Kernel Methods

CMSC 422
MARINE CARPUAT
marine@cs.umd.edu

Slides credit: Piyush Rai
Beyond linear classification

• Problem: linear classifiers
  – Easy to implement and easy to optimize
  – But limited to linear decision boundaries

• What can we do about it?
  – Last week: Neural networks
    • Very expressive but harder to optimize (non-convex objective)
  – Today: Kernels
Kernel Methods

• Goal: keep advantages of linear models, but make them capture non-linear patterns in data!

• How?
  – By mapping data to higher dimensions where it exhibits linear patterns
Classifying non-linearly separable data with a linear classifier: examples

Non-linearly separable data in 1D

Becomes linearly separable in new 2D space defined by the following mapping:

\[ x \rightarrow \{ x, x^2 \} \]
Classifying non-linearly separable data with a linear classifier: examples

Non-linearly separable data in 2D

Becomes linearly separable in the 3D space defined by the following transformation:

\[ \mathbf{x} = \{x_1, x_2\} \rightarrow \mathbf{z} = \{x_1^2, \sqrt{2}x_1x_2, x_2^2\} \]
Defining feature mappings

• Map an original feature vector \( x = \langle x_1, x_2, x_3, \ldots, x_D \rangle \) to an expanded version \( \phi(x) \)

• Example: quadratic feature mapping represents feature combinations
  \[
  \phi(x) = \langle 1, 2x_1, 2x_2, 2x_3, \ldots, 2x_D, \\
  x_1^2, x_1x_2, x_1x_3, \ldots, x_1x_D, \\
  x_2x_1, x_2^2, x_2x_3, \ldots, x_2x_D, \\
  x_3x_1, x_3x_2, x_3^2, \ldots, x_2x_D, \\
  \ldots, \\
  x_Dx_1, x_Dx_2, x_Dx_3, \ldots, x_D^2 \rangle
  \]
Feature Mappings

• Pros: can help turn non-linear classification problem into linear problem

• Cons: “feature explosion” creates issues when training linear classifier in new feature space
  – More computationally expensive to train
  – More training examples needed to avoid overfitting
Kernel Methods

• Goal: keep advantages of linear models, but make them capture non-linear patterns in data!

• How?
  – By mapping data to higher dimensions where it exhibits linear patterns
  – By rewriting linear models so that the mapping never needs to be explicitly computed
The Kernel Trick

• Rewrite learning algorithms so they only depend on **dot products between two examples**

• Replace dot product $\phi(x)^\top \phi(z)$ by **kernel function** $k(x, z)$ which computes the dot product **implicitly**
Example of Kernel function

Consider two examples $\mathbf{x} = \{x_1, x_2\}$ and $\mathbf{z} = \{z_1, z_2\}$

Let’s assume we are given a function $k$ (kernel) that takes as inputs $\mathbf{x}$ and $\mathbf{z}$

$$k(\mathbf{x}, \mathbf{z}) = (\mathbf{x}^\top \mathbf{z})^2$$
$$= (x_1 z_1 + x_2 z_2)^2$$
$$= x_1^2 z_1^2 + x_2^2 z_2^2 + 2x_1 x_2 z_1 z_2$$
$$= (x_1^2, \sqrt{2}x_1 x_2, x_2^2) \top (z_1^2, \sqrt{2}z_1 z_2, z_2^2)$$
$$= \phi(\mathbf{x}) \top \phi(\mathbf{z})$$

The above $k$ implicitly defines a mapping $\phi$ to a higher dimensional space

$$\phi(\mathbf{x}) = \{x_1^2, \sqrt{2}x_1 x_2, x_2^2\}$$
Another example of Kernel Function (see CML 9.1)

\[ \phi(x) = \langle 1, 2x_1, 2x_2, 2x_3, \ldots, 2x_D, \\
x_1^2, x_1x_2, x_1x_3, \ldots, x_1x_D, \\
x_2x_1, x_2^2, x_2x_3, \ldots, x_2x_D, \\
x_3x_1, x_3x_2, x_3^2, \ldots, x_2x_D, \\
\ldots, \\
x Dx_1, x Dx_2, x Dx_3, \ldots, x_D^2 \rangle \]

What is the function \( k(x, z) \) that can implicitly compute the dot product \( \phi(x) \cdot \phi(z) \)?

\[ \phi(x) \cdot \phi(z) = 1 + x_1z_1 + x_2z_2 + \cdots + x_Dz_D + x_1^2z_1^2 + \cdots + x_1x_Dz_1z_D + \\
\cdots + x_Dx_1z_Dz_1 + x_Dx_2z_Dz_2 + \cdots + x_D^2z_D^2 \]
\[ = 1 + 2 \sum_d x_dz_d + \sum_d \sum_e x_dx_ez_dz_e \]
\[ = 1 + 2x \cdot z + (x \cdot z)^2 \]
\[ = (1 + x \cdot z)^2 \]
Kernels: Formally defined

Recall: Each kernel $k$ has an associated feature mapping $\phi$

$\phi$ takes input $x \in \mathcal{X}$ (input space) and maps it to $\mathcal{F}$ ("feature space")

Kernel $k(x, z)$ takes two inputs and gives their similarity in $\mathcal{F}$ space

$$
\phi : \mathcal{X} \rightarrow \mathcal{F} \quad \quad k : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}, \quad k(x, z) = \phi(x)^{\top} \phi(z)
$$

$\mathcal{F}$ needs to be a vector space with a dot product defined on it

Also called a Hilbert Space
Kernels: Mercer’s condition

- Can *any* function be used as a kernel function?
  - No! it must satisfy Mercer’s condition.

For $k$ to be a kernel function

- There must exist a Hilbert Space $\mathcal{F}$ for which $k$ defines a dot product
- The above is true if $K$ is a **positive definite function**

$$\int dx \int dz f(x)k(x, z)f(z) > 0$$

For all square integrable functions $f$
Kernels: Constructing combinations of kernels

Let \( k_1, k_2 \) be two kernel functions then the following are as well

- \( k(x, z) = k_1(x, z) + k_2(x, z) \): direct sum
- \( k(x, z) = \alpha k_1(x, z) \): scalar product
- \( k(x, z) = k_1(x, z)k_2(x, z) \): direct product
Commonly Used Kernel Functions

Linear (trivial) Kernel:
\[ k(x, z) = x^\top z \] (mapping function \( \phi \) is identity - no mapping)

Quadratic Kernel:
\[ k(x, z) = (x^\top z)^2 \quad \text{or} \quad (1 + x^\top z)^2 \]

Polynomial Kernel (of degree \( d \)):
\[ k(x, z) = (x^\top z)^d \quad \text{or} \quad (1 + x^\top z)^d \]

Radial Basis Function (RBF) Kernel:
\[ k(x, z) = \exp[-\gamma \|x - z\|^2] \]
The Kernel Trick

• Rewrite learning algorithms so they only depend on **dot products between two examples**

• Replace dot product $\phi(x)^T \phi(z)$ by **kernel function** $k(x, z)$ which computes the dot product **implicitly**
“Kernelizing” the perceptron

- Naïve approach: let’s explicitly train a perceptron in the new feature space

Algorithm 28 $\textbf{PerceptronTrain}(D, \text{MaxIter})$

1. $w \leftarrow 0$, $b \leftarrow 0$  // initialize weights and bias
2. for $\text{iter} = 1 \ldots \text{MaxIter}$ do
3.   for all $(x, y) \in D$ do
4.     $a \leftarrow w \cdot \phi(x) + b$  // compute activation for this example
5.     if $ya \leq 0$ then
6.       $w \leftarrow w + y \phi(x)$  // update weights
7.       $b \leftarrow b + y$  // update bias
8.     end if
9.   end for
10. end for
11. return $w, b$

Can we apply the Kernel trick?  Not yet, we need to rewrite the algorithm using dot products between examples
“Kernelizing” the perceptron

- Perceptron Representer Theorem

“During a run of the perceptron algorithm, the weight vector w can always be represented as a linear combination of the expanded training data”

Proof by induction
(on board + see CIML 9.2)
"Kernelizing" the perceptron

• We can use the perceptron representer theorem to compute activations as a **dot product** between examples

\[ w \cdot \phi(x) + b = \left( \sum_n \alpha_n \phi(x_n) \right) \cdot \phi(x) + b \quad \text{definition of } w \]  

(9.6)

\[ = \sum_n \alpha_n \left[ \phi(x_n) \cdot \phi(x) \right] + b \quad \text{dot products are linear} \]  

(9.7)
“Kernelizing” the perceptron

Algorithm 29 **KernelizedPerceptronTrain**($D$, $MaxIter$)

1: $\alpha \leftarrow 0$, $b \leftarrow 0$ // initialize coefficients and bias
2: for iter = 1 ... $MaxIter$ do
3:   for all $(x_n, y_n) \in D$ do
4:     $a \leftarrow \sum_m \alpha_m \phi(x_m) \cdot \phi(x_n) + b$ // compute activation for this example
5:     if $y_n a \leq 0$ then
6:       $\alpha_n \leftarrow \alpha_n + y_n$ // update coefficients
7:       $b \leftarrow b + y$ // update bias
8:   end if
9: end for
10: end for
11: return $\alpha$, $b$

- Same training algorithm, but doesn’t explicitly refers to weights $w$ anymore only depends on dot products between examples
- We can apply the kernel trick!
Kernel Methods

• Goal: keep advantages of linear models, but make them capture non-linear patterns in data!

• How?
  – By mapping data to higher dimensions where it exhibits linear patterns
  – By rewriting linear models so that the mapping never needs to be explicitly computed
Discussion

• Other algorithms can be kernelized:
  – See CIML for K-means
  – We’ll talk about Support Vector Machines next

• Do Kernels address all the downsides of “feature explosion”?
  – Helps reduce computation cost during training
  – But overfitting remains an issue
What you should know

• Kernel functions
  – What they are, why they are useful, how they relate to feature combination

• Kernelized perceptron
  – You should be able to derive it and implement it