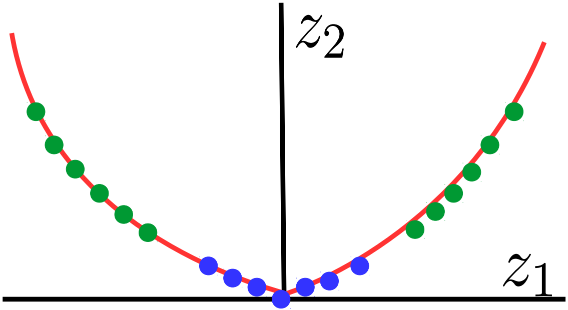
- Can't separate using a linear hyperplane

Linear Models for Nonlinear Problems!

- Consider mapping each x to two-dimensions as $x \to \boldsymbol{z} = [z_1, z_2] = [x, x^2]$

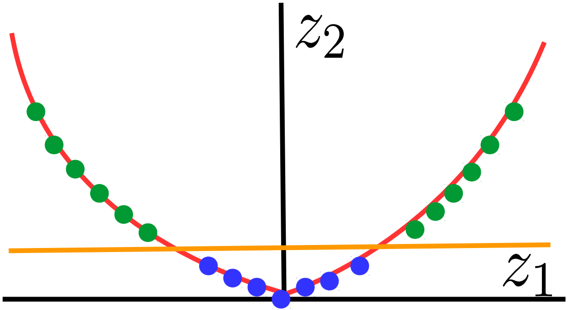
Linear Models for Nonlinear Problems!

- Consider mapping each x to two-dimensions as $x \rightarrow \boldsymbol{z} = [z_1, z_2] = [x, x^2]$



Linear Models for Nonlinear Problems!

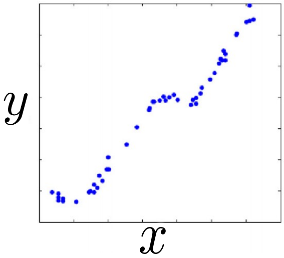
- Consider mapping each x to two-dimensions as $x \rightarrow \boldsymbol{z} = [z_1, z_2] = [x, x^2]$



- Data now becomes linearly separable in the two-dimensional space

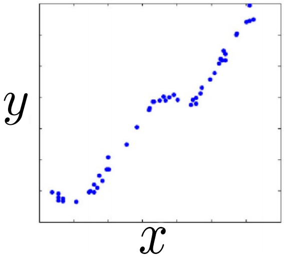
Linear Models for Nonlinear Problems!

- Consider this regression problem with one-dimensional inputs



Linear Models for Nonlinear Problems!

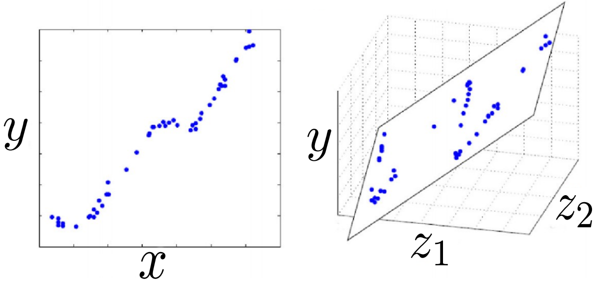
- Consider this regression problem with one-dimensional inputs



- Linear regression won't work well

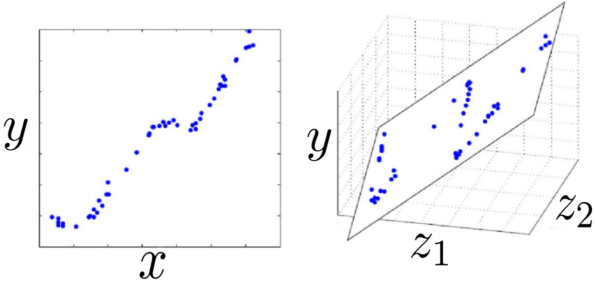
Linear Models for Nonlinear Problems!

- Consider mapping each x to two-dimensions as $x \rightarrow \boldsymbol{z} = [z_1, z_2] = [x, \cos(x)]$



Linear Models for Nonlinear Problems!

- Consider mapping each x to two-dimensions as $x \to \boldsymbol{z} = [z_1, z_2] = [x, \cos(x)]$



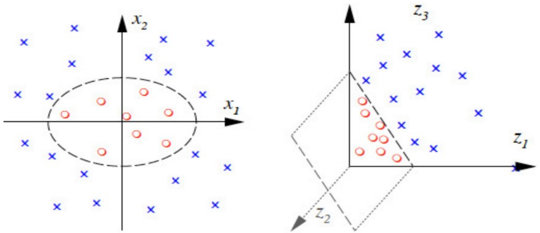
- Now we can fit a linear regression model in two-dimensional input space

Linear Models for Nonlinear Problems!

- Essentially, can use some function ϕ to map/transform inputs to a "nice" space

$$\phi : \mathbb{R}^2 \rightarrow \mathbb{R}^3$$

$$(x_1, x_2) \mapsto (z_1, z_2, z_3) := (x_1^2, \sqrt(2)x_1x_2, x_2^2)$$

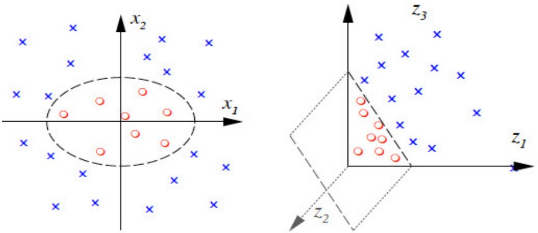


Linear Models for Nonlinear Problems!

- Essentially, can use some function ϕ to map/transform inputs to a "nice" space

$$\phi : \mathbb{R}^2 \rightarrow \mathbb{R}^3$$

$$(x_1, x_2) \mapsto (z_1, z_2, z_3) := (x_1^2, \sqrt(2)x_1x_2, x_2^2)$$



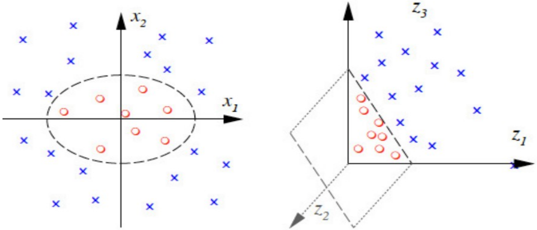
- .. and then happily apply a linear model in the new space!

Linear Models for Nonlinear Problems!

- Essentially, can use some function ϕ to map/transform inputs to a "nice" space

$$\phi : \mathbb{R}^2 \to \mathbb{R}^3$$

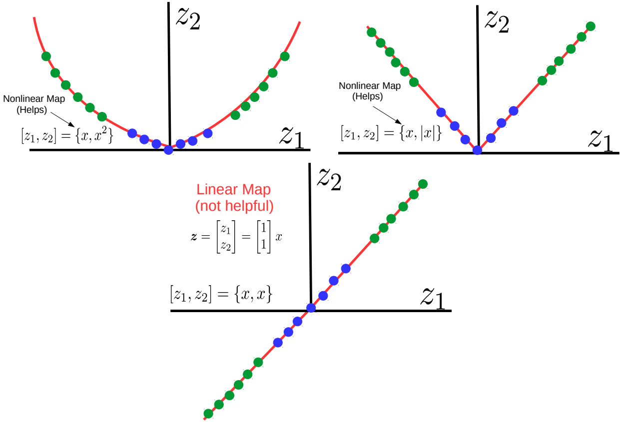
$$(x_1, x_2) \mapsto (z_1, z_2, z_3) := (x_1^2, \sqrt(2)x_1x_2, x_2^2)$$



- .. and then happily apply a linear model in the new space!
- Linear in the new space but nonlinear in the original space!

Not Every Mapping is Helpful

- Not every mapping helps in learning nonlinear patterns. Must at least be nonlinear!
- For the nonlinear classification problem we saw earlier, consider some possible mappings



How to get these “good” (nonlinear) mappings?

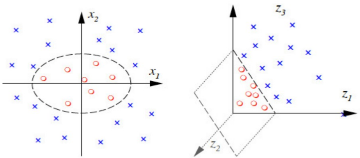
- Can try to learn the mapping from the data itself (e.g., using deep learning - later)

How to get these “good” (nonlinear) mappings?

- Can try to learn the mapping from the data itself (e.g., using deep learning - later)
- There are also pre-defined “good” mappings (e.g., provided by kernel functions - today’s topic)

$$\phi : \mathbb{R}^2 \to \mathbb{R}^3$$

$$(x_1, x_2) \mapsto (z_1, z_2, z_3) := (x_1^2, \sqrt(2)x_1x_2, x_2^2)$$

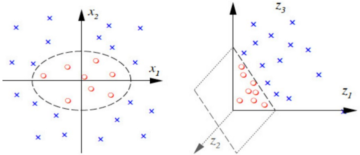


How to get these "good" (nonlinear) mappings?

- Can try to learn the mapping from the data itself (e.g., using deep learning - later)
- There are also pre-defined "good" mappings (e.g., provided by kernel functions - today's topic)

$$\phi : \mathbb{R}^2 \to \mathbb{R}^3$$

$$(x_1, x_2) \mapsto (z_1, z_2, z_3) := (x_1^2, \sqrt{(2)}x_1x_2, x_2^2)$$



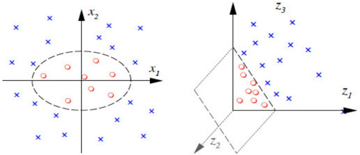
- Looks like I have to compute these mapping using ϕ. That would be quite expensive!

How to get these "good" (nonlinear) mappings?

- Can try to learn the mapping from the data itself (e.g., using deep learning - later)
- There are also pre-defined "good" mappings (e.g., provided by kernel functions - today's topic)

$$\phi : \mathbb{R}^2 \rightarrow \mathbb{R}^3$$

$$(x_1, x_2) \mapsto (z_1, z_2, z_3) := (x_1^2, \sqrt(2)x_1x_2, x_2^2)$$



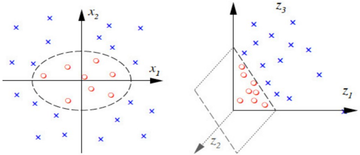
- Looks like I have to compute these mapping using ϕ. That would be quite expensive!
- Thankfully, not always. For example, when using kernels, you get these for (almost) free

How to get these "good" (nonlinear) mappings?

- Can try to learn the mapping from the data itself (e.g., using deep learning - later)
- There are also pre-defined "good" mappings (e.g., provided by kernel functions - today's topic)

$$\phi : \mathbb{R}^2 \to \mathbb{R}^3$$

$$(x_1, x_2) \mapsto (z_1, z_2, z_3) := (x_1^2, \sqrt(2)x_1x_2, x_2^2)$$



- Looks like I have to compute these mapping using ϕ. That would be quite expensive!
- Thankfully, not always. For example, when using kernels, you get these for (almost) free
  - A kernel defines an "implicit" mapping for the data

Kernels as (Implicit) Feature Maps

- Consider two data points $\boldsymbol{x} = \{x_1, x_2\}$ and $\boldsymbol{z} = \{z_1, z_2\}$ (each in 2 dims)

Kernels as (Implicit) Feature Maps

- Consider two data points $\boldsymbol{x} = \{x_1, x_2\}$ and $\boldsymbol{z} = \{z_1, z_2\}$ (each in 2 dims)
- Suppose we have a function k which takes as inputs $\boldsymbol{x}$ and $\boldsymbol{y}$ and computes

$$k(\boldsymbol{x}, \boldsymbol{z}) \quad = \quad (\boldsymbol{x}^\top \boldsymbol{z})^2$$

Kernels as (Implicit) Feature Maps

- Consider two data points $\boldsymbol{x} = \{x_1, x_2\}$ and $\boldsymbol{z} = \{z_1, z_2\}$ (each in 2 dims)
- Suppose we have a function k which takes as inputs $\boldsymbol{x}$ and $\boldsymbol{y}$ and computes

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

Kernels as (Implicit) Feature Maps

- Consider two data points $\boldsymbol{x} = \{x_1, x_2\}$ and $\boldsymbol{z} = \{z_1, z_2\}$ (each in 2 dims)
- Suppose we have a function k which takes as inputs $\boldsymbol{x}$ and $\boldsymbol{y}$ and computes

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

Kernels as (Implicit) Feature Maps

- Consider two data points $\boldsymbol{x} = \{x_1, x_2\}$ and $\boldsymbol{z} = \{z_1, z_2\}$ (each in 2 dims)
- Suppose we have a function k which takes as inputs $\boldsymbol{x}$ and $\boldsymbol{y}$ and computes

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

Kernels as (Implicit) Feature Maps

- Consider two data points $\boldsymbol{x} = \{x_1, x_2\}$ and $\boldsymbol{z} = \{z_1, z_2\}$ (each in 2 dims)
- Suppose we have a function k which takes as inputs $\boldsymbol{x}$ and $\boldsymbol{y}$ and computes

$$
\begin{aligned}
k(\boldsymbol{x}, \boldsymbol{z}) &= (\boldsymbol{x}^\top \boldsymbol{z})^2 \\
&= (x_1 z_1 + x_2 z_2)^2 \\
&= x_1^2 z_1^2 + x_2^2 z_2^2 + 2 x_1 x_2 z_1 z_2 \\
&= (x_1^2, \sqrt{2} x_1 x_2, x_2^2)^\top (z_1^2, \sqrt{2} z_1 z_2, z_2^2) \\
&= \phi(\boldsymbol{x})^\top \phi(\boldsymbol{z}) \quad \text{(an inner product)}
\end{aligned}
$$

Kernels as (Implicit) Feature Maps

- Consider two data points $\boldsymbol{x} = \{x_1, x_2\}$ and $\boldsymbol{z} = \{z_1, z_2\}$ (each in 2 dims)
- Suppose we have a function k which takes as inputs $\boldsymbol{x}$ and $\boldsymbol{y}$ and computes

$$
\begin{aligned}
k(\boldsymbol{x}, \boldsymbol{z}) &= (\boldsymbol{x}^\top \boldsymbol{z})^2 \\
&= (x_1 z_1 + x_2 z_2)^2 \\
&= x_1^2 z_1^2 + x_2^2 z_2^2 + 2 x_1 x_2 z_1 z_2 \\
&= (x_1^2, \sqrt{2} x_1 x_2, x_2^2)^\top (z_1^2, \sqrt{2} z_1 z_2, z_2^2) \\
&= \phi(\boldsymbol{x})^\top \phi(\boldsymbol{z}) \quad \text{(an inner product)}
\end{aligned}
$$

- k (known as "kernel function") implicitly defines a mapping ϕ to a higher-dim space

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

Kernels as (Implicit) Feature Maps

- Consider two data points $\boldsymbol{x} = \{x_1, x_2\}$ and $\boldsymbol{z} = \{z_1, z_2\}$ (each in 2 dims)
- Suppose we have a function k which takes as inputs $\boldsymbol{x}$ and $\boldsymbol{y}$ and computes

$$
\begin{aligned}
k(\boldsymbol{x}, \boldsymbol{z}) &= (\boldsymbol{x}^\top \boldsymbol{z})^2 \\
&= (x_1 z_1 + x_2 z_2)^2 \\
&= x_1^2 z_1^2 + x_2^2 z_2^2 + 2 x_1 x_2 z_1 z_2 \\
&= (x_1^2, \sqrt{2} x_1 x_2, x_2^2)^\top (z_1^2, \sqrt{2} z_1 z_2, z_2^2) \\
&= \phi(\boldsymbol{x})^\top \phi(\boldsymbol{z}) \quad \text{(an inner product)}
\end{aligned}
$$

- k (known as "kernel function") implicitly defines a mapping ϕ to a higher-dim space

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

.. and computes inner-product based similarity $\phi(\boldsymbol{x})^\top \phi(\boldsymbol{z})$ in that space

Kernels as (Implicit) Feature Maps

- Consider two data points $\boldsymbol{x} = \{x_1, x_2\}$ and $\boldsymbol{z} = \{z_1, z_2\}$ (each in 2 dims)
- Suppose we have a function k which takes as inputs $\boldsymbol{x}$ and $\boldsymbol{y}$ and computes

$$\begin{aligned} k(\boldsymbol{x}, \boldsymbol{z}) &= (\boldsymbol{x}^\top \boldsymbol{z})^2 \\ &= (x_1 z_1 + x_2 z_2)^2 \\ &= x_1^2 z_1^2 + x_2^2 z_2^2 + 2x_1 x_2 z_1 z_2 \\ &= (x_1^2, \sqrt{2}x_1 x_2, x_2^2)^\top (z_1^2, \sqrt{2} z_1 z_2, z_2^2) \\ &= \phi(\boldsymbol{x})^\top \phi(\boldsymbol{z}) \quad \text{(an inner product)} \end{aligned}$$

- k (known as "kernel function") implicitly defines a mapping ϕ to a higher-dim space

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

.. and computes inner-product based similarity $\phi(\boldsymbol{x})^\top \phi(\boldsymbol{z})$ in that space

- We didn't need to pre-define/compute the mapping ϕ to compute $k(\boldsymbol{x}, \boldsymbol{z})$

Kernels as (Implicit) Feature Maps

- Consider two data points $\boldsymbol{x} = \{x_1, x_2\}$ and $\boldsymbol{z} = \{z_1, z_2\}$ (each in 2 dims)
- Suppose we have a function k which takes as inputs $\boldsymbol{x}$ and $\boldsymbol{y}$ and computes

$$\begin{aligned} k(\boldsymbol{x}, \boldsymbol{z}) &= (\boldsymbol{x}^\top \boldsymbol{z})^2 \\ &= (x_1 z_1 + x_2 z_2)^2 \\ &= x_1^2 z_1^2 + x_2^2 z_2^2 + 2 x_1 x_2 z_1 z_2 \\ &= (x_1^2, \sqrt{2} x_1 x_2, x_2^2)^\top (z_1^2, \sqrt{2} z_1 z_2, z_2^2) \\ &= \phi(\boldsymbol{x})^\top \phi(\boldsymbol{z}) \quad \text{(an inner product)} \end{aligned}$$

- k (known as "kernel function") implicitly defines a mapping ϕ to a higher-dim space

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

.. and computes inner-product based similarity $\phi(\boldsymbol{x})^\top \phi(\boldsymbol{z})$ in that space

- We didn't need to pre-define/compute the mapping ϕ to compute $k(\boldsymbol{x}, \boldsymbol{z})$
- We can simply use the definition of the kernel - $(\boldsymbol{x}^\top \boldsymbol{z})^2$ in this case

Kernels as (Implicit) Feature Maps

- Consider two data points $\boldsymbol{x} = \{x_1, x_2\}$ and $\boldsymbol{z} = \{z_1, z_2\}$ (each in 2 dims)
- Suppose we have a function k which takes as inputs $\boldsymbol{x}$ and $\boldsymbol{y}$ and computes

$$
\begin{aligned}
k(\boldsymbol{x}, \boldsymbol{z}) &= (\boldsymbol{x}^\top \boldsymbol{z})^2 \\
&= (x_1 z_1 + x_2 z_2)^2 \\
&= x_1^2 z_1^2 + x_2^2 z_2^2 + 2 x_1 x_2 z_1 z_2 \\
&= (x_1^2, \sqrt{2} x_1 x_2, x_2^2)^\top (z_1^2, \sqrt{2} z_1 z_2, z_2^2) \\
&= \phi(\boldsymbol{x})^\top \phi(\boldsymbol{z}) \quad \text{(an inner product)}
\end{aligned}
$$

- k (known as "kernel function") implicitly defines a mapping ϕ to a higher-dim space

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

  .. and computes inner-product based similarity $\phi(\boldsymbol{x})^\top \phi(\boldsymbol{z})$ in that space

- We didn't need to pre-define/compute the mapping ϕ to compute $k(\boldsymbol{x}, \boldsymbol{z})$
- We can simply use the definition of the kernel - $(\boldsymbol{x}^\top \boldsymbol{z})^2$ in this case
- Also, evaluating $k(\boldsymbol{x}, \boldsymbol{z}) = \phi(\boldsymbol{x})^\top \phi(\boldsymbol{z})$ is almost as fast as computing the inner product $\boldsymbol{x}^\top \boldsymbol{z}$

Kernel Functions

- Every kernel function k implicitly defines a feature mapping ϕ

Kernel Functions

- Every kernel function k implicitly defines a feature mapping ϕ
- ϕ takes input $\boldsymbol{x} \in \mathcal{X}$ (input space) and maps it to $\mathcal{F}$ (new "feature space")

Kernel Functions

- Every kernel function k implicitly defines a feature mapping ϕ
- ϕ takes input $\boldsymbol{x} \in \mathcal{X}$ (input space) and maps it to $\mathcal{F}$ (new "feature space")
- The kernel function k can be seen as taking two points as inputs and computing their inner-product based similarity in the $\mathcal{F}$ space

$$
\begin{aligned}
\phi &: \mathcal{X} \to \mathcal{F} \\
k &: \mathcal{X} \times \mathcal{X} \to \mathbb{R}, \quad k(\boldsymbol{x}, \boldsymbol{z}) = \phi(\boldsymbol{x})^\top \phi(\boldsymbol{z})
\end{aligned}
$$

Kernel Functions

- Every kernel function k implicitly defines a feature mapping ϕ
- ϕ takes input $\boldsymbol{x} \in \mathcal{X}$ (input space) and maps it to $\mathcal{F}$ (new "feature space")
- The kernel function k can be seen as taking two points as inputs and computing their inner-product based similarity in the $\mathcal{F}$ space

$$
\begin{aligned}
\phi &: \mathcal{X} \to \mathcal{F} \\
k &: \mathcal{X} \times \mathcal{X} \to \mathbb{R}, \quad k(\boldsymbol{x}, \boldsymbol{z}) = \phi(\boldsymbol{x})^\top \phi(\boldsymbol{z})
\end{aligned}
$$

- $\mathcal{F}$ needs to be a *vector space* with a *dot product* defined on it
  - Also called a *Hilbert Space*

Kernel Functions

- Every kernel function k implicitly defines a feature mapping ϕ
- ϕ takes input $\boldsymbol{x} \in \mathcal{X}$ (input space) and maps it to $\mathcal{F}$ (new "feature space")
- The kernel function k can be seen as taking two points as inputs and computing their inner-product based similarity in the $\mathcal{F}$ space

$$
\begin{aligned}
\phi &: \mathcal{X} \rightarrow \mathcal{F} \\
k &: \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}, \quad k(\boldsymbol{x}, \boldsymbol{z}) = \phi(\boldsymbol{x})^\top \phi(\boldsymbol{z})
\end{aligned}
$$

- $\mathcal{F}$ needs to be a *vector space* with a *dot product* defined on it
  - Also called a *Hilbert Space*
- Is *any* function k with $k(\boldsymbol{x}, \boldsymbol{z}) = \phi(\boldsymbol{x})^\top \phi(\boldsymbol{z})$ for some ϕ, a kernel function?

Kernel Functions

- Every kernel function k implicitly defines a feature mapping ϕ
- ϕ takes input $\boldsymbol{x} \in \mathcal{X}$ (input space) and maps it to $\mathcal{F}$ (new "feature space")
- The kernel function k can be seen as taking two points as inputs and computing their inner-product based similarity in the $\mathcal{F}$ space

$$\begin{aligned} \phi &: \mathcal{X} \rightarrow \mathcal{F} \\ k &: \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}, \quad k(\boldsymbol{x}, \boldsymbol{z}) = \phi(\boldsymbol{x})^\top \phi(\boldsymbol{z}) \end{aligned}$$

- $\mathcal{F}$ needs to be a *vector space* with a *dot product* defined on it
  - Also called a *Hilbert Space*
- Is *any* function k with $k(\boldsymbol{x}, \boldsymbol{z}) = \phi(\boldsymbol{x})^\top \phi(\boldsymbol{z})$ for some ϕ, a kernel function?
  - No. The function k must satisfy **Mercer's Condition**

Kernel Functions

- For k to be a kernel function

Kernel Functions

- For k to be a kernel function
  - k must define a dot product for some Hilbert Space $\mathcal{F}$

Kernel Functions

- For k to be a kernel function
  - k must define a dot product for some Hilbert Space $\mathcal{F}$
  - Above is true if k is symmetric and positive semi-definite (p.s.d.) function

Kernel Functions

- For k to be a kernel function
  - k must define a dot product for some Hilbert Space $\mathcal{F}$
  - Above is true if k is symmetric and positive semi-definite (p.s.d.) function (though there are exceptions; there are also "indefinite" kernels).

Kernel Functions

- For k to be a kernel function
  - k must define a dot product for some Hilbert Space $\mathcal{F}$
  - Above is true if k is symmetric and positive semi-definite (p.s.d.) function (though there are exceptions; there are also "indefinite" kernels).
  - The function k is p.s.d. if the following holds

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

.. for all functions f that are "square integrable", i.e., $\int f(\boldsymbol{x})^2 d\boldsymbol{x} < \infty$

Kernel Functions

- For k to be a kernel function
  - k must define a dot product for some Hilbert Space $\mathcal{F}$
  - Above is true if k is symmetric and positive semi-definite (p.s.d.) function (though there are exceptions; there are also "indefinite" kernels).
  - The function k is p.s.d. if the following holds

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

  .. for all functions f that are "square integrable", i.e., $\int f(\boldsymbol{x})^2 d\boldsymbol{x} < \infty$

  - This is the Mercer's Condition

Kernel Functions

- For k to be a kernel function
  - k must define a dot product for some Hilbert Space $\mathcal{F}$
  - Above is true if k is symmetric and positive semi-definite (p.s.d.) function (though there are exceptions; there are also "indefinite" kernels).
  - The function k is p.s.d. if the following holds

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

  .. for all functions f that are "square integrable", i.e., $\int f(\boldsymbol{x})^2 d\boldsymbol{x} < \infty$
  - This is the Mercer's Condition
- Let k_1, k_2 be two kernel functions then the following are as well:
  - $k(\boldsymbol{x}, \boldsymbol{z}) = k_1(\boldsymbol{x}, \boldsymbol{z}) + k_2(\boldsymbol{x}, \boldsymbol{z})$: direct sum
  - $k(\boldsymbol{x}, \boldsymbol{z}) = \alpha k_1(\boldsymbol{x}, \boldsymbol{z})$: scalar product
  - $k(\boldsymbol{x}, \boldsymbol{z}) = k_1(\boldsymbol{x}, \boldsymbol{z}) k_2(\boldsymbol{x}, \boldsymbol{z})$: direct product

Kernel Functions

- For k to be a kernel function
  - k must define a dot product for some Hilbert Space $\mathcal{F}$
  - Above is true if k is symmetric and positive semi-definite (p.s.d.) function (though there are exceptions; there are also "indefinite" kernels).
  - The function k is p.s.d. if the following holds

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

  .. for all functions f that are "square integrable", i.e., $\int f(\boldsymbol{x})^2 d\boldsymbol{x} < \infty$
  - This is the Mercer's Condition
- Let k_1, k_2 be two kernel functions then the following are as well:
  - $k(\boldsymbol{x},\boldsymbol{z}) = k_1(\boldsymbol{x},\boldsymbol{z}) + k_2(\boldsymbol{x},\boldsymbol{z})$: direct sum
  - $k(\boldsymbol{x},\boldsymbol{z}) = \alpha k_1(\boldsymbol{x},\boldsymbol{z})$: scalar product
  - $k(\boldsymbol{x},\boldsymbol{z}) = k_1(\boldsymbol{x},\boldsymbol{z})k_2(\boldsymbol{x},\boldsymbol{z})$: direct product
  - Kernels can also be constructed by composing these rules

Some Examples of Kernel Functions

- Linear (trivial) Kernel:

$$k(\boldsymbol{x}, \boldsymbol{z}) = \boldsymbol{x}^\top \boldsymbol{z} \text{ (mapping function } \phi \text{ is identity)}$$

Some Examples of Kernel Functions

- Linear (trivial) Kernel:

$$k(\boldsymbol{x}, \boldsymbol{z}) = \boldsymbol{x}^\top \boldsymbol{z} \text{ (mapping function } \phi \text{ is identity)}$$

- Quadratic Kernel:

$$k(\boldsymbol{x}, \boldsymbol{z}) = (\boldsymbol{x}^\top \boldsymbol{z})^2 \quad \text{or} \quad (1 + \boldsymbol{x}^\top \boldsymbol{z})^2$$

Some Examples of Kernel Functions

- Linear (trivial) Kernel:

$$k(\boldsymbol{x}, \boldsymbol{z}) = \boldsymbol{x}^\top \boldsymbol{z} \text{ (mapping function } \phi \text{ is identity)}$$

- Quadratic Kernel:

$$k(\boldsymbol{x}, \boldsymbol{z}) = (\boldsymbol{x}^\top \boldsymbol{z})^2 \quad \text{or} \quad (1 + \boldsymbol{x}^\top \boldsymbol{z})^2$$

- Polynomial Kernel (of degree d):

$$k(\boldsymbol{x}, \boldsymbol{z}) = (\boldsymbol{x}^\top \boldsymbol{z})^d \quad \text{or} \quad (1 + \boldsymbol{x}^\top \boldsymbol{z})^d$$

Some Examples of Kernel Functions

- Linear (trivial) Kernel:

$$k(\boldsymbol{x}, \boldsymbol{z}) = \boldsymbol{x}^\top \boldsymbol{z} \text{ (mapping function } \phi \text{ is identity)}$$

- Quadratic Kernel:

$$k(\boldsymbol{x}, \boldsymbol{z}) = (\boldsymbol{x}^\top \boldsymbol{z})^2 \quad \text{or} \quad (1 + \boldsymbol{x}^\top \boldsymbol{z})^2$$

- Polynomial Kernel (of degree d):

$$k(\boldsymbol{x}, \boldsymbol{z}) = (\boldsymbol{x}^\top \boldsymbol{z})^d \quad \text{or} \quad (1 + \boldsymbol{x}^\top \boldsymbol{z})^d$$

- Radial Basis Function (RBF) of "Gaussian" Kernel:

$$k(\boldsymbol{x}, \boldsymbol{z}) = \exp[-\gamma ||\boldsymbol{x} - \boldsymbol{z}||^2]$$

  - γ is a hyperparameter (also called the kernel bandwidth)

Some Examples of Kernel Functions

- Linear (trivial) Kernel:

$$k(\boldsymbol{x}, \boldsymbol{z}) = \boldsymbol{x}^\top \boldsymbol{z} \text{ (mapping function } \phi \text{ is identity)}$$

- Quadratic Kernel:

$$k(\boldsymbol{x}, \boldsymbol{z}) = (\boldsymbol{x}^\top \boldsymbol{z})^2 \quad \text{or} \quad (1 + \boldsymbol{x}^\top \boldsymbol{z})^2$$

- Polynomial Kernel (of degree d):

$$k(\boldsymbol{x}, \boldsymbol{z}) = (\boldsymbol{x}^\top \boldsymbol{z})^d \quad \text{or} \quad (1 + \boldsymbol{x}^\top \boldsymbol{z})^d$$

- Radial Basis Function (RBF) of "Gaussian" Kernel:

$$k(\boldsymbol{x}, \boldsymbol{z}) = \exp[-\gamma ||\boldsymbol{x} - \boldsymbol{z}||^2]$$

  - γ is a hyperparameter (also called the kernel bandwidth)
  - The RBF kernel corresponds to an infinite dimensional feature space $\mathcal{F}$ (i.e., you can't actually write down or store the map $\phi(\boldsymbol{x})$ explicitly)

Some Examples of Kernel Functions

- Linear (trivial) Kernel:

$$k(\boldsymbol{x}, \boldsymbol{z}) = \boldsymbol{x}^\top \boldsymbol{z} \text{ (mapping function } \phi \text{ is identity)}$$

- Quadratic Kernel:

$$k(\boldsymbol{x}, \boldsymbol{z}) = (\boldsymbol{x}^\top \boldsymbol{z})^2 \quad \text{or} \quad (1 + \boldsymbol{x}^\top \boldsymbol{z})^2$$

- Polynomial Kernel (of degree d):

$$k(\boldsymbol{x}, \boldsymbol{z}) = (\boldsymbol{x}^\top \boldsymbol{z})^d \quad \text{or} \quad (1 + \boldsymbol{x}^\top \boldsymbol{z})^d$$

- Radial Basis Function (RBF) of "Gaussian" Kernel:

$$k(\boldsymbol{x}, \boldsymbol{z}) = \exp[-\gamma||\boldsymbol{x} - \boldsymbol{z}||^2]$$

  - γ is a hyperparameter (also called the kernel bandwidth)
  - The RBF kernel corresponds to an infinite dimensional feature space $\mathcal{F}$ (i.e., you can't actually write down or store the map $\phi(\boldsymbol{x})$ explicitly)
  - Also called "stationary kernel": only depends on the distance between $\boldsymbol{x}$ and $\boldsymbol{z}$ (translating both by the same amount won't change the value of $k(\boldsymbol{x}, \boldsymbol{z})$)

Some Examples of Kernel Functions

- Linear (trivial) Kernel:

$$k(\boldsymbol{x}, \boldsymbol{z}) = \boldsymbol{x}^\top \boldsymbol{z} \text{ (mapping function } \phi \text{ is identity)}$$

- Quadratic Kernel:

$$k(\boldsymbol{x}, \boldsymbol{z}) = (\boldsymbol{x}^\top \boldsymbol{z})^2 \quad \text{or} \quad (1 + \boldsymbol{x}^\top \boldsymbol{z})^2$$

- Polynomial Kernel (of degree d):

$$k(\boldsymbol{x}, \boldsymbol{z}) = (\boldsymbol{x}^\top \boldsymbol{z})^d \quad \text{or} \quad (1 + \boldsymbol{x}^\top \boldsymbol{z})^d$$

- Radial Basis Function (RBF) of "Gaussian" Kernel:

$$k(\boldsymbol{x}, \boldsymbol{z}) = \exp[-\gamma ||\boldsymbol{x} - \boldsymbol{z}||^2]$$

  - γ is a hyperparameter (also called the kernel bandwidth)
  - The RBF kernel corresponds to an infinite dimensional feature space $\mathcal{F}$ (i.e., you can't actually write down or store the map $\phi(\boldsymbol{x})$ explicitly)
  - Also called "stationary kernel": only depends on the distance between $\boldsymbol{x}$ and $\boldsymbol{z}$ (translating both by the same amount won't change the value of $k(\boldsymbol{x}, \boldsymbol{z})$)
- Kernel hyperparameters (e.g., d, γ) need to be chosen via cross-validation

RBF Kernel = Infinite Dimensional Mapping

- We saw that the RBF/Gaussian kernel is defined as

$$k(\boldsymbol{x}, \boldsymbol{z}) = \exp[-\gamma||\boldsymbol{x} - \boldsymbol{z}||^2]$$

- Using this kernel corresponds to mapping data to infinite dimensional space

RBF Kernel = Infinite Dimensional Mapping

- We saw that the RBF/Gaussian kernel is defined as

$$k(\boldsymbol{x}, \boldsymbol{z}) = \exp[-\gamma ||\boldsymbol{x} - \boldsymbol{z}||^2]$$

- Using this kernel corresponds to mapping data to infinite dimensional space
- This is explained below (assume $\boldsymbol{x}$ and $\boldsymbol{z}$ to be scalar and $\gamma = 1$):

$$k(\boldsymbol{x}, \boldsymbol{z}) \quad = \quad \exp[-(x - z)^2]$$

RBF Kernel = Infinite Dimensional Mapping

- We saw that the RBF/Gaussian kernel is defined as

$$k(\boldsymbol{x}, \boldsymbol{z}) = \exp[-\gamma||\boldsymbol{x} - \boldsymbol{z}||^2]$$

- Using this kernel corresponds to mapping data to infinite dimensional space
- This is explained below (assume $\boldsymbol{x}$ and $\boldsymbol{z}$ to be scalar and $\gamma = 1$):

$$\begin{aligned} k(\boldsymbol{x}, \boldsymbol{z}) &= \exp[-(x - z)^2] \\ &= \exp(-x^2)\exp(-z^2)\exp(2xz) \end{aligned}$$

RBF Kernel = Infinite Dimensional Mapping

- We saw that the RBF/Gaussian kernel is defined as

$$k(\boldsymbol{x}, \boldsymbol{z}) = \exp[-\gamma ||\boldsymbol{x} - \boldsymbol{z}||^2]$$

- Using this kernel corresponds to mapping data to infinite dimensional space
- This is explained below (assume $\boldsymbol{x}$ and $\boldsymbol{z}$ to be scalar and $\gamma = 1$):

$$\begin{aligned}
k(\boldsymbol{x}, \boldsymbol{z}) &= \exp[-(x - z)^2] \\
&= \exp(-x^2)\exp(-z^2)\exp(2xz) \\
&= \exp(-x^2)\exp(-z^2)\sum_{k=1}^{\infty}\frac{2^k x^k z^k}{k!}
\end{aligned}$$

RBF Kernel = Infinite Dimensional Mapping

- We saw that the RBF/Gaussian kernel is defined as

$$k(\boldsymbol{x}, \boldsymbol{z}) = \exp[-\gamma ||\boldsymbol{x} - \boldsymbol{z}||^2]$$

- Using this kernel corresponds to mapping data to infinite dimensional space
- This is explained below (assume $\boldsymbol{x}$ and $\boldsymbol{z}$ to be scalar and $\gamma = 1$):

$$\begin{aligned} k(\boldsymbol{x}, \boldsymbol{z}) &= \exp[-(x-z)^2] \\ &= \exp(-x^2)\exp(-z^2)\exp(2xz) \\ &= \color{red}{\exp(-x^2)}\color{blue}{\exp(-z^2)}\color{black}{\sum_{k=1}^{\infty}\frac{2^k \color{red}{x^k}\color{blue}{z^k}}{k!}} \\ &= \color{red}{\boldsymbol{\phi(x)}}^\top\color{blue}{\boldsymbol{\phi(z)}} \qquad \text{(the constants } 2^k \text{ and } k! \text{ are subsumed in } \phi) \end{aligned}$$

RBF Kernel = Infinite Dimensional Mapping

- We saw that the RBF/Gaussian kernel is defined as

$$k(\boldsymbol{x}, \boldsymbol{z}) = \exp[-\gamma ||\boldsymbol{x} - \boldsymbol{z}||^2]$$

- Using this kernel corresponds to mapping data to infinite dimensional space
- This is explained below (assume $\boldsymbol{x}$ and $\boldsymbol{z}$ to be scalar and $\gamma = 1$):

$$\begin{aligned} k(\boldsymbol{x}, \boldsymbol{z}) &= \exp[-(x - z)^2] \\ &= \exp(-x^2)\exp(-z^2)\exp(2xz) \\ &= \exp(-x^2)\exp(-z^2)\sum_{k=1}^{\infty}\frac{2^k x^k z^k}{k!} \\ &= \phi(\boldsymbol{x})^\top \phi(\boldsymbol{z}) \qquad \text{(the constants } 2^k \text{ and } k! \text{ are subsumed in } \phi) \end{aligned}$$

- This shows that $\phi(x)$ and $\phi(z)$ are infinite dimensional vectors

RBF Kernel = Infinite Dimensional Mapping

- We saw that the RBF/Gaussian kernel is defined as

$$k(\boldsymbol{x}, \boldsymbol{z}) = \exp[-\gamma ||\boldsymbol{x} - \boldsymbol{z}||^2]$$

- Using this kernel corresponds to mapping data to infinite dimensional space
- This is explained below (assume $\boldsymbol{x}$ and $\boldsymbol{z}$ to be scalar and $\gamma = 1$):

$$\begin{aligned} k(\boldsymbol{x}, \boldsymbol{z}) &= \exp[-(x-z)^2] \\ &= \exp(-x^2)\exp(-z^2)\exp(2xz) \\ &= \exp(-x^2)\exp(-z^2)\sum_{k=1}^{\infty}\frac{2^k x^k z^k}{k!} \\ &= \boldsymbol{\phi(x)}^\top \boldsymbol{\phi(z)} \qquad \text{(the constants } 2^k \text{ and } k! \text{ are subsumed in } \phi) \end{aligned}$$

- This shows that $\phi(x)$ and $\phi(z)$ are infinite dimensional vectors
    - But we didn't have to compute these to get $k(\boldsymbol{x}, \boldsymbol{z})$

The Kernel Matrix

- The kernel function k defines the Kernel Matrix $\mathbf{K}$ over the data

The Kernel Matrix

- The kernel function k defines the Kernel Matrix $\mathbf{K}$ over the data
- Given N examples $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$, the (i, j)-th entry of $\mathbf{K}$ is defined as:

$$K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \phi(\boldsymbol{x}_i)^\top \phi(\boldsymbol{x}_j)$$

The Kernel Matrix

- The kernel function k defines the Kernel Matrix $\mathbf{K}$ over the data
- Given N examples $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$, the (i,j)-th entry of $\mathbf{K}$ is defined as:

$$K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \phi(\boldsymbol{x}_i)^\top \phi(\boldsymbol{x}_j)$$

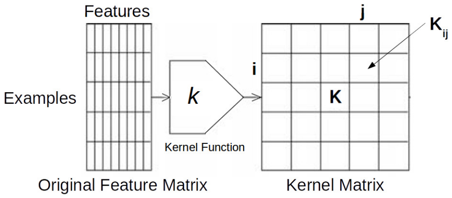
- K_{ij}: Similarity between the i-th and j-th example in the feature space $\mathcal{F}$

The Kernel Matrix

- The kernel function k defines the Kernel Matrix $\mathbf{K}$ over the data
- Given N examples $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$, the (i,j)-th entry of $\mathbf{K}$ is defined as:

$$K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \phi(\boldsymbol{x}_i)^\top \phi(\boldsymbol{x}_j)$$

- K_{ij}: Similarity between the i-th and j-th example in the feature space $\mathcal{F}$
- $\mathbf{K}$: $N \times N$ matrix of pairwise similarities between examples in $\mathcal{F}$ space

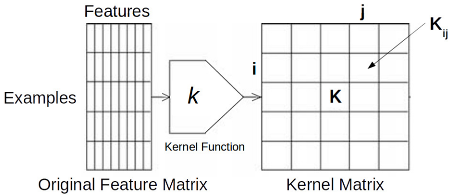


The Kernel Matrix

- The kernel function k defines the Kernel Matrix $\mathbf{K}$ over the data
- Given N examples $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$, the (i,j)-th entry of $\mathbf{K}$ is defined as:

$$K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \phi(\boldsymbol{x}_i)^\top \phi(\boldsymbol{x}_j)$$

- K_{ij}: Similarity between the i-th and j-th example in the feature space $\mathcal{F}$
- $\mathbf{K}$: $N \times N$ matrix of pairwise similarities between examples in $\mathcal{F}$ space



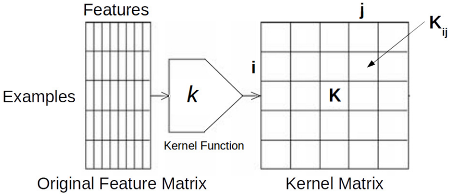
- $\mathbf{K}$ is a symmetric and positive definite matrix

The Kernel Matrix

- The kernel function k defines the Kernel Matrix $\mathbf{K}$ over the data
- Given N examples $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$, the (i,j)-th entry of $\mathbf{K}$ is defined as:

$$K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \phi(\boldsymbol{x}_i)^\top \phi(\boldsymbol{x}_j)$$

- K_{ij}: Similarity between the i-th and j-th example in the feature space $\mathcal{F}$
- $\mathbf{K}$: $N \times N$ matrix of pairwise similarities between examples in $\mathcal{F}$ space



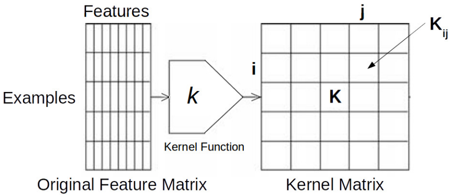
- $\mathbf{K}$ is a symmetric and positive definite matrix
- For a P.D. matrix: $\boldsymbol{z}^\top \mathbf{K} \boldsymbol{z} > 0, \quad \forall \boldsymbol{z} \in \mathbb{R}^N$ (also, all eigenvalues positive)

The Kernel Matrix

- The kernel function k defines the Kernel Matrix $\mathbf{K}$ over the data
- Given N examples $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$, the (i, j)-th entry of $\mathbf{K}$ is defined as:

$$K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \phi(\boldsymbol{x}_i)^\top \phi(\boldsymbol{x}_j)$$

- K_{ij}: Similarity between the i-th and j-th example in the feature space $\mathcal{F}$
- $\mathbf{K}$: $N \times N$ matrix of pairwise similarities between examples in $\mathcal{F}$ space



- $\mathbf{K}$ is a symmetric and positive definite matrix
- For a P.D. matrix: $\boldsymbol{z}^\top \mathbf{K} \boldsymbol{z} > 0, \quad \forall \boldsymbol{z} \in \mathbb{R}^N$ (also, all eigenvalues positive)
- The Kernel Matrix $\mathbf{K}$ is also known as the Gram Matrix

Using Kernels

- Kernels can turn a linear model into a nonlinear one

Using Kernels

- Kernels can turn a linear model into a nonlinear one
- Recall: Kernel $k(\boldsymbol{x}, \boldsymbol{z})$ represents a dot product in some high dimensional feature space $\mathcal{F}$

Using Kernels

- Kernels can turn a linear model into a nonlinear one
- Recall: Kernel $k(\boldsymbol{x}, \boldsymbol{z})$ represents a dot product in some high dimensional feature space $\mathcal{F}$
- Any learning model in which, during training and test, inputs only appear as dot products $(\boldsymbol{x}_i^\top \boldsymbol{x}_j)$ can be kernelized (i.e., non-linearlized)

Using Kernels

- Kernels can turn a linear model into a nonlinear one
- Recall: Kernel $k(\boldsymbol{x}, \boldsymbol{z})$ represents a dot product in some high dimensional feature space $\mathcal{F}$
- Any learning model in which, during training and test, inputs only appear as dot products ($\boldsymbol{x}_i^\top \boldsymbol{x}_j$) can be kernelized (i.e., non-linearlized)
  - .. by replacing the $\boldsymbol{x}_i^\top \boldsymbol{x}_j$ terms by $\phi(\boldsymbol{x}_i)^\top \phi(\boldsymbol{x}_j) = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$

Using Kernels

- Kernels can turn a linear model into a nonlinear one
- Recall: Kernel $k(\boldsymbol{x}, \boldsymbol{z})$ represents a dot product in some high dimensional feature space $\mathcal{F}$
- Any learning model in which, during training and test, inputs only appear as dot products $(\boldsymbol{x}_i^\top \boldsymbol{x}_j)$ can be kernelized (i.e., non-linearlized)
  - .. by replacing the $\boldsymbol{x}_i^\top \boldsymbol{x}_j$ terms by $\phi(\boldsymbol{x}_i)^\top \phi(\boldsymbol{x}_j) = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$
- Most learning algorithms can be easily kernelized

Using Kernels

- Kernels can turn a linear model into a nonlinear one
- Recall: Kernel $k(\boldsymbol{x}, \boldsymbol{z})$ represents a dot product in some high dimensional feature space $\mathcal{F}$
- Any learning model in which, during training and test, inputs only appear as dot products $(\boldsymbol{x}_i^\top \boldsymbol{x}_j)$ can be kernelized (i.e., non-linearlized)
  - .. by replacing the $\boldsymbol{x}_i^\top \boldsymbol{x}_j$ terms by $\phi(\boldsymbol{x}_i)^\top \phi(\boldsymbol{x}_j) = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$
- Most learning algorithms can be easily kernelized
  - Distance based methods, Perceptron, SVM, linear regression, etc.

Using Kernels

- Kernels can turn a linear model into a nonlinear one
- Recall: Kernel $k(\boldsymbol{x}, \boldsymbol{z})$ represents a dot product in some high dimensional feature space $\mathcal{F}$
- Any learning model in which, during training and test, inputs only appear as dot products ($\boldsymbol{x}_i^\top \boldsymbol{x}_j$) can be kernelized (i.e., non-linearlized)
  - .. by replacing the $\boldsymbol{x}_i^\top \boldsymbol{x}_j$ terms by $\phi(\boldsymbol{x}_i)^\top \phi(\boldsymbol{x}_j) = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$
- Most learning algorithms can be easily kernelized
  - Distance based methods, Perceptron, SVM, linear regression, etc.
  - Many of the unsupervised learning algorithms too can be kernelized (e.g., K-means clustering, Principal Component Analysis, etc. - will see later)

Using Kernels

- Kernels can turn a linear model into a nonlinear one
- Recall: Kernel $k(\boldsymbol{x}, \boldsymbol{z})$ represents a dot product in some high dimensional feature space $\mathcal{F}$
- Any learning model in which, during training and test, inputs only appear as dot products $(\boldsymbol{x}_i^\top \boldsymbol{x}_j)$ can be kernelized (i.e., non-linearlized)
  - .. by replacing the $\boldsymbol{x}_i^\top \boldsymbol{x}_j$ terms by $\phi(\boldsymbol{x}_i)^\top \phi(\boldsymbol{x}_j) = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$
- Most learning algorithms can be easily kernelized
  - Distance based methods, Perceptron, SVM, linear regression, etc.
  - Many of the unsupervised learning algorithms too can be kernelized (e.g., K-means clustering, Principal Component Analysis, etc. - will see later)
  - Let's look at two examples: Kernelized SVM and Kernelized Ridge Regression

Example 1:
Kernel (Nonlinear) SVM

Kernelized SVM Training

- Recall the soft-margin SVM dual problem:

$$\boxed{\text{Soft-Margin SVM:} \quad \max_{\boldsymbol{\alpha} \leq C} \; \mathcal{L}_D(\alpha) = \boldsymbol{\alpha}^\top \mathbf{1} - \frac{1}{2}\boldsymbol{\alpha}^\top \mathbf{G}\boldsymbol{\alpha}}$$

.. where we had defined $G_{mn} = y_m y_n \boldsymbol{x}_m^\top \boldsymbol{x}_n$

Kernelized SVM Training

- Recall the soft-margin SVM dual problem:

$$\boxed{\text{Soft-Margin SVM:} \quad \max_{\boldsymbol{\alpha} \leq C} \mathcal{L}_D(\alpha) = \boldsymbol{\alpha}^\top \mathbf{1} - \frac{1}{2}\boldsymbol{\alpha}^\top \mathbf{G}\boldsymbol{\alpha}}$$

.. where we had defined $G_{mn} = y_m y_n \boldsymbol{x}_m^\top \boldsymbol{x}_n$

- Can simply replace $G_{mn} = y_m y_n \boldsymbol{x}_m^\top \boldsymbol{x}_n$ by $y_m y_n K_{mn}$

Kernelized SVM Training

- Recall the soft-margin SVM dual problem:

$$\boxed{\text{Soft-Margin SVM:} \quad \max_{\boldsymbol{\alpha} \leq C} \quad \mathcal{L}_D(\alpha) = \boldsymbol{\alpha}^\top \mathbf{1} - \frac{1}{2}\boldsymbol{\alpha}^\top \mathbf{G}\boldsymbol{\alpha}}$$

.. where we had defined $G_{mn} = y_m y_n \boldsymbol{x}_m^\top \boldsymbol{x}_n$

- Can simply replace $G_{mn} = y_m y_n \boldsymbol{x}_m^\top \boldsymbol{x}_n$ by $y_m y_n K_{mn}$
  - .. where $K_{mn} = k(\boldsymbol{x}_m, \boldsymbol{x}_n) = \phi(\boldsymbol{x}_m)^\top \phi(\boldsymbol{x}_n)$ for a suitable kernel function k

Kernelized SVM Training

- Recall the soft-margin SVM dual problem:

$$\boxed{\text{Soft-Margin SVM:} \quad \max_{\alpha \leq C} \; \mathcal{L}_D(\alpha) = \boldsymbol{\alpha}^\top \mathbf{1} - \frac{1}{2}\boldsymbol{\alpha}^\top \mathbf{G}\boldsymbol{\alpha}}$$

  .. where we had defined $G_{mn} = y_m y_n \boldsymbol{x}_m^\top \boldsymbol{x}_n$
- Can simply replace $G_{mn} = y_m y_n \boldsymbol{x}_m^\top \boldsymbol{x}_n$ by $y_m y_n K_{mn}$
  - .. where $K_{mn} = k(\boldsymbol{x}_m, \boldsymbol{x}_n) = \phi(\boldsymbol{x}_m)^\top \phi(\boldsymbol{x}_n)$ for a suitable kernel function k
- The problem can be solved just like the linear SVM case

Kernelized SVM Training

- Recall the soft-margin SVM dual problem:

$$\boxed{\text{Soft-Margin SVM:} \quad \max_{\alpha \leq C} \ \mathcal{L}_D(\alpha) = \boldsymbol{\alpha}^\top \mathbf{1} - \frac{1}{2}\boldsymbol{\alpha}^\top \mathbf{G}\boldsymbol{\alpha}}$$

  .. where we had defined $G_{mn} = y_m y_n \boldsymbol{x}_m^\top \boldsymbol{x}_n$

- Can simply replace $G_{mn} = y_m y_n \boldsymbol{x}_m^\top \boldsymbol{x}_n$ by $y_m y_n K_{mn}$
  - .. where $K_{mn} = k(\boldsymbol{x}_m, \boldsymbol{x}_n) = \phi(\boldsymbol{x}_m)^\top \phi(\boldsymbol{x}_n)$ for a suitable kernel function k
- The problem can be solved just like the linear SVM case
- The new SVM learns a linear separator in kernel-induced feature space $\mathcal{F}$

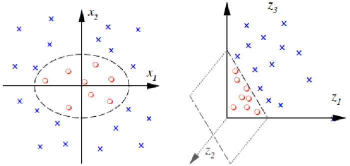
Kernelized SVM Training

- Recall the soft-margin SVM dual problem:

$$\boxed{\text{Soft-Margin SVM:} \quad \max_{\boldsymbol{\alpha} \leq C} \quad \mathcal{L}_D(\alpha) = \boldsymbol{\alpha}^\top \mathbf{1} - \frac{1}{2}\boldsymbol{\alpha}^\top \mathbf{G}\boldsymbol{\alpha}}$$

.. where we had defined $G_{mn} = y_m y_n \boldsymbol{x}_m^\top \boldsymbol{x}_n$

- Can simply replace $G_{mn} = y_m y_n \boldsymbol{x}_m^\top \boldsymbol{x}_n$ by $y_m y_n K_{mn}$
    - .. where $K_{mn} = k(\boldsymbol{x}_m, \boldsymbol{x}_n) = \phi(\boldsymbol{x}_m)^\top \phi(\boldsymbol{x}_n)$ for a suitable kernel function k
- The problem can be solved just like the linear SVM case
- The new SVM learns a linear separator in kernel-induced feature space $\mathcal{F}$
    - This corresponds to a non-linear separator in the original space $\mathcal{X}$



Kernelized SVM Prediction

- Note that the SVM weight vector for the kernelized case can be written as

$$\mathbf{w} = \sum_{n=1}^{N} \alpha_n y_n \phi(\mathbf{x}_n)$$

Kernelized SVM Prediction

- Note that the SVM weight vector for the kernelized case can be written as

$$\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n y_n \phi(\boldsymbol{x}_n)$$

- Prediction for a new test example $\boldsymbol{x}$ (assume $b = 0$)

$$y = \text{sign}(\boldsymbol{w}^\top \phi(\boldsymbol{x}))$$

Kernelized SVM Prediction

- Note that the SVM weight vector for the kernelized case can be written as

$$\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n y_n \phi(\boldsymbol{x}_n)$$

- Prediction for a new test example $\boldsymbol{x}$ (assume $b = 0$)

$$y = \text{sign}(\boldsymbol{w}^\top \phi(\boldsymbol{x})) = \text{sign}(\sum_{n=1}^{N} \alpha_n y_n \phi(\boldsymbol{x}_n)^\top \phi(\boldsymbol{x})) =$$

Kernelized SVM Prediction

- Note that the SVM weight vector for the kernelized case can be written as

$$\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n y_n \phi(\boldsymbol{x}_n)$$

- Prediction for a new test example $\boldsymbol{x}$ (assume $b = 0$)

$$y = \text{sign}(\boldsymbol{w}^\top \phi(\boldsymbol{x})) = \text{sign}(\sum_{n=1}^{N} \alpha_n y_n \phi(\boldsymbol{x}_n)^\top \phi(\boldsymbol{x})) = \text{sign}(\sum_{n=1}^{N} \alpha_n y_n k(\boldsymbol{x}_n, \boldsymbol{x}))$$

Kernelized SVM Prediction

- Note that the SVM weight vector for the kernelized case can be written as

$$\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n y_n \phi(\boldsymbol{x}_n)$$

- Prediction for a new test example $\boldsymbol{x}$ (assume $b = 0$)

$$y = \text{sign}(\boldsymbol{w}^\top \phi(\boldsymbol{x})) = \text{sign}(\sum_{n=1}^{N} \alpha_n y_n \phi(\boldsymbol{x}_n)^\top \phi(\boldsymbol{x})) = \text{sign}(\sum_{n=1}^{N} \alpha_n y_n k(\boldsymbol{x}_n, \boldsymbol{x}))$$

- Note: $\boldsymbol{w}$ can be stored explicitly as a vector only if the feature map $\phi(.)$ can be explicitly written

Kernelized SVM Prediction

- Note that the SVM weight vector for the kernelized case can be written as

$$\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n y_n \phi(\boldsymbol{x}_n)$$

- Prediction for a new test example $\boldsymbol{x}$ (assume $b = 0$)

$$y = \text{sign}(\boldsymbol{w}^\top \phi(\boldsymbol{x})) = \text{sign}(\sum_{n=1}^{N} \alpha_n y_n \phi(\boldsymbol{x}_n)^\top \phi(\boldsymbol{x})) = \text{sign}(\sum_{n=1}^{N} \alpha_n y_n k(\boldsymbol{x}_n, \boldsymbol{x}))$$

- Note: $\boldsymbol{w}$ can be stored explicitly as a vector only if the feature map $\phi(.)$ can be explicitly written
- In general, kernelized SVMs have to store the training data (at least the support vectors for which α_n's are nonzero) even at the test time

Kernelized SVM Prediction

- Note that the SVM weight vector for the kernelized case can be written as

$$\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n y_n \phi(\boldsymbol{x}_n)$$

- Prediction for a new test example $\boldsymbol{x}$ (assume $b = 0$)

$$y = \text{sign}(\boldsymbol{w}^\top \phi(\boldsymbol{x})) = \text{sign}(\sum_{n=1}^{N} \alpha_n y_n \phi(\boldsymbol{x}_n)^\top \phi(\boldsymbol{x})) = \text{sign}(\sum_{n=1}^{N} \alpha_n y_n k(\boldsymbol{x}_n, \boldsymbol{x}))$$

- Note: $\boldsymbol{w}$ can be stored explicitly as a vector only if the feature map $\phi(.)$ can be explicitly written
- In general, kernelized SVMs have to store the training data (at least the support vectors for which α_n's are nonzero) even at the test time
- Thus the prediction time cost of kernel SVM scales linearly in N

Kernelized SVM Prediction

- Note that the SVM weight vector for the kernelized case can be written as

$$\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n y_n \phi(\boldsymbol{x}_n)$$

- Prediction for a new test example $\boldsymbol{x}$ (assume $b = 0$)

$$y = \text{sign}(\boldsymbol{w}^\top \phi(\boldsymbol{x})) = \text{sign}(\sum_{n=1}^{N} \alpha_n y_n \phi(\boldsymbol{x}_n))^\top \phi(\boldsymbol{x})) = \text{sign}(\sum_{n=1}^{N} \alpha_n y_n k(\boldsymbol{x}_n, \boldsymbol{x}))$$

- Note: $\boldsymbol{w}$ can be stored explicitly as a vector only if the feature map $\phi(.)$ can be explicitly written
- In general, kernelized SVMs have to store the training data (at least the support vectors for which α_n's are nonzero) even at the test time
- Thus the prediction time cost of kernel SVM scales linearly in N
- For unkernelized version $\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n y_n \boldsymbol{x}_n$ can be computed and stored as a $D \times 1$ vector. Thus training data need not be stored and the prediction cost is constant w.r.t. N ($\boldsymbol{w}^\top \boldsymbol{x}$ can be computed in $O(D)$ time).

Example 2:
Kernel (Nonlinear) Ridge Regression

Ridge Regression: Revisited

- Recall the ridge regression problem

$$\boldsymbol{w} = \arg\min_{\boldsymbol{w}} \sum_{n=1}^{N} (y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2 + \lambda \boldsymbol{w}^\top \boldsymbol{w}$$

Ridge Regression: Revisited

- Recall the ridge regression problem

$$\boldsymbol{w} = \arg\min_{\boldsymbol{w}} \sum_{n=1}^{N} (y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2 + \lambda \boldsymbol{w}^\top \boldsymbol{w}$$

- The solution to this problem was

$$\boldsymbol{w} = (\sum_{n=1}^{N} \boldsymbol{x}_n \boldsymbol{x}_n^\top + \lambda \mathbf{I}_D)(\sum_{n=1}^{N} y_n \boldsymbol{x}_n) = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}_D)^{-1} \mathbf{X}^\top \boldsymbol{y}$$

Ridge Regression: Revisited

- Recall the ridge regression problem

$$\boldsymbol{w} = \arg\min_{\boldsymbol{w}} \sum_{n=1}^{N} (y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2 + \lambda \boldsymbol{w}^\top \boldsymbol{w}$$

- The solution to this problem was

$$\boldsymbol{w} = (\sum_{n=1}^{N} \boldsymbol{x}_n \boldsymbol{x}_n^\top + \lambda \mathbf{I}_D)(\sum_{n=1}^{N} y_n \boldsymbol{x}_n) = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}_D)^{-1} \mathbf{X}^\top \boldsymbol{y}$$

- Inputs don't appear as inner-products here

Ridge Regression: Revisited

- Recall the ridge regression problem

$$\boldsymbol{w} = \arg\min_{\boldsymbol{w}} \sum_{n=1}^{N} (y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2 + \lambda \boldsymbol{w}^\top \boldsymbol{w}$$

- The solution to this problem was

$$\boldsymbol{w} = (\sum_{n=1}^{N} \boldsymbol{x}_n \boldsymbol{x}_n^\top + \lambda \mathbf{I}_D)(\sum_{n=1}^{N} y_n \boldsymbol{x}_n) = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}_D)^{-1} \mathbf{X}^\top \boldsymbol{y}$$

- Inputs don't appear as inner-products here . They actually do! :-)

Ridge Regression: Revisited

- Recall the ridge regression problem

$$\mathbf{w} = \arg\min_{\mathbf{w}} \sum_{n=1}^{N}(y_n - \mathbf{w}^\top \mathbf{x}_n)^2 + \lambda \mathbf{w}^\top \mathbf{w}$$

- The solution to this problem was

$$\mathbf{w} = (\sum_{n=1}^{N} \mathbf{x}_n \mathbf{x}_n^\top + \lambda \mathbf{I}_D)(\sum_{n=1}^{N} y_n \mathbf{x}_n) = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}_D)^{-1} \mathbf{X}^\top \mathbf{y}$$

- Inputs don't appear as inner-products here . They actually do! :-)
- Matrix inversion lemma: $(\mathbf{F}\mathbf{H}^{-1}\mathbf{G} - \mathbf{E})^{-1}\mathbf{F}\mathbf{H}^{-1} = \mathbf{E}^{-1}\mathbf{F}(\mathbf{G}\mathbf{E}^{-1}\mathbf{F} - \mathbf{H})^{-1}$

Ridge Regression: Revisited

- Recall the ridge regression problem

$$\boldsymbol{w} = \arg\min_{\boldsymbol{w}} \sum_{n=1}^{N} (y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2 + \lambda \boldsymbol{w}^\top \boldsymbol{w}$$

- The solution to this problem was

$$\boldsymbol{w} = (\sum_{n=1}^{N} \boldsymbol{x}_n \boldsymbol{x}_n^\top + \lambda \mathbf{I}_D)(\sum_{n=1}^{N} y_n \boldsymbol{x}_n) = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}_D)^{-1} \mathbf{X}^\top \boldsymbol{y}$$

- Inputs don't appear as inner-products here . They actually do! :-)
- Matrix inversion lemma: $(\mathbf{F}\mathbf{H}^{-1}\mathbf{G} - \mathbf{E})^{-1}\mathbf{F}\mathbf{H}^{-1} = \mathbf{E}^{-1}\mathbf{F}(\mathbf{G}\mathbf{E}^{-1}\mathbf{F} - \mathbf{H})^{-1}$
- The lemma allows us to rewrite $\boldsymbol{w}$ as

$$\boldsymbol{w} = \mathbf{X}^\top (\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I}_N)^{-1} \boldsymbol{y}$$

Ridge Regression: Revisited

- Recall the ridge regression problem

$$\boldsymbol{w} = \arg\min_{\boldsymbol{w}} \sum_{n=1}^{N} (y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2 + \lambda \boldsymbol{w}^\top \boldsymbol{w}$$

- The solution to this problem was

$$\boldsymbol{w} = (\sum_{n=1}^{N} \boldsymbol{x}_n \boldsymbol{x}_n^\top + \lambda \mathbf{I}_D)(\sum_{n=1}^{N} y_n \boldsymbol{x}_n) = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}_D)^{-1} \mathbf{X}^\top \boldsymbol{y}$$

- Inputs don't appear as inner-products here . They actually do! :-)
- Matrix inversion lemma: $(\mathbf{F}\mathbf{H}^{-1}\mathbf{G} - \mathbf{E})^{-1}\mathbf{F}\mathbf{H}^{-1} = \mathbf{E}^{-1}\mathbf{F}(\mathbf{G}\mathbf{E}^{-1}\mathbf{F} - \mathbf{H})^{-1}$
- The lemma allows us to rewrite $\boldsymbol{w}$ as

$$\boldsymbol{w} = \mathbf{X}^\top (\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I}_N)^{-1} \boldsymbol{y} = \mathbf{X}^\top \boldsymbol{\alpha} = \sum_{n=1}^{N} \alpha_n \boldsymbol{x}_n$$

Ridge Regression: Revisited

- Recall the ridge regression problem

$$\mathbf{w} = \arg\min_{\mathbf{w}} \sum_{n=1}^{N} (y_n - \mathbf{w}^\top \mathbf{x}_n)^2 + \lambda \mathbf{w}^\top \mathbf{w}$$

- The solution to this problem was

$$\mathbf{w} = (\sum_{n=1}^{N} \mathbf{x}_n \mathbf{x}_n^\top + \lambda \mathbf{I}_D)(\sum_{n=1}^{N} y_n \mathbf{x}_n) = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}_D)^{-1} \mathbf{X}^\top \mathbf{y}$$

- Inputs don't appear as inner-products here . They actually do! :-)
- Matrix inversion lemma: $(\mathbf{F}\mathbf{H}^{-1}\mathbf{G} - \mathbf{E})^{-1}\mathbf{F}\mathbf{H}^{-1} = \mathbf{E}^{-1}\mathbf{F}(\mathbf{G}\mathbf{E}^{-1}\mathbf{F} - \mathbf{H})^{-1}$
- The lemma allows us to rewrite $\mathbf{w}$ as

$$\mathbf{w} = \mathbf{X}^\top (\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I}_N)^{-1} \mathbf{y} = \mathbf{X}^\top \boldsymbol{\alpha} = \sum_{n=1}^{N} \alpha_n \mathbf{x}_n$$

  where $\boldsymbol{\alpha} = (\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I}_N)^{-1}\mathbf{y} = (\mathbf{K} + \lambda \mathbf{I}_N)^{-1}\mathbf{y}$ is an $N \times 1$ vector of dual variables, and $K_{nm} = \mathbf{x}_n^\top \mathbf{x}_m$

Ridge Regression: Revisited

- Recall the ridge regression problem

$$\boldsymbol{w} = \arg\min_{\boldsymbol{w}} \sum_{n=1}^{N} (y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2 + \lambda \boldsymbol{w}^\top \boldsymbol{w}$$

- The solution to this problem was

$$\boldsymbol{w} = (\sum_{n=1}^{N} \boldsymbol{x}_n \boldsymbol{x}_n^\top + \lambda \mathbf{I}_D)(\sum_{n=1}^{N} y_n \boldsymbol{x}_n) = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}_D)^{-1} \mathbf{X}^\top \boldsymbol{y}$$

- Inputs don't appear as inner-products here . They actually do! :-)
- Matrix inversion lemma: $(\mathbf{F}\mathbf{H}^{-1}\mathbf{G} - \mathbf{E})^{-1}\mathbf{F}\mathbf{H}^{-1} = \mathbf{E}^{-1}\mathbf{F}(\mathbf{G}\mathbf{E}^{-1}\mathbf{F} - \mathbf{H})^{-1}$
- The lemma allows us to rewrite $\boldsymbol{w}$ as

$$\boldsymbol{w} = \mathbf{X}^\top (\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I}_N)^{-1} \boldsymbol{y} = \mathbf{X}^\top \boldsymbol{\alpha} = \sum_{n=1}^{N} \alpha_n \boldsymbol{x}_n$$

  where $\boldsymbol{\alpha} = (\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I}_N)^{-1} \boldsymbol{y} = (\mathbf{K} + \lambda \mathbf{I}_N)^{-1} \boldsymbol{y}$ is an $N \times 1$ vector of dual variables, and $K_{nm} = \boldsymbol{x}_n^\top \boldsymbol{x}_m$

- Note: $\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n \boldsymbol{x}_n$ is known as "dual" form of ridge regression solution. However, so far it is still a linear model. But now it is easily kernelizable.

Kernel (Nonlinear) Ridge Regression

- With the dual form $\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n \boldsymbol{x}_n$, we can kernelize ridge regression

Kernel (Nonlinear) Ridge Regression

- With the dual form $\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n \boldsymbol{x}_n$, we can kernelize ridge regression
- Choosing some kernel k with an associated feature map ϕ, we can write

$$\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n \phi(\boldsymbol{x}_n) = \sum_{n=1}^{N} \alpha_n k(\boldsymbol{x}_n, .)$$

where $\boldsymbol{\alpha} = (\mathbf{K} + \lambda \mathbf{I}_N)^{-1} \boldsymbol{y}$ and $K_{nm} = \phi(\boldsymbol{x}_n)^\top \phi(\boldsymbol{x}_m) = k(\boldsymbol{x}_n, \boldsymbol{x}_m)$

Kernel (Nonlinear) Ridge Regression

- With the dual form $\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n \boldsymbol{x}_n$, we can kernelize ridge regression
- Choosing some kernel k with an associated feature map ϕ, we can write

$$\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n \phi(\boldsymbol{x}_n) = \sum_{n=1}^{N} \alpha_n k(\boldsymbol{x}_n, .)$$

  where $\boldsymbol{\alpha} = (\mathbf{K} + \lambda \mathbf{I}_N)^{-1}\boldsymbol{y}$ and $K_{nm} = \phi(\boldsymbol{x}_n)^\top \phi(\boldsymbol{x}_m) = k(\boldsymbol{x}_n, \boldsymbol{x}_m)$
- Prediction for a new test input $\boldsymbol{x}$ will be

$$y = \boldsymbol{w}^\top \phi(\boldsymbol{x})$$

Kernel (Nonlinear) Ridge Regression

- With the dual form $\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n \boldsymbol{x}_n$, we can kernelize ridge regression
- Choosing some kernel k with an associated feature map ϕ, we can write

$$\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n \phi(\boldsymbol{x}_n) = \sum_{n=1}^{N} \alpha_n k(\boldsymbol{x}_n, .)$$

  where $\boldsymbol{\alpha} = (\mathbf{K} + \lambda \mathbf{I}_N)^{-1} \boldsymbol{y}$ and $K_{nm} = \phi(\boldsymbol{x}_n)^\top \phi(\boldsymbol{x}_m) = k(\boldsymbol{x}_n, \boldsymbol{x}_m)$
- Prediction for a new test input $\boldsymbol{x}$ will be

$$y = \boldsymbol{w}^\top \phi(\boldsymbol{x}) = \sum_{n=1}^{N} \alpha_n \phi(\boldsymbol{x}_n)^\top \phi(\boldsymbol{x})$$

Kernel (Nonlinear) Ridge Regression

- With the dual form $\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n \boldsymbol{x}_n$, we can kernelize ridge regression
- Choosing some kernel k with an associated feature map ϕ, we can write

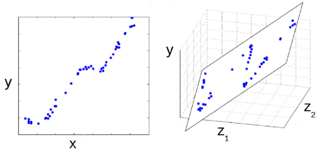
$$\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n \phi(\boldsymbol{x}_n) = \sum_{n=1}^{N} \alpha_n k(\boldsymbol{x}_n, .)$$

  where $\boldsymbol{\alpha} = (\mathbf{K} + \lambda \mathbf{I}_N)^{-1} \boldsymbol{y}$ and $K_{nm} = \phi(\boldsymbol{x}_n)^\top \phi(\boldsymbol{x}_m) = k(\boldsymbol{x}_n, \boldsymbol{x}_m)$

- Prediction for a new test input $\boldsymbol{x}$ will be

$$y = \boldsymbol{w}^\top \phi(\boldsymbol{x}) = \sum_{n=1}^{N} \alpha_n \phi(\boldsymbol{x}_n)^\top \phi(\boldsymbol{x}) = \sum_{n=1}^{N} \alpha_n k(\boldsymbol{x}_n, \boldsymbol{x})$$

- Thus, using the kernel, we effectively learn a nonlinear regression model



Kernel (Nonlinear) Ridge Regression

- With the dual form $\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n \boldsymbol{x}_n$, we can kernelize ridge regression
- Choosing some kernel k with an associated feature map ϕ, we can write

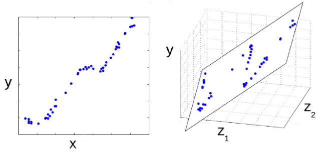
$$\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n \phi(\boldsymbol{x}_n) = \sum_{n=1}^{N} \alpha_n k(\boldsymbol{x}_n, .)$$

where $\boldsymbol{\alpha} = (\mathbf{K} + \lambda \mathbf{I}_N)^{-1}\boldsymbol{y}$ and $K_{nm} = \phi(\boldsymbol{x}_n)^\top \phi(\boldsymbol{x}_m) = k(\boldsymbol{x}_n, \boldsymbol{x}_m)$

- Prediction for a new test input $\boldsymbol{x}$ will be

$$y = \boldsymbol{w}^\top \phi(\boldsymbol{x}) = \sum_{n=1}^{N} \alpha_n \phi(\boldsymbol{x}_n)^\top \phi(\boldsymbol{x}) = \sum_{n=1}^{N} \alpha_n k(\boldsymbol{x}_n, \boldsymbol{x})$$

- Thus, using the kernel, we effectively learn a nonlinear regression model



- Note: Just as in kernel SVM, prediction cost scales in N

Learning with Kernels: Some Aspects

- Choice of the right kernel is important

Learning with Kernels: Some Aspects

- Choice of the right kernel is important
- Some kernels (e.g., RBF) work well for many problems but hyperparameters of the kernel function may need to be tuned via cross-validation

Learning with Kernels: Some Aspects

- Choice of the right kernel is important
- Some kernels (e.g., RBF) work well for many problems but hyperparameters of the kernel function may need to be tuned via cross-validation
- There is a huge literature on learning the right kernel from data

Learning with Kernels: Some Aspects

- Choice of the right kernel is important
- Some kernels (e.g., RBF) work well for many problems but hyperparameters of the kernel function may need to be tuned via cross-validation
- There is a huge literature on learning the right kernel from data
  - Learning a combination of multiple kernels (Multiple Kernel Learning)

Learning with Kernels: Some Aspects

- Choice of the right kernel is important
- Some kernels (e.g., RBF) work well for many problems but hyperparameters of the kernel function may need to be tuned via cross-validation
- There is a huge literature on learning the right kernel from data
  - Learning a combination of multiple kernels (Multiple Kernel Learning)
  - Bayesian kernel methods (e.g., Gaussian Processes) can learn the kernel hyperparameters from data (thus can be seen as learning the kernel)

Learning with Kernels: Some Aspects

- Choice of the right kernel is important
- Some kernels (e.g., RBF) work well for many problems but hyperparameters of the kernel function may need to be tuned via cross-validation
- There is a huge literature on learning the right kernel from data
  - Learning a combination of multiple kernels (Multiple Kernel Learning)
  - Bayesian kernel methods (e.g., Gaussian Processes) can learn the kernel hyperparameters from data (thus can be seen as learning the kernel)
- Various other alternatives to learn the "right" data representation

Learning with Kernels: Some Aspects

- Choice of the right kernel is important
- Some kernels (e.g., RBF) work well for many problems but hyperparameters of the kernel function may need to be tuned via cross-validation
- There is a huge literature on learning the right kernel from data
  - Learning a combination of multiple kernels (Multiple Kernel Learning)
  - Bayesian kernel methods (e.g., Gaussian Processes) can learn the kernel hyperparameters from data (thus can be seen as learning the kernel)
- Various other alternatives to learn the "right" data representation
  - **Adaptive Basis Functions** (learn basis functions and weights from data)

$$f(\boldsymbol{x}) = \sum_{m=1}^{M} w_m \phi_m(\boldsymbol{x})$$

Learning with Kernels: Some Aspects

- Choice of the right kernel is important
- Some kernels (e.g., RBF) work well for many problems but hyperparameters of the kernel function may need to be tuned via cross-validation
- There is a huge literature on learning the right kernel from data
  - Learning a combination of multiple kernels (Multiple Kernel Learning)
  - Bayesian kernel methods (e.g., Gaussian Processes) can learn the kernel hyperparameters from data (thus can be seen as learning the kernel)
- Various other alternatives to learn the "right" data representation
  - **Adaptive Basis Functions** (learn basis functions and weights from data)

$$f(\boldsymbol{x}) = \sum_{m=1}^{M} w_m \phi_m(\boldsymbol{x})$$

.. various methods can be seen as learning adaptive basis functions, e.g., (Deep) Neural Networks, mixture of experts, Decision Trees, etc.

Learning with Kernels: Some Aspects

- Choice of the right kernel is important
- Some kernels (e.g., RBF) work well for many problems but hyperparameters of the kernel function may need to be tuned via cross-validation
- There is a huge literature on learning the right kernel from data
  - Learning a combination of multiple kernels (Multiple Kernel Learning)
  - Bayesian kernel methods (e.g., Gaussian Processes) can learn the kernel hyperparameters from data (thus can be seen as learning the kernel)
- Various other alternatives to learn the "right" data representation
  - **Adaptive Basis Functions** (learn basis functions and weights from data)

$$f(\boldsymbol{x}) = \sum_{m=1}^{M} w_m \phi_m(\boldsymbol{x})$$

    .. various methods can be seen as learning adaptive basis functions, e.g., (Deep) Neural Networks, mixture of experts, Decision Trees, etc.
  - Kernel methods use a "fixed" set of basis functions or "landmarks". The basis functions are the training data points themselves; also see the next slide.

Kernels: Viewed as Defining Fixed Basis Functions

- Consider each row (or column) of the $N \times N$ kernel matrix (it's symmetric)

$$K = \begin{bmatrix} k(x_1,x_1) & k(x_1,x_2) & \cdots & k(x_1,x_n) \\ k(x_2,x_1) & k(x_2,x_2) & \cdots & k(x_2,x_n) \\ \vdots & \vdots & \ddots & \vdots \\ k(x_n,x_1) & k(x_n,x_2) & \cdots & k(x_n,x_n) \end{bmatrix}$$

Kernels: Viewed as Defining Fixed Basis Functions

- Consider each row (or column) of the $N \times N$ kernel matrix (it's symmetric)

$$\boldsymbol{K} = \begin{bmatrix} k(x_1,x_1) & k(x_1,x_2) & \cdots & k(x_1,x_n) \\ k(x_2,x_1) & k(x_2,x_2) & \cdots & k(x_2,x_n) \\ \vdots & \vdots & \ddots & \vdots \\ k(x_n,x_1) & k(x_n,x_2) & \cdots & k(x_n,x_n) \end{bmatrix}$$

- For each input $\boldsymbol{x}_n$, we can define the following N dimensional vector

$$K(n,:) = [k(\boldsymbol{x}_n, \boldsymbol{x}_1)\ k(\boldsymbol{x}_n, \boldsymbol{x}_2)\ \ldots\ k(\boldsymbol{x}_n, \boldsymbol{x}_N)]$$

Kernels: Viewed as Defining Fixed Basis Functions

- Consider each row (or column) of the $N \times N$ kernel matrix (it's symmetric)

$$
\boldsymbol{K} = \begin{bmatrix} k(x_1,x_1) & k(x_1,x_2) & \cdots & k(x_1,x_n) \\ k(x_2,x_1) & k(x_2,x_2) & \cdots & k(x_2,x_n) \\ \vdots & \vdots & \ddots & \vdots \\ k(x_n,x_1) & k(x_n,x_2) & \cdots & k(x_n,x_n) \end{bmatrix}
$$

- For each input $\boldsymbol{x}_n$, we can define the following N dimensional vector

$$
K(n,:) = [k(\boldsymbol{x}_n, \boldsymbol{x}_1) \; k(\boldsymbol{x}_n, \boldsymbol{x}_2) \; \ldots \; k(\boldsymbol{x}_n, \boldsymbol{x}_N)]
$$

- Can think of this as a new feature vector (with N features) for inputs $\boldsymbol{x}_n$. Each feature represents the similarity of $\boldsymbol{x}_n$ with one of the inputs.

Kernels: Viewed as Defining Fixed Basis Functions

- Consider each row (or column) of the $N \times N$ kernel matrix (it's symmetric)

$$\boldsymbol{K} = \begin{bmatrix} k(x_1,x_1) & k(x_1,x_2) & \cdots & k(x_1,x_n) \\ k(x_2,x_1) & k(x_2,x_2) & \cdots & k(x_2,x_n) \\ \vdots & \vdots & \ddots & \vdots \\ k(x_n,x_1) & k(x_n,x_2) & \cdots & k(x_n,x_n) \end{bmatrix}$$

- For each input $\boldsymbol{x}_n$, we can define the following N dimensional vector

$$K(n,:) = [k(\boldsymbol{x}_n, \boldsymbol{x}_1) \; k(\boldsymbol{x}_n, \boldsymbol{x}_2) \; \ldots \; k(\boldsymbol{x}_n, \boldsymbol{x}_N)]$$

- Can think of this as a new feature vector (with N features) for inputs $\boldsymbol{x}_n$. Each feature represents the similarity of $\boldsymbol{x}_n$ with one of the inputs.

- Thus these new features are basically defined in terms of similarities of each input with a fixed set of basis points or "landmarks" $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N$

Kernels: Viewed as Defining Fixed Basis Functions

- Consider each row (or column) of the $N \times N$ kernel matrix (it's symmetric)

$$\boldsymbol{K} = \begin{bmatrix} k(x_1,x_1) & k(x_1,x_2) & \cdots & k(x_1,x_n) \\ k(x_2,x_1) & k(x_2,x_2) & \cdots & k(x_2,x_n) \\ \vdots & \vdots & \ddots & \vdots \\ k(x_n,x_1) & k(x_n,x_2) & \cdots & k(x_n,x_n) \end{bmatrix}$$

- For each input $\boldsymbol{x}_n$, we can define the following N dimensional vector

$$K(n,:) = [k(\boldsymbol{x}_n, \boldsymbol{x}_1) \; k(\boldsymbol{x}_n, \boldsymbol{x}_2) \; \ldots \; k(\boldsymbol{x}_n, \boldsymbol{x}_N)]$$

- Can think of this as a new feature vector (with N features) for inputs $\boldsymbol{x}_n$. Each feature represents the similarity of $\boldsymbol{x}_n$ with one of the inputs.
- Thus these new features are basically defined in terms of similarities of each input with a fixed set of basis points or "landmarks" $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N$
- In general, the set of basis points or landmarks can be any set of points (not necessarily the data points) and can even be learned (which is what Adaptive Basis Function methods basically do).

Learning with Kernels: Some Aspects (Contd.)

- Storage/computational efficiency can be a bottleneck when using kernels

Learning with Kernels: Some Aspects (Contd.)

- Storage/computational efficiency can be a bottleneck when using kernels
- Training phase usually requires computing and keeping the $N \times N$ kernel matrix $\mathbf{K}$ in memory
  - $O(DN^2)$ to compute $\mathbf{K}$ matrix, $O(N^2)$ space to store

Learning with Kernels: Some Aspects (Contd.)

- Storage/computational efficiency can be a bottleneck when using kernels
- Training phase usually requires computing and keeping the $N \times N$ kernel matrix $\mathbf{K}$ in memory
  - $O(DN^2)$ to compute $\mathbf{K}$ matrix, $O(N^2)$ space to store
- Need to store training data (or at least support vectors in case of SVMs) at test time

Learning with Kernels: Some Aspects (Contd.)

- Storage/computational efficiency can be a bottleneck when using kernels
- Training phase usually requires computing and keeping the $N \times N$ kernel matrix $\mathbf{K}$ in memory
  - $O(DN^2)$ to compute $\mathbf{K}$ matrix, $O(N^2)$ space to store
- Need to store training data (or at least support vectors in case of SVMs) at test time
  - .. just like nearest neighbors methods
- Test time prediction can be slow: need to compute $\sum_{n=1}^{N} \alpha_n k(\boldsymbol{x}_n, \boldsymbol{x})$.
  - Can be made faster if very few α_n's are nonzero (e.g., in SVM)

Learning with Kernels: Some Aspects (Contd.)

- Storage/computational efficiency can be a bottleneck when using kernels
- Training phase usually requires computing and keeping the $N \times N$ kernel matrix $\mathbf{K}$ in memory
  - $O(DN^2)$ to compute $\mathbf{K}$ matrix, $O(N^2)$ space to store
- Need to store training data (or at least support vectors in case of SVMs) at test time
  - .. just like nearest neighbors methods
- Test time prediction can be slow: need to compute $\sum_{n=1}^{N} \alpha_n k(\boldsymbol{x}_n, \boldsymbol{x})$.
  - Can be made faster if very few α_n's are nonzero (e.g., in SVM)
- There is a huge literature on speeding up kernel methods
  - Approximating the kernel matrix using a set of kernel-derived new features
  - Identifying a small set of landmark points in the training data
  - .. and a lot more

Kernels: Concluding Notes

- Kernels give a modular way to learn nonlinear patterns using linear models

Kernels: Concluding Notes

- Kernels give a modular way to learn nonlinear patterns using linear models
  - All you need to do is replace the inner products with the kernel

Kernels: Concluding Notes

- Kernels give a modular way to learn nonlinear patterns using linear models
  - All you need to do is replace the inner products with the kernel
- All the computations remain as efficient as in the original space

Kernels: Concluding Notes

- Kernels give a modular way to learn nonlinear patterns using linear models
  - All you need to do is replace the inner products with the kernel
- All the computations remain as efficient as in the original space
- A very general notion of similarity: Can define similarities between objects even though they can't be represented as vectors. Many kernels are tailor-made for specific types of data

Kernels: Concluding Notes

- Kernels give a modular way to learn nonlinear patterns using linear models
  - All you need to do is replace the inner products with the kernel
- All the computations remain as efficient as in the original space
- A very general notion of similarity: Can define similarities between objects even though they can't be represented as vectors. Many kernels are tailor-made for specific types of data
  - Strings (string kernels): DNA matching, text classification, etc.

Kernels: Concluding Notes

- Kernels give a modular way to learn nonlinear patterns using linear models
  - All you need to do is replace the inner products with the kernel
- All the computations remain as efficient as in the original space
- A very general notion of similarity: Can define similarities between objects even though they can't be represented as vectors. Many kernels are tailor-made for specific types of data
  - Strings (string kernels): DNA matching, text classification, etc.
  - Trees (tree kernels): Comparing parse trees of phrases/sentences