Nonlinear Learning with Kernels

Piyush Rai

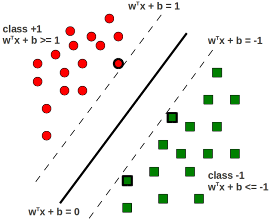
Machine Learning (CS771A)

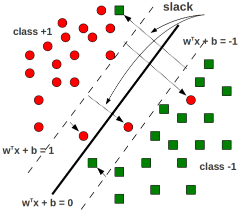
Aug 26, 2016

Recap

Support Vector Machines

- Looked at hard and soft-margin SVMs





- The dual objectives for hard-margin and soft-margin SVM

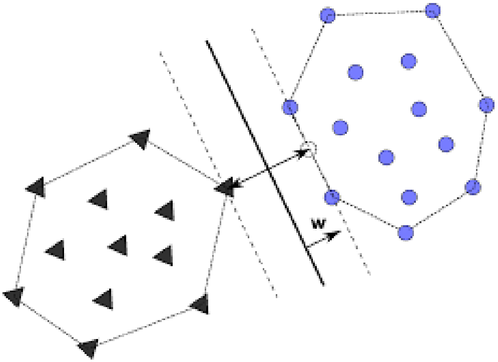
$$\text{Hard-Margin SVM:} \quad \max_{\boldsymbol{\alpha} \geq 0} \; \mathcal{L}_D(\alpha) = \boldsymbol{\alpha}^\top \mathbf{1} - \frac{1}{2}\boldsymbol{\alpha}^\top \mathbf{G}\boldsymbol{\alpha} \qquad \text{s.t.} \quad \sum_{n=1}^{N} \alpha_n y_n = 0$$

$$\text{Soft-Margin SVM:} \quad \max_{\boldsymbol{\alpha} \leq C} \; \mathcal{L}_D(\alpha) = \boldsymbol{\alpha}^\top \mathbf{1} - \frac{1}{2}\boldsymbol{\alpha}^\top \mathbf{G}\boldsymbol{\alpha} \qquad \text{s.t.} \quad \sum_{n=1}^{N} \alpha_n y_n = 0$$

where $\mathbf{G}$ is an $N \times N$ matrix with $G_{mn} = y_m y_n \boldsymbol{x}_m^\top \boldsymbol{x}_n$, and $\mathbf{1}$ is a vector of 1s

SVM Dual Formulation: A Geometric View

- Convex Hull Interpretation†: Solving the SVM dual is equivalent to finding the shortest line connecting the convex hulls of both classes (the SVM's hyperplane will be the perpendicular bisector of this line)



† See: "Duality and Geometry in SVM Classifiers" by Bennett and Bredensteiner

Loss Function Minimization View of SVM

- Recall, we want for each training example: $y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b) \geq 1 - \xi_n$

Loss Function Minimization View of SVM

- Recall, we want for each training example: $y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b) \geq 1 - \xi_n$
- Can think of our loss as basically the sum of the slacks $\xi_n \geq 0$, which is

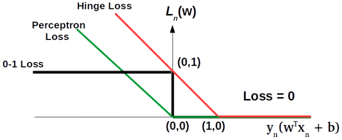
$$\ell(\boldsymbol{w}, b) = \sum_{n=1}^{N} \ell_n(\boldsymbol{w}, b) = \sum_{n=1}^{N} \xi_n = \sum_{n=1}^{N} \max\{0, 1 - y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b)\}$$

Loss Function Minimization View of SVM

- Recall, we want for each training example: $y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b) \geq 1 - \xi_n$
- Can think of our loss as basically the sum of the slacks $\xi_n \geq 0$, which is

$$\ell(\boldsymbol{w}, b) = \sum_{n=1}^{N} \ell_n(\boldsymbol{w}, b) = \sum_{n=1}^{N} \xi_n = \sum_{n=1}^{N} \max\{0, 1 - y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b)\}$$

- This is called **"Hinge Loss"**. Can also learn SVMs by minimizing this loss via stochastic sub-gradient descent (can also add a regularizer on $\boldsymbol{w}$, e.g., ℓ_2)

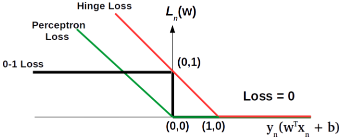


Loss Function Minimization View of SVM

- Recall, we want for each training example: $y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b) \geq 1 - \xi_n$
- Can think of our loss as basically the sum of the slacks $\xi_n \geq 0$, which is

$$\ell(\boldsymbol{w}, b) = \sum_{n=1}^{N} \ell_n(\boldsymbol{w}, b) = \sum_{n=1}^{N} \xi_n = \sum_{n=1}^{N} \max\{0, 1 - y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b)\}$$

- This is called **"Hinge Loss"**. Can also learn SVMs by minimizing this loss via stochastic sub-gradient descent (can also add a regularizer on $\boldsymbol{w}$, e.g., ℓ_2)



- Recall that, Perceptron also minimizes a sort of similar loss function

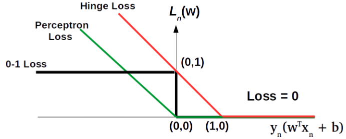
$$\ell(\boldsymbol{w}, b) = \sum_{n=1}^{N} \ell_n(\boldsymbol{w}, b) = \sum_{n=1}^{N} \max\{0, -y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b)\}$$

Loss Function Minimization View of SVM

- Recall, we want for each training example: $y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b) \geq 1 - \xi_n$
- Can think of our loss as basically the sum of the slacks $\xi_n \geq 0$, which is

$$\ell(\boldsymbol{w}, b) = \sum_{n=1}^{N} \ell_n(\boldsymbol{w}, b) = \sum_{n=1}^{N} \xi_n = \sum_{n=1}^{N} \max\{0, 1 - y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b)\}$$

- This is called **"Hinge Loss"**. Can also learn SVMs by minimizing this loss via stochastic sub-gradient descent (can also add a regularizer on $\boldsymbol{w}$, e.g., ℓ_2)



- Recall that, Perceptron also minimizes a sort of similar loss function

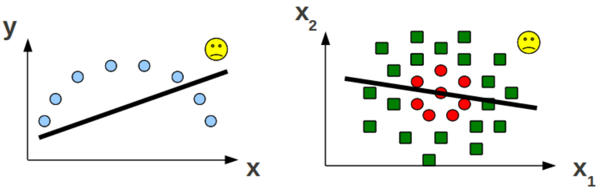
$$\ell(\boldsymbol{w}, b) = \sum_{n=1}^{N} \ell_n(\boldsymbol{w}, b) = \sum_{n=1}^{N} \max\{0, -y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b)\}$$

- Perceptron, SVM, Logistic Reg., all minimize convex approximations of the 0-1 loss (optimizing which is NP-hard; moreover it's non-convex/non-smooth)

Learning with Kernels

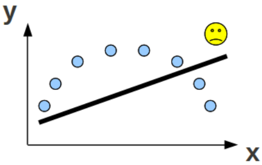
Linear Models

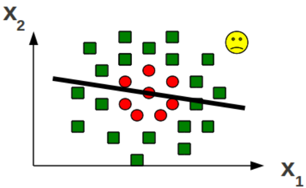
- Linear models (e.g., linear regression, linear SVM) are nice and interpretable but have limitations. Can't learn "difficult" nonlinear patterns.



Linear Models

- Linear models (e.g., linear regression, linear SVM) are nice and interpretable but have limitations. Can't learn "difficult" nonlinear patterns.





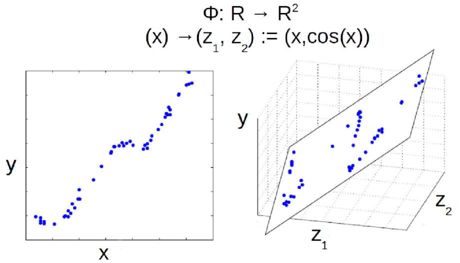
- Reason: Linear models rely on "linear" notions of similarity/distance

$$
\begin{aligned}
Sim(\boldsymbol{x}_n, \boldsymbol{x}_m) &= \boldsymbol{x}_n^\top \boldsymbol{x}_m \\
Dist(\boldsymbol{x}_n, \boldsymbol{x}_m) &= (\boldsymbol{x}_n - \boldsymbol{x}_m)^\top (\boldsymbol{x}_n - \boldsymbol{x}_m)
\end{aligned}
$$

.. which wouldn't work well if the patterns we want to learn are nonlinear

Kernels to the Rescue

- Kernels, using a **feature mapping** ϕ, map data to a new space where the original learning problem becomes "easy" (e.g., a linear model can be applied)

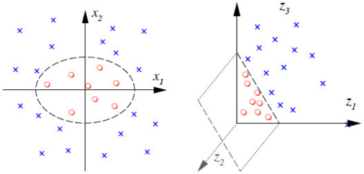


Kernels to the Rescue

- Kernels, using a **feature mapping** ϕ, map data to a new space where the original learning problem becomes "easy" (e.g., a linear model can be applied)

$$\Phi : R^2 \rightarrow R^3$$

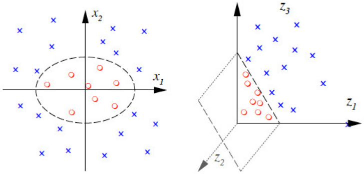
$$(x_1, x_2) \mapsto (z_1, z_2, z_3) := (x_1^2, \sqrt{(2)} x_1 x_2, x_2^2)$$



Kernels to the Rescue

- Kernels, using a **feature mapping** ϕ, map data to a new space where the original learning problem becomes "easy" (e.g., a linear model can be applied)

$$\Phi : R^2 \rightarrow R^3$$
$$(x_1, x_2) \mapsto (z_1, z_2, z_3) := (x_1^2, \sqrt{(2)}x_1x_2, x_2^2)$$

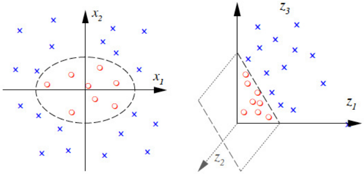


- Aha, nice! But we might have two potential issues here..

Kernels to the Rescue

- Kernels, using a **feature mapping** ϕ, map data to a new space where the original learning problem becomes "easy" (e.g., a linear model can be applied)

$$\Phi : R^2 \rightarrow R^3$$
$$(x_1, x_2) \mapsto (z_1, z_2, z_3) := (x_1^2, \sqrt{(2)}x_1x_2, x_2^2)$$



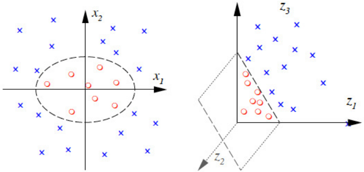
- Aha, nice! But we might have two potential issues here..
  - Constructing these mappings be expensive, especially when the new space is very high dimensional

Kernels to the Rescue

- Kernels, using a **feature mapping** ϕ, map data to a new space where the original learning problem becomes "easy" (e.g., a linear model can be applied)

$$\Phi : R^2 \rightarrow R^3$$

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

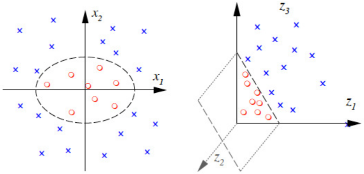


- Aha, nice! But we might have two potential issues here..
  - Constructing these mappings be expensive, especially when the new space is very high dimensional
  - Storing and using these mappings in later computations can be expensive (e.g., we may need to compute innner products in very high dimensional spaces)

Kernels to the Rescue

- Kernels, using a **feature mapping** ϕ, map data to a new space where the original learning problem becomes "easy" (e.g., a linear model can be applied)

$$\Phi : R^2 \rightarrow R^3$$
$$(x_1, x_2) \mapsto (z_1, z_2, z_3) := (x_1^2, \sqrt{(2)}x_1x_2, x_2^2)$$



- Aha, nice! But we might have two potential issues here..
  - Constructing these mappings be expensive, especially when the new space is very high dimensional
  - Storing and using these mappings in later computations can be expensive (e.g., we may need to compute innner products in very high dimensional spaces)
- Kernels side-step these issues by defining an "implicit" feature map

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 $\boldsymbol{z}$ and computes

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

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 $\boldsymbol{z}$ 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 $\boldsymbol{z}$ 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 $\boldsymbol{z}$ 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 $\boldsymbol{z}$ 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 $\boldsymbol{z}$ and computes

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

- The above k implicitly defines a mapping ϕ to a higher dimensional 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 $\boldsymbol{z}$ and computes

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

- The above k implicitly defines a mapping ϕ to a higher dimensional 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 $\boldsymbol{z}$ and computes

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

- The above k implicitly defines a mapping ϕ to a higher dimensional 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 $\boldsymbol{z}$ and computes

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

- The above k implicitly defines a mapping ϕ to a higher dimensional 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 $k(\boldsymbol{x}, \boldsymbol{z})$
- The function k is known as the kernel function

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 $\boldsymbol{z}$ and computes

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

- The above k implicitly defines a mapping ϕ to a higher dimensional 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 $k(\boldsymbol{x}, \boldsymbol{z})$
- The function k is known as the kernel function
- Also, evaluating k is almost as fast as computing inner products

Kernel Functions

- Any kernel function k implicitly defines an associated feature mapping ϕ

Kernel Functions

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

Kernel Functions

- Any kernel function k implicitly defines an associated 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

- Any kernel function k implicitly defines an associated 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

- Any kernel function k implicitly defines an associated 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*
- 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

- Any kernel function k implicitly defines an associated 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*
- 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 $\mathcal{F}$ for which

Kernel Functions

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

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

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

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:
  - For simplicity, let's assume $\boldsymbol{x}$ and $\boldsymbol{z}$ to be scalars (i.e., each example in the original space is defined by a single feature) and also assume $\gamma = 1$

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

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:
  - For simplicity, let's assume $\boldsymbol{x}$ and $\boldsymbol{z}$ to be scalars (i.e., each example in the original space is defined by a single feature) and also assume $\gamma = 1$

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

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

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:

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

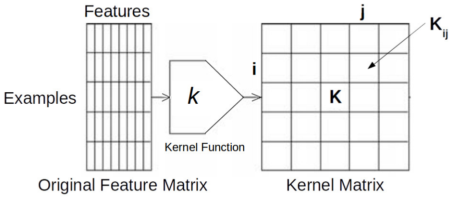
- K_{ij}: Similarity between the i-th and j-th example in the feature space $\mathcal{F}$

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

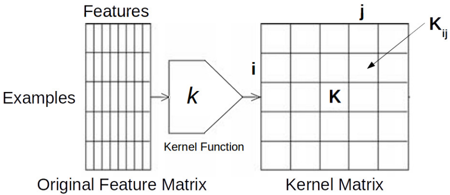


The Kernel Matrix

- 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



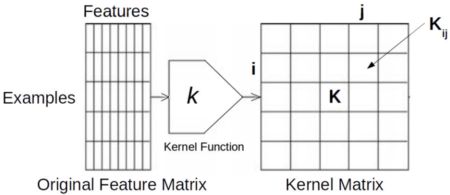
- $\mathbf{K}$ is a symmetric and positive definite matrix

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



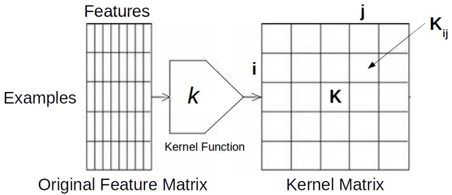
- $\mathbf{K}$ is a symmetric and positive definite matrix
- For a P.D. matrix: $\boldsymbol{z}^\top \mathbf{K} \boldsymbol{z} > 0, \quad \forall \boldsymbol{z} \in \mathbb{R}^N$ (also, all eigenvalues positive)

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.)

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.)
  - 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:

$$\text{Soft-Margin SVM:} \quad \max_{\boldsymbol{\alpha} \leq C} \mathcal{L}_D(\alpha) = \boldsymbol{\alpha}^\top \mathbf{1} - \frac{1}{2}\boldsymbol{\alpha}^\top \mathbf{G}\boldsymbol{\alpha} \quad \text{s.t.} \quad \sum_{n=1}^{N} \alpha_n y_n = 0$$

.. 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:

$$\text{Soft-Margin SVM:} \quad \max_{\boldsymbol{\alpha} \leq C} \mathcal{L}_D(\alpha) = \boldsymbol{\alpha}^\top \mathbf{1} - \frac{1}{2}\boldsymbol{\alpha}^\top \mathbf{G}\boldsymbol{\alpha} \quad \text{s.t.} \quad \sum_{n=1}^{N} \alpha_n y_n = 0$$

.. 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:

$$\text{Soft-Margin SVM:} \quad \max_{\boldsymbol{\alpha} \leq C} \; \mathcal{L}_D(\alpha) = \boldsymbol{\alpha}^\top \mathbf{1} - \frac{1}{2}\boldsymbol{\alpha}^\top \mathbf{G}\boldsymbol{\alpha} \qquad \text{s.t.} \quad \sum_{n=1}^{N} \alpha_n y_n = 0$$

.. 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:

$$\text{Soft-Margin SVM:} \quad \max_{\boldsymbol{\alpha} \leq C} \mathcal{L}_D(\alpha) = \boldsymbol{\alpha}^\top \mathbf{1} - \frac{1}{2}\boldsymbol{\alpha}^\top \mathbf{G}\boldsymbol{\alpha} \quad \text{s.t.} \quad \sum_{n=1}^{N} \alpha_n y_n = 0$$

  .. 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:

$$\text{Soft-Margin SVM:} \quad \max_{\boldsymbol{\alpha} \leq C} \mathcal{L}_D(\alpha) = \boldsymbol{\alpha}^\top \mathbf{1} - \frac{1}{2}\boldsymbol{\alpha}^\top \mathbf{G}\boldsymbol{\alpha} \qquad \text{s.t.} \quad \sum_{n=1}^{N} \alpha_n y_n = 0$$

.. 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 $\mathcal{F}$

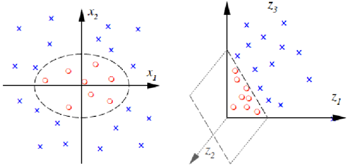
Kernelized SVM Training

- Recall the soft-margin SVM dual problem:

$$\text{Soft-Margin SVM:} \quad \max_{\alpha \leq C} \mathcal{L}_D(\alpha) = \boldsymbol{\alpha}^\top \mathbf{1} - \frac{1}{2}\boldsymbol{\alpha}^\top \mathbf{G}\boldsymbol{\alpha} \qquad \text{s.t.} \quad \sum_{n=1}^{N} \alpha_n y_n = 0$$

.. 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

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

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

Kernelized SVM Prediction

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

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

Kernelized SVM Prediction

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

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

Kernelized SVM Prediction

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

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

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

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

Kernelized SVM Prediction

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

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

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

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

Kernelized SVM Prediction

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

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

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

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

- Note: For the above representation, $\boldsymbol{w}$ can be stored explicitly as a vector only if the feature map $\phi(.)$ of the kernel k can be explicitly written

Kernelized SVM Prediction

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

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

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

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

- Note: For the above representation, $\boldsymbol{w}$ can be stored explicitly as a vector only if the feature map $\phi(.)$ of the kernel k 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

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

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

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

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

- Note: For the above representation, $\boldsymbol{w}$ can be stored explicitly as a vector only if the feature map $\phi(.)$ of the kernel k 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

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

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

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

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

- Note: For the above representation, $\boldsymbol{w}$ can be stored explicitly as a vector only if the feature map $\phi(.)$ of the kernel k 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

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

- 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 are. :-)

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 are. :-)
- Matrix inversion lemma: $(\mathbf{FH}^{-1}\mathbf{G} - \mathbf{E})^{-1}\mathbf{FH}^{-1} = \mathbf{E}^{-1}\mathbf{F}(\mathbf{GE}^{-1}\mathbf{F} - \mathbf{H})^{-1}$

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 are. :-)
- 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 are. :-)
- 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

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

- 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 are. :-)
- 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 $K_{nm} = \boldsymbol{x}_n^\top \boldsymbol{x}_m$

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 are. :-)
- Matrix inversion lemma: $(\mathbf{FH}^{-1}\mathbf{G} - \mathbf{E})^{-1}\mathbf{FH}^{-1} = \mathbf{E}^{-1}\mathbf{F}(\mathbf{GE}^{-1}\mathbf{F} - \mathbf{H})^{-1}$
- 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

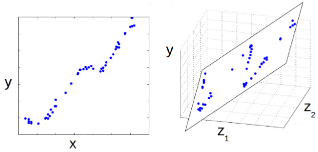
$$\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n \phi(\boldsymbol{x}_n) = \sum_{n=1}^{N} \alpha_n k(\boldsymbol{x}_n, .)$$

  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

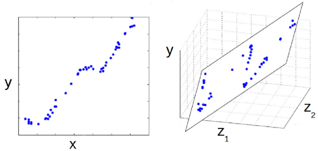
$$\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n \phi(\boldsymbol{x}_n) = \sum_{n=1}^{N} \alpha_n k(\boldsymbol{x}_n, .)$$

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

The Representer Theorem

Theorem

Let $\mathcal{X}$ be some vector space and let $\phi()$ be a map from $\mathcal{X}$ to $\mathcal{F}$, a Reproducing Kernel Hilbert Space[†] *(RKHS) associated with kernel $k(.,.) : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$. Let f be some function defined in this RKHS, let $\ell(.,.)$ be a loss function, and let $R(.)$ be some non-decreasing function. Then the minimizer $\hat{f}$ of the loss function*

$$\sum_{n=1}^{N} \ell(y_n, f(\boldsymbol{x}_n)) + R(||f||^2)$$

can be written as

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

where $\alpha_n \in \mathbb{R}$, $1 \leq n \leq N$

[†] For a short intro, see: “From Zero to Reproducing Kernel Hilbert Spaces in Twelve Pages or” by Hal Daumé

The Representer Theorem

Theorem

Let $\mathcal{X}$ be some vector space and let $\phi()$ be a map from $\mathcal{X}$ to $\mathcal{F}$, a Reproducing Kernel Hilbert Space[†] *(RKHS) associated with kernel $k(.,.) : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$. Let f be some function defined in this RKHS, let $\ell(.,.)$ be a loss function, and let $R(.)$ be some non-decreasing function. Then the minimizer $\hat{f}$ of the loss function*

$$\sum_{n=1}^{N} \ell(y_n, f(\boldsymbol{x}_n)) + R(||f||^2)$$

can be written as

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

where $\alpha_n \in \mathbb{R}$, $1 \leq n \leq N$

Thus the solution (the minimizer of our regularized loss function) lies in the span of the inputs (mapped to $\mathcal{F}$). We already saw this in the case of SVM and kernel ridge regression. This property is however more generally applicable.

† For a short intro, see: "From Zero to Reproducing Kernel Hilbert Spaces in Twelve Pages or" by Hal Daumé

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)

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

- 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
- Need to store the kernel matrix even at test time. This can be an issue (remember that $\mathbf{K}$ is an $N \times N$ matrix which can be very massive)

Learning with Kernels: Some Aspects (Contd.)

- Storage/computational efficiency can be a bottleneck when using kernels
- Need to store the kernel matrix even at test time. This can be an issue (remember that $\mathbf{K}$ is an $N \times N$ matrix which can be very massive)
- Test time prediction can be slow because to make a prediction, we 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.

Learning with Kernels: Some Aspects (Contd.)

- Storage/computational efficiency can be a bottleneck when using kernels
- Need to store the kernel matrix even at test time. This can be an issue (remember that $\mathbf{K}$ is an $N \times N$ matrix which can be very massive)
- Test time prediction can be slow because to make a prediction, we 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.
- 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