CSCI567 Machine Learning (Fall 2017)

Prof. Fei Sha

U of Southern California

Lecture on Sept. 21, 2017

Outline

Outline

Administrative stuff

- Homework 2 To Be Released on Friday
- Homework 1 Due: please follow the instructions to submit your homework

Outline

Outline

Motivation

How to choose nonlinear basis function for classification and 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.

Motivation

How to choose nonlinear basis function for classification and 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.

- We have seen the neural network approach: it learns features from data together with the classifier or regression function
- Any other approaches?

Kernel methods

In this lecture, we will show that we can sidestep the issue of choosing which $\phi(\cdot)$ to use — instead, we will choose *equivalently* a *kernel function*.

We will motivate our approach by re-examining something we have already known.

Regularized least square

Our objective is to minimize the following regularized residual sum of squares

$$J(\boldsymbol{w}) = \frac{1}{2}\sum_{n}(y_n - \boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n))^2 + \frac{\lambda}{2}\|\boldsymbol{w}\|_2^2$$

Regularized least square

Our objective is to minimize the following regularized residual sum of squares

$$J(\boldsymbol{w}) = \frac{1}{2}\sum_n (y_n - \boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n))^2 + \frac{\lambda}{2}\|\boldsymbol{w}\|_2^2$$

Its solution $\boldsymbol{w}^*$ is given by

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

We can certainly solve $\boldsymbol{w}^*$ now (as before).

Optimal solution to regularized least square (see Lecture 4 too)

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

where the solution is formulated in terms of design matrix $\boldsymbol{\Phi}$ and the target vector $\boldsymbol{y}$

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

Alternative solution

While the previous steps are valid and good, we will take a different route by not rushing into solving it – but instead, examining the structure of the solution

$$\lambda \boldsymbol{w} = (y_n - \boldsymbol{w}^{\mathrm{T}} \boldsymbol{\phi}(\boldsymbol{x}_n))\boldsymbol{\phi}(\boldsymbol{x}_n))$$

Optimal Solution

The optimal parameter vector is a linear combination of features

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

where we have designated $\frac{1}{\lambda}(y_n - \boldsymbol{w}^{*\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n))$ as α_n.
The design matrix $\boldsymbol{\Phi}$ is now transposed, thus is is made of column vectors and is given by

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

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 $\boldsymbol{w}^*$!

Important observation

The optimal parameter vector is a linear combination of features

$$\boldsymbol{w}^* = \sum_n \alpha_n \boldsymbol{x}_n, \text{ or } \quad \boldsymbol{w}^* = \sum_n \alpha_n \boldsymbol{\phi}(\boldsymbol{x}_n)$$

- This is true for regularized linear regression
- This is true for perceptron
- This is true for many other types of algorithms.

But how to find α_n?

Dual formulation

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

$$J(\boldsymbol{\alpha}) = \frac{1}{2}\boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha} - (\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{y})^{\mathrm{T}}\boldsymbol{\alpha} + \frac{\lambda}{2}\boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha}$$

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*

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

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

Examples of kernel matrix

Let us assume we have 3 data points

$$x_1 = -1, x_2 = 0, x_3 = 1$$

And we have the following nonlinear mapping

$$\phi(x) = \begin{pmatrix} 1 \\ x \\ e^x \end{pmatrix}$$

How to compute the kernel matrix?

Calculation of the mapping

$$x_1 \to \boldsymbol{\phi}(x_1) = \begin{pmatrix} 1 \\ -1 \\ e^{-1} \end{pmatrix}, x_2 \to \boldsymbol{\phi}(x_2) = \begin{pmatrix} 1 \\ 0 \\ e^{0} \end{pmatrix}, x_3 \to \boldsymbol{\phi}(x_3) = \begin{pmatrix} 1 \\ 1 \\ e^{1} \end{pmatrix}$$

Kernel matrix

$$\boldsymbol{K} = \begin{pmatrix} K_{11} & K_{12} & K_{13} \\ K_{21} & K_{22} & K_{23} \\ K_{31} & K_{32} & K_{33} \end{pmatrix}$$

where, for example,

$$K_{13} = \boldsymbol{\phi}(x_1)^{\mathrm{T}} \boldsymbol{\phi}(x_3) = 1 \times 1 + (-1) \times 1 + e^{-1} \times e^{1} = 1$$

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 $J(\boldsymbol{\alpha})$

Derivation in the following is supplementary

$$
\begin{aligned}
J(\boldsymbol{w}) &= \frac{1}{2}\sum_n (y_n - \boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n))^2 + \frac{\lambda}{2}\|\boldsymbol{w}\|_2^2 \\
&= \frac{1}{2}\|\boldsymbol{y} - \boldsymbol{\Phi}\boldsymbol{w}\|_2^2 + \frac{\lambda}{2}\|\boldsymbol{w}\|_2^2 \\
&= \frac{1}{2}\|\boldsymbol{y} - \boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha}\|_2^2 + \frac{\lambda}{2}\|\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha}\|_2^2 \\
&= \frac{1}{2}\|\boldsymbol{y} - \boldsymbol{K}\boldsymbol{\alpha}\|_2^2 + \frac{\lambda}{2}\boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha} \\
&= \frac{1}{2}\boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{K}^{\mathrm{T}}\boldsymbol{K}\boldsymbol{\alpha} - \boldsymbol{y}^{\mathrm{T}}\boldsymbol{K}\boldsymbol{\alpha} + \frac{\lambda}{2}\boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{K}\boldsymbol{\alpha} \\
&= \frac{1}{2}\boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{K}^2\boldsymbol{\alpha} - (\boldsymbol{K}\boldsymbol{y})^{\mathrm{T}}\boldsymbol{\alpha} + \frac{\lambda}{2}\boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{K}\boldsymbol{\alpha} = J(\boldsymbol{\alpha})
\end{aligned}
$$

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

Optimal $\boldsymbol{\alpha}$

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

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

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

Optimal $\boldsymbol{\alpha}$

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

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

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

From this, we can compute the parameter vector

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

Compare to the old solution

Previous approach

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

Now

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

Key difference

Kernel matrix $\boldsymbol{K} = \boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}$ is not the same as the second-moment (covariance) matrix $\boldsymbol{C} = \boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}$

1. $\boldsymbol{C}$ has a size of $M \times M$ while $\boldsymbol{K}$ is $N \times N$.
2. When $N \leq D$, using $\boldsymbol{K}$ is more computationally advantageous

Has this been helping?

Computing α^* need only K

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

Computing w^* need to know Φ

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

What is the difference?

To compute $\boldsymbol{K}$, the exact form of $\boldsymbol{\phi}(\cdot)$ is not essential — as long as we know how to get inner products $\boldsymbol{\phi}(\boldsymbol{x}_m)^{\mathrm{T}} \boldsymbol{\phi}(\boldsymbol{x}_n)$.

Now, I am asking you to believe me that indeed we can compute $\boldsymbol{K}$ without knowing what $\boldsymbol{\phi}(\cdot)$ is at all!

This is a "trick" known as *kernel trick*.

In fact, we really do not need to know $\boldsymbol{w}$

Because computing prediction needs only inner products too!

Suppose we need to make a prediction on a new data point $\boldsymbol{x}$, we thus compute

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

In fact, we really do not need to know $\boldsymbol{w}$

Because computing prediction needs only inner products too!

Suppose we need to make a prediction on a new data point $\boldsymbol{x}$, we thus compute

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

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

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* $\boldsymbol{\phi}(\boldsymbol{x}_n)^T\boldsymbol{\phi}(\boldsymbol{x})$.

Summary

We can design algorithms such that we need only the inner products of transformed features between any pair of data points

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

Our next step is to show that in fact, we can get the those inner products by computing

$$K(\boldsymbol{x}_m, \boldsymbol{x}_n)$$

which is a type of special functions over the original feature space of $\boldsymbol{x}$, without even knowing what $\boldsymbol{\phi}(\boldsymbol{x})$ is!

Inner products between features

Due to their central roles, 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 $\boldsymbol{\phi}(\boldsymbol{x})$:

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

Inner products between features

Due to their central roles, 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 $\boldsymbol{\phi}(\boldsymbol{x})$:

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

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 knowing* $\boldsymbol{\phi}(\cdot)$.

A more challenging example

Consider the following mapping, which is paramterized by a parameter

$$\boldsymbol{\psi}_\theta(\boldsymbol{x}) = \begin{pmatrix} \cos(\theta x_1) \\ \sin(\theta x_1) \\ \cos(\theta x_2) \\ \sin(\theta x_2) \end{pmatrix}$$

The inner product for transformed $\boldsymbol{x}_m$ and $\boldsymbol{x}_n$ is thus,

$$\boldsymbol{\psi}_\theta(\boldsymbol{x}_m)^{\mathrm{T}} \boldsymbol{\psi}_\theta(\boldsymbol{x}_n) = \cos(\theta(x_{m1} - x_{n1})) + \cos(\theta(x_{m2} - x_{n2}))$$

Note that, once again, *the inner product can be computed alternatively with the function in the right-hand-side, which depends on the original features only.* We will make it further more interesting.

Concatenating into a long feature vector

Now, consider $(L+1)$ θs, drawn from $[0\ 2\pi]$ evenly, and make another mapping

$$\boldsymbol{\phi}_L(\boldsymbol{x}) = \begin{pmatrix} \boldsymbol{\psi}_0(\boldsymbol{x}) \\ \boldsymbol{\psi}_{\frac{2\pi}{L}}(\boldsymbol{x}) \\ \boldsymbol{\psi}_{2\frac{2\pi}{L}}(\boldsymbol{x}) \\ \vdots \\ \boldsymbol{\psi}_{L\frac{2\pi}{L}}(\boldsymbol{x}) \end{pmatrix}$$

Concatenating into a long feature vector

Now, consider $(L+1)$ θs, drawn from $[0\ 2\pi]$ evenly, and make another mapping

$$\boldsymbol{\phi}_L(\boldsymbol{x}) = \begin{pmatrix} \boldsymbol{\psi}_0(\boldsymbol{x}) \\ \boldsymbol{\psi}_{\frac{2\pi}{L}}(\boldsymbol{x}) \\ \boldsymbol{\psi}_{2\frac{2\pi}{L}}(\boldsymbol{x}) \\ \vdots \\ \boldsymbol{\psi}_{L\frac{2\pi}{L}}(\boldsymbol{x}) \end{pmatrix}$$

What is the inner product?

$$\begin{aligned} \boldsymbol{\phi}_L(\boldsymbol{x}_m)^{\mathrm{T}}\boldsymbol{\phi}_L(\boldsymbol{x}_n) &= \sum_{l=0}^{L} \boldsymbol{\psi}_{l\frac{2\pi}{L}}(\boldsymbol{x}_m)^{\mathrm{T}}\boldsymbol{\psi}_{l\frac{2\pi}{L}}(\boldsymbol{x}_n) \\ &= \sum_{l=0}^{L} \cos(l\frac{2\pi}{L}(x_{m1}-x_{n1})) + \cos(l\frac{2\pi}{L}(x_{m2}-x_{n2})) \end{aligned}$$

What if $L = +\infty$?

Instead of summing up $(L+1)$ terms, we will be integrating

$$\begin{aligned}
\boldsymbol{\phi}_\infty(\boldsymbol{x}_m)^{\mathrm{T}}\boldsymbol{\phi}_\infty(\boldsymbol{x}_n) &= \lim_{L\to+\infty} \boldsymbol{\phi}_L(\boldsymbol{x}_m)^{\mathrm{T}}\boldsymbol{\phi}_L(\boldsymbol{x}_n) \\
&= \int_0^{2\pi} \cos(\theta(x_{m1} - x_{n1})) + \cos(\theta(x_{m2} - x_{n2}))\, d\theta \\
&= 1 - \frac{\sin(2\pi(x_{m1} - x_{n1}))}{x_{m1} - x_{n1}} + 1 - \frac{\sin(2\pi(x_{m2} - x_{n2}))}{x_{m2} - x_{n2}}
\end{aligned}$$

While as before, the right-hand-side depends on only the original features. It actually computes the inner product of two *infinite-dimensional* feature vectors! (Since $L \to +\infty$, the number of $\boldsymbol{\psi}_{l\frac{2\pi}{L}}(\boldsymbol{x})$ is infinite, hence the dimensionality of $\boldsymbol{\phi}_\infty(\boldsymbol{x})$.)

In other words, while we cannot write down every dimension of $\boldsymbol{\phi}_\infty(\boldsymbol{x})$, we can compute its inner product easily using a function defined on the original finite feature space.

Kernel functions

Definition: a (positive semidefinite) 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 $\boldsymbol{\phi}(\cdot)$.

Kernel functions

Definition: a (positive semidefinite) 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 $\boldsymbol{\phi}(\cdot)$.

Examples we have seen

$$k(\boldsymbol{x}_m, \boldsymbol{x}_n) = (\boldsymbol{x}_m^{\mathrm{T}} \boldsymbol{x}_n)^2$$

$$k(\boldsymbol{x}_m, \boldsymbol{x}_n) = 2 - \frac{\sin(2\pi(x_{m1} - x_{n1}))}{x_{m1} - x_{n1}} - \frac{\sin(2\pi(x_{m2} - x_{n2}))}{x_{m2} - x_{n2}}$$

Kernel functions

Definition: a (positive semidefinite) 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 $\boldsymbol{\phi}(\cdot)$.

Examples we have seen

$$k(\boldsymbol{x}_m, \boldsymbol{x}_n) = (\boldsymbol{x}_m^{\mathrm{T}} \boldsymbol{x}_n)^2$$

$$k(\boldsymbol{x}_m, \boldsymbol{x}_n) = 2 - \frac{\sin(2\pi(x_{m1} - x_{n1}))}{x_{m1} - x_{n1}} - \frac{\sin(2\pi(x_{m2} - x_{n2}))}{x_{m2} - x_{n2}}$$

Examples that are not kernels

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

are not our desired kernel function as it cannot be written as inner products between two vectors.

Conditions for being a positive semidefinite kernel function

Mercer theorem (loosely), a bivariate function $k(\cdot, \cdot)$ is a positive semidefinite 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. We also refer $k(\cdot, \cdot)$ as a positive semidefinite kernel.

Flashback: why using 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

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

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

Examples of 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 $c \geq 0$ and d is a positive integer.

Examples of 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 $c \geq 0$ and d is a positive integer.

Gaussian kernel, RBF kernel, or Gaussian RBF kernel

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

Examples of 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 $c \geq 0$ and d is a positive integer.

Gaussian kernel, RBF kernel, or Gaussian RBF kernel

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

Most of those kernels have parameters to be tuned: d, c, σ^2, etc. They are hyper parameters and are often tuned on holdout data or with cross-validation.

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 (the sum of the diagonal elements is called the trace of a matrix, which equals to the sum of the matrix's eigenvalues. In our case, the trace is zero.)

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, using which kernel, or which kernels to compose a new kernel, remains somewhat as "black art", though most people will start with polynomial and Gaussian RBF kernels.

Kernelization 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., *"kernerlizing"* the methods, thus, introducing nonlinear features/basis.

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

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

Kernelized nearest neighbor classifier

In nearest neighbor classifier, 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 neighbor classifier

In nearest neighbor classifier, 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 kerneled 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 neighbor classifier

In nearest neighbor classifier, 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 kerneled 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 neighbor classifier

In nearest neighbor classifier, 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 kerneled 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

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

Take-home exercise

You have seen examples of kernelizing

- linear regression
- nearest neighbor

But can you kernelize the following?

-
- Logistic (or multinomial logistic) regression

You are welcome to Google search the answers after you spend sometime (say about 1 hour or so) on this exercise.