CSCI567 Machine Learning (Spring 2019)

Prof. Victor Adamchik

U of Southern California

Feb. 19, 2019

Administration

Midterm Exam

- time: Feb. 27 (Wednesday) from 5 pm to 8 pm
- location: THH 101 and THH 201
- you can start preparing after this lecture
- practice exam will be posted
- TA's Review: next Tuesday during the lecture time

Midterm Exam

Instructions:

- This is a closed-book exam.
- Questions should be answered concisely.
- Write legibly, avoid cursive writings.
- Partial credit may be available.
- Use only scratch paper provided in the exam packet.

Outline

Outline

Motivation

Recall the question: *how to choose nonlinear basis* $\phi : \mathbb{R}^{\mathsf{D}} \to \mathbb{R}^{\mathsf{M}}$?

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

- neural network is one approach: learn ϕ from data
- **kernel method** is another one: sidestep the issue of choosing ϕ by using *kernel functions*

Case study: regularized linear regression

Recall the regularized least square solution:

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

$$\boldsymbol{X} = \begin{pmatrix} \boldsymbol{x}_1^{\mathrm{T}} \\ \boldsymbol{x}_2^{\mathrm{T}} \\ \vdots \\ \boldsymbol{x}_{\mathsf{N}}^{\mathrm{T}} \end{pmatrix}, \boldsymbol{\Phi} = \begin{pmatrix} \boldsymbol{\phi}(\boldsymbol{x}_1)^{\mathrm{T}} \\ \boldsymbol{\phi}(\boldsymbol{x}_2)^{\mathrm{T}} \\ \vdots \\ \boldsymbol{\phi}(\boldsymbol{x}_{\mathsf{N}})^{\mathrm{T}} \end{pmatrix}$$

Here $\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}$ is D $\times$ D matrix, and $\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}$ is M $\times$ M matrix,

Issue: M *could be huge or even infinity!*

We will rewrite the solution in a different form.

Another solution

Another minimizer is

$$\boldsymbol{w}^* = \boldsymbol{\Phi}^{\mathrm{T}}(\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}} + \lambda\boldsymbol{I})^{-1}\boldsymbol{y}$$
$$\boldsymbol{w}^* = \boldsymbol{\Phi}^{\mathrm{T}}(\boldsymbol{K} + \lambda\boldsymbol{I})^{-1}\boldsymbol{y}$$
$$\boldsymbol{w}^* = \boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha} = \sum_{n=1}^{N}\alpha_n\boldsymbol{\phi}(\boldsymbol{x}_n)$$

where $\boldsymbol{K} = \boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}} \in \mathbb{R}^{\mathsf{N}\times\mathsf{N}}$ is the Gram/Kernel matrix

and $\boldsymbol{\alpha}$ is a new vector.

Solution $\boldsymbol{w}^*$ is a linear combination of features!

Gram matrix

We call $\boldsymbol{K} = \boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}$ **Gram matrix** or **kernel matrix** where the (i, j) entry is

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

Therefore,

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

Another solution

Here we prove that two solutions

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

are the same.

$$\begin{aligned}&(\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi} + \lambda\boldsymbol{I}_1)^{-1}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{y} \\ &= (\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi} + \lambda\boldsymbol{I}_1)^{-1}\boldsymbol{\Phi}^{\mathrm{T}}(\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}} + \lambda\boldsymbol{I}_2)(\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}} + \lambda\boldsymbol{I}_2)^{-1}\boldsymbol{y} \\ &= (\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi} + \lambda\boldsymbol{I}_1)^{-1}(\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}} + \lambda\boldsymbol{\Phi}^{\mathrm{T}})(\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}} + \lambda\boldsymbol{I}_2)^{-1}\boldsymbol{y} \\ &= (\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi} + \lambda\boldsymbol{I}_1)^{-1}(\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi} + \lambda\boldsymbol{I}_1)\boldsymbol{\Phi}^{\mathrm{T}}(\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}} + \lambda\boldsymbol{I}_2)^{-1}\boldsymbol{y} \\ &= \boldsymbol{\Phi}^{\mathrm{T}}(\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}} + \lambda\boldsymbol{I}_2)^{-1}\boldsymbol{y}\end{aligned}$$

Then what is the difference?

First, computing $(\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}} + \lambda\boldsymbol{I})^{-1} = (\boldsymbol{K} + \lambda\boldsymbol{I})^{-1} = \boldsymbol{\alpha}$ can be more efficient than computing $(\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi} + \lambda\boldsymbol{I})^{-1}$ when $\mathsf{N} \leq \mathsf{M}$.

More importantly, computing $(\boldsymbol{K} + \lambda\boldsymbol{I})^{-1}$ *only requires computing inner products in the new feature space!*

Now we can conclude that the exact form of $\boldsymbol{\phi}(\cdot)$ is not essential; *all we need is computing inner products* $\boldsymbol{\phi}(\boldsymbol{x})^T\boldsymbol{\phi}(\boldsymbol{x}')$.

For some $\boldsymbol{\phi}$ it is indeed possible to compute $\boldsymbol{\phi}(\boldsymbol{x})^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}')$ without computing/knowing $\boldsymbol{\phi}$. This is the *kernel trick*.

Why is this helpful?

The prediction of $\boldsymbol{w}^*$ on a new example $\boldsymbol{x}$ is

$$\boldsymbol{w}^{*\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}) = \left(\sum_{n=1}^{N}\alpha_n\boldsymbol{\phi}(\boldsymbol{x}_n)^{\mathrm{T}}\right)\boldsymbol{\phi}(\boldsymbol{x}) = \sum_{n=1}^{N}\alpha_n\left(\boldsymbol{\phi}(\boldsymbol{x}_n)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x})\right)$$

Therefore we do not really need to know a nonlinear mapping ϕ, only inner products in the new feature space matter!

Kernel methods are exactly about computing inner products *without knowing* ϕ.

Examples of kernel matrix

3 data points in $\mathbb{R}$

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

$\boldsymbol{\phi}$ is polynomial basis with degree 4:

$$\boldsymbol{\phi}(x) = \begin{pmatrix} 1 \\ x \\ x^2 \\ x^3 \end{pmatrix}$$

$$\boldsymbol{\phi}(x_1) = \begin{pmatrix} 1 \\ -1 \\ 1 \\ -1 \end{pmatrix} \quad \boldsymbol{\phi}(x_2) = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad \boldsymbol{\phi}(x_3) = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

Calculation of the Gram matrix

$$\boldsymbol{\phi}(x_1) = \begin{pmatrix} 1 \\ -1 \\ 1 \\ -1 \end{pmatrix} \quad \boldsymbol{\phi}(x_2) = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad \boldsymbol{\phi}(x_3) = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

Gram/Kernel matrix

$$\boldsymbol{K} = \begin{pmatrix} \boldsymbol{\phi}(x_1)^{\mathrm{T}}\boldsymbol{\phi}(x_1) & \boldsymbol{\phi}(x_1)^{\mathrm{T}}\boldsymbol{\phi}(x_2) & \boldsymbol{\phi}(x_1)^{\mathrm{T}}\boldsymbol{\phi}(x_3) \\ \boldsymbol{\phi}(x_2)^{\mathrm{T}}\boldsymbol{\phi}(x_1) & \boldsymbol{\phi}(x_2)^{\mathrm{T}}\boldsymbol{\phi}(x_2) & \boldsymbol{\phi}(x_2)^{\mathrm{T}}\boldsymbol{\phi}(x_3) \\ \boldsymbol{\phi}(x_3)^{\mathrm{T}}\boldsymbol{\phi}(x_1) & \boldsymbol{\phi}(x_3)^{\mathrm{T}}\boldsymbol{\phi}(x_2) & \boldsymbol{\phi}(x_3)^{\mathrm{T}}\boldsymbol{\phi}(x_3) \end{pmatrix}$$
$$= \begin{pmatrix} 4 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 4 \end{pmatrix}$$

Example of the kernel trick

Consider the following polynomial basis $\boldsymbol{\phi} : \mathbb{R}^2 \to \mathbb{R}^3$:

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

What is the inner product between $\boldsymbol{\phi}(\boldsymbol{x})$ and $\boldsymbol{\phi}(\boldsymbol{x}')$?

$$\begin{aligned} \boldsymbol{\phi}(\boldsymbol{x})^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}') &= {x_1}^2{x'_1}^2 + 2x_1x_2x'_1x'_2 + {x_2}^2{x'_2}^2 \\ &= (x_1x'_1 + x_2x'_2)^2 = (\boldsymbol{x}^{\mathrm{T}}\boldsymbol{x}')^2 = k(\boldsymbol{x}, \boldsymbol{x}') \end{aligned}$$

Therefore, the inner product in the new space is simply a function of the inner product in the original space.

Count the number of multiplications.

Another example

$\boldsymbol{\phi} : \mathbb{R}^{\mathsf{D}} \to \mathbb{R}^{2\mathsf{D}}$ is parameterized by θ:

$$\boldsymbol{\phi}_\theta(\boldsymbol{x}) = \begin{pmatrix} \cos(\theta x_1) \\ \sin(\theta x_1) \\ \vdots \\ \cos(\theta x_{\mathsf{D}}) \\ \sin(\theta x_{\mathsf{D}}) \end{pmatrix}$$

What is the inner product between $\boldsymbol{\phi}_\theta(\boldsymbol{x})$ and $\boldsymbol{\phi}_\theta(\boldsymbol{x}')$?

$$\begin{aligned} \boldsymbol{\phi}_\theta(\boldsymbol{x})^{\mathrm{T}}\boldsymbol{\phi}_\theta(\boldsymbol{x}') &= \sum_{d=1}^{\mathsf{D}} \cos(\theta x_d)\cos(\theta x'_d) + \sin(\theta x_d)\sin(\theta x'_d) \\ &= \sum_{d=1}^{\mathsf{D}} \cos(\theta(x_d - x'_d)) = k(\boldsymbol{x}, \boldsymbol{x}') \end{aligned}$$

Once again, *the inner product in the new space is a simple function of the features in the original space*.

More complicated example

Based on the previous example mapping $\boldsymbol{\phi}_\theta$, we define a new one $\boldsymbol{\phi}_L : \mathbb{R}^{\mathsf{D}} \to \mathbb{R}^{2\mathsf{D}(L+1)}$ as follows:

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

What is the inner product between $\boldsymbol{\phi}_L(\boldsymbol{x})$ and $\boldsymbol{\phi}_L(\boldsymbol{x}')$?

$$\begin{aligned} \boldsymbol{\phi}_L(\boldsymbol{x})^{\mathrm{T}} \boldsymbol{\phi}_L(\boldsymbol{x}') &= \sum_{\ell=0}^{L} \boldsymbol{\phi}_{\frac{2\pi\ell}{L}}(\boldsymbol{x})^{\mathrm{T}} \boldsymbol{\phi}_{\frac{2\pi\ell}{L}}(\boldsymbol{x}') \\ &= \sum_{\ell=0}^{L} \sum_{d=1}^{\mathsf{D}} \cos\left(\frac{2\pi\ell}{L}(x_d - x'_d)\right) \end{aligned}$$

Infinite dimensional mapping

Let us set $L \to \infty$. This means that $\boldsymbol{\phi}_L(\boldsymbol{x})$ vector has infinite dimension. Clearly we cannot compute $\boldsymbol{\phi}_L(\boldsymbol{x})$, but we can still compute the inner product:

$$\begin{aligned} \boldsymbol{\phi}_\infty(\boldsymbol{x})^{\mathrm{T}} \boldsymbol{\phi}_\infty(\boldsymbol{x}') &= \int_0^{2\pi} \sum_{d=1}^{\mathsf{D}} \cos(\theta(x_d - x'_d))\, d\theta \\ &= \sum_{d=1}^{\mathsf{D}} \frac{\sin(2\pi(x_d - x'_d))}{x_d - x'_d} \end{aligned}$$

Again, a simple function of the original features.

Note that using this mapping in linear regression, we are *learning a weight $\boldsymbol{w}^*$ with infinite dimension!*

Kernel functions

Definition: a function $k : \mathbb{R}^{\mathsf{D}} \times \mathbb{R}^{\mathsf{D}} \to \mathbb{R}$ is called a *(positive semidefinite) kernel function* if there exists a function $\boldsymbol{\phi} : \mathbb{R}^{\mathsf{D}} \to \mathbb{R}^{\mathsf{M}}$ so that for any $\boldsymbol{x}, \boldsymbol{x}' \in \mathbb{R}^{\mathsf{D}}$,

$$k(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{\phi}(\boldsymbol{x})^{\mathrm{T}} \boldsymbol{\phi}(\boldsymbol{x}')$$

Kernel functions are used to quantify similarity between a pair of points $\boldsymbol{x}$ and $\boldsymbol{x}'$.

Examples we have seen

$$\begin{aligned} k(\boldsymbol{x}, \boldsymbol{x}') &= \boldsymbol{x}^{\mathrm{T}} \boldsymbol{x}' \\ k(\boldsymbol{x}, \boldsymbol{x}') &= (\boldsymbol{x}^{\mathrm{T}} \boldsymbol{x}')^2 \\ k(\boldsymbol{x}, \boldsymbol{x}') &= \sum_{d=1}^{\mathsf{D}} \frac{\sin(2\pi(x_d - x'_d))}{x_d - x'_d} \end{aligned}$$

Using kernel functions

Choosing a nonlinear basis $\boldsymbol{\phi}$ becomes choosing a kernel function.

As long as computing the kernel function is more efficient, we should apply the kernel trick.

Gram/kernel matrix becomes:

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

In fact, k is a kernel if and only if $\boldsymbol{K}$ is positive semidefinite for *any $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, ..., $\boldsymbol{x}_N$* (**Mercer theorem**).

Using kernel functions

The prediction on a new example $\boldsymbol{x}$ is

$$\boldsymbol{w}^{*\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}) = \left(\sum_{n=1}^{N}\alpha_n\boldsymbol{\phi}(\boldsymbol{x}_n)^{\mathrm{T}}\right)\boldsymbol{\phi}(\boldsymbol{x}) = \sum_{n=1}^{N}\alpha_n\left(\boldsymbol{\phi}(\boldsymbol{x}_n)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x})\right)$$
$$= \sum_{n=1}^{N}\alpha_n k(\boldsymbol{x}_n, \boldsymbol{x})$$

More examples of kernel functions

Two most commonly used kernel functions in practice:

Polynomial kernel

$$k(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^{\mathrm{T}}\boldsymbol{x}' + c)^d$$

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

Gaussian kernel or Radial basis function (RBF) kernel

$$k(\boldsymbol{x}, \boldsymbol{x}') = e^{-\frac{\|\boldsymbol{x}-\boldsymbol{x}'\|_2^2}{2\sigma^2}}$$

for some $\sigma > 0$.

Think about *what the corresponding ϕ is* for each kernel.

Composing kernels

Creating more kernel functions using the following rules:

If $k_1(\cdot,\cdot)$ and $k_2(\cdot,\cdot)$ are kernels, the followings are kernels too

- **linear combination**: $\alpha k_1(\cdot,\cdot) + \beta k_2(\cdot,\cdot)$ if $\alpha, \beta \geq 0$
- **product**: $k_1(\cdot,\cdot)k_2(\cdot,\cdot)$
- **exponential**: $e^{k(\cdot,\cdot)}$
- ...

Verify using the definition of kernel!

Kernelizing ML algorithms

There are two **main aspects** of kernelized algorithms:

- the solution is expressed as a linear combination of training examples
- algorithm relies only on inner products between data points

Kernel trick is applicable to **many** ML algorithms:

- nearest neighbor classifier
- perceptron
- logistic regression
- ...

Kernelizing NNC

Regular KNN algorithm has two shortcomings.

- all neighbors receive equal weight
- the number of neighbors must be chosen globally.

Kernel addresses these issues.

Instead of selected nearest neighbors, all neighbors are used, but with different weights.

Closer neighbors receive higher weight.

The weighting function is a kernel.

One of the most common way is the Gaussian kernel

$$k(\boldsymbol{x}, \boldsymbol{x}') = e^{-\frac{\|\boldsymbol{x}-\boldsymbol{x}'\|_2^2}{2\sigma^2}}$$

Kernelizing NNC

Kernel Binary Classification Algorithm.

Given

- training data $\mathcal{D} = \{(\boldsymbol{x}_n, y_n), n = 1, 2, \ldots, \mathsf{N}\}$, where $y_n \in \{-1, 1\}$
- kernel function $k(\boldsymbol{x}_i, \boldsymbol{x}_j)$
- input $\boldsymbol{x}$ to classify

Return the class given by

$$\text{sign}\left(\sum_{n=1}^{\mathsf{N}} k(\boldsymbol{x}, \boldsymbol{x_n}) y_n\right)$$

Kernelizing NNC

For NNC with **L2 distance**, $\|\boldsymbol{x} - \boldsymbol{x}'\|_2^2$ is not a valid kernel.

But we can covert the norm

$$d(\boldsymbol{x}, \boldsymbol{x}') = \|\boldsymbol{x} - \boldsymbol{x}'\|_2^2 = \boldsymbol{x}^{\mathrm{T}}\boldsymbol{x} + \boldsymbol{x}'^{\mathrm{T}}\boldsymbol{x}' - 2\boldsymbol{x}^{\mathrm{T}}\boldsymbol{x}'$$

into the following kernel function

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

which by definition is the **L2 distance in a new feature space**

$$d^{\mathrm{KERNEL}}(\boldsymbol{x}, \boldsymbol{x}') = \|\boldsymbol{\phi}(\boldsymbol{x}) - \boldsymbol{\phi}(\boldsymbol{x}')\|_2^2$$

Kernelizing Perceptron

Perceptron Algorithm:

- Pick $(\boldsymbol{x}_n, y_n)$ randomly
- Compute $y^* = \text{sign}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n)$
- If $y^* \neq y_n$ then
  - $\boldsymbol{w} = \boldsymbol{w} + y_n\boldsymbol{x}_n$
- Solution $\boldsymbol{w}$ is a linear combination of features.

- Pick $(\boldsymbol{x}_n, y_n)$ randomly
- Compute $y^* = \text{sign}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n))$
- If $y^* \neq y_n$ then
  - $\boldsymbol{w} = \boldsymbol{w} + y_n\boldsymbol{\phi}(\boldsymbol{x}_n)$
  - $\alpha_n = \alpha_n + y_n$
- Solution $\boldsymbol{w} = \sum_n \alpha_n\boldsymbol{\phi}(\boldsymbol{x}_n)$ is a linear combination of features.

Kernelizing Perceptron

Kernelized Perceptron Algorithm:

- Pick $(\boldsymbol{x}_n, y_n)$ randomly
- Compute $y^* = \text{sign}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n))$
- If $y^* \neq y_n$ then
  - $\boldsymbol{w} = \boldsymbol{w} + y_n\boldsymbol{\phi}(\boldsymbol{x}_n)$
  - $\alpha_n = \alpha_n + y_n$
- Solution $\boldsymbol{w} = \sum_n \alpha_n \boldsymbol{\phi}(\boldsymbol{x}_n)$ is a linear combination of features.

- Pick $(\boldsymbol{x}_n, y_n)$ randomly
- Compute $y^* = \text{sign}\left(\sum_i \alpha_i k(\boldsymbol{x}_i, \boldsymbol{x}_n)\right)$
- If $y^* \neq y_n$ then
  - $\alpha_n = \alpha_n + y_n$
- Solution $\boldsymbol{w} = \sum_n \alpha_n k(\boldsymbol{x}_n, \boldsymbol{x})$.

Kernelizing Perceptron

XOR example

Kernel function: $k(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^{\mathrm{T}}\boldsymbol{x}')^2$

Four 2-dimensional training points:

$$\begin{pmatrix} x_1 & x_2 & y \\ 1 & 1 & 1 \\ -1 & 1 & -1 \\ -1 & -1 & 1 \\ 1 & -1 & -1 \end{pmatrix}$$

Gram matrix $\boldsymbol{K}$:

$$\begin{pmatrix} 4 & 0 & 4 & 0 \\ 0 & 4 & 0 & 4 \\ 4 & 0 & 4 & 0 \\ 0 & 4 & 0 & 4 \end{pmatrix}$$

$k(\boldsymbol{x}_1, \boldsymbol{x}_1) = \left((1,1)^{\mathrm{T}}(1,1)\right)^2 = 4$ $\quad k(\boldsymbol{x}_1, \boldsymbol{x}_2) = \left((1,1)^{\mathrm{T}}(-1,1)\right)^2 = 0$

$k(\boldsymbol{x}_1, \boldsymbol{x}_3) = \left((1,1)^{\mathrm{T}}(-1,-1)\right)^2 = 4$ $\quad k(\boldsymbol{x}_1, \boldsymbol{x}_4) = \left((1,1)^{\mathrm{T}}(1,-1)\right)^2 = 0$

Kernelizing Perceptron

XOR example

Initialization: $\boldsymbol{\alpha} = (0,0,0,0)$.

First round

$\boldsymbol{x}_1$: compute $y^* = \text{sign}\left(\sum_{i=1}^4 \alpha_i k(\boldsymbol{x}_i, \boldsymbol{x}_1)\right) = \text{sign}(0) = -1 \neq y_1$
Thus, $\alpha_1 = y_1 = 1$.

$\boldsymbol{x}_2$: compute $y^* = \text{sign}((1,0,0,0)^{\mathrm{T}}(0,4,0,4)) = \text{sign}(0) = -1 = y_2$

$\boldsymbol{x}_3$: compute $y^* = \text{sign}((1,0,0,0)^{\mathrm{T}}(4,0,4,0)) = \text{sign}(4) = 1 = y_3$

$\boldsymbol{x}_4$: compute $y^* = \text{sign}((1,0,0,0)^{\mathrm{T}}(0,4,0,4)) = \text{sign}(0) = -1 = y_4$

Kernelizing Perceptron

XOR example

$\boldsymbol{\alpha} = (1,0,0,0)$.

Second round.

$\boldsymbol{x}_1$: compute $y^* = \text{sign}((1,0,0,0)^{\mathrm{T}}(4,0,4,0)) = \text{sign}(4) = 1 = y_1$

$\boldsymbol{x}_2$: compute $y^* = \text{sign}((1,0,0,0)^{\mathrm{T}}(0,4,0,4)) = \text{sign}(0) = -1 = y_2$

$\boldsymbol{x}_3$: compute $y^* = \text{sign}((1,0,0,0)^{\mathrm{T}}(4,0,4,0)) = \text{sign}(4) = 1 = y_3$

$\boldsymbol{x}_4$: compute $y^* = \text{sign}((1,0,0,0)^{\mathrm{T}}(0,4,0,4)) = \text{sign}(0) = -1 = y_4$

Converged! The prediction on a new example $\boldsymbol{x}$ is

$$\sum_{i=1}^4 \alpha_i k(\boldsymbol{x}_i, \boldsymbol{x}) = k(\boldsymbol{x}_1, \boldsymbol{x}) = \left((1,1)^{\mathrm{T}}\boldsymbol{x}\right)^2 = (x_1 + x_2)^2$$

Outline

Supervised learning v.s unsupervised learning

There are different types of machine learning problems

- **supervised learning** (what we have discussed by now)
  All data is labeled
  Aim to predict, e.g. classification and regression
- **unsupervised learning**
  All data is unlabeled
  Aim to discover hidden and latent patterns and explore data

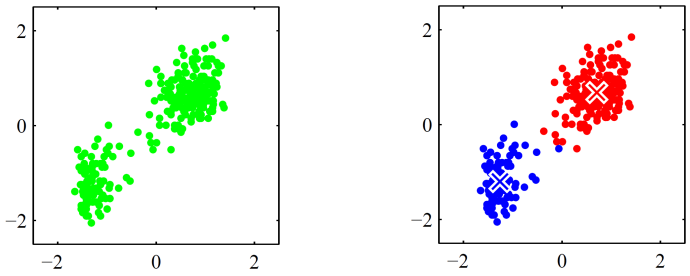
Today's focus: one important unsupervised learning problem: **clustering**

Clustering: informal definition

Given: a set of data points (feature vectors), *without labels*

Output: group the data into some clusters, which means

- assign each point to a specific cluster
- find the center (representative/prototype/...) of each cluster

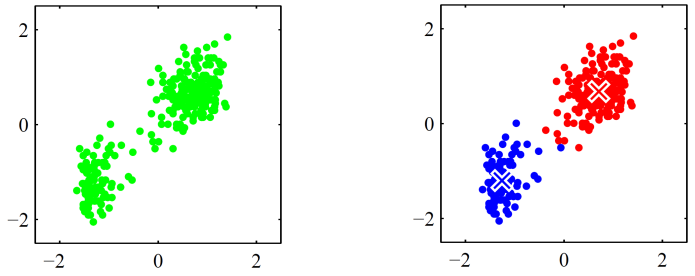


Clustering: formal definition

Given: data points $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N \in \mathbb{R}^{\mathsf{D}}$ and #clusters K we want to find

Output: group the data into K clusters, which means

- find an assignment $\gamma_{nk} \in \{0, 1\}$ s.t. if a data point $n \in [N]$ belongs to a cluster $k \in [K]$ then $\gamma_{nk} = 1$ and $\sum_{k \in [K]} \gamma_{nk} = 1$.
- find the cluster centers $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_K \in \mathbb{R}^{\mathsf{D}}$



Many applications

One example: **image compression** (vector quantization)

- each pixel is a point
- perform clustering over these points
- replace each point by the center of the cluster it belongs to




Original image Large $K \longrightarrow$ Small K

Formal Objective

Key difference from supervised learning problems: no labels given, which means *no ground-truth to even measure the quality of your answer!*

Still, we can turn it into an optimization problem, e.g. through the popular **"K-means" objective**: find γ_{nk} and $\boldsymbol{\mu}_k$ to minimize

$$F\left(\{\gamma_{nk}\}, \{\boldsymbol{\mu}_k\}\right) = \sum_{n=1}^{N} \sum_{k=1}^{K} \gamma_{nk} \|\boldsymbol{x}_n - \boldsymbol{\mu}_k\|_2^2$$

i.e. the **sum of distances of each point to its center**.

Unfortunately, finding the exact minimizer is *NP-hard!*

Alternating minimization

Instead, use a heuristic that alternatively minimizes over $\{\gamma_{nk}\}$ and $\{\boldsymbol{\mu}_k\}$:

Initialize $\{\gamma_{nk}^{(1)}\}$ and $\{\boldsymbol{\mu}_k^{(1)}\}$

For $t = 1, 2, \ldots$

- fix centers $\{\boldsymbol{\mu}_k^{(t)}\}$, find assignments $\{\gamma_{nk}^{(t+1)}\}$

$$\{\gamma_{nk}^{(t+1)}\} = \underset{\{\gamma_{nk}\}}{\operatorname{argmin}} F\left(\{\gamma_{nk}\}, \{\boldsymbol{\mu}_k^{(t)}\}\right)$$

- fix assignments$\{\gamma_{nk}^{(t+1)}\}$, find new centers $\{\boldsymbol{\mu}_k^{(t+1)}\}$

$$\{\boldsymbol{\mu}_k^{(t+1)}\} = \underset{\{\boldsymbol{\mu}_k\}}{\operatorname{argmin}} F\left(\{\gamma_{nk}^{(t+1)}\}, \{\boldsymbol{\mu}_k\}\right)$$

A closer look

The first step (fixed centers, find assignments)

$$\underset{\{\gamma_{nk}\}}{\operatorname{argmin}} F\left(\{\gamma_{nk}\}, \{\boldsymbol{\mu}_k\}\right) = \underset{\{\gamma_{nk}\}}{\operatorname{argmin}} \sum_{n} \sum_{k} \gamma_{nk} \|\boldsymbol{x}_n - \boldsymbol{\mu}_k\|_2^2$$

is simply to **assign each $\boldsymbol{x}_n$ to the closest $\boldsymbol{\mu}_k$**, i.e.

$$\gamma_{nk} = \mathbb{I}\left[k == \underset{c}{\operatorname{argmin}} \|\boldsymbol{x}_n - \boldsymbol{\mu}_c\|_2^2\right]$$

for all $k \in [K]$ and $n \in [N]$.

A closer look

The second step (fixed assignments, find centers)

$$\underset{\{\boldsymbol{\mu}_k\}}{\operatorname{argmin}} F\left(\{\gamma_{nk}\}, \{\boldsymbol{\mu}_k\}\right) = \underset{\{\gamma_{nk}\}}{\operatorname{argmin}} \sum_n \sum_k \gamma_{nk} \|\boldsymbol{x}_n - \boldsymbol{\mu}_k\|_2^2$$

We will do it for each cluster.

The center is simply **an average of the points in that cluster** (hence the name)

$$\boldsymbol{\mu}_k = \frac{\sum_n \gamma_{nk} \boldsymbol{x}_n}{\sum_n \gamma_{nk}}$$

for each $k \in [K]$.

The K-means algorithm, S. Lloyd (1957)

Step 0 Initialization (choose K centers)

Step 1 Fix the centers $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_K$, assign each point to the closest center:

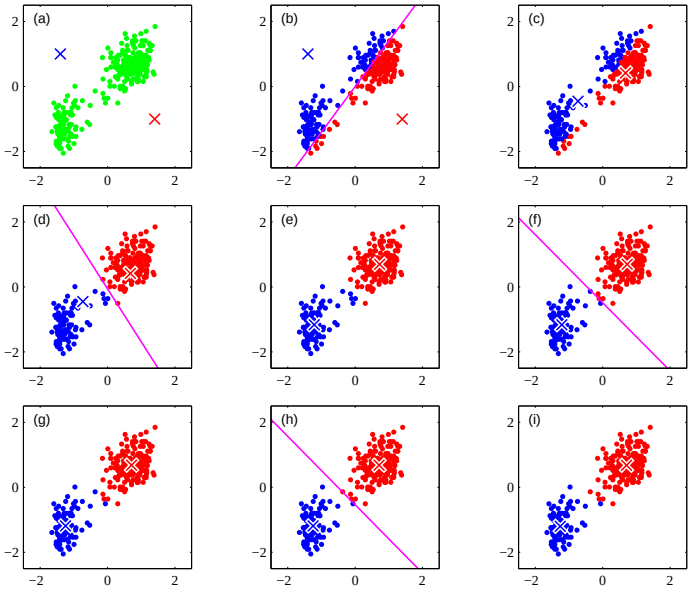
$$\gamma_{nk} = \mathbb{I}\left[k == \underset{c}{\operatorname{argmin}} \|\boldsymbol{x}_n - \boldsymbol{\mu}_c\|_2^2\right]$$

Step 2 Fix the assignment $\{\gamma_{nk}\}$, update the centers

$$\boldsymbol{\mu}_k = \frac{\sum_n \gamma_{nk} \boldsymbol{x}_n}{\sum_n \gamma_{nk}}$$

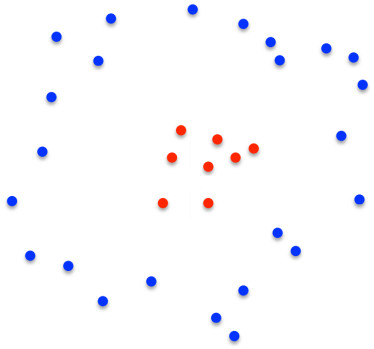
Step 3 Repeat Steps 1 and 2 until the centers no longer change.

An example



K-means algorithm is a heuristic!

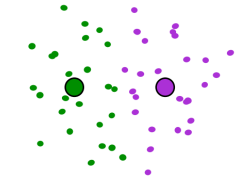
K-means is not always able to properly cluster:



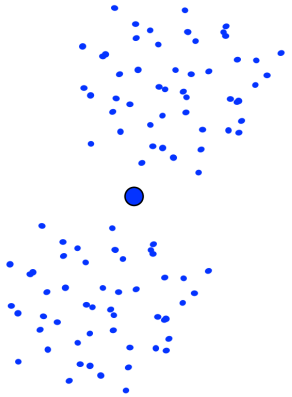
K-means algorithm is a heuristic!

It does matter how you initialize the centers!

In the following example $K = 3$:



Would be better to have one cluster here



… and two clusters here

How to initialize?

A bad selection for the initial centers can lead to a very poor clustering of data.

It also may lead a very long to converge.

There are different ways to initialize:

- randomly pick K points as initial centers
- as it turns out, good initial centers are ones that arent close to each other. (e.g. K-means++, 2007)

How to initialize?

The K-means++ algorithm.

The algorithm selects initial centers that arent close to each other, then uses K-means algorithm for clustering.

The high-level pseudo-code for the K-means++:

- select a data point at random as the first center
- loop K-1 times
  - compute distance squared $d(x)^2$ from each point to the nearest cluster center
  - select a point that has largest probability $\frac{d(x)^2}{\sum_x d(x)^2}$ as the next center

Convergence

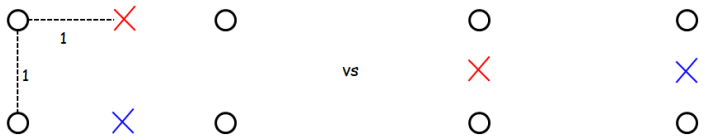
It will converge in a finite number of iterations to a local minimum.

- objective decreases at each step
- objective is lower bounded by 0
- #possible_assignments is finite (K^N, exponentially large though)
- it may take *exponentially* many iterations to converge
- it might not converge to the *global* minimum

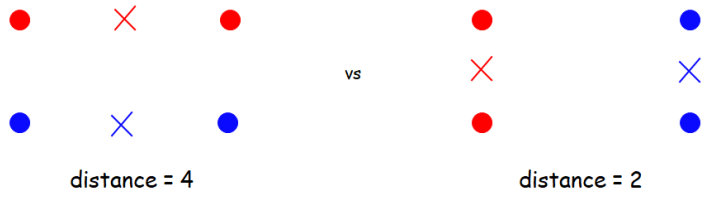
Local minimum v.s global minimum

Simple example: 4 data points, 2 clusters, 2 different initializations.

We initialize the centers by the mean of two points.



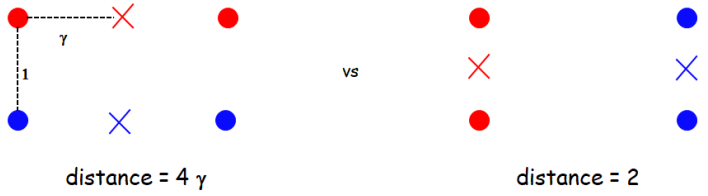
K-means converges immediately in both cases.



Local minimum v.s global minimum

In the left picture we get a local minimum, but in the right - a global minimum!

Moreover, local minimum can be *arbitrarily worse* if we increase the width of this "rectangle" to 2γ.



So, we get stuck at a local minimum.
Initialization matters a lot!

Cluster Quality Measures

We need to define a measure of cluster quality Q and then try different values of K until we get an optimal value for Q

There are different metrics for evaluating clustering algorithms, depending on what types of clusters we want

K-means emphasizes similarity of data within clusters:

$$Q = \sum_{k=1}^{K} \frac{1}{C_k} \sum_{\boldsymbol{x} \in C_k} \|\boldsymbol{x} - \boldsymbol{\mu}_k\|_2^2$$

where C_k is the number of data points in cluster k.

Cluster Quality Measures

Other Quality measures:

The aim is to identify sets of clusters that are compact and at the same time are well separated

- Dunn Index
- Davies-Bouldin Index
- Silhouette Index