Kernel Methods

Sriram Sankararaman

The instructor gratefully acknowledges Fei Sha, Ameet Talwalkar, Eric Eaton, and Jessica Wu whose slides are heavily used, and the many others who made their course material freely available online.

Outline

Supervised learning

We aim to build a function $h(\boldsymbol{x})$ to predict the true value y associated with $\boldsymbol{x}$. If we make a mistake, we incur a *loss*

$$\ell(y, h(\boldsymbol{x}))$$

Supervised learning

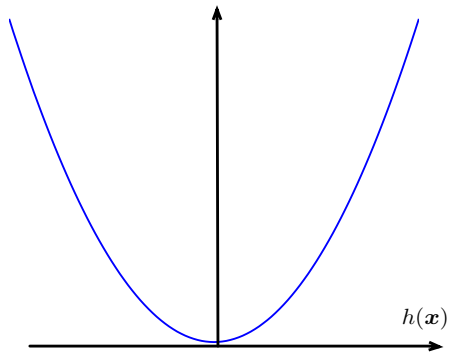
We aim to build a function $h(\boldsymbol{x})$ to predict the true value y associated with $\boldsymbol{x}$. If we make a mistake, we incur a *loss*

$$\ell(y, h(\boldsymbol{x}))$$

Example:squared loss function for regression when y is continuous

$$\ell(y, h(\boldsymbol{x})) = [h(\boldsymbol{x}) - y]^2$$

Ex: when $y = 0$



Other types of loss functions

For classification: $0/1$ loss

$$\ell(y, h(\boldsymbol{x})) = \mathbf{1}\{y \neq h(\boldsymbol{x})\}$$

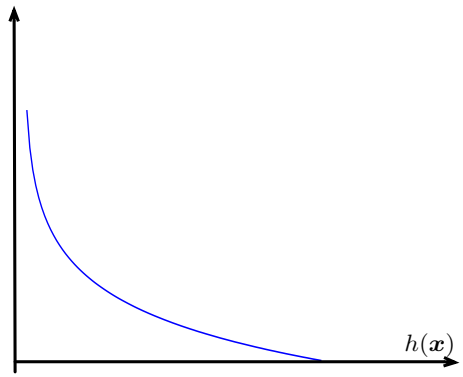
Assumes h outputs values in $\{-1, +1\}$.

Other types of loss functions

For classification: $logistic$ loss

$$\ell(y, h(\boldsymbol{x})) = -y \log h(\boldsymbol{x}) - (1-y) \log[1-h(\boldsymbol{x})]$$

Assumes h outputs values in $[0, 1]$.
Ex: when $y = 1$



Measure how good our hypothesis h is

Risk/ expected test loss: assume we know the true distribution of data $p(\boldsymbol{x}, y)$, the *risk* is

$$\mathcal{R}[h(\boldsymbol{x})] = \sum_{\boldsymbol{x},y} \ell(h(\boldsymbol{x}), y) p(\boldsymbol{x}, y)$$

Measure how good our hypothesis h is

Risk/ expected test loss: assume we know the true distribution of data $p(\boldsymbol{x}, y)$, the *risk* is

$$\mathcal{R}[h(\boldsymbol{x})] = \sum_{\boldsymbol{x},y} \ell(h(\boldsymbol{x}), y) p(\boldsymbol{x}, y)$$

However, we cannot compute $R[h(\boldsymbol{x})]$, so we use *empirical risk/training error*, given a training dataset $\mathcal{D}$

$$\mathcal{R}^{\text{EMP}}[h(\boldsymbol{x})] = \frac{1}{N} \sum_{n} \ell(h(\boldsymbol{x}_n), y_n)$$

Measure how good our hypothesis h is

Risk/ expected test loss: assume we know the true distribution of data $p(\boldsymbol{x}, y)$, the *risk* is

$$\mathcal{R}[h(\boldsymbol{x})] = \sum_{\boldsymbol{x},y} \ell(h(\boldsymbol{x}), y) p(\boldsymbol{x}, y)$$

However, we cannot compute $R[h(\boldsymbol{x})]$, so we use *empirical risk/training error*, given a training dataset $\mathcal{D}$

$$\mathcal{R}^{\text{EMP}}[h(\boldsymbol{x})] = \frac{1}{N} \sum_{n} \ell(h(\boldsymbol{x}_n), y_n)$$

Intuitively, as $N \to +\infty$,

$$\mathcal{R}^{\text{EMP}}[h(\boldsymbol{x})] \to \mathcal{R}[h(\boldsymbol{x})]$$

Pick a good hypothesis h

A good hypothesis h is one that minizes risk/expected test loss $\mathcal{R}[h(\boldsymbol{x})]$ but we cannot even compute it!

Instead pick h that minimizes empirical risk/training error $\mathcal{R}^{\text{EMP}}[h(\boldsymbol{x})]$

This strategy is known as empirical risk minimization.

How this relates to what we have learned?

So far, we have been doing empirical risk minimization (ERM)

- For linear regression, $h_{\boldsymbol{w},b}(\boldsymbol{x}) = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + b$, and we use squared loss
- For logistic regression, $h_{\boldsymbol{w},b}(\boldsymbol{x}) = \sigma(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + b)$, and we use logistic loss
- Finding the best h is achieved by searching for $(\boldsymbol{w}, b)$ that minimizes the trainiing error/empirical risk.

How this relates to what we have learned?

So far, we have been doing empirical risk minimization (ERM)

- For linear regression, $h_{\boldsymbol{w},b}(\boldsymbol{x}) = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + b$, and we use squared loss
- For logistic regression, $h_{\boldsymbol{w},b}(\boldsymbol{x}) = \sigma(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + b)$, and we use logistic loss
- Finding the best h is achieved by searching for $(\boldsymbol{w}, b)$ that minimizes the trainiing error/empirical risk.

ERM might be problematic

- If $h(\boldsymbol{x})$ is complicated enough,

$$\mathcal{R}^{\mathrm{EMP}}[h(\boldsymbol{x})] \to 0$$

- But then $h(\boldsymbol{x})$ is unlikely to do well in predicting things out of the training dataset $\mathcal{D}$
- This is called *poor generalization* or *overfitting*. We have just discussed approaches to address this issue.

Regularizer

Instead of $\mathcal{R}^{\text{emp}}$, use

$$\begin{aligned} &\arg\min_{\boldsymbol{w},b} \mathcal{R}^{\text{EMP}}[h_{\boldsymbol{w},b}(x)] + \lambda R(\boldsymbol{w}, b) \\ &= \arg\min_{\boldsymbol{w},b} \frac{1}{N} \sum_n \ell(y_n, h_{\boldsymbol{w},b}(\boldsymbol{x}_n)) + \lambda R(\boldsymbol{w}, b) \end{aligned} \tag{1}$$

Loss functions ℓ, $\hat{y} = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + b$

- Zero/One: $\ell(y, h(x)) = \mathbf{1}\{y \neq h(x)\}$
- Squared loss: $\ell(y, h(\boldsymbol{x})) = [h(\boldsymbol{x}) - y]^2$
- Logistic loss: $\ell(y, h(\boldsymbol{x})) = -y \log h(\boldsymbol{x}) - (1 - y) \log[1 - h(\boldsymbol{x})]$

Regularizer

Instead of $\mathcal{R}^{\text{emp}}$, use

$$\begin{aligned} &\arg\min_{\boldsymbol{w},b} \mathcal{R}^{\text{EMP}}[h_{\boldsymbol{w},b}(x)] + \lambda R(\boldsymbol{w},b) \\ &= \arg\min_{\boldsymbol{w},b} \frac{1}{N}\sum_n \ell(y_n, h_{\boldsymbol{w},b}(\boldsymbol{x}_n)) + \lambda R(\boldsymbol{w},b) \end{aligned} \tag{1}$$

Regularizer

- Squared 2-norm: $R(\boldsymbol{w},b) = \|\boldsymbol{w}\|_2^2 = \sum_{d=1}^D {w_d}^2$
- 1-norm: $R(\boldsymbol{w},b) = \|\boldsymbol{w}\|_1 = \sum_{d=1}^D |w_d|$.
- 0-norm: $R(\boldsymbol{w},b) = \sum_{d=1}^D \mathbf{1}\{w_d \neq 0\}$.
- p-norm: $R(\boldsymbol{w},b) = \|\boldsymbol{w}\|_p = (\sum_{d=1}^D |w_d|^p)^{\frac{1}{p}}$

Framework for supervised learning

- Pick:
  - Model/hypotheses
  - Loss function
  - Regularizer
  - Algorithm to solve optimization problem

These choices lead to different learning algorithms

Framework for machine learning

- Application
  - Labeled vs unlabeled data
  - Labeled: supervised learning. Type of label: categorical (classification), quantitative (regression)
  - Unlabeled: unsupervised learning.
- Model/hypotheses
- Optimization problem
- Algorithm to solve optimization problem

Outline

Motivation

- Linear models are convenient.
  - Computationally efficient for learning (training) and prediction.
- We would like our models to be "expressive".
  - If it is not expressive enough, it will underfit.
  - Too expressive models can overfit.

Motivation

How to increase the expressive power of linear models?

- Map the feature vector $\boldsymbol{x}$ to an expanded version $\phi(\boldsymbol{x}) : \boldsymbol{x} \in \mathbb{R}^D \rightarrow \boldsymbol{z} \in \mathbb{R}^M$.
- Use a non-linear basis function $\phi(\boldsymbol{x})$ as input to linear model.

Motivation

How to increase the expressive power of linear models?

- Map the feature vector $\boldsymbol{x}$ to an expanded version $\phi(\boldsymbol{x}) : \boldsymbol{x} \in \mathbb{R}^D \rightarrow \boldsymbol{z} \in \mathbb{R}^M$.
- Use a non-linear basis function $\phi(\boldsymbol{x})$ as input to linear model.
- Difficulty: When M is large, computational difficulty.

Motivation

How to choose nonlinear basis function for regression?

$$\boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x})$$

where $\boldsymbol{\phi}(\cdot)$ maps the original feature vector $\boldsymbol{x}$ to a M-dimensional *new* feature vector. In the following, we will show that we can sidestep the issue of choosing which $\boldsymbol{\phi}(\cdot)$ to use — instead, we will choose *equivalently* a *kernel function* that are often computationally efficient to work with.

Example

$$\phi(\boldsymbol{x}) = \begin{pmatrix} 1 \\ \sqrt{2}x_1 \\ \sqrt{2}x_2 \\ \vdots \\ \sqrt{2}x_D \\ {x_1}^2 \\ x_1 x_2 \\ \vdots \\ x_1 x_D \\ x_2 x_1 \\ {x_2}^2 \\ \vdots \\ x_2 x_D \\ \vdots \\ x_D x_1 \\ \vdots \\ {x_D}^2 \end{pmatrix}$$

Example

Computing $\boldsymbol{w}^{\mathrm{T}}\phi(\boldsymbol{x})$ is $\mathrm{O}(D^2)$.

Example

Many learning algorithms can be rewritten to depend on the instances $\boldsymbol{x}_i, \boldsymbol{x}_j$ only through inner products $\phi(\boldsymbol{x}_i)^{\mathrm{T}}\phi(\boldsymbol{x}_j)$.

Why is this helpful ?

$$\phi(\boldsymbol{x}_i)^{\mathrm{T}}\phi(\boldsymbol{x}_j) = (1 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{x}_j)^2$$

Inner product can be computed in $\mathrm{O}(D)$.

Outline

Kernelized ridge regression

Ridge regression

$$J(\boldsymbol{w}) = \sum_n (y_n - \boldsymbol{w}^{\mathrm{T}} \boldsymbol{\phi}(\boldsymbol{x}_n))^2 + \lambda \|\boldsymbol{w}\|_2^2$$

Kernelized ridge regression

Ridge regression

$$J(\boldsymbol{w}) = \sum_n (y_n - \boldsymbol{w}^{\mathrm{T}} \boldsymbol{\phi}(\boldsymbol{x}_n))^2 + \lambda \|\boldsymbol{w}\|_2^2$$

Its solution is given by

$$\nabla J(\boldsymbol{w}) = 2 \sum_n (y_n - \boldsymbol{w}^{\mathrm{T}} \boldsymbol{\phi}(\boldsymbol{x}_n))(-\boldsymbol{\phi}(\boldsymbol{x}_n)) + 2\lambda \boldsymbol{w} = 0$$

Solution

The optimal parameter vector is a linear combination of features

$$\widehat{\boldsymbol{w}} = \sum_n \frac{1}{\lambda}(y_n - \boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n))\boldsymbol{\phi}(\boldsymbol{x}_n) = \sum_n \alpha_n \boldsymbol{\phi}(\boldsymbol{x}_n) = \boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha}$$

where we have designated $\frac{1}{\lambda}(y_n - \boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n))$ as α_n. And the matrix $\boldsymbol{\Phi}$ is the *design matrix* made of *transformed* features.

$$\boldsymbol{\Phi} = \begin{pmatrix} \boldsymbol{\phi}(\boldsymbol{x}_1)^{\mathrm{T}} \\ \boldsymbol{\phi}(\boldsymbol{x}_2)^{\mathrm{T}} \\ \vdots \\ \boldsymbol{\phi}(\boldsymbol{x}_N)^{\mathrm{T}} \end{pmatrix} \in \mathbb{R}^{N \times M}$$

where M is the dimensionality of $\boldsymbol{\phi}(\boldsymbol{x})$.

Solution

The optimal parameter vector is a linear combination of features

$$\widehat{\boldsymbol{w}} = \sum_n \frac{1}{\lambda}(y_n - \boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n))\boldsymbol{\phi}(\boldsymbol{x}_n) = \sum_n \alpha_n \boldsymbol{\phi}(\boldsymbol{x}_n) = \boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha}$$

where we have designated $\frac{1}{\lambda}(y_n - \boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n))$ as α_n. And the matrix $\boldsymbol{\Phi}$ is the *design matrix* made of *transformed* features.

$$\boldsymbol{\Phi} = \begin{pmatrix} \boldsymbol{\phi}(\boldsymbol{x}_1)^{\mathrm{T}} \\ \boldsymbol{\phi}(\boldsymbol{x}_2)^{\mathrm{T}} \\ \vdots \\ \boldsymbol{\phi}(\boldsymbol{x}_N)^{\mathrm{T}} \end{pmatrix} \in \mathbb{R}^{N \times M}$$

where M is the dimensionality of $\boldsymbol{\phi}(\boldsymbol{x})$.

Of course, we do not know what $\boldsymbol{\alpha}$ (the vector of all α_n) corresponds to $\widehat{\boldsymbol{w}}$!

Dual formulation

We substitute $\widehat{\boldsymbol{w}} = \boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha}$ into $J(\boldsymbol{w})$, and obtain the following function of $\boldsymbol{\alpha}$

$$J(\boldsymbol{\alpha}) = \boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha} - 2(\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{y})^{\mathrm{T}}\boldsymbol{\alpha} + \lambda\boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha} + const$$

Dual formulation

We substitute $\widehat{\boldsymbol{w}} = \boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha}$ into $J(\boldsymbol{w})$, and obtain the following function of $\boldsymbol{\alpha}$

$$J(\boldsymbol{\alpha}) = \boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha} - 2(\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{y})^{\mathrm{T}}\boldsymbol{\alpha} + \lambda\boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha} + const$$

Before we show how $J(\boldsymbol{\alpha})$ is derived, we make an important observation. We see repeated structures $\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}$, to which we refer as *Gram matrix* or *kernel matrix*

$$\boldsymbol{K} = \boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}$$

$$= \begin{pmatrix} \boldsymbol{\phi}(\boldsymbol{x}_1)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_1) & \boldsymbol{\phi}(\boldsymbol{x}_1)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_2) & \cdots & \boldsymbol{\phi}(\boldsymbol{x}_1)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_N) \\ \boldsymbol{\phi}(\boldsymbol{x}_2)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_1) & \boldsymbol{\phi}(\boldsymbol{x}_2)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_2) & \cdots & \boldsymbol{\phi}(\boldsymbol{x}_2)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_N) \\ \cdots & \cdots & \cdots & \cdots \\ \boldsymbol{\phi}(\boldsymbol{x}_N)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_1) & \boldsymbol{\phi}(\boldsymbol{x}_N)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_2) & \cdots & \boldsymbol{\phi}(\boldsymbol{x}_N)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_N) \end{pmatrix} \in \mathbb{R}^{N\times N}$$

Properties of the matrix $\boldsymbol{K}$

- Symmetric

$$K_{mn} = \boldsymbol{\phi}(\boldsymbol{x}_m)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n) = \boldsymbol{\phi}(\boldsymbol{x}_n)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_m) = K_{nm}$$

- Positive semidefinite: for any vector $\boldsymbol{a}$

$$\boldsymbol{a}^{\mathrm{T}}\boldsymbol{K}\boldsymbol{a} = (\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{a})^{\mathrm{T}}(\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{a}) \geq 0$$

The derivation of $J(\boldsymbol{\alpha})$

$$
\begin{aligned}
J(\boldsymbol{w}) &= \sum_n (y - \boldsymbol{w}^{\mathrm{T}} \boldsymbol{\phi}(\boldsymbol{x}_n))^2 + \lambda \|\boldsymbol{w}\|_2^2 \\
&= \|\boldsymbol{y} - \boldsymbol{\Phi}\boldsymbol{w}\|_2^2 + \lambda \|\boldsymbol{w}\|_2^2
\end{aligned}
$$

The derivation of $J(\boldsymbol{\alpha})$

$$
\begin{aligned}
J(\boldsymbol{w}) &= \sum_n (y - \boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n))^2 + \lambda \|\boldsymbol{w}\|_2^2 \\
&= \|\boldsymbol{y} - \boldsymbol{\Phi}\boldsymbol{w}\|_2^2 + \lambda \|\boldsymbol{w}\|_2^2 \\
J(\boldsymbol{\alpha}) &= \|\boldsymbol{y} - \boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha}\|_2^2 + \lambda \|\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha}\|_2^2
\end{aligned}
$$

The derivation of $J(\boldsymbol{\alpha})$

$$
\begin{aligned}
J(\boldsymbol{w}) &= \sum_n (y - \boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n))^2 + \lambda \|\boldsymbol{w}\|_2^2 \\
&= \|\boldsymbol{y} - \boldsymbol{\Phi}\boldsymbol{w}\|_2^2 + \lambda \|\boldsymbol{w}\|_2^2 \\
J(\boldsymbol{\alpha}) &= \|\boldsymbol{y} - \boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha}\|_2^2 + \lambda \|\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha}\|_2^2 \\
&= \|\boldsymbol{y} - \boldsymbol{K}\boldsymbol{\alpha}\|_2^2 + \lambda \boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha}
\end{aligned}
$$

The derivation of $J(\boldsymbol{\alpha})$

$$
\begin{aligned}
J(\boldsymbol{w}) &= \sum_n (y - \boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n))^2 + \lambda \|\boldsymbol{w}\|_2^2 \\
&= \|\boldsymbol{y} - \boldsymbol{\Phi}\boldsymbol{w}\|_2^2 + \lambda \|\boldsymbol{w}\|_2^2 \\
J(\boldsymbol{\alpha}) &= \|\boldsymbol{y} - \boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha}\|_2^2 + \lambda \|\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha}\|_2^2 \\
&= \|\boldsymbol{y} - \boldsymbol{K}\boldsymbol{\alpha}\|_2^2 + \lambda \boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha} \\
&= (\boldsymbol{y} - \boldsymbol{K}\boldsymbol{\alpha})^{\mathrm{T}} (\boldsymbol{y} - \boldsymbol{K}\boldsymbol{\alpha}) + \lambda \boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha}
\end{aligned}
$$

The derivation of $J(\boldsymbol{\alpha})$

$$
\begin{aligned}
J(\boldsymbol{w}) &= \sum_n (y - \boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n))^2 + \lambda\|\boldsymbol{w}\|_2^2 \\
&= \|\boldsymbol{y} - \boldsymbol{\Phi}\boldsymbol{w}\|_2^2 + \lambda\|\boldsymbol{w}\|_2^2 \\
J(\boldsymbol{\alpha}) &= \|\boldsymbol{y} - \boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha}\|_2^2 + \lambda\|\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha}\|_2^2 \\
&= \|\boldsymbol{y} - \boldsymbol{K}\boldsymbol{\alpha}\|_2^2 + \lambda\boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha} \\
&= (\boldsymbol{y} - \boldsymbol{K}\boldsymbol{\alpha})^{\mathrm{T}}(\boldsymbol{y} - \boldsymbol{K}\boldsymbol{\alpha}) + \lambda\boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha} \\
&= \boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{K}^{\mathrm{T}}\boldsymbol{K}\boldsymbol{\alpha} - 2\boldsymbol{y}^{\mathrm{T}}\boldsymbol{K}\boldsymbol{\alpha} + \lambda\boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{K}\boldsymbol{\alpha} + const
\end{aligned}
$$

The derivation of $J(\boldsymbol{\alpha})$

$$
\begin{aligned}
J(\boldsymbol{w}) &= \sum_n (y - \boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n))^2 + \lambda \|\boldsymbol{w}\|_2^2 \\
&= \|\boldsymbol{y} - \boldsymbol{\Phi}\boldsymbol{w}\|_2^2 + \lambda \|\boldsymbol{w}\|_2^2 \\
J(\boldsymbol{\alpha}) &= \|\boldsymbol{y} - \boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha}\|_2^2 + \lambda \|\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha}\|_2^2 \\
&= \|\boldsymbol{y} - \boldsymbol{K}\boldsymbol{\alpha}\|_2^2 + \lambda \boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha} \\
&= (\boldsymbol{y} - \boldsymbol{K}\boldsymbol{\alpha})^{\mathrm{T}}(\boldsymbol{y} - \boldsymbol{K}\boldsymbol{\alpha}) + \lambda \boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha} \\
&= \boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{K}^{\mathrm{T}}\boldsymbol{K}\boldsymbol{\alpha} - 2\boldsymbol{y}^{\mathrm{T}}\boldsymbol{K}\boldsymbol{\alpha} + \lambda \boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{K}\boldsymbol{\alpha} + const \\
&= \boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{K}^2\boldsymbol{\alpha} - 2(\boldsymbol{K}\boldsymbol{y})^{\mathrm{T}}\boldsymbol{\alpha} + \lambda \boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{K}\boldsymbol{\alpha} = J(\boldsymbol{\alpha})
\end{aligned}
$$

where we have used the property that $\boldsymbol{K}$ is symmetric.

Optimal $\boldsymbol{\alpha}$

$$\nabla J(\boldsymbol{\alpha}) = 2\boldsymbol{K}^2\boldsymbol{\alpha} - 2\boldsymbol{K}\boldsymbol{y} + 2\lambda\boldsymbol{K}\boldsymbol{\alpha} = 0$$

which leads to (assuming $\boldsymbol{K}$ is invertible).

$$\widehat{\boldsymbol{\alpha}} = (\boldsymbol{K} + \lambda\boldsymbol{I})^{-1}\boldsymbol{y}$$

Note that we only need to know $\boldsymbol{K}$ in order to compute $\widehat{\boldsymbol{\alpha}}$ — the exact form of $\boldsymbol{\phi}(\cdot)$ is not essential — as long as we know how to get inner products $\boldsymbol{\phi}(\boldsymbol{x}_m)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n)$. That observation will give rise to the use of *kernel function*.

Optimal $\boldsymbol{\alpha}$

$$\nabla J(\boldsymbol{\alpha}) = 2\boldsymbol{K}^2\boldsymbol{\alpha} - 2\boldsymbol{K}\boldsymbol{y} + 2\lambda\boldsymbol{K}\boldsymbol{\alpha} = 0$$

which leads to (assuming $\boldsymbol{K}$ is invertible).

$$\widehat{\boldsymbol{\alpha}} = (\boldsymbol{K} + \lambda\boldsymbol{I})^{-1}\boldsymbol{y}$$

Note that we only need to know $\boldsymbol{K}$ in order to compute $\widehat{\boldsymbol{\alpha}}$ — the exact form of $\boldsymbol{\phi}(\cdot)$ is not essential — as long as we know how to get inner products $\boldsymbol{\phi}(\boldsymbol{x}_m)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n)$. That observation will give rise to the use of *kernel function*.

Note that computing the parameter vector does require knowledge of $\boldsymbol{\Phi}$

$$\widehat{\boldsymbol{w}} = \boldsymbol{\Phi}^{\mathrm{T}}\widehat{\boldsymbol{\alpha}} = \boldsymbol{\Phi}^{\mathrm{T}}(\boldsymbol{K} + \lambda\boldsymbol{I})^{-1}\boldsymbol{y}$$

Computing prediction needs only inner products too!

Given a test point $\boldsymbol{\phi}(\boldsymbol{x})$, for prediction, we must compute:

$$\widehat{\boldsymbol{w}}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}) = \boldsymbol{y}^{\mathrm{T}}(\boldsymbol{K} + \lambda I)^{-1}\boldsymbol{\Phi}\boldsymbol{\phi}(\boldsymbol{x})$$

Computing prediction needs only inner products too!

Given a test point $\phi(\boldsymbol{x})$, for prediction, we must compute:

$$\widehat{\boldsymbol{w}}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}) = \boldsymbol{y}^{\mathrm{T}}(\boldsymbol{K}+\lambda I)^{-1}\boldsymbol{\Phi}\boldsymbol{\phi}(\boldsymbol{x})$$

$$= \boldsymbol{y}^{\mathrm{T}}(\boldsymbol{K}+\lambda I)^{-1}\begin{pmatrix} \boldsymbol{\phi}(\boldsymbol{x}_1)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}) \\ \boldsymbol{\phi}(\boldsymbol{x}_2)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}) \\ \vdots \\ \boldsymbol{\phi}(\boldsymbol{x}_N)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}) \end{pmatrix} = \boldsymbol{y}^{\mathrm{T}}(\boldsymbol{K}+\lambda I)^{-1}\boldsymbol{k}_{\boldsymbol{x}}$$

where we have used the property that $(\boldsymbol{K}+\lambda I)^{-1}$ is symmetric (as $\boldsymbol{K}$ is) and use $\boldsymbol{k}_{\boldsymbol{x}}$ as a shorthand notation for the column vector.
Note that, to make a prediction, once again, we *only need to know how to get* $\boldsymbol{\phi}(\boldsymbol{x}_n)^T\boldsymbol{\phi}(\boldsymbol{x})$.

Outline

Inner products between features

Let us examine more closely the inner products $\boldsymbol{\phi}(\boldsymbol{x}_m)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n)$ for a pair of data points $\boldsymbol{x}_m$ and $\boldsymbol{x}_n$.

Polynomial-based nonlinear basis functions consider the following $\boldsymbol{\phi}(\boldsymbol{x})$:

$$\boldsymbol{\phi} : \boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \rightarrow \boldsymbol{\phi}(\boldsymbol{x}) = \begin{pmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{pmatrix}$$

Inner products between features

Let us examine more closely the inner products $\boldsymbol{\phi}(\boldsymbol{x}_m)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n)$ for a pair of data points $\boldsymbol{x}_m$ and $\boldsymbol{x}_n$.

Polynomial-based nonlinear basis functions consider the following $\boldsymbol{\phi}(\boldsymbol{x})$:

$$\boldsymbol{\phi} : \boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \rightarrow \boldsymbol{\phi}(\boldsymbol{x}) = \begin{pmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{pmatrix}$$

This gives rise to an inner product in a special form,

$$\begin{aligned} \boldsymbol{\phi}(\boldsymbol{x}_m)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n) &= x_{m1}^2x_{n1}^2 + 2x_{m1}x_{m2}x_{n1}x_{n2} + x_{m2}^2x_{n2}^2 \\ &= (x_{m1}x_{n1} + x_{m2}x_{n2})^2 = (\boldsymbol{x}_m^{\mathrm{T}}\boldsymbol{x}_n)^2 \end{aligned}$$

Namely, the inner product can be computed by a function $(\boldsymbol{x}_m^{\mathrm{T}}\boldsymbol{x}_n)^2$ defined in terms of the original features, *without computing $\boldsymbol{\phi}(\cdot)$.*

Kernel functions

Some intuition

- To use kernelized ridge regression, we need to be able to compute an inner product between a test point and any training point.
- Since we need this for any possible pair of points, we need a function that takes a pair of points and computes an inner product.
- This is the kernel function $k(\cdot, \cdot)$.
- Given two inputs, $k(\cdot, \cdot)$ tells us how "similar" or "close" these inputs are in the space defined by the function ϕ.

Common kernel functions

Polynomial kernel function with degree of d

$$k(\boldsymbol{x}_m, \boldsymbol{x}_n) = (\boldsymbol{x}_m^{\mathrm{T}} \boldsymbol{x}_n + c)^d$$

for $\boldsymbol{x}_m, \boldsymbol{x}_n \in \mathbb{R}^D$, $c \geq 0$ and d is a positive integer.

Common kernel functions

Polynomial kernel function with degree of d

$$k(\boldsymbol{x}_m, \boldsymbol{x}_n) = (\boldsymbol{x}_m^{\mathrm{T}} \boldsymbol{x}_n + c)^d$$

for $\boldsymbol{x}_m, \boldsymbol{x}_n \in \mathbb{R}^D$, $c \geq 0$ and d is a positive integer.

Gaussian kernel, RBF kernel, or Gaussian RBF kernel

$$k(\boldsymbol{x}_m, \boldsymbol{x}_n) = e^{-\|\boldsymbol{x}_m - \boldsymbol{x}_n\|_2^2 / 2\sigma^2}$$

- Only depends on difference between two inputs
- Corresponds to a feature space with *infinite* dimensions (but we can work directly with the original features)!

Common kernel functions

Polynomial kernel function with degree of d

$$k(\boldsymbol{x}_m, \boldsymbol{x}_n) = (\boldsymbol{x}_m^{\mathrm{T}} \boldsymbol{x}_n + c)^d$$

for $\boldsymbol{x}_m, \boldsymbol{x}_n \in \mathbb{R}^D$, $c \geq 0$ and d is a positive integer.

Gaussian kernel, RBF kernel, or Gaussian RBF kernel

$$k(\boldsymbol{x}_m, \boldsymbol{x}_n) = e^{-\|\boldsymbol{x}_m - \boldsymbol{x}_n\|_2^2 / 2\sigma^2}$$

- Only depends on difference between two inputs
- Corresponds to a feature space with *infinite* dimensions (but we can work directly with the original features)!

These kernels have hyperparameters to be tuned: d, c, σ^2

Kernel functions

Definition: a kernel function $k(\cdot, \cdot)$ is a bivariate function that satisfies the following properties. For any $\boldsymbol{x}_m$ and $\boldsymbol{x}_n$,

$$k(\boldsymbol{x}_m, \boldsymbol{x}_n) = k(\boldsymbol{x}_n, \boldsymbol{x}_m) \text{ and } k(\boldsymbol{x}_m, \boldsymbol{x}_n) = \boldsymbol{\phi}(\boldsymbol{x}_m)^{\mathrm{T}} \boldsymbol{\phi}(\boldsymbol{x}_n)$$

for *some* function $\boldsymbol{\phi}(\cdot)$.

Not very useful though

Kernel functions

Definition: a kernel function $k(\cdot, \cdot)$ is a bivariate function that satisfies the following properties. For any $\boldsymbol{x}_m$ and $\boldsymbol{x}_n$,

$$k(\boldsymbol{x}_m, \boldsymbol{x}_n) = k(\boldsymbol{x}_n, \boldsymbol{x}_m) \text{ and } k(\boldsymbol{x}_m, \boldsymbol{x}_n) = \boldsymbol{\phi}(\boldsymbol{x}_m)^{\mathrm{T}} \boldsymbol{\phi}(\boldsymbol{x}_n)$$

for *some* function $\boldsymbol{\phi}(\cdot)$.

Not very useful though

Examples we have seen

$$k(\boldsymbol{x}_m, \boldsymbol{x}_n) = (\boldsymbol{x}_m^{\mathrm{T}} \boldsymbol{x}_n)^2$$
$$k(\boldsymbol{x}_m, \boldsymbol{x}_n) = e^{-\|\boldsymbol{x}_m - \boldsymbol{x}_n\|_2^2 / 2\sigma^2}$$

Kernel functions

Definition: a kernel function $k(\cdot, \cdot)$ is a bivariate function that satisfies the following properties. For any $\boldsymbol{x}_m$ and $\boldsymbol{x}_n$,

$$k(\boldsymbol{x}_m, \boldsymbol{x}_n) = k(\boldsymbol{x}_n, \boldsymbol{x}_m) \text{ and } k(\boldsymbol{x}_m, \boldsymbol{x}_n) = \boldsymbol{\phi}(\boldsymbol{x}_m)^{\mathrm{T}} \boldsymbol{\phi}(\boldsymbol{x}_n)$$

for *some* function $\boldsymbol{\phi}(\cdot)$.

Not very useful though

Examples we have seen

$$k(\boldsymbol{x}_m, \boldsymbol{x}_n) = (\boldsymbol{x}_m^{\mathrm{T}} \boldsymbol{x}_n)^2$$
$$k(\boldsymbol{x}_m, \boldsymbol{x}_n) = e^{-\|\boldsymbol{x}_m - \boldsymbol{x}_n\|_2^2 / 2\sigma^2}$$

Example that is not a kernel

$$k(\boldsymbol{x}_m, \boldsymbol{x}_n) = \|\boldsymbol{x}_m - \boldsymbol{x}_n\|_2^2$$

(we'll see why later)

Conditions for being a positive semidefinite kernel function

Mercer theorem (loosely), a bivariate function $k(\cdot, \cdot)$ is a kernel function, if and only if, for *any N and any $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, ..., and $\boldsymbol{x}_N$*, the matrix

$$\boldsymbol{K} = \begin{pmatrix} k(\boldsymbol{x}_1, \boldsymbol{x}_1) & k(\boldsymbol{x}_1, \boldsymbol{x}_2) & \cdots & k(\boldsymbol{x}_1, \boldsymbol{x}_N) \\ k(\boldsymbol{x}_2, \boldsymbol{x}_1) & k(\boldsymbol{x}_2, \boldsymbol{x}_2) & \cdots & k(\boldsymbol{x}_2, \boldsymbol{x}_N) \\ \vdots & \vdots & \vdots & \vdots \\ k(\boldsymbol{x}_N, \boldsymbol{x}_1) & k(\boldsymbol{x}_N, \boldsymbol{x}_2) & \cdots & k(\boldsymbol{x}_N, \boldsymbol{x}_N) \end{pmatrix}$$

is positive semidefinite.

Why $\|\boldsymbol{x}_m - \boldsymbol{x}_n\|_2^2$ is not a positive semidefinite kernel?

Use the definition of positive semidefinite kernel function. We choose $N = 2$, and compute the matrix

$$\boldsymbol{K} = \begin{pmatrix} 0 & \|\boldsymbol{x}_1 - \boldsymbol{x}_2\|_2^2 \\ \|\boldsymbol{x}_1 - \boldsymbol{x}_2\|_2^2 & 0 \end{pmatrix}$$

This matrix cannot be positive semidefinite as it has both *negative* and positive eigenvalues.

Recap: why use kernel functions?

Without specifying $\phi(\cdot)$, the kernel matrix

$$\boldsymbol{K} = \begin{pmatrix} k(\boldsymbol{x}_1, \boldsymbol{x}_1) & k(\boldsymbol{x}_1, \boldsymbol{x}_2) & \cdots & k(\boldsymbol{x}_1, \boldsymbol{x}_N) \\ k(\boldsymbol{x}_2, \boldsymbol{x}_1) & k(\boldsymbol{x}_2, \boldsymbol{x}_2) & \cdots & k(\boldsymbol{x}_2, \boldsymbol{x}_N) \\ \vdots & \vdots & \vdots & \vdots \\ k(\boldsymbol{x}_N, \boldsymbol{x}_1) & k(\boldsymbol{x}_N, \boldsymbol{x}_2) & \cdots & k(\boldsymbol{x}_N, \boldsymbol{x}_N) \end{pmatrix}$$

is exactly the same as

$$\boldsymbol{K} = \boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}$$

$$= \begin{pmatrix} \boldsymbol{\phi}(\boldsymbol{x}_1)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_1) & \boldsymbol{\phi}(\boldsymbol{x}_1)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_2) & \cdots & \boldsymbol{\phi}(\boldsymbol{x}_1)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_N) \\ \boldsymbol{\phi}(\boldsymbol{x}_2)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_1) & \boldsymbol{\phi}(\boldsymbol{x}_2)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_2) & \cdots & \boldsymbol{\phi}(\boldsymbol{x}_2)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_N) \\ \cdots & \cdots & \cdots & \cdots \\ \boldsymbol{\phi}(\boldsymbol{x}_N)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_1) & \boldsymbol{\phi}(\boldsymbol{x}_N)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_2) & \cdots & \boldsymbol{\phi}(\boldsymbol{x}_N)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_N) \end{pmatrix}$$

'Kernel trick'

Many learning methods depend on computing *inner products* between features — we have seen the example of regularized least squares. For those methods, we can use a kernel function in the place of the inner products, i.e., *"kernelizing"* the methods, thus, introducing nonlinear features.

We will present one more to illustrate this "trick" by kernelizing the nearest neighbor classifier.

When we talk about support vector machines, we will see the trick one more time.

Outline

Kernelized nearest neighbors

In nearest neighbor classification, the most important quantity to compute is the (squared) distance between two data points $\boldsymbol{x}_m$ and $\boldsymbol{x}_n$

$$d(\boldsymbol{x}_m, \boldsymbol{x}_n) = \|\boldsymbol{x}_m - \boldsymbol{x}_n\|_2^2 = \boldsymbol{x}_m^{\mathrm{T}}\boldsymbol{x}_m + \boldsymbol{x}_n^{\mathrm{T}}\boldsymbol{x}_n - 2\boldsymbol{x}_m^{\mathrm{T}}\boldsymbol{x}_n$$

Kernelized nearest neighbors

In nearest neighbor classification, the most important quantity to compute is the (squared) distance between two data points $\boldsymbol{x}_m$ and $\boldsymbol{x}_n$

$$d(\boldsymbol{x}_m, \boldsymbol{x}_n) = \|\boldsymbol{x}_m - \boldsymbol{x}_n\|_2^2 = \boldsymbol{x}_m^{\mathrm{T}}\boldsymbol{x}_m + \boldsymbol{x}_n^{\mathrm{T}}\boldsymbol{x}_n - 2\boldsymbol{x}_m^{\mathrm{T}}\boldsymbol{x}_n$$

We replace all the inner products in the distance with a kernel function $k(\cdot, \cdot)$, arriving at the kernel distance

$$d^{\mathrm{KERNEL}}(\boldsymbol{x}_m, \boldsymbol{x}_n) = k(\boldsymbol{x}_m, \boldsymbol{x}_m) + k(\boldsymbol{x}_n, \boldsymbol{x}_n) - 2k(\boldsymbol{x}_m, \boldsymbol{x}_n)$$

Kernelized nearest neighbors

In nearest neighbor classification, the most important quantity to compute is the (squared) distance between two data points $\boldsymbol{x}_m$ and $\boldsymbol{x}_n$

$$d(\boldsymbol{x}_m, \boldsymbol{x}_n) = \|\boldsymbol{x}_m - \boldsymbol{x}_n\|_2^2 = \boldsymbol{x}_m^{\mathrm{T}}\boldsymbol{x}_m + \boldsymbol{x}_n^{\mathrm{T}}\boldsymbol{x}_n - 2\boldsymbol{x}_m^{\mathrm{T}}\boldsymbol{x}_n$$

We replace all the inner products in the distance with a kernel function $k(\cdot, \cdot)$, arriving at the kernel distance

$$d^{\mathrm{KERNEL}}(\boldsymbol{x}_m, \boldsymbol{x}_n) = k(\boldsymbol{x}_m, \boldsymbol{x}_m) + k(\boldsymbol{x}_n, \boldsymbol{x}_n) - 2k(\boldsymbol{x}_m, \boldsymbol{x}_n)$$

The distance is equivalent to compute the distance between $\boldsymbol{\phi}(\boldsymbol{x}_m)$ and $\boldsymbol{\phi}(\boldsymbol{x}_n)$

$$d^{\mathrm{KERNEL}}(\boldsymbol{x}_m, \boldsymbol{x}_n) = d(\boldsymbol{\phi}(\boldsymbol{x}_m), \boldsymbol{\phi}(\boldsymbol{x}_n))$$

where the $\boldsymbol{\phi}(\cdot)$ is the nonlinear mapping function implied by the kernel function.

Kernelized nearest neighbors

In nearest neighbor classification, the most important quantity to compute is the (squared) distance between two data points $\boldsymbol{x}_m$ and $\boldsymbol{x}_n$

$$d(\boldsymbol{x}_m, \boldsymbol{x}_n) = \|\boldsymbol{x}_m - \boldsymbol{x}_n\|_2^2 = \boldsymbol{x}_m^{\mathrm{T}}\boldsymbol{x}_m + \boldsymbol{x}_n^{\mathrm{T}}\boldsymbol{x}_n - 2\boldsymbol{x}_m^{\mathrm{T}}\boldsymbol{x}_n$$

We replace all the inner products in the distance with a kernel function $k(\cdot, \cdot)$, arriving at the kernel distance

$$d^{\mathrm{KERNEL}}(\boldsymbol{x}_m, \boldsymbol{x}_n) = k(\boldsymbol{x}_m, \boldsymbol{x}_m) + k(\boldsymbol{x}_n, \boldsymbol{x}_n) - 2k(\boldsymbol{x}_m, \boldsymbol{x}_n)$$

The distance is equivalent to compute the distance between $\boldsymbol{\phi}(\boldsymbol{x}_m)$ and $\boldsymbol{\phi}(\boldsymbol{x}_n)$

$$d^{\mathrm{KERNEL}}(\boldsymbol{x}_m, \boldsymbol{x}_n) = d(\boldsymbol{\phi}(\boldsymbol{x}_m), \boldsymbol{\phi}(\boldsymbol{x}_n))$$

where the $\boldsymbol{\phi}(\cdot)$ is the nonlinear mapping function implied by the kernel function. The nearest neighbor of a point $\boldsymbol{x}$ is thus found with

$$\arg\min_n d^{\mathrm{KERNEL}}(\boldsymbol{x}, \boldsymbol{x}_n)$$

There are infinite numbers of kernels to use!

Rules of composing kernels (this is just a partial list)

- if $k(\boldsymbol{x}_m, \boldsymbol{x}_n)$ is a kernel, then $ck(\boldsymbol{x}_m, \boldsymbol{x}_n)$ is also if $c > 0$.
- if both $k_1(\boldsymbol{x}_m, \boldsymbol{x}_n)$ and $k_2(\boldsymbol{x}_m, \boldsymbol{x}_n)$ are kernels, then $\alpha k_1(\boldsymbol{x}_m, \boldsymbol{x}_n) + \beta k_2(\boldsymbol{x}_m, \boldsymbol{x}_n)$ are also if $\alpha, \beta \geq 0$
- if both $k_1(\boldsymbol{x}_m, \boldsymbol{x}_n)$ and $k_2(\boldsymbol{x}_m, \boldsymbol{x}_n)$ are kernels, then $k_1(\boldsymbol{x}_m, \boldsymbol{x}_n)k_2(\boldsymbol{x}_m, \boldsymbol{x}_n)$ are also.
- if $k(\boldsymbol{x}_m, \boldsymbol{x}_n)$ is a kernel, then $e^{k(\boldsymbol{x}_m, \boldsymbol{x}_n)}$ is also.
- $\ldots$

In practice, choosing an appropriate kernel is an "art"

People typically start with polynomial and Gaussian RBF kernels or incorporate domain knowledge.

Summary

- Kernels allow us to design algorithms that use rich set of features while being computationally efficient.
- Many machine learning algorithms can be "kernelized".
- Picking kernels is an art.
- We still need to tune hyperparameters.