Kernel Methods

CS534

Key concepts:

Feature mapping to address non-linear separability

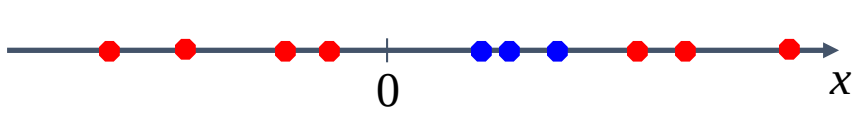
Kernel trick to avoid explicit feature mapping

Definition of Kernel functions

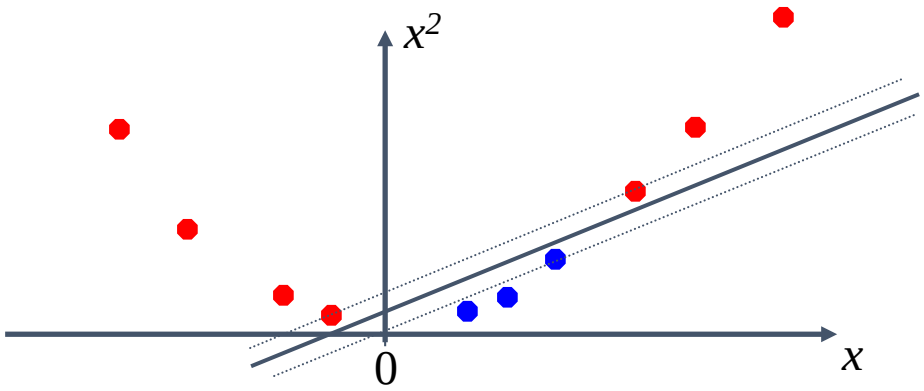
Kernelized perceptron

Kernelized linear regression with L2 regularization

Nonlinearly separable data



x

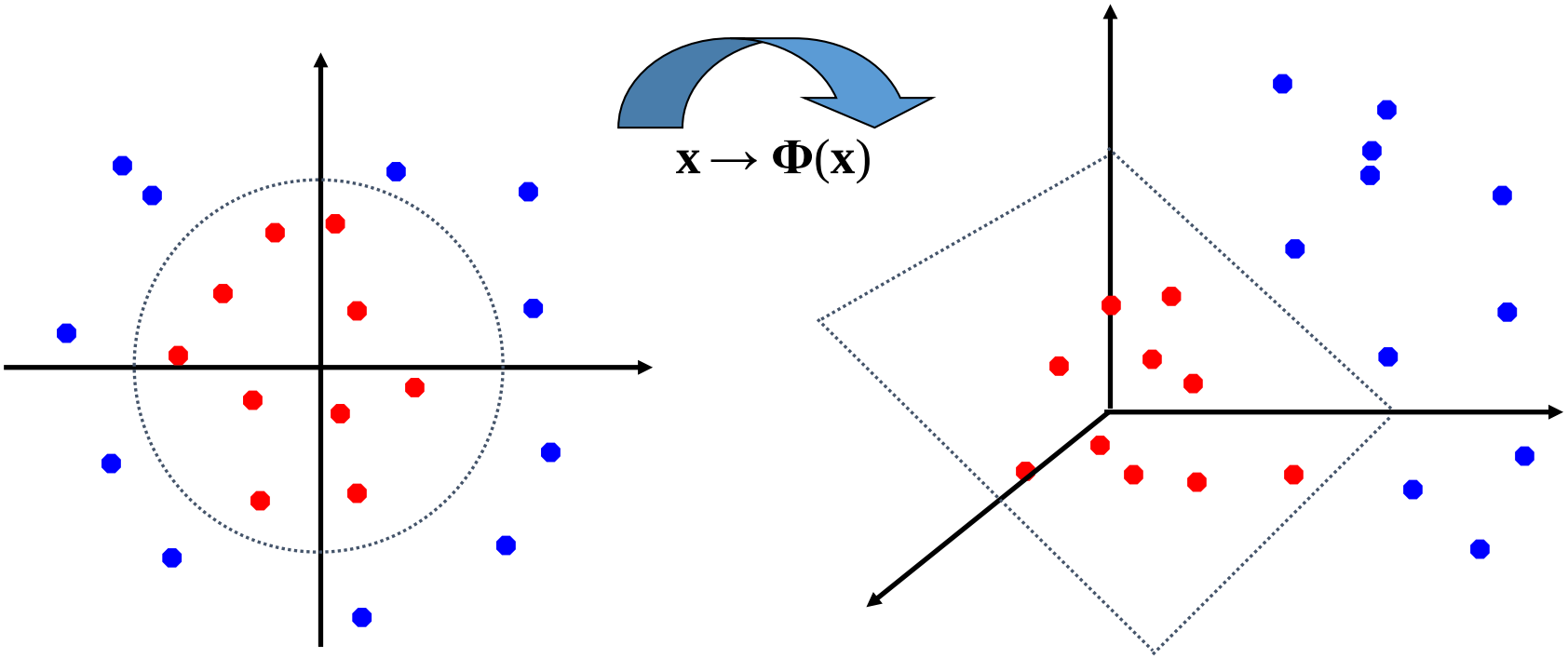


(x, x^2)

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

Non-linear classifier via feature mapping

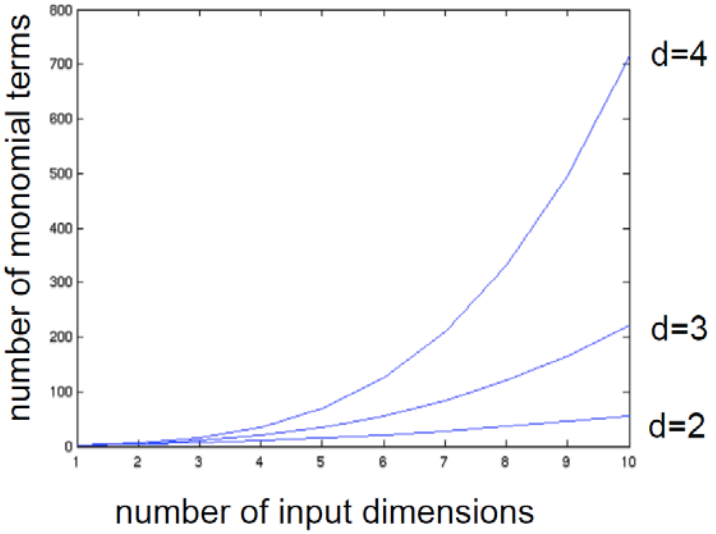
- General idea: For any data set, the ***original input space*** can always be mapped to some higher-dimensional **feature spaces** such that the data is linearly separable:



Example: Quadratic Feature Space

- Assume *m* input dimensions $\mathbf{x} = (x_1, \cdots, x_m)$
- Number of quadratic terms:
- 1+m+m+m(m-1)/2 ≈ $O(m^2)$

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



$$\Phi(\mathbf{x}) = \begin{pmatrix} 1 \\ \sqrt{2}x_1 \\ \sqrt{2}x_2 \\ \vdots \\ \sqrt{2}x_m \\ x_1^2 \\ x_2^2 \\ \vdots \\ x_m^2 \\ \sqrt{2}x_1x_2 \\ \sqrt{2}x_1x_3 \\ \vdots \\ \sqrt{2}x_1x_m \\ \sqrt{2}x_2x_3 \\ \vdots \\ \sqrt{2}x_1x_m \\ \vdots \\ \sqrt{2}x_{m-1}x_m \end{pmatrix}$$

Constant Term

Linear Terms

Pure Quadratic Terms

Quadratic Cross-Terms

Revisiting perceptron

Given current weight w_t , predict for x by:$\hat{y}(x) = sign(w_t \cdot x)$

If mistake at time t, update the weight:

$$w_{t+1} = w_t + y_t x_t$$

If remember all the mistakes made up to t in s_t, we have:

$$w_t = \sum_{i \in s_t} y_i x_i$$

The prediction rule now becomes:

$$\hat{y}(x) = sign(\sum_{i \in S_t} y_i x_i \cdot x)$$

Now data is mapped to higher dimension:

$$\hat{y}(x) = sign\left(\sum_{i \in S_t} y_i \phi(x_i) \cdot \phi(x)\right)$$

Dot product in the Quadratic Feature Space

Explicit mapping takes $O(m^2)$ time

$$\Phi(a)\cdot\Phi(b)=\begin{pmatrix}1\\ \sqrt{2}a_1\\ \sqrt{2}a_2\\ \vdots\\ \sqrt{2}a_m\\ a_1^2\\ a_2^2\\ \vdots\\ a_m^2\\ \sqrt{2}a_1a_2\\ \sqrt{2}a_1a_3\\ \vdots\\ \sqrt{2}a_1a_m\\ \sqrt{2}a_2a_3\\ \vdots\\ \sqrt{2}a_1a_m\\ \vdots\\ \sqrt{2}a_{m-1}a_m\end{pmatrix}\bullet\begin{pmatrix}1\\ \sqrt{2}b_1\\ \sqrt{2}b_2\\ \vdots\\ \sqrt{2}b_m\\ b_1^2\\ b_2^2\\ \vdots\\ b_m^2\\ \sqrt{2}b_1b_2\\ \sqrt{2}b_1b_3\\ \vdots\\ \sqrt{2}b_1b_m\\ \sqrt{2}b_2b_3\\ \vdots\\ \sqrt{2}b_1b_m\\ \vdots\\ \sqrt{2}b_{m-1}b_m\end{pmatrix}\begin{matrix}1\\ +\\ \sum_{i=1}^{m}2a_ib_i\\ +\\ \sum_{i=1}^{m}a_i^2b_i^2\\ +\\ \sum_{i=1}^{m}\sum_{j=i+1}^{m}2a_ia_jb_ib_j\end{matrix}$$

$$\Phi(\mathbf{a})\bullet\Phi(\mathbf{b})=1+2\sum_{i=1}^{m}a_ib_i+\sum_{i=1}^{m}a_i^2b_i^2+\sum_{i=1}^{m}\sum_{j=i+1}^{m}2a_ia_jb_ib_j$$

Consider the following function:

$$(a\cdot b+1)^2$$

$$=(a\cdot b)^2+2(a\cdot b)+1$$

$$=(\sum_{i=1}^{m}a_ib_i)^2+2\sum_{i=1}^{m}a_ib_i+1$$

$$=\sum_{i=1}^{m}\sum_{j=1}^{m}a_ia_jb_ib_j+2\sum_{i=1}^{m}a_ib_i+1$$

$$=\sum_{i=1}^{m}a_i^2b_i^2+2\sum_{i=1}^{m}\sum_{j=i+1}^{m}a_ia_jb_ib_j+2\sum_{i=1}^{m}a_ib_i+1$$

$\kappa(a,b)=(a\cdot b+1)^2$ computes the dot product in quadratic space in $O(m)$ time

Kernel functions: 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: $O(m)$
  - With explicit mapping: $O(m^2)$

Revisiting perceptron

Given current weight w_t , predict for x by:$\hat{y}(x) = sign(w_t \cdot x)$

If mistake at time t, update the weight:

$$w_{t+1} = w_t + y_t x_t$$

If remember all the mistakes made up to t in s_t, we have:

$$w_t = \sum_{i \in s_t} y_i x_i$$

The prediction rule now becomes:

$$\hat{y}(x) = sign(\sum_{i \in S_t} y_i x_i \cdot x)$$

Now data is mapped to higher dimension:

$$\hat{y}(x) = sign\left(\sum_{i \in S_t} y_i \phi(x_i) \cdot \phi(x)\right) = sign\left(\sum_{i \in S_t} y_i K(x_i, x)\right)$$

Kernelizing perceptron

Let $\mathbf{w} \leftarrow (0,0,\ldots,0)$
Repeat if $iter \leq iters$
  for every training example $m = 1,\ldots,n$
    $u_m = \mathbf{w}^T \mathbf{x}_m$
    if $y_m u_m \leq 0$ $\mathbf{w} \leftarrow \mathbf{w} + y_m \mathbf{x}_m$
  $iter = iter + 1$

Remembering all mistakes in s

$S = [\,]$ //list of mistaken examples
Repeat if $iter \leq iters$
  for every training example $m = 1,\ldots,n$
    $u_m = \sum_{i \in S} y_i \kappa(x_i, x_m)$
    if $y_m u_m \leq 0$ add m to S
  $iter = iter + 1$

Kernelizing perceptron

$S = [\,]$ //list of mistaken examples
Repeat if $iter \leq iters$
  for every training example $m = 1, \ldots, n$

$$u_m = \sum_{i \in S} y_i \kappa(x_i, x_m)$$

  if $y_m u_m \leq 0$ add m to S
  $iter = iter + 1$

More efficiently, count for each example the number of times it gets misclassified.

Let $\alpha_i = 0 \;\; \forall i = 1, \ldots, n$
Repeat if $iter \leq iters$
  for every training example $m = 1, \ldots, n$

$$u_m = \sum_{i=1 \; to \; n} \alpha_i y_i \kappa(x_i, x_m)$$

  if $y_m u_m \leq 0$ $\alpha_m \leftarrow \alpha_m + 1$
  $iter = iter + 1$

What functions are kernels?

- 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

$$K = \begin{bmatrix} \kappa(x^1, x^1) & \kappa(x^1, x^2) & \dots & \kappa(x^1, x^m) \\ \kappa(x^2, x^1) & \kappa(x^2, x^2) & \dots & \kappa(x^2, x^m) \\ \dots & \dots & \dots & \dots \\ \kappa(x^m, x^1) & \kappa(x^m, x^2) & \dots & \kappa(x^m, x^m) \end{bmatrix}_{mxm}$$

- Kernel matrices are square and symmetric
- Mercer theorem:

A function K is a kernel function iff for any finite sample $x_1, x_2, \dots, x_m$, its corresponding kernel matrix is positive semi-definite (has non-negative eigenvalues)

Closure property of kernels

If K_1 and K_2 are kernel functions, then the following are all kernel functions:

- $K(x, y) = K_1(x, y) + K_2(x, y)$
  - $\phi = \phi_1 + \phi_2$
- $K(x, y) = aK_1(x, y)$, where $a > 0$
  - $\phi = \sqrt{a}\phi_1$
- $K(x, y) = K_1(x, y)K_2(x, y)$
  - If ϕ_1 has N_1 features and ϕ_2 has N_2 features
  - ϕ will have $N_1 \times N_2$ features: $\phi_{ij} = \phi_{1i} \cdot \phi_{2j}$

Key Choices in Applying Kernel

- Selecting the kernel function
  - In practice, we often try different kernel functions and use (cross-) validation to choose
  - Linear kernel, polynomial kernels (with low degrees) and RBF kernels are popular choices
  - One can also construct a kernel using linear combinations of different kernels and learn the combination parameters (kernel learning)
  - There are many kernel functions defined for non-traditional data, e.g., graph kernel, set kernel, string kernel etc.
- Selecting the kernel parameter
  - Very strong impact on performance
  - Often the optimal range is reasonably large
  - Grid search with (cross-)validation

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., $\mathbf{w} = \sum_i \alpha_i \mathbf{x}_i$

Kernelizing linear regression

- Learned Function:
  - Original: $\hat{f}(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$
  - Kernelized: $\hat{f}(\mathbf{x}) = \sum_i \alpha_i \kappa(\mathbf{x}_i, \mathbf{x})$ (by plugging in $\mathbf{w} = \sum_i \alpha_i \mathbf{x}_i$ and replace dot product with kernel function)
- Objective:
  - Original: $\frac{1}{2}\|\mathbf{y} - \boldsymbol{X}\mathbf{w}\|^2 + \frac{\lambda}{2}\mathbf{w}^T\mathbf{w}$
  - Kernelized: $\frac{1}{2}\|\mathbf{y} - \boldsymbol{\alpha}K\|^2 + \frac{\lambda}{2}\boldsymbol{\alpha}^T K\boldsymbol{\alpha}$

This is referred to as the dual of the original optimization problem

K is the kernel (gram) matrix of the training data

- Closed form solution:

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

General application of Kernel methods

- Many learning tasks are framed as optimization problems and their solutions are some weighted sum of the input training examples
- By 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 is often expressed in terms of the dot product between examples
- The kernel trick can be applied by replacing dot product with kernel functions.
- This allows the use of high dimensional feature spaces without having to pay the price of working with the high dimensional features