# Machine Learning (CSE 446): Kernel Methods

Sham M Kakade



University of Washington
cse446-staff@cs.washington.edu

# Can We Have Nonlinearity *and* Convexity?

| | expressiveness | convexity |
|---|---|---|
| Linear classifiers | ☹ | ☺ |
| Neural networks | ☺ | ☹ |

# Can We Have Nonlinearity *and* Convexity?

| | expressiveness | convexity |
|---|---|---|
| Linear classifiers | ☹ | ☺ |
| Neural networks | ☺ | ☹ |

**Kernel** methods: a family of approaches that give us nonlinear decision boundaries without giving up convexity.

# Notation

Let $\mathbf{x} = \langle x_1, x_2, \ldots, x_d \rangle$.

# Conjunctive/Product Features

See slides 23–32 in the 10/13 "practical issues" lecture.

Consider two binary features, $\phi_j$ and $\phi_{j'}$. A new *conjunction* feature can be defined by:

$$\phi_{j \wedge j'}(x) = \phi_j(x) \wedge \phi_{j'}(x) \quad \text{equivalently} \quad x_{d+1} = x_j \wedge x_{j'}$$

# Conjunctive/Product Features

See slides 23–32 in the 10/13 "practical issues" lecture.

Consider two binary features, $\phi_j$ and $\phi_{j'}$. A new *conjunction* feature can be defined by:

$$\phi_{j \wedge j'}(x) = \phi_j(x) \wedge \phi_{j'}(x) \quad \text{equivalently} \quad x_{d+1} = x_j \wedge x_{j'}$$

Generalization: take the *product* of two features.

# Conjunctive/Product Features

See slides 23–32 in the 10/13 “practical issues” lecture.

Consider two binary features, $\phi_j$ and $\phi_{j'}$. A new *conjunction* feature can be defined by:

$$\phi_{j \wedge j'}(x) = \phi_j(x) \wedge \phi_{j'}(x) \quad \text{equivalently} \quad x_{d+1} = x_j \wedge x_{j'}$$

Generalization: take the *product* of two features.

Bigger generalization: take all the products!

$$\begin{aligned}
\boldsymbol{\phi}(\mathbf{x}) &= \text{vector}(\langle 1; \mathbf{x}\rangle \langle 1; \mathbf{x}\rangle^\top) \\
&= \langle \begin{array}{ccccc}
1, & x_1, & x_2, & \ldots, & x_d, \\
x_1, & x_1^2, & x_1 \cdot x_2, & \ldots, & x_1 \cdot x_d, \\
x_2, & x_2 \cdot x_1, & x_2^2, & \ldots, & x_2 \cdot x_d, \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
x_{d-1}, & x_{d-1} \cdot x_1, & x_{d-1} \cdot x_2, & \ldots, & x_{d-1} \cdot x_d, \\
x_d, & x_d \cdot x_1, & x_d \cdot x_2, & \ldots, & x_d^2
\end{array} \rangle
\end{aligned}$$

# The Kernel Trick

Some learning algorithms, like the perceptron, can be rewritten so that the only thing you do with feature vectors is take *inner products between them*.

# The Kernel Trick

Some learning algorithms, like the perceptron, can be rewritten so that the only thing you do with feature vectors is take *inner products between them*.

Note that: $\phi(\mathbf{x}) \cdot \phi(\mathbf{v})$

$$
\begin{array}{cccccccccc}
= & 1 & + & x_1 v_1 & + & x_2 v_2 & + \cdots + & x_d v_d \\
+ & x_1 v_1 & + & x_1^2 v_1^2 & + & x_1 x_2 v_1 v_2 & + \cdots + & x_1 x_d v_1 v_d \\
+ & x_2 v_2 & + & x_2 x_1 v_2 v_1 & + & x_2^2 v_2^2 & + \cdots + & x_2 x_d v_2 v_d \\
+ & \vdots & \vdots & \vdots & \vdots & \vdots & & \\
+ & x_d v_d & + & x_d x_1 v_d v_1 & + & x_d x_2 v_d v_2 & + \cdots + & x_d^2 v_d^2
\end{array}
$$

$$= 1 + 2 \cdot \sum_{j=1}^{d} x_j v_j + \sum_{j=1}^{d} \sum_{k=1}^{d} x_j x_k v_j v_k$$

$$= 1 + 2 \cdot \mathbf{x} \cdot \mathbf{v} + (\mathbf{x} \cdot \mathbf{v})^2$$

$$= (1 + \mathbf{x} \cdot \mathbf{v})^2$$

# Kernels

A **kernel** function (implicitly) computes:

$$K(\mathbf{x}, \mathbf{v}) = \boldsymbol{\phi}(\mathbf{x}) \cdot \boldsymbol{\phi}(\mathbf{v})$$

for some $\phi$. Typically it is *cheap* to compute $K(\cdot, \cdot)$, and we never explicitly represent $\boldsymbol{\phi}(\mathbf{v})$ for any vector $\mathbf{v}$.

# Kernels

A **kernel** function (implicitly) computes:

$$K(\mathbf{x}, \mathbf{v}) = \boldsymbol{\phi}(\mathbf{x}) \cdot \boldsymbol{\phi}(\mathbf{v})$$

for some $\boldsymbol{\phi}$. Typically it is *cheap* to compute $K(\cdot, \cdot)$, and we never explicitly represent $\boldsymbol{\phi}(\mathbf{v})$ for any vector $\mathbf{v}$.

Some kernels:

$$\begin{aligned}
\text{quadratic} \quad & K^{\text{quad}}(\mathbf{x}, \mathbf{v}) = (1 + \mathbf{x} \cdot \mathbf{v})^2 \\
\text{cubic} \quad & K^{\text{cubic}}(\mathbf{x}, \mathbf{v}) = (1 + \mathbf{x} \cdot \mathbf{v})^3 \\
\text{polynomial} \quad & K_p^{\text{poly}}(\mathbf{x}, \mathbf{v}) = (1 + \mathbf{x} \cdot \mathbf{v})^p \\
\text{radial basis function} \quad & K_\gamma^{\text{rbf}}(\mathbf{x}, \mathbf{v}) = \exp\left(-\gamma \left\|\mathbf{x} - \mathbf{v}\right\|_2^2\right) \\
\text{hyperbolic tangent} \quad & \tilde{K}^{\text{tanh}}(\mathbf{x}, \mathbf{v}) = \tanh(1 + \mathbf{x} \cdot \mathbf{v}) \quad \text{(not a kernel)} \\
\text{all conjunctions} \quad & K^{\text{all conj}}(\mathbf{x}, \mathbf{v}) = \prod_{j=1}^{d} (1 + x_j v_j) \quad \text{(for binary features)}
\end{aligned}$$

# Perceptron Learning Algorithm

**Data**: $D = \langle(\mathbf{x}_n, y_n)\rangle_{n=1}^{N}$, number of epochs $E$
**Result**: weights $\mathbf{w}$ and bias $b$

```
initialize: w = 0 and b = 0;
for e ∈ {1, ..., E} do
    for n ∈ {1, ..., N}, in random order do
        # predict
        ŷ = sign(w · x_n + b);
        if ŷ ≠ y_n then
            # update
            w ← w + y_n · x_n;
            b ← b + y_n;
        end
    end
end
return w, b
```

**Algorithm 1:** PerceptronTrain

# Perceptron Representer Theorem

At every stage of learning, there exist $\langle \alpha_1, \alpha_2, \ldots, \alpha_N \rangle$ such that

$$\mathbf{w} = \sum_{n=1}^{N} \alpha_n \cdot \mathbf{x}_n = \boldsymbol{\alpha}^\top \mathbf{X}$$

In other words, $\mathbf{w}$ is always in the span of the training data.

# Perceptron Learning Algorithm (with $\phi$)

**Data**: $D = \langle(\mathbf{x}_n, y_n)\rangle_{n=1}^{N}$, number of epochs $E$
**Result**: weights $\mathbf{w}$ and bias $b$
initialize: $\mathbf{w} = \mathbf{0}$ and $b = 0$;
**for** $e \in \{1, \ldots, E\}$ **do**
    **for** $n \in \{1, \ldots, N\}$, *in random order* **do**
        # predict
        $\hat{y} = \text{sign}\left(\mathbf{w} \cdot \boldsymbol{\phi}(\mathbf{x}_n) + b\right)$;
        **if** $\hat{y} \neq y_n$ **then**
            # update
            $\mathbf{w} \leftarrow \mathbf{w} + y_n \cdot \boldsymbol{\phi}(\mathbf{x}_n)$;
            $b \leftarrow b + y_n$;
        **end**
    **end**
**end**
return $\mathbf{w}, b$

**Algorithm 2:** PerceptronTrain with $\phi$ (explicit)

# Prediction

$$\begin{aligned}
\hat{y} &= \text{sign}\left(\mathbf{w} \cdot \boldsymbol{\phi}(\mathbf{x}_n) + b\right) \\
&= \text{sign}\left(\sum_{i=1}^{N} \alpha_i \cdot \boldsymbol{\phi}(\mathbf{x}_i) \cdot \boldsymbol{\phi}(\mathbf{x}_n) + b\right) \\
&= \text{sign}\left(\sum_{i=1}^{N} \alpha_i \cdot K(\mathbf{x}_i, \mathbf{x}_n) + b\right)
\end{aligned}$$

# The Update

$$\mathbf{w}^{\text{(new)}} \leftarrow \mathbf{w}^{\text{(old)}} + y_n \cdot \boldsymbol{\phi}(\mathbf{x}_n)$$

$$\sum_{i=1}^{N} \alpha_i^{\text{(new)}} \cdot \boldsymbol{\phi}(\mathbf{x}_i) \leftarrow \sum_{i=1}^{N} \alpha_i^{\text{(old)}} \cdot \boldsymbol{\phi}(\mathbf{x}_i) + y_n \cdot \boldsymbol{\phi}(\mathbf{x}_n)$$

$$\sum_{i \neq n} \alpha_i^{\text{(new)}} \cdot \boldsymbol{\phi}(\mathbf{x}_i) + \alpha_n^{\text{(new)}} \cdot \boldsymbol{\phi}(\mathbf{x}_n) \leftarrow \sum_{i \neq n} \alpha_i^{\text{(old)}} \cdot \boldsymbol{\phi}(\mathbf{x}_i) + (\alpha_n^{\text{(old)}} + y_n) \cdot \boldsymbol{\phi}(\mathbf{x}_n)$$

$$\alpha_n^{\text{(new)}} \cdot \boldsymbol{\phi}(\mathbf{x}_n) \leftarrow (\alpha_n^{\text{(old)}} + y_n) \cdot \boldsymbol{\phi}(\mathbf{x}_n)$$

$$\alpha_n^{\text{(new)}} \leftarrow \alpha_n^{\text{(old)}} + y_n$$

# $\phi(\mathbf{x}_n)$ is Never Explicitly Computed!

$$\text{predict:} \quad \hat{y} = \text{sign}\left(\sum_{i=1}^{N} \alpha_i \cdot K(\mathbf{x}_i, \mathbf{x}_n) + b\right)$$

$$\text{update:} \quad \alpha_n^{(\text{new})} \leftarrow \alpha_n^{(\text{old})} + y_n$$

We only calculate inner products of such vectors.

# Kernelized Perceptron Learning Algorithm

**Data**: $D = \langle(\mathbf{x}_n, y_n)\rangle_{n=1}^{N}$, number of epochs $E$
**Result**: weights $\boldsymbol{\alpha}$ and bias $b$
initialize: $\boldsymbol{\alpha} = \mathbf{0}$ and $b = 0$;
**for** $e \in \{1, \ldots, E\}$ **do**
    **for** $n \in \{1, \ldots, N\}$, *in random order* **do**
        # predict
        $\hat{y} = \text{sign}\left(\sum_{i=1}^{N} \alpha_i \cdot K(\mathbf{x}_i, \mathbf{x}_n) + b\right)$;
        **if** $\hat{y} \neq y_n$ **then**
            # update
            $\alpha_n \leftarrow \alpha_n + y_n$;
            $b \leftarrow b + y_n$;
        **end**
    **end**
**end**
return $\boldsymbol{\alpha}, b$

**Algorithm 3:** KernelizedPerceptronTrain