Kernel Methods

CS534

Key concepts:

Feature mapping to address non-linear separability

The kernel trick to avoid explicit feature mapping

Definition of Kernel functions

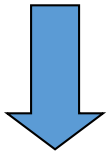
Kernelized perceptron

Kernelized linear regression with L2 regularization

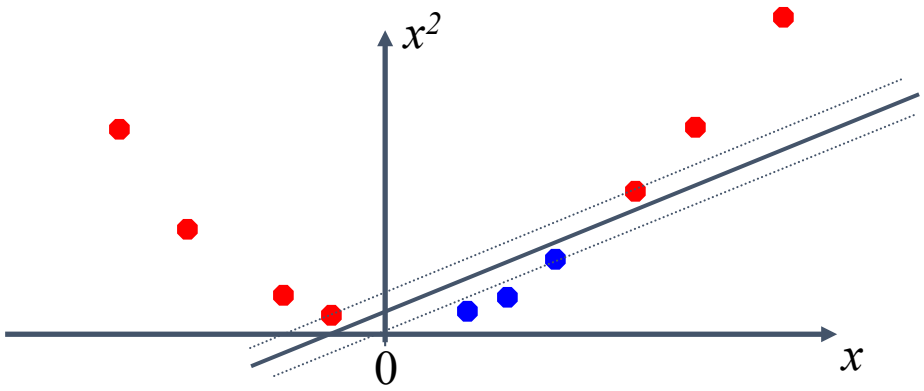
Nonlinearly Separable Data

Original
1-d space

x



Mapped
2-d space

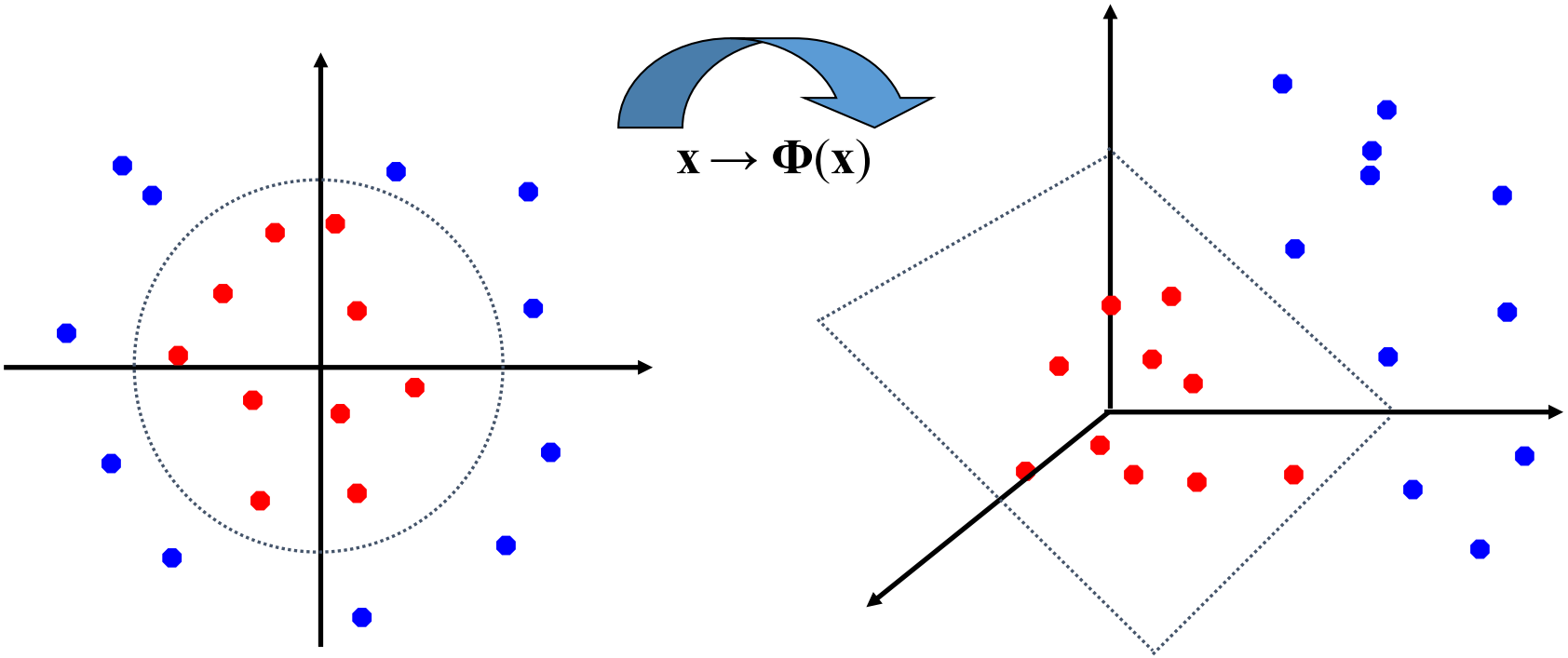


(x, x^2)

Mapping the data to a higher dimensional space can introduce linear separability for data that is not linearly separable in the original input space

Non-linear Classifier via Feature Mapping

- General idea: For <u>any</u> data set, the ***original input space*** can always be mapped to some higher-dimensional **feature spaces** such that the data is linearly separable:



Original input space

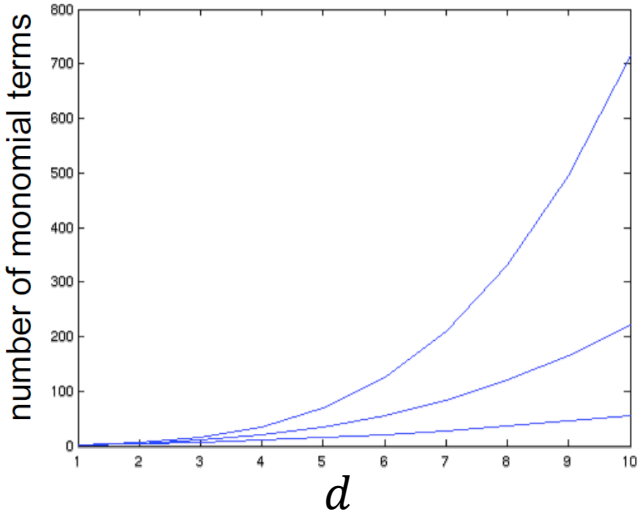
Mapped feature space

Example: Quadratic Feature Space

- Assume d input dimensions $\boldsymbol{x} = (x_1, \cdots, x_d)$
- Number of quadratic terms:

$$1 + d + d + d(d-1)/2 \approx O(d^2)$$

- What if we want to consider even higher order features?
  - For cubic feature space: $O(d^3)$
  - The number of dimensions after mapping increase rapidly with increasing **order $\boldsymbol{m}$**



$$\Phi(x) = \begin{pmatrix} 1 \\ \sqrt{2}x_1 \\ \sqrt{2}x_2 \\ \vdots \\ \sqrt{2}x_d \\ x_1^2 \\ x_2^2 \\ \vdots \\ x_d^2 \\ \sqrt{2}x_1x_2 \\ \sqrt{2}x_1x_3 \\ \vdots \\ \sqrt{2}x_1x_d \\ \vdots \\ \sqrt{2}x_{d-1}x_d \end{pmatrix}$$

Constant term

Linear terms

Pure quadratic terms

Quadratic Cross-terms

Revisiting Perceptron

Given current weight $\mathbf{w}_t$, predict for $\mathbf{x}$ by: $\hat{y}(\mathbf{x}) = sign(\mathbf{w}_t^T \mathbf{x})$

$\mathbf{w}_t$ was created when $\mathbf{w}_{t-1}$ makes a mistake on a training example $(\mathbf{x}, y)$ via:

$$\mathbf{w}_t = \mathbf{w}_{t-1} + y\mathbf{x}$$

Let S_t stores (the indices of) all the previous mistakes leading up to $\mathbf{w}_t$, we have:

$$\mathbf{w}_t = \sum_{i \in S_t} y_i \mathbf{x}_i$$

Note S_t may contain repetitions: if an example was misclassified multiple times, it will appear multiple times.

We can then rewrite the prediction rule as:

$$\hat{y}(\mathbf{x}) = sign(\sum_{i \in S_t} y_i \mathbf{x}_i^T \mathbf{x})$$

Now if data is mapped to a higher dimension by Φ, we simply replace $\mathbf{x}$ with $\Phi(\mathbf{x})$:

$$\hat{y}(\mathbf{x}) = sign\left(\sum_{i \in S_t} y_i \Phi(\mathbf{x}_i)^T \Phi(\mathbf{x})\right)$$

Restructuring the Perceptron algorithm

Let $\mathbf{w} \leftarrow (0,0,\dots,0)$
Repeat if $iter \leq iters$
  for every training example $i = 1,\dots,n$
    $u_i = \mathbf{w}^T \mathbf{x}_i$
    if $y_i u_i \leq 0$ $\mathbf{w} \leftarrow \mathbf{w} + y_i \mathbf{x}_i$
  $iter = iter + 1$

Original Perceptron

New version by Remembering all mistakes in S

$S = [\,]$ //list of mistaken examples
Repeat if $iter \leq iters$
  for every training example $i = 1,\dots,n$
    $u_i = \sum_{j \in S} y_j \mathbf{x}_j^T \mathbf{x}_i$
    if $y_i u_i \leq 0$ add i to S
  $iter = iter + 1$

If we wish to use mapped feature space, simply replace $\mathbf{x}_j^T \mathbf{x}_i$ with $\Phi(\mathbf{x}_j)^T \Phi(\mathbf{x}_i)$

Dot product in the Quadratic Feature Space

Explicit mapping takes $O(d^2)$ time. Consider two d-dimensional vectors $\mathbf{a}$ and $\mathbf{b}$:

$$\Phi(\boldsymbol{a})^T\Phi(\boldsymbol{b}) = 1 + 2\sum_{i=1}^{d} a_i b_i + \sum_{i=1}^{d} a_i^2 b_i^2 + \sum_{i=1}^{d}\sum_{j=i+1}^{d} 2a_i a_j b_i b_j$$

$\Phi(\boldsymbol{a})^T\Phi(\boldsymbol{b})=$

$$\begin{pmatrix} 1 \\ \sqrt{2}a_1 \\ \sqrt{2}a_2 \\ \vdots \\ \sqrt{2}a_d \\ a_1^2 \\ a_2^2 \\ \vdots \\ a_d^2 \\ \sqrt{2}a_1a_2 \\ \sqrt{2}a_1a_3 \\ \vdots \\ \sqrt{2}a_1a_d \\ \vdots \\ \sqrt{2}a_{d-1}a_d \end{pmatrix} \cdot \begin{pmatrix} 1 \\ \sqrt{2}b_1 \\ \sqrt{2}b_2 \\ \vdots \\ \sqrt{2}b_d \\ b_1^2 \\ b_2^2 \\ \vdots \\ b_d^2 \\ \sqrt{2}b_1b_2 \\ \sqrt{2}b_1b_3 \\ \vdots \\ \sqrt{2}b_1b_d \\ \vdots \\ \sqrt{2}b_{d-1}b_d \end{pmatrix}$$

Consider the following function:

$$(\mathbf{a}^T\mathbf{b}+1)^2 = (\mathbf{a}^T\mathbf{b})^2 + 2(\mathbf{a}^T\mathbf{b}) + 1$$

$$= \left(\sum_{i=1}^{d} a_i b_i\right)^2 + 2\sum_{i=1}^{d} a_i b_i + 1$$

$$= \sum_{i=1}^{d}\sum_{j=1}^{d} a_i a_j b_i b_j + 2\sum_{i=1}^{d} a_i b_i + 1$$

$$= \sum_{i=1}^{d} a_i^2 b_i^2 + 2\sum_{i=1}^{d}\sum_{j=i+1}^{d} a_i a_j b_i b_j + 2\sum_{i=1}^{d} a_i b_i + 1$$

$\kappa(\mathbf{a},\mathbf{b}) = (\mathbf{a}^T\mathbf{b}+1)^2$ computes the dot product in quadratic space in $O(d)$ time

The kernel trick

- **Definition:** A function $\kappa(\mathbf{x}, \mathbf{x}')$ is called a ***kernel function*** if $\kappa(\mathbf{x}, \mathbf{x}') =< \phi(\mathbf{x}), \phi(\mathbf{x}') >$ for some mapping function ϕ
- **Implication**: we can simply replace any occurrences of dot product $< \mathbf{x} \cdot \mathbf{x}' >$ with a kernel function κ that computes $\kappa(\mathbf{x}, \mathbf{x}') =< \phi(\mathbf{x}), \phi(\mathbf{x}') >$
- **Implication**: we do not need to explicitly compute the mapping of the features --- significant computational savings
- For example, to compute the dot product in the quadratic space:
  - With the quadratic kernel function: $O(d)$
  - With explicit mapping: $O(d^2)$

Kernelizing Perceptron

$S = [\,]$ //list of mistaken examples
Repeat if $iter \leq iters$
  for every training example $i = 1, \ldots, n$
    $u_i = \sum_{j \in S} y_j \mathbf{x}_j^T \mathbf{x}_i$
    if $y_i u_i \leq 0$ add i to S
  $iter = iter + 1$

Original Perceptron (restructured)

Replacing $\mathbf{x}_j^T \mathbf{x}_i$ with $\kappa(\mathbf{x}_j, \mathbf{x}_i)$

$S = [\,]$ //list of mistaken examples
Repeat if $iter \leq iters$
  for every training example $i = 1, \ldots, n$
    $u_i = \sum_{j \in S} y_j \kappa(\mathbf{x}_j, \mathbf{x}_i)$
    if $y_i u_i \leq 0$ add i to S
  $iter = iter + 1$

Making it better

$S = [\,]$ //list of mistaken examples

Repeat if $iter \leq iters$

for every training example $i = 1, \ldots, n$

$$u_i = \sum_{j \in S} y_j \kappa(\mathbf{x}_j, \mathbf{x}_i)$$

if $y_i u_i \leq 0$ add i to S

$iter = iter + 1$

Remember all mistakes in S

Keep a counter for each example

Let $\alpha_i = 0 \;\; \forall i = 1, \ldots, n$

Repeat if $iter \leq iters$

for every training example $i = 1, \ldots, n$

$$u_i = \sum_{j=1 \text{ to } n} \alpha_j y_j \kappa(\mathbf{x}_j, \mathbf{x}_i)$$

if $y_i u_i \leq 0$ $\alpha_i \leftarrow \alpha_i + 1$

$iter = iter + 1$

Kernel functions

- A kernel function can be intuitively viewed as computing some similarity measure between examples
- In practice, we directly use the kernel functions without explicitly stating the transformation Φ
- Given a kernel function, finding its corresponding transformation can be very cumbersome or impossible
  - RBF kernel's mapped space has infinite dimensions
- Not all functions are kernels
  - For some functions there does not exist a corresponding mapping $\Phi(\mathbf{x})$
- If you have a good similarity measure, can we use it as a kernel?

Kernel Function or Not

- Consider a finite set of m points, we define the kernel (Gram) marix as

$$K = \begin{bmatrix} \kappa(\mathbf{x}_1, \mathbf{x}_1) & \kappa(\mathbf{x}_1, \mathbf{x}_2) & \dots & \kappa(\mathbf{x}_1, \mathbf{x}_n) \\ \kappa(\mathbf{x}_2, \mathbf{x}_1) & \kappa(\mathbf{x}_2, \mathbf{x}_2) & \dots & \kappa(\mathbf{x}_2, \mathbf{x}_n) \\ \dots & \dots & \dots & \dots \\ \kappa(\mathbf{x}_n, \mathbf{x}_1) & \kappa(\mathbf{x}_n, \mathbf{x}_2) & \dots & \kappa(\mathbf{x}_n, \mathbf{x}_n) \end{bmatrix}_{n \times n}$$

- Kernel matrices by definition are square and symmetric

Mercer theorem:

A function κ is a kernel function if and only if for any finite sample $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n$, its corresponding kernel matrix is positive semi-definite (i.e., only has non-negative eigenvalues)

Some common kernel functions

- Linear kernel: $\kappa(a, b) = (a \cdot b)$ (ϕ is identity mapping)
- Polynomial of degree d (without lower order terms): $\kappa(a, b) = (a \cdot b)^d$
- Polynomial of degree up to d(with lower order terms): $\kappa(a, b) = (a \cdot b + 1)^d$
- Radial-Basis-Function (RBF) or Gaussian kernel:

$$\kappa(a, b) = \exp\left(-\frac{(a-b)^2}{2\sigma^2}\right)$$

Closure Property of Kernels

If κ_1 and κ_2 are kernel functions, then the following are all kernel functions:

- $\kappa(x, y) = \kappa_1(x, y) + \kappa_2(x, y)$
  - $\Phi = \text{concatenation of } \Phi_1 \text{and } \Phi_2$
- $\kappa(x, y) = a\kappa_1(x, y)$, where $a > 0$
  - $\Phi = \sqrt{a}\Phi_1$
- $\kappa(x, y) = \kappa_1(x, y)\kappa_2(x, y)$
  - If Φ_1 has N_1 features and Φ_2 has N_2 features
  - Φ will have $N_1 \times N_2$ features: $\Phi_{ij} = \Phi_{1i} \cdot \Phi_{2j}$

One can repeatedly apply the closure property to compose kernel functions, e.g., $a\kappa_1 + b\kappa_2 + \kappa_1\kappa_2$ is a kernel function for $a, b > 0$

Key Choices in Applying Kernel

- Selecting the kernel function
  - (cross-) validation to rescue
  - Popular choices: Linear kernel, polynomial kernels (with low degrees)
  - Start simple, go more complex only if needed.
  - Can learn to combine different kernels (kernel learning)
    - $K = \alpha_1 K_1 + \alpha_2 K_2 + \cdots + \alpha_k K_k,$ s.t. $\alpha_i \geq 0$ for $i = 1, \ldots, k$
  - Kernel functions are defined for many non-traditional non-Euclidean data, e.g., graph kernel, set kernel, string kernel etc.
- Selecting the kernel parameter
  - can have strong impact on performance
  - the optimal range can be reasonably large
  - grid search with (cross-)validation is commonly used

Revisiting Linear Regression

- Consider the L2 regularized linear regression

$$\max_{\mathbf{w}} \frac{1}{2} \|\mathbf{y} - \boldsymbol{X}\mathbf{w}\|^2 + \frac{\lambda}{2} \mathbf{w}^T \mathbf{w}$$

where $\mathbf{X}$ is the data matrix, each row contains the features of one training example

$\mathbf{y}$ is the vector of ground truth predictions for all training examples

- Closed form solution:

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

where $\boldsymbol{I}$ is the identity matrix

- $\mathbf{w}$ lies in the space spanned by the training examples, i.e., $\mathbf{w} = \sum_i \alpha_i \mathbf{x}_i$
- Instead of solving the optimization problem in the space of $\mathbf{w}$, instead, we can directly solve for $\alpha_i' s$

Kernelizing Linear Regression

- Learned Function:
  - Original: $\hat{f}(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$
  - Kernelized: $\hat{f}(\mathbf{x}) = \sum_i \alpha_i \kappa(\mathbf{x}_i, \mathbf{x})$ (plug in $\mathbf{w} = \sum_i \alpha_i \Phi(\mathbf{x}_i)$ and replace dot product with κ)
- Objective:
  - Original: $\frac{1}{2}\|\mathbf{y} - \boldsymbol{X}\mathbf{w}\|^2 + \frac{\lambda}{2}\mathbf{w}^T\mathbf{w}$ ← solving for **w,** primal problem
  - Kernelized: $\frac{1}{2}\|\mathbf{y} - K\boldsymbol{\alpha}\|^2 + \frac{\lambda}{2}\boldsymbol{\alpha}^T K\boldsymbol{\alpha}$ ← solving for $\boldsymbol{\alpha}$**,** dual problem

K is the kernel (gram) matrix of the training data: $K_{ij} = \kappa(\mathbf{x}_i, \mathbf{x}_j)$

- Closed form solution:

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

General Applications of Kernel Methods

- Many learning algorithms formulate optimization problems and the solutions are some weighted sum of the input training examples
- Explicitly formulating the optimization in the space of the weights, we arrive at the so called “dual formulation” of the optimization problems
- The dual problem can be expressed using dot products between examples
- Apply the kernel trick by replacing dot product with kernel functions.
- This allows the use of high dimensional feature spaces without having to pay the price of computing and working with high dimensional mapped features
- Many types of kernels, wide applicability including some for data that are not naturally in vector form, like graphs, strings, sets etc.